Advanced Topics in Databases

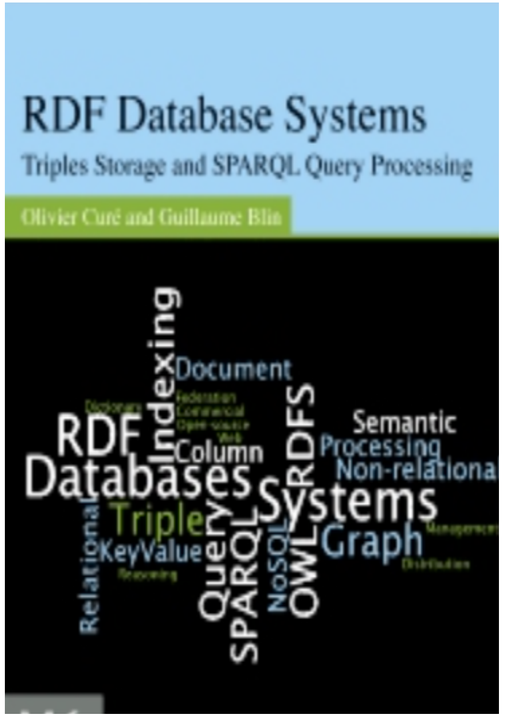
Summer Term 2017

Applied Compression Techniques in RDF Stores

Marcus Pinnecke

This lecture based in parts of the text book

RDF Database Systems, 1st Edition — Triples Storage and SPARQL Query Processing by Curé & Blin, Print Book ISBN :9780127999579



Outline

BigData & Linked Data

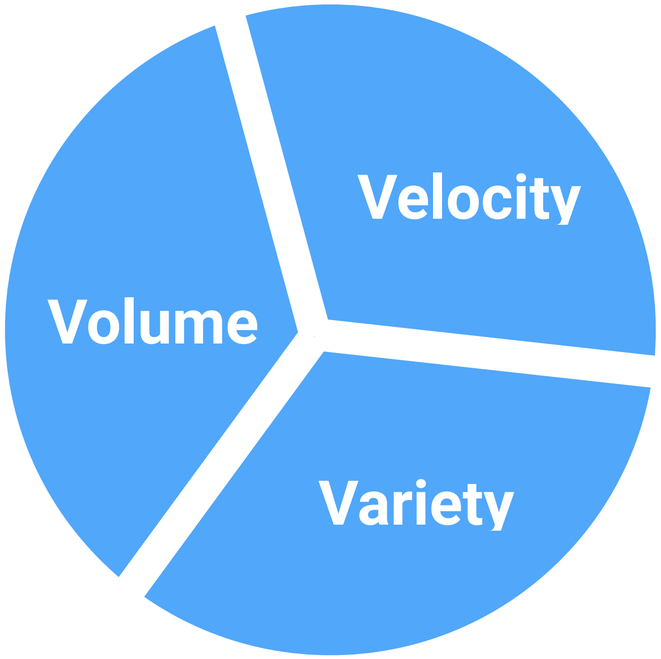
Part I

BigData(I)

- **Todays** 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,…
  - …

↗ [5]

BigData (2)



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
  - …

↗ [5]

Graph Data (I)

- Graph is the most generic data structure
- Data is modeled in terms of nodes and edges.
- Several models exists
  - Simple graphs
  - Multi graphs
  - Weighted graphs,
  - Attributed graphs
  - …

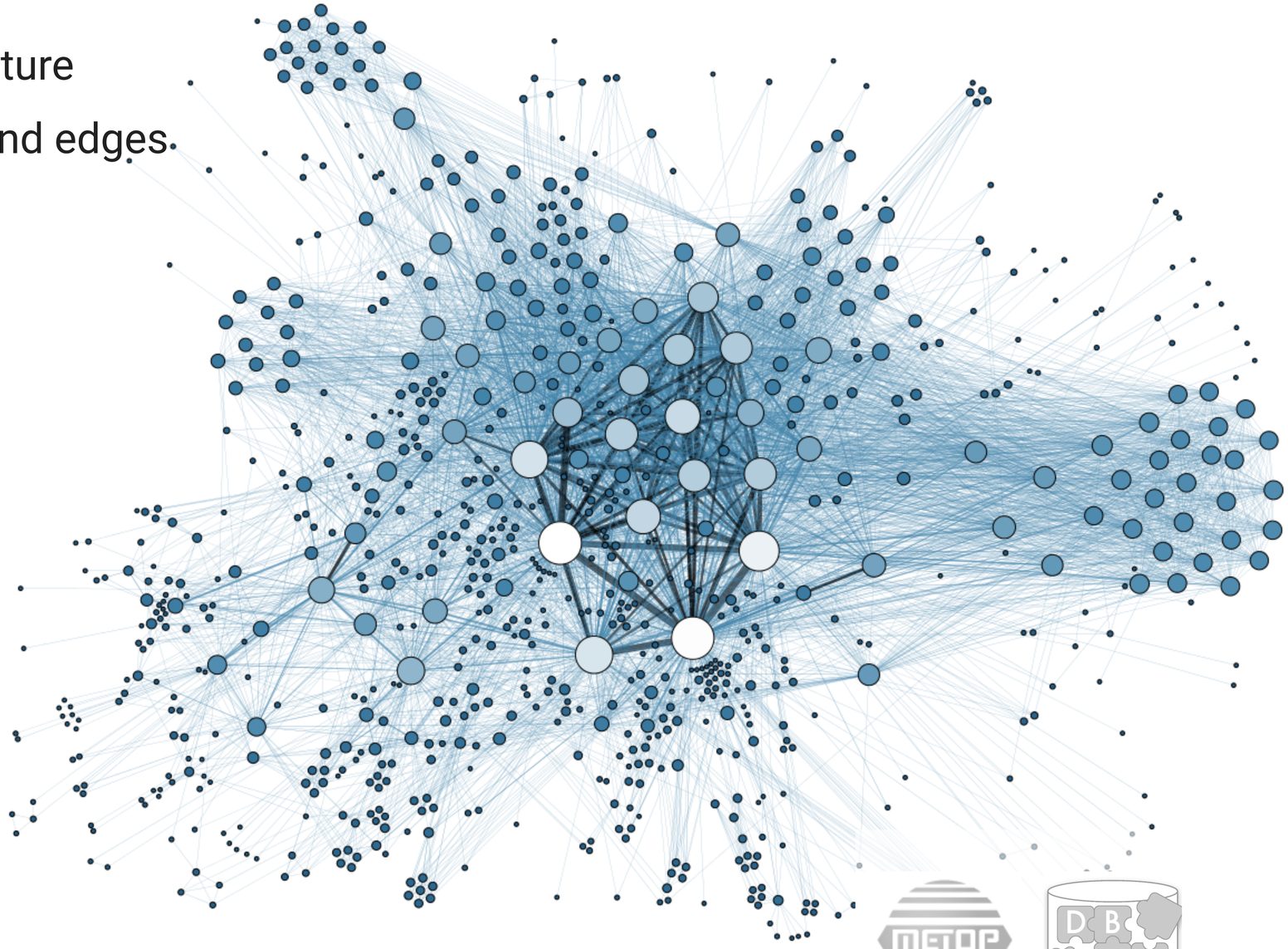


Image taken from: https://upload.wikimedia.org/wikipedia/commons/9/9b/Social_Network_Analysis_Visualization.png

Graph Data (II) What is 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*

…

↗ [5]

Semantic Web

Part II

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**)

Sir Tim Berners-Lee

[1] think about IRIs as unicode variant of generalized URLs, e.g., http://www.example.com/rdf/vocabulary#size

↗ [6]

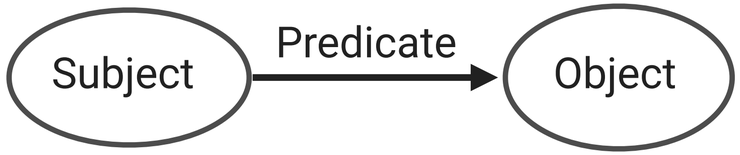
RDF

RDF Concept and Abstract Syntax (I)

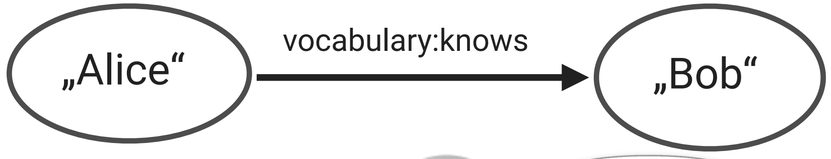
Resource Description Framework (RDF)

- 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 (***s**ubject, **p**redicate, **o**bject)=(**s**,**p**,**o**)* forms an RDF **statement**
  - A binary relationship **p** between resources **s** and **o**



↗ [7]

RDF Concept and Abstract Syntax (II)

RDF Example

Subject	Predicate	Object
course:AdvDB	xmpl:fullName	„Advanced Topics in Databases“
course:AdvDB	xmpl:occurs	terms:summer
course:AdvDB	xmpl:lecturer	ovgu:Bronesk
ovgu:Bronesk	xmpl:name	„David Broneske“
ovgu:Bronesk	rdf:type	xmpl:human

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

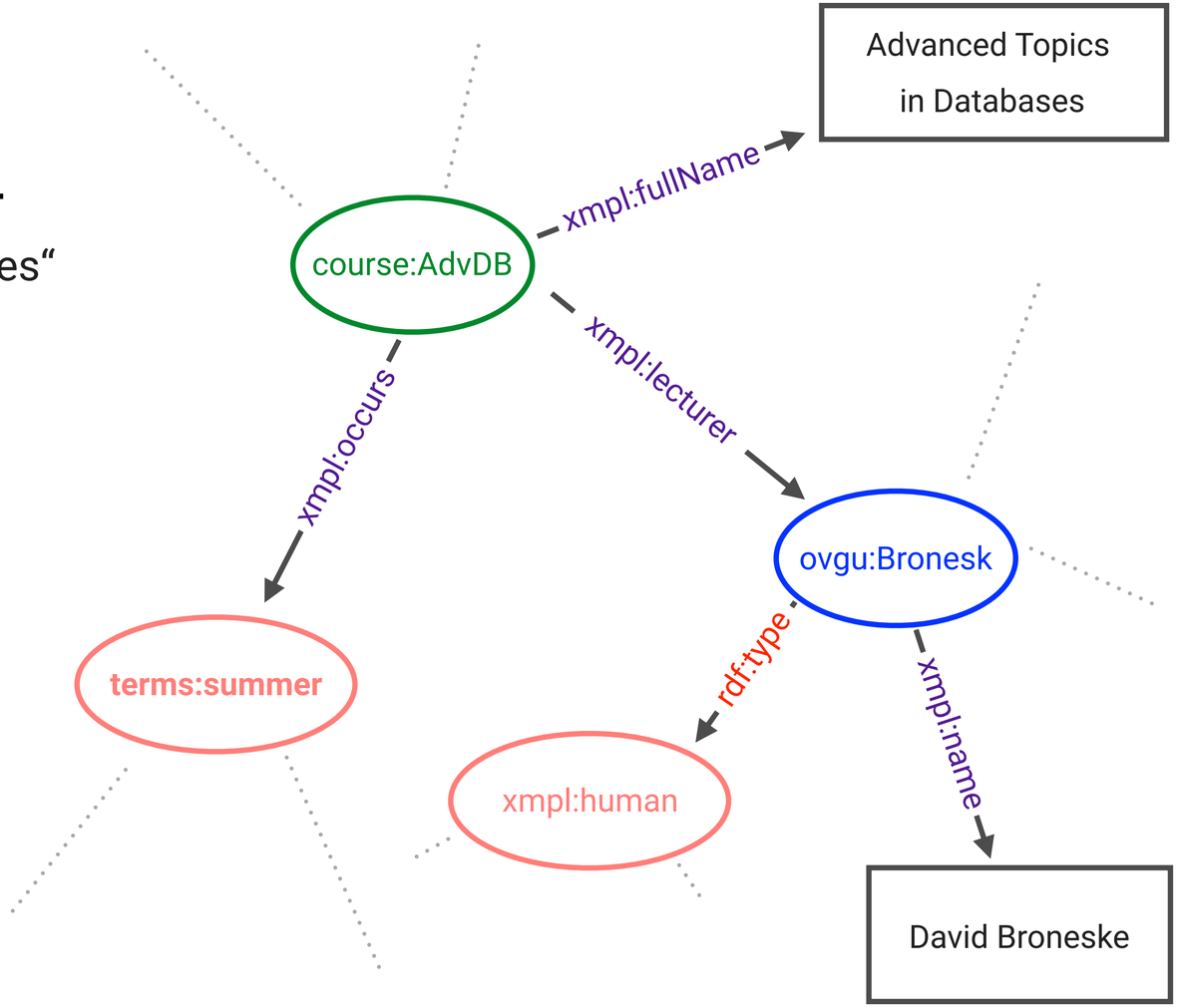


Figure: Example RDF graph (visualization)

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

↗ [7]

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**

↗ [7]

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#)

↗ [7]

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

↗ [2,9]

Semantic Web Architecture Stack

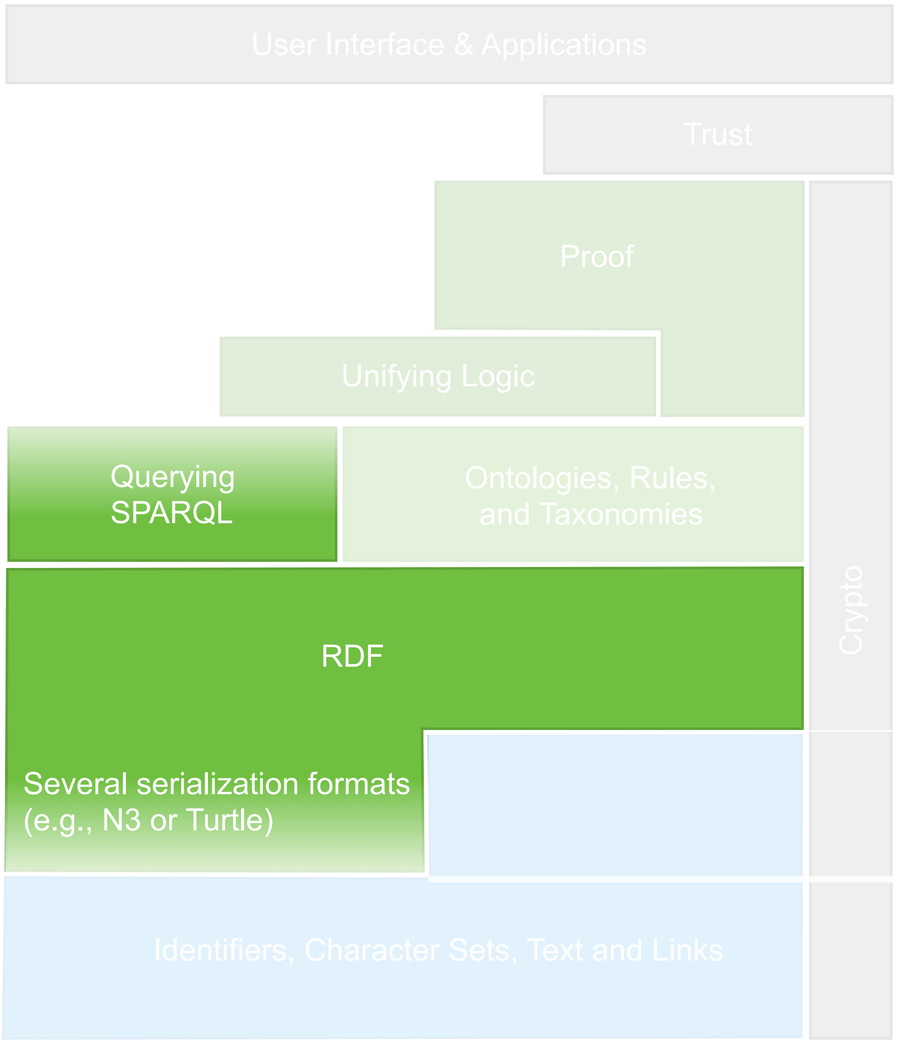
topics for database community

Indexing

Query Processing

...

Storage



Unrealized
Semantic Web Technologies
Web Technologies

Semantic Web Cake (simplified)

↗ [2,9]

RDF Dictionaries

Storage of Triple Values

- or -

How to store and access large collections of strings

Part III

Motivation

Subject	Predicate	Object
	...	
ovgu:Bronesk	rdf:type	xmpl:human
	...	

172B

Figure: Example RDF graph

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)

Dictionary Encoding & Compression (I)

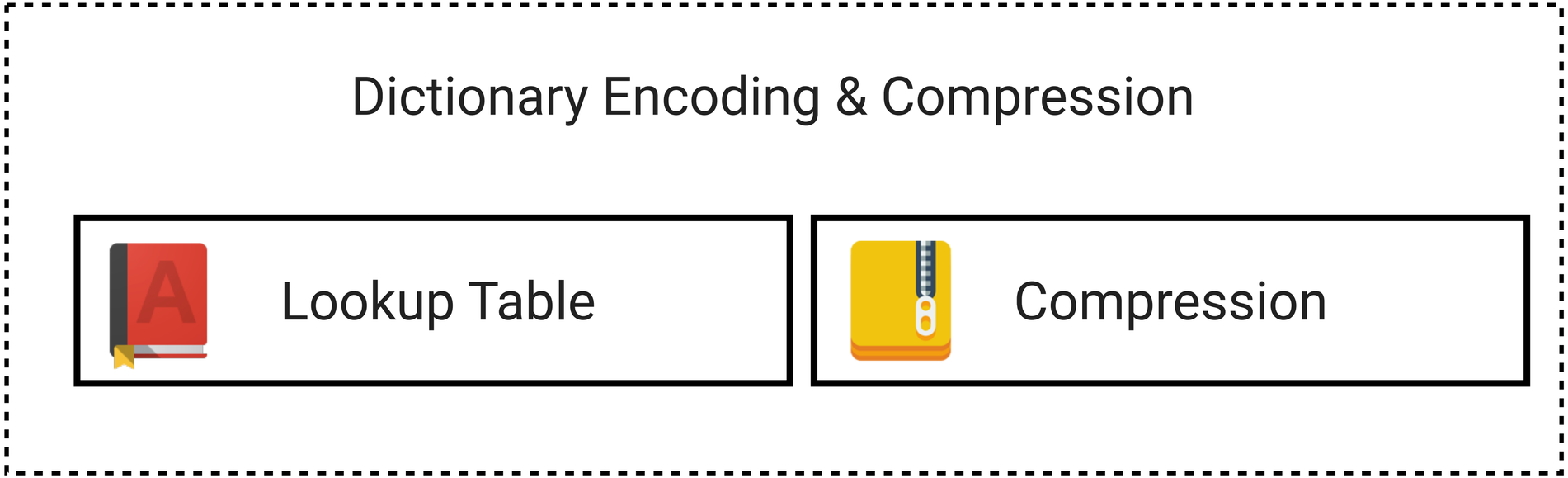


Image taken from https://cdn1.iconfinder.com/data/icons/google_jfk_icons_by_carlosjj/128/dictionary.png and http://iconshow.me/file-zip-alt-icon

↗ [3]

Dictionary Encoding & Compression (II)

Dictionary Encoding & Compression

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
 }
```


↗ [3]

Dictionary Encoding & Compression (III)

Dictionary Encoding & Compression

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
 }
```


↗ [3]

Dictionary Encoding & Compression (IV)

Dictionary Encoding & Compression

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
 }
```


↗ [3]

Dictionary Encoding & Compression (V)

Dictionary Encoding & Compression

Two tasks…

- **3. Compression**: Compress encoded collection `collection'`
- **4. Compression**: …

↗ [3]

Lockup Table

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

`locate(`http://www.example.com/rdf/course#**AdvDB**`) = 1`

`extract(1) =` http://www.example.com/rdf/course#**AdvDB**

↗ [3]

Query Processing with Dictionary Encoded Values (I)

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

```
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

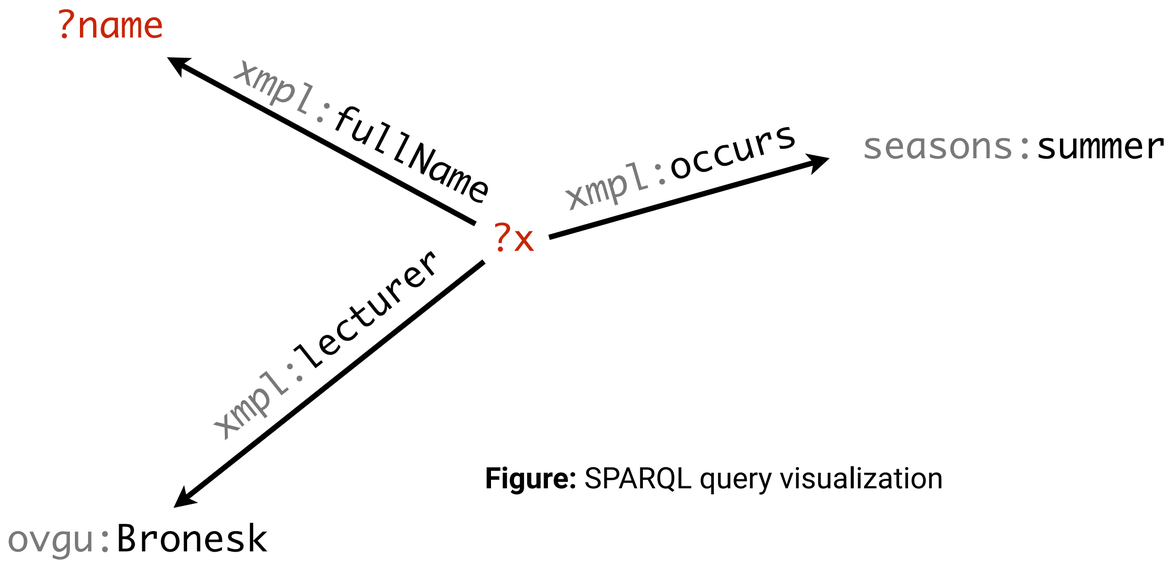


Figure: SPARQL query visualization

Query Processing with Dictionary Encoded Values (II)

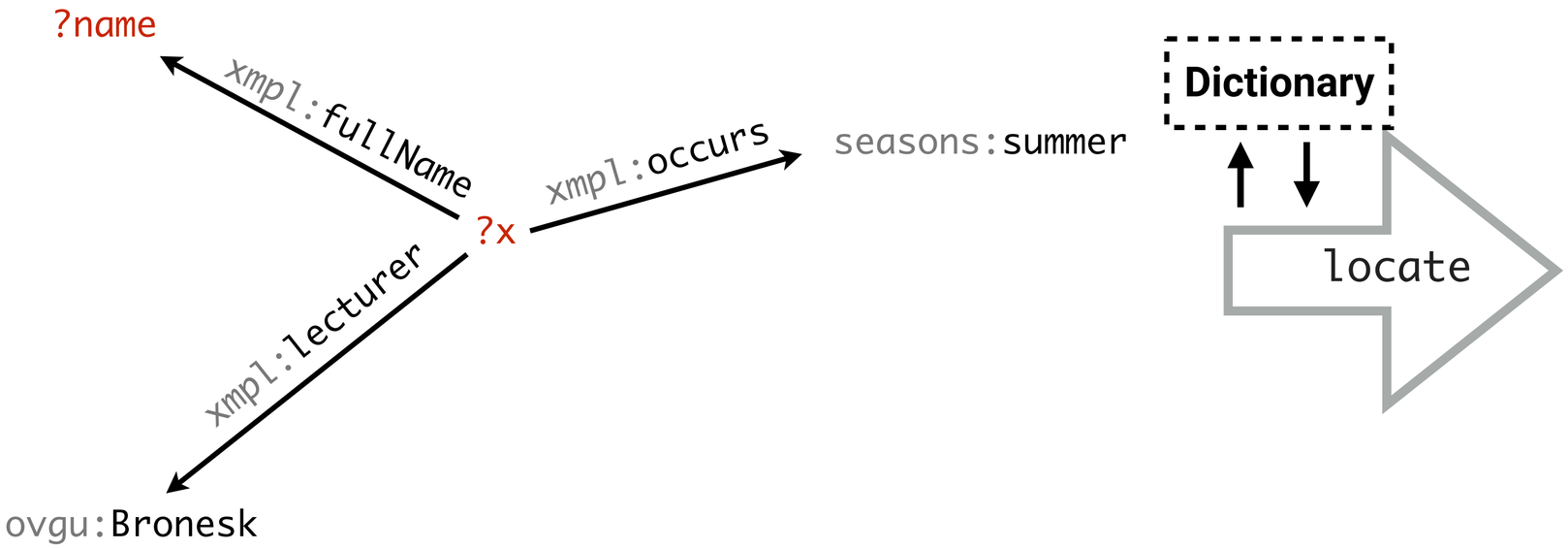


Figure: SPARQL query visualization

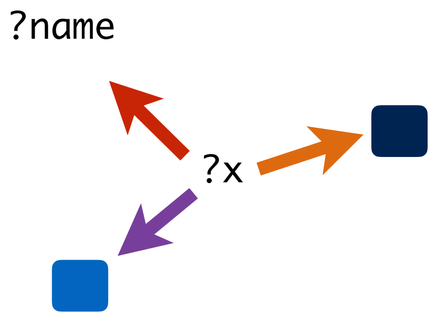


Figure: SPARQL query (encoded)

Query Processing with Dictionary Encoded Values (III)

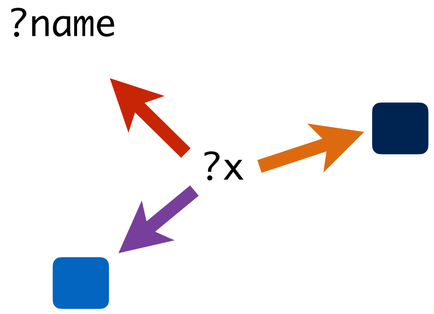


Figure: SPARQL query (encoded)

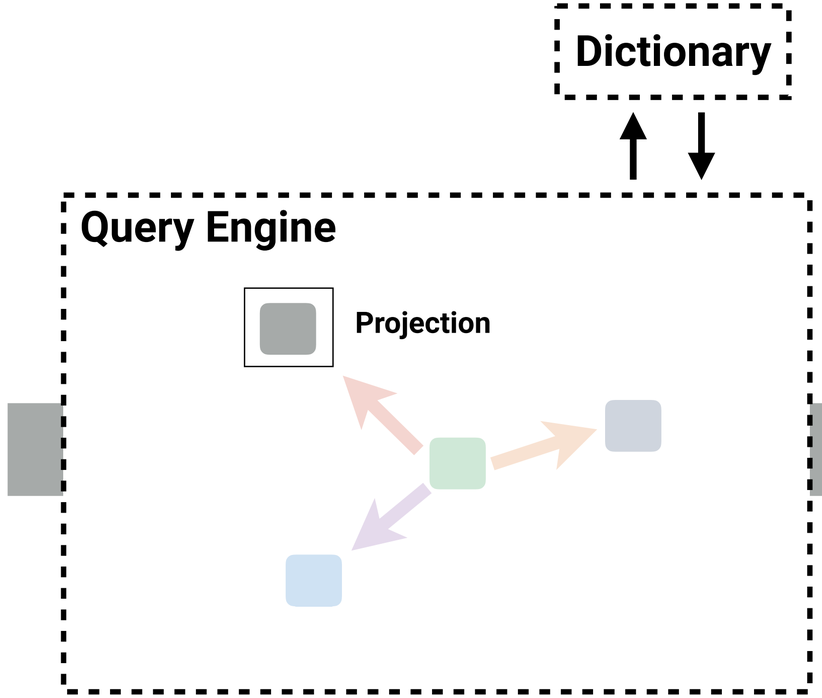


Figure: Query Processing

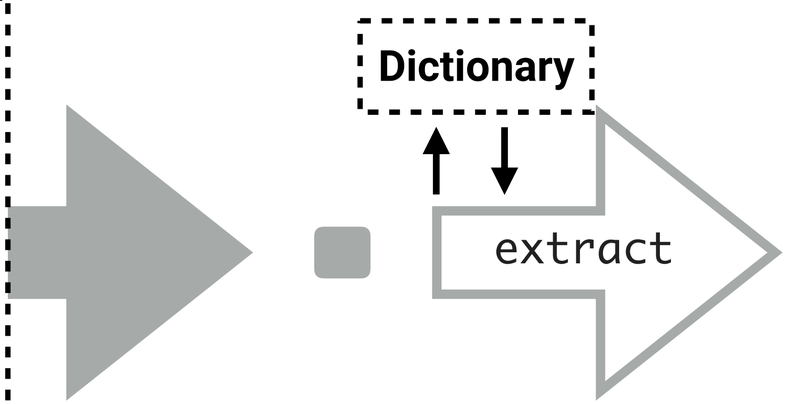


Figure: Query Result (encoded)

Advanced Topics
in Databases

Figure: Query Result (decoded)

Huffman Coding

Huffman Coding (I)

- **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

↗ [3]

Huffman Coding (II)

- **Huffman Coding (General)**
  - **Input** vector of probabilities $\mathbf{p}=(p_1,p_2,\ldots,p_n)$, values $\mathbf{v}=(v_1,v_2,\ldots,v_n)$ and a code alphabet A
  - **Output** optimal code, i.e., minimum length code

$$Huff(\mathbf{p},A)=(code(v_1),code(v_2),\ldots,code(v_n))$$

  - Used to reduce the number of bits needed for a message
  - **Idea:** The higher the probability the less bits needed (and vice versa)

↗ [3]

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* $w = 1$

$$\mathbf{p} = (\quad 0.45, \quad 0.2, \quad 0.2, \quad 0.15 \quad)$$

↗ [3]

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* $w = 1$

$$\mathbf{p} = (\quad 0.45, \quad 0.2, \quad 0.2, \quad 0.15 \quad)$$

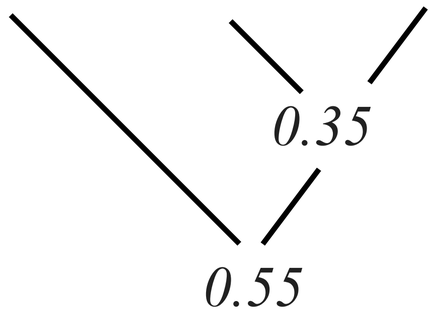
$$0.35$$

↗ [3]

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* $w = 1$

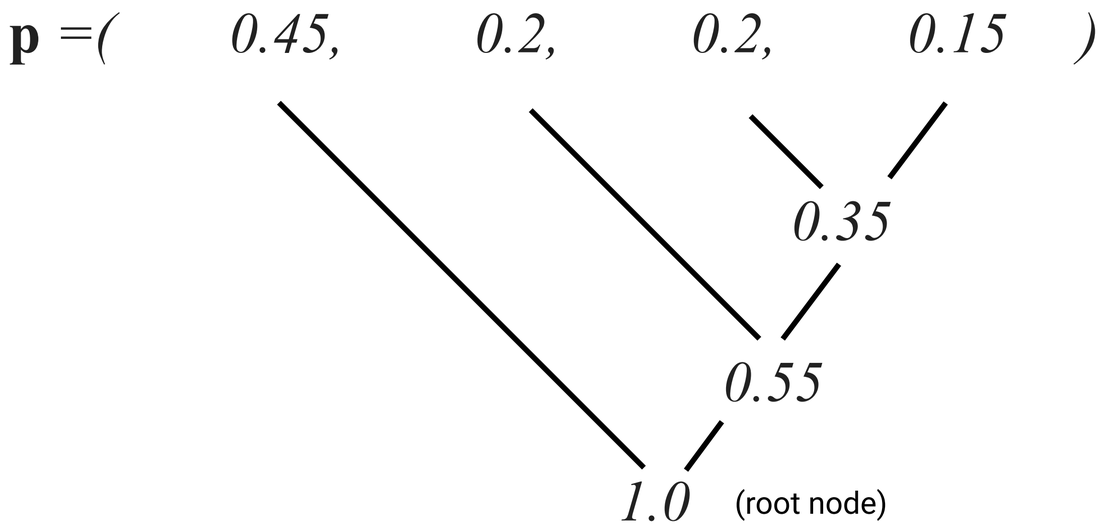
$$\mathbf{p} = (\quad 0.45, \quad 0.2, \quad 0.2, \quad 0.15 \quad)$$



↗ [3]

Huffman Coding (III)

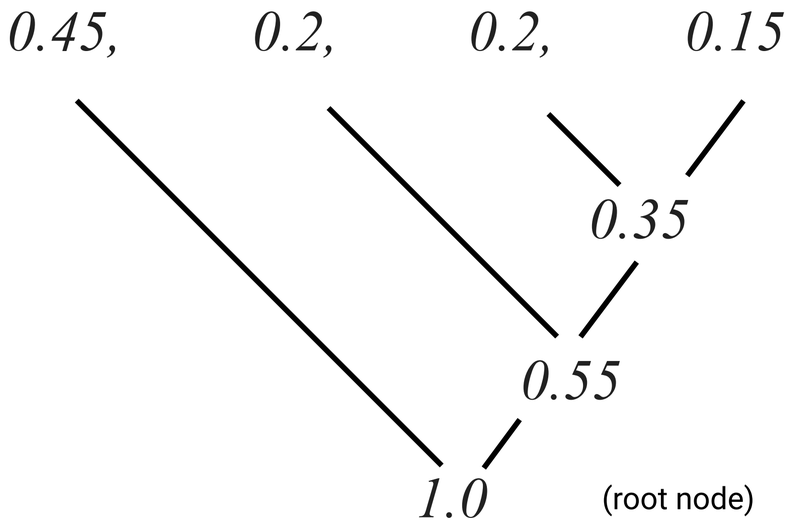
- **Huffman Coding (General)**
  - **Input** $\mathbf{p}=(0.45, 0.2, 0.2, 0.15)$, $\mathbf{v}=(v_1,v_2,v_3,v_4)$, and code alphabet $A=\{0,1\}$
  - **Huffman tree** T**:** Recursively create a node $w=u+v$ with edges to the 2 smallest entries u and v in $\mathbf{p}$ *until* $w = 1$



↗ [3]

Huffman Coding (IV)

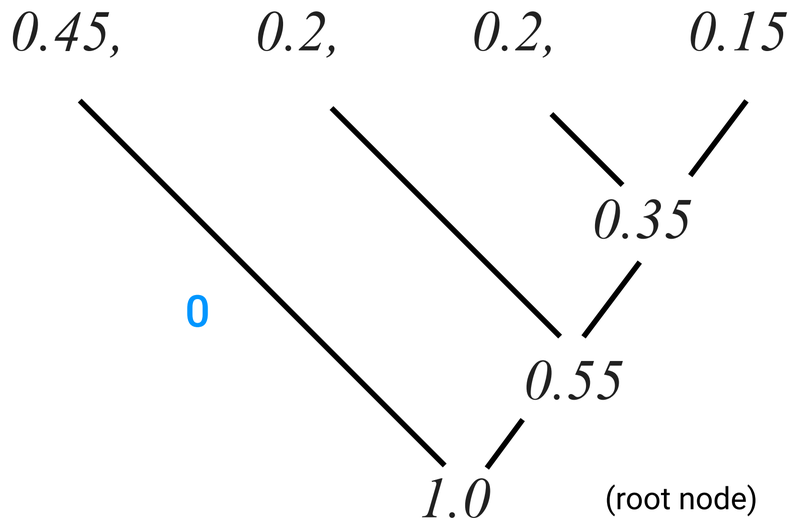
- **Huffman Coding (General)**
  - Assign symbols from A to paths in T, e.g., 0 for left branch, 1 for right branch



↗ [3]

Huffman Coding (IV)

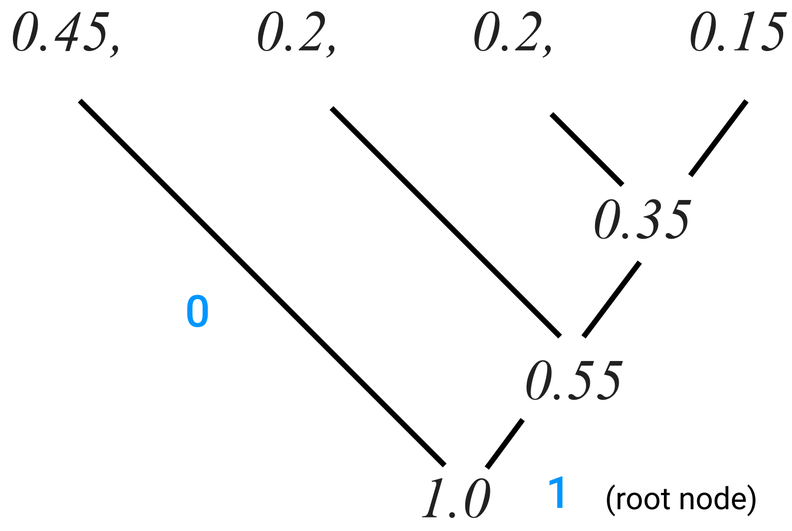
- **Huffman Coding (General)**
  - Assign symbols from A to paths in T, e.g., 0 for left branch, 1 for right branch



↗ [3]

Huffman Coding (IV)

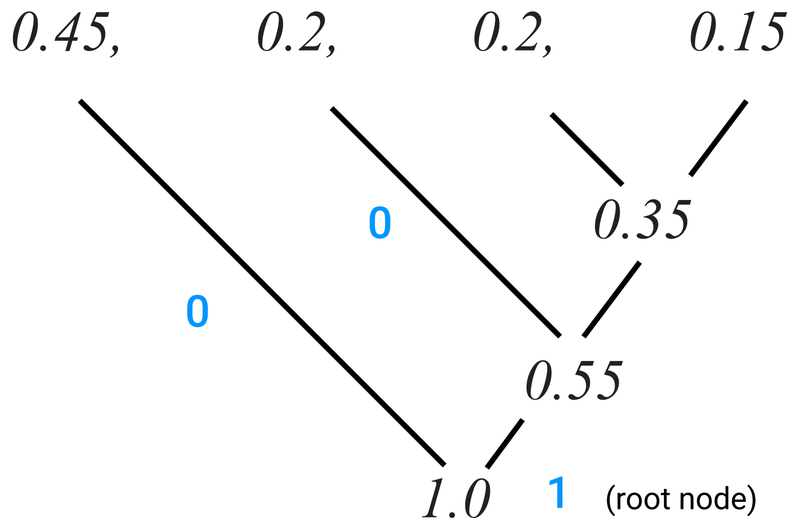
- **Huffman Coding (General)**
  - Assign symbols from A to paths in T, e.g., 0 for left branch, 1 for right branch



↗ [3]

Huffman Coding (IV)

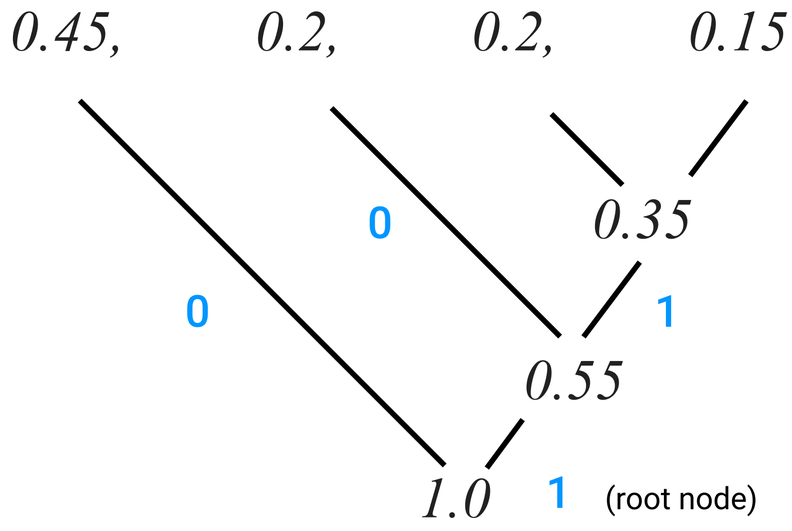
- **Huffman Coding (General)**
  - Assign symbols from A to paths in T, e.g., 0 for left branch, 1 for right branch



↗ [3]

Huffman Coding (IV)

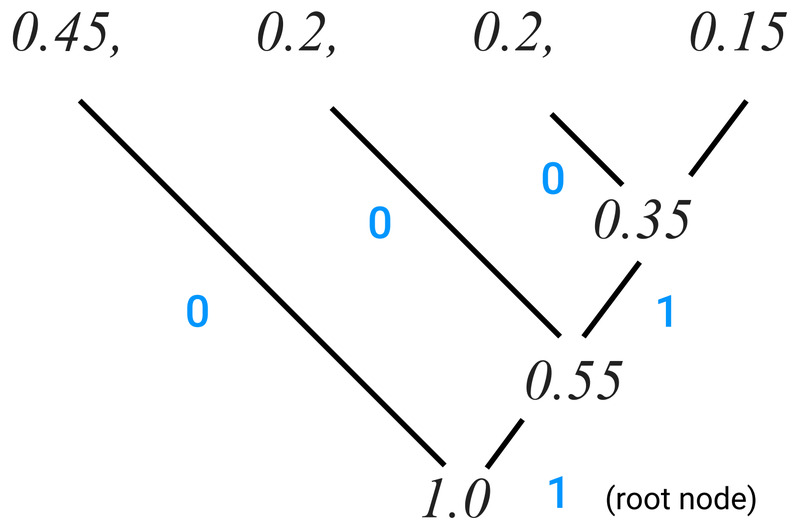
- **Huffman Coding (General)**
  - Assign symbols from A to paths in T, e.g., 0 for left branch, 1 for right branch



↗ [3]

Huffman Coding (IV)

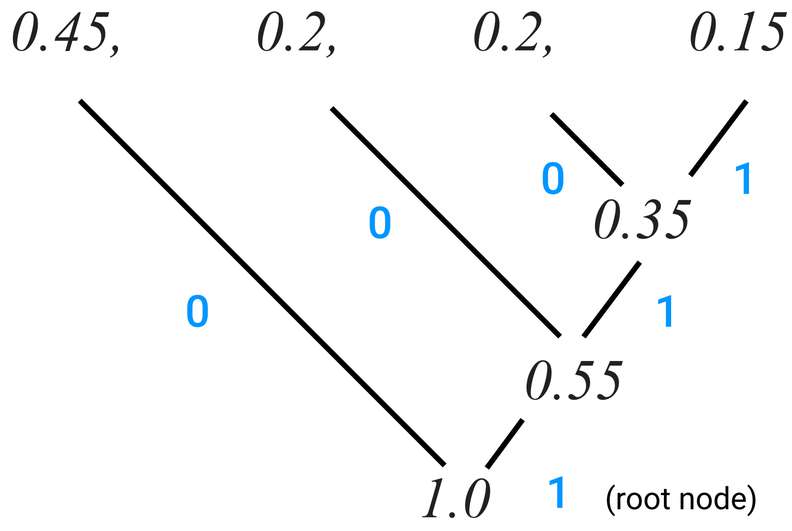
- **Huffman Coding (General)**
  - Assign symbols from A to paths in T, e.g., 0 for left branch, 1 for right branch



↗ [3]

Huffman Coding (IV)

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

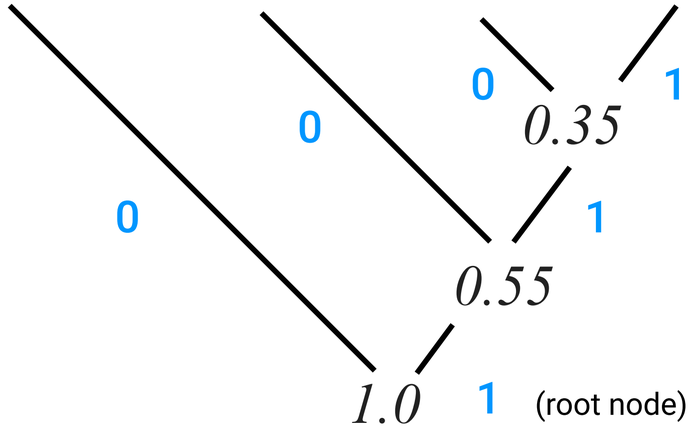


↗ [3]

Huffman Coding (IV)

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

i	0,	1,	2,	4
$code(v_i)$				
p_i	0.45,	0.2,	0.2,	0.15

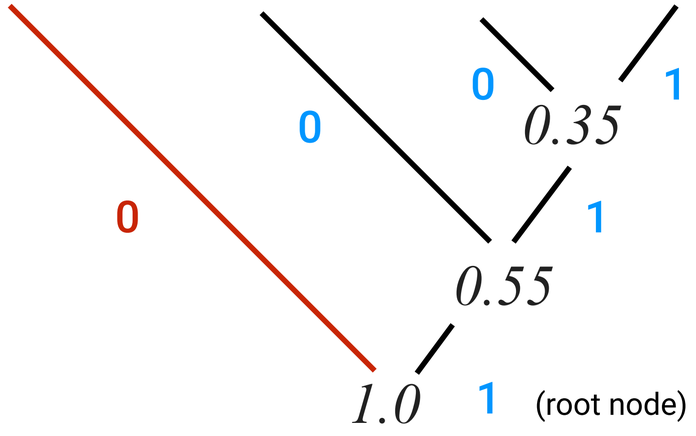


↗ [3]

Huffman Coding (IV)

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

i	0,	1,	2,	4
$code(v_i)$	0			
p_i	0.45,	0.2,	0.2,	0.15

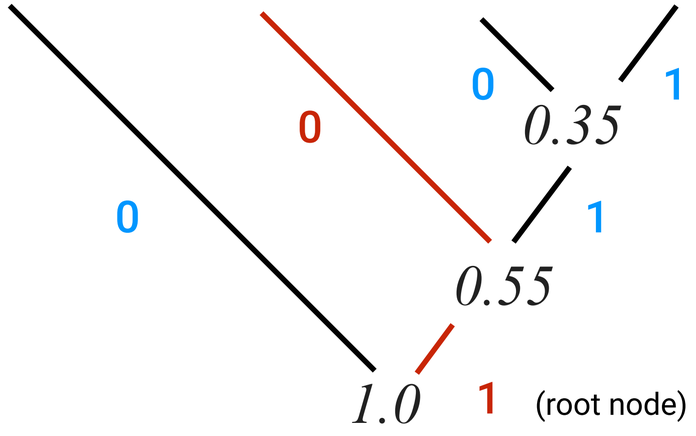


↗ [3]

Huffman Coding (IV)

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

i	0,	1,	2,	4
$code(v_i)$		10		
p_i	0.45,	0.2,	0.2,	0.15

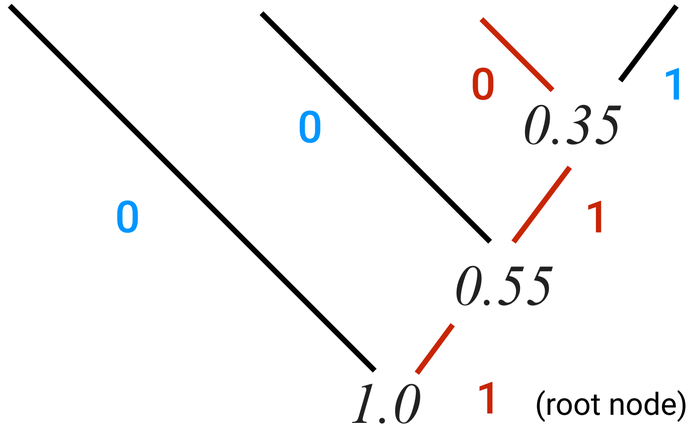


↗ [3]

Huffman Coding (IV)

- **Huffman Coding (General)**
  - Assign code word $code(v_i)$ for each value v_i in $\mathbf{v}$ 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

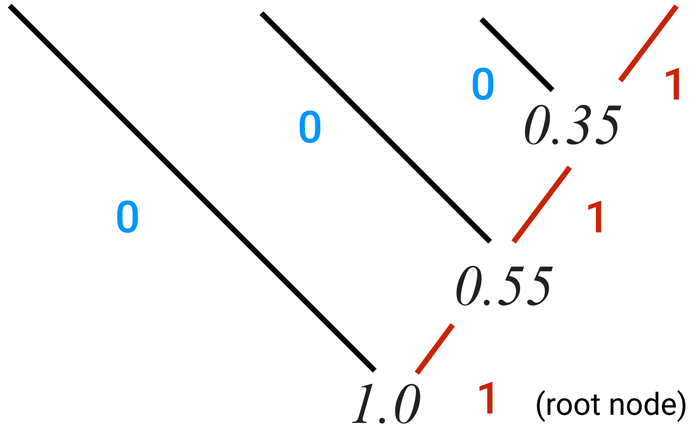


↗ [3]

Huffman Coding (IV)

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

i	0,	1,	2,	4
$code(v_i)$				111
p_i	0.45,	0.2,	0.2,	0.15

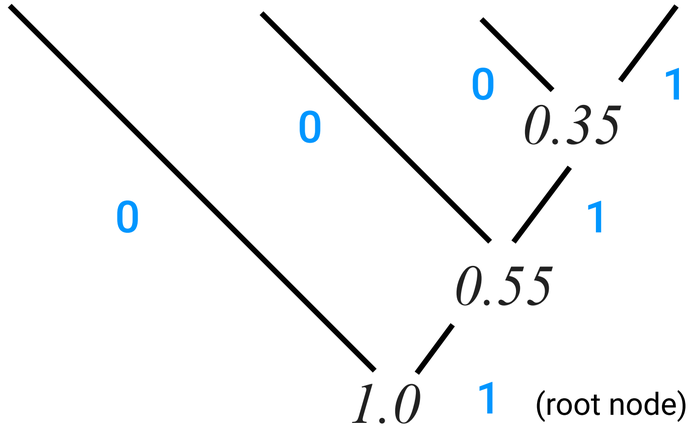


↗ [3]

Huffman Coding (IV)

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

i	0,	1,	2,	4
$code(v_i)$	0	10	110	111
p_i	0.45,	0.2,	0.2,	0.15



↗ [3]

Huffman Coding (IV)

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

i	0,	1,	2,	4
$code(v_i)$	0	10	110	111
p_i	0.45,	0.2,	0.2,	0.15

↗ [3]

Huffman Coding (V)

- **Huffman Coding (Text Compression)**
  - To encode a word $w=w_1w_2\ldots w_m$ with w_j is a value in $\mathbf{v}=(v_1,v_2,\ldots,v_n)$ for $j=1,2,\ldots,m$, run *Huff* where the i-th element in $\mathbf{p}$ 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

↗ [3]

Huffman Coding (V)

- **Huffman Coding (Text Compression)**
  - **Example** $w=,,mississippi''$ with $m=11, \Sigma = \{m,i,s,p\}$
    - $\mathbf{p} = (h_m(,,m``), h_m(,,i``), h_m(,,s``), h_m(,,p``))$

↗ [3]

Huffman Coding (V)

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

↗ [3]

Huffman Coding (V)

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

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

↗ [3]

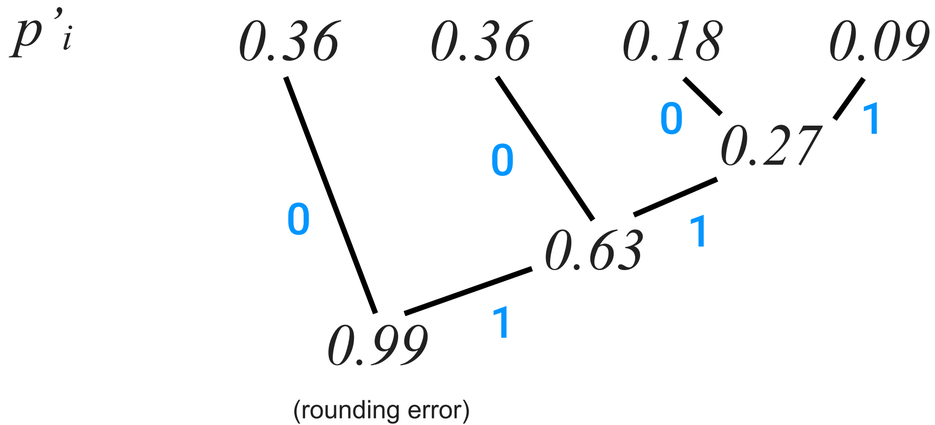
Huffman Coding (V)

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

↗ [3]

Huffman Coding (V)

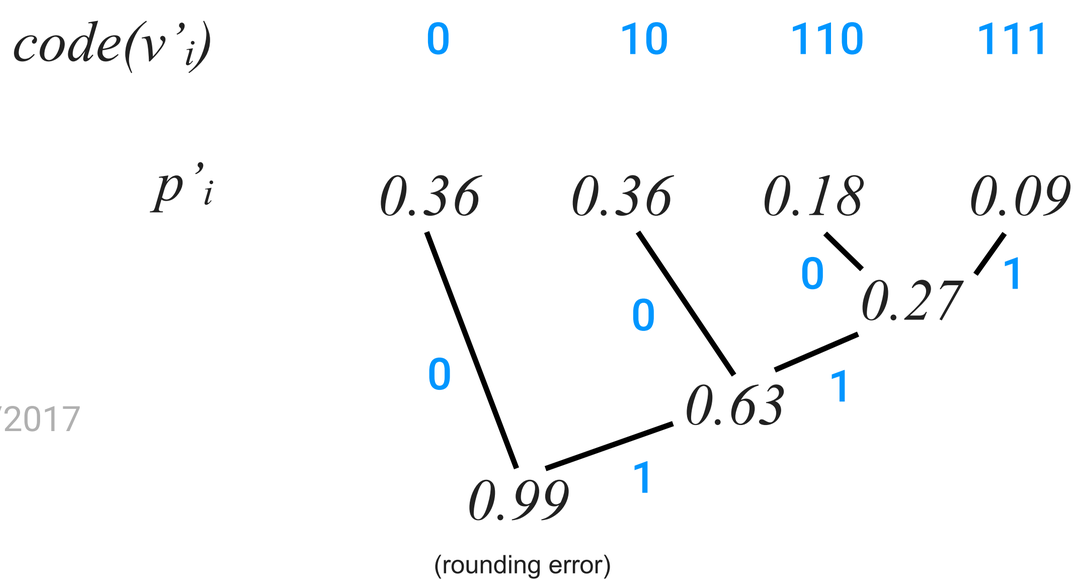
- **Huffman Coding (Text Compression)**
  - **Example** w=„*mississippi*" with $m=11, \Sigma = \{m,i,s,p\}$
    - $\mathbf{p} = (h_m(„m“), h_m(„i“), h_m(„s“), h_m(„p“)) = (0.09, 0.36, 0.36, 0.18)$
    - $\mathbf{p'} = (0.36, 0.36, 0.18, 0.09)$ „ $\mapsto$ $\mathbf{v}'=(i,s,p,m)$“



↗ [3]

Huffman Coding (V)

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

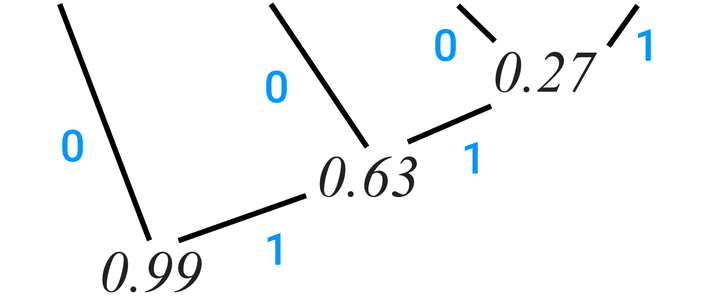


↗ [3]

Huffman Coding (V)

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

$code(v'_i)$	0	10	110	111
v'_i	i	s	p	m
p'_i	0.36	0.36	0.18	0.09



(rounding error)

↗ [3]

Huffman Coding (V)

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

$code(v'_i)$	0	10	110	111
v'_i	i	s	p	m

↗ [3]

Huffman Coding (V)

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

$code(v'_i)$	0	10	110	111
v'_i	i	s	p	m

Encoding *mississippi* $\xrightarrow{Huff}$

111	0	10	10	0	10	10	0	110	110	0
m	i	s	s	i	s	s	i	p	p	i

↗ [3]

Huffman Coding (V)

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

$code(v'_i)$	0	10	110	111
v'_i	i	s	p	m

Encoding

$$mississippi \xrightarrow{Huff} \begin{array}{c} 111\ 0\ 10\ 10\ 0\ 10\ 10\ 0\ 110110\ 0 \\ m\ \ i\ \ s\ \ s\ \ i\ \ s\ \ s\ \ i\ \ p\ \ p\ \ i \end{array}$$

Decoding

$$111\,0\,10\,10\,0\,10\,10\,0\,110\,110\,0 \xrightarrow{Huff^{-1}} m\ \ i\ \ s\ \ s\ \ i\ \ s\ \ s\ \ i\ \ p\ \ p\ \ i$$

Decoding is not ambiguous. Why?

↗ [3]

Huffman Coding (V)

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

$code(v'_i)$	0	10	110	111
v'_i	i	s	p	m

Assuming ASCII encoding of w: $\#bits(mississippi) = 8 \cdot 11 = 88$ bits

$\#bits(Huff(mississippi)) = 21$ bits

76% space saving

↗ [3]

Huffman Coding (VI)

- **Huffman Coding for RDF String Dictionary**
  - Assume list of values in RDF triples

$$w_1 = DataLake$$
$$w_2 = DataBase$$
$$w_3 = DataHub$$

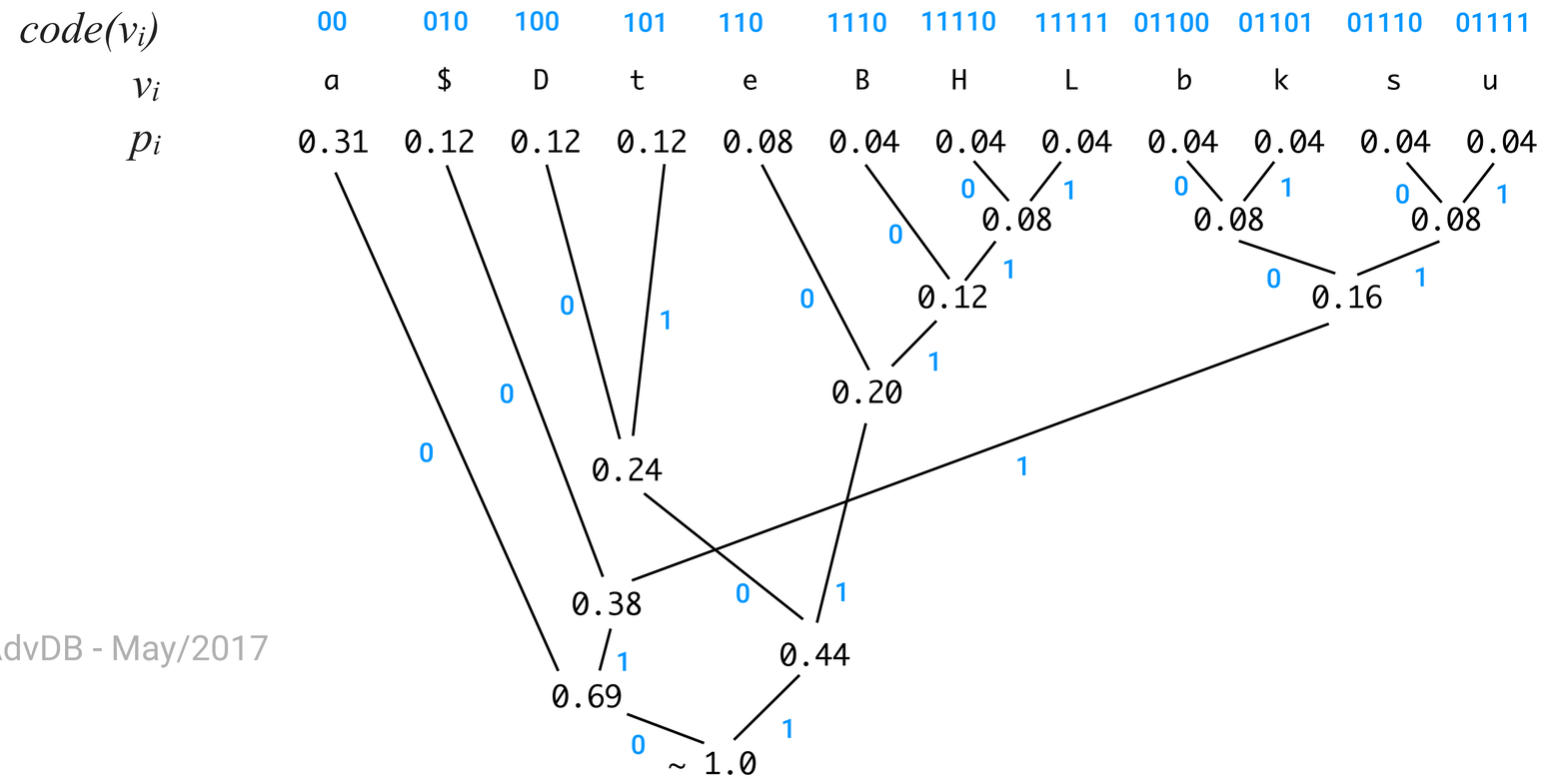
  - Append unique string terminator character (e.g., $) to words w_1, w_2 and w_3

$$w_1' = w_1\$$$
$$w_2' = w_2\$$$
$$w_3' = w_3\$$$

↗ [3]

Huffman Coding (VI)

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



↗ [3]

Huffman Coding (VII)

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

$DataLake\$ = w_1' \mapsto$ 100 00 101 00 11111 00 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

$code(v_i)$	00	010	100	101	110	1110	11110	11111	01100	01101	01110	01111
v_i	a	$	D	t	e	B	H	L	b	k	s	u

String terminator symbol

↗ [3]

Huffman Coding (VIII)

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

	DataLake$				*DataBase$*				*DataHub$*			
array B	10000101	00111110	00110111	0010000	10000101	00111000	01110110	0100000	10000101	00111100	11110110	0010000
offset	0	1	2	3	4	5	6	7	8	9	10	11

pos($Huff$(„DataLake$“), **B**) = 0 $\quad$ $Huff^{-1}$(**B**[0]) = „DataLake“

pos($Huff$(„DataBase$“), **B**)= 4 $\quad$ $Huff^{-1}$(**B**[4]) = „DataBase“

pos($Huff$(„DataHub$“), **B**) = 8 $\quad$ $Huff^{-1}$(**B**[8]) = „DataHub“

↗ [3]

Hashing

Hash Functions + Hash Tables

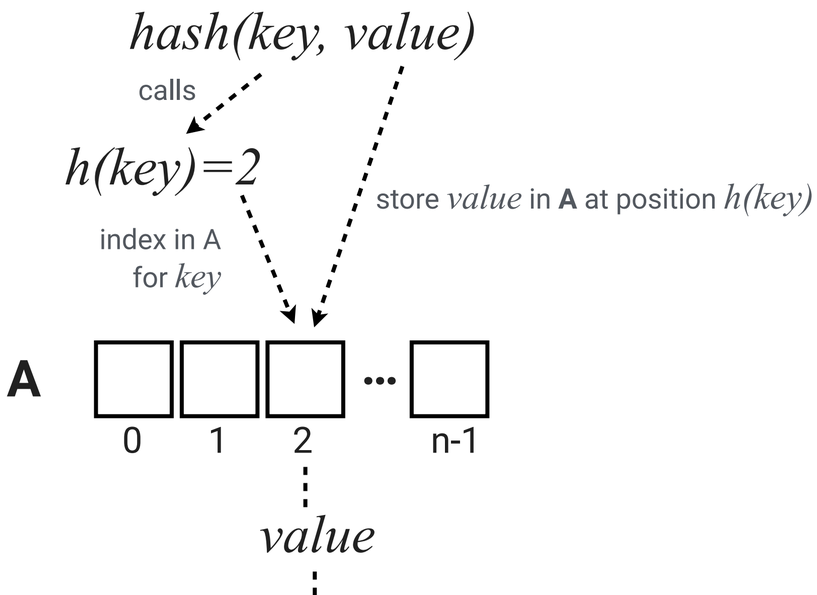
Hash Function

- **Hash *Function***
  - An injective function $h{:}K \longmapsto H$ 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

↗ [3]

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**



↗ [3]

Hash Tables using Huffman Coding

Dictionaries with Huffman Coding (I)

- **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**)

↗ [3]

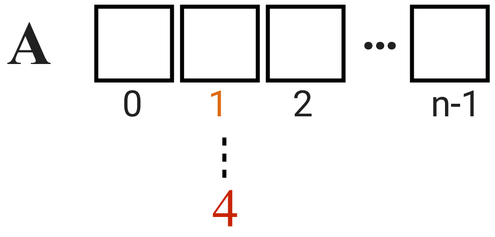
Dictionaries with Huffman Coding (I)

- **Hash Tables + Huffman Coding (Example)**

$$hash(\underline{Huff(,DataBase\$')}, \underline{pos(Huff(,DataBase\$'), \mathbf{B})})$$

$$h(10000101001110000111011001 0)=1$$

(assume h gives that hash value)



↗ [3]

Dictionaries with Huffman Coding (II)

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

`locate(`,Database‘`)` = $h(Huff($,Database$‘$))$ = 1

`extract(1)` = $\text{Huff}^{-1}(\mathbf{B}[1])$ = ,Database‘

↗ [3]

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)

↗ [3]

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)

↗ [3]

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 sub*(***S***, start, end)* that is in **S** (*start* including, *end* excluding)
  - For instance, sub*("Topics in Databases", 3, 8)* = *"ics i"*

↗ [3]

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 sub*(***S***, start, end)* that is in **S** (*start* including, *end* excluding)
  - For instance, sub*("Topics in Databases", 3, 8)* = *"ics i"*
- **Prefix** is a substring **pre***(***S***, end)* with start index 0, i.e., **pre***(***S***, end)* = sub*(***S***, 0, end)*

↗ [3]

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 sub*(***S***, start, end)* that is in **S** (*start* including, *end* excluding)
  - For instance, sub*("Topics in Databases", 3, 8)* = *"ics i"*
- **Prefix** is a substring **pre***(***S***, end)* with start index 0, i.e., **pre***(***S***, end)* = sub*(***S***, 0, end)*
- **Suffix** is a substring **suf***(***S***, start)* with end index |**S**|, i.e., **suf***(***S***, start)* = sub*(***S***, start,* |**S**|*)*

↗ [3]

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

↗ [3]

Incremental Coding (II)

Input (sorted)	Common Prefix	Compressed Output
courses:AdvDB	no preceding word	
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 $\mathbf{S}$ is string $\text{sub}(\mathbf{S}, \textit{start}, \textit{end})$ that is in $\mathbf{S}$ ($\textit{start}$ including, $\textit{end}$ excluding)
  - For instance, $\text{sub}(\text{“}\textit{Topics in Databases}\text{“}, 3, 8) = \text{“}\textit{ics i}\text{“}$
- **Prefix** is a substring $\mathbf{pre}(\mathbf{S}, \textit{end})$ with start index 0, i.e., $\mathbf{pre}(\mathbf{S}, \textit{end}) = \text{sub}(\mathbf{S}, 0, \textit{end})$
- **Suffix** is a substring $\mathbf{suf}(\mathbf{S}, \textit{start})$ with end index $|\mathbf{S}|$, i.e., $\mathbf{suf}(\mathbf{S}, \textit{start}) = \text{sub}(\mathbf{S}, \textit{start}, |\mathbf{S}|)$
- Store $(|\mathbf{p}|, \mathbf{suf}(\mathbf{S}, |\mathbf{p}|))$ for the i-th entry $\mathbf{S}$, if $\mathbf{S}$ shares suffix $\mathbf{p}$ with $(i\text{-}1)$-th entry, and $(0, \mathbf{S})$ otherwise

↗ [3]

Incremental Coding (II)

Input (sorted)	Common Prefix	Compressed Output
courses:AdvDB	no preceding word	0, courses:AvDB
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 $\mathbf{S}$ is string $\text{sub}(\mathbf{S}, \textit{start}, \textit{end})$ that is in $\mathbf{S}$ ($\textit{start}$ including, $\textit{end}$ excluding)
  - For instance, $\text{sub}(\text{“}\textit{Topics in Databases}\text{“}, 3, 8) = \text{“}\textit{ics i}\text{“}$
- **Prefix** is a substring $\mathbf{pre}(\mathbf{S}, \textit{end})$ with start index 0, i.e., $\mathbf{pre}(\mathbf{S}, \textit{end}) = \text{sub}(\mathbf{S}, 0, \textit{end})$
- **Suffix** is a substring $\mathbf{suf}(\mathbf{S}, \textit{start})$ with end index $|\mathbf{S}|$, i.e., $\mathbf{suf}(\mathbf{S}, \textit{start}) = \text{sub}(\mathbf{S}, \textit{start}, |\mathbf{S}|)$
- Store $(|\mathbf{p}|, \mathbf{suf}(\mathbf{S}, |\mathbf{p}|))$ for the i-th entry $\mathbf{S}$, if $\mathbf{S}$ shares suffix $\mathbf{p}$ with $(i\text{-}1)$-th entry, and $(0, \mathbf{S})$ otherwise

↗ [3]

Incremental Coding (II)

Input (sorted)	Common Prefix	Compressed Output
courses:AdvDB	no preceding word	0, courses:AvDB
ovgu:Breß	no common prefix	
ovgu:Bronesk		
ovgu:Dorok		
ovgu:Meister		
vocab:occurs		

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

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

↗ [3]

Incremental Coding (II)

Input (sorted)	Common Prefix	Compressed Output
courses:AdvDB	no preceding word	0, courses:AvDB
ovgu:Breß	no common prefix	0, 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 $\mathbf{S}$ is string sub$(\mathbf{S}, start, end)$ that is in $\mathbf{S}$ ($start$ including, end excluding)
  - For instance, sub$(\text{“}Topics\ in\ Databases\text{“}, 3, 8)$ = $\text{“}ics\ i\text{“}$
- **Prefix** is a substring $\mathbf{pre}(\mathbf{S}, end)$ with start index 0, i.e., $\mathbf{pre}(\mathbf{S}, end) = \text{sub}(\mathbf{S}, 0, end)$
- **Suffix** is a substring $\mathbf{suf}(\mathbf{S}, start)$ with end index $|\mathbf{S}|$, i.e., $\mathbf{suf}(\mathbf{S}, start) = \text{sub}(\mathbf{S}, start, |\mathbf{S}|)$
- Store $(|\mathbf{p}|, \mathbf{suf}(\mathbf{S}, |\mathbf{p}|))$ for the i-th entry $\mathbf{S}$, if $\mathbf{S}$ shares suffix $\mathbf{p}$ with $(i\text{-}1)$-th entry, and $(0, \mathbf{S})$ otherwise

↗ [3]

Incremental Coding (II)

Input (sorted)	Common Prefix	Compressed Output
courses:AdvDB	no preceding word	0, courses:AvDB
ovgu:Breß	no common prefix	0, ovgu:Breß
ovgu:Bronesk	ovgu:Br	
ovgu:Dorok		
ovgu:Meister		
vocab:occurs		

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

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

↗ [3]

Incremental Coding (II)

Input (sorted)	Common Prefix	Compressed Output
courses:AdvDB	no preceding word	0, courses:AvDB
ovgu:Breß	no common prefix	0, ovgu:Breß
ovgu:Bronesk	ovgu:Br	7, onesk
ovgu:Dorok		
ovgu:Meister		
vocab:occurs		

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

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

↗ [3]

Incremental Coding (II)

Input (sorted)	Common Prefix	Compressed Output
courses:AdvDB	no preceding word	0, courses:AvDB
ovgu:Breß	no common prefix	0, ovgu:Breß
ovgu:Bronesk	ovgu:Br	7, onesk
ovgu:Dorok	ovgu:	
ovgu:Meister		
vocab:occurs		

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

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

↗ [3]

Incremental Coding (II)

Input (sorted)	Common Prefix	Compressed Output
courses:AdvDB	no preceding word	0, courses:AvDB
ovgu:Breß	no common prefix	0, ovgu:Breß
ovgu:Bronesk	ovgu:Br	7, onesk
ovgu:Dorok	ovgu:	5, Dorok
ovgu:Meister		
vocab:occurs		

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

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

↗ [3]

Incremental Coding (II)

Input (sorted)	Common Prefix	Compressed Output
courses:AdvDB	no preceding word	0, courses:AvDB
ovgu:Breß	no common prefix	0, ovgu:Breß
ovgu:Bronesk	ovgu:Br	7, onesk
ovgu:Dorok	ovgu:	5, Dorok
ovgu:Meister	ovgu:	
vocab:occurs		

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

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

↗ [3]

Incremental Coding (II)

Input (sorted)	Common Prefix	Compressed Output
courses:AdvDB	no preceding word	0, courses:AvDB
ovgu:Breß	no common prefix	0, ovgu:Breß
ovgu:Bronesk	ovgu:Br	7, onesk
ovgu:Dorok	ovgu:	5, Dorok
ovgu:Meister	ovgu:	5, Meister
vocab:occurs		

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

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

↗ [3]

Incremental Coding (II)

Input (sorted)	Common Prefix	Compressed Output
courses:AdvDB	no preceding word	0, courses:AvDB
ovgu:Breß	no common prefix	0, ovgu:Breß
ovgu:Bronesk	ovgu:Br	7, onesk
ovgu:Dorok	ovgu:	5, Dorok
ovgu:Meister	ovgu:	5, Meister
vocab:occurs	no common prefix	

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

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

↗ [3]

Incremental Coding (II)

Input (sorted)	Common Prefix	Compressed Output
courses:AdvDB	no preceding word	0, courses:AvDB
ovgu:Breß	no common prefix	0, ovgu:Breß
ovgu:Bronesk	ovgu:Br	7, onesk
ovgu:Dorok	ovgu:	5, Dorok
ovgu:Meister	ovgu:	5, Meister
vocab:occurs	no common prefix	0, vocab:occurs

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

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

↗ [3]

Incremental Coding (III)

- **Optimization** Use delta coding for *length* when storing pair *(length,* **suffix***)* as well

Compressed Output

0	c	o	u	r	s	e	s	:	A	d	v	D	B				0, courses:AvDB
0	o	v	g	u	:	B	r	e	ß								0, ovgu:Breß
+7	o	n	e	s	k												7, onesk
-2	D	o	r	o	k												5, Dorok
0	M	e	i	s	t	e	r										5, Meister

↑ delta encoded **prefix** *length*

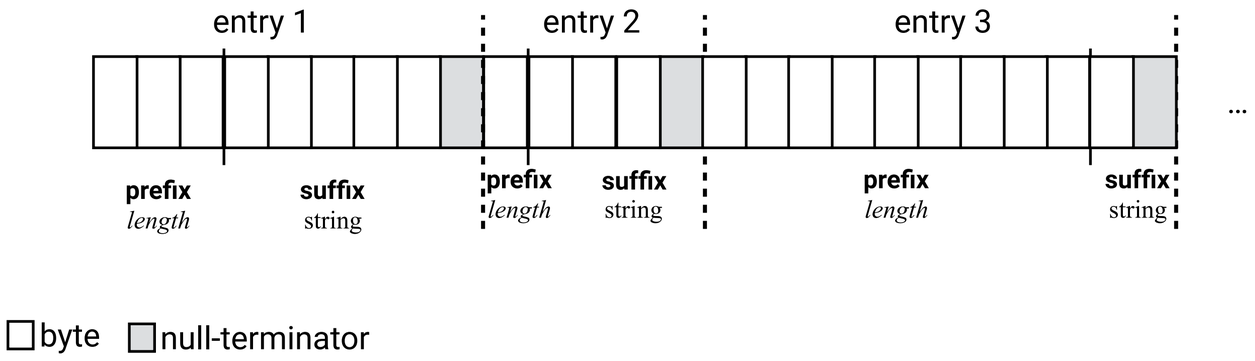
↑ delta encoded **suffix**

...

↗ [3]

Plain Front Coding

- Further optimizations possible, e.g., variable-length encoded prefix-length and variable-length suffix strings (Plain Front Coding)



↗ [3]

Trie-Based

Trie-Based

- *Trie* related to Information Re*trie*val
- 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“**)

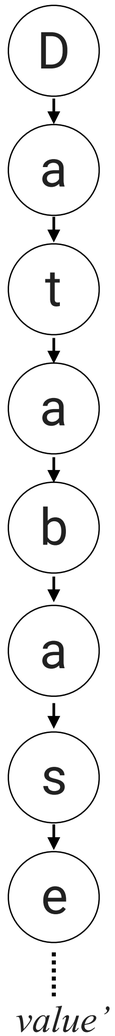


Figure: Prefix Tree

Trie-Based

- *Trie* related to Information Re*trie*val
- 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“**)

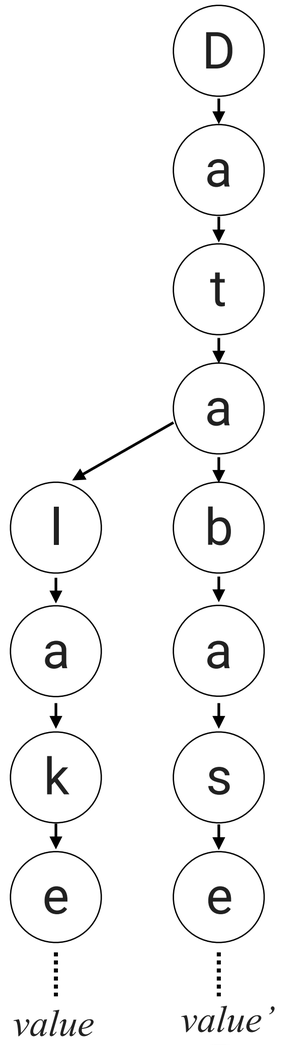


Figure: Prefix Tree

↗ [11]

Trie-Based

- *Trie* related to Information Re*trie*val
- 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“**)

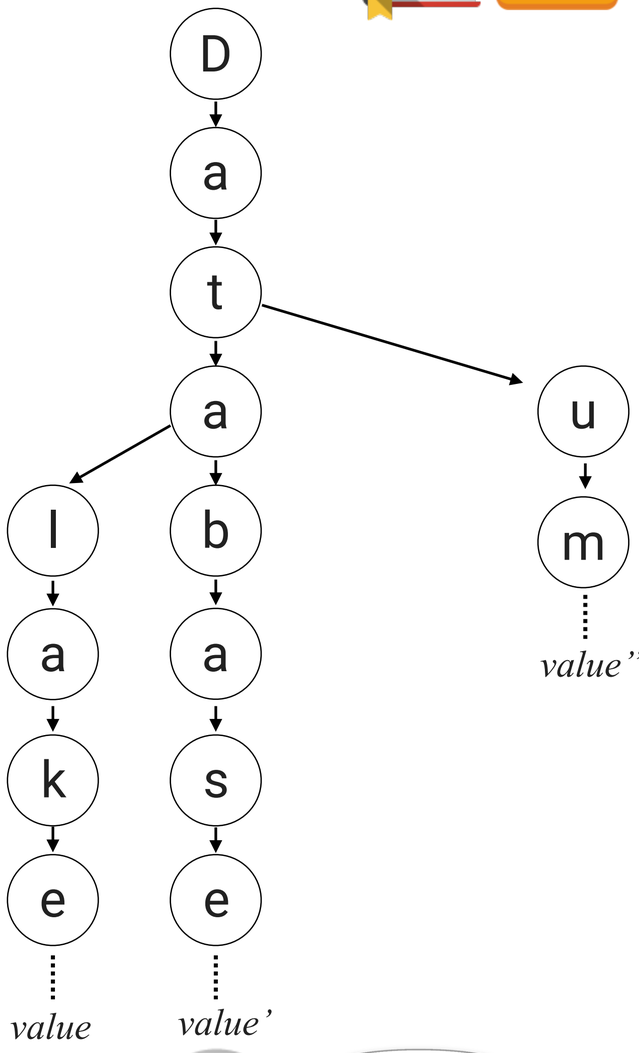


Figure: Prefix Tree

↗ [11]

Trie-Based

- *Trie* related to Information Re*trie*val
- 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“**)

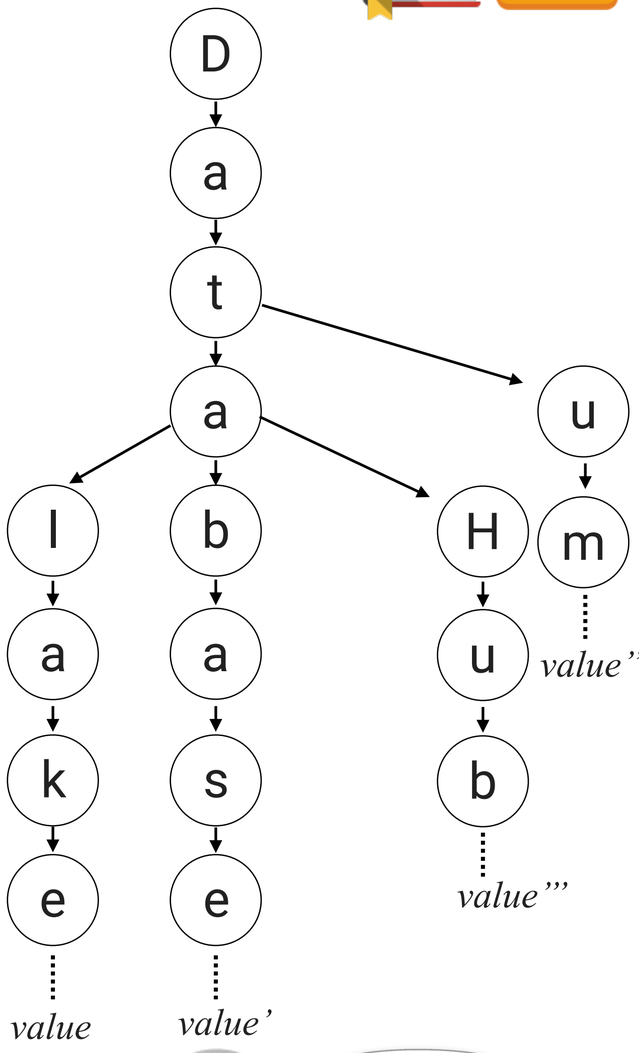


Figure: Prefix Tree

Trie-Based

- *Trie* related to Information Re*trie*val
- 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“**)

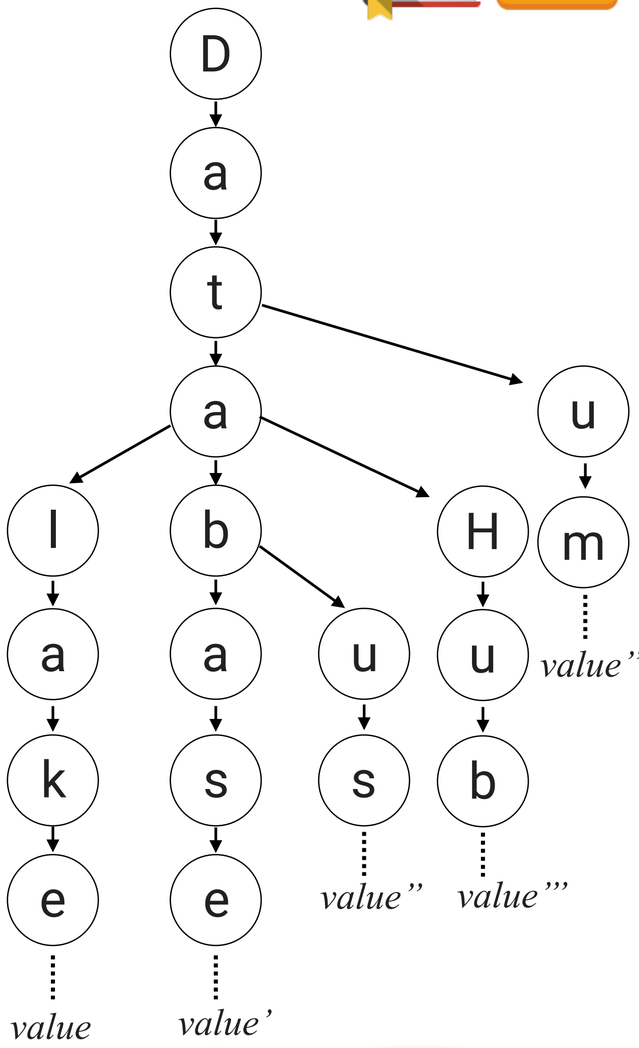


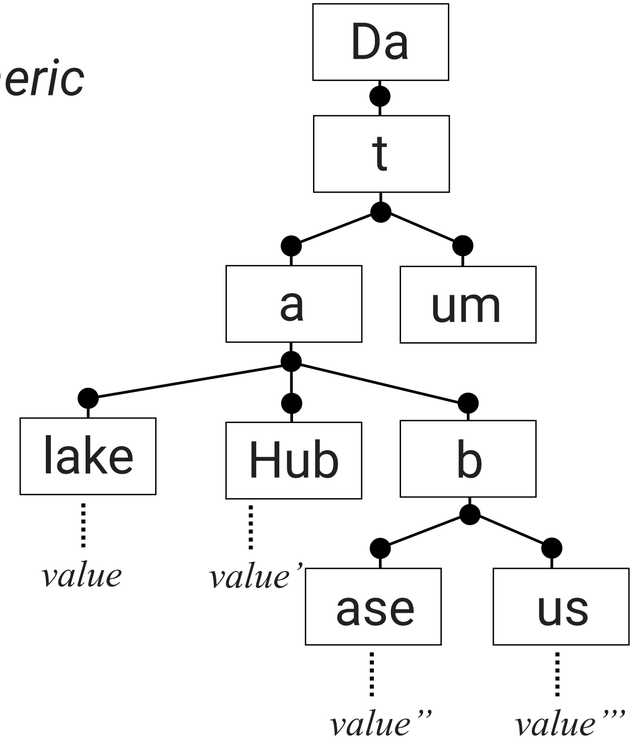
Figure: Prefix Tree

↗ [11]

PATRICIA Trie-Based

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



https://www.ietf.org/rfc/rfc3987.txt

↗ [11]

Grammar-Based Codes

Grammar-Based (I)

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

“alibaba“ ⟶

```
S → ABB
A → ali
B → ba
```

  - Covered in this lecture
    - **Sequitur**
    - **RE-PAIR**

↗ [2]

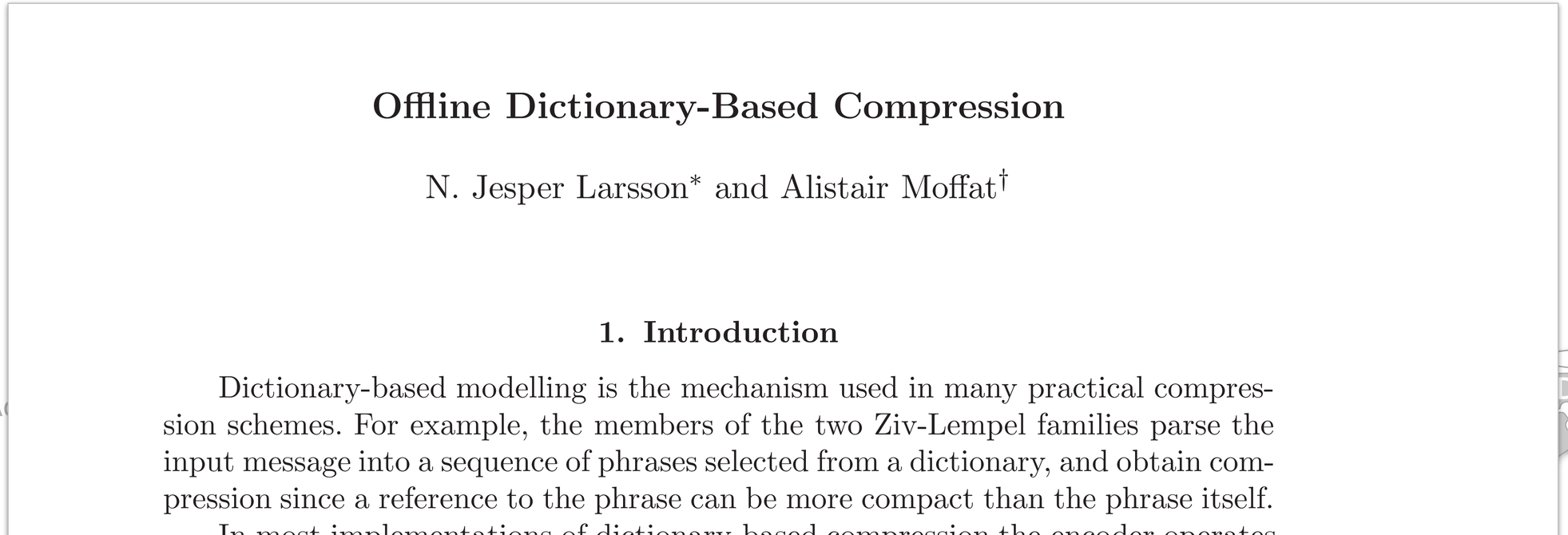
Grammar-Based (II)

- **Sequitur algorithm**
  - Rule with start symbol $S \rightarrow w$ is created
  - Input w is linearly scanned yielding digrams $d_1, d_2, \ldots, d_m$
    - $w =$ "abaaba" → $d_1 =$ "ab", $d_2 =$ "ba", $d_3 =$ "aa", $d_4 =$ "ab", $d_5 =$ "ba"
  - Two constraints during grammar construction are considered
    - **Digram uniqueness**: if a digram d occurring multiple time are replaced in w by a non-terminal D, and a rule $D \rightarrow d$ is created
      - S → abaaba ···▶ S → AaAa, A → ab ···▶ S → BB, B → Aa, A → ab
    - **Rule utility**: Rules occurring only once on right side are removed by their content
      - S → BB, B → Aa, A → ab ···▶ <u>S → BB, B → aba</u>

↗ [2]

Grammar-Based (III)

- **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

↗ [2,4]

Wrap Up

Part IV

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

Literature (I)

[1] Curé, Olivier and Blin, Guillaume. *Chapter One - Introduction,* RDF Database Systems. Morgan Kaufmann. Pages 1 - 8, ISBN 978-0-12-799957-9, 2015

[2] Curé, Olivier and Blin, Guillaume. *Chapter Three - RDF and the Semantic Web Stack,* RDF Database Systems. Morgan Kaufmann. Pages 41 - 80, ISBN 978-0-12-799957-9, 2015

[3] Curé, Olivier and Blin, Guillaume. *Chapter Four - RDF Dictionaries: String Encoding* , RDF Database Systems. Morgan Kaufmann. Pages 81 - 104, ISBN 978-0-12-799957-9, 2015

[4] Larsson, N Jesper and Moffat, Alistair. *Off-line dictionary-based compression*. Proceedings of the IEEE volume 88 number 11, pages 1722–1732, 2000

[5] Srinivasa S. *Data, Storage and Index Models for Graph Databases.* Graph Data Management: Techniques and Applications. Pages 47–70, 2011

[6] Tim Berners-Lee. Semantic Web Road map, url http://www.w3.org/DesignIssues/Semantic.html, last access 2016/06/13

Literature (II)

[7] W3C RDF 1.1 Concepts and Abstract Syntax, url http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/, last access 2016/06/13

[8] W3C XML Schema Part 2: Datatypes Second Edition, url http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/#dt-basetype, last access 2016/06/13

[9] Tim Berners-Lee. Semantic Web - XML2000, url http://www.w3.org/2000/Talks/1206-xml2k-tbl/slide10-0.html, last access 2016/06/13

[10] RFC 3987: Internationalized Resource Identifiers (IRIs), url https://www.ietf.org/rfc/rfc3987.txt, last access 2016/06/13

[11] Blochwitz Ch., Joseph J.M., Backasch R., Pionteck T., Werner S., Heinrich S., Groppe S.: *An optimized radix-tree for hardware-accelerated dictionary generation for semantic web databases*. Pages 1--7, To appear, ReConFig 2015, 2016

Further Reading

- Bazoobandi H.R., De Rooij S, Urbani J, ten Teije A., Frank van Harmelen F., and Bal H. *A Compact In-Memory Dictionary for RDF data*. ESWC2015, 2015
- Bannai H.. *Grammar Compression.* Encyclopedia of Algorithms. Pages 861–866, 2016
- Mavlyutov, R. and Wylot, M. and Cudré-Mauroux, P.. *A Comparison of Data Structures to Manage URIs on the Web of Data*. ESWC Springer, pages 137-151, 2015
- Nader H. Bshouty and Geoffrey T. Falk. *Compression of Dictionaries via Extensions to Front Coding*.ICCI'92. pages 361–364, 1992
- Leis V., Kemper A. and Neumann T.. *The adaptive radix tree: ARTful indexing for main-memory databases*. ICDE 2013, pages 38–49, 2013