XML Compression Technologies

Traditional Compression Technology

Data Compression

- Pros
  - Disk space reduction
  - Network bandwidth saving
  - Better system performance
- Cons
  - Processing overhead
  - Loss of some subtle information

Traditional Compression Schemes

- Lossy Compression
  - DCT, Wavelet
- Lossless Compression
  - Static compression
    - Uses fixed statistics or no statistics is used
  - Semi-adaptive compression (2 scans of data)
    - Statistics gathering and compression
  - Adaptive compression (1 scan of data)
    - Dynamic statistics gathering

Lossless Compression(static)

- Dictionary Encoding
  - Assigns an ID to each new word

  input: ABC ABC BC DDD

  Compressed Data: 1 1 2 3

  Dictionary: ABC =1, BC = 2, DDD=3
- Binary Encoding
  - Binary representation of numeric data

  input: “100” “20” “50”

  Encoding: 100 20 50

Lossless Compression(static)

- Differential Encoding (or Delta Encoding)
  - Replaces a data item with a code value that defines its relationship to a specific data item

ex)

input: 100 120 130

Compressed Data: 100 20 30

input: Johnson Jonah Jones Jorgenson

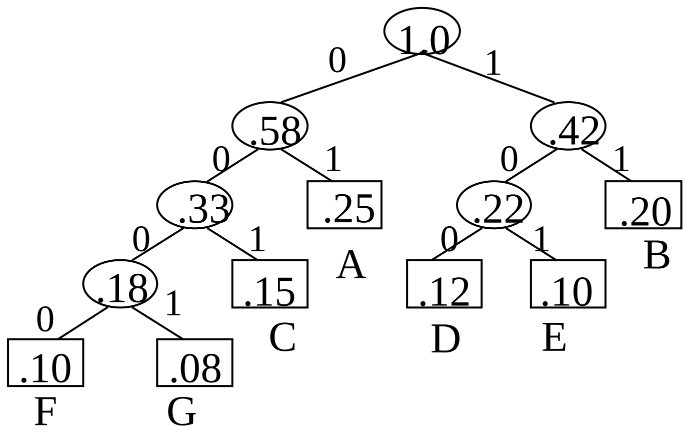
Compressed Data: (0) Johnson (2)nah (3)es (2)rgenson

Lossless Compression(semi-adaptive)

- Huffman Encoding
  - Assigns shorter codes to more frequently appearing symbols and longer codes to less frequently appearing symbols
  - ex)

input: ACE

Encoding:01001101



Huffman tree

Lossless Compression(adaptive)

- LZ(Lempel-Ziv) Coding
  - Adaptive dictionary encoding
  - Converts variable-length strings into fixed-length codes

  Input: {A B AB AA ABA}

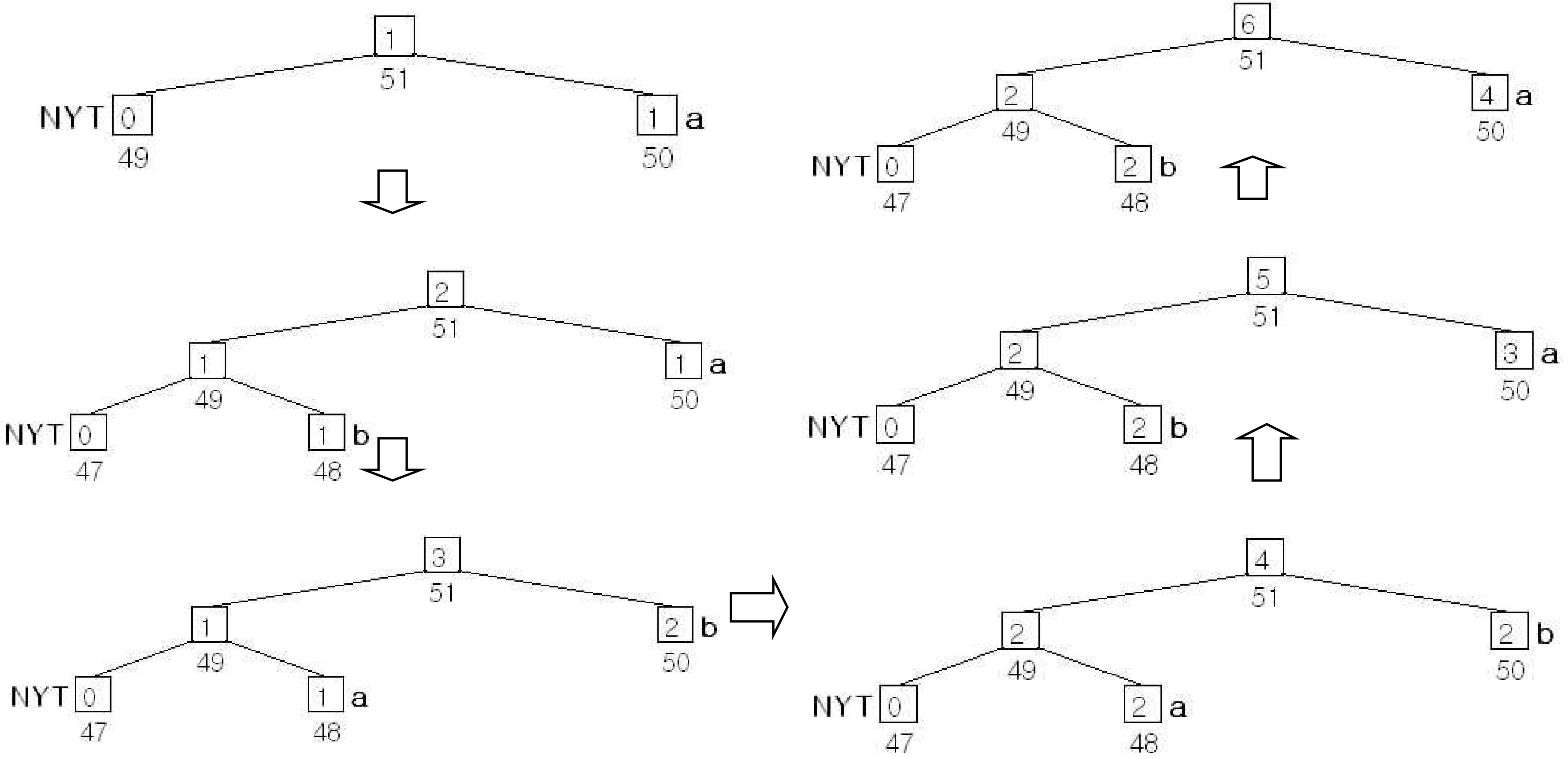
  Compressed Data: {(0,A)(0,B)(1,B)(1,A)(3,A)}

  - new table entry is coded as (i,c)
    - i : the codeword for the existing table entry(12 bit)
    - c : the appended character(8bit)

Lossless Compression(adaptive)

- Adaptive Huffman Coding

input : abbaaa Compressed Data : a 0b 01 01 01 1



XML Compression Technology

Introduction

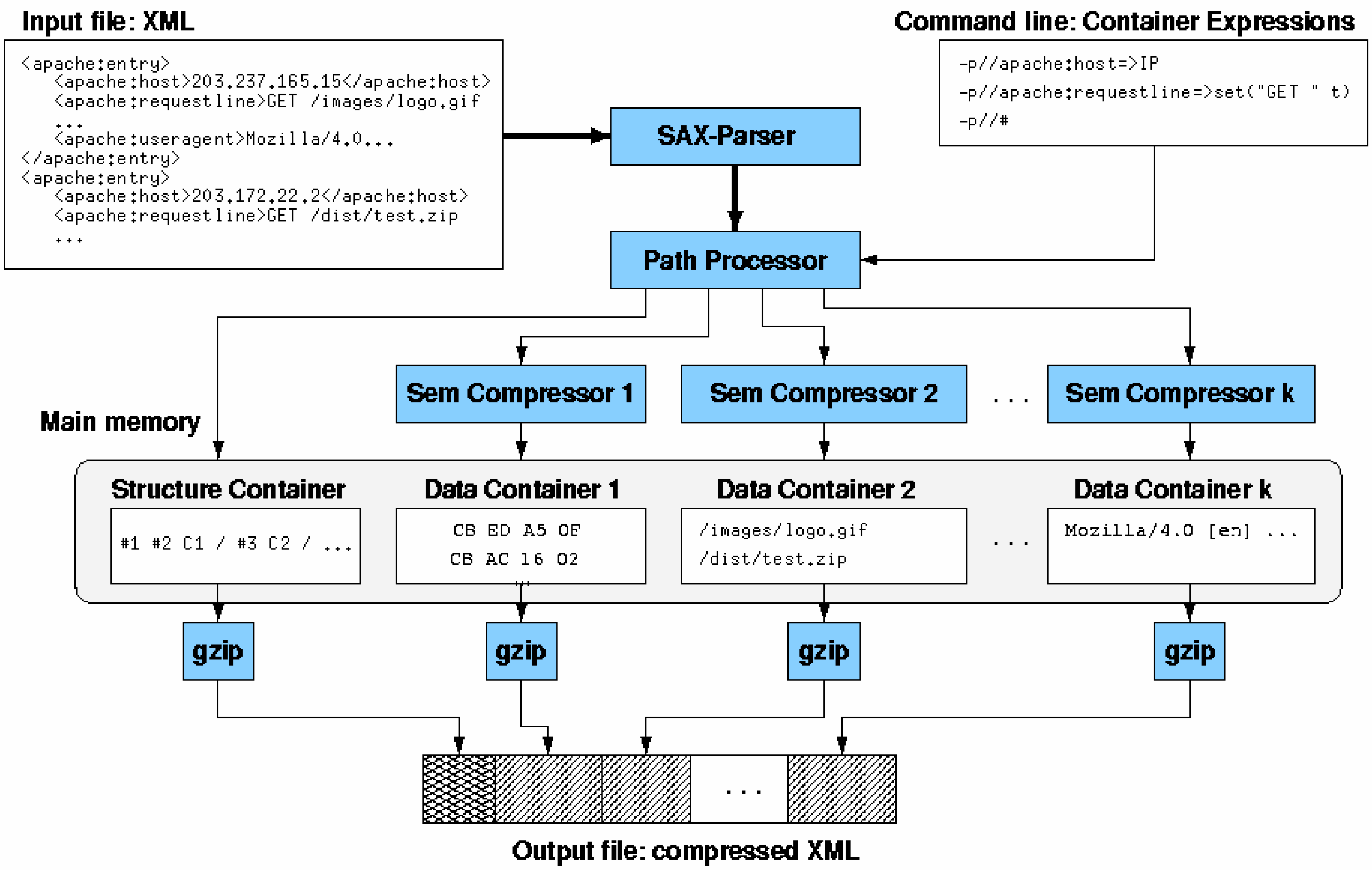
- Currently, large portions of XML data are in native file format
- Disk space and network bandwidth are expensive
- Efficient management of file based XML is needed
- XML compression can be useful
- Applications:
  - XML Search Engines
  - PDA

XML Compression

- XMILL: Hartmut Liefke, Dan Suciu, An Efficient Compressor for XML Data. SIGMOD 2000
- XGrind: Pankaj Tolani, Jayant R. Haritsa, A Query-friendly XML Compressor. ICDE 2002
- XPRESS: Jun-ki Min, Myung-Jae Park, Chin-Wan Chung, A Queriable Compression for XML Data. SIGMOD 2003

XMILL

- **Not intended to support direct querying** the compressed document.
- Physically separates structure(e.g., tag) and content(e.g. value)
  - Tags: dictionary encoding
  - Values: no encoding or user specified encoding
    - Need human's interference
- Groups semantically related data values into containers
- Finally, recompressed by a built-in compression library *zlib*(adaptive compression)



Input file: XML
<apache:entry>
<apache:host>203.237.165.15</apache:host>
<apache:requestline>GET /images/logo.gif
...
<apache:useragent>Mozilla/4.0...
</apache:entry>
<apache:entry>
<apache:host>203.172.22.2</apache:host>
<apache:requestline>GET /dist/test.zip
...
Command line: Container Expressions
-p//apache:host=>IP
-p//apache:requestline=>set("GET " t)
-p//#
SAX-Parser
Path Processor
Sem Compressor 1
Sem Compressor 2
. . .
Sem Compressor k
Main memory
Structure Container
#1 #2 C1 / #3 C2 / ...
Data Container 1
CB ED A5 0F
CB AC 16 02
Data Container 2
/images/logo.gif
/dist/test.zip
. . .
Data Container k
Mozilla/4.0 [en] ...
gzip
gzip
gzip
gzip
. . .
Output file: compressed XML

Drawback of XMILL

- No XML Schema-aware
  - Even though there is information on XML data, XMILL ignores
- Direct query evaluation is not possible
  - When a document is compressed by XMILL, the entire document needs to be decompressed for query evaluation.
- No existing XML indexes can be used for efficient query processing

XGrind

- Homomorphic Compression
  - Preserves the structure of the original XML data in compressed XML data
  - Thus, Supports direct querying the compressed XML data

```
<A>
    <B> v1 </B>
    <B></B>
    <B>v2</B>
</A>
```

(a) Original XML

```
T1
    T2 encode(v1) /
    T2 /
    T2 encode(v2) /
/
```

(b) Homomorphic

- XML indexes can still be used on compressed XML document

XGrind

- XML Schema-aware: DTD
  - Tag : Dictionary Encoding
  - Values : Two kinds of data
    - General Value : Huffman Encoding
    - Enumeration Typed Value : Dictionary Encoding
- Use existing methods

A Compression Example of Xgrind

```
<!- student.xml -->
<STUDENT rollno = "604100418">
      <NAME>Pankaj Tolani</NAME>
      <YEAR>2000</YEAR>
      <PROG>Master of Engineering</PROG>
      <DEPT name = "Computer Science">
</STUDENT>
```

Fragment of the Student DB

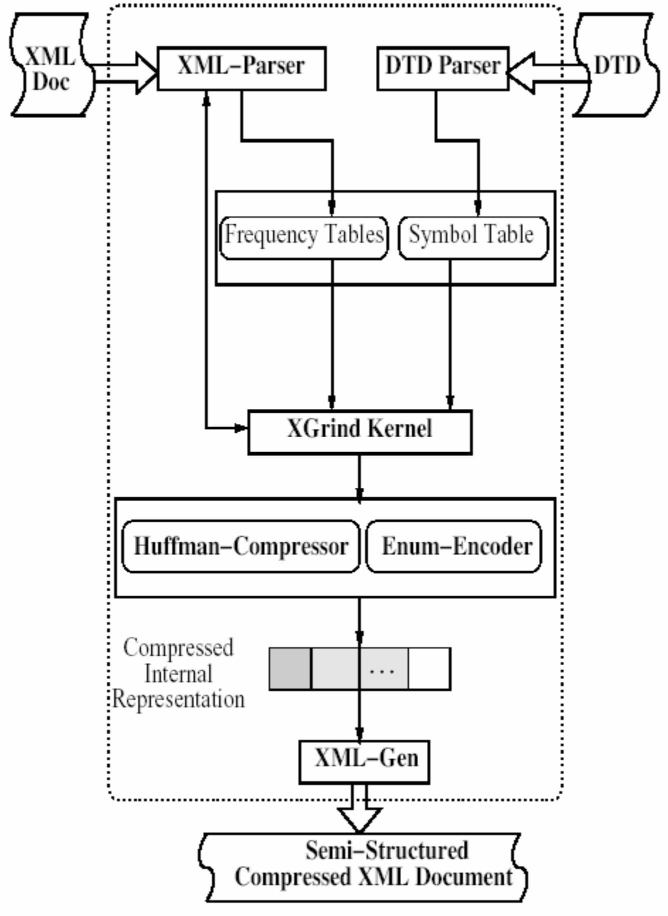
```
<!- DTD for the Student database -->
<!ELEMENT STUDENT (NAME, YEAR, PROG, DEPT)>
<!ATTLIST STUDENT rollno CDATA #REQUIRED>
<!ELEMENT NAME (#PCDATA)>
<!ELEMENT YEAR (#PCDATA)>
<!ELEMENT PROG (#PCDATA)>
<!ELEMENT DEPT EMPTY>
<!ATTLIST DEPT name (Computer Science
| Electrical Engineering
...
| Physics | Chemistry)
>
```

DTD for the Student DB

T0 A0 *nahuff*(604100418)
    T1 *nahuff*(Pankaj Tolani)
    T2 *nahuff*(2000)
    T3 *nahuff*(Master of Engineering)
    T4 A1 *enum*(Computer Science)

Abstract view of XGrind document

XGrind


XML Doc
XML–Parser
DTD Parser
DTD
Frequency Tables
Symbol Table
XGrind Kernel
Huffman–Compressor
Enum–Encoder
Compressed Internal Representation
. . .
XML–Gen
Semi–Structured Compressed XML Document

XGrind

- Requires 2 scans of XML data
  - Get statistics first: frequency and symbol tables
  - Encode the XML document

- Some queries such as range queries still require the partial decompression of compressed XML data
  - Huffman and dictionary encodings lose the relationships among values (loss of semantic)

$$v1>v2 \Rightarrow c1>c2$$

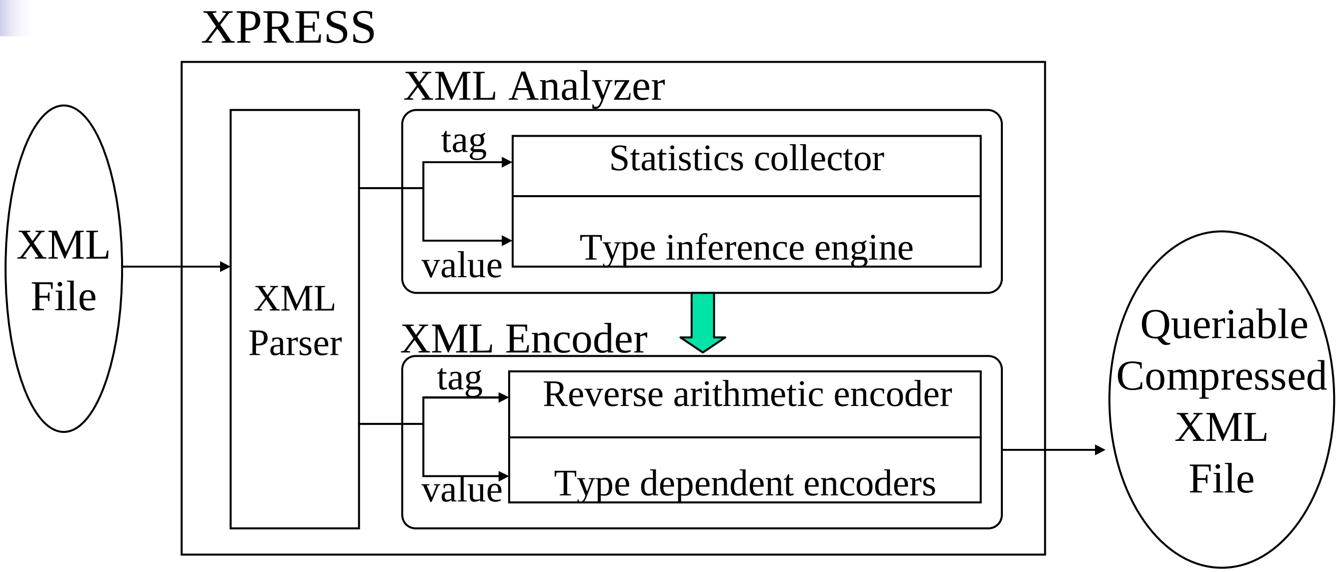
XPRESS

- Goal
  - Save disk space and network bandwidth
  - Support efficient and direct processing of queries to the compressed XML Data
- Homomorphic Compression
  - Preserves the structure of the original XML data in compressed XML data

XPRESS

- Semi-Adaptive : 2 scan
  - Statistics, required in the compression phase, are collected and fixed during the preliminary scan.
  - Adaptive
    - Preliminary scan is not required
    - The encoded value of a symbol is changed according to statistics => depending on the location
    - To evaluate queries, complete decompression is not required
  - 2 scan overhead is compensated by the frequent query evaluation

Architecture of XPRESS



- Semi-adaptive
  - First Scan → XML Analyzer : statistics gathering
  - Second Scan → XML Encoder : compression

XPRESS

- Reverse Arithmetic Encoding
  - Inspired by arithmetic encoding [Witten el. al, 1987]
  - XML query is based on path expressions
  - Existing XML Compressors transform a tag to an identifier
    - Query processor keeps the trace (i.e., path) of each element.
  - Encodes a label path of element *e* as an interval in [0,1)

Reverse Arithmetic Encoding

1. Partitions the entire interval

Element	Frq	Interval_T
book	0.1	[0.0, 0.1)
author	0.1	[0.1, 0.2)
title	0.1	[0.2, 0.3)
section	0.3	[0.3, 0.6)
subsection	0.3	[0.6, 0.9)
subtitle	0.1	[0.9, 1.0)

go Example

2. Encodes the simple path $P = p_1. \ldots .p_n$ of *e* into an interval $[min_e, max_e)$

Element	Path	Interval_T	interval
book	book	[0.0, 0.1)	[0.0, 0.1)
section	book.section	[0.3, 0.6)	[0.3, 0.33)
subsection	book.section.subsection	[0.6, 0.9)	[0.69, 0.699)

Reduce the Interval_{Pn} in proportion to the interval of $P' = p_1. \ldots .p_{n-1}$

For P = book.section.subsection, Interval_{Pn}=[0.6,0.9), P' = book.section

[0.6+0.3*0.3 = 0.69, 0.6+0.3*0.33=0.699)

Reverse Arithmetic Encoding

- The interval generated by reverse arithmetic encoding satisfies the following property:

Property1

If path P is represented as interval I, then all intervals for suffixes of P contain I.

EX)

The interval for book.section.subsection is [0.69,0.699)

The interval for section.subsection is [0.69, 0.78)

The interval for subsection is [0.6,0.9)

Therefore, $[0.6,0.9) \supseteq [0.69,0.78) \supseteq [0.69,0.699)$

- Based on Property1, the label path expression is efficiently evaluated.
- A Tag is replaced by the minimum value of the interval

XPRESS

- Automatic Type Inference Engine
  - Infers the types of data values of element e by simple inductive rules during preliminary scan phase

Rules)

Digits ➔ Integer

Digits with a dot ➔ Float

Strings whose number of distinct values is less than 128 ➔ Enumerated

General strings ➔ String

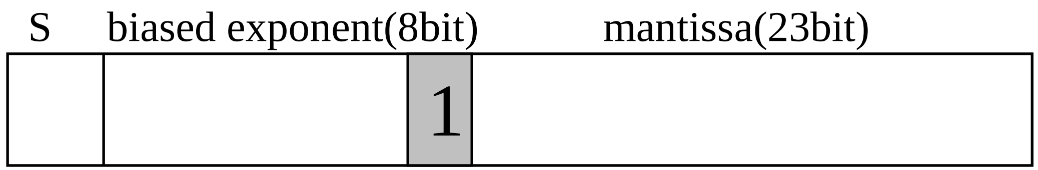
Implementation of XPRESS

Encoder	Description
u8	integer where max-min < 2^7
u16	integer where 2^7 <= max-min < 2^{15}
u32	integer where 2^{15}<= max-min < 2^{31}
f32	float
dict8	dictionary encoder for enumerated string
huff	huffman encoder for general string

- Encoders for Data Values
  - Numeric data (integer, float): Binary + Differential encoding
  - Enumerated String :Dictionary encoding
  - General String: Huffman encoding
    - (length[1byte], subsequence)+, where length of subsequence is less than 2^7=128 byte
  - MSB is always 0

Implementation of XPRESS

- Approximated Reverse Arithmetic Encoder for Tags



32bit floating point representation

  - Represents the interval of path p in [1.0, 2.0)
    - S = 0, E= 0111 111**1**
  - By cutting of the 1^{st} byte, MSB is always 1
    - Thus, the query processor can distinguish tag and data values
  - End tags are replaced by 0x80
  - To reduce the size, truncates the last byte

Drawback of XPRESS

- The XML data is flattened, even though it keeps the structure information.
- Thus, output of queries need to be generated from scratch
  - Consider a query

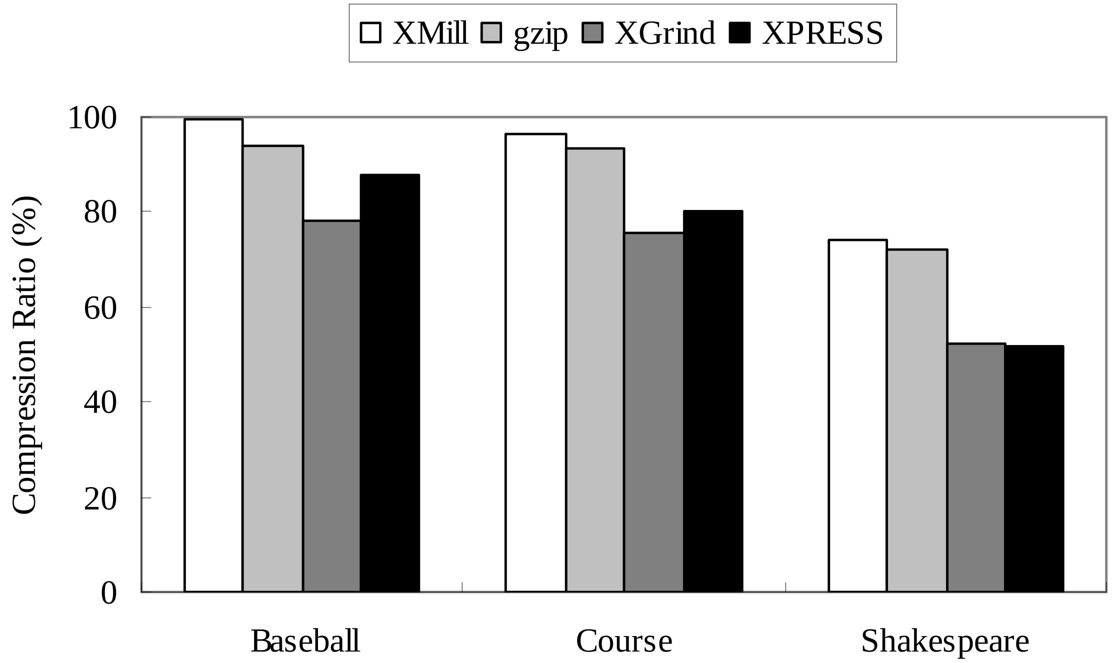
Experiments

- Machine
  - Sun Ultra Sparc II 168Mhz, 384Mbyte
- Data Set
  - Baseball : statistics of 1998 Major League
  - Course : courses held in U. of Washington
  - Shakespeare : plays of Shakespeare

Dataset	Size(Mbyte)	Depth	Tag	Numeric	Enum
Baseball	17.06	6	46	19	5
Course	12.28	6	18	5	4
Shakespeare	15.3	5	21	0	0

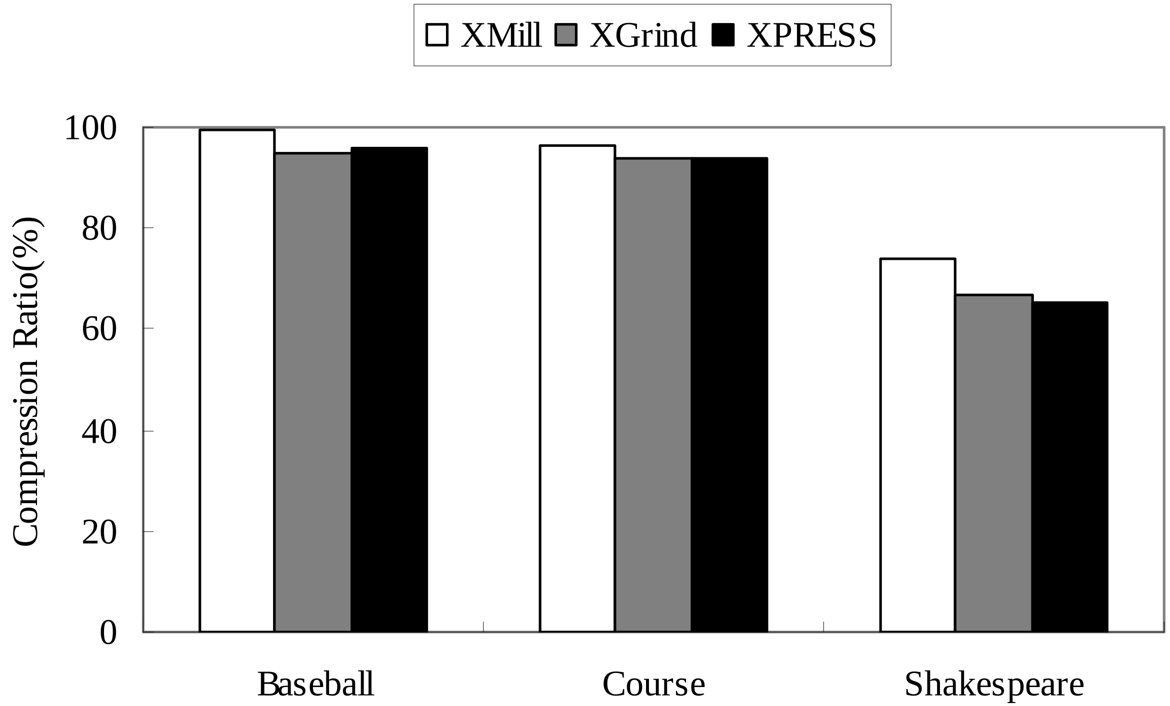
Experiments

- Compression Ratio



Experiments

- Effect of zlib

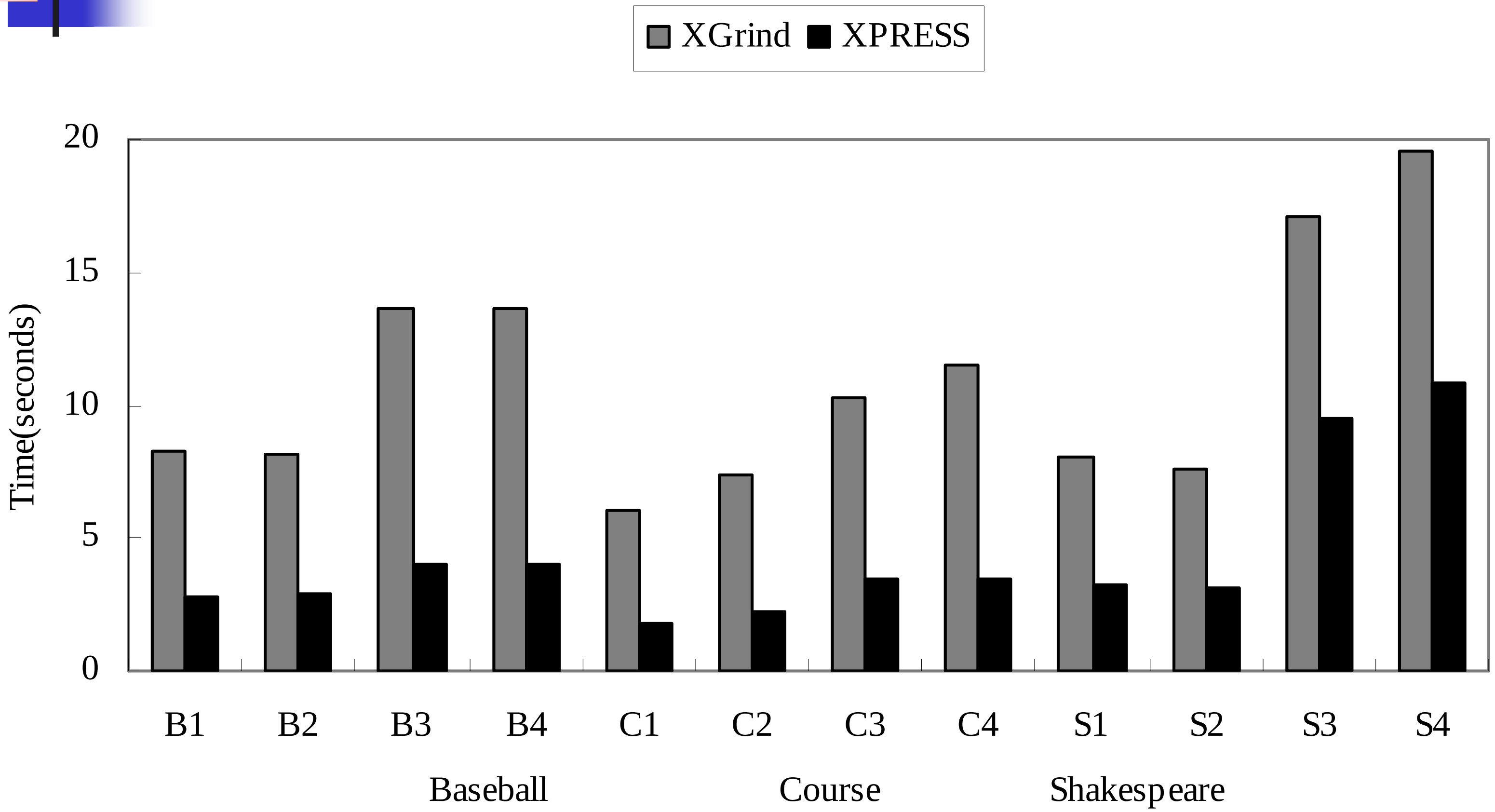


Experiments

- Queries
  - First character indicates the data set
  - Second digit denotes the query type
    - 1: simple path expression
    - 2: partial matching path expression
    - 3: complicated path expression
    - 4: path expression with range predicate

Query name	Query definition
B1	/SEASON/LEAGUE/DIVISION/TEAM/PLAYER/GIVEN_NAME
B2	//TEAM/PLAYER/SURNAME
B3	/SEASON/LEAGUE//TEAM/TEAM_CITY
B4	/SEASON/LEAGUE//TEAM[TEAM_CITY >= Chicago and TEAM_CITY <= Toronto]
C1	/root/course/selection/session/place/building
C2	//session/time
C3	/root/course//session/time/start_time
C4	/root/course//session/time[start_time >= 800 and start_time <= 1200]
S1	/PLAY/ACT/SCENE/SPEECH/STAGEDIR
S2	//PGROUP/PERSONA
S3	/PLAY/ACT//SPEECH/SPEAKER
S4	/PLAY/ACT//SPEECH[SPEAKER>= CLEOPATRA and SPEAKER <= PHILO]

Experiments



On the average, the query performance of XPRESS is 2.83 times better than that of XGrind.

Path Queries on Compressed XML

[Buneman, Grohe, Koch: VLDB'03]

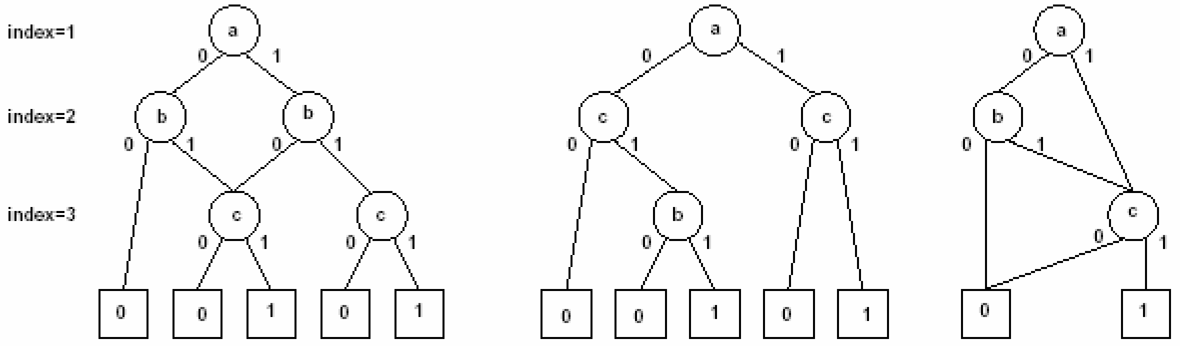
Motivation

- XML tree can be directly compressed using techniques from symbolic model checking
- Compression with sharing subtrees can be highly eff ective
- Compressed tree can be queried directly through a process of manipulating selections of nodes and partial decompression
- Storing compressed text separately from remaining tr ee skeleton in memory is not efficient for query processing on large documents
- Goal: Compress skeleton so that path queries are pos sible on this compressed skeleton.

Motivation

- Compression of XML tree skeletons by sub-tree sharing can be seen as a direct generalization of compression of Boolean functions into Ordered Binary Decision Diagrams(OBDDs).
- Thus, the efficient algorithms for OBDDs can be used for evaluating path queries directly on compressed skeletons.

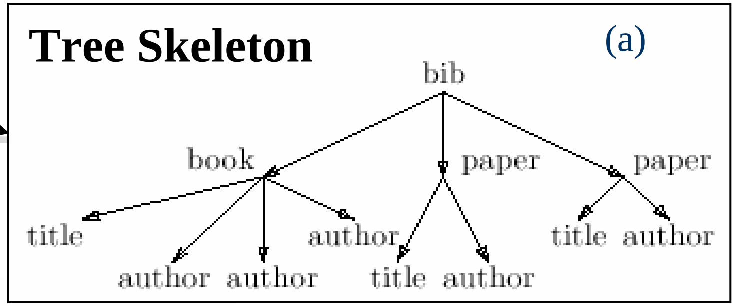
- **Binary Decision Diagram (BDD)**
  - tree or rooted DAG where each vertex denotes a binary decision
  - Example: $F = (a+b)c$

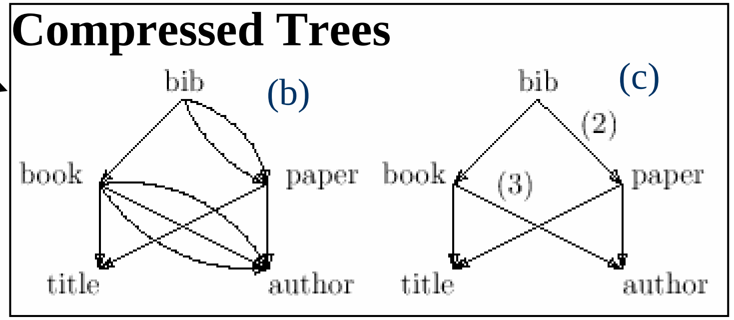


(a) (b) (c)

- **Implementation: each non-leaf vertex v has**
  - a pointer index(v) to a variable
  - two children low(v) and high(v)
- **Reduced Ordered Binary Decision Diagrams have no redundant subtrees:**
  - no vertex with low(v) = high(v)
  - no pair $\{u, v\}$ with isomorphic subgraphs rooted in u and v

Original Skeleton can be recovered by the appropriate depth-first traversal of the compresses skeleton





- <bib>
  - <book>
    - <title>**Foundati… …ases**</title>
    - <author>**Abiteboul**</author>
    - <author>**Hull**</author>
    - <author>**Vianu**</author>
  - </book>
  - <paper>
    - <title>**A Relational Model for Large Shared Data Banks**</title>
    - <author>**Codd**</author>
  - </paper>
  - <paper>
    - <title>**The Complexity of Relational Query Languages**</title>
    - <author>**Vardi**</author>
  - </paper>
- </bib>

Property of Compressed Representation by sub-tree sharing

- Compressed skeletons are easy to compute and allow us to query the data in a natural way.
- Each node in the compressed skeleton represents a set of nodes in the uncompressed tree.
- The purpose of a path query is to select a set of nodes in the uncompressed tree.
- However this set can be represented by a subset of the nodes in a partially decompressed skeleton

XML Storage

- Infeasible approaches
  - Subtree-based indexing/caching mechanisms
  - Relational DBMS use to keep node information in tuples
- Better: Separate string data and document structure
  - String data stored and indexed using conventional approaches
  - Document structure stored as a tree (“skeleton”) with nodes keeping element and attribute names
  - Used in XMILL as a compression scheme

XML Storage (cont'd)

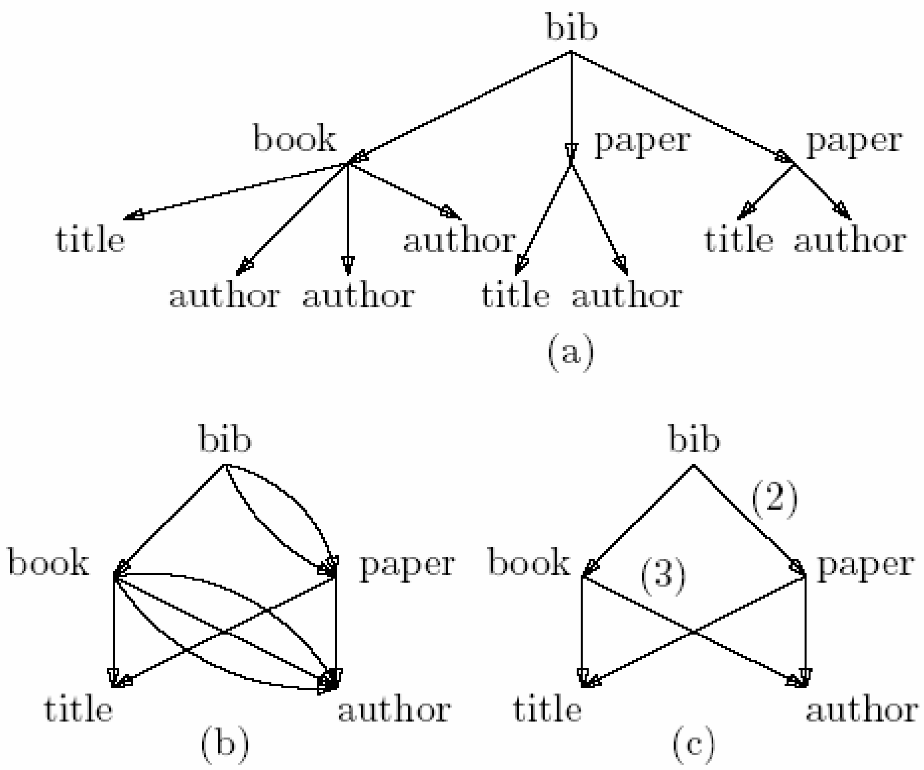
- String data needed for localized processing
  - Easily compressed by conventional methods
- Skeleton is needed for navigational aspect of queries
  - Usually small, can fit wholly in main memory
  - Can we compress for large skeletons?
  - How to avoid compression/decompression overhea d (time and space) during query evaluation?

Proposed Compression Scheme

- Based on sharing of common nodes
- Independent of DTD
- Generic structure, capable of expressing other information than just elements and attributes (eg. string match, query result)
- Original document structure is preserved
  - Bisimulation: Each node in compressed tree corresponds to a number of nodes in uncompressed tree
  - Partial decompression is possible
  - Naturally extends to query processing on compressed skeletons

Example

```
- <bib>
  - <book>
      <title>Foundations of Databases</title>
      <author>Abiteboul</author>
      <author>Hull</author>
      <author>Vianu</author>
    </book>
  - <paper>
      <title>A Relational Model for Large Shared Data Banks</title>
      <author>Codd</author>
    </paper>
  - <paper>
      <title>The Complexity of Relational Query Languages</title>
      <author>Vardi</author>
    </paper>
  </bib>
```



- Common subtrees are shared
- Edges are ordered
- In (c) consecutive multiple edges are joined and marked their multi plicity

Query Processing

- How to evaluate a XPath query on a compressed instance?
- A subset of XPath Query is defined and discussed how to process a Core Xpath Query?

Location Path is composed of Location Steps:

- A location step has three parts:
  - ✓ An axis, which specifies the tree relationship between the nodes selected by the location step and the context node.
  - ✓ A node test, which specifies the node type.
  - ✓ Zero or more predicates, which use arbitrary expressions to further refine the set of nodes selected by the location step.

```
Child :: chapter [child::title='Introduction']
```

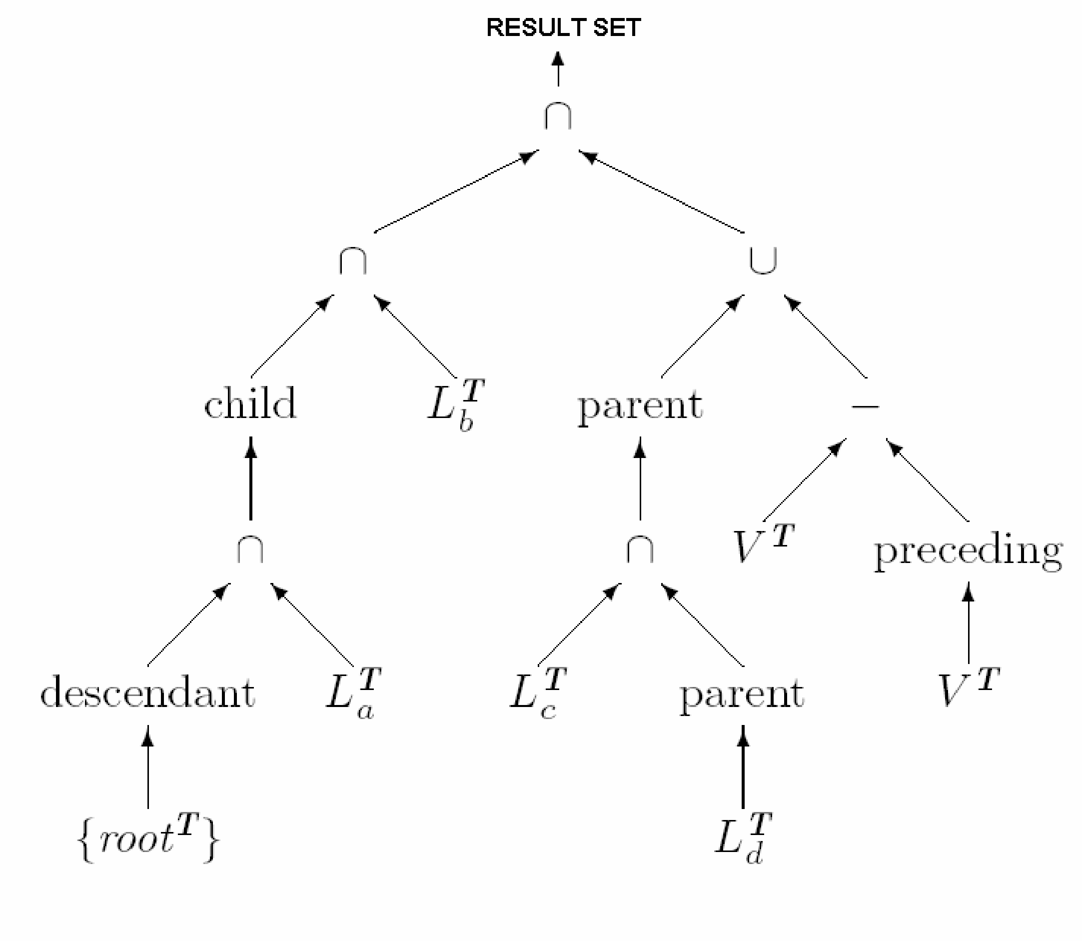
Axis **Node Test** **Predicate**

Selects the chapter children of the context node that have one or more title children with string-value equal to Introduction.

An Example Core XPATH Query

/ descendant::a / child::b[child::c / child::d or not(following::*)]

The intuition in Core XPath, which **reduces** the **query evaluation problem** to **manipulating sets rather than binary relations**, is to **reverse paths** in conditions to **direct the computation** of node sets in the query **towards the root of the query tree**.

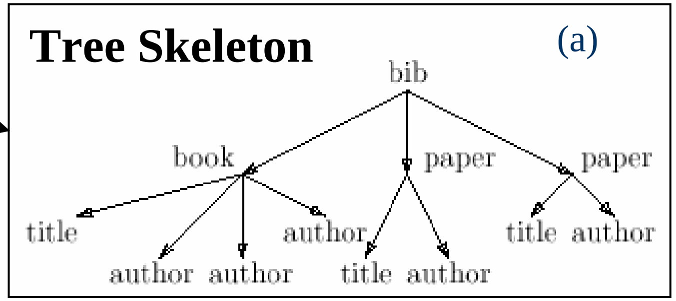


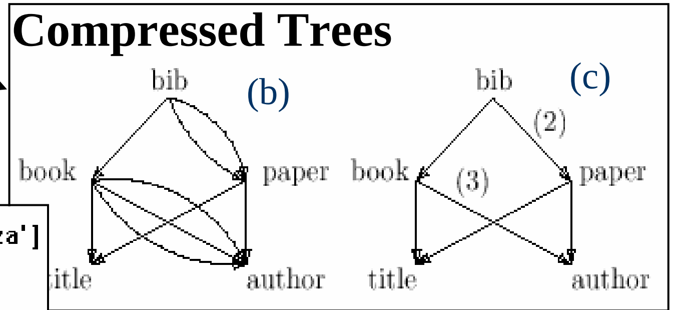
Operations On Compressed Instances

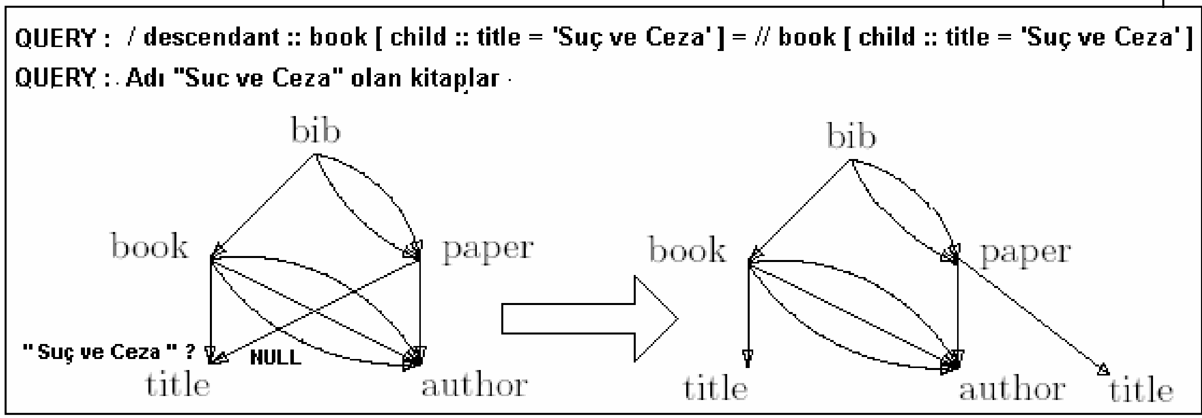
- Goal is to avoid full de-compression when it is not necessary.
- The idea is to traverse the DAG of the input instance starting from the root, visiting each node v only once.
- We choose a new selection of v on the basis of the selection of its ancestors, and split v if different predecessors of v require different selections. We remember which node we have copied to avoid doing it repeatedly.

Operations On Compressed Instances

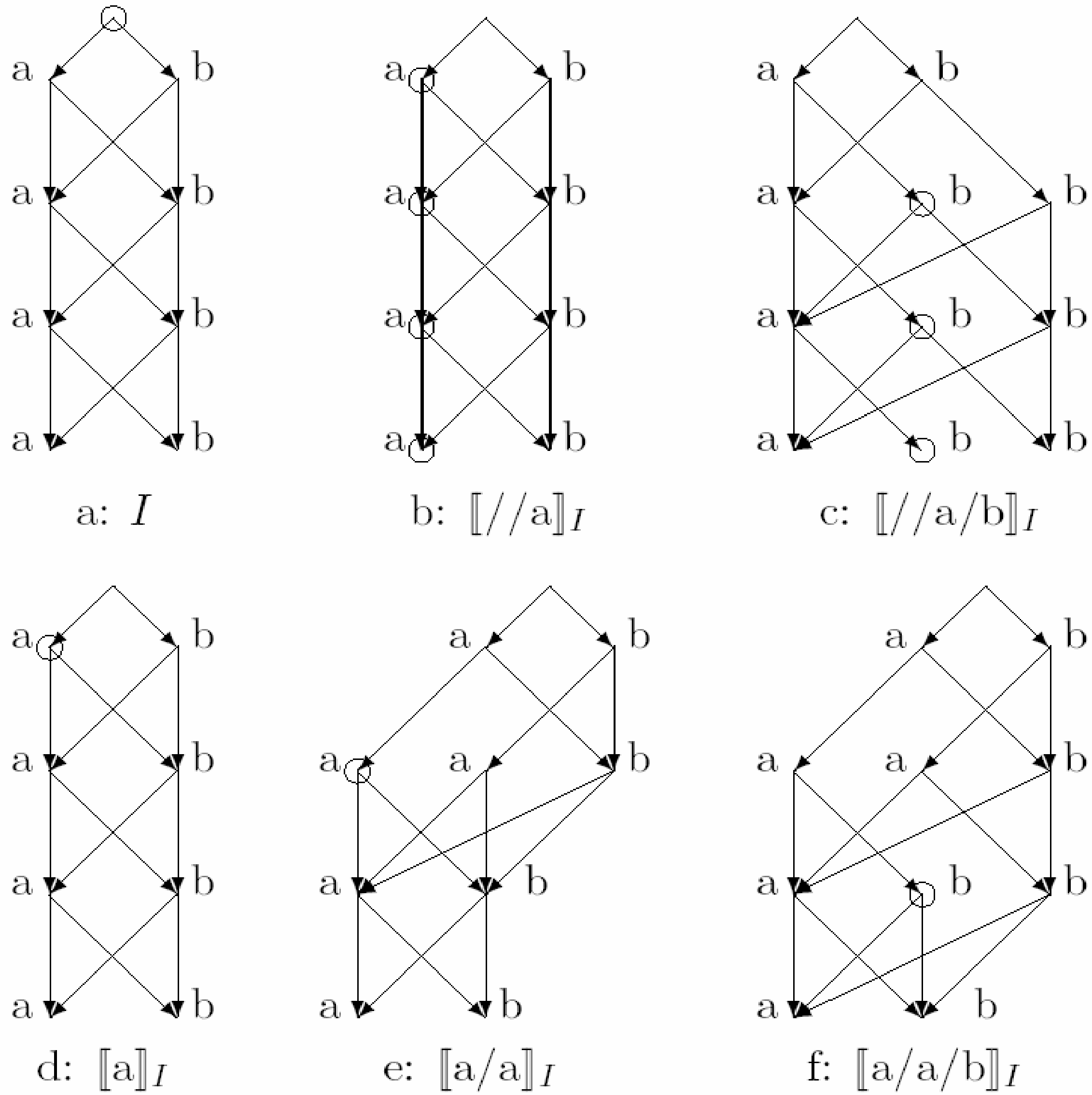
Original Skeleton can be recovered by the appropriate depth-first traversal of the compresses skeleton







EXAMPLE :



a: I b: $[\![//a]\!]_I$ c: $[\![//a/b]\!]_I$

d: $[\![a]\!]_I$ e: $[\![a/a]\!]_I$ f: $[\![a/a/b]\!]_I$

Complexity of Decomposition

- Unfortunately, compressed trees may be decompressed exponentially in the worst case even on very simple queries.
- However, the decompression is only exponential in the size of the queries (but not the data), which tend to be small.

Questions & Comments