Grouping of Semistructured Data for Efficient Query Processing

Mathias Neumüller

A thesis presented for the degree of
Doctor of Philosophy

2004

Department of Computer and Information Sciences,
University of Strathclyde in Glasgow

DECLARATION OF AUTHOR'S RIGHTS

The copyright of this thesis belongs to the author under the terms of the United Kingdom Copyright Acts as qualified by University of Strathclyde Regulation 3.51. Due acknowledgement must always be made of the use of any material contained in, or derived from, this thesis.

ABSTRACT

With the emergence of large-scale distributed computing applications semistructured data models have gained significant importance. Current practical semistructured data management systems can often not provide the performance required by practical applications.

This work describes a model for the optimisation of semistructured data processing based on data groupings. Such groupings are of fundamental importance for efficient querying of semistructured data. The semistructured model does not imply the natural organisation of data that characterises rigidly structured representations. Instead, data groupings in the semistructured case must be derived from the data itself or its applications.

This thesis presents a number of such possible data groupings and formalises them into a concept of domains. Different classes of domains are identified and the impact on different data sources is evaluated. A particular definition is then used to implement an efficient physical representation using an approach based on dictionary compression adapted from relational data management. Finally this approach is combined with a data grouping aimed at the efficient resolution of structural constraints.

ACKNOWLEDGEMENTS

Dedication

To my late father

Firstly, I would like to acknowledge the continuous support and trust I have received from my supervisor Mr John N. Wilson. He has encouraged me to take up research in the field of databases and made this PhD possible. His never-ending enthusiasm for this work has secured its successful conclusion. I am also grateful for the generous departmental stipend and the numerous financial contributions to conference, workshop and summer school visits.

I am indebted to many colleagues for their advise and support. The SNAQue team and I shared a lot of interest in the area of semistructured data processing and their comments on my work were often helpful. I particularly enjoyed the cooperation with George Russell on TypEx. Thanks to the EFoCS Software Engineering group I was able to get to know many and use a few aspects of modern software development.

The departmental Ultimate Frisbee team helped me to find a balance between theoretical work and physical exercise. Most of the team members have become good friends. I was also driven into the arms of both the Strathclyde Mountaineering and Canoe Club, for which I am very thankful. Apart from taking my out into the Scottish wilderness I have found many friends there. Their continuous lust for adventure has helped me through many hard phases of this journey.

Most importantly I need to recognise the role of my friends, both here in Glasgow and beyond. Matt Munro has accompanied me for the entirety of my time in the department, as fellow student, Frisbee player and mountaineer. He was a true friend throughout this time and I hope our friendship will last much longer. Nahoum and Stephanie are also exceptionally good friends, whose support has often helped me through hard times and who could offer me a continental haven in Glasgow. Oliver has maintained contact with me during my five year absence from Germany and welcomed me whenever I went there. I also appreciate the three winters I spent with friends from the University of Hannover, two times in Norway and once in Slovenia, and the people that came to visit me in Scotland. Thank you all!

Last but not least I would like to thank my mother for her support during my long stay abroad. I am aware of the hardship she had to endure with both her children so far away in difficult times.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Code and Data

"Show me your [code] and conceal your [data structures],
and I shall continue to be mystified.
Show me your [data structures],
and I won't usually need your [code]; it'll be obvious."[1]

Fred Brooks, *The Mythical Man Month*, 1975

This thesis analyses the problem of efficient processing of semistructured data. In order to allow this a fundamental concept is introduced and detailed throughout this work. The concept discussed is that of data grouping or, more technically, that of graph clustering. The focus of this work lies on finding suitable data organisations that support efficient data management. Both theoretical and practical aspects of data groupings are developed in order to show that this concept is indeed fundamental to understanding and improving semistructured querying processes.

Querying semistructured data, i.e. embedding sub-graphs potentially encoding regular expressions into larger data graphs, is a well known problem. Algorithms for this task were developed in the early stages of computer science by the graph theory community. This thesis explores how such algorithms can be adapted to the requirements of today's environment.

1.1 The Environment for Semistructured Data Processing

Over the last few years the environment in which semistructured data models are used has changed significantly. It took decades to evolve from a theoretical problem to its widespread application in science and industry, primarily driven by the emergence of web technologies. Data sets are growing with exponential

[1] Actually, he said "flowcharts" and "tables". But allowing for almost thirty years of terminological and cultural shift, it is almost the same point.

speed and cannot be dealt with using conventional approaches. Globally valid schemata are often impossible to impose in situations where semantically similar but syntactically different data sources need to be integrated and processed across the boundaries of single organisations. In order to allow the automated processing of such heterogeneous sources, the XML standard was developed as a concrete syntax for the representation of semistructured data sources.

Now mechanisms that can deal with the amounts of data occurring in practice are needed. Many of the algorithms discussed in the past by graph theoreticians do not scale to the sizes of today's practical problems. Simpler approaches are needed that can deal with a useful subset of this wide area with an acceptable performance. The focus of this work rests on efficient querying of such large sets of semistructured data. This is motivated by a new type of application, especially significant in the area of data-centric sciences, such as bioinformatics or particle research, where data is gathered automatically. In such environments, the queries or computations do not have the properties that are generally assumed for unrestricted, ad hoc online processes. In addition, the data does not change significantly in a period of time which is comparable to the time needed for its processing, i.e. it can be considered to be (semi-)static.

1.2 Structure and Contribution of the Thesis

The main contribution of this work is an investigation of the importance of data groupings to the semistructured querying process. On a theoretical level a novel model is proposed in Chapter 3 that conceptualises the general query optimisation process for semistructured data. Data groupings are central to the presented model. As a consequence it motivates a re-evaluation of the concept of domains for the semistructured data model, which is discussed in Chapter 4. On a more practical level Chapter 5 analyses how the concept of data grouping influences data management using the example of dictionary compression. This approach, which was previously applied in relational database compression, can be re-established for data-centric semistructured databases if a suitable grouping of the data is effected first, i.e. if domains can be established in semistructured data. The final chapter of this thesis investigates how the restructuring of semistructured data influences query execution performance. For this purpose a combination of existing indexing methods is proposed to allow the efficient querying of semistructured data using both structural and value constraints (Chapter 6). So far research has

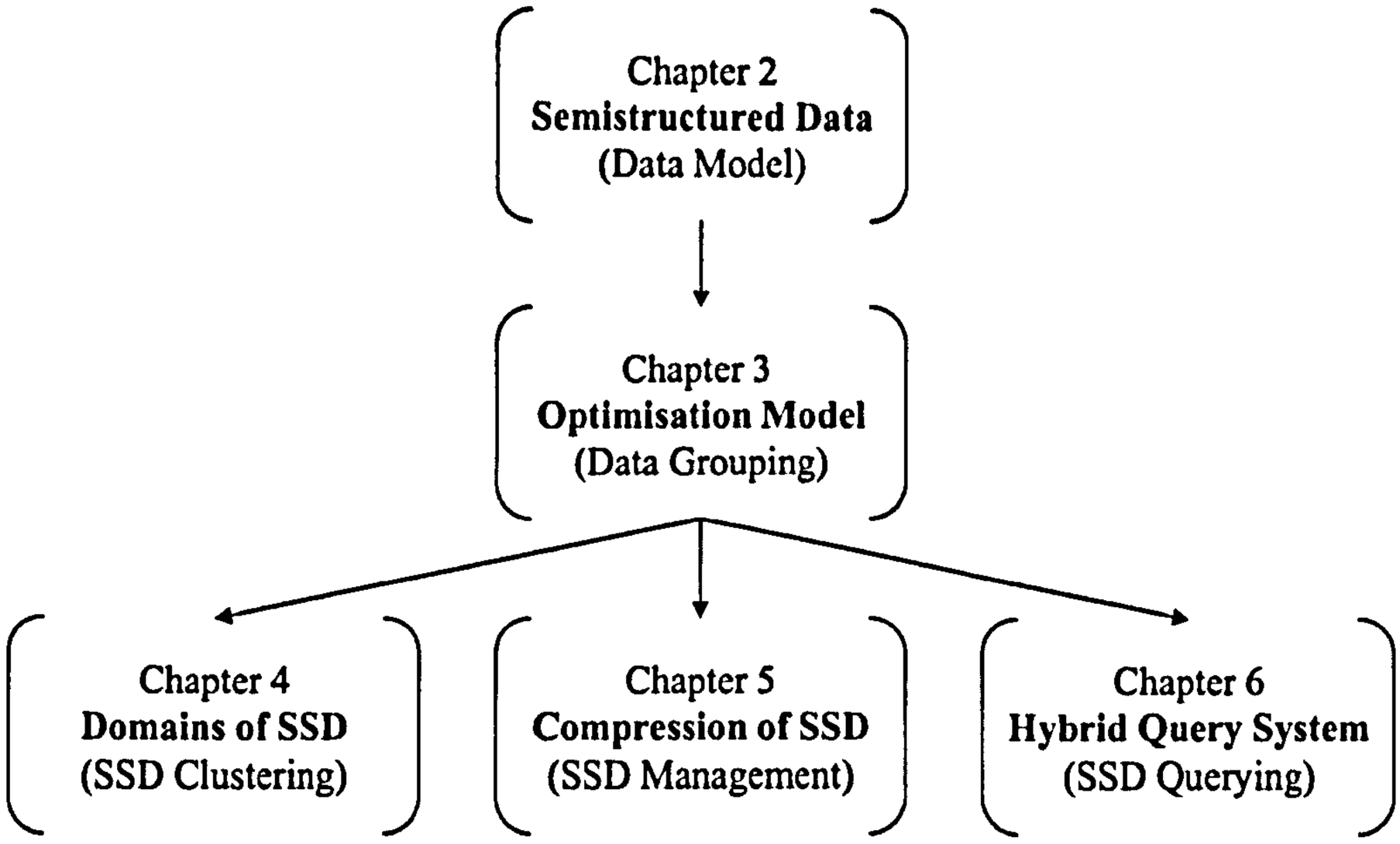


Fig. 1.1: The structure of the main argument of the thesis

concentrated on one or the other of these constraints, which typically leads to performance problems on the part ignored.

Taken as a whole, Chapters 4 – 6 validate the optimisation model presented in Chapter 3 and thus also support the importance of data groupings in general. Individually they clearly show that there exists a compromise between complexity of query languages together with the mechanisms required to support them and the performance of such systems. Figure 1.1 visualises the structure of the core chapters of this thesis. The initial step in this process is to explain the concept of semistructured data and define the terminology used in the rest of the thesis.

2. SEMISTRUCTURED DATA

Semistructured Data

"...data that is neither raw data nor strictly typed."

Serge Abiteboul, *ICDT 1997*, Delphi, Greece

This chapter introduces fundamental concepts and terminology of semistructured data models and management systems, which are essential for the understanding of the following work. It starts by giving a historical motivation for the current interest in semistructured data, then develops a data model based on elements from graph theory. Query languages, query mechanisms and associated indexing techniques are discussed before finally other research in the area of semistructured data processing is reviewed.

2.1 Historical Motivation for Semistructured Data

The term semistructured data (SSD) describes a number of data models with varying definitions. One approach is to identify SSD with originally unstructured information, such as textual documents, which are annotated with describing metadata in order to ease automatic processing. It is thus a subset of unstructured data. However, the term is more frequently used to distinguish SSD models from structured ones, particularly the relational model. However, from a theoretical point of view, SSD is a superset of structured data, i.e. all structured data sources can be managed by a semistructured DBMS but not vice versa. Figure 2.1 depicts these relationships. For the purpose of this thesis SSD will be defined as follows:

Definition 2.1 (Semistructured Data): Semistructured data comprises those documents that combine some atomic pieces of data with some meaningful structural relationships between these atoms. The atomic data and its structure (or metadata) are combined in a single document and thus inseparable from a processing point of view.

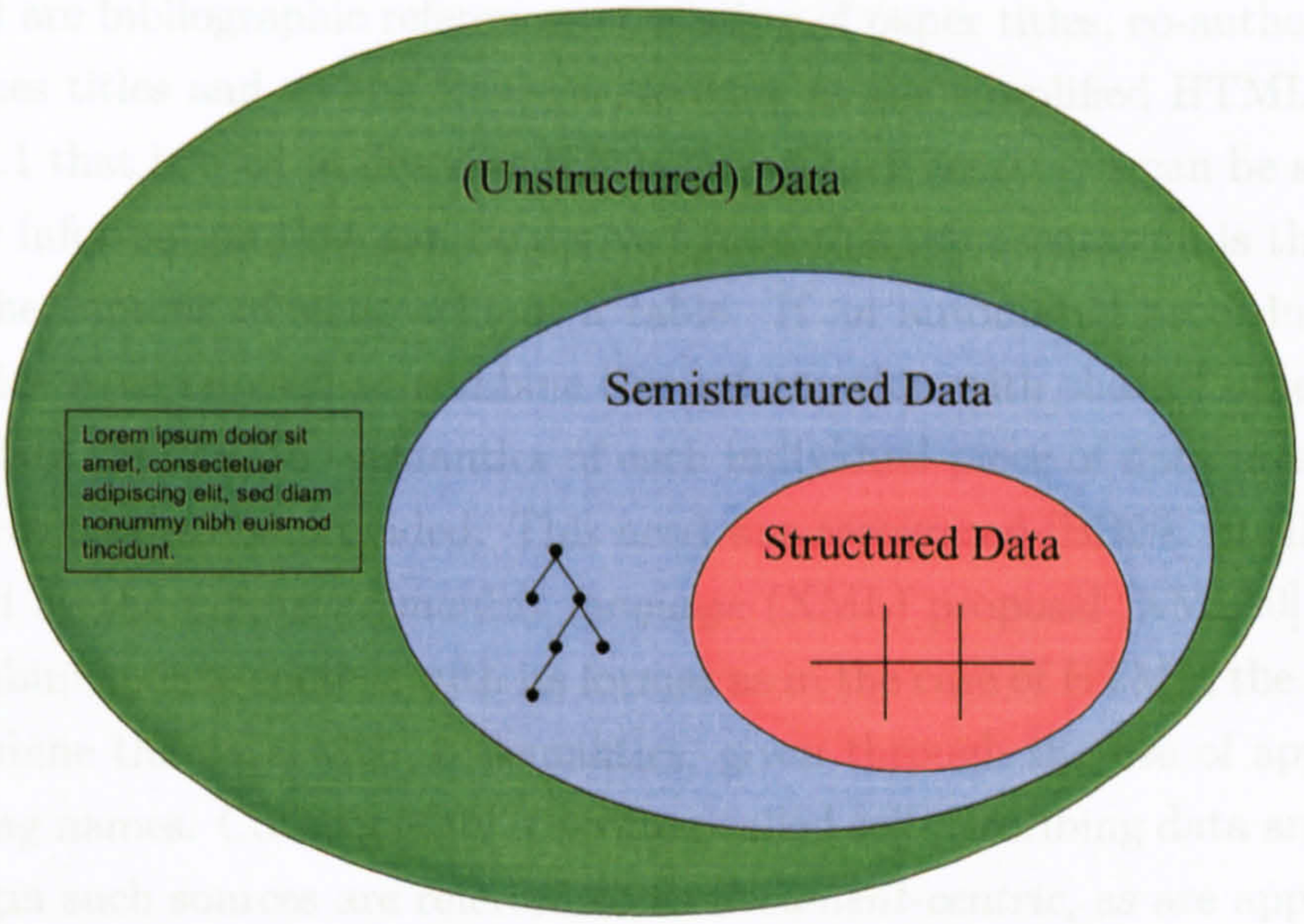


Fig. 2.1: The realms of unstructured, semistructured and structured data

The semantics of semistructured data depend on both the atomic data and their structural relationships. Semistructured data does not require an *a priori* schema, but can rather present its own schema as part of the data. The reasons for the current interest in SSD are introduced in the following sections before in Section 2.2 a sound theoretical definitions of a model for SSD is presented.

2.1.1 From Unstructured Data to Semistructured Data

Though SSD models have existed for some time they began to attract widespread attention in the wake of the more recent globalisation of computing achieved by web technologies [ABS00]. The Internet's most successful service, the World Wide Web (WWW) [BLCG92], is based on the hypertext markup language HTML [HTM99]. Documents in this format are essentially unstructured in terms of this thesis. Although they follow certain encoding rules, which imply a structure similar to that of SSD, this structure is unrelated to the information they present. This is due to the fact that they are meant for human consumption, with their associated metadata solely used to describe their presentation. A human reader must derive the semantics of the data from contextual information and background knowledge. A scientist looking at the author's DBLP (Digital Bibliography & Library Project) page shown in Figure 2.2 can derive that the information

displayed are bibliographic references consisting of paper titles, co-author names, conferences titles and so on. However, looking at the simplified HTML code in Listing 2.1 that is used to describe this page, no such semantics can be attached. The only information that can be derived from this representation is that it describes the content of some cells of a table. If an automated procedure like a web service is to be used to combine this information with that of other bibliographic data sources, the semantics of each individual piece of data presented in each source need to be provided. This need was recognised [BB99, BLHL01] and addressed by the *extensible markup language* (XML) proposal [XML00]. Rather than combining data content with its format as in the case of HTML, the aim here is to combine the data with its semantics, given through the use of application specific tag names. Consequently it is often called self-describing data and due to their origin such sources are referred to as *document-centric*, as are applications working on them.

This complies with Definition 2.1 for SSD, of which XML documents are an example. Listing 2.2 shows an example of a single entry from the page shown in Figure 2.2 in this format. The set of data sources provided in such a format and made available over the Internet is commonly referred to as the *semantic web* [BLHL01].

2.1.2 From Structured Data to Semistructured Data

Equally contributing to the interest in SSD models is the move to expose to the web information previously managed in a closed DBMS. Such data usually adheres to a strict schema, e.g. information stored in relational databases. Although the process of publication does not change the structured nature of the data itself, the open environment in which it is exchanged requires some compromise as far as its encoding is concerned. The drift from closed, centralised, client/server environments, in which schema constraints are known and can be enforced, to open, distributed, peer-to-peer systems favours the laxer constraints imposed by SSD models. Data sources and their applications that originate from such a scenario are referred to as *data-centric*.

Even within the domain of strictly controlled database management systems there exist reasons to drop some of the constraints imposed by the rigorously structured relational data model. The decomposition of data into functional dependencies, also known as normalisation [Cod70], aids efficient processing. How-

dblp.uni-trier.de

Mathias Neumüller

List of publications from the DBLP Bibliography Server - FAQ

Coauthor Index - Ask others: ACM DL - ACM Guide - CiteSeer - CSB - Google

2003		
3	EE	George Russell, Mathias Neumüller, Richard C. H. Connor: TypEx: A Type Based Approach to XML Stream Querying. WebDB 2003: 55-60
2002		
2		Mathias Neumüller: Compact Data Structures for Querying XML. EDBT PhD Workshop 2002: 127-130
1	EE	Mathias Neumüller, John N. Wilson: Improving XML Processing Using Adapted Data Structures. Web, Web-Services, and Database Systems 2002: 206-220

Coauthor Index

1	Richard C. H. Connor	[3]
2	George Russell	[3]
3	John N. Wilson	[1]

DBLP: [Home | Search: Author, Title | Conferences | Journals]
Michael Ley (ley@uni-trier.de) Mon May 10 16:57:27 2004

Fig. 2.2: The DBLP page with the author's bibliographic information

```
...

<table border=1>

<tr>
<th colspan=3>2003</th>
</tr>

<tr>
<td><a name="p3">3</a></td>
<td><a ref="http://www.cse.ogi.edu/webdb03/papers/10.pdf">EE</a></td>
<td>George Russell, Mathias Neum&uuml;ller, Richard C. H. Connor:
TypEx: A Type Based Approach to XML Stream Querying.
<a href="...">WebDB 2003</a>: 55-60</td>
</tr>

<tr>
<th colspan=3>2002</th>
</tr>

<tr>
<td><a name="p2">2</a></td>
<td> </td>
<td>Mathias Neum&uuml;ller:
Compact Data Structures for Querying XML.
<a href="...">EDBT PhD Workshop 2002</a>: 127-130</td>
</tr>

...

</table>
```

Listing 2.1: An excerpt of the HTML code describing the page shown in Figure 2.2, stripped of colour and alignment information

```
<inproceedings mdate="2003-06-23" key="conf/webdb/RussellNC03">
<author>George Russell</author>
<author>Mathias Neum&uuml;ller</author>
<author>Richard C. H. Connor</author>
<title>TypEx: A Type Based Approach to XML Stream Querying.</title>
<pages>55-60</pages>
<year>2003</year>
<crossref>conf/webdb/2003</crossref>
<booktitle>WebDB</booktitle>
<ee>http://www.cse.ogi.edu/webdb03/papers/10.pdf</ee>
<url>db/conf/webdb/webdb2003.html#RussellNC03</url>
</inproceedings>
```

Listing 2.2: The semistructured XML encoding of the first publication entry shown in Figure 2.2

PAPER				
ID	TITLE	YEAR	WORKSHOP	PAGES
p1	Improving XML Processing Using Adapted Data Structures	2002	Web, Web-Services, and Database Systems	206–220
p2	Compact Data Structures for Querying XML	2002	EDBT PhD Workshop	127–130
p3	TypEx: A Type Based Approach to XML Stream Querying	2003	WebDB	55–60

COAUTHOR	
PAPER	AUTHOR
p1	a2
p1	a4
p2	a2
p3	a1
p3	a2
p3	a3

AUTHOR	
ID	NAME
a1	Richard C. H. Connor
a2	Mathias Neumüller
a3	George Russell
a4	John N. Wilson

Fig. 2.3: The author's bibliographic information in a relational schema

ever it also breaks up data into small syntactically homogeneous pieces, a process that hinders comprehension of the information as a whole. Figure 2.3 shows a possible relational presentation of the information shown in Figure 2.2. It shows how the first normal form, which forbids set-valued attributes, often detracts from representational simplicity. In the DBLP example provided, it seems to be unnatural to factor out the names of the authors into a separate relation as done in Figure 2.3, rather than keeping them with the paper information. Consequently research has been performed to overcome this limitation resulting in *non first normal form* (NFNF) systems [AB84]. SSD models aim to overcome the same limitation by keeping together semantically related data. Structured data, by contrast, is focused on grouping syntactically homogeneous data.

Finally schema evolution provides a further reason why an SSD format might be used to represent data that obeys a regular structure. Even if the structure of a given dataset is known at any given point in time, the source's properties might change as its applications develop. Technical progress in the life sciences has produced automated measurement equipment that captures more and more experimental data. In addition, manual annotations on automatically captured

data can complicate its structure. At the same time, previously captured data remains of interest and is used in combination with more recent, potentially differently structured data from the same domain. Applications preceding a revised schema only have partial knowledge of it, but may still be executed successfully if they only depend on parts of the data that remained unchanged. Schema evolution is one of the remaining interesting research areas in the world of relational databases [RB01], for which it presents a sizable problem. SSD however adapts gracefully to such changing requirements.

2.2 Semistructured Data Model

This section introduces the terminology that will be used throughout the rest of the thesis. It follows standard definitions from graph theory but extends them to define a more specific form of a graph, called *data graph* that will be used as the formal data model in the remaining chapters.

2.2.1 Graph Theoretical Background

The following section, which is based on general graph theory [Wil75], defines terms in the context of simple graphs and directed graphs. Although data graphs, introduced in Section 2.2.2 are strictly speaking different entities from the graphs presented here, they are similar enough to apply the same terminology.

2.2.1.1 Graphs and Digraphs

Conceptually SSD is typically represented as a graph, where vertices are used to represent pieces of information and arcs are used to represent their structural relationships. In fact Buneman states that graph models are "the unifying idea in semistructured data."[1].

Definition 2.2 (Simple Graph): A *simple graph* G is a pair $(V(G), A(G))$, where $V(G)$ is a finite, non empty set of elements called *vertices* and $A(G)$ is a finite set of unordered pairs of distinct elements of $V(G)$ called *arcs*. An arc $\{u, v\} \in A(G)$ is said to *join* the vertices u and v.

Definition 2.3 (Simple Digraph): A *simple directed graph* or *simple digraph* D is a pair $(V(D), A(D))$, where $V(D)$ is a finite, non empty set of elements called

[1] Peter Buneman, PODS 1997, Tucson, Arizona, USA

vertices and $A(D)$ is a finite set of ordered pairs of distinct elements of V called *arcs*. An arc $(u, v) \in A(D)$ is called an arc from u to v.

Figure 2.4(a) and 2.4(b) show examples of a simple graph and a simple digraph over the set of vertices $V(G) = V(D) = \{u, v, w, x, y, z\}$. The undirected graph G contains the arcs $A(G) = \{\{u, v\}, \{u, x\}, \{v, x\}, \{w, z\}, \{x, y\}\}$ and the directed graph D contains the arcs $A(D) = \{(u, v), (u, x), (v, x), (w, z), (x, y)\}$. G is called the *underlying graph* of D, i.e. the graph that is created by removing the direction of the arcs. A subgraph S of a graph G is a graph, whose set of vertices $V(S)$ is a subset of $V(G)$ and whose set of arcs $A(S)$ is a subset of $A(G)$.

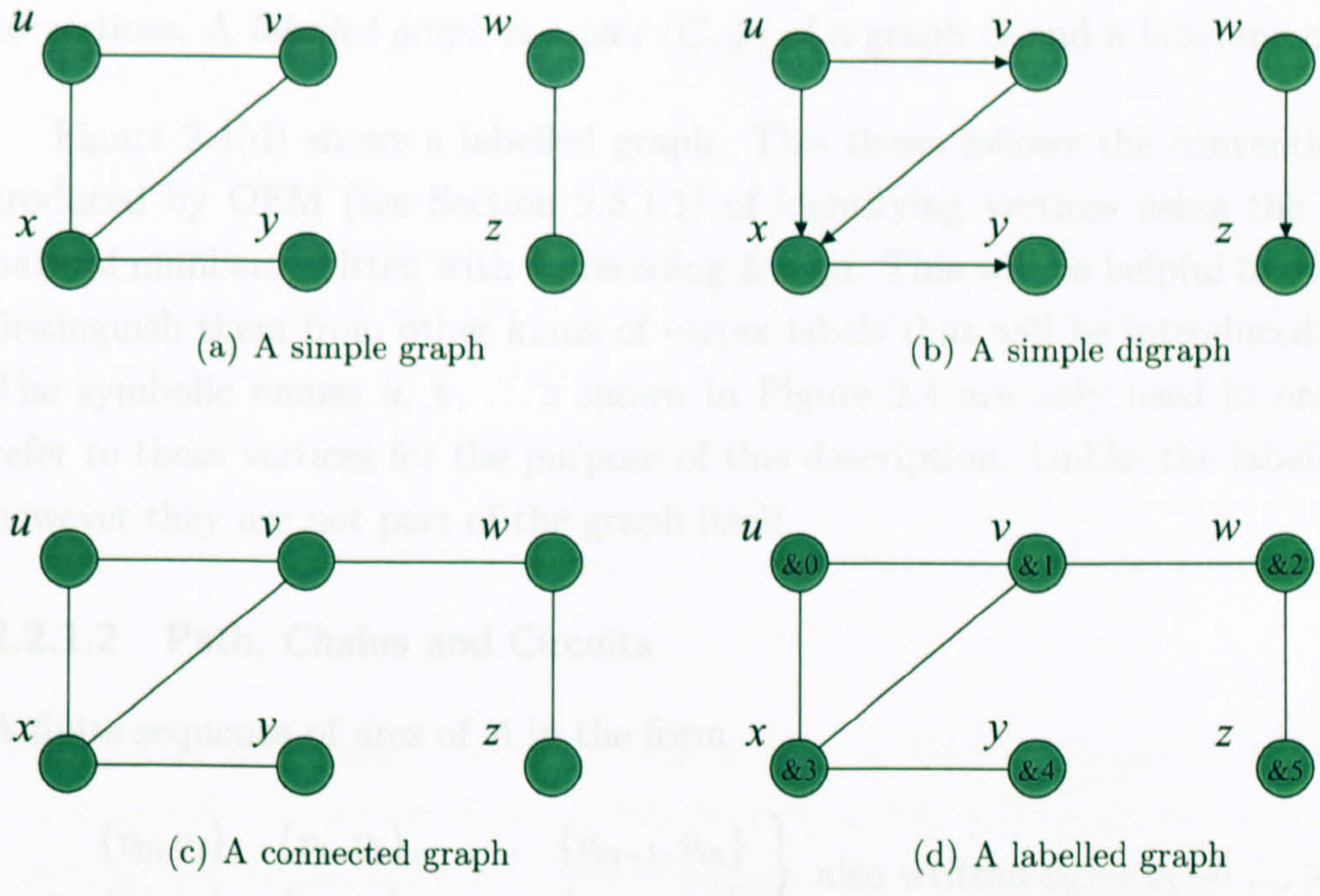
These definitions are based on sets rather than families of arcs and thus exclude multiple (equally directed in the case of the digraph) arcs between two vertices. Equally the condition that the two vertices of an arc need to be distinct prevents the existence of *loops*, i.e. arcs connecting vertices to themselves. For convenience the $V(G)$ or $V(D)$ will be referred to as V and $A(G)$ or $A(D)$ as A whenever the graph G or digraph D is implied by the context and called *vertex-set* and *arc-set* respectively.

Note on terminology: The term "arc" is deliberately favoured here over the term "edge" more commonly encountered in other literature on graph theory. This will be used to aid the distinction between general graph arcs and the more specific edges of a tree defined later in this section.

Definition 2.4 (Connected Graph): A graph G is *connected* if it cannot be expressed as the union of two disjoint graphs, i.e. the union of their vertex- and arc-sets respectively. This means that there exist no two non-empty subgraphs of a connected graph G that have no vertex in common.

The graph shown in Figure 2.4(a) is not connected as it can be generated by the union of the disjoint graphs $G_1 = (\{u, v, x, y\}, \{\{u, v\}, \{u, x\}, \{v, x\}, \{x, y\}\})$ and $G_2 = (\{w, z\}, \{\{w, z\}\})$. However, the graph shown in Figure 2.4(c) is connected. For the remainder of this thesis all graphs considered will be connected. For *disconnected graphs*, i.e. graphs with more than one *connected components*, any statement derived for connected graphs is true for the disjoint subgraphs given by the connected components.

For some applications it will be beneficial to identify a vertex of a given graph uniquely. To this end a bijective, i.e. one-to-one, mapping from vertices to a set of unique identifiers is defined here.



(a) A simple graph

(b) A simple digraph

(c) A connected graph

(d) A labelled graph

(e) A directed tree

(f) A rooted, directed tree

(g) A tag labelled tree

(h) A data graph

Fig. 2.4: Illustration of different types of graphs

Definition 2.5 (Graph Labelling): A *graph labelling* φ of a graph G is a bijective mapping from the vertex-set V to a set of identifiers Φ. The mapping function $\varphi : V \rightarrow \Phi$ maps vertices to identifiers, its inverse $\varphi^{-1} : \Phi \rightarrow V$ maps identifiers to vertices. A *labelled graph* is a pair (G, φ) of a graph G and a labelling of G.

Figure 2.4(d) shows a labelled graph. This thesis follows the convention introduced by OEM (see Section 2.5.1.1) of identifying vertices using the set of natural numbers written with a preceding &-sign. This will be helpful in order to distinguish them from other kinds of vertex labels that will be introduced later. The symbolic names u, v, $\ldots z$ shown in Figure 2.4 are only used in order to refer to these vertices for the purpose of this description. Unlike the labelling φ however they are not part of the graph itself.

2.2.1.2 Path, Chains and Circuits

A finite sequence of arcs of A in the form

$$\left.\begin{array}{llllll} & \{v_0, v_1\}, & \{v_1, v_2\}, & \ldots, & \{v_{m-1}, v_m\} \\ \text{or} & (v_0, v_1), & (v_1, v_2), & \ldots, & (v_{m-1}, v_m) \end{array}\right\} \text{ also written } v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_m$$

is called an *arc-sequence* from the *initial vertex* v_0 to the *final vertex* v_m in G. The number m of arcs in an arc-sequence is called its *length*. A *path* is an arc-sequence in which all arcs are distinct. If in addition all vertices are distinct, except possibly $v_0 = v_m$, it is called a *chain*. A path or chain with $v_0 = v_m$ is *closed*, and a closed chain is called a *circuit*.

The arc-sequence $x \rightarrow y \rightarrow x$ in Figure 2.4(a) has length 2 but does not form a path, because the arcs $\{x, y\}$ and $\{y, x\}$ are identical in an undirected graph. $y \rightarrow x \rightarrow u \rightarrow v \rightarrow x$ is a path, but neither a chain nor closed. The sequence $x \rightarrow u \rightarrow v \rightarrow x$ is a chain, and since its initial and final vertex is x, it is closed and thus a circuit.

The definitions and terms used for subgraphs, connectedness, graph labellings, arc-sequences, paths, chains and circuits can equally be applied to digraphs.

2.2.1.3 Trees and Rooted Graphs

Given any connected graph G containing circuits, one can choose one circuit from G and remove one of its arcs. The remaining graph is still connected. This process is repeated until there are no circuits left. The resulting graph is a *tree*

and will be called a *spanning tree* of G. Notice that due to the arbitrary choice of the vertex to be removed from a circuit, a general graph can have many spanning trees.

Definition 2.6 (Tree): A connected graph T with no circuits is called a *tree*.

If this definition is applied unchanged to directed graphs, the connected component over the vertices u, v, x and y on the left side of Figure 2.4(b) forms a tree, because it is free of (directed) circles. This does not coincide with intuition, which requires that every vertex of a tree has at most a single incoming arc. For this reason the definition of directed trees will be based on the circuit-freeness of both the digraph and its underlying undirected graph. The digraph shown in Figure 2.4(e) forms a tree whereas by contrast the left-hand component of Figure 2.4(b) is not a tree according to Definition 2.6.

Note on terminology: Because trees will be of high importance throughout the remainder of the thesis, they will be distinguished from general graphs by means of the terminology and symbols used. The vertices of a tree T will be called *nodes* from here on and its vertex-set will be denoted by $N(T)$ and called *node-set*. Equally, the arcs of a tree T will be called *edges* and its arc-set will be called the *edge-set*, denoted by $E(T)$.

Definition 2.7 (Root, Rooted Graph): A vertex r of a graph G is called the *root* of G if there exists a chain from r to every vertex in $V(G)$. The triple $(V(G), A(G), r)$ is called a *rooted graph*.

For undirected graphs, Definition 2.7 is equivalent to Definition 2.4 on connectedness and every vertex of a connected graph is a root. General digraphs however do not need to have a root. In fact a directed tree has at most one root node. This will be proved by contradiction. Assume a directed tree T has two root nodes r_1 and r_2. By Definition 2.7 there exists a directed chain from r_1 to r_2. For the same reason there also exists a directed chain from r_2 to r_1. Thus there exists a directed cycle $r_1 \rightarrow r_2 \rightarrow r_1$ which violates the definition of directed trees.

Figure 2.4(f) shows a rooted, directed tree. This tree is rooted by the node u, as there exists a unique, directed chain from u to all other nodes. The tree in Figure 2.4(e) is not rooted. Because there exists no chain from any node to u,

only u itself is a possible root node. However there does not exist a chain from u to all other nodes, e.g. the node v can not be reached following a directed chain from u. Thus u is not a root node and the graph is not rooted.

If there exists a chain $n_1 \rightarrow n_2$ between two nodes n_1 and n_2 of a rooted tree, n_1 is called the *ancestor* of n_2 and n_2 is called the *descendant* of n_1. There can be at most one such chain according to Definition 2.6. If such a chain is non-empty, n_1 and n_2 are called *true ancestor* and *true descendant* respectively. If the length of the chain between n_1 and n_2 is one, n_1 is called the *parent* of n_2, which is called the *child* of n_1. The *level* of a node in a rooted tree is the length of the directed chain from the root node to the node in question, e.g. the root node has level zero, its direct children level one and so on. The *height* of a tree is the maximum level of any of its nodes.

The root node u of the tree shown in Figure 2.4(f) has level zero. It is the ancestor of all nodes in this tree. Node z has level three and is the child of node w. It is also a true descendant of the nodes u, v and w.

2.2.2 Data Graphs, Views and XML Documents

All SSD can be described as being isomorphic to a directed graph. Every vertex in the graph represents a piece of data and every arc represents a structural relationship. Note that this leads back to the network [DBT71] and hierarchical data models [BL82] used before the advent of the relational model.

The structure of the data is encoded in the shape or topology of the graph. Arcs in the graph represent relationships between the individual data items. Thus the occurrence of two pieces of atomic data, i.e. the two character strings "Abiteboul" and "Bunemann", below a common vertex imply that these entities are somehow related. However the shape alone does not suffice to indicate the meaning of different items connected to a common vertex. All SSD models thus label either arcs or vertices using tags drawn from an alphabet that carries some application specific semantics. For the above stated example the meaning would be clearer if the arcs connecting the two atomic items with the third vertex were labelled with the tag `author`. This would allow the conclusion that these two strings are actually the names of two co-authors.

For tree data models, the equivalent distinction between edge labelling and node labelling becomes irrelevant [ABS00], as edge labels can be generated from a node labelling by taking the label of every node and attaching it to its single

incoming edge and vice versa. For general graphs however, this mechanism can not be applied as there might be multiple incoming arcs for every vertex. In this case a single distinguished arc must be chosen for the transformation from vertex labelling to arc labelling.

2.2.2.1 Data Graphs

Before the final construct of a data graph is addressed, a last additional definition will be introduced, associating each vertex of a graph with such a tag label as shown in Figure 2.4(g). Unlike the vertex labels introduced in Definition 2.5, these tag labels do not need to be unique, i.e. a *tag labelling* is surjective but not injective in general, i.e. a many-to-one function.

Definition 2.8 (Tag Labelling): A *tag labelling* $\lambda : V \rightarrow \Sigma$ is a surjective mapping function from the set of vertices V of a digraph D to a given finite alphabet Σ of tag names. A *tag labelled* or *tagged graph* is a pair (D, λ).

The last construct defined in this section pulls together all the definitions provided above into a single definition of a *semistructured data graph.* This is done in order to bring the general theoretical construct of digraphs closer to the requirements of SSD processing in general and XML in particular. One more preliminary is needed, which partitions a vertex set into two disjoint subsets called *atomic* and *complex.* The semantics associated with these sets will be deferred until after the definition. Until then the only difference between them will be that members of the atomic vertex subset represent atomic information, which can be represented by a character string for the purposes of this thesis. By contrast complex vertices combine the information of other vertices by the means of a set of outgoing arcs.

Definition 2.9 (Data Graph): A *data graph* DG is an octuple

$$DG = (V, A, E, r, \varphi, \lambda, atom, value)$$

of a vertex-set V and an arc-set A such that (V, A) is a connected digraph, an edge-set $E \subset A$ such that (V, E, r) forms a spanning, rooted tree of (V, A) with root r, a graph labelling $\varphi : V \rightarrow \Phi$ and a tag labelling $\lambda : V \rightarrow \Sigma$, a function $atom : V \rightarrow boolean$ distinguishing vertices from the atomic and complex subset of V and a partial function $value : V \rightarrow string$ giving the string representation for a given vertex from the atomic subset.

The tag alphabet Σ consists of the fixed entry ROOT, the fixed entry DATA for all data graphs containing at least a single atomic vertex and a finite set of application specific tag labels, which are distinct from these. The vertex labelling function $\lambda : V \rightarrow \Sigma$ returns $\lambda(v) =$ ROOT if and only if $v = r$, i.e. v is the root of (N, E, v) and $\lambda(v) =$ DATA if and only if $atom(v) =$ **true**. Equally the partial *value* function is defined for all $v \in V$ with $atom(v) =$ **true** and undefined otherwise. The atomic and complex vertex-subsets will be called $V^A = \{v \in V | atom(v)\}$ and $V^C = \{v \in V | \neg atom(v)\}$ respectively. These subsets are disjunct, i.e. $V^A \cap V^C = \{\}$, and their union is a cover of the vertex-set, i.e. $V^A \cup V^C = V$. Thus $\{V^A, V^C\}$ is a partition of the vertex-set V.

This definition provides the data model used for this thesis. It has the advantage of being a superset of many of the data models assumed by researchers in this area. This is important, as the thesis will frequently discuss such works in the concepts that are developed as part of it.

Figure 2.4(h) shows an example of a data graph. Its vertex set is $V = \{u, v, w, x, y, z\}$, its arc-set is $A = \{(u, v), (u, x), (v, w), (v, x), (w, z), (x, y)\}$, its edge-set $E = \{(u, v), (u, x), (v, w), (w, z), (x, y)\}$ and its root is u. The values for the functions λ and φ can be derived from the figure, the tag alphabet is $\Sigma =$ {ROOT, DATA, paper, author, title}. The function *atom* is **true** for the vertices y and z and **false** otherwise. The values of the function *value* is the string "abc" and "xyz" for the vertices y and z and undefined for all other vertices. Arcs that are present in the arc-set A and the edge-set E are shown solid, while arcs that are not part of the edge-set E are shown dashed. Vertices that are part of the complex subset are shown as solid circles and atomic vertices are shown as dashed circles. Their fixed tag label DATA is shown in Figure 2.4(h), but will be omitted in all further data graphs for reasons of simplicity.

2.2.2.2 Tree-View versus Graph-View of a Data Graph

As stated above, SSD models are based on digraphs, hence generally instances of SSD sources will contain cycles, e.g. a paper written by an author, who has written a paper that was written by this author and so on. Such a situation could be created by adding another arc from vertex **&3** to vertex **&1** of Figure 2.4(h). However, this assumption often increases the complexity of the algorithms used to process SSD and can lead to non-determinism for algorithms traversing the infinite path paper → author → paper → ... As many practical data sources

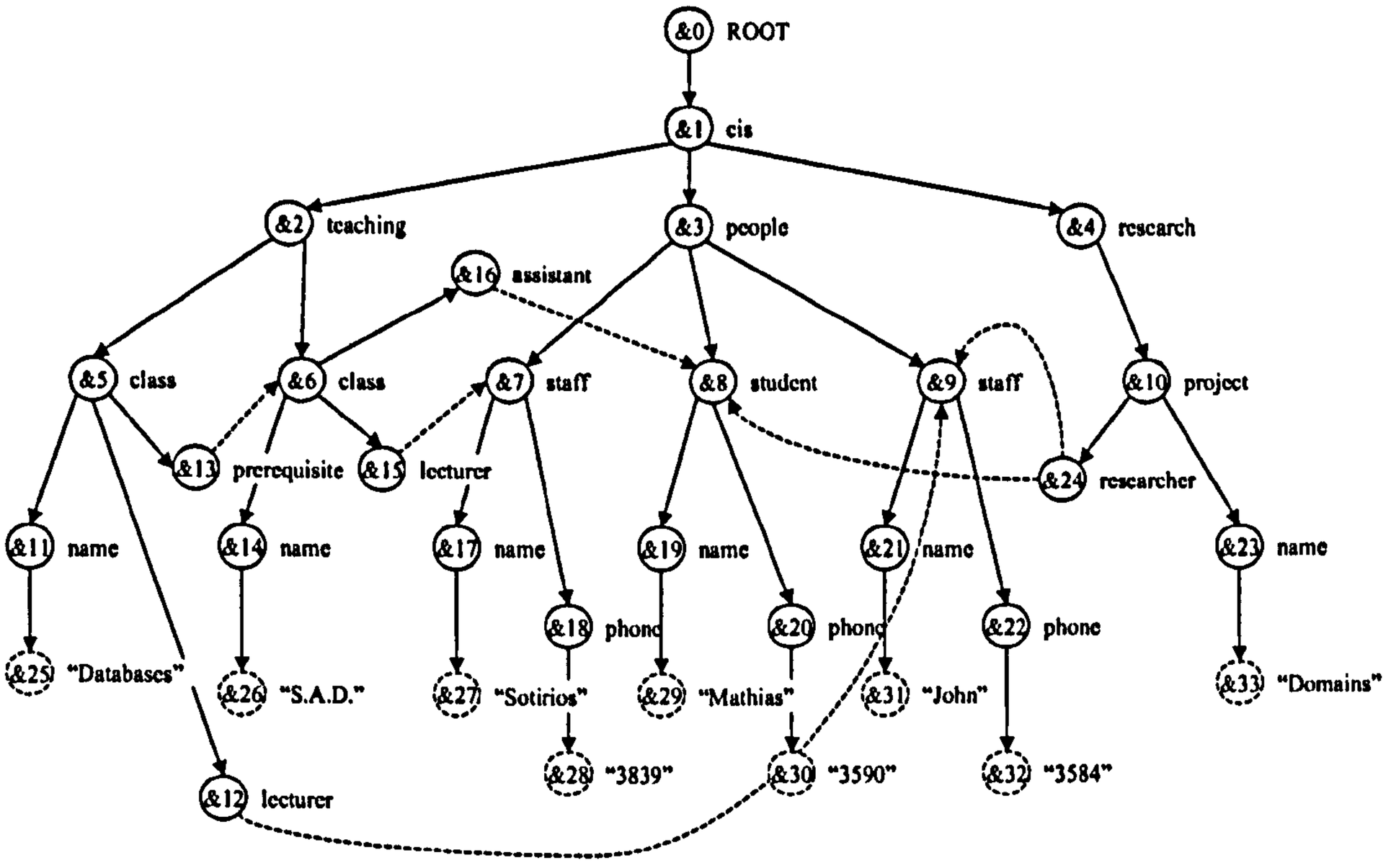


Fig. 2.5: An example of a data graph

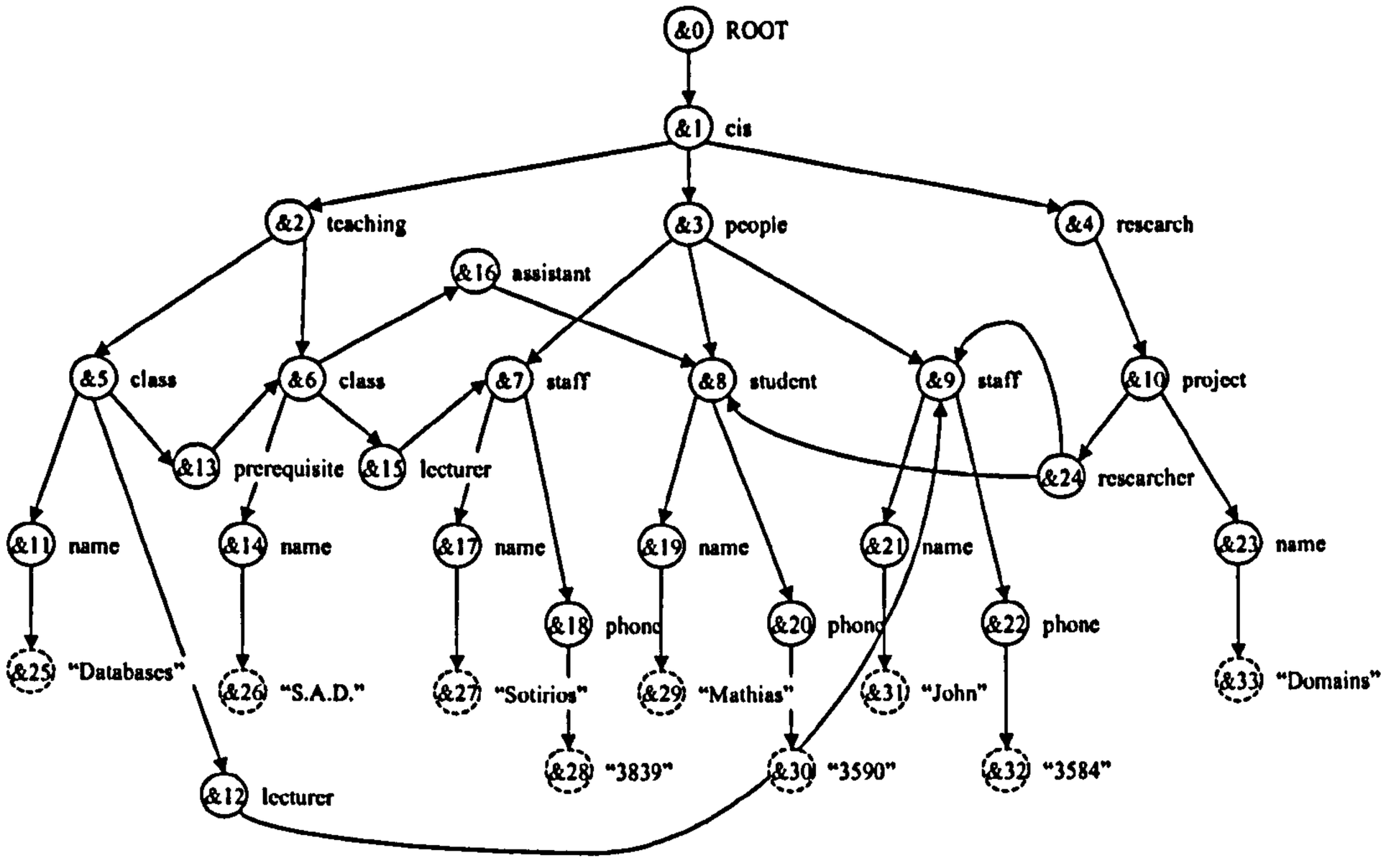
are often of hierarchical nature, i.e. free of cycles, this generality is often dropped in favour of a simpler tree based data model [XML01].

Formally this important issue is addressed by the definition of two distinct views of the data graph, its graph-view and its tree-view.

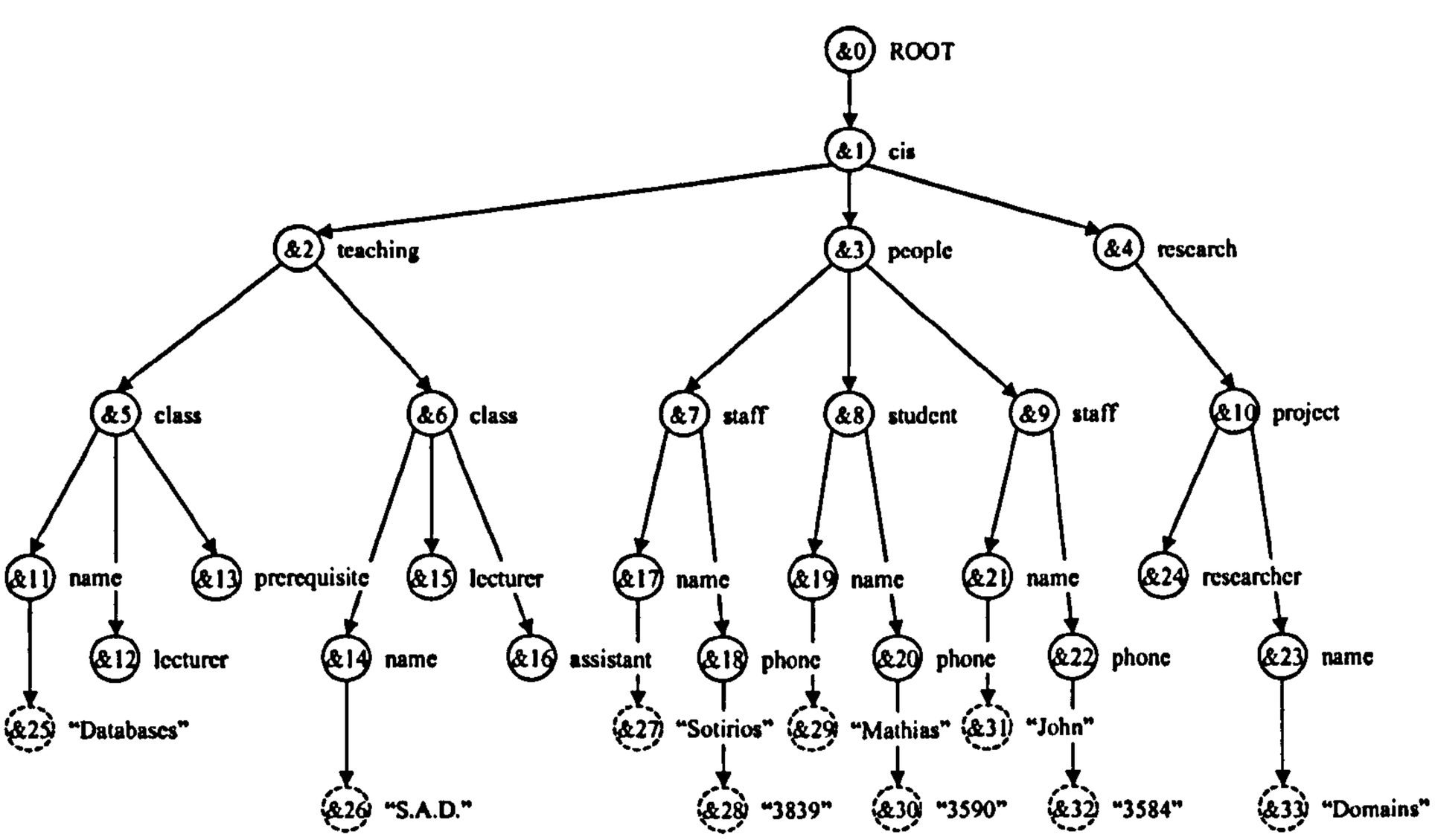
Definition 2.10 (Graph-view): The directed *graph-view* $D(DG)$ of a data graph DG with $DG = (V, A, E, r, \varphi, \lambda, atom, value)$ is the tagged digraph given by the triple (V, A, λ).

Definition 2.11 (Tree-view, Distinguished Spanning Tree): The *tree-view* of a data graph $DG = (V, A, E, r, \varphi, \lambda, atom, value)$ is given by the quadruple (V, E, r, λ) and denoted $T(DG)$. Following Definition 2.9, $T(DG)$ is a spanning, rooted tree of $D(DG)$ and will be called the *distinguished spanning tree* of the data graph DG in the context of this thesis.

Figure 2.5 shows a more complex data graph than that of Figure 2.4(h). Two different views of the graph of Figure 2.5 are shown in Figure 2.6. In the graph-view shown in part (a), spanning tree edges and additional graph arcs are indistinguishable, whereas the tree-view of part (b) only shows the edges of the distinguished spanning tree. Notice that the referencing nodes of the tree-view



(a) The graph-view of the data graph shown in Figure 2.5



(b) The tree-view of the data graph shown in Figure 2.5

Fig. 2.6: The graph- and tree-view of the example data graph shown in Figure 2.5

&12, **&13**, **&15**, **&16** and **&24** are still part of the complex subset, although they have no outgoing edges in this view.

These definitions allow the arbitrary, potentially cyclic, digraph $D(DG)$ to be replaced by the guaranteed cycle-free tree representation $T(DG)$ over the identical vertex-set V for operations that do not depend on the additional graph arcs. It is possible to define an auxiliary function $tree : A \rightarrow boolean$ with

$$tree(a) = \begin{cases} \text{true for } a \in E \\ \text{false for } a \notin E \end{cases}$$

on DG, which can be used to decide whether a given arc a is part of the distinguished spanning tree $T(DG)$.

2.2.2.3 XML Documents and Data Graphs

The distinction between tree edges and graph arcs will be beneficial in order to match the data model presented here against that made explicit by the XML Infoset [XML01]. XML, like every other flat file format, can only encode non-cyclic data sources directly, but offers several mechanism to encode arbitrary graphs indirectly, e.g. through the use of special ID:IDREF attribute pairs or application specific extensions such as XPointer [XLi01]. The element-subelement relationships encoded in an XML document define the distinct, spanning tree of the data graph presented, i.e. the edges of the tree $T(DG)$. These edges lead from an element to its subelements in terms of the XML document structure, or from parent to child node in terms of the tree. Any additional relationships, such as those encoded using ID:IDREF-references, are additional graph arcs only present in the arc-set A. They start at the referencing vertex, e.g. the element containing an IDREF-attribute, and lead to the referenced vertex, e.g. the element containing the respective ID-attribute. Following query standards such as XPath, which restrict the operations possible on such implied arcs, several researchers have also used data models that reflect this distinction [GMW99, KB+02].

Another peculiarity of the XML format is the choice between two distinct ways of representing values associated with a given element. One can either use a subelement with the desired tag name containing atomic information or make use of a CDATA-attribute encoding a key-value pair [Bra03]. For the purpose of this thesis such a distinction is deemed irrelevant and not directly supported by the data model. However, for applications depending on this distinction, one

variant can be identified by annotating resulting vertex-labels with a reserved prefix code, e.g. by labelling a *name* entity encoded using a CDATA-attribute `@name`, whilst labelling a subelement with `name`. A discussion of the importance of document order is postponed until Section 2.2.3.1. Other specialties of the XML standard such as comments and processing instructions, which it inherited from its SGML origin [SGM86], will be ignored within this thesis.

The data graph presented in Figure 2.5 is the one generated from the XML document shown in Listing 2.3.

2.2.3 Order and Identifier Based Models for Semistructured Data

The following sections discuss the significant differentiation between SSD models based on vertex order versus those models based on vertex identity. These concepts are essentially complementary. The data model represented by the data graphs of Definition 2.9 use a concept of explicit identity as detailed in Section 2.2.3.2. Other models, usually based on a document-centric viewpoint, prefer to incorporate a concept of order. Such models are described in Section 2.2.3.1. Whether order or identity should be reflected in the logical model is arguable. Which approach is favoured depends largely on the point of view from which the model was developed.

2.2.3.1 Sibling Order Based Models

For researchers working on document-centric SSD, i.e. looking at semistructured data as a refinement of unstructured data, the question of order among the individual data atoms is important. Such data is primarily meant for human consumption, often encoding natural language, which crucially depends on order. Since XML arose as an interchange or serialisation format, its physical implementation also implies an order. Query languages such as XPath [XPa99, XPa03], which are defined specifically for use with XML, consequently support order-related query operations. In terms of a graph model this corresponds to the question whether the outgoing arcs of a vertex are ordered or not.

SSD models based on the assumption of a sequential access, like compressors [LS00, SM01] or aimed at supporting order-dependent query languages like XPath [BGK03] thus consider as ordered the edges starting from a common vertex. In

```
<?xml version="1.0" ?>
<!DOCTYPE cis [
<!ELEMENT cis ( teaching, people, research ) >
<!ELEMENT teaching ( class* ) >
<!ELEMENT class ( name ) >
<!ATTLIST class id ID #REQUIRED
                lecturer IDREF #IMPLIED
                assistant IDREFS #IMPLIED
                prerequisite IDREFS #IMPLIED >
<!ELEMENT people ( (student|staff)* ) >
<!ELEMENT student ( name, phone? ) >
<!ATTLIST student id ID #REQUIRED>
<!ELEMENT staff ( name, phone ) >
<!ATTLIST staff id ID #REQUIRED>
<!ELEMENT research ( project* ) >
<!ELEMENT project ( name ) >
<!ATTLIST project id ID #REQUIRED
                  researchers IDREFS #IMPLIED>
<!ELEMENT name ( #PCDATA ) >
<!ELEMENT phone ( #PCDATA ) >
]>
<cis>
        <teaching>
                <class id="CIS.234" lecturer="s199524875" assistant="s200155317">
                        <name>SAD</name>
                </class>
                <class id="CIS.356" lecturer="s198568459" prerequisite="CIS.234">
                        <name>DB</name>
                </class>
        </teaching>
        <people>
                <student id="s200155317">
                        <name>Mathias</name>
                        <phone>3590</phone>
                </student>
                <staff id="s198568459">
                        <name>John</name>
                        <phone>3584</phone>
                </staff>
                <staff id="s199524875">
                        <name>Sotirios</name>
                        <phone>3839</phone>
                </staff>
        </people>
        <research>
                <project id="SSD" researchers="s200155317_s198568459">
                        <name>Domains</name>
                </project>
        </research>
</cis>
```

Listing 2.3: The XML encoding of the data graph shown in Figure 2.5

these models, individual vertices can be addressed by specifying the order of the sibling edges of its unique path from the root in the distinguished spanning tree.

2.2.3.2 Vertex or Node Identity Based Models

Instead of storing the order of siblings, one can provide each vertex with an explicit address or identifier. The relational model defined by Codd [Cod70] is based entirely on set theory and thus does not use the concept of either identity or order. It is rather founded on tuple equality, which is based on the equality of the attribute values of a tuple. Thus changing the value of an attribute of a tuple is indistinguishable from deleting the old tuple and creating a new tuple containing the new attribute value. No relation can contain two tuples with identical attribute values. Its successor, the object-oriented database model however, introduces the concept of object identity. Here two distinguishable objects with equal attributes can co-exist in the same relation.

Researchers from the database community usually see the semistructured model as a further development of the object-orientated approach and thus assign identifiers to each vertex of the data graph. These are, for example, part of the OEM data model (Section 2.5.1.1) and correspond to the graph labelling φ which is part of Definition 2.9 of data graphs, which will serve as data model for this thesis.

For data models that do not make use of this feature, the labelling can be dropped from the definition. The XML standard represents a compromise, which does not enforce identities for every vertex but does provide the means of supplying them by the use of special attributes of type ID.

2.3 Querying Semistructured Data

Query languages fulfil three different functions in a database management system [ABS00]. Firstly they allow the selection of the subset of a potentially large database, which is relevant to the computation to be performed. Secondly they allow the joining of partial results from multiple sources in order to derive new facts by combination. The third function is the restructuring of data according to a desired output format, potentially creating new data in the database. These functional areas exist equally in query languages designed for structured and semistructured data. Bergholz and Freytag [BF00] proposed a similar decompo-

sition of queries over SSD into a selective "what" and a constructive "how" part used for their work on schema based querying.

This thesis is only concerned with the first function, i.e. the selection of relevant data. Section 2.3.1 thus derives a syntax and associated semantics for a purely selective query language. Following this Section 2.3.2 looks at how this is incorporated in query languages designed particularly for use with XML. The concluding Section 2.3.3 describes a range of general evaluation strategies for queries on SSD. The issues of joins and result construction are addressed for example by Abiteboul et al. [ABS00] and not further discussed here.

2.3.1 Path Expressions as a Selective Query Languages

This thesis is only concerned with the selective functionality of a query language, i.e. the process by which different groupings of SSD can be used to efficiently select relevant subsets of a database for a given computation. This is based on the assumption of very low query selectivity, which is reasonable for the environment depicted in Chapter 1. In such circumstances subsequent steps like joins and transformation algorithms can benefit from the reduced data volume, improving their performance. The following sections define a syntax and semantics for path expressions that can be used to perform the task of vertex selection.

2.3.1.1 Linear Path Expressions

Virtually all query languages for SSD are based around the concept of path expressions, i.e. a way of specifying structural relationships between vertices using paths in the data graph. This section discusses *linear path expressions*, i.e. expressions matching a single, non-branching path in the data graph.

The basic form of path expressions are *simple path expressions*, which are *absolute* (their matching process starts at the root of the data graph), *complete* (they specify every label along the matched path) and exclude regular expressions on the tag labels. Such path expressions have the form $/l_1/\ldots/l_n$, where l_i stands for any label from the label alphabet Σ. A simple path expression matches a vertex v_n of DG, if there exists an arc-sequence $v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_n$ in DG with $v_0 = root$ and $\lambda(v_i) = l_i$ for all $i \in [1, n]$. For example the arc-sequence $\&0 \rightarrow \&1 \rightarrow \&3 \rightarrow \&8$ of the data graph shown in Figure 2.5 matches the simple path expression `/cis/people/student`.

Simple path expressions require a good knowledge of the structure of the data graph, as a complete path from the root to the vertex of interest needs to be specified. In order to make full use of the flexibility offered by the semistructured model, *partial* and *relative* path expressions are supported by all query languages. Here parts of the path that are irrelevant to the computation can be left out. Partial query expressions leave out label constraints matched by intermediate sections of an arc-sequence and replace them by the descendant operator '//'. Relative path expressions start at an arbitrary vertex rather than the root, indicated by a leading descendant operator. Notice that the latter is just a syntactical convenience for a partial expression involving the root label predicate and the first label predicate specified in the relative path expression, i.e. $//l_1$ is just a shorthand notation for $///l_1$. Instead of an arc-sequence, a path-sequence in the data graph is being matched to a path expression of the form $//l_1//\ldots//l_n$. For example the partial, relative path expression `//people//name` selects the names of staff and students alike and is matched by the set of paths $\{\&3 \rightarrow \&7 \rightarrow \&17, \&3 \rightarrow \&8 \rightarrow \&19, \&3 \rightarrow \&9 \rightarrow \&21\}$ of the data graph presented in Figure 2.5.

Regular path expressions are path expressions in which the tag label constants l_i are replaced with *label expressions* λ_i. Label expressions follow standard conventions for regular languages. A minimal grammar used for these expressions throughout this thesis is $\lambda ::= l \mid \lambda_1|\lambda_2 \mid *$, where l denotes a label constant from the alphabet Σ, $\lambda_1|\lambda_2$ defines a choice between two label expressions, i.e. a disjunctive query, and '*' denotes the wildcard matching every label from Σ.

All path expressions presented so far are *forward facing*, i.e. they follow the natural direction of the arcs in the data graph or more specifically, lead from ancestor nodes to descendants in its tree-view. For the linear path expressions presented here this is sufficient as every *backward facing* path expression can be transformed into a forward facing path expression [OM+02] by inverting the order of the label constraints and replacing backward with forward path separators.[2] For example the query for a `name` vertex, which can be reached from a `people` vertex, can be replaced by a query for `people` which have an outgoing arc to a `name` vertex. However, this will be different in the case of *branching path expressions*, which are described in the next section.

[2] Whether or not such a query is semantically identical depends on the expected output, i.e. whether the complete path in the data graph matching the query is returned or solely its final vertex, matching the rightmost predicate. This question will be deferred until Section 2.3.1.4.

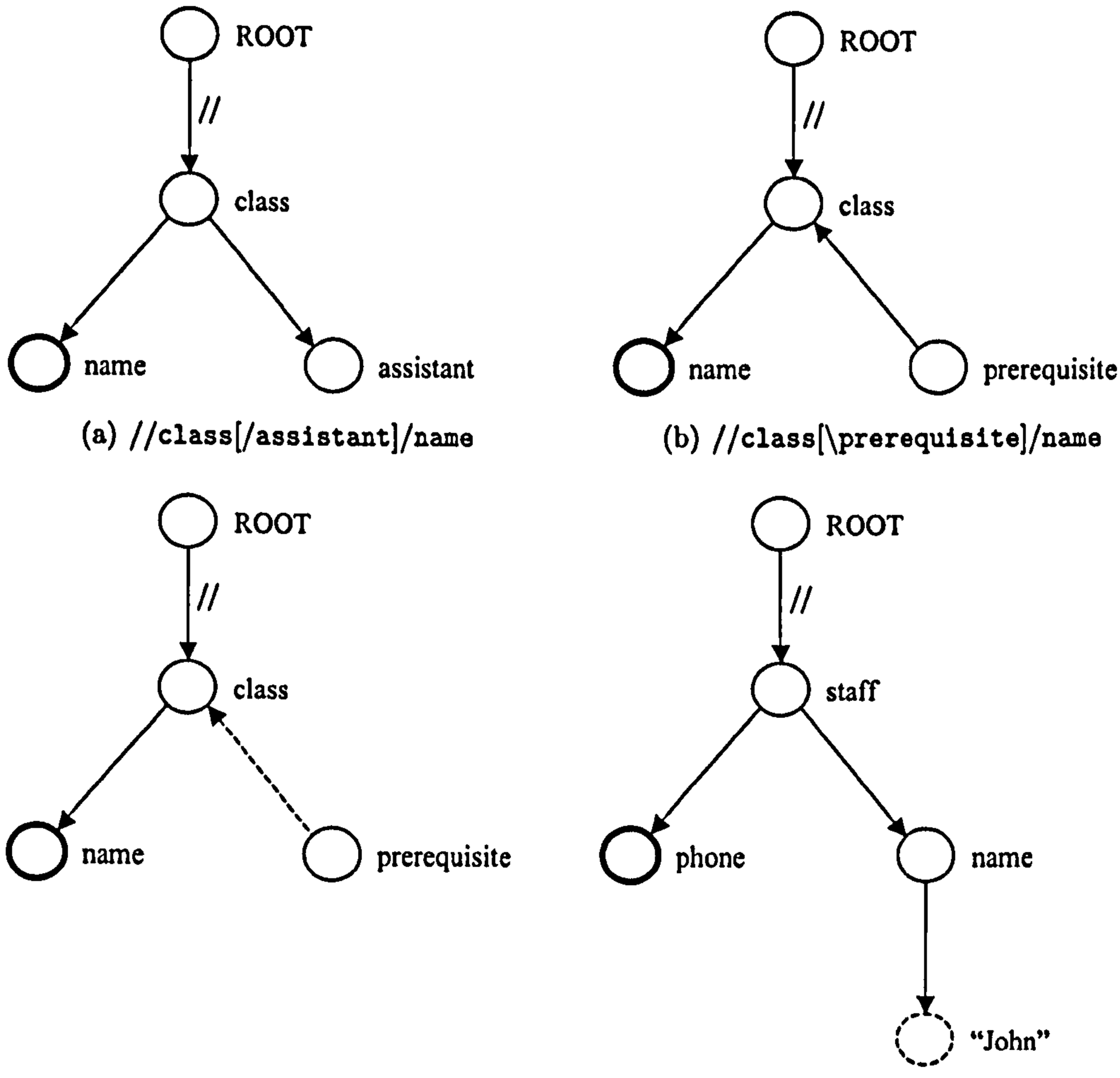


Fig. 2.7: Example graphs of branching path expressions

2.3.1.2 Branching Path Expressions

Linear path expressions alone cannot be used to encode some of the more complex structural constraints which could usefully be imposed on the vertices being selected, e.g. one cannot specify the selection of the names of all classes that have an associated teaching assistant, i.e. express *conjunctive structural constraints*. Such queries can easily be represented as graphs, as has been done for the specified example in Figure 2.7(a). In order to distinguish the vertices of a query graph from those of a data graph, its vertices will be called predicates. For reasons of simplicity all query graphs considered in this thesis are non-cyclic and connected, i.e. they can be represented by a tree. In addition, a query graph will always

contain a predicate matching the root vertex of the data graph, even if this is suppressed in the linear syntax by the use of an initial '//' operator.

The `class` predicate of Figure 2.7(a) is matched by the vertex **&6** of the data graph shown in Figure 2.5. Notice that this predicate is shown in bold. It is called the *output predicate*, i.e. the set of vertices of the data graph that match this predicate will form the result set. Following the terminology used by Kaushik et al. [KB+02], the path leading from the root of the query graph to the output predicate will be called the *primary path* of the query. All other paths form structural constraints on this path. This helps to define a flat representation of a query graph by writing the primary path in the form introduced for linear path expressions above and annotating the tag predicates relating to branching points in the query graph with structural predicates, which are enclosed in square brackets. If multiple such structural constraints are attached to a branching point, the individual expressions are separated by the '&'-sign and need to be matched by the same vertex of a data graph to return a match. Thus the example presented in Figure 2.7(a) can be written as `//class[/assistant]/name`.

The notation used so far is sufficient to express all possible queries for tree patterns in the tree-view of the data graph. However, for general queries on the graph-view of a data graph one might be interested in incoming as well as outgoing paths. Thus backward facing path expressions are introduced in the same way as forward facing path expressions. A single backward arc matches the query operator '\' in the path expression and a backward path of arbitrary length matches the operator '\\'. Thus the query asking for all class names that are the prerequisite for something else can be expressed as `//class[\prerequisite]/name` and is shown in Figure 2.7(b). The result of this query for the graph-view of Figure 2.6(a) is the set containing solely vertex **&6**.

Notice that supporting such general query graphs with multiple incoming paths to a predicate is only required for the graph-view, in which all arcs are indistinguishable. Working on the data graph itself with its two different types of arcs, one might want to specify that the arc incident from the `prerequisite` vertex is actually an additional arc rather than an edge of the spanning tree. This reflects the expressive power of XPath, in which IDREF references also need to be specified. To supplement the forward and backward path operators, reference path operators are introduced, which are only matched by the additional arcs not contained in the spanning tree. These are denoted by '⇒' and '⇐' and match these additional arcs in or against their natural direction respectively. Notice

that these operators are always matched by arcs and never by paths in the data graph. Thus the query for names of classes that are prerequisites is actually written //class[⇐prerequisite]/name in the data model provided by the data graph. Graphically such arcs will be displayed as dashed arrows as shown in Figure 2.7(c) following the convention used for the data graph itself.

2.3.1.3 Atomic value predicates

The previous section was concerned with placing structural constraints on vertices that match predicates of the query graph, thus those predicates will be called *structural predicates.* Similarly *atomic value predicates* can be used in order to restrict the returned vertex-set based on the data attached to atomic vertices of the data graph. This is equivalent to the selection operation in the relational algebra. The selective functionality of a query language can be further subdivided into a binding process, here represented by structural predicates, and a filtering process, here represented by atomic value predicates [ABS00]. The initial step requires that vertices of the query graph representing tag expressions are bound to vertices of the data graph according to structural constraints. The subsequent filtering process removes such bindings that do not comply with the given atomic value constraints. Because techniques for data selection based on atomic values are known from relational database research, most research on querying SSD concentrates on structural predicates, e.g. the work on type projection presented in Appendix A or that of Buneman et al. [BGK03] and Kaushik et al. [KS+02, KB+02] on structural summaries for SSD. However, the integration of atomic value predicates into the first stage of the query process is often beneficial as such predicates can have considerably lower selectivity than structural predicates. This is especially true if the data has originated from a regular source, in which case the structure of the resulting data graph is fairly regular and consequently structural predicates select a significant part of the data.

Only leaf nodes of the distinguished spanning tree can possibly match atomic value predicates. Thus syntactically, atomic value predicates are separated by an '='-sign from the last label expression of a path expression, with their associated matching expression enclosed in double quotes in the flattened query syntax. In the graph representation they will be shown as dashed vertices in imitation of the notation used for atomic vertices of the data graph. As an example one could ask for the telephone number for a member of staff named "John" as depicted in

Figure 2.7(d). This example query would be matched by vertex &22 of the data graph shown in Figure 2.5.

2.3.1.4 Semantic Variations on Query Results

There is agreement between different query languages on how to match vertices of data graphs to the predicates of given query expressions. However there exist variations on the semantics of the expression as a whole, i.e. of what result they return. Here three different possible variations on the semantics attached to the path expressions introduced above will be discussed. The relative linear path expression Query 2.1 will be used to discuss these variations in semantics.

Query 2.1: `//people//name`

The first possible option is to return the set of complete embeddings of vertices of the data graph in the query graph. Thus Query 2.1 containing two predicates will return the set of pairs $\{(\&3, \&17), (\&3, \&19), (\&3, \&21)\}$. This is very useful if the example query forms a part of a more complex query, especially if joins on several of the returned vertices are to be performed. Due to the fact that the complete embeddings are returned, this option will be referred to as *query embedding*.

If however the query stands on its own and one is only interested in the vertices addressed by the query expression, i.e. the final predicate on its primary path, this complete embedding may be wasteful. In this case an interpretation which only returns the set of vertices mapping this distinguished predicate is more appropriate. Applying this semantic option to Query 2.1 returns the set of vertices $\{\&17, \&19, \&21\}$. This is the variant being adapted by this thesis unless specified otherwise. It will be called a *path expression* as indicated in the previous sections. In its flattened representation, the last predicate of the primary path is the predicate whose embeddings are returned. If a query is presented as graph the output predicate will be shown in bold.

Another option is to return the embeddings of vertices into the first specified predicate of a query expression, i.e. the predicate that is adjacent to the implied predicate matching the root of the data graph. The data graph shown in Figure 2.5 can be embedded into Query 2.1 in three different ways, however in all cases it is vertex &3 being embedded into the `people` predicate. Thus the result set contains only this single entry using these semantics. This semantic option

will be called *tree pattern query* as it decides at which vertices a given query graph can be embedded into a data graph. This behaviour can be expressed using the path expressions introduced above by specifying all other predicates as structural predicates of the first one, i.e. by rewritting Query 2.1 in the the form `//people[//name]`.

In some situations a complete embedding of any vertices can be wasteful. If the only question to be answered is whether a certain pattern exists, the matching does not need to be completed, but can be terminated as soon as it becomes evident that this is the case. Thus the result of a query with this semantic is the boolean value **true** or **false**. Such semantics are used for structural predicates of branching path expressions, i.e. all path expressions apart from the primary path are resolved using these semantics.

2.3.2 Query Languages for XML

XML is a practical embodiment of SSD that has attracted widespread attention. Consequently many languages have been designed to achieve the task of querying XML documents. Most notably among them is XQuery [XQu03], a declarative query language for XML, which has been defined by the W3C Architecture Domain[3]. XQuery can be broken down into a selective part, which is addressed by the XPath [XPa99, XPa03] language, and join and construction mechanisms, which are addressed by the XQuery language definition itself [XQu03]. As stated in the introduction to this section, the focus of this thesis will rest on the selective part represented by XPath. It should be noted here that, due to its historical development, XPath is a self-contained query language. It contains a number of expressions, usually implemented through some core functions of the language, that extend beyond the expressive capabilities of its purely selective location paths. XPath Version 2.0 [XPa03] is in fact a Turing-complete language. Both XPath and XQuery can be used in data-centric and document-centric environments. XPath's location paths form the most important kind of expressions in XPath. Their result is always a list of nodes as defined in the XML Infoset data model [XML01]. This is very similar to the branching path expressions discussed in the previous section and thus well-suited for a data-centric environment. However, the following example shows how to use XQuery in document-centric environment, returning the entire document fragment rooted at the vertex being

[3] `http://www.w3.org/Architecture`

selected by a location path by binding the selected node and returning its content. Notice that this differs from the semantics introduced in Section 2.3.1.4.

Query 2.2: `for $s in /cis/staff/people/student return $s`

Query 2.2 would return the following result containing all information about students if applied to the XML document shown in Figure 2.3.

```
<student id="s200155317">
        <name>Mathias</name>
        <phone>3590</phone>
</student>
```

Another important standard of the W3C is XSLT [XSL99], a rule based language for XML document transformation, which is also based on XPath as a selection language. Unlike XQuery it is pattern based and more often used in data-centric environments.

The importance of other XML query languages such as XML-QL [DF+98] and XQL [RLS98], which filled the gap created by the lengthy standardisation process leading to the XQuery standard, will probably diminish in the near future. A comparative study of XML-QL, XQL and three other languages was performed by Bonifati and Ceri [BC00]. All of these query languages represent functional subsets of the XQuery standard.

2.3.3 Query Evaluation Strategies

As noted in Section 2.3.1.3, the selectivity of individual predicates of a query expression influence its execution performance. Thus different strategies for query execution can be employed in order to improve the efficiency if statistical information about the selectivity of query predicates is available.

Three such strategies, a top-down, a bottom-up and a hybrid variant, will be described using simple example queries over the data graph shown in Figure 2.5. For reasons of simplicity, all queries presented here can be encoded in a single linear path, i.e. no predicate has more than one child predicate, and only arcs which are part of the spanning tree are followed. Despite this Query 2.4 and Query 2.5 do not represent simple path expressions, because they return the embeddings for predicates different from the leaves and contain regular expressions. In general the concepts described in the following section can be extended to more complex query graphs, i.e. arbitrary branching path expressions.

2.3.3.1 Top-down Querying

The *top-down* query evaluation strategy is the most natural in that it follows the vertices of the source in their specified direction until all requirements are fulfilled. This behaviour will be demonstrated using Query 2.3.

Query 2.3: `//project/name/DATA`

In this example at first all `project` vertices will be identified. A search for `name` vertices will be performed by following outgoing arcs from the identified `project` vertices. From there, again only following outgoing arcs, a further search for atomic vertices concludes the query. This can be compared to finding information in a well-structured book starting from the table of contents. Using the data graph of Figure 2.5 only a small number of vertices would be visited if one assumes access to a label map. This allows the matching process to start at the single `project` vertex within the graph, which only has two outgoing arcs, one to the requested `name` vertex and one leading to an irrelevant `researcher` vertex. The former vertex only has a single outgoing arc, leading to an atomic value vertex as required. Thus this strategy has visited only four vertices of the data graph in order to embed them into three predicates, making this a good strategy.

2.3.3.2 Bottom-up Querying

The *bottom-up* query evaluation strategy represents the inverse to the top-down strategy. Here occurrences of leaf vertices in general and the target atomic data vertices in particular are identified and their structural constraints are validated by traversing the arcs of the source in their inverse direction. This strategy will be described using the following example query on our example data graph.

Query 2.4: `//staff[/*/DATA` = "John"]

In this example at first all occurrences of the string "John" are sought and then validated for an incoming arc from an arbitrarily tagged vertex. This vertex in turn is validated to have an incoming arc from a vertex with the tag label `staff`, whose vertex identifier will form the result set. This can be compared to finding information using the index of a book. Since there is only a single atomic vertex with value "John", such a strategy would lead to a very limited search on the data graph of Figure 2.5, making this a good query strategy. In fact, since

the query is only considering those arcs of the data graph, which are also edges in its tree-view, every vertex has at most one such arc. Thus, starting from a fitting leaf predicate, either a valid vertex is found or a mismatch detected after at most two steps for this query despite the wildcard. If a top-down approach was used, not only would the subtree rooted at the non-matching `staff` vertex &7 have been searched, but the algorithms would have also visited the `phone` vertices &18 and &22 together with their attached atomic value vertices, although they are irrelevant for the query result.

2.3.3.3 Hybrid Querying

The decision about whether a top-down or bottom-up strategy is more appropriate is usually based on statistical knowledge about the selectivity of top- or bottom-level predicates in the query tree. If this knowledge is not available or the selectivities are of comparable order, a mixture of both, called a *hybrid* querying strategy, might be the most successful solution. This will be illustrated using Query 2.5. The syntax used for Query 2.5 was slightly extended to allow for regular expressions on the atomic value predicate. Its meaning is to select all atomic values starting with the character sequence "35".

Query 2.5: `//staff/*[/DATA = "35*"]`

In this case, part of the query is evaluated in a top-down fashion and part of the query is evaluated bottom-up. This gives two supersets of the query answer. At the point where the top-down and bottom-up parts of the query meet, the intersection of these sets is formed, giving the query result. For the example query neither a top-down nor a bottom-up strategy would prove particular successful. The top-down approach would visit all vertices below the two staff nodes as in the previous example. Due to the higher selectivity of the atomic value predicate, the bottom-up strategy would also travel along the unnecessary path leading upwards from the atomic value vertex &30. This can be avoided if a top-down approach is used to match all `staff` vertices and their four children &17, &18, &21 and &22, while a bottom-up query selects the two matching atomic value nodes &30 and &32. Their parents &20 and &22 can now be intersected with the previously computed set to provide the queries proper result of {&22}. This strategy does not improve the efficiency of the particular query on the given example. However this is mainly due to its limited size and complexity.

The different query strategies presented in this section are aimed at achieving optimal efficiency on a given data graph. However, as has been noted before, many SSD sources contain a significant regular core [DFS99]. In Query 2.3 from above it is unnecessary to validate the occurrence of an atomic value vertex below a `name` vertex as all such vertices fulfill this requirement. Such structural similarities should be validated once only and not once per instance. In some environments this could be achieved by looking at the document schema, but in general such a schema might be too inaccurate or even absent. Thus automatically built summaries, describing important features of the source are required.

2.4 Indexing Semistructured Data

Access speed to large, semistructured data sources can be significantly improved by the means of indexing, i.e. by providing direct access to particularly important aspects of the source using a secondary data structure. In contrast to the relational case however, the structure of the source is not necessarily flat, thus SSD in general requires more complex index structures.

2.4.1 Linear Index Structures

If one is to limit the indexing to a flattened view of a particular source, e.g. the atomic values occurring in a given context, indexing techniques known from relational database research can be applied with little or no change. Two very simple, yet fundamental indices of this kind are presented here, the tag or label index and the atomic value index.

Example 2.1 (Tag Index): A *tag index* is a mapping providing the list of vertex identifiers V_l for every tag label $l \in \Sigma$ with $V_l = \{oid(v) | v \in V \text{ with } \lambda(v) = l\}$ of a data graph.

Figure 2.8 presents a tag index for the example data source presented in Figure 2.5. It allows the quick location of all vertices bearing a certain label tag. Tag indices are useful, for example, in order to resolve relative path expressions, i.e. path expressions that can start from an arbitrary vertex and not necessarily the root vertex.

Notice that a tag index implies a partition on the data graph as every vertex carries exactly one label from Σ. This will be used in Example 4.1 of Section 4.2.1,

l	V_l
DATA	&25, &26, &27, &28, &29, &30, &31, &32, &33
ROOT	&0
assistant	&16
cis	&1
class	&5, &6
lecturer	&12, &15
name	&11, &14, &17, &19, &21, &23
people	&3
phone	&18, &20, &22
prerequisite	&13
project	&10
research	&4
researcher	&24
staff	&7, &9
student	&8
teaching	&2

Fig. 2.8: The tag index for the example source

where different equivalence relationships on vertices will be exploited in order to define domains.

Example 2.2 (Atomic Value Index): An *atomic value index* is a mapping that provides a list of vertex identifiers V_s for every character string s with $V_s = \{oid(v)|v \in V^A \text{ with } value(v) = s\}$ of a data graph.

Figure 2.9 presents an atomic value index for the example data source presented in Figure 2.5. An atomic value index is useful to resolve value based queries, regardless of the context in which the corresponding atomic vertex appears.

Notice that an atomic value index does not imply a partition as defined above, as complex vertices are not mapped by any index entry. It does however define a partition on the subset V^A of atomic vertices. In document-centric situations, where keyword searches or regular expressions on atomic values are to be performed, other linear indexing techniques such as inverted lists can be appropriate. Lore's text index (Tindex) [MW+98] is an example of such an approach. This however lies outside the scope of this thesis.

Notice that although these indices were presented as tables in this section, they will usually be implemented using advanced data structures such as B-Trees known from relational database research.

s	V_s
3584	&32
3590	&30
3839	&28
Databases	&25
Domains	&33
John	&31
Mathias	&29
S.A.D.	&26
Sotirios	&27

Fig. 2.9: The atomic value index for the example source

2.4.2 Nonlinear Index Structures

Value based indices presented above have the advantage that they can exploit indexing techniques originally developed for relational systems. However, they only work over limited views of the data graph and require different query algorithms to be used for indexed and non-indexed data. The indices described in this section relate vertices of data graphs to vertices of index graphs. Since both data and index graphs can be expressed as digraphs, the generated indices can be managed using the same system used for the data graph itself.

Definition 2.12 (Index Graph, Extent): It is possible to construct an associated digraph $I(DG, S) = (V(I), A(I), ext)$ for a data graph DG and a given set of subsets of its vertex-set $S = \{s_i | i \in \mathbb{N} \text{ with } s_i \subset V(DG)\}$, whose vertex-set V(I) contains one vertex u_i per subset s_i in S. The set of vertices contained in s_i are associated with u_i by means of the relation $ext : V(I) \rightarrow \{V(DG)\}$ and is called the *extent* of the vertex u_i or $ext(u_i)$. If there exist at least one vertex v_i of $V(DG)$ in $ext(u_i)$ which has an arc to a vertex v_j of $V(DG)$ in $ext(u_j)$, $A(I)$ will contain an arc from u_i to u_j. The triple $I(DG, S) = (V(I), A(I), ext)$ will then be called the *index graph* of DG with respect to S.

Notice that this definition works on arbitrary sets of subsets of the vertices of the data graph. Thus the relation between vertices of the data graph and the vertices of the index graph must be neither injective nor surjective. This definition allows the creation of both redundant and incomplete index graphs based on arbitrary clusterings of the vertex-set. Chapter 3 will make use of this fact and present a general model of how to use this abstraction in a general optimisation approach.

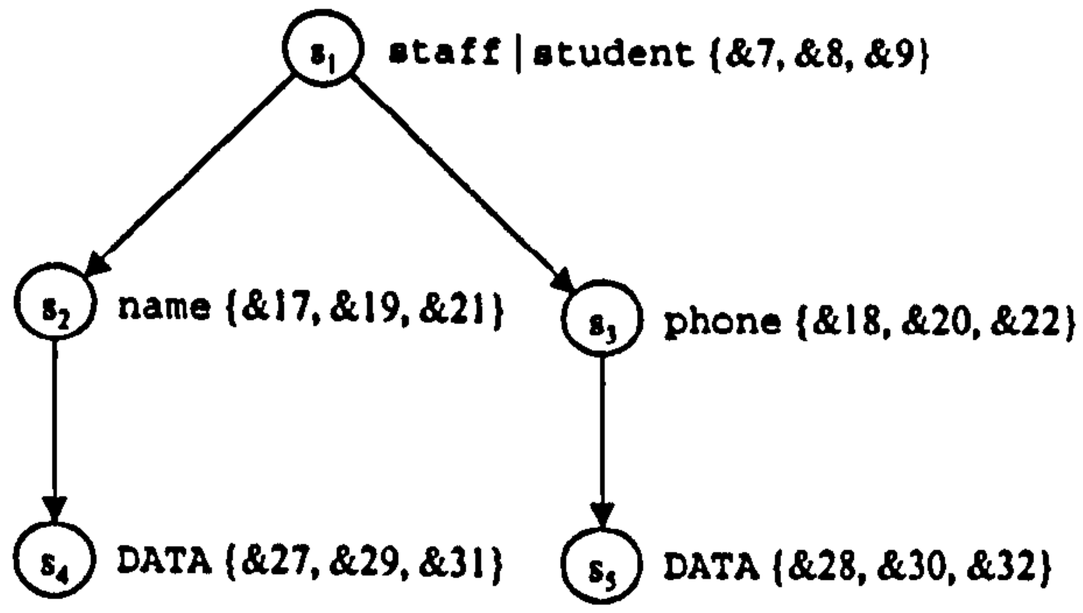


Fig. 2.10: The index graph of the data shown in Figure 2.5 for a set S of subsets with $S = \{\{\&7, \&8, \&9\}, \{\&17, \&19, \&21\}, \{\&18, \&20, \&22\}, \{\&27, \&29, \&31\}, \{\&28, \&30, \&32\}\}$

Figure 2.10 shows the index graph of the data graph shown in Figure 2.5 with respect to a set S of subsets, which were designed to include all information stored about people. In this case it was manually designed to abstract away from the differences in tag names used for `staff` and `student` entries.

However, as Chapter 4 will show, such indices can be automatically derived based on mathematical properties of the data graph. An important special case of this general definition is based on equivalence relationships between the vertices of a data graph. If the members of every equivalence class are used as the subsets of S, the set of subsets becomes a partition of the vertex-set V(DG). Consequently the relation between vertices of the data graph and vertices of the index graph becomes a surjective mapping, which defines an endomorphism of DG on $I(DG)$. Because every vertex of the data graph can only belong to a single equivalence class, the size of $I(DG)$ is limited by the size of DG in this case.

2.5 Literature on Semistructured Data Processing

This section reviews literature that is concerned with SSD management and querying in general and in particular the associated summarisation techniques that motivate the proposed concept of data groupings. More specific literature on individual topics of SSD processing will be discussed in the individual chapters of this thesis as it becomes relevant. In particular Chapter 5 will look at compact representation of SSD and Chapter 6 will look at indexing and querying mechanisms.

2.5.1 Semistructured Data Management Systems

In this part, systems and approaches for SSD management are reviewed. The focus is based on the range of problems associated with SSD processing addressed by an approach rather than on individual features of a specific implementation.

2.5.1.1 Lore, OEM and Lorel

Lore (for *lightweight object repository*) of Stanford University is the most comprehensive and exhaustive academic experimental data management system designed for SSD. Its main components are the *object exchange model* (OEM) that acts as data model, storage management, the query language *Lorel* (for Lore language), a number of indexing mechanisms including structural summaries in form of *DataGuides* that describe the source schema, query optimisation and evaluation engine and user interfaces. The Lore system has been described both in terms of its general architecture [MA+97] and details regarding its individual components. Only those parts that are of outstanding importance for the presented work are reviewed here. These components include the data model OEM, which was taken from the related *Tsimmis* project [PGMW95] and parts of its query language [AQ+97] and evaluation engine [MW99]. Its indexing mechanisms [GW97] will be reviewed in Section 2.5.2.1.

The OEM [PGMW95] is an SSD model that can be viewed as an arc-labelled, directed graph. The vertices in the graph are called objects, which have an unique object identifier (OID) and are either complex or atomic. Atomic objects have no outgoing arcs and contain values from one of a list of predefined atomic types. Complex objects can have outgoing arcs, their value is a set of (label, subobject) pairs. The label describes the relationship of the object with its subobject. Finally the OEM data model defines names, which represent entry points to the graph and serve as aliases for particular objects.

The original data model was slightly modified with the advent of XML as a generally accepted representation of SSD [GMW99]. Essentially this meant that the original arc-labelled graph was replaced with a vertex-labelled graph. This difference, however, is irrelevant for most scientific issues surrounding the management of SSD. Another more important mismatch between OEM and XML is the fact that subelements in XML are ordered whereas subobjects in OEM are not. Inversely OEM contains a concept of object identity whereas there is no such

concept in XML. These complementary concepts were discussed in Section 2.2.3 already.

Lorel [AQ+97] is essentially an extension of the *object query language* (OQL) [Cat94], following the SELECT-FROM-WHERE syntax of the relational SQL language. It extends relational query languages in two directions. Firstly it allows path expressions in the FROM part. These expressions can contain regular expressions in order to deal with irregularities and lack of knowledge of the structure of the data. Secondly it incorporates a multitude of type coercion mechanisms since type information in SSD is often incomplete or absent. Thus typing issues must be addressed at the time of querying rather than at the time of data definition.

2.5.1.2 STORED

Deutsch, Fernandez and Suciu [DFS99] describe an approach to SSD management that is entirely based on re-use of existing technology and algorithms. The core of their system is a purpose designed query language called *STORED* (for semistructured to relational data), which allows the specification of bidirectional mapping rules between a semistructured data graph and a relational representation. A core hypothesis of this paper is that most semistructured databases contain a regular core that can easily be managed in a relational system. Data that is not contained in this regular core must be maintained by an SSD management system, i.e. some persistent graph repository. However such data is expected to be of much smaller quantity and thus less critical in terms of performance. This hypothesis is supported by their experiments and justifies their approach of re-use rather than re-engineering.

The actual query language is surrounded by a number of algorithms that perform the translation between the SSD model and the relational core plus overflow graph. These algorithms generate appropriate mappings based on either an instance of data or a selection of representative queries. They also allow reconstruction of the original SSD on demand and provide facilities for the translation of queries over the semistructured instance into equivalent queries over the generated storage format.

2.5.1.3 SilkRoute

A similar approach based on mappings between semistructured and structured representation through the use of a purpose designed query language is employed by the SilkRoute project [FTS00]. The important difference here is the different perspective of the scenario. Whereas STORED was designed to allow the management of SSD in a relational store, SilkRoute aims to allow access to a relational store by means of a semistructured query language. Fernández, Tan and Suciu describe a mapping language called RXL (for *Relational to XML Transformation Language*) that is general enough to allow the transformation of relational data into an arbitrary semistructured format as governed by an external schema in form of a DTD. This semistructured view of the data is never materialised but used to translate application XML-QL queries into equivalent RXL and SQL queries, which can be executed over the relational store.

2.5.1.4 Production-state XML Databases

A number of so called *native XML databases* (NXD) have been developed in order to satisfy the demands of XML based applications. The systems described here are prototypical for a number of different approaches.

Tamino [SW00, Sch01] developed by Software AG combines a conventional relational database with a purpose-built semistructured database engine. Modules are provided for the seamless integration of both parts. This is the only commercially available database that is solely aimed at SSD. Other commercial database vendors are providing extensions to their (object) relational databases, which typically map the SSD model to that used by the database.

Xindice [Sta01a] is an open source database that is essentially a management system for small XML documents. XPath and other querying mechanisms are provided. However, given its target application of content management, its focus lies on fast delivery of entire, typically document-centric XML documents, rather than the answering of arbitrary data-centric queries.

eXist [Mei02] on the other hand is based on a labelling scheme (cf. Section 6.3.1) that allows decisions to be made about ancestor/descendant-relationships between vertices without the complete traversal of the connecting path. Among

other interfaces it provides a XQuery engine. It is thus well suited to answer both data-centric and document-centric queries.

2.5.2 Summary Structures or Indices of Semistructured Data

Index structures allow direct access to individual vertices of a data graph. This is crucial for efficient processing of SSD, as tree and graph traversal algorithms are expensive in terms of I/O performance since every vertex visited could require a disk access. The aim of indexing is to find summaries of the data that can be held in main memory to avoid disk access costs.

2.5.2.1 Representative Objects and DataGuides

Nestorov et al. [NU+97] were first to identify the need for partial schema information as a means of data summarisation for SSD. Their work is based on the OEM data models and aims to discover a schema for a given instance of hierarchical data. The full representative object (FRO) essentially comprises all possible paths in the instance data. Degree-k representative objects (k-RO) only contain paths up to a certain length k. They are consequently less expensive to compute and store but are only correct for path queries up to this length. FROs can be seen as nondeterministic finite automata (NFA), thus standard algorithms for determination and minimisation can be applied to gain a minimal FRO.

DataGuides [GW97] are used in Lore as a path index structure. They are essentially an FRO, i.e. they contain every path starting from the root of a data graph in the data instance. Duplicate paths are only represented once, making the DataGuide considerably smaller than the data instance if that instance contains a regular core. The vertices of the DataGuide contain references to the vertices accessible by the equivalent path in the data instance and serve as a structural index. They are utilised to guide query-by-example operations. Additional metadata, e.g. statistics about value distributions, can also be stored in these vertices in order to aid query processing. These techniques only consider linear path expressions, that is forward path expression from the root or a given context to the vertex sought.

2.5.2.2 1-Index, 2-Index and t-Index

In order to address more complicated, branching path expressions, the more general approach developed by Milo and Suciu [MS99] can index more complicated relationships between vertices of a data graph than a single, linear incoming path. Index coverage is decided by user-provided templates and may be derived from a typical query load. Consequently indices are dependent on at least partial knowledge of the data source and its applications. However, due to this use of a priori knowledge, the index generation is more efficient and linearly bounded in space by the document size. Apart from the general template based indices (t-Index), two special instances are investigated. The first, so-called 1-Index is identical to the DataGuide described above for tree data, i.e. it aggregates all vertices with an identical incoming path. The 2-Index catalogues all relationships, i.e. path expressions, between any two vertices of the data graph.

2.5.2.3 Identifying Structure by Sharing Common Subtrees

Whereas DataGuides summarise vertices with common ancestry, thus allowing linear queries to be executed against the index graph, Buneman et al. [BGK03] follow the opposite intuition. Their structural summary is based on sharing common subtrees, that is they identify nodes that have common paths from the context node to (but excluding) the data leaves and thus support branching path expressions. Their approach is based on the concept of bisimilarity, which is taken from an approach adopted from symbolic model checking. The generated summary structure can be used to resolve all XPath location steps in linear time, resulting in query times exponential in the the size of the query but only linear in respect of the size of the compressed data. Note that due to the strong adaption toward XPath, the system is heavily dependent on document order and only deals with trees.

2.5.2.4 Covering indices for branching path expressions

Kaushik et al. generalise the problem, first for linear paths on graph structured data [KS+02] and then for generally branching path expressions with forward and backward axes [KB+02]. As Buneman et al. they base their approach on bisimilarity but, coming from the graph theoretical side rather than having XPath processing in mind, they do not base their similarity relationship on document

order. One of their central observations is that a covering index graph for this problem class is often only little smaller than the data instance it is describing and thus does not improve the efficiency of the query evaluation. They parameterise their algorithm in order to allow more compact but non-covering indices to be generated. Their algorithm allows the specification of the maximal length of incoming and outgoing path expressions and tag labels to be included in its computation. It also allows the specification of the query depth, which is a measure of the inherent complexity of the query, essentially the number of "twists" between incoming and outgoing paths in the query graph. Their specific definition of bisimilarity and their algorithms used to compute it are used in several parts of this thesis and described in Appendix B, which is essential for understanding some of the advanced issues addressed by this thesis.

2.6 Summary

This chapter has presented a review of the motivation behind SSD models, its properties and query languages and processing systems. This thesis will make particular use of the data model developed throughout Section 2.2 that describes a semistructured data source as a vertex labelled, digraph (Definition 2.9). One important property of this graph is that it exposes two possible views, a tree-view (Definition 2.11), defining a rooted distinct spanning tree, and a graph-view (Definition 2.10), allowing the representation of arbitrary data sources. Most of the techniques and work on SSD processing detailed in Sections 2.3 to 2.5 were described in terms of this data model.

3. AN OPTIMISATION MODEL FOR QUERY PROCESSING

Problem Decomposition

Divide et impera.
Divide and rule.

Niccolo Machiavelli, 1469 – 1527

Appropriate models are fundamental for understanding, comparing and contrasting the different approaches to query optimisation. Such a high-level model is described in this chapter, which explains how the concept of data grouping affects multiple aspects of semistructured query processing. The model will act as a high-level guide to this thesis, but can also be used in its own rights. To demonstrate this, Example 3.2 describes work performed by other researchers in terms of the model presented. Since its initial design [NW04] the model has been revised to better support this degree of generality.

3.1 Introduction and Model Overview

The wealth of research into SSD processing in general and XML in particular, has resulted in a large number of approaches for storage, indexing and querying. A selection of these representing different approaches has been reviewed in Section 2.5. Here a unifying model is presented as a framework for understanding the common core of these algorithms. The key idea behind this model is the assumption that most practical queries are based on a particular pattern of data, which can be deduced from the query and then captured using an optimised data structure amendable to efficient processing techniques.

To aid understanding, the problem of query optimisation is divided into four distinct steps, which are presented in Figure 3.1 and briefly described below:

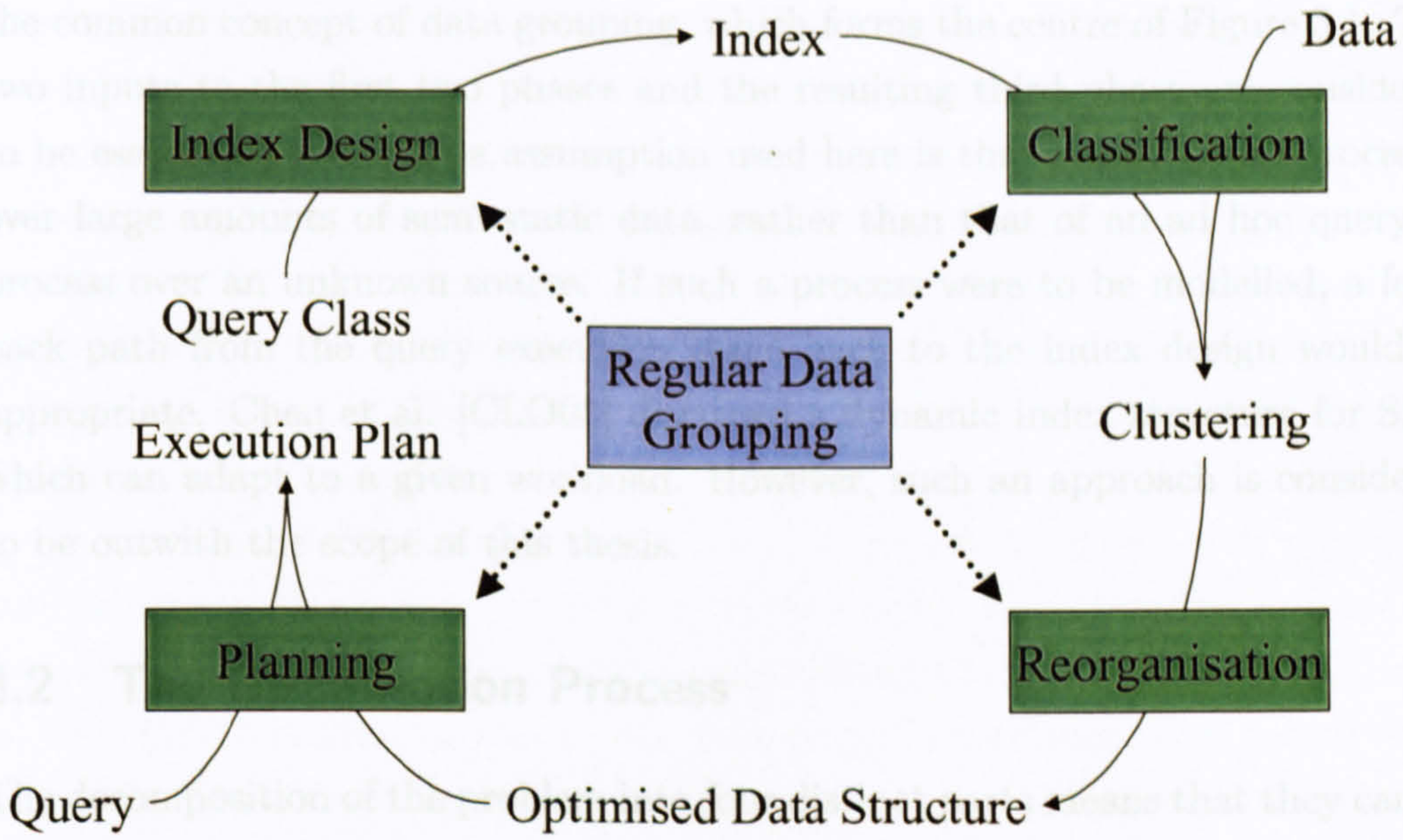


Fig. 3.1: The four phases of the query optimisation process

Index design: Given a particular query pattern or class, an index that supports its easy evaluation is designed. This step consists of the analysis of the expressive power of the query language, which is needed as an input parameter for this step.

Graph clustering: The vertices of the database instance are clustered according to the expressive power of the index. This exposes patterns in the data specific to the given instance, which is required as input parameter.

Data reorganisation: The discovered structure is used to optimise the storage of the graph structure including the data containing the atomic vertices. This transforms the graph into a physical representation suitable for the particular combination of query class and data instance.

Query planning: Based on the generated data structure and captured instance properties, a plan for the efficient execution of a particular query can be generated.

The representation of the process shown in Figure 3.1 is not closed, but contains a gap between the first and last step. The model thus describes a sequential

rather than iterative process. However, all phases of this model are influenced by the common concept of data grouping, which forms the centre of Figure 3.1. The two inputs to the first two phases and the resulting third phase are considered to be essentially fixed. The assumption used here is that of automatic processes over large amounts of semi-static data, rather than that of an ad hoc querying process over an unknown source. If such a process were to be modelled, a feedback path from the query execution stage back to the index design would be appropriate. Chen et al. [CLO03] discusses a dynamic index structure for SSD, which can adapt to a given workload. However, such an approach is considered to be outwith the scope of this thesis.

3.2 The Optimisation Process

The decomposition of the problem into four distinct parts means that they can be addressed individually. In general not all steps need to be present in a particular approach and many research papers address only some steps of the specified model. This thesis will present research on the three later phases of this model, which are concerned with the application of data groupings. The conceptually important logical classification of data is discussed in detail in Chapter 4 in terms of domains. The exploitation of this concept for physical organisation of the data is described in Chapter 5. Finally Chapter 6 analyses the impact that data groupings have on the performance of several query algorithms, thus aiding query planning. The initial process of designing indices from a given query language is addressed in this chapter only. The experimental system described in Chapter 6 will make use of the covering index for path expressions designed by Kaushik et al. [KB+02], which is described in Appendix B. The focal point of this thesis is how to derive and utilise data groupings gained from such an language theoretical analysis. The following sections describe the individual steps of the model in more detail.

3.2.1 Index Design

Given a particular query pattern or class, indices can be devised that support easy evaluation. Depending on the complexity of the query language, a combination of more than one index might be used. For example, a query such as finding an `author` named "Miller" can be resolved using a combination of a tag index

(Example 2.1), used to find all `author` vertices, and an atomic value index (Example 2.2), used to find all atomic vertices with a value of "Miller". The output of both index requests can then be combined, e.g. using an ancestor-descendant merge join algorithm as presented by Li and Moon [LM01]. Lore's link index (*Lindex*) or value index (*Vindex*) are examples of such conventional, linear index structures [MW+98]. Their text index (*Tindex*) is similar to a full text index or inverted list typically used in information retrieval systems. By contrast their path index (*Pindex* or *DataGuide*) belongs to the class of non-linear index graphs described in Section 2.4.2.

In the case of SSD, index graphs are an important example of structural indices. In general an index graph is generated using some transformation of the data graph. This kind of approach to constructing a covering index graph based on bisimilarity between nodes of the data graph is described by Kaushik et al. [KB+02]. Since the purpose of indexing is to reduce the amount of data that needs to be processed, such transformations usually result in a simplification of the original graph. However, for very expressive query languages such as XQuery or its constituent part XPath, it is infeasible to devise a covering index, i.e. an index that allows the resolution of all queries using the index alone. Even for simpler languages, such as the *branching path expressions*, introduced in Section 2.3.1, experimental results show that covering index graphs are often as complex as the data they are trying to index [KB+02]. Since such index graphs are endomorphic to the data they index, with the same query algorithms being applied to both, the original aim of simplifying the query execution is subverted. Ramanan [Ram03] analyses the relationships between different semistructured query languages and relates their expressiveness to the complexity of a minimal covering index structure. This helps to design indices for relatively small but important fractions of a given query language. In this case user queries need to be broken down to their constituent parts. These can then be addressed efficiently by such partial indices and the results produced by combining the partial results computed from them. An alternative approach is to use approximate indices, i.e. indices which are not covering for the given query, but are safe. In this case a query can still be executed against an index, but the results returned are a superset of the true result and thus require an additional verification step. Kaushik et al. [KS+02] proposes such an approach based on local similarity.

3.2.2 Data Classification

In this step the data instance, seen as a graph, is clustered by classifying the individual vertices with respect to the designed index. The early use of the data instance itself is a fundamental divergence from the approach taken in relational database systems, where statistical information on the cardinality of relations are estimated based on the physical database schema and used for query optimisation [Ioa96]. The model presented here requires the use of information contained in the data graph at an early stage in the optimisation process in order to discover patterns that occur in it. This is necessary due to the fact that even if schema information is available it often does not describe the relevant properties of the data source sufficiently. This is primarily caused by the fact that semistructured schema languages are based on regular expressions, allowing an unbounded number of concrete data structures for a given schema. A study in this field [Cho02] that divided a number of schemata into three categories, one for data-centric applications, one for document-centric applications and one for data exchange applications found instances of recursive DTDs, i.e. schemata that allowed elements as subelements of itself, in all these categories. It is also not uncommon to find XML documents, which do not comply with their associated schema, i.e. which are invalid with respect to it.[1]

The clustering itself exposes existing structures in the data instance, i.e. from all possible structures governed by the query pattern, only those occurring in the specific data instance are included in the further optimisation process. This forms an instance specific schema, which can be used for data exploration and query formulation [GW97], e.g. for querying by example [Zlo75].

An important example of such a classification is based on an equivalence relationship on the vertex-set of the data graph. If clusters are defined based on equivalence classes, the clustering is a partition of the vertex-set in this case, because every vertex belongs to exactly one equivalence class. Consequently the size of the clustering, i.e. the number of blocks of the partition, is limited by the number of vertices in the data graph. However, if an arbitrary classification process is used, i.e. if vertices from the data graph can occur in more than one cluster, the size of the clustering can be larger than the size of the data graph and is unbounded in general.

[1] One example of such an invalid document is the XML version of the Mondial database, which can be found at `http://www.cs.washington.edu/research/xmldatasets/www/repository.html`

3.2.3 Data Reorganisation

The patterns discovered by the classification process described in Section 3.2.2 can also be used to physically regroup the data, both the graph structure encoded using complex vertices and the atomic data represented by its leaves. This results in the creation of syntactically or semantically homogeneous domains. Note that the semantics of these domains are indirectly dependent on the initial choice of a class of queries, but not on the possible applications or computations to be performed over the result of such queries. This issue will be addressed further in Section 4.2, where the distinction between application dependent and independent domains will be discussed.

Based on the properties of the actual data management system, this results in a physical data representation that is appropriate with respect to the class of queries and the specific data instance. If the encoded source contained a highly regular core, it may often be useful to map this regular core to a relational system in order to make use of its functionality and performance. This is an approach that has been investigated thoroughly, e.g. by Deutsch et al. [DFS99], Florescu and Kossman [FK99], Shanmugasundaram et al. [ST+99] and Kudrass and Conrad [KC02].

If a significant proportion of the source is irregular in its structure, an approach based on graphs might be better suited than a flat representation. The clusters resulting from the classification can be used as the vertex set of a new graph, forming an index graph as detailed in Section 2.4.2. Such index graphs can be used as secondary access structures to a data graph [GW97, KB+02, BGK03] or extended to replace the data graph they index altogether. In this case they also represent a physical reorganisation of the initial data structure. If vertex references stored in the extent of an index graph as described in Definition 2.12 are replaced with a representation of the vertex itself, the index, i.e. a secondary data structure, mutates to a primary data structure. Such an approach is presented in Chapter 6.

3.2.4 Query Planning

Based on the generated data structure, specific queries of the general class can be evaluated efficiently if appropriate strategies are employed. This last step consists of finding algorithms that make use of the partial pre-computation, which has been effected by the previous data classification and reorganisation. The previous

steps can also generate valuable metadata about the source, especially a statistical source description, which can be used here by a query optimiser in order to choose an appropriate query strategy. Two important strategies for query evaluation are the *top-down* and *bottom-up* approaches, which were presented in Section 2.3.3. They can be effectively employed if the cardinality of subsets of vertices matching individual predicates of a query are known [FH+02] by starting the embedding and matching process at predicates with low selectivity.

Such query planning and optimisation processes have been studied in detail in the relational case [JK84, Ioa96]. Approaches that make use of conventional DBMS by mapping semistructured data into relations can make use of their query optimisers. Florescu and Kossman [FK99] followed such an approach and showed the influence different mapping strategies have on the query performance. In the semistructured case, research on query plan optimisation is still limited. McHugh and Widom described its concepts based on the example of Lore's DataGuides [MW99]. Kaushik et al. [KS+02] compares the influence of three different index graphs, their A(k)-index, the 1-index and the DataGuides, on the evaluation of linear path queries using a fixed query algorithm. The specific effects a particular class of data clustering has on a number of evaluation strategies will be discussed in Chapter 6 of this thesis.

3.3 Exemplary Query Processing Systems in Terms of the Model

In order to justify the approach chosen, three example systems are presented in terms of the model described in Section 3.2. Firstly, the relatively simple approach chosen for the initial research on compressing SSD [NW02] will be described in Example 3.1. This will be further detailed in Chapter 5. Secondly, the research by Kaushik et al. [KB+02] will be described in Example 3.2. Their work defines a covering index for branching path expressions based on bisimilarity. The last example emphasises the generality of the model (Example 3.3).

All example systems will be presented using the same data source, whose data graph is shown in Figure 3.2. It represents a bibliographic database containing information about publications and people associated with them.

Example 3.1 (Label-Value Predicates): To aid comprehension of the model a simple class of queries is assumed first. The query class covered can be used to ask for

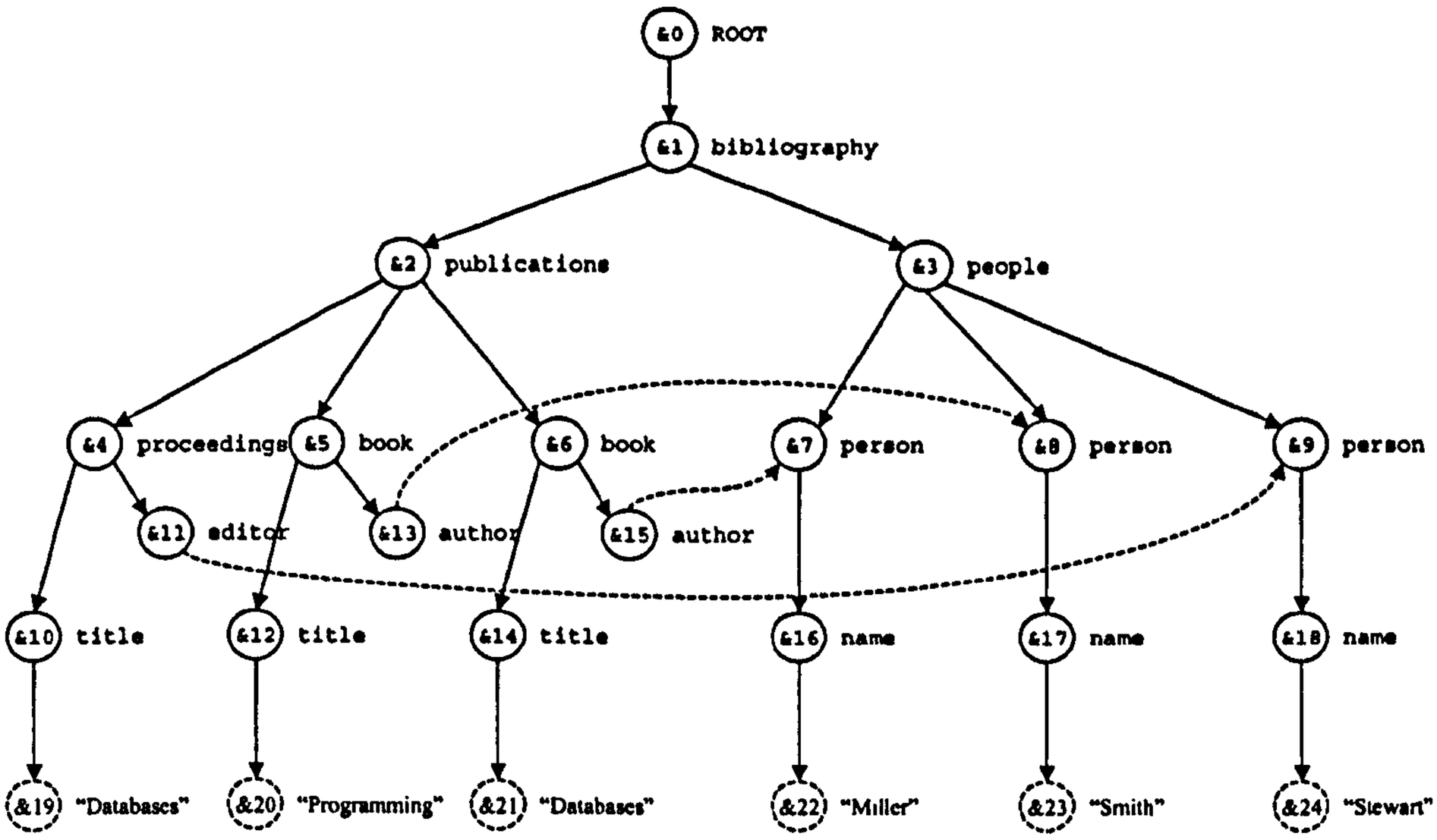


Fig. 3.2: The bibliographic database used for the example systems

vertices connected to an atomic vertex representing a particular value through an outgoing arc, i.e. the path pattern

Query 3.1: $//\$label[/\texttt{DATA} = \$value]$

where $\$label$ and $\$value$ are variables.

To resolve such queries efficiently, a two-dimensional index $I(\$label, \$value)$ is needed in order to locate all complex vertices that have outgoing arcs to atomic vertices.

Note that the resulting clustering is partial in general, as it indexes complex vertices only, and can be overlapping if several atomic vertices are adjacent to a single complex vertex, i.e. several index entries can refer to a single vertex of the data graph.

However, the resulting structure will only index existing vertex labels. Within these entries storage will be allocated only to data atoms that can be adjacent to such a vertex in the source. This results in a maximal size of the index of $|\Sigma| \times |V^A|$. The physical data storage can be realised as, for example a two-dimensional sorted list shown in Figure 3.3.

Querying then only requires executing two binary searches over the dimensions of these lists, in which a number of the identities of vertices in the original data graph can be stored and returned. The resulting data structure and query

algorithms are reminiscent of those described in Chapter 5, which is primarily concerned with compression of SSD. However, seeing this work as a query problem for the anticipated class of label-value predicate queries, one can transfer the techniques used there to the optimisation model presented here.

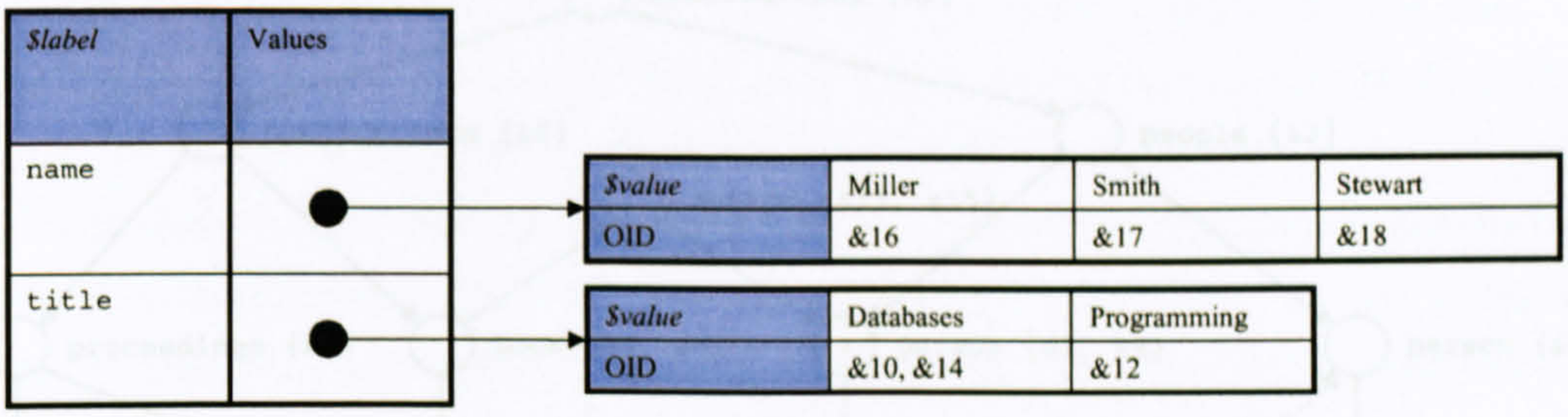


Fig. 3.3: The index structure used to resolve label-value predicates

Example 3.2 (Branching Path Expressions): The optimisation model will now be used to describe a more complex class of queries, in this case structural branching path expressions as described in Section 2.3.1 and analysed by Kaushik et al. [KB+02].

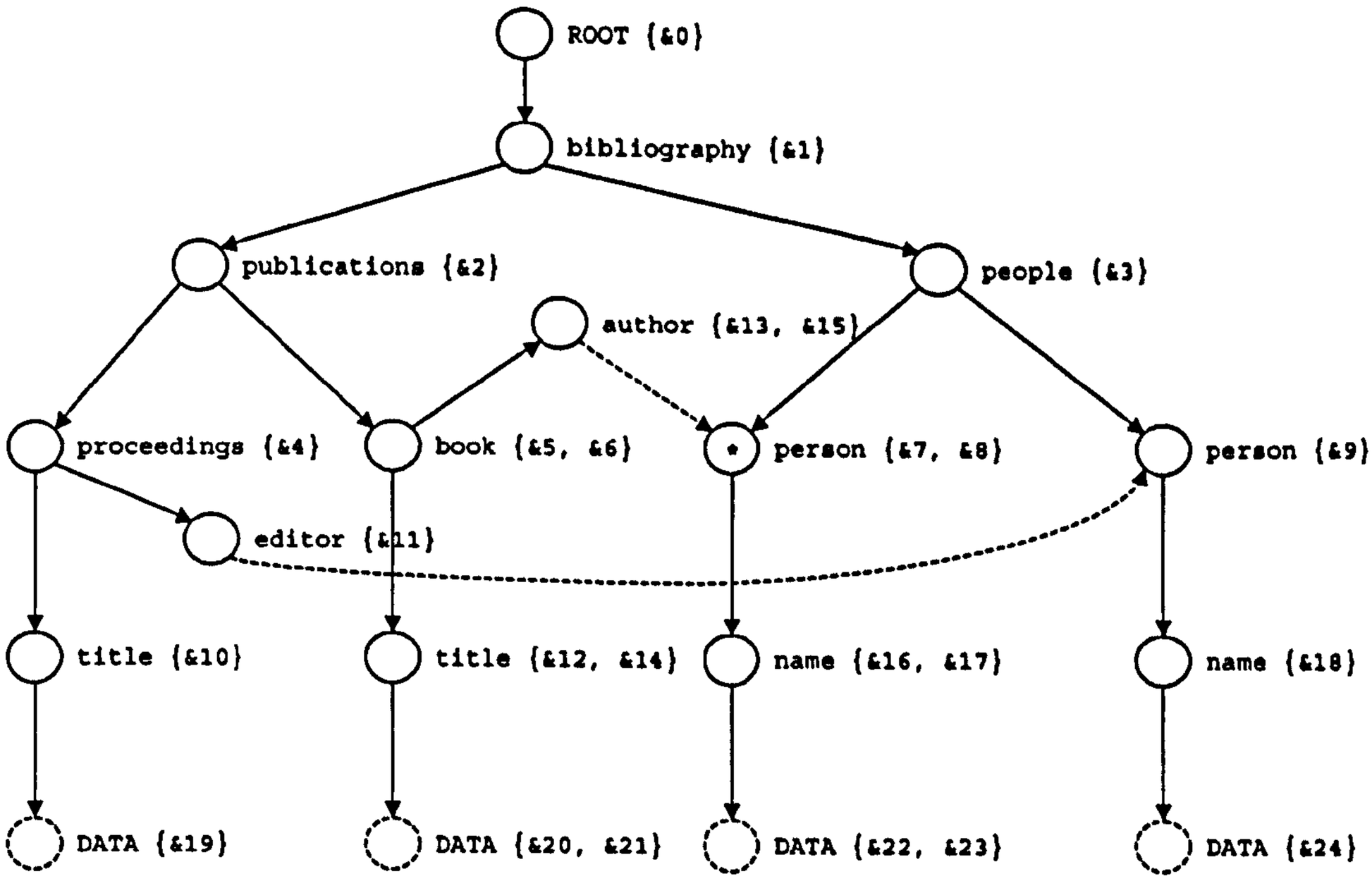
Figure 3.4 shows the covering index graph and an example query of this class. Due to the higher expressiveness of the language the reader is referred to the original paper for the derivation of the covering index. Appendix B summarises its mathematical background and the algorithms used for its computation, which are essential for understanding this example.

The classification of the data is based on bisimilarity between the vertices of the graph, i.e. if they carry the same tag label and are reachable by an identical set of incoming and outgoing paths. Details about this form of graph classification is given in Example 4.7 in the context of domains of SSD.

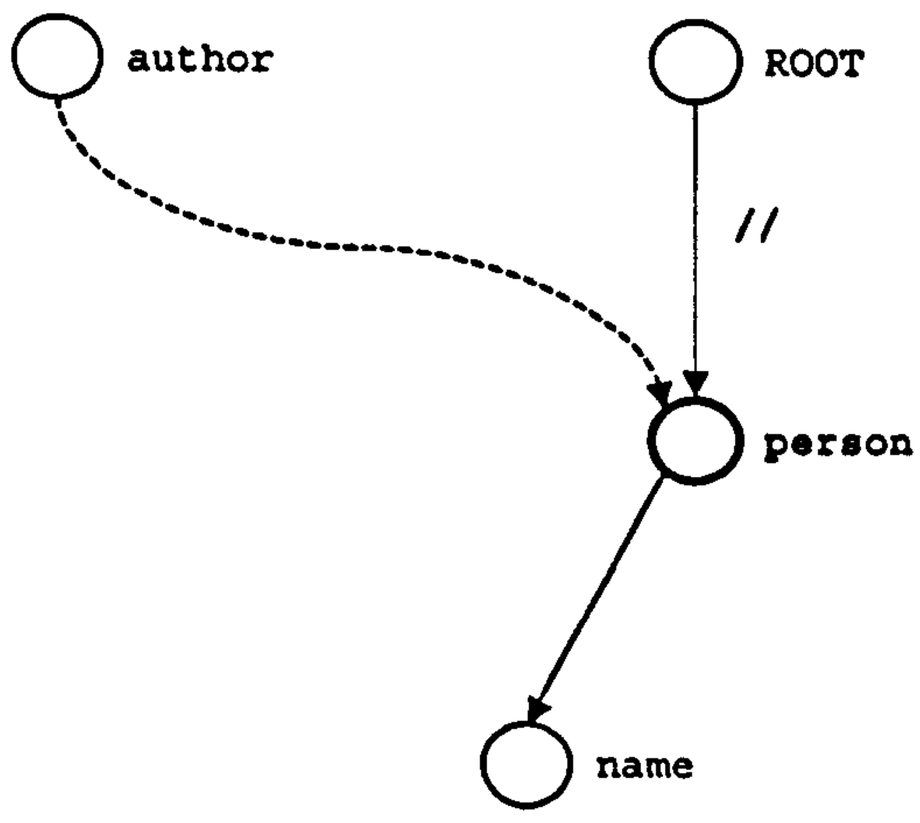
The data reorganisation follows immediately from this classification. Each equivalence class forms a vertex in the index graph created, with references to the vertices of the original data graph being stored in the extend of the index vertices. This is shown in Figure 3.4(a).

Because the index is covering for all branching path expressions, this index graph acts as a primary data structure for answering queries of this class. An example query from this class is Query 3.2, which is shown in Figure 3.4(b).

Query 3.2: `//person[/name & ⇐author]`



(a) The F&B-index graph of the example source



(b) The query graph of the path expression //person[/name & ⇐author]

Fig. 3.4: Branching Path Expressions: The covering index for the example source and a query from this class

Because the index graph is an endomorphism and bisimilar to the data graph, one can embed the index graph directly into the query graph shown in Figure 3.4(b) in order to resolve this query. This works in the same way as in the case of embedding the original data graph into the query graph, with only a reinterpretation of the results required to obtain the required outcome. Rather than returning the vertices of the index graph, which fulfill the query predicates, in this case the single vertex marked with an '*' in Figure 3.4(a), now the union of their extents forms the answer to the query, which is the set containing the two vertex references &7 and &8. Notice that the original research focuses on structural queries over graphs, a completely different class than that presented in the last example. Nevertheless all steps of the optimisation model can be successfully identified in this work.

Example 3.3 (Tree Depth Queries): The last example is presented to show the generality of the approach. It shows what happens if the optimisation process is started based on an arbitrary class of queries. The designed query system should efficiently support a request for nodes at a particular level of the tree-view of a given data graph irrespective of the vertex type or its tag label. A query for nodes at level two of the tree hierarchy looks like the following as a branching path expression:

Query 3.3: /*/*

A covering index for this query class can be constructed from a data classification based on the level of the nodes in the distinguished spanning tree. Notice that this partitions the entire node-set into equivalence classes based on their tree level.

Physically this can be represented using either a conventional value based index or as an index graph similar to the one described in Example 3.2. Figure 3.5 shows the example graph with the associated blocks of the partition (Figure 3.5(a)) and the resulting index graph (Figure 3.5(b)). Note that the computed index graph always has the same shape, a tree consisting of vertices of degree one, i.e. a linked list, which has the same depth as the original tree-view of the data graph.

For the conventional index structure, the length of the query needs to be computed, with the result being used as the key for an index look-up. For the index graph shown in Figure 3.5(b) the query can be executed by embedding the

index graph into the query graph shown in Figure 3.5(c) starting at the root of the index. In both cases a list of vertex identifiers stored as value of the index entry or in the extend of the index graph respectively forms the result of the query, in this case the vertex identifier &2 and &3.

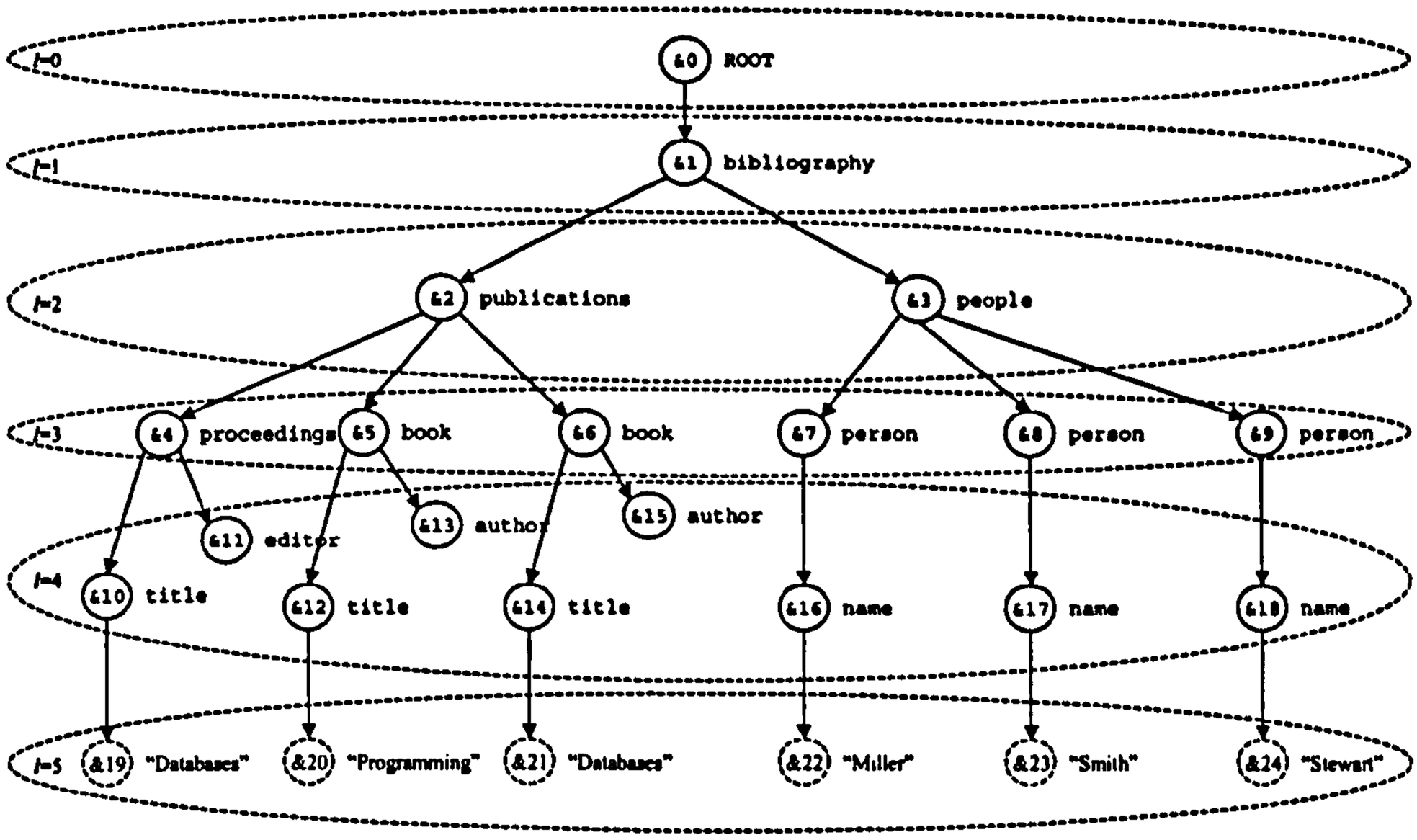
The index designed could also be used for other types of queries, e.g. to find all vertices within a set distance of a given context vertex. Note that the index is no longer covering for this problem class, as not every vertex in a block corresponding to a particular depth in the tree-view is guaranteed to have children in the following equivalence class. The `editor` node, for example, has level four in the tree-view, but is not adjacent to a node at level five. However, the index graph shown in Figure 3.5(b) is still useful to produce a reduced candidate set, whose entries need to be validated at a later processing stage. Clearly this is not a particularly useful class of queries and thus processing mechanism. However it serves to underline the generality of the approach, i.e. that an arbitrary query class can be used to derive an index implying a data classification, which can be used to physically reorganise the data in a way amendable to efficient processing strategies.

3.4 Summary

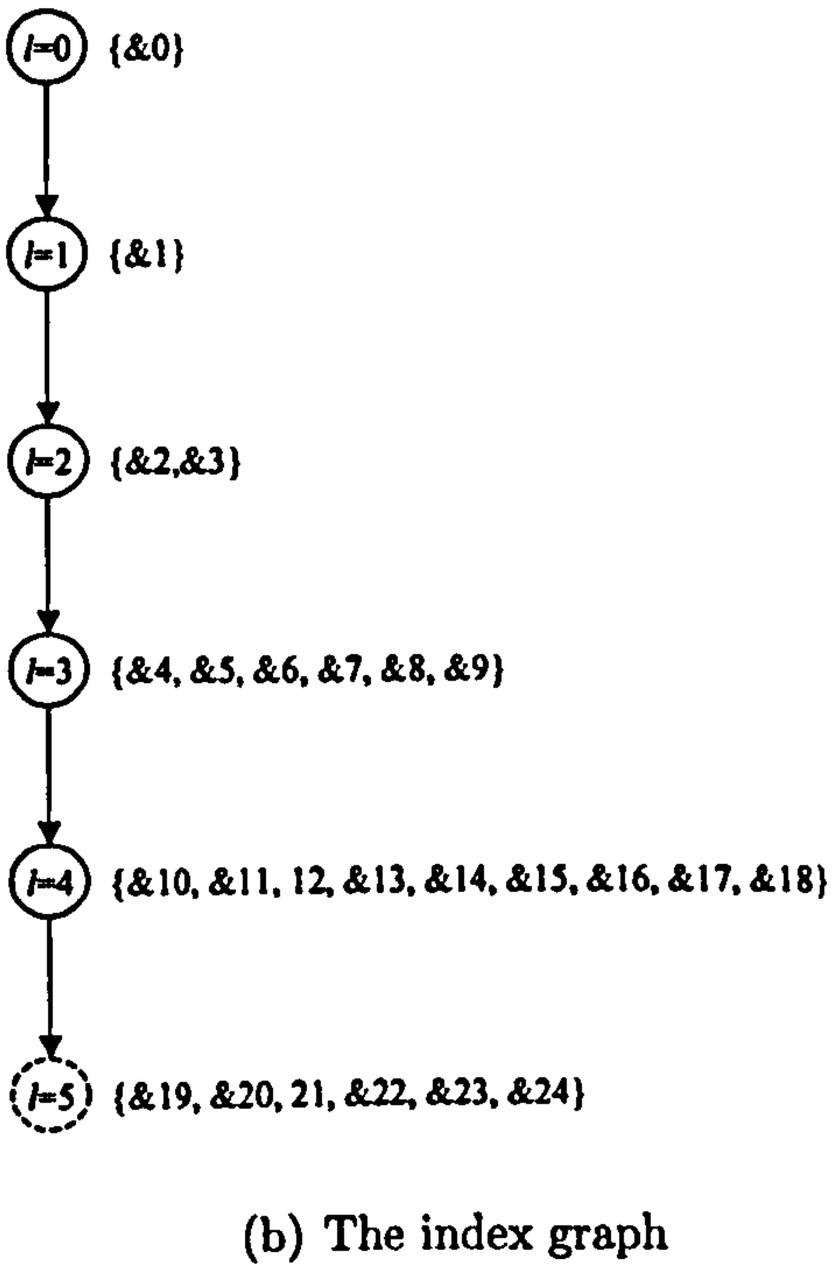
An abstract four step model for the general query optimisation process for SSD management systems was presented in this chapter. To the best of the author's knowledge, no such model exists for the semistructured case yet. At the same time it differs significantly from models known for the relational case, as its optimisation is based on data instances rather than schema information.

The core concept utilised by all phases of this model is that of data grouping or, with respect to the data graph model used in this thesis, that of graph clusters. This is the central aspect of the optimisation model depicted in Figure 3.1. At first an index is designed to suit a particular class of queries. Then the atoms of the data source, i.e. the vertex-set of the data graph, are classified with respect to this index. The generated data clustering can be used to reorganise the data physically in order to match optimised query algorithms.

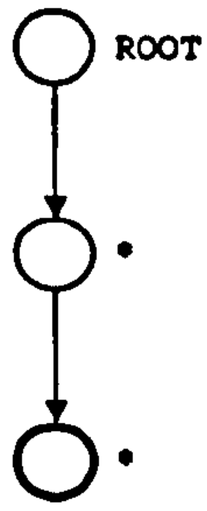
The model described in this chapter was illustrated using three very different examples. Example 3.1 presents the author's work on data compression in the context of the model and Example 3.2 describes research on covering indices for structural queries by Kaushik et al [KB+02]. Finally, Example 3.3 describes



(a) The tree-view of the data graph and overlaid blocks based on node level



(b) The index graph



(c) The query graph

Fig. 3.5: Tree Depth Queries: A data source, its covering index graph and an example query from this class

an entirely hypothetical query problem. All these examples can be successfully explained in terms of this model, emphasising its generality.

The next chapter examines the theoretical foundations of data classification in general, whereas the successive chapters will make use of particular classifications. Chapter 5 shows the impact a particular data grouping has on efficient data storage using the example of compression. Finally, Chapter 6 establishes a framework in which the effects of different data groupings on the query execution phase become obvious.

4. DOMAINS IN SEMISTRUCTURED DATA

Domains and Databases

"Domains effectively give us a vocabulary –
the things we can talk about in our database."

Chris Date, *DBMS Interview*, October 1994

The model presented in the previous chapter has emphasised the need to cluster data carrying similar semantics in order to aid processing. Similarity was defined in terms of the expressive power of a class of queries. In this chapter the idea of data clusters will be formalised into one of domains and it will be shown how some exemplary definitions affect a number of data sources.

4.1 Introduction to Domains in Databases

Domains are a concept of the relational model. By contrast to the rigorously structured case however, the concept of domains in SSD is much harder to capture. The chapter starts by looking back at the origins of data grouping for the purpose of compression before a new concept of domains based on graph theory is developed. It will be shown that this concept naturally arises from the interpretation of research on indexing SSD.

4.1.1 An Information Theoretical Approach

Although the resulting data clusters are presented in the context of database research and called domains, the foundation of this work really originates in information theory. Information theory describes an information source as a sequence of symbols from a fixed, finite alphabet L. The probability of the occurrence of a specific symbol can be derived from the statistical properties

of the source. A simple case is that of an order-0 Markov sources where each symbol has a fixed probability, which is independent from the context in which it appears. However, neither the assumption of a single source nor that of the independence of the context holds in case of structured data. Different columns in a database table are populated with values from different sources or domains as they are called in database theory. However, the values from different rows or tuples, which occur in the same column, are drawn from a single domain. As a consequence, structured data must be modelled as the output of a collection of several, potentially interdependent sources. As SSD contains by definition at least some structure, the same applies. Formally the SSD model does not bound the occurring values, i.e. they cannot be modelled as originating from a finite source. However, this limitation has no relevance for the work presented in this thesis as the presented definitions are only concerned with instances of SSD documents, which are finite.

XMill, an early XML compression system [LS00], uses the homogeneity of data atoms occurring in the same context in order to improve compression. It models an XML document as the output of $1 + k$ sources $A, B_1, \ldots, B_k$, with alternating symbols from A and one of the B_i, where i is defined by the preceding symbol from A. A is the set of tag labels of the document and the B_i emit the atomic values occurring immediately after the associated tag label indexed by i. Liefke and Suciu [LS00] use this model to reorganise their document into homogeneous containers according to these sources. Their observation is that a document transformed in this way compresses better than the original using standard compressors such as *gzip*. No further discussion of their compression mechanism will be presented here beyond the observation that in performing it they found a first possible definition of domains in SSD. Their mechanism detects domains given by the direct parent of an atomic vertex and uses a single global domain for all structural information. This is in fact a derivation of domains by parent vertex, which is introduced in Example 4.2 in Section 4.2.1.

4.1.2 A Graph Theoretical Approach

A similar concept can be developed in terms of graph theory, as shown in the query optimisation model of Chapter 3. Interpreting a semistructured database as a graph as detailed in Section 2.2 allows mathematical concepts developed for graph theory to be directly applied. The objects of interest in this interpretation

are sets of the vertices of a data graph and, more specifically, partitions of its vertex-set. For the discussion here, a one-to-one relationship between subsets and domains will be assumed, that is every subset implies exactly one domain and vice versa. In this case, graph clustering algorithms based on the properties of the graph's vertices can be used to define the domains in which the vertices reside.

In general, these subsets or domains will neither be disjoint nor cover the vertex-set, thus a vertex of the data graph can belong to zero, one or several domains. This is an important difference to the structured case, in which every value belongs to exactly one domain that is fixed at the schema definition stage. However, special cases will be derived in Section 4.2.1 for situations in which every vertex belongs to at least one domain and cases where each vertex belongs to exactly one domain. In the latter case the set of subsets is actually a partition and its subsets are the blocks of this partition. This is an important case that can be based upon an arbitrary equivalence relationship on the vertex-set, with the members of each equivalence class being assigned to one block of the partition.

Examples of domains based on properties of a graph will be detailed in Section 4.2. That section will also provide a mathematical definition (Definition 4.2) of the special case of domains based on a equivalence relationship, which can be used to define index graphs (Definition 2.12).

4.1.3 Motivation for the Identification of Domains

The reason for identifying data domains in SSD is threefold. Firstly, domains conceptually bridge the gap between application semantics and data syntax. Domains, once established, attach semantics to pieces of data, thus transforming raw data to useful information. The actual semantics in this context might be defined by an application, a query language or some intermediate storage model, which implies certain assumptions about the data. Secondly, domains can be used as a valuable tool for data optimisation. They can capture regular aspects of the concrete source of a given application, which are hidden by the generally unrestricted SSD model. These regular fragments can then be handled using existing regular mechanisms, which are well-studied and optimised in terms of efficiency. Thirdly, domains can also be seen as the output of a pre-computation. As data items are clustered in accordance with some common properties, the knowledge of these properties is available at no cost for future operations on this data and does not

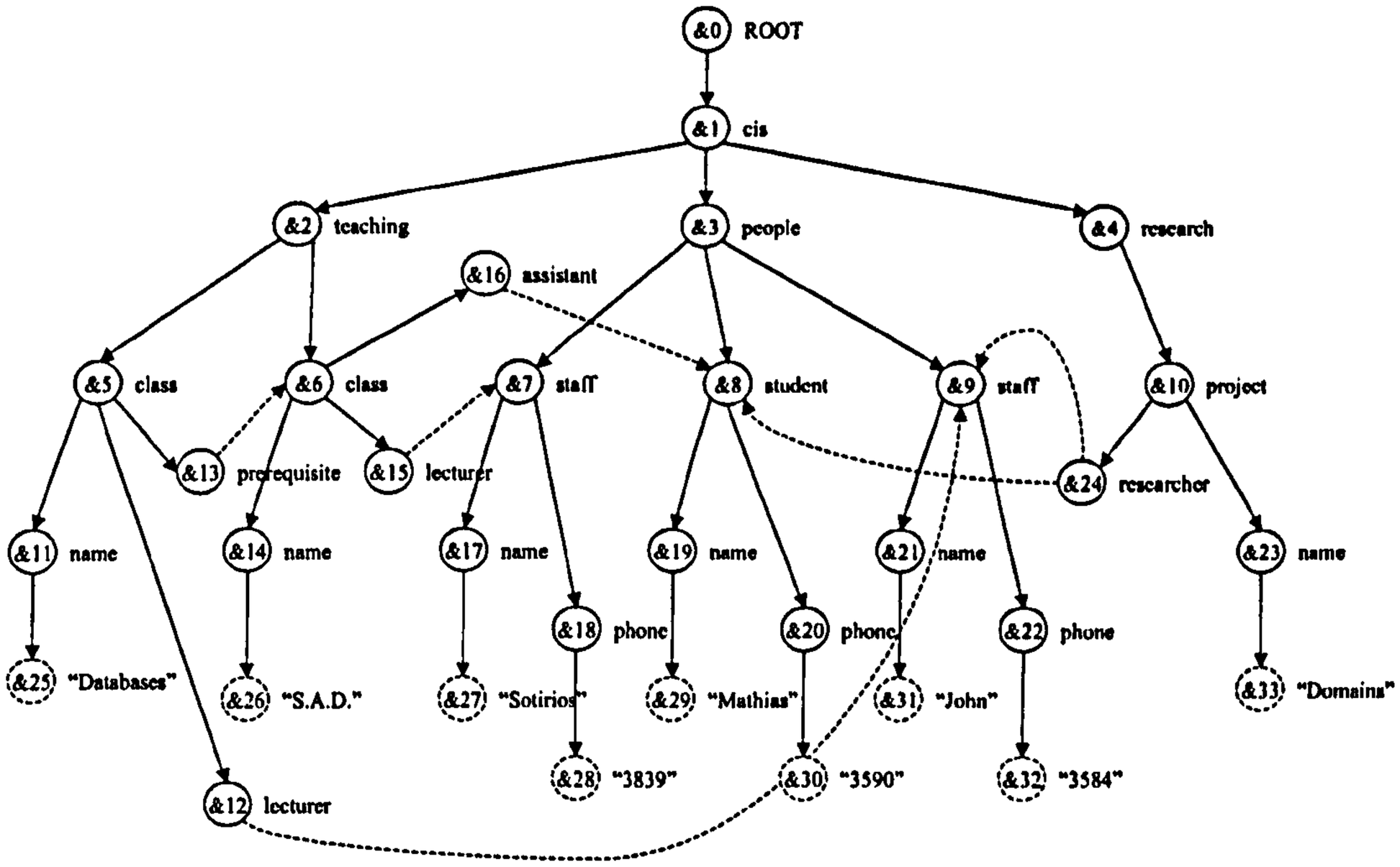


Fig. 4.1: The data graph representation of the example source (reproduction of Figure 2.5)

need to be computed again. In a write-once, read-multiple times scenario this in itself might result in improved efficiency.

4.2 Definitions of Domains for Semistructured Data

The dependence of domains on application specific semantics implies that there exists a multitude of definitions for the concept of domains in SSD. In general domains for SSD can only be determined with respect to a specific application, but not from the data itself. Note that in the case of a particular query language, the application is that of a query processor, replacing the application semantics with those of the query language itself. In structured data management, domain semantics are fixed during the schema design stage. For SSD the schema design process often occurs at a later stage and can be omitted altogether. This seems to contrast with the term "semantic web" [BLHL01] often used for self-describing data formats such as XML. However, this term is usually used in the context of a fixed application that implies the required semantics.

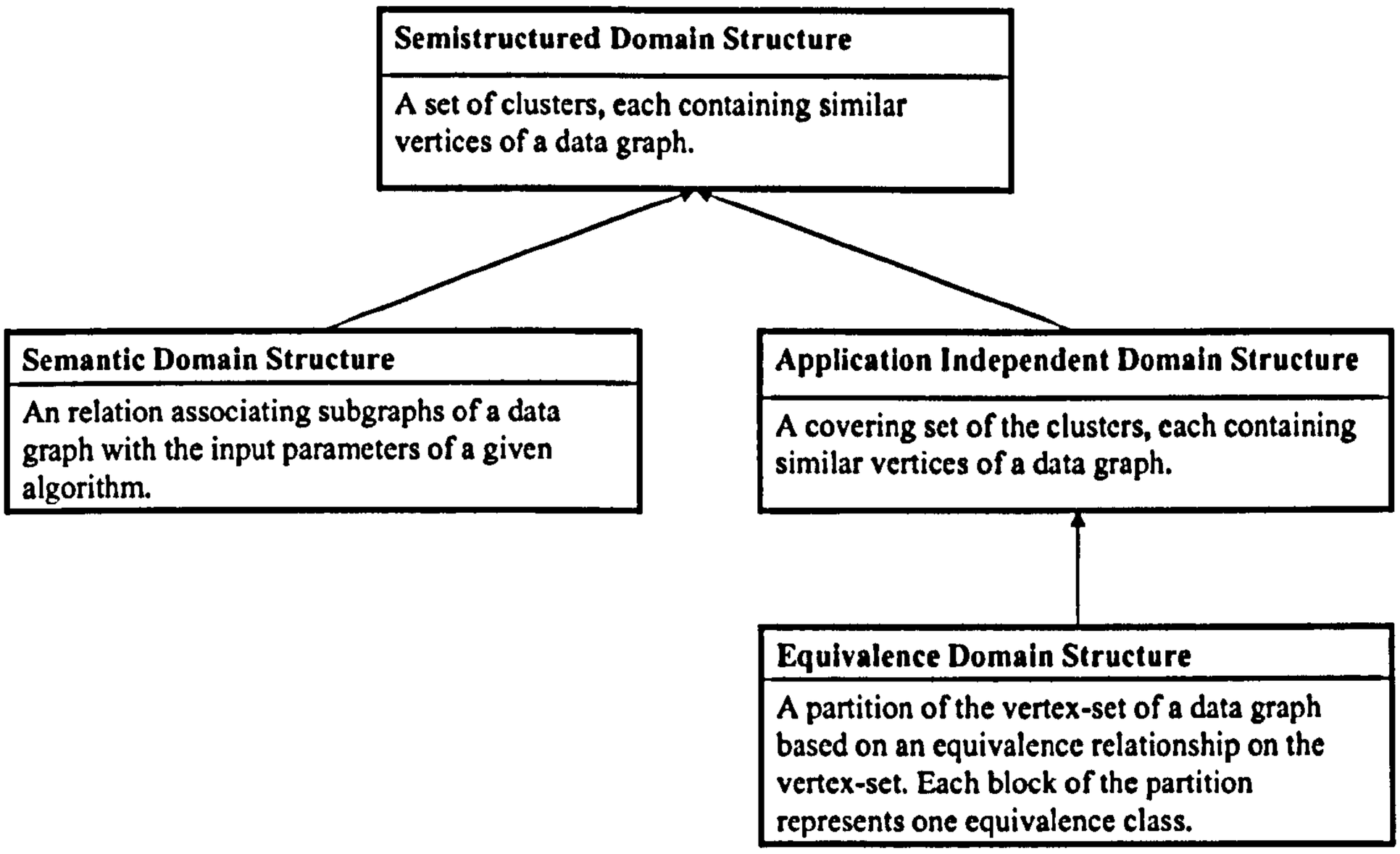


Fig. 4.2: The taxonomy of domains presented in this chapter

This section will introduce eight different definitions of domains for SSD. Using a common example the impact that different definitions have on the domains identified will be shown. The data graph representation of the example source is shown in Figure 4.1, which is identical to the data graph of Figure 2.5. It shows some information about the Department of Computer and Information Sciences of the University of Strathclyde. Some of the example definitions presented below result in a different domain structure depending whether only the set of tree-edges (solid) or all arcs of the data graph will be used. This corresponds to the duality represented by the graph- and tree-views introduced in Definitions 2.10 and 2.11 respectively. As in previous chapters the terminology will shift from vertices to nodes and from arcs to edges in case of the tree view. Notice that the chosen example graph is acyclic, which bounds the number of occurring domains specified by the following examples to a finite number.

4.2.1 Application independent domains

Despite the dependence of domains on semantics, a concept of domains which is independent of any specific application or query language is developed here. Instead of application semantics the concept presented in this section will be

based solely on the graph representation of a given source. Thus the semantics of the domains are based on graph properties. Note that such domains can only be approximate for applications that are not based on the graph's property used for its definition and require subsequent reinterpretation through the specific application in order to bridge this semantic mismatch. If for example the data about classes presented in Figure 4.1 was classified into such classes having an incoming arc from a `prerequisite` vertex and classes that do not have such an arc, an application merely printing the name of all existing classes would have to join these two distinct domains. Conversely, if the vertices of the graph were grouped by their tag label alone, but the application was to display only classes without prerequisites, the domain of classes would have to be restricted, e.g. by individually validating its members before the output can be produced.

Because the data requirements of the application are unknown at the time the domain structure is computed in this scenario, it is important that all data present in a given source is mapped to at least one domain. Thus the set of subsets of the vertex-set corresponding to the domains will define a cover of the vertex-set in general. Before the set of example definitions is presented, a few more terms required to explain them are introduced here. The relationships between the different kinds of domains introduced throughout this chapter are also shown in Figure 4.2.

Definition 4.1 (Application independent domain structure): A covering set C of subsets c_i of the vertex-set V of a data graph defines the *structure of application independent domains* d_i, i.e. $\bigcup_i c_i = V$. The members of d_i are identical to the members of c_i.

As indicated before the case where the set of subsets is not only covering but also non-overlapping, i.e. where each vertex of a given data graph belongs to exactly one domain, will be of particular importance and is formalised below.

Definition 4.2 (Equivalence domain structure): An *equivalence domain structure* D is formed by the set of all subsets b_i of the vertex-set V of a data graph that belong to the same equivalent class according to some specified equivalence relationship R on its vertex-set V. The resulting subsets b_i of vertices define the blocks of a partition $P = \{b_i\}$ of V.

This effectively bases the definition of equivalence domains on the definition of equivalence between vertices of the data graph. Multiple equivalence relationships

can be applied, resulting in different domains for the same source. This approach is further illustrated using the Examples 4.1 – 4.7. Because only domains actually occurring in a particular data source will be considered in accordance with the model developed in Chapter 3 and because data graphs and thus their vertex-sets are finite, the number of domains of a particular source governed by a specific definition is also finite. More specifically it is bounded by the size of the vertex-set, as in the worst case every vertex belongs to a different equivalence class and thus forms its own domain.

However, for application independent domains in general, which are not necessarily based on a partition but a cover of its vertex-set, this does not hold. A variant of Example 4.3 will highlight this situation by describing a potentially unbounded domain space.

In addition an application independent domain can be classified based on the set of members it comprises, i.e. the type of its vertices:

Definition 4.3 (Atomic domain): An application independent domain, whose set of members contains only atomic vertices, forms an *atomic domain.*

Definition 4.4 (Complex domain): An application independent domain, whose set of members contains only complex vertices, forms a *complex domain.*

Definition 4.5 (Mixed domain): An application independent domain, whose set of members contains at least one atomic and one complex vertex, forms a *mixed domain.*

Such domains will be denoted by a_i, c_i and m_i respectively, with the symbol d_i being used if the distinction is irrelevant in the current context.

Most of the following definitions are motivated by other research on SSD processing. The data clusters will be depicted in the way they arise in the context of the original work with the derived domain structure, i.e. the set of all discovered domains, superimposed. Tables 4.1 and 4.2 ignore the origins of the definitions and solely present the relations between the vertices of the example source and the various domains implied by the examples below. They can be found on page 74 and 75 respectively and visualise the domains using a common colour for the vertices belonging to a common domain.

Example 4.1 (Label domains): This is the simplest possible definition of domains presented here. All vertices carrying an identical label are grouped together.

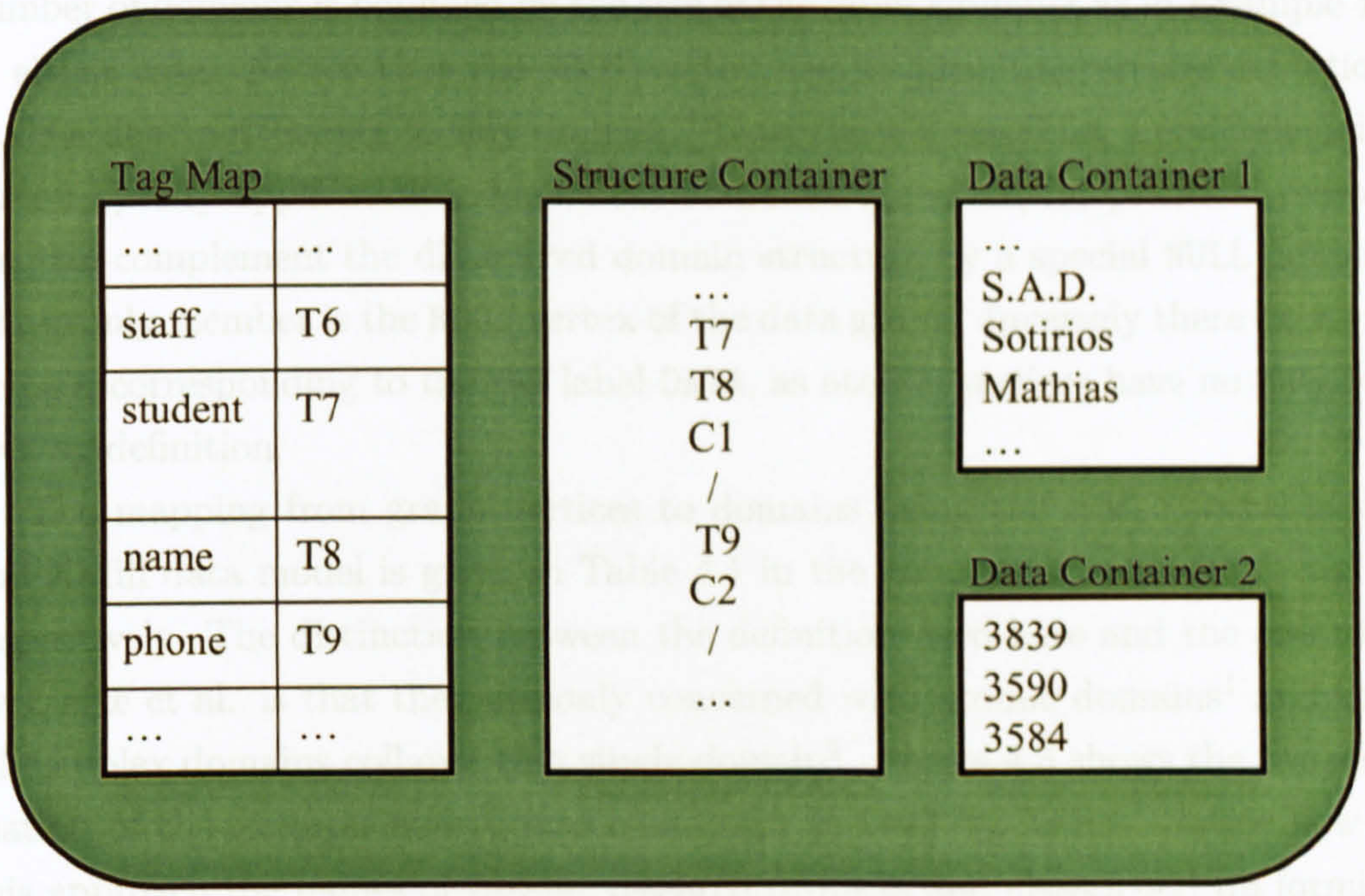


Fig. 4.3: The containers of the example source as identified by XMill

According to the data model of Section 2.2.2, all atomic values are contained in vertices labelled with the distinct tag "DATA". Thus there exist only a single atomic domain here, which contains all atomic data. Beyond this there exists one complex domain per occurring, source-specific tag name including one for the root vertex. The number of domains is thus bounded by the size of the label alphabet Σ. The domain structure specified by this approach is equivalent to a label map, as used by many SSD management systems and discussed in Example 2.1 of Section 2.4.1.

Example 4.2 (Parent domains): This definition, which is based on the equality of the label of a parent node of the context vertex in the distinct spanning tree is similar to the the transformation used for the XMill compression system by Liefke and Suciu [LS00]. Their work is based on a tree data model coinciding with the tree view defined in 2.11, in which the concept of a parent node naturally arises. For the general graph view of Definition 2.10, a vertex might have multiple parent vertices, one per incoming arc. Thus vertices with more than one incoming arc might belong to more than one domain. Therefore a cover of the vertex-set is defined here rather than a partition, i.e. this definition only defines equivalence domains on the tree-view but not on the graph-view of a data graph. The total

number of domains is bounded by the size of the label alphabet as in Example 4.1 in either case. Notice that the ROOT vertex has no incoming arc by definition. It thus does not belong to any domain. Thus strictly speaking parent domains do not specify application independent domains. However, for practical reasons one can complement the discovered domain structure by a special NULL domain, whose only member is the ROOT vertex of the data graph. Inversely there exists no domain corresponding to the tag label DATA, as atomic vertices have no outgoing arcs by definition.

The mapping from graph vertices to domains using the tree, general graph and XMill data model is given in Table 4.1 in the columns labelled T, G and X respectively. The distinction between the definition used here and the one used by Liefke et al. is that they are only concerned with atomic domains[1] and thus all complex domains collapse to a single domain[2]. Figure 4.3 shows the transformation of the example source into containers as used by XMill. Notice how in this approach the names of people, research projects and classes end up forming one atomic domain name (Data Container 1 in Figure 4.3). This does not form a semantically homogeneous domain. This effect of accidental combination of unrelated information will be called *domain mixing*. XMill provides a language called "container expressions", which can be used to manually avoid such unwanted mixing.

Example 4.3 (Path domains): This domain definition tries to avoid domain mixing by taking the entire path from the root into account. Equivalence is based on the equality of the path from the root to the vertex in question. As in the previous example, this might result in a vertex being part of more than one domain, i.e. in general path domains do not define equivalence domains. In fact cyclic graphs will result in an unbounded number of domains. However, the equivalent definition based on the distinguished spanning tree is bounded by the size of the node-set as every node of a tree can be reached by exactly one path from the root and thus defines an equivalence domain structure.

In Table 4.1 the variants for tree and general graph data model are indicated by the columns headed by T and G. Figure 4.4 shows the strong DataGuide as defined by Goldman and Widom [GW97] of the example document. It represents a path index of the spanning tree. For the purpose of this definition, the vertices

[1] called "data containers" in the context of their work
[2] called "structural container"

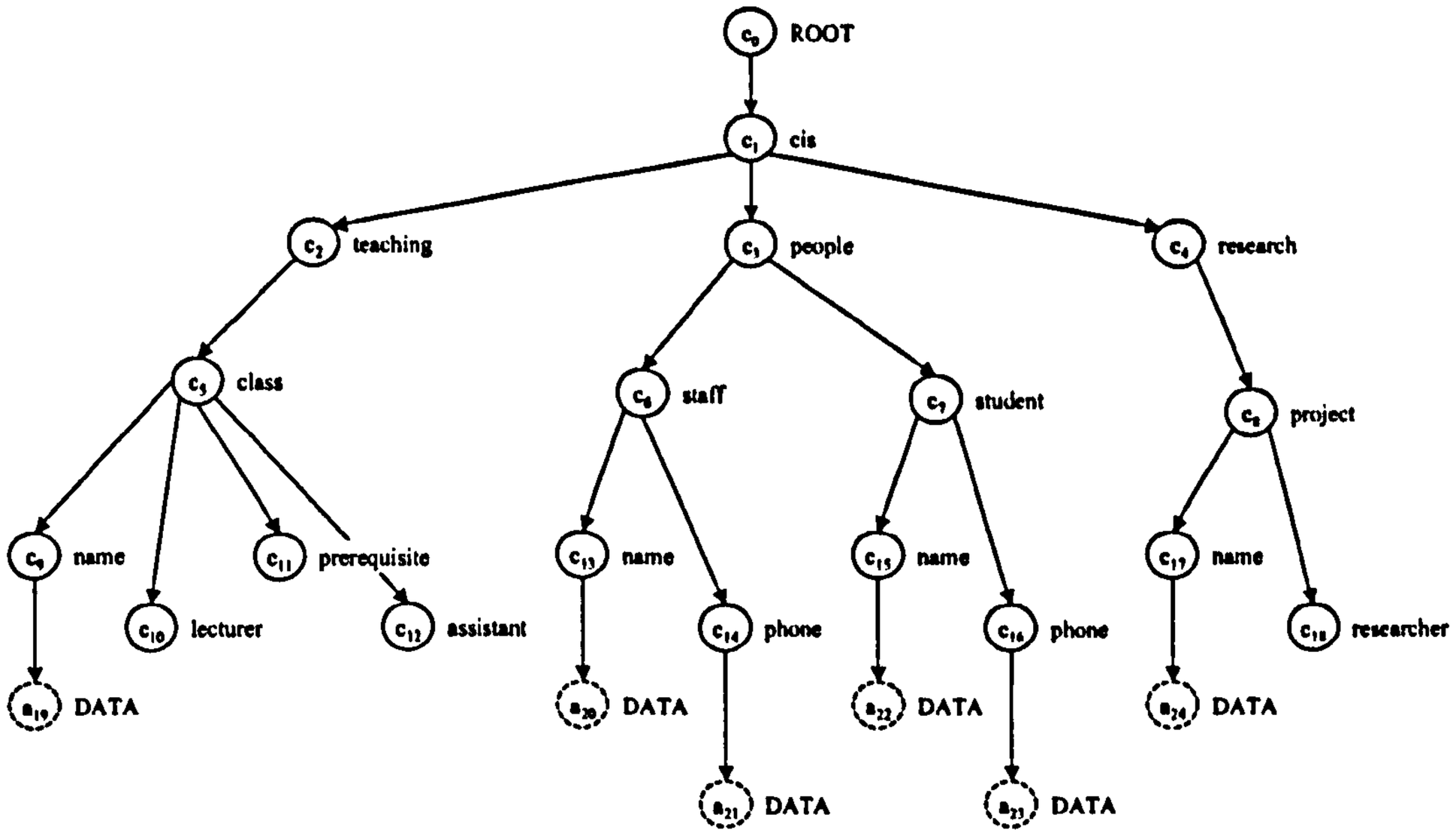


Fig. 4.4: The strong DataGuide of the example database implies path domains

of the DataGuide define domains. Under this model the atomic domains of phone numbers seen in the last example is split into two, one for `student` phone numbers and one for `staff` phone numbers. For most applications, e.g. for compression purposes, such a separation will be counterproductive. This effect of accidental separation of semantically related information will be called *domain splitting*.

Example 4.4 (Depth domains):

Definition 4.2 allows for the characterisation of domains on the basis of node level, i.e. the length of the incoming path from the root in the distinguished spanning tree. Here all nodes occurring at the same depth in the tree-view are grouped together. This clearly fulfills Definition 4.2 from above, although it is harder to see the possible use of the implied data grouping. However, as Example 3.3 has shown, the linear path expression

Query 4.1: /*/*/*

for example, which selects all nodes with an incoming, arbitrarily labelled path of length three from the `ROOT` node, can be resolved effectively using a partition generated by this definition of an equivalence domain structure. It selects all members of domain c_3 in Figure 4.5, which shows the depth domains for the

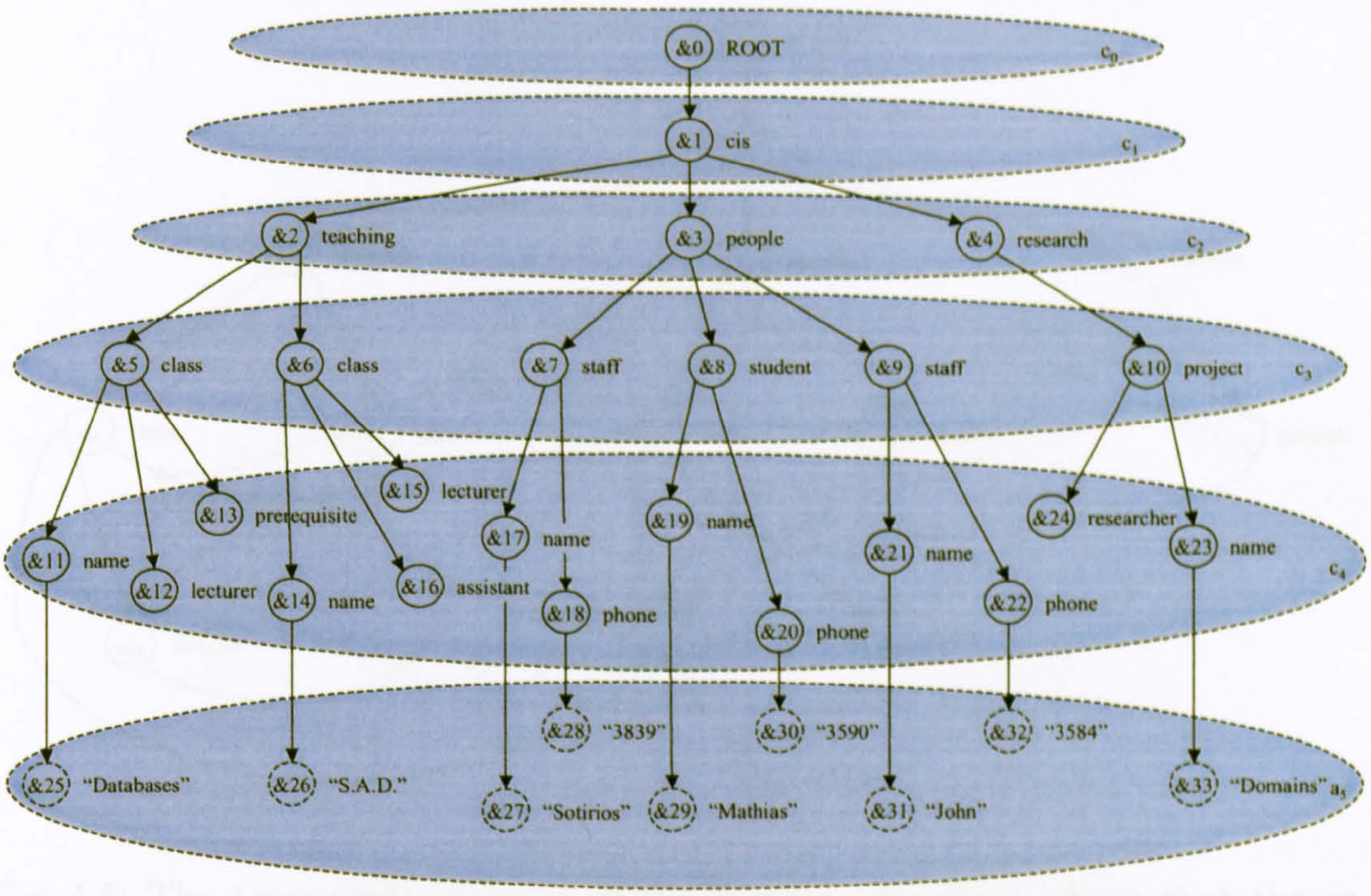


Fig. 4.5: The depth domains superimposed on the tree-view of the data graph

example source. Though in our example source all resulting domains are either atomic or complex, in general this will not be the case. As the labels of the nodes are not taken into account by the equivalence relationship, the occurrence of mixed domains is possible. Particularly it will occur in the equivalent definition for the graph-view of the example document. This variant suffers from the same drawbacks as the path based domain definition for graphs in Example 4.3, i.e. it is not an example of an equivalence domain and will result in an unbounded number of domains for cyclic sources. Once again the mappings for either variant are included in Table 4.2 and indicated by the columns T and G respectively.

Example 4.5 (Skeleton domains): This definition is based on the work performed by Buneman et al. [BGK03] on compressing the structural component of a document tree, referred to as document skeleton in the original work. Their approach is based on the concept of forward bisimilarity of a node, i.e. the equality of the collection of outgoing paths. According to their definition, two nodes in the tree are considered bisimilar, if they have the same tag label and the same ordered sequence of bisimilar child nodes. This definition identifies common substructures in a document, e.g. a `staff` node with one `name` and one `phone` child nodes. Notice how the definition splits the two `class` vertices into different domains, be-

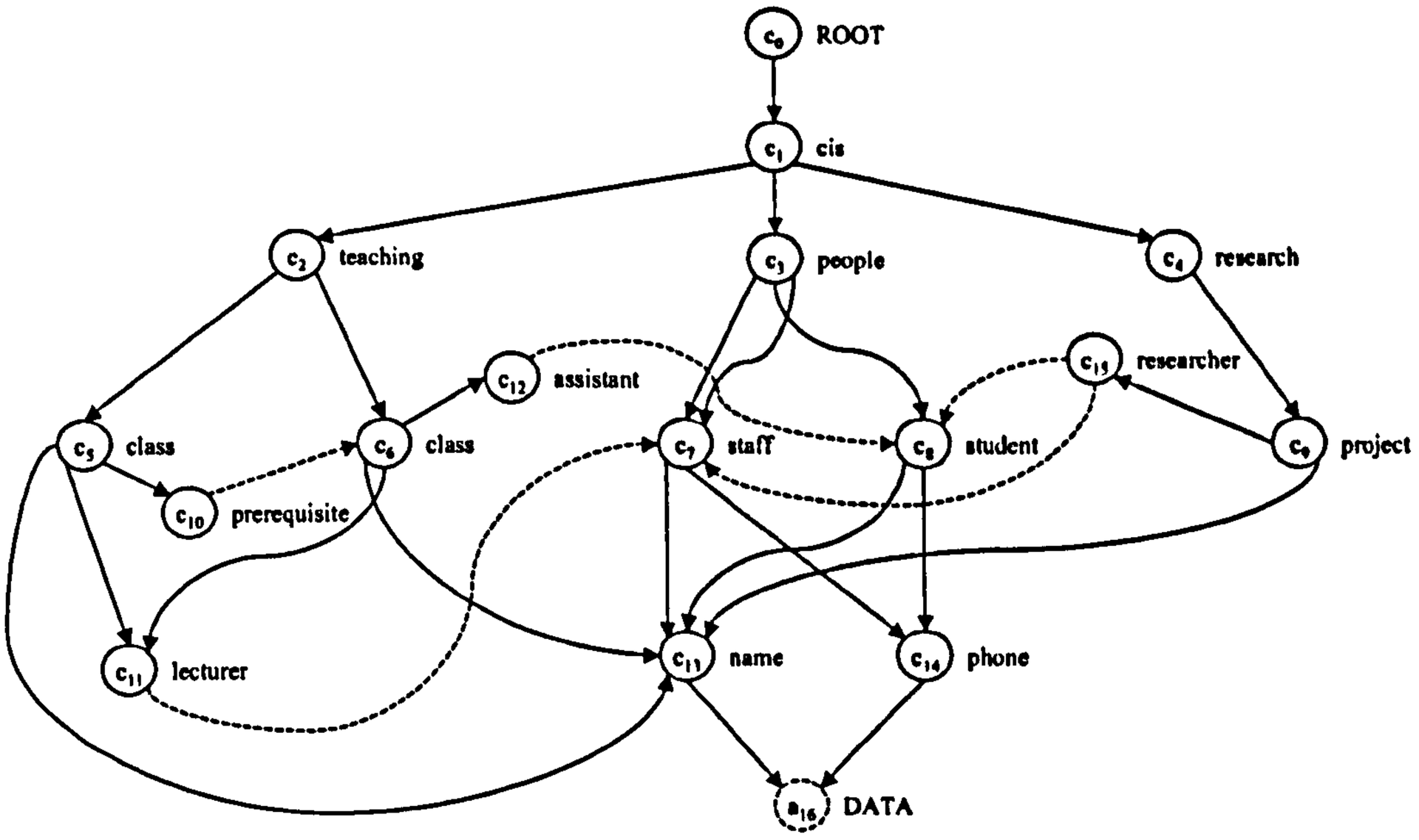


Fig. 4.6: The (extended) skeleton of the example database. Note that the order of arcs is important for this approach. Outgoing arcs are ordered counterclockwise starting from the top of the source vertex.

cause they are the root of differently structured subtrees, while it keeps the two `staff` vertices together. As Figure 4.6 shows, the same definition can be applied to graphs, if the outgoing arcs of a vertex are considered to form an ordered sequence. In the particular example used, the domain structure remains unchanged in this case. In general, however, it might lead to a refined domain structure, for example if the two `staff` vertices of the example source were not equivalent because the order of their outgoing arcs differed, domain c_{11} in Figure 4.6 would also split as it has an outgoing edge to the `staff` domains. Nevertheless, this would still form an equivalence domain, as vertex bisimilarity generally defines an equivalence relationship and thus every vertex of the graph belongs to exactly one equivalence class. Note that the original work by Buneman was only concerned with the structural part of the document and thus consists of only the vertices defining complex domains in Figure 4.6.

Due to the fact that only outgoing arcs are taken into account for the computation of the bisimilarity, all atomic data belongs to the same atomic domain because, by definition, atomic vertices have no outgoing arcs. This is identical to the behaviour of the label domain defined in Example 4.1. The skeleton domains

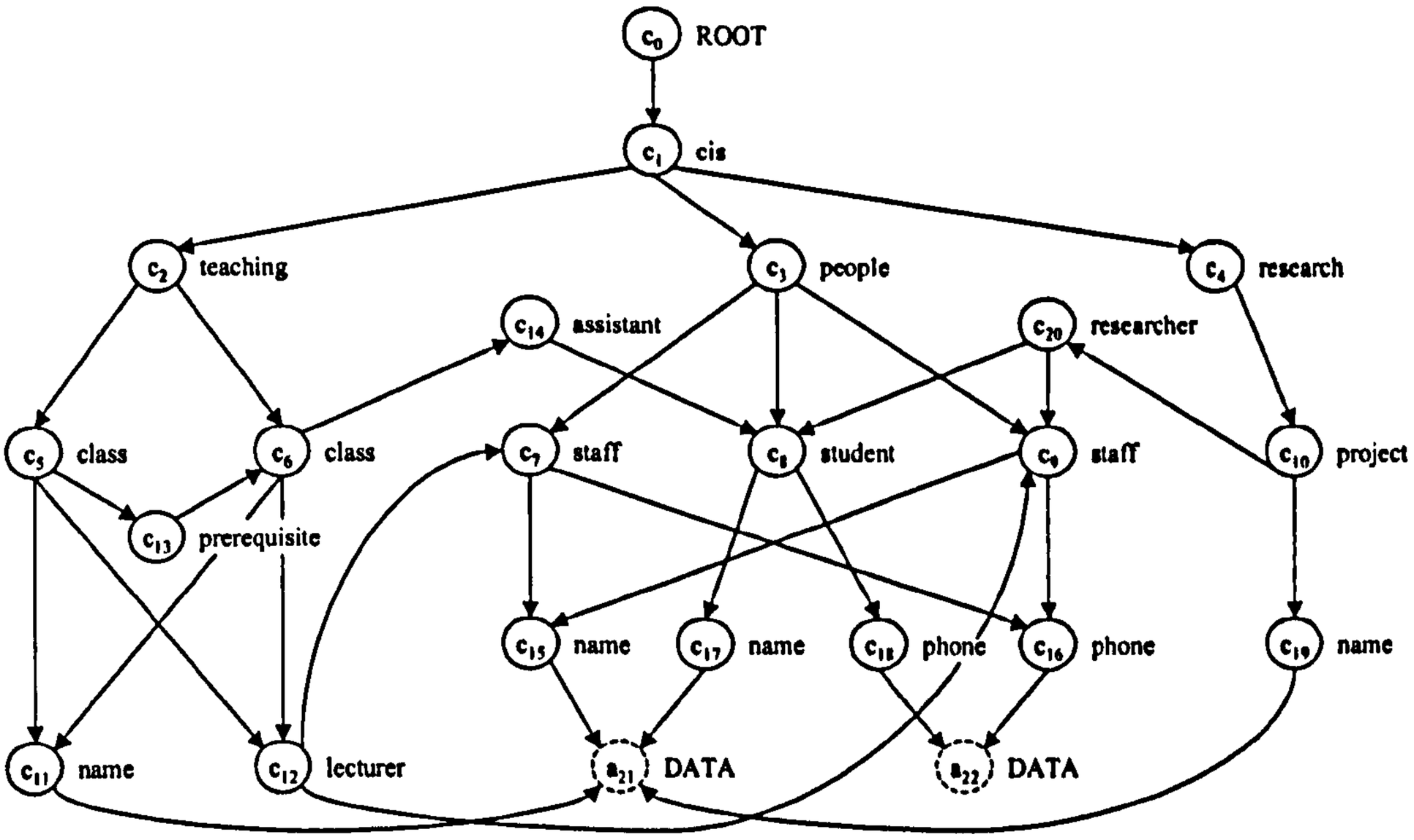


Fig. 4.7: The A(1)-index graph of the example source

approach is the only one presented here that takes sibling order into account. A `staff` node with a `name` and `phone` child node (in this order) for example is not considered to be equivalent to a `staff` node with a `phone` and a `name` child node (in that order) and thus according to Definition 4.2 each belongs to a different domain. This is a consequence of the target query language of XPath chosen in the original work, which contains several order-dependent query operations. It also explains the restriction to trees of their research.

Example 4.6 (Local backward bisimilarity domains): An alternative application of the concept of bisimilarity to obtain a partition of the vertex-set of general data graphs is studied by Kaushik et al. [KS+02]. Sibling order and multiplicity are dropped due to the less expressive target language of forward facing, linear path expressions. Instead of basing node equivalence on outgoing arcs, it is based on incoming arcs, i.e. they define their equivalence classes based on backward bisimilarity. This is similar to the path index described in Example 4.3. However, in this definition the entire set of incoming paths is used to determine the domain of a vertex rather than defining one domain per incoming path. Thus this definition implies a partition of the vertex-set and hence a set of equivalence domains unlike the path index, which only defines a set cover.

An index graph as defined in Definition 2.12 based on this partition is covering

for all forward facing, linear path expressions. Since path expressions in practice are often of limited length, the length of the paths taken into consideration to compute the bisimulation can also be limited, thus exploiting local bisimilarity. The definition of bisimilarity changes to a recursive definition, where a vertex is k-bisimilar if it has the same tag label and has the same set of incoming arcs from vertices that are $(k-1)$-bisimilar. The length parameter k used to compute the graph's bisimulation can be used as an optimisation parameter to trade the balance between index size and index coverage.

Figure 4.7 shows the index graph of the example source based on 1-bisimilarity, i.e. vertices are considered equivalent if they have the same tag label and the are connected through incoming arcs to sets of vertices, which have identical tag labels. Again every vertex of the index graph implies a domain. Notice how the two `staff` vertices that were contained in a single domain in Example 4.5 are now separated into two different domains here due to their different incoming paths. One `staff` vertex is reachable by a `researcher` vertex while the other one is not. As in the case of the parent domain there exist two atomic domains, one for names and one for phone numbers. Again class names, person names and project names are mixed in one domain, although this could be prevented by increasing the bisimilarity length parameter k to 2. The original research by Kaushik et al. focused solely on structural querying and thus only considers vertices representing complex domains in Figure 4.7.

Example 4.7 (Forward and backward bisimilarity domains): Kaushik et al. [KB+02] finally considered the case where both incoming and outgoing paths are taken into consideration for the computation of the bisimulation. This means that this approach combines structural properties like those exposed by skeleton domains with contextual properties such as those identified by local backward bisimilarity domains. The resulting index graph is covering for general branching path expressions without value predicates. These onerous conditions on vertex equivalence lead to very complex domain structures in the context of this chapter and prohibitively large index graphs by comparison with earlier research on index graphs, such as the DataGuides of Example 4.3 and the compressed skeletons of Example 4.5. As in the previous example the size can be reduced by using two optimisation parameters k_b and k_f that restrict the length for both incoming and outgoing paths to be taken into consideration. Additionally the number of allowed "twists" in the query graph can be restricted to further reduce the com-

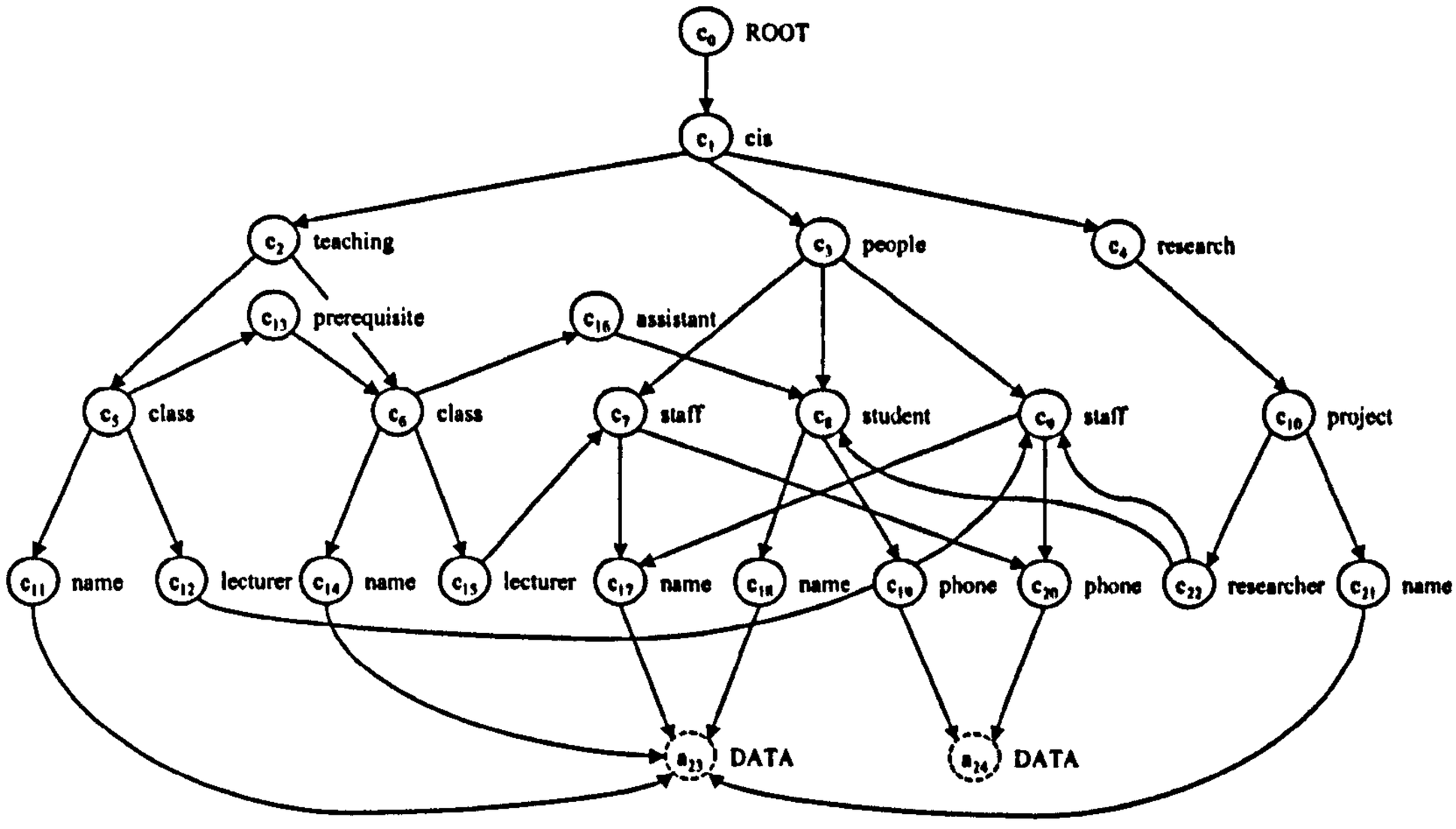


Fig. 4.8: The (1,1)-F+B-Index graph of the example source

plexity of the resulting domain structure. This intrinsic complexity measure of a query was called the "tree depth" *td* by Kaushik et al. Details of this optimisation parameter are given in Appendix B and should be consulted in order to become familiar with their notations used here. In the most general case, i.e. for $k_b = k_f = td = \infty$, the tree depth is unrestricted and the resulting index is called the F&B-index. The special case of the index that results from a single iteration of the algorithm presented in Figure B.2 and is covering for queries with $td = 1$, i.e. query graphs with at most one predicate where incoming and outgoing paths meet, is called the F+B-index.

Figure 4.8 shows the index graph based on (1,1)-F+B-bisimilarity, i.e. an index that is covering for queries whose arcs are matched by paths of the data graph of length one and have at least one vertex where incoming and outgoing paths meet. Notice that the `name` and `lecturer` vertices occurring below `class` vertices have been split, although their incoming path of length one are identical. This is due to the fact that they can be distinguished based on their siblings by using a branching path expressions whose length does not exceed one. For example the branching path expression `class[/prerequisite]/name` selects vertex **&11** but not vertex **&14**, which were combined in domain c_{11} in Figure 4.7 but are split into the domains c_{11} and c_{14} in Figure 4.8. Again the original research focused solely on structural querying and thus would ignore all atomic domains

in the presented example.

The examples above show that different implementations of domains result in different data groupings. The complete mappings between the various variants of the provided definitions and the example dataset is given in Table 4.1 and 4.2. As was explained in Chapter 3, the suitability of a definition is highly dependent on the class of queries to be answered. Performance aspects of the various indices resulting from these data groupings were studied by Goldman et al. [GW97], Kaushik et al. [KB+02], Buneman et al. [BGK03] and others. Beyond performance the number of the identified domains can also be used as metrics to characterise a particular source. This will be demonstrated in Section 4.3, where the impact that competing definitions have on real data sources will be evaluated.

4.2.1.1 Relationships between application independent domain definitions

The definitions provided above have been presented in the order of increasing complexity of the resulting domain structures, which primarily coincides with the order in which they were addressed by research in related areas. Having established these definitions it is possible to identify a number of relationships between them.

In general domains by label are identical to those defined by local bisimilarity with length 0, i.e. those corresponding to the vertices of the A(0)- and the (0,0)-F+B-index graph. If the document order is ignored, skeleton domains are identical to $(\infty, 0)$-bisimilarity domains, i.e. those based on a partition which is refined in terms of forward bisimilarity until a fix-point is reached and not refined in terms of backward bisimilarity at all. If one drops the additional graph arcs and restricts oneself to the spanning tree there are even more relationships to discover. Now, path domains become identical to domains based on unbounded backward bisimilarity, i.e. those based on the A(∞)- or $(0, \infty)$-F+B-index. In general the A(k)- and $(0, k)$-F+B-index are identical and so are the domains on which they are based. The domains by parent however are not equal to those based on the $A(1)$-index as one could expect, as their definition does not depend on the label of the context vertex itself. However if one restricts oneself to atomic data nodes, as done for the XMill variant of this approach, the assumption is true for this restricted subset because the label of atomic data nodes is fixed by the data model. In this selection of definitions only those based on depth are not closely related to any others, which follows from the nature of their defini-

Example definition		4.2			4.3	
Vertex		Parent			Path	
oid	label/value	T	G	X	T	G
0	ROOT	c_0 [1]	c_0 [1]	c_0	c_0	c_0
1	cis	c_1	c_1	c_0	c_1	c_1
2	teaching	c_2	c_2	c_0	c_2	c_2
3	people	c_2	c_2	c_0	c_3	c_3
4	research	c_2	c_2	c_0	c_4	c_4
5	class	c_3	c_3	c_0	c_5	c_5
6	class	c_3	c_3 c_{12}	c_0	c_5	c_5 c_{25}
7	staff	c_4	c_4 c_{14}	c_0	c_6	c_6 c_{26}
8	student	c_4	c_4 c_{13} c_{15}	c_0	c_7	c_7 c_{27} c_{28} c_{29}
9	staff	c_4	c_4 c_{14} c_{15}	c_0	c_6	c_6 c_{26} c_{30}
10	project	c_5	c_5	c_0	c_8	c_8
11	name	c_6	c_6	c_0	c_9	c_9
12	lecturer	c_6	c_6	c_0	c_{10}	c_{10}
13	prerequisite	c_6	c_6	c_0	c_{11}	c_{11}
14	name	c_6	c_6	c_0	c_9	c_9 c_{31}
15	lecturer	c_6	c_6	c_0	c_{10}	c_{10} c_{32}
16	assistant	c_6	c_6	c_0	c_{12}	c_{12} c_{33}
17	name	c_7	c_7	c_0	c_{13}	c_{13} c_{34} c_{35}
18	phone	c_7	c_7	c_0	c_{14}	c_{14} c_{36} c_{37}
19	name	c_8	c_8	c_0	c_{15}	c_{15} c_{38} c_{39} c_{40}
20	phone	c_8	c_8	c_0	c_{16}	c_{16} c_{41} c_{42} c_{43}
21	name	c_7	c_7	c_0	c_{13}	c_{13} c_{34} c_{44}
22	phone	c_7	c_7	c_0	c_{14}	c_{14} c_{36} c_{45}
23	name	c_9	c_9	c_0	c_{17}	c_{17}
24	researcher	c_9	c_9	c_0	c_{18}	c_{18}
25	Databases	a_{10}	a_{10}	a_1	a_{19}	a_{19}
26	S.A.D.	a_{10}	a_{10}	a_1	a_{19}	a_{19} a_{46}
27	Sotirios	a_{10}	a_{10}	a_1	a_{20}	a_{20} a_{47} a_{48}
28	3839	a_{11}	a_{11}	a_2	a_{21}	a_{21} a_{49} a_{50}
29	Mathias	a_{10}	a_{10}	a_1	a_{22}	a_{22} a_{51} a_{52} a_{53}
30	3590	a_{11}	a_{11}	a_2	a_{23}	a_{23} a_{54} a_{55} a_{56}
31	John	a_{10}	a_{10}	a_1	a_{20}	a_{20} a_{47} a_{57}
32	3584	a_{11}	a_{11}	a_2	a_{21}	a_{21} a_{49} a_{58}
33	Domains	a_{10}	a_{10}	a_1	a_{24}	a_{24}
34	total	12	16	3	25	59

Tab. 4.1: Overview of the relation between the vertices of the example source and the domains defined in Example 4.2 – 4.3.
[1] the special `NULL` domain is described in Example 4.2

Example definition		4.1	4.4		4.5	4.6	4.7
Vertex		Label	Depth		Bisimilarity		
oid	label/value		T	G	$(0, \infty)$	$(1, 0)$	$(1, 1)$
0	ROOT	c_0	c_0	c_0	c_0	c_0	c_0
1	cis	c_1	c_1	c_1	c_1	c_1	c_1
2	teaching	c_2	c_2	c_2	c_2	c_2	c_2
3	people	c_3	c_2	c_2	c_3	c_3	c_3
4	research	c_4	c_2	c_2	c_4	c_4	c_4
5	class	c_5	c_3	c_3	c_5	c_5	c_5
6	class	c_5	c_3	c_3 m_5	c_6	c_6	c_6
7	staff	c_6	c_3	c_3 m_5 m_7	c_7	c_7	c_7
8	student	c_7	c_3	c_3 m_5 m_7	c_8	c_8	c_8
9	staff	c_6	c_3	c_3 m_5	c_7	c_9	c_9
10	project	c_8	c_3	c_3	c_9	c_{10}	c_{10}
11	name	c_9	c_4	c_4	c_{10}	c_{11}	c_{11}
12	lecturer	c_{10}	c_4	c_4	c_{11}	c_{12}	c_{12}
13	prerequisite	c_{11}	c_4	c_4	c_{12}	c_{13}	c_{13}
14	name	c_9	c_4	c_4 c_6	c_{10}	c_{11}	c_{14}
15	lecturer	c_{10}	c_4	c_4 c_6	c_{11}	c_{12}	c_{15}
16	assistant	c_{12}	c_4	c_4 c_6	c_{13}	c_{14}	c_{16}
17	name	c_9	c_4	c_4 c_6 c_8	c_{10}	c_{15}	c_{17}
18	phone	c_{13}	c_4	c_4 c_6 c_8	c_{14}	c_{16}	c_{18}
19	name	c_9	c_4	c_4 c_6 c_6	c_{10}	c_{17}	c_{19}
20	phone	c_{13}	c_4	c_4 c_6 c_8	c_{14}	c_{18}	c_{20}
21	name	c_9	c_4	c_4 c_6	c_{10}	c_{17}	c_{19}
22	phone	c_{13}	c_4	c_4 c_6	c_{14}	c_{18}	c_{20}
23	name	c_9	c_4	c_4	c_{10}	c_{19}	c_{21}
24	researcher	c_{14}	c_4	c_4	c_{15}	c_{20}	c_{22}
25	Databases	a_{15}	a_5	m_5	a_{16}	a_{21}	a_{23}
26	S.A.D.	a_{15}	a_5	m_5 m_7	a_{16}	a_{21}	a_{23}
27	Sotirios	a_{15}	a_5	m_5 m_7 a_9	a_{16}	a_{21}	a_{23}
28	3839	a_{15}	a_5	m_5 m_7 a_9	a_{16}	a_{22}	a_{24}
29	Mathias	a_{15}	a_5	m_5 m_7 a_9	a_{16}	a_{21}	a_{23}
30	3590	a_{15}	a_5	m_5 m_7 a_9	a_{16}	a_{22}	a_{24}
31	John	a_{15}	a_5	m_5 m_7	a_{16}	a_{21}	a_{23}
32	3584	a_{15}	a_5	m_5 m_7	a_{16}	a_{22}	a_{24}
33	Domains	a_{15}	a_5	m_5	a_{16}	a_{21}	a_{23}
34	total	16	6	10	17	23	25

Tab. 4.2: Overview of the relations between the vertices of the example source and the domains defined in Example 4.1 and 4.4 – 4.7

tion. Particularly, all other definitions can be reduced to a special form of those defined by bisimilarity, though this general definition does not take order into account. Similar considerations from the perspective of indexing were developed by Nestorov et al [NU+97]. This observation will be used again in chapter 6, which will be partly based on the concept of bisimilarity.

4.2.2 Application dependent domains

If the semantics of the particular application are known, one does not need to maintain the degree of generality provided by the application independent domains. Instead an arbitrary subgraph classification can be used, which matches exactly the requirements of the application and thus does not require subsequent analysis of the classified data.

Definition 4.6 (Semantic domain): Two subgraphs of a data graph belong to the same *semantic domain* with respect to a given application, if they can be processed by the application in the same way.

Note that this does not mean that values from the same domain are indistinguishable, but solely that they will serve as valid input to some given algorithm. They can still be distinguished by the computation based on their actual value or the order in which they occurr in the document if this is taken into account by the application's data model. Again, this approach will be illustrated using a practical example.

Example 4.8 (Type domains): It is possible to capture the data requirements of a specific application by means of the types used. Types carry associated semantics, and thus do not need to be interpreted by the application. However, they require the provision of a mapping between the generic data model of the data graph and the type system of the application. A possible approach is to map tag labels to field names, complex vertices to structured types such as objects, and atomic vertices to atomic types of the targeted type system. Siblings in the tree-view carrying identical tag labels can be thought of as implying collection-like types such as arrays. An implementation based on type projection [CL+01] applying these assumptions over streams of SSD is described in Appendix A. Figure 4.9 shows the result of projecting the type **Person**, whose definition is shown in Listing 4.1, over the example source.

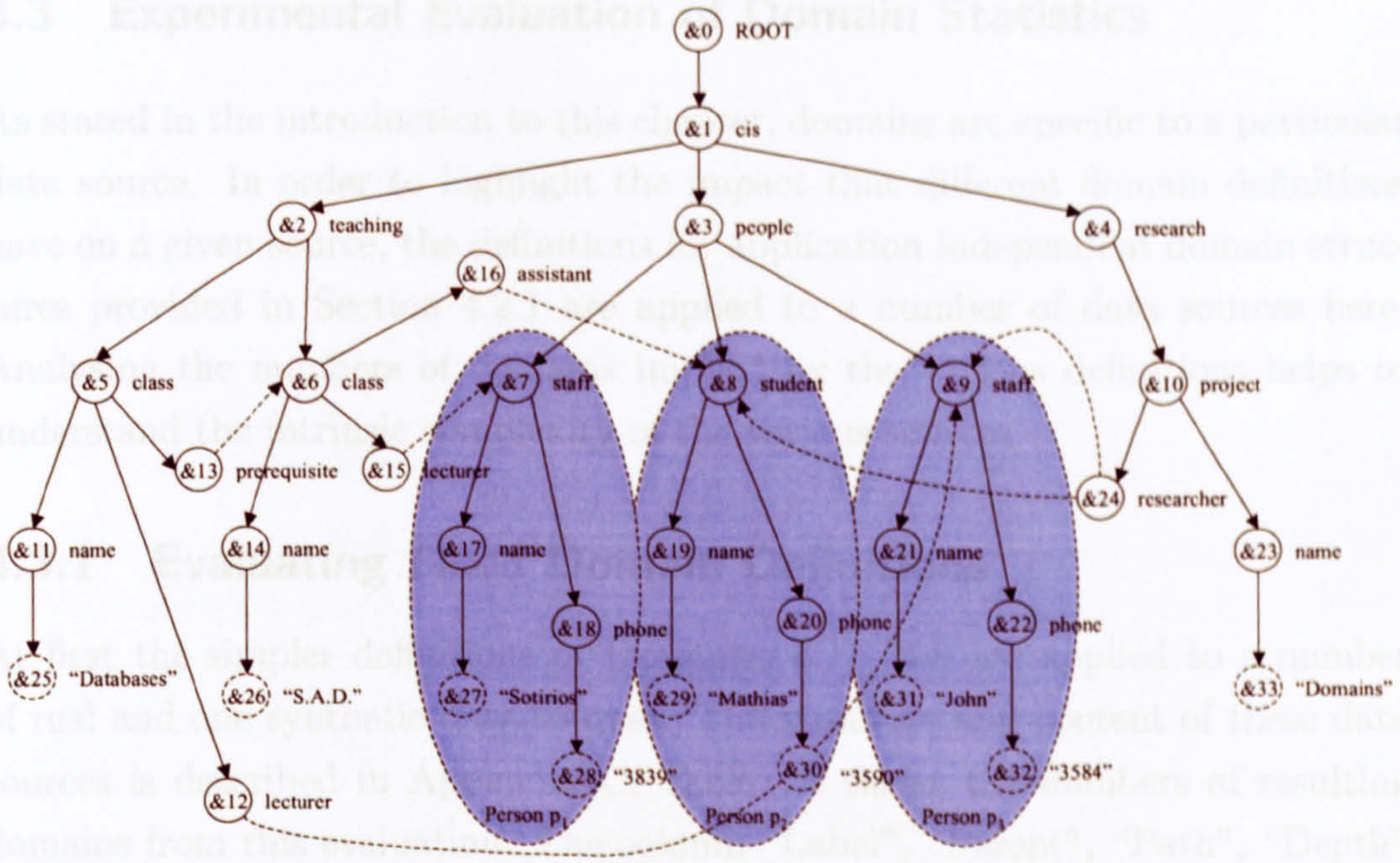


Fig. 4.9: The **Person** type projected over the example graph defines semantic domains

```
class Person {
  String name;
  int phone;
}
```

Listing 4.1: The **Person** type

Notice that the mapping between the data graph and the application dependent domains is not complete and does not need to be non-overlapping as in the example provided. This means that one deals with a non-covering set of vertex subsets here as opposed to the vertex-set cover required in the case of application independent domains. In fact the initial clustering problem based on the properties of the data source itself has been transformed into a classification problem according to some externally provided classes. In general this approach will be too specific to be of use in a database management system. It is mentioned for completeness with respect to the definition of domains for SSD presented above. However, the remainder of this thesis is restricted to the more tractable problem of application independent domain definitions.

4.3 Experimental Evaluation of Domain Statistics

As stated in the introduction to this chapter, domains are specific to a particular data source. In order to highlight the impact that different domain definitions have on a given source, the definitions for application independent domain structures provided in Section 4.2.1 are applied to a number of data sources here. Analysing the numbers of domains implied by the various definitions helps to understand the intrinsic complexity of the various sources.

4.3.1 Evaluating Fixed Domain Definitions

At first the simpler definitions of Examples 4.1 – 4.5 are applied to a number of real and one synthetic data sources. The structure and content of these data sources is described in Appendix C. Table 4.3 shows the numbers of resulting domains from this evaluation. The column "Label", "Parent", "Path", "Depth" and "Skeleton" give the number of distinct domains for the definitions given in the Examples 4.1 – 4.5 respectively based on the tree-view of the source. Notice that none but the XMark source make use of explicit cross references, i.e. their data graphs coincide with their tree- and graph-views. The behaviour of the XMark dataset is analysed in more detail in Section 4.3.2. Vertically the number of domains are broken down by their content, i.e. how many of the identified domains contain atomic data, complex content, or a mixture of both. Wherever a schema in form of a DTD was available it was used during parsing to avoid the creation of unwanted domains for whitespace characters occurring between the mark-up of the XML representation. However, for one of the sources, the Nasa data set, no DTD was available and as a consequence such unwanted domains could not be excluded.

4.3.1.1 Data sources with a regular structure

The results in Table 4.3 clearly show that by applying competing domain definitions one can easily identify inherently regular sources, which were encoded in a semistructured format. Such a source from the collection of example documents is the domain name server database (DNS). For this source the number of atomic domains defined by the tag name of the parent node and by the complete path from the root node are identical, indicating a fixed, regular schema. In fact not only the number but also the contents of these domains are identical. Thus no

Source	Schema	Size $\|V\|$	Label	Parent	Path	Depth	Skeleton
Total number of domains							
Macbeth	DTD	7.3k	18	17	38	8	104
XMark (tiny)	DTD	322k	78	77	933	14	13662
Nasa	—	1.5M	72	70	221	10	8840
DNS 100k	DTD	1.8M	15	14	25	5	19
DBLP	DTD	7.1M	42	41	282	8	5538
Number of atomic domains							
Macbeth	DTD	7.3k	1	8	15	1	1
XMark (tiny)	DTD	322k	1	34	405	1	1
Nasa	—	1.5M	1	29	109	1	1
DNS 100k	DTD	1.8M	1	11	11	1	1
DBLP	DTD	7.1M	1	24	136	1	1
Number of complex domains							
Macbeth	DTD	7.3k	17	8	23	3	103
XMark (tiny)	DTD	322k	77	39	528	5	13661
Nasa	—	1.5M	71	2	112	2	8839
DNS 100k	DTD	1.8M	14	3	14	4	18
DBLP	DTD	7.1M	41	10	146	4	5537
Number of mixed domains							
Macbeth	DTD	7.3k	0	1	0	4	0
XMark (tiny)	DTD	322k	0	4	0	8	0
Nasa	—	1.5M	0	39	0	7	0
DNS 100k	DTD	1.8M	0	0	0	0	0
DBLP	DTD	7.1M	0	7	0	3	0

Tab. 4.3: The number of domains discovered using different domain definitions

linear path expression can distinguish any of the atomic vertices of this source that could not be distinguished by the name of its parent vertex alone. Equally the number of complex domains defined by tag labels is identical to those defined by complete paths. This shows that every vertex with a specific tag label only occurs in a fixed context of the tree-view. However, if the structure of entries in this source is considered using the skeleton domain structure, the domains for `server` entries is broken into three distinct domains, those with three, four and five parts to their domain name. Another strong indication for a regular source is the fact that only one of the five discovered depths domains contains atomic values. All other sources result in mixed domains for this definition, but in the DNS database all atomic values sit at a fixed depth of the tree. In general the number of domains is very low in comparison to the number of nodes in the document tree and varies only slightly for different domain structures.

4.3.1.2 Data sources with fixed context and varying content

To a lesser extent, the Shakespeare and the Nasa datasets show a similar behaviour. The number of complex domains identified using complete paths is only slightly bigger than the number of domains identified by tag label names alone. This indicates that a particular element can only occur in very few contexts throughout the document, i.e. that it complies to a fairly strict schema. In the Shakespeare play for example, `line` vertices, can only appear as children of `speech` vertices, but nowhere else in the data tree, e.g. not attached to a `stagedir` vertex. However, the excessive increase of the number of domains identified by the skeleton approach indicates that the order and cardinality of individual child nodes varies greatly. In the example there is a multitude of different `speech` subtrees, with and without stage directions and varying numbers of `line` vertices attached to it. This justifies the semistructured encoding chosen as it would be tedious work to normalise it in order to store it in a relational system. The Shakespearean play thus emphasises the importance of order for sources containing information encoding natural languages.

4.3.1.3 Data sources with varying context and content

Finally there are a number of documents, in which neither the context or the content of a particular vertex is restricted. They typically make extensive use of regular expressions in their schema definition, allowing a great deal of flexibility,

or have no schema at all. Here similar elements can occur in many different contexts, detectable through a significant increase between the number of domains identified using tag labels alone versus the number of domains implied by complete paths. Examples of such sources are the XMark benchmark dataset and, to a lesser extend, the DBLP database. If the complete structure of the entries is incorporated in the definition using the skeleton approach, the number of domains increases even further, here by more than an order of magnitude each.

4.3.2 Evaluating Parameterised Domain Definitions

Unlike the domain structures defined in Examples 4.1 – 4.5, Examples 4.6 and 4.7 define a family of domain structures. Thus it is interesting to measure the influence of their input parameters on the number of domains created. In particular one might be interested to find the points in the parameter space at which the resulting domain structure changes most rapidly and conversely where it is stable with respect to variations. If such points can be identified algorithmically the task of finding parameters that represent useful indices might be automated.

In comparison to the domain analysis performed in Section 4.3.1, computing the parameterised bisimilarity domain structure of data source is a computationally expensive task. Thus the analysis is restricted to a single source here, a scaled-down version of the XMark dataset. It represents the most challenging source of those introduced in the previous section, as it contains many additional graph arcs beyond the distinct spanning tree. The other sources, though they all expose some degree of semistructuredness, do not contain such arcs. The analysis of bisimilarity domains on such sources would thus produce similar results as provided by path and skeleton domains as they coincide with backward and forward bisimilarity domains for hierarchical data sources.

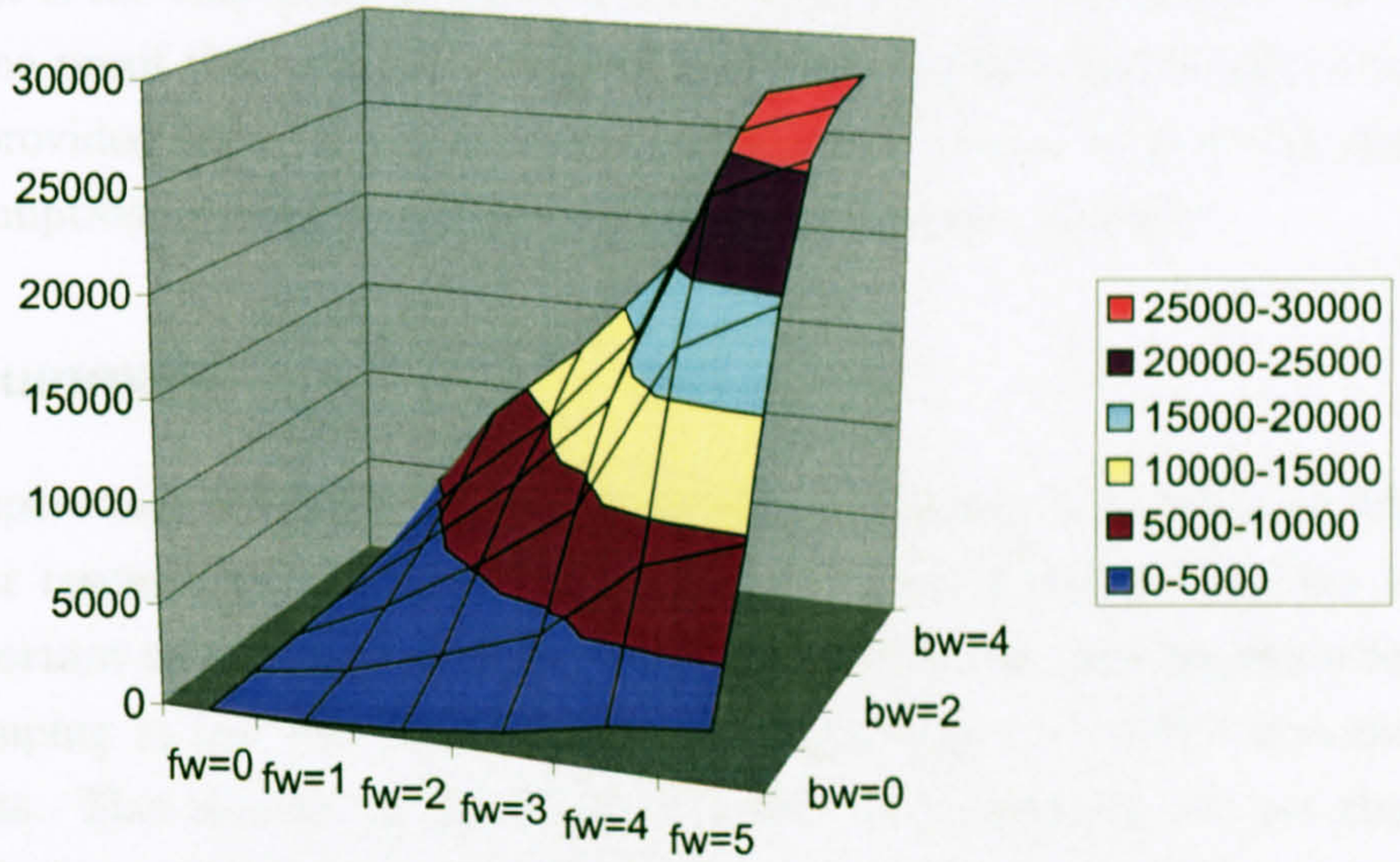
Table 4.4 shows the number of domains computed for the 1 MB XMark dataset, whose data graph contains 32,864 vertices. Notice that this presentation includes the results based on local backward bisimilarity of Example 4.6 in the the part of column with $k_f = 0$ and $d = 1$ (cyan). Particularly the number of domains shown for the case $k_b = 0$ of this subset coincides with the number of domains by label (Example 4.1, blue). The measurement results are also presented as a surface diagram in Figure 4.10 in order to visualise the complexity of the domain structure over the parameter space. The shape of the surface almost follows the normal distribution, only the rate with which it approaches the maximal number

Tree depth d	Backward bisimilarity k_b	Forward bisimilarity k_f					
		0	1	2	3	4	5
1	0	78	192	482	1119	2118	2624
1	1	156	888	2291	4020	8047	9650
1	2	266	2440	5288	8674	17149	19594
1	3	481	4200	8024	12824	23512	25567
1	4	877	5939	9965	15341	25733	27370
1	5	1426	8464	12545	17683	26892	28029
2	0	78	482	2118	3042	3321	3348
2	1	266	4276	12284	15680	19310	20354
2	2	877	9539	22161	26537	27873	28138
2	3	2025	15053	26884	28615	28772	28818
2	4	4384	18159	28232	28857	28873	28873
2	5	7336	18769	28401	28869	28873	28873
3	0	78	1119	3042	3339	3353	3353
3	1	481	10381	22146	24890	26330	26552
3	2	2025	18947	28462	28829	28851	28856
3	3	5648	21549	28839	28873	28873	28873
3	4	11573	22478	28873	28873	28873	28873
3	5	13403	22575	28873	28873	28873	28873

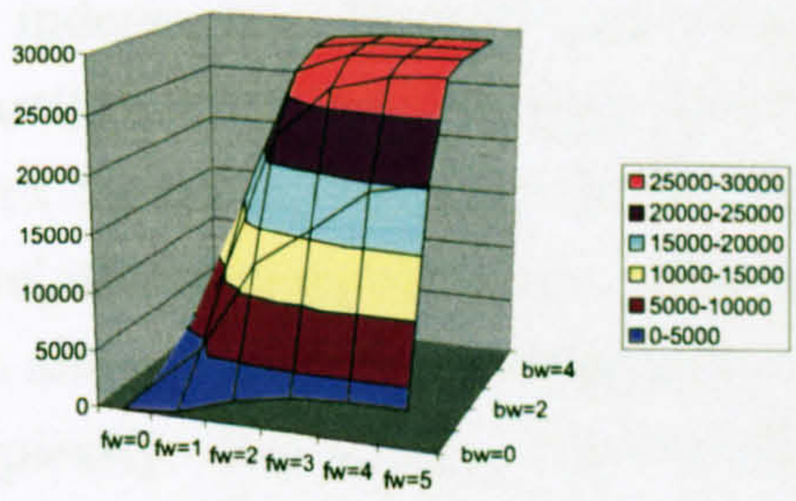
Tab. 4.4: The number of domains for the XMark dataset containing 32,864 vertices based on local bisimilarity

of domains increases with the tree depth parameter.

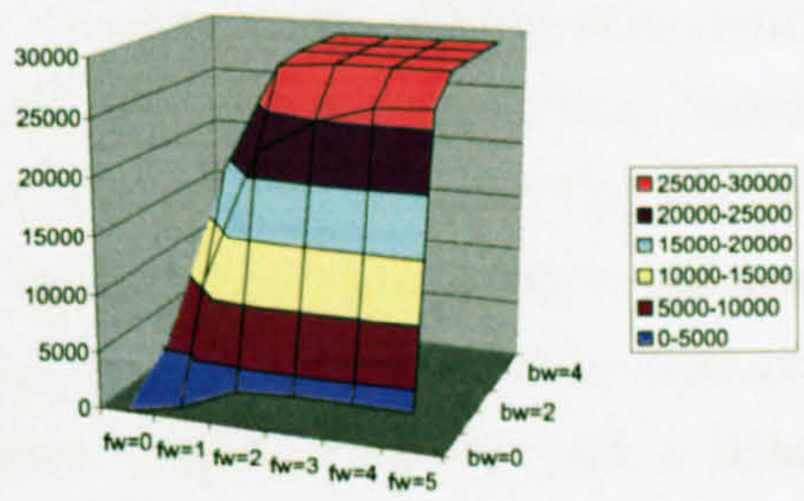
The analysis of the domain cardinality does not point to any distinguished points in the parameter space. Thus only limited conclusions can be drawn here. Firstly, for the source used for this experiment the domain structure becomes unpractically large even for moderate parameters for tree depth, forward and backward bisimilarity. Finely-grained indices based on such data groupings will approach the size of the data graph itself, thus undermining their initial purpose of summarisation. This conclusion is in line with the observation performed by Kaushik et al. [KB+02]. In this particular example, there exist 28,873 different non-empty classes of vertices which can be distinguished by structural path expression, out of the total number of 32,864 vertices in the data graph. The area of the parameter space in which this finest possible classification is reached is shown in red in Table 4.4. If one is to make practical use of data groupings based on bisimilarity, one either has to limit the aspects of the data source being classified or limit the length of the local bisimulation to very short paths. Both approaches were detailed in Kaushik's work. The first depends on detailed knowledge of the



(a) Tree Depth $td = 1$



(b) Tree Depth $td = 2$



(c) Tree Depth $td = 3$

Fig. 4.10: The number of domains for the XMark dataset containing 32,864 vertices based on local bisimilarity

application of the data. In the absence of such information only the second alternative is practical, which will serve as a foundation for the work presented in Chapter 6.

Another conclusion one can draw from the normal distribution of Figure 4.10, is that its source is synthetic, i.e. randomly generated. As the synthetic XMark document is the only truly graph shaped example source available for this experiment, the proof that real-life, graph-shaped sources will behave differently can not be provided here. It remains as a task for the future, to prove or disprove this assumption once more suitable data sources become available.

4.4 Summary

This chapter has re-evaluated several works on efficient processing of SSD, in particular research on indexing, in terms of the data classifications they imply. It is important to note that practically all research in this area implies a form of data grouping in one way or another, thus supporting the central statement of this thesis. This chapter in particular contrasts and compares the set theoretical foundations behind these varied clusters and builds up a concept of domains and domain structures. Though strictly speaking domains for SSD can only be inferred in the presence of an application, it has been shown that vertex classifications based on the data graph alone can serve as valuable approximations thereof. Equivalence domain structures present a special case of these application independent domains and are based on an equivalence relationship defining a partition over the vertex-set of a data graph. From these the group based on vertex bisimilarity is most flexible in its application. By limiting the bisimilarity relationship to paths of a fixed length, one gains an optimisation parameter that allows trading the coarseness of the resulting domain structure against its complexity. For a data structure like an index graph based on such a domain structure this means one can trade index precision for index size.

5. IMPROVING QUERY PROCESSING USING DATA COMPRESSION

Occam's Razor

Pluralitas non est ponenda sine neccesitate.
Entities should not be multiplied without necessity.

William of Occam, ca. 1285 – 1349

The previous chapters introduced a model for efficient processing of SSD based on data grouping and developed the theory behind various such groupings. This chapter will show how a specific grouping can help to improve the physical data representation without restricting query performance in the desired application area.

Data compression can be used in order to improve query performance in the relational field. Research [CMW98, CB03] in this area has shown that it is beneficial to represent a table as a set of columns and compress each of these columns separately, thus exposing its homogeneity. The approach is particularly useful for queries with very low selectivity over large datasets. Such settings commonly occur in areas of science that are now being investigated in the context of SSD management.

It is thus useful to investigate how compression can be adapted towards SSD management systems in general and XML in particular. Here the results from the implementation of a prototype compressed Document Object Model [Neu02, NW02] are presented in the wider context of the efficient processing of SSD. The approach chosen requires the translation of concepts such as domains taken from the area of relational data management systems to the semistructured model. A relatively simple method was developed that is sufficient for the requirements of the anticipated query class. The research indicates that it is possible to optimise some query operations over compact XML structures using the developed approach.

5.1 Introduction to Querying Compressed Data

Emerging standards such as XPath and XQuery are founded on the vision of XML as both a standard for document and data interchange between applications and also as a structure that may need to be queried directly in much the same way as a database system. Whilst the principles of querying hierarchical data structures were developed early in the history of computer science [TL76], further development of direct querying capability for XML data sources requires close attention to be paid to issues of acceptable performance. Whereas in hierarchical databases the designer had full control over the physical representation and database schema, XML documents are only defined in terms of a loosely defined data model and a textual representation. Physical storage varies widely and schema design is often an ad hoc process carried out by designers with skills in a specific application domain rather than database design skills. Emerging native XML databases (NXDs [Sta01b]) such as those reviewed in Section 2.5.1.4 are designed to make XML applications independent from the physical storage in much the some way as relational databases do. They offer tailor-made storage solutions for XML documents and allow access to the data using a standardised interface such as the Document Object Model (DOM).

The consequences of applying compression directly to appropriate XML data is explored in order to maximise the use of main memory storage in query processing. Due to the inherent performance limitation of external XML representations such as database mappings and textual XML files the research is focused on native, memory internal representations.

5.2 Compression Systems for XML Data

The verbosity and resulting size of XML documents has motivated research in the field of data compression and management. The requirements of compression algorithm in data management systems differ from those in general compressors mainly due to the fact that it is important to preserve random access to the compressed data instance. General purpose compressors usually aim at maximising compression ratios. An overview of general purpose compressors is provided in [LH87]. Some more recent approaches are reviewed here, which are specific to XML processing.

5.2.1 XML Compressors for Storage and Transmission

One of the earliest works on XML compression is XMill by Liefke and Suciu [LS00] and was already introduced in Section 4.1.1. Their concern lies in the area of conventional compressors, i.e. achieving maximum compression for the purpose of efficient transmission and storage. It is based on a transformation, splitting the input document in a number of parallel streams. The main stream, containing the document structure is encoded by using tokens for element and attribute names. The document data is stored in separate containers, by default according to the tag labels of their containing element, though this can be manually specified. These streams are finally compressed using a standard compressor, such as gzip [Ala96]. Optionally the user can specify semantic encoders for specific data types, e.g. differential, run-length or dictionary encoders. Since this is a sequential algorithm it is unsuitable for direct querying. However, this is the first work that explicitly recognises that SSD cannot be modelled as a homogeneous source, but should be treated as a structured data source.

Millau [GS00, SM01] is a sequential compressor that is based on the WAP binary XML (WBXML) content format [WBX01]. Millau is specifically aimed at data transmission, enabling stream based access via SAX and DOM interfaces and consequently avoiding a second parsing at the receiving system. The extended version [SM01] introduces some interesting concepts like differential DTD encoding and DTD patterns. However, the impact of those concepts on the compression performance is found to be limited.

Cheney [Che01] systematically analyses the effects of diverse compression algorithms on XML data. In addition to the models defined by XMill Cheney introduces a new data model based on SAX parsing events. This model results in some new pre-compression transformations, corresponding to data groupings in terms of this thesis, whose effect on a multitude of standard compressors is measured. The main result shows that the choice of a compression algorithm has a significant impact on the success of pre-compression transformations and vice versa. It is shown that compressors with a large compression context or grammar-based compressors perform equally well without such prior transformation as performed by XMill. A large compression context in this sense is a window of a sliding window compressor that is significantly larger than the size of the regular building blocks of the source, or a similar block size of a block-sorting algorithm.

5.2.2 XML Compressors for Querying and Management

Tolani and Haritsa [TH02] describe a semi-static, query friendly compression scheme for XML, which allows direct querying of the compressed representation on the element level. Metadata and enumerated attributes are dictionary encoded, all remaining data is individually Huffman encoded. Statistics for the Huffman encoding are collected for atomic data contained within nodes carrying equal tag labels corresponding to the domain structure used by XMill. This allows good compression ratios even for small atomic data sections, at the cost of a second pass for compression. This approach defines a homomorphism on the document encoding. Consequently the compressed structures are queried in the some way as an uncompressed source, i.e. by scanning the entire source for the compressed query terms and path names. The same statistics used during source compression are used during the query process and only the result set is uncompressed.

Buneman et al. [BGK03] describe a structural compression strategy based on sharing of subtrees of a hierarchical data source. It is applicable to the structure of SSD documents only, i.e. all atomic data is removed or ignored for the process. The paper describes how to resolve all XPath location steps against this representation in time that is exponential in the size of the query. The mechanism for detecting the shared subtrees is briefly described in Section 2.5.2.3 and serves in Example 4.5 to define skeleton domains.

5.3 Dictionary Compression in Databases

The relational database model is founded on the concept that data can be normalised into regular table structures. This is a useful simplification but many applications, especially Internet-based information systems, require the storage and processing of at least partially irregular data structures. A data model that supports the representation of SSD has the potential to overcome the limitations of relational structures. However, data centric XML applications tend to suffer from poor performance of the underlying technology since this is based on assumptions of document-centricity rather than data-centricity.

MEMBERSHIPS	
MEMBER	ACTIVITY
Miller	Volleyball
Miller	Golf
Smith	Volleyball
Wood	Golf

ACTIVITIES	
ACTIVITY	LOCATION
Rugby	Activities room
Volleyball	Gym
Golf	Gym

Fig. 5.1: The uncompressed example relations

5.3.1 Fundamentals and Assumptions

Compression is a potential route to achieve better performance in SSD processing. The work by Cockshott et al. [CMW98] on relational database systems shows that compression can result in significant performance benefits by moving more of the workload from secondary into primary storage, i.e. from relatively slow disk storage into fast RAM. However, this approach can only be successful if individual data atoms remain accessible. Consequently serial, variable length compression algorithms such as LZW [Wel84] are not appropriate. The dictionary compression method used by Cockshott et al. enabled them to compress the data off-line and resolve queries by decompressing only the output data. Dictionary encoding shows good compression ratios for relational data as well as for verbose XML documents, especially if they are machine generated using typically a rather small vocabulary as assumed in the context of this work.

5.3.2 Compressing Relational Data

The benefit of dictionary-based storage methods is that data can be represented using only minimal bit patterns. At the same time direct addressability is preserved. This is a fundamental requirement for efficient database compression. Figure 5.1 shows simple example relations about sports clubs that will serve as examples throughout this chapter. The data is already in third normal form, i.e. it exposes the functional dependencies of the dataset and is thus fairly compact. However, by using minimal bit strings to represent values, the data elements can be stored in an even more compact form. There is, of course, still the overhead of dictionaries needed to convert the tokens, although these can also be compressed using a second order compression mechanism like the one suggested by Hoque et al. [HMW02, Hoq03].

The relational data would typically be stored in tables with fixed length at-

MEMBERSHIPS	
MEMBER	ACTIVITY
00	01
00	10
01	01
10	10

ACTIVITIES	
ACTIVITY	LOCATION
00	0
01	1
10	1

Fig. 5.2: The compressed example relations

ACTIVITY	
Token	Lexeme
00	Rugby
01	Volleyball
10	Golf

MEMBER	
Token	Lexeme
00	Miller
01	Smith
10	Wood

LOCATION	
Token	Lexeme
0	Activities Room
1	Gym

Fig. 5.3: Dictionaries of the compressed example relations

tributes. Thus the table **ACTIVITIES** would have a size of (10+15) characters ×3 tuples ×8 bits/character = 600 bits. Inspection of the tables suggests that it would be possible to represent the information content in a more compact form by using codes to represent the attribute values rather than using the domain values themselves in the relation. The attribute ACTIVITY contains three different values. Therefore, in its most compact form, it could be represented as a two bit integer. Since there are only two different values in the LOCATION column, it could be represented as a one bit integer, i.e. as a boolean value. The compressed integer representation of the relations is shown in Figure 5.2.

The effect of representing data in this encoded format is to reduce the space occupied by **ACTIVITIES** to 9 bits and **MEMBERSHIPS** to 16 bits. It is necessary to add the size of the dictionaries to this, which allow the conversion between codes and attribute domain values and vice versa (Figure 5.3). Dictionaries can be represented as lists of domain values. In the case of all non-unique attributes the number of entries in the dictionary will be less than the number of tuples in the relation. Typically only the key field will be unique, whereas most other attributes will range over a fairly restricted domain in the context of machine generated data sources.

```
<?xml version="1.0" ?>
<club>
      ...
      <activities>
          <activity>
              Volleyball
          </activity>
          <location>
              Gym
          </location>
      </activities>
      ...
      <memberships>
          <member>
              Miller
          </member>
          <activity>
              Volleyball
          </activity>
          <activity>
              Golf
          </activity>
      </memberships>
      ...
</club>
```

Fig. 5.4: The sports club example data represented as XML document

Type	↗
Document	-
Element	1
...	
Element	2
Element	3
Text	2
/Element	3
Element	4
Text	2
/Element	4
/Element	2
...	
Element	5
Element	6
Text	1
/Element	6
Element	3
Text	2
/Element	3
Element	3
Text	3
/Element	3
/Element	5
...	
/Element	1
/Document	-

↘	Element
1	club
2	activities
3	activity
4	location
5	memberships
6	member

↘	Text:activity
1	Rugby
2	Volleyball
3	Golf

↘	Text:location
1	Activities room
2	Gym

↘	Text:member
1	Miller
2	Smith
3	Wood

Fig. 5.5: The structure (l.) of the compressed XML document together with the associated dictionaries (r.)

```
<?xml version="1.0" ?>
<club>
        ...
        <activities>
            <activity>
                Volleyball
            </activity>
            <location>
                Gym
            </location>
        </activities>
        ...
        <memberships>
            <member>
                Miller
            </member>
            <activity>
                Volleyball
            </activity>
            <activity>
                Golf
            </activity>
        </memberships>
        ...
</club>
```

Fig. 5.4: The sports club example data represented as XML document

Type	↗
Document	-
Element	1
...	
Element	2
Element	3
Text	2
/Element	3
Element	4
Text	2
/Element	4
/Element	2
...	
Element	5
Element	6
Text	1
/Element	6
Element	3
Text	2
/Element	3
Element	3
Text	3
/Element	3
/Element	5
...	
/Element	1
/Document	-

↘	Element
1	club
2	activities
3	activity
4	location
5	memberships
6	member

↘	Text:activity
1	Rugby
2	Volleyball
3	Golf

↘	Text:location
1	Activities room
2	Gym

↘	Text:member
1	Miller
2	Smith
3	Wood

Fig. 5.5: The structure (l.) of the compressed XML document together with the associated dictionaries (r.)

5.3.3 Compressing Semistructured Data

One possible representation of the example data as an XML document is shown in part in Figure 5.4. It can be seen that the complete representation would contain almost the same redundancy that is exploited in the approach shown above for relations. The only reduction in redundancy is achieved by allowing set valued attributes, here two `activities` are stored below one `member` entry. But there also exists further redundancy caused by repetitions in the document structure. Depending on the degree of flexibility of an associated document schema, provided either in the form of a DTD or XML Schema document, the structure may be known in advance. Even if only well-formedness is assumed, the name of any closing tag is guaranteed by the XML syntax and thus redundant.

The structure of the compressed representation is shown in Figure 5.5. The implementation of the compression strategy results in the structure being separated from the content. The tokenized structure representation is very closely related to the textual representation, allowing mixed content, comments and other XML specific data items to be included, which lie out with the data model provided in Section 2.2.2. They were included in the original research [Neu01] for reasons of standard compliance and will be ignored as part of this re-evaluation. As in the original XML document, the order of individual entries is important as it encodes sibling order. In combination with special start and end tags it also encodes ancestor/descendant relationships. References from the structure point to different dictionaries that store the document data. Note that the metadata, e.g. element tag labels and attribute names, are stored in a global context or domain, whereas the data content is stored in context dependent dictionaries corresponding to atomic domains as defined by their parent tag labels. This allows related information to be kept together as proposed by the model introduced in Chapter 3. The concept of compression by column of the relational approach is replaced with that of compression by containing element node in case of semistructured data. Multiple entries of the same string within one dictionary domain are avoided, thus reducing the redundancy. The compression algorithm used for the prototype's data structure is applied equally to data and metadata. This approach results in a significant reduction in the volume of the data stored.

5.3.4 Querying Compressed Data

There are two fundamentally different ways to query compressed data sources. The entire data can be decompressed and then queried in its uncompressed form. Alternatively, one can compress the query and then resolve it on the compressed data, decompressing only the result set.

Using the first approach, it is possible to benefit from compression only if the retrieval of the compressed file followed by its decompression takes less time than the retrieval of the larger, uncompressed version. However, the smaller the selectivity of a query, the less efficient this approach will be. Many queries in data-centric applications will return only a small subset of the actual data. Thus it is more desirable to translate the query itself into the compressed domain and only to return and decompress the actual result set in such an environment.

In the case of dictionary compression this is very easy. The lexemes occurring in the query are sought in the document dictionaries. If no matching dictionary entry exists, the query will yield no result. If matching tokens for the lexemes exist, these in turn are sought in the compressed document structure. Comparisons of these short binary tokens are typically faster than string comparisons using the uncompressed representation.

5.4 Experimental System Design

The Dictionary compression based Document Object Model (DDOM) is based on the architecture described in the previous sections and is implemented in Java. It supports read-only access on a document once it is parsed or generated. A structure such as this is appropriate for the targeted application areas, such as mining of large, practically static, scientific data sets.

5.4.1 Storage

One of the major differences between dictionary compression in relational systems and SSD is the definition of the dictionary domains corresponding to atomic value domains introduced in Chapter 4. In the relational case this is relatively simple. Every attribute of a given relation is associated with its own domain. All values of a given attribute belong to the associated domain. More than one attribute may share one domain, although this can be hard to detect automatically. In the case of XML data, the concept of domains is ambiguous as Chapter 4 has shown.

For the work presented here we chose the following approach that is based on domains by parent (Example 4.2). As a first criterion the node type according to the DOM standard is used to distinguish different domains. Element and Text nodes for example, exist in different domains. Additionally, atomic vertices of the data graph, i.e. text nodes and attribute values of the DOM tree, are stored in subdomains defined by their immediate parent node. In the example document, the values of the Text nodes for "Miller", "Smith" and "Wood" are stored in the domain `Text:member` as shown in the right part of Figure 5.5. Note that this is a technical simplification due to the fact that no application semantic is attached to the data. Consequently domains remain arguable as Chapter 4 has shown. In the XML version of Shakespeare's plays described in Appendix C.2 for example, `TITLE` elements occur in many contexts, for example inside the `PERSONAE`, `ACT` and `PLAY` elements. These form semantically different domains (e.g. *play titles* opposed to *act titles*). However, syntactically there is just one definition of the `TITLE` content model in the corresponding document schema. In the approach presented here all titles would be stored in one common dictionary, following the syntactical or application independent definition. This also helps to avoid the creation of too many sparsely populated dictionaries. The problem of identifying which elements belong to which domain cannot be resolved automatically in the absence of the application semantics and is further discussed in Chapter 4. The approach being used here is based on the simple approach of Example 4.2. The experimental results of Section 5.5.1 shows that it works well for data-centric documents that usually do not contain highly nested data structures.

The difficulty of separating individual domains also influences the representation of the structure. The structure of a document is stored in the form of an array as shown in the left part of Figure 5.5 with both columns represented by integer number types of appropriate length. The DOM node type of an entry is stored in a 8-bit integer for reasons of technical simplicity arising from the chosen implementation platform of Java. This could be easily reduced to five bits, four bits to indicate one of the twelve possible DOM node types [DOM00] and one bit to distinguish opening and closing tags. More importantly is the fact that references to all possible domains occur in the structure array. This is the result of domain mixing caused by the chosen characterisation of domains. It is impossible to use minimal bit patterns to store the references in such a design. The experimental system uses 32-bit integers, which represent a significant waste, especially for domains with very low cardinality.

The structure array together with the associated dictionaries contain the entire data of an XML document. No tree of DOM nodes is stored to reduce memory consumption. Only the **Document** node that contains the compressed structure and the dictionaries exists at any given time. However, the methods supplied by the DOM interface are required to return **Node** objects. These are generated dynamically and contain only a reference into the structure array. As soon as no further external reference to such a **Node** object exists, it can be garbage collected. Internally the DDOM uses methods that work directly on the structure array to avoid the overhead of frequent object generation and destruction. Externally however this mechanism is necessary to achieve DOM conformance.

5.4.2 Querying

The DOM interface does not support querying directly beyond providing a method *getElementsByTagName* for the DOM node types **Document** and **Element**, which allows the selection of descendant nodes by name. The implementation of this method in the prototype system follows the idea of doing as much work in the compressed domain as possible. Hence the tag label that is passed in as an argument is sought in the dictionary containing the tag label alphabet and its compressed representation is then sought in the corresponding part of the structure array. In the basic implementation, this is done by linear scanning, resulting in a $\mathcal{O}(n)$ runtime behaviour. For a scan over parts of the structure array corresponding to the entire document or element nodes with many descendants, n is of significant size. Consequently it is desirable to improve upon this by using indices. This will be described in Section 5.4.3. However, even in the absence of indices the prototype implementation avoids costly string comparisons and is able to detect empty results for queries for non-existing tag labels at an early stage in query processing.

Complex queries need to be resolved using additional query support. Two different engines were used in order to perform the experiments described in Section 5.5.2. The first is an external XQL query engine, which accesses the structure using the provided DOM interface. The other is a purpose built query system that makes use of the advanced representation of DDOM. These are described in the experimental Sections 5.5.2.1 and 5.5.2.2 respectively.

5.4.3 Indexing

To avoid linear scans for tokens in the structure array, one can search for them in the domain dictionaries instead and amend each entry with a list of positions in the structure array, where the corresponding token occurs. Due to the table-like structure the dictionaries and the linear address-space of the structure array this can be implemented using techniques known from the relational world, i.e. by implementing the dictionaries in form of B-Trees or inverted lists. Because both the tag label alphabet and the individual atomic value dictionaries are compressed and managed using the same dictionary approach, the same mechanism can be used for indexing. The created index structures act as a value index in case of the atomic domain dictionaries and as as tag label index in the case of the metadata dictionaries, i.e. element labels and attribute names.

If not only the positions of the start of a node in the structure arrays is stored in the index, but also the corresponding end position, one can quickly verify whether a given data atom is a descendant of a particular element node or not. This will be described further as part of the hybrid design presented in Chapter 6. For the remainder of this chapter indices will only be used in order to allow a true bottom-up search based on atomic values, i.e. to identify positions in the structure array where a particular data atom actually occurs. Following this initial step linear scans will be used as described in the previous section in order to resolve the remaining parts of a given query.

5.5 Performance Analysis

Test documents of various sizes were generated from a database used by a domain name server (DNS). This sources is described in Appendix C.1. It is a fairly regular source, whose entries describe properties of servers known to the domain name server. A few of the properties of each server are optional. These are represented by **null** values in the relation presentation but are omitted in its semistructured representation.

5.5.1 Memory Consumption

Figure 5.6 and Table 5.1 shows the memory consumption of several representations of the same XML data as a function of the database cardinality. Compressed and uncompressed textual XML documents are compared with different DOM

Imlementation	100 Entries	1k Entries	10k Entries	100k Entries
XML/gzip	1.4 KB	9.7 KB	83 KB	890 KB
XML	21 KB	213 KB	2100 KB	20700 KB
DDOM	622 KB	989 KB	3550 KB	23761 KB
DDOM+Index	649 KB	1197 KB	5869 KB	41690 KB
Xerces	210 KB	1659 KB	14297 KB	140231 KB
Crimson	195 KB	1916 KB	19300 KB	191000 KB

Tab. 5.1: Memory consumption of different representations of the DNS data

implementations, all of which using Java as common platform. The DDOM implementation requires less memory than the widely available Xerces and Crimson implementations[1] for all but the smallest document size. The developed system incurs an overhead with small documents. Both, Xerces and Crimson require the heap to be enlarged from 64 MB to 256 MB in order to process the largest document in this experiment. Note that the index structure described in Section 5.4.3 adds very little overhead to the DDOM representation in terms of space. Even the relatively small document with only 1000 entries is still smaller in the compressed and fully indexed version than in its conventional, non-indexed representation. However, even the basic DDOM representation required still more memory than the textual representation. This is partly caused by the Java implementation, which always uses 16 bit representations for individual characters whereas the textual XML file is using 8 bit encoding. Both conventional DOM implementations show a linear growth with database cardinality. By contrast, the DDOM shows sub-linear growth. The gzip compressed file size is used as a practical measure of the document entropy. The graph clearly shows that there is still a large potential for further memory savings.

5.5.2 Query Performance

Measuring the performance of a storage/query system for XML in itself is somewhat complicated. The requirements of different applications vary widely. This is reflected in the large number of XML query languages [BC00]. Standard benchmarks for XML databases are just emerging [SW+01]. Since XML is based on the idea of documents, document-centric storage systems are further developed than data-centric approaches. Queries in the document-centric domain are typically limited to locating entire documents that match certain requirements or

[1] Both available at `http://xml.apache.org`.

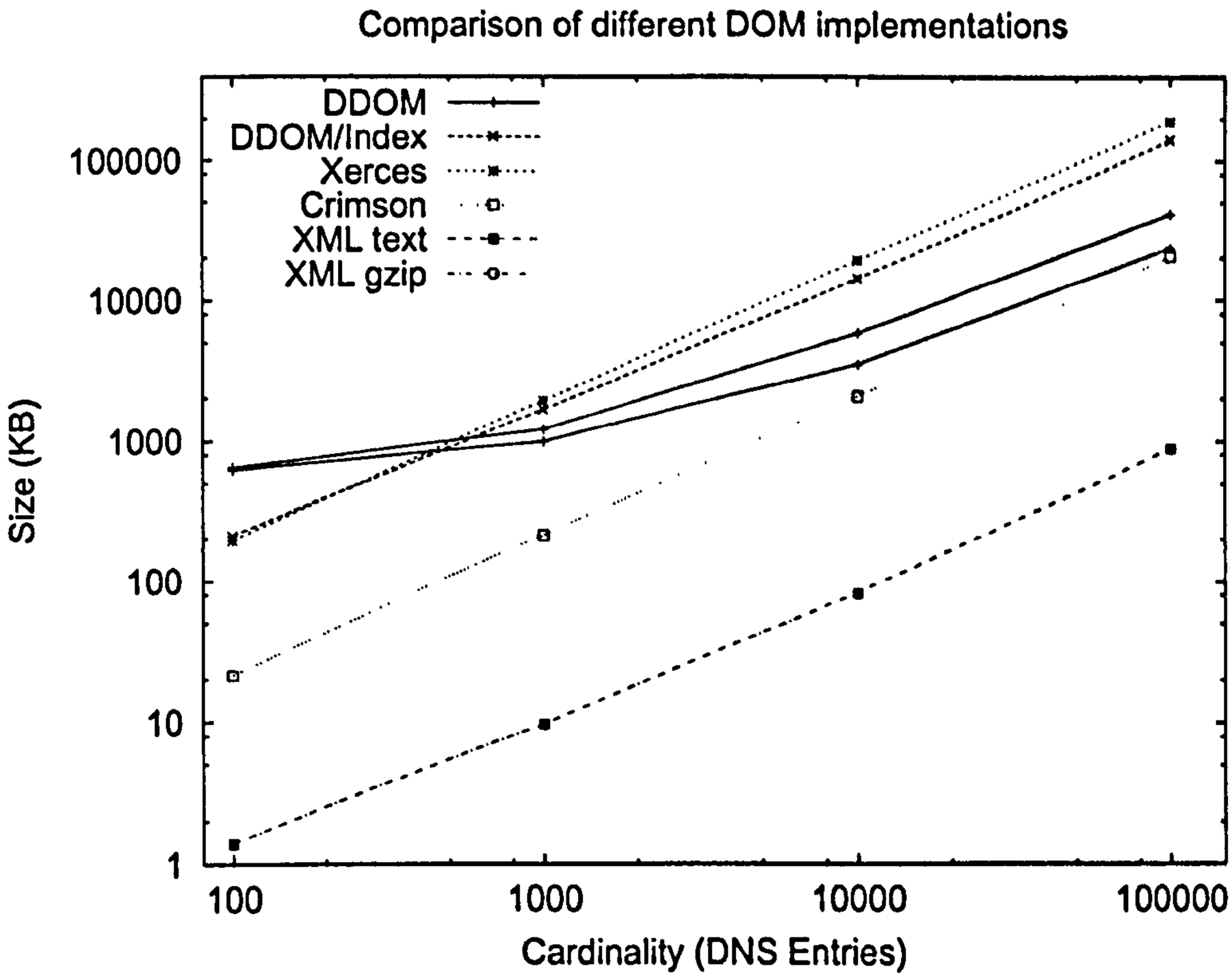


Fig. 5.6: Memory consumption of different representations of the DNS data

to the execution of relatively simple transformations. Therefore, storage is often based on information retrieval systems using full-text indices. In contrast, data-centric applications frequently use relational databases as a back end. Queries in such systems are typically aimed at retrieving only a small fraction of a set of documents. Since this work concentrates on data-centric applications, only the performance of queries with low selectivity is analysed.

5.5.2.1 Querying using an external query engine

At first the performance achievable by the prototype implementation was compared with other approaches using standard interfaces and query languages. In this case the query engine is external to the implementation and unaware of the underlying compression approach. This is problematic as the only comparison possible is the measurement of wall-clock time. Thus, the quality of the actual implementation could mask fundamental technical differences. It is however, the only method possible which is based on standards alone and thus reproducible.

Query Q as BPE Selectivity $\sigma \times 10^{-3}$	//LEVEL3 34.8	//LEVEL4 0.4	//LEVEL5 0.0
Xerces/DOM	2	4	2
DDOM/DOM	152	114	2
Xerces/XQL	2787	2297	2231
DDOM/XQL	7325	6017	3731
Xindice simple	13999	6944	6720
Xindice complete	15825	9347	9514

Tab. 5.2: Query execution times in ms for different selectivities and query systems

The implementations chosen for the comparison are publicly available and widely used and thus considered to be reasonably well designed and implemented.

Figure 5.7 and Table 5.2 show the results of a simple node selection query based on tag labels. The query chosen can be resolved in a single step by all systems tested. Furthermore, the cardinality of the result can be controlled easily by the tag predicate used. The results indicate some of the problems and possibilities in terms of the performance of queries posed on XML data. Querying for the same results using different queries mechanisms, storage strategies and implementations shows a wide range of performance variations. The measurements were performed using a single XML document containing 10,000 entries from the DNS database, i.e. a source representing information about 10,000 individual servers. This corresponds to a data graph with 182,896 vertices that does not contain any additional graph edges, i.e. is tree shaped. The query selects nodes with a certain tag label, in this case the nodes containing the fourth, fifth and sixth component of the domain names of each individual server. The selectivity of this query is decreasing as almost all domain names contained in the example document have four parts, but none has six parts. The chosen query provides the chance of comparing the DOM interface directly to any higher level query system. It is the most commonly used selection operator in XML query processing, forming a part of practically all more complex queries, and can be used to show the variance in performance achievable.

The DDOM implementation was compared to the Xerces DOM implementation and also to the native, disk-based XML database Xindice[2] [Sta01a]. Firstly the DOM method *getElementsByTagName("LEVELn")* was used to resolve the query. It was called on the document element and directly returns the required results in form of a **NodeList**. Due to the DOM interface restrictions this will

[2] Available at `http://xml.apache.org/xindice`

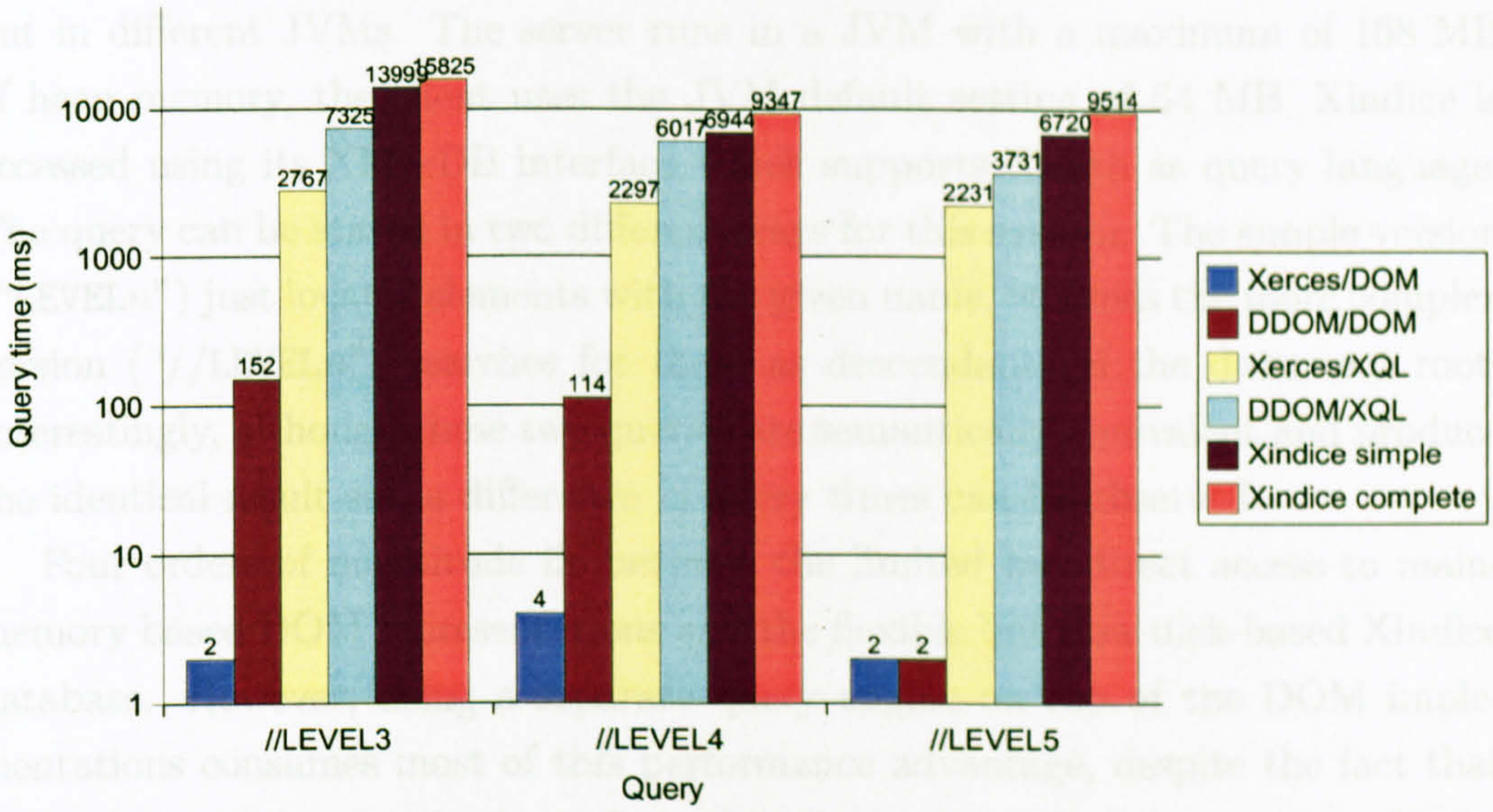


Fig. 5.7: Query execution times in *ms* for different selectivities and query systems

include the instantiation of one **Element** object per result for the DDOM implementation.

To allow a comparison with a more realistic system, that would be capable of handling more complicated queries, the query was repeated with an XQL query engine on top of the two DOM implementations. XQL is a predecessor of the current XQuery proposal and similar in syntax and semantics to the branching path expressions introduced in Section 2.3.1. The corresponding query in XQL is identical to its branching path expression and is shown below:

Query 5.1: //LEVELn

The query engine used was taken from the GMD-IPSI XQL Engine[3] implementation and is not optimised for use with either of the implementations and thus restricted to the standard DOM interface. DOM representation, query engine and the proprietary query application run in a single Java virtual machine (JVM) with a default maximum heap memory space of 64 MB.

Finally the measurements were repeated using the native XML database. No indexes were generated to allow a comparison with the DOM implementations that also do not use indices. For Xindice, measurements were performed on a

[3] Available at `http://xml.darmstadt.gmd.de/xql`.

warm cache. Database engine and query application run on a single computer but in different JVMs. The server runs in a JVM with a maximum of 168 MB of heap memory, the client uses the JVM default setting of 64 MB. Xindice is accessed using its XML:DB interface which supports XPath as query language. The query can be stated in two different ways for this system. The simple version ("LEVELn") just locates elements with the given name, whereas the more complex version ("//LEVELn") searches for these as descendants of the document root. Interestingly, although these two queries are semantically equivalent and produce the identical result set, a difference in query times can be observed.

Four orders of magnitude lie between the limited but direct access to main-memory based DOM representations and the flexible but slow disk-based Xindice database. However, using a separate query engine on top of the DOM implementations consumes most of this performance advantage, despite the fact that the entire querying process is performed in the computer's main memory. It also becomes apparent that the DDOM implementation suffers from the dynamic object creation as its performance diminishes with higher query selectivity. This limitation is exaggerated by the fact that the external XQL engine performs four complete traversals of the DOM tree using only methods supplied by the **Node** interface, rather than using the more specialised methods of the **Element** or **Document** nodes that are able to perform the required task much faster.

5.5.2.2 Querying using a custom-built query engine

In order to separate the influence of the interface and external query engine from those of the data representations on the query performance, a simple query engine was implemented, which works directly on the developed data structure. Unlike the XQL engine used for the previous experiments, this query engine does not provide a query parser, planner and optimizer. Instead it provides direct access to very basic query operators, such as finding the parent or child of a given node. These micro-operations make best possible use of the available data structures, i.e. they make use of indices when these are available and do not instantiate intermediate results. Table 5.3 shows the operations available to this custom-built query engine.

Six user queries were manually encoded using these micro-operations and executed against the different representations of the DNS database with 10,000 entries. These queries are shown in Table 5.4. Figure 5.8 and Table 5.5 show the

Operation	Semantics
$\sigma_T(t : \text{tag}) \rightarrow$ node-set	Selects the set of nodes whose tag label equals t.
$\sigma_V(v : \text{value}) \rightarrow$ node-set	Selects the set of atomic nodes whose content equals v.
$\sigma_{TV}(t : \text{tag}, v : \text{value}) \rightarrow$ node-set	Selects the set of nodes with tag label t containing a direct atomic child node whose content equals v.
$\sigma_{AV}(a : \text{key}, v : \text{value}) \rightarrow$ node-set	Selects the set of attributes with key a and value v.
$\triangleleft(n : \text{node-set}) \rightarrow$ node-set	Selects the set of nodes which are children of the nodes in n.
$\triangleright(n : \text{node-set}) \rightarrow$ node-set	Selects the set of nodes which are parents of at least one of the nodes in n.
node-set $\cap$ node-set $\rightarrow$ node-set	Selects the intersection of the two sets of nodes.
node-set $\cup$ node-set $\rightarrow$ node-set	Selects the union of the two sets of nodes.

Tab. 5.3: The query operations available to the custom-built query engine and their associated semantics

results off this experiment.

The DDOM representation benefits from the decreasing selectivity of queries S1a to S1c. As expected the query times approach zero for empty result sets, as this can be detected early in the dictionary representation. The fact that the Xerces implementation shows a similar behaviour indicates that this implementation probably also makes use of an internal label map.

The results also emphasize the effect the index has on the DDOM structure. The fully indexed version compares quite favourably with the standard Xerces

Q	Query as BPE	Execution plan in μ-operations
S1a	//LEVEL3	σ_T(LEVEL3)
S1b	//LEVEL4	σ_T(LEVEL4)
S1c	//LEVEL5	σ_T(LEVEL5)
S2	//SERVER[/LEVEL?[a]/DATA="strath"]	σ_T(SERVER)$\cap$ $\triangleright(\sigma_{TV}$(LEVEL?[a], "strath"))
S3	//SERVER[/IP1/DATA="63"]	σ_T(SERVER) $\cap \triangleright(\sigma_{TV}$(IP, "63"))
P4	//HOSTNAME/*	$\triangleleft(\sigma_T$(HOSTNAME))

[a] LEVEL1 $\cup \ldots \cup$ LEVEL6

Tab. 5.4: The example queries as BPE and their execution strategies in μ-operations of the native query engine

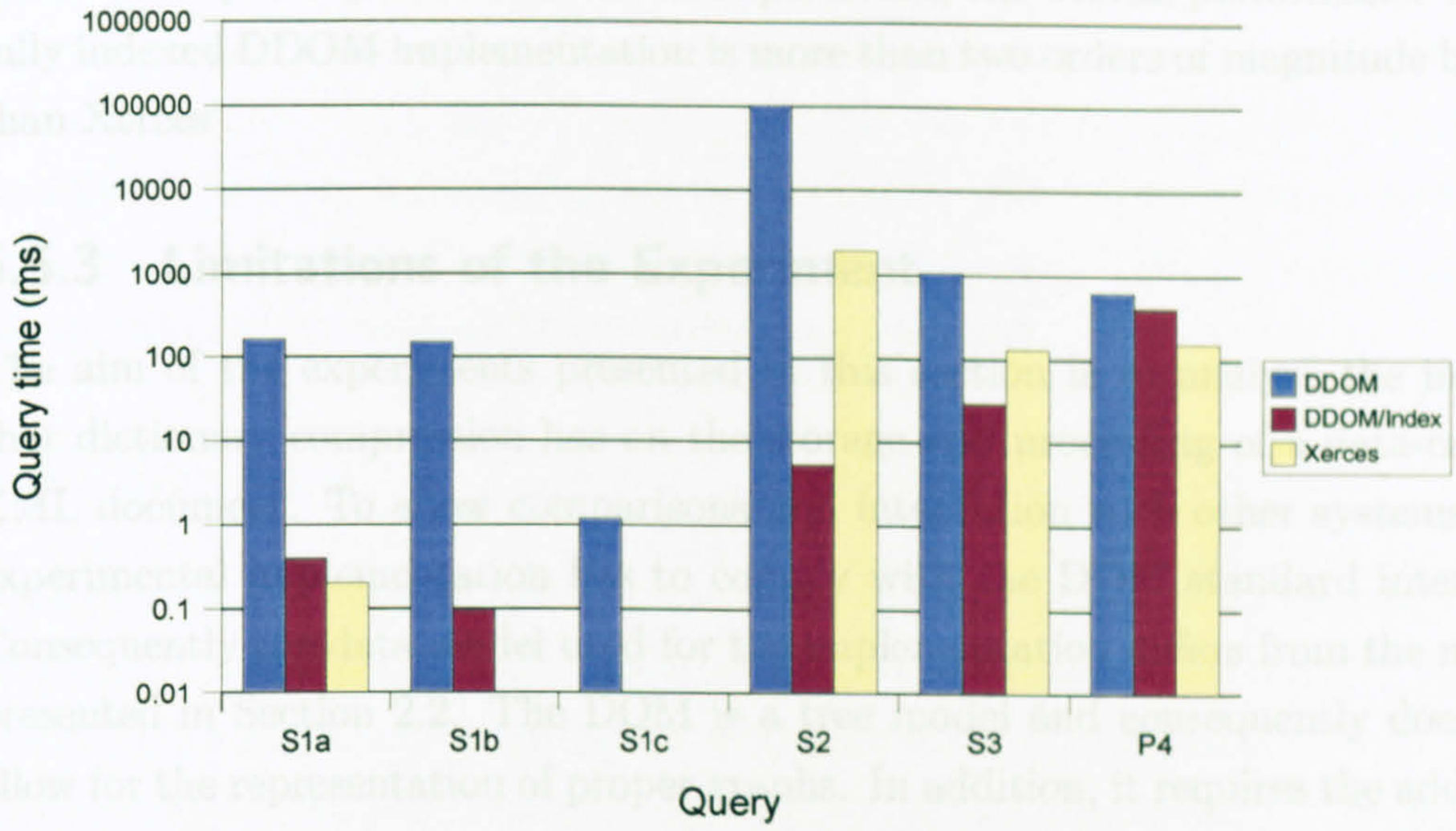


Fig. 5.8: Query execution times in *ms* off the example queries using the native query engine

Query Q	S1a	S1b	S1c	S2	S3	P4
Selectivity $\sigma \times 10^{-3}$	34.8	0.4	0.0	0.1	1.3	54.7
DDOM	164	152	1.2	102000	1059	615
DDOM/Index	0.4	0.1	< 0.1	5.2	29.6	406
Xerces	0.2	< 0.1	< 0.1	1915	130	153

Tab. 5.5: Query execution times in *ms* off the example queries using the native query engine

implementation, despite its lower memory footprint. This is particularly interesting for queries $S2$ and $S3$ as these queries contain structural predicates, which cannot easily be executed against the structure array. However, due to the reduced data quantity of the initial value predicate, the overall performance of the fully indexed DDOM implementation is more than two orders of magnitude better than Xerces'.

5.5.3 Limitations of the Experiment

The aim of the experiments presented in this section is to analyse the impact that dictionary compression has on the storage and processing of a data-centric XML document. To allow comparisons and integration with other systems, the experimental implementation has to comply with the DOM standard interface. Consequently the data model used for the implementation differs from the model presented in Section 2.2. The DOM is a tree model and consequently does not allow for the representation of proper graphs. In addition, it requires the addition of support for document order and the DOM-specific node types, which, for example, provide a distinction of attributes values and character data.

The experiments with external query engines based on the DOM interface demonstrate that this interface severely limits the performance of data-centric applications as shown in Figure 5.7. More importantly the suggested approach only addresses the efficient processing of atomic value based predicate queries, but not the resolution of complex structural queries, which form an interesting aspect of SSD processing. For this reason the method presented here, which is helpful for the resolution of atomic value based queries, will be combined with a technique which is more appropriate for structural predicates. This will be discussed in the following chapter.

5.5.4 Experimental conclusions

The DDOM prototype implementation demonstrates a significant space saving compared with standard DOM implementations. The evidence suggests that the entropy of typical data-centric XML documents is such that considerable savings beyond those achieved by the prototype implementation should be possible. Because of the wide variety of XML-based applications, the influence of the data semantic needs to be analysed more closely to push these boundaries forward.

Adaptive techniques may be possible that compress domain specific data more effectively. At the lowest level, tokens have been presented as integers rather than minimal bit-strings. Further savings in space may be achievable by using optimised data structures.

In the relational domain, Cockshott et al. [CMW98] have previously demonstrated that the dictionary compression in database systems provides significant benefits in terms of enhanced query performance. The results reported here suggest that similar benefits can be achieved for querying XML data structures if the queries can be resolved in the compressed domain.

5.6 Summary

This chapter has detailed the importance of appropriate physical representation of data for a particular task. Efficient querying of data-centric sources was enabled using a dictionary compression technique adapted from the relational world. Dictionary compression requires the identification and separation of homogeneous domains, which was achieved using a grouping of data based on the tag label of their parent node in the distinct spanning tree. It is an example of a physical data reorganisation based on a specific domain definition (Example 4.2) as described in Section 3.2.3 of the chapter on optimising queries over SSD. The experimental results confirm the observation gained from the relational case that storage space can be saved and at the same time queries can be answered efficiently.

The more fundamental results of these experiments is that optimisation techniques designed for the relational world can be adapted and used in the semistructured case if the fundamental concept they are based on exists in both environments. In the example presented these preliminaries are the existence of syntactically homogeneous value domains, which allow the creation of compact dictionaries and aid the execution of atomic value based query predicates. The concept is integral to RDBMS and was addressed in Chapter 4 for the semistructured case.

6. COMBINING STRUCTURAL AND ATOMIC DATA GROUPINGS

Good Design

"Smart data structures and dumb code
works a lot better than the other way around."

Eric S. Raymond, *The Cathedral and the Bazaar*, 1999

This concluding investigation of data groupings shows how different groupings can be combined to allow the resolution of queries combining value and structure predicates. The model introduced in Chapter 3 has shown that different query classes or aspects of queries require different data groupings and thus imply different definitions of domains over the data as explained in Chapter 4. Domain structures based on bisimilarity are useful to resolve structural aspects of a query as explained in Example 4.7. A special case of the parent domain described in Example 4.2 was used in Chapter 5 to address querying of atomic value predicates. Here a hybrid[1] system combining these two approaches is presented.

6.1 Introduction to Hybrid Querying

The analysis of dictionary compression based systems on the one hand and structural indices based on bisimilarity on the other makes clear that although efficient in their own domain, they both fall short of being a solution to the general semi-structured querying problem. Dictionary compression, like all other flattened representations of SSD, can deal with value based predicates efficiently but becomes obstructive for structural queries. Inversely, approaches based on vertex bisimilarity can be used to evaluate structural predicates efficiently but can only

[1] Hybrid in the context of this chapter refers to the combination of structure and value predicates and not to the combination of top-down and bottom-up query execution strategies as detailed in Section 2.3.3.3.

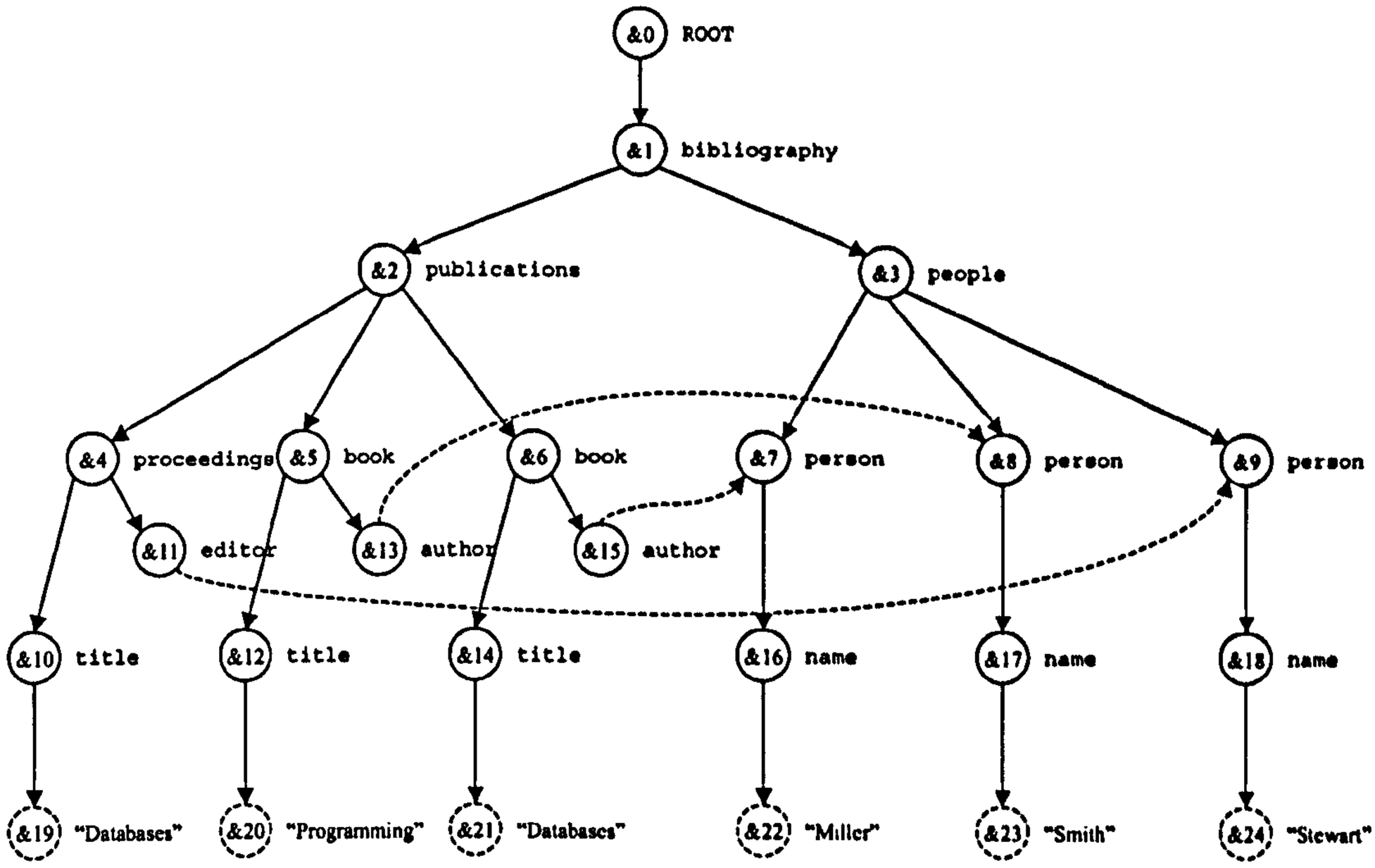


Fig. 6.1: The data graph of the example source

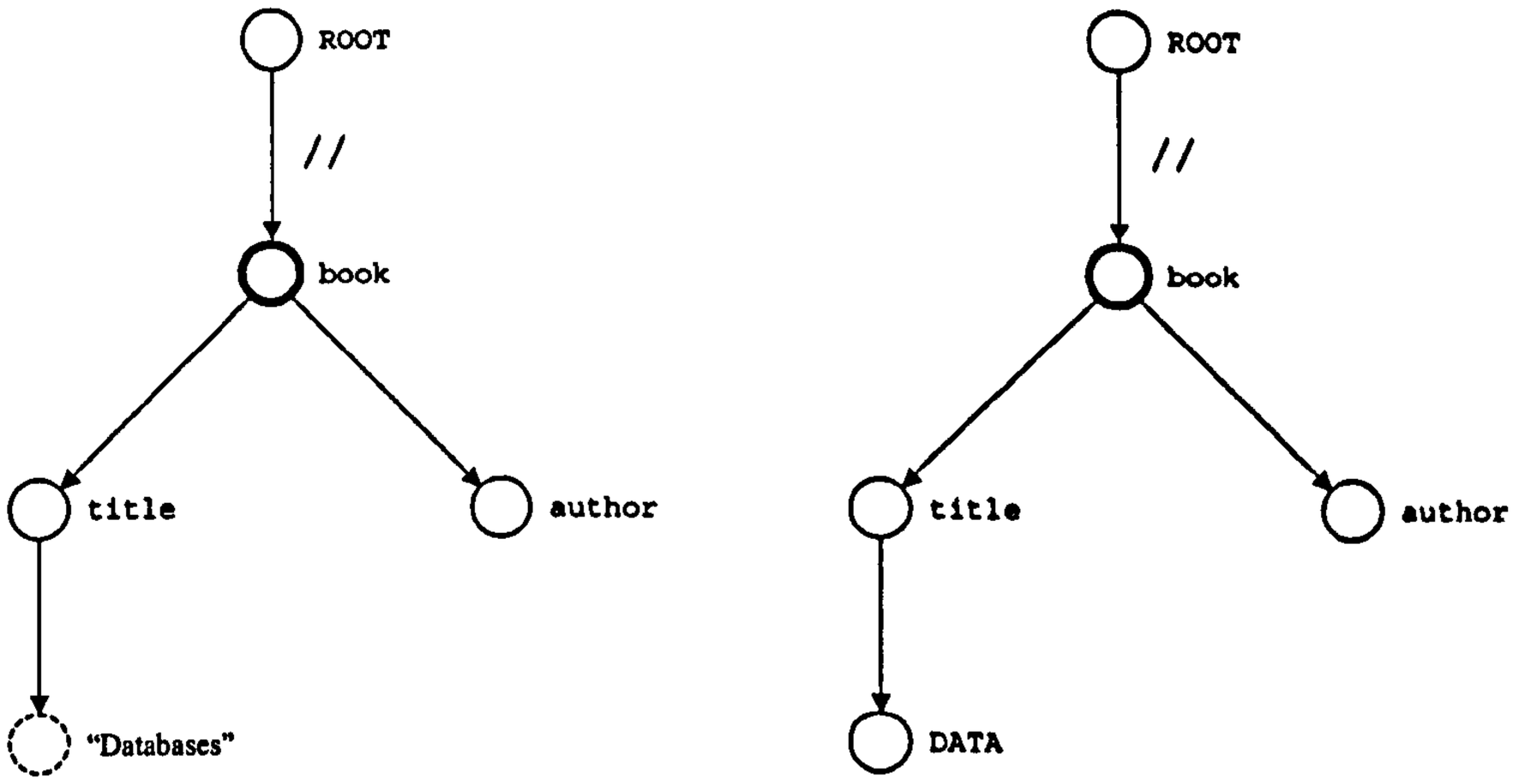
deal with value predicates in a second validation stage, leading to unnecessarily large intermediate results. An integration of the two approaches is presented here. This approach maintains the key strength of its individual components at the cost of adding overheads in terms of redundant data being stored.

6.1.1 Motivating Example

This section describes the evaluation of an example query using the two different approaches mentioned above. However, the query being used contains both structural and value predicates, thus neither approach can provide a covering index for it. The problems encountered due to this fact will be highlighted. Figure 6.1 shows the data graph of the example source. Query 6.1 used on the graph is illustrated in Figure 6.2(a).

Query 6.1 (Books on Databases): `//book[/author & /title/DATA="Databases"]`

Since Query 6.1 contains only forward facing query axes, it can always be answered by a single traversal of the tree [OM+02]. One common technique is to transform the query into an automaton that is driven by the events created



(a) The complete query graph (b) The structural component of (a)

Fig. 6.2: The graph representation of the example query

by an in-order traversal of the data graph, an approach similar to that presented in Appendix A. However, this technique becomes impractical for very large data instances. For that reason index structures are employed by most DBMS.

6.1.1.1 Query Evaluation Using a Structural Index Graph

An index from the family of index graphs defined by the work of Kaushik et al. [KB+02] is selected as a structure index. The condition posed by the incoming descendant arc from the ROOT vertex to the book vertex is trivially true for all vertices in any data graph and thus not considered throughout the querying process. Thus the longest forward facing path in the query graph has length two (book/title/DATA). There are no backward directed paths and the query has depth one as explained in Appendix B. Thus the (2,0)-F+B-index shown in Figure 6.3 is the smallest covering index for the *structural part* of the example query, i.e. the query graph in which all value predicates have been replaced by a structural leaf predicate that selects vertices with the special tag label DATA. The set of vertex identifiers presented next to each vertex of this index graph represents its extend as defined in Definition 2.12. The family of indices does not index atomic data, i.e. no member of this index family is covering for the complete query of Figure 6.2(a).

Embedding the structural part of the query graph shown in Figure 6.2(b) into

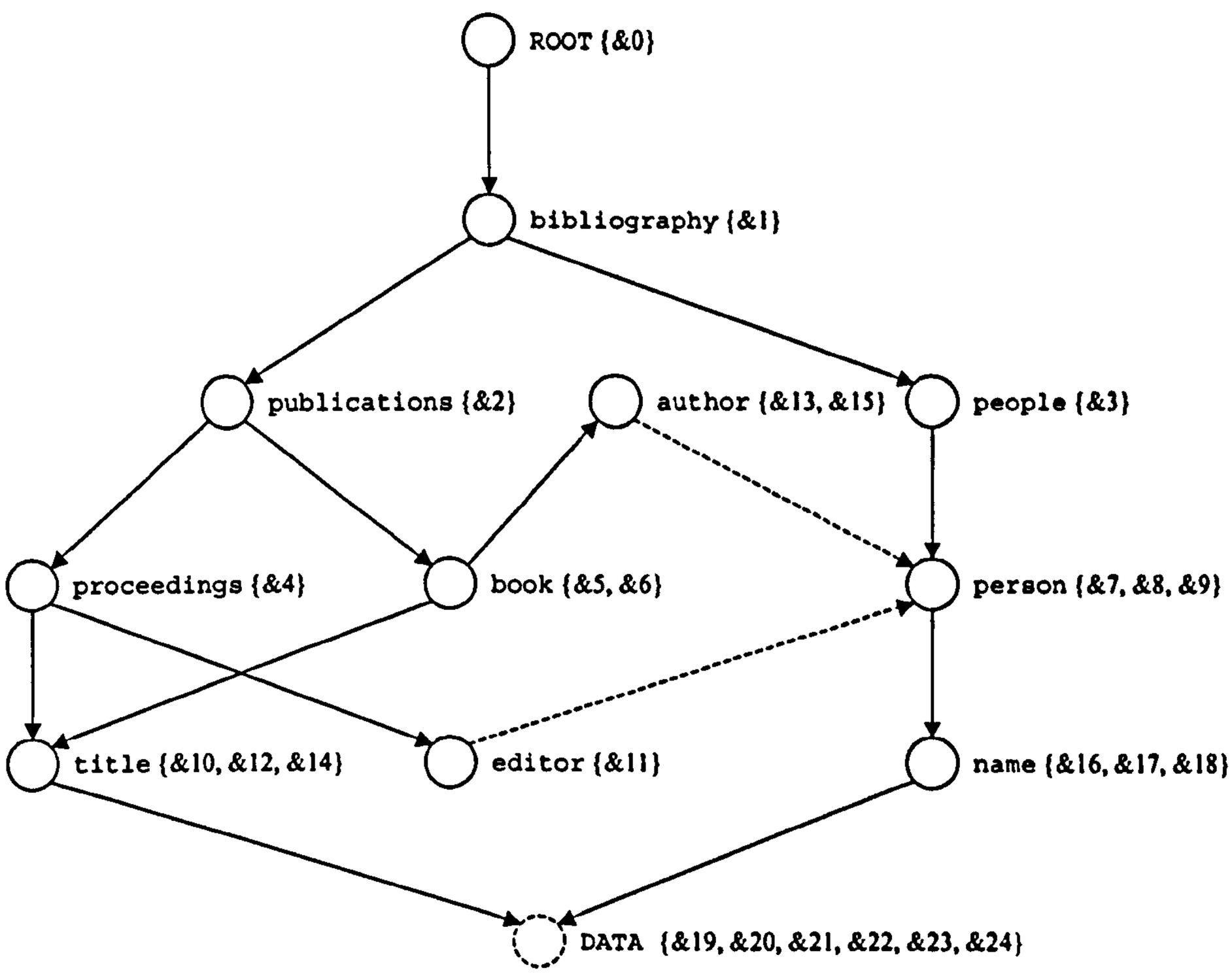


Fig. 6.3: The (2,0)-F+B-index graph used as structural index to the data graph shown in Figure 6.1. The sets of vertex identifiers represent the extent of the index vertices.

the index graph of Figure 6.3 can be done using the same algorithm that could be used to embed it into the data graph since the graphs are bisimilar. This means that the vertices of the index graph have the same properties as the vertices of the data graph with respect to the structural query. Because the index is covering [KB+02] such a computation will result in the same answer. Thus the complexity of the embedding process remains unchanged, but the size of the graph has been reduced from 25 vertices in Figure 6.1 to twelve vertices in Figure 6.3. Such a reduction in size can be expected for most SSD sources, as most practical data graphs contain only very few structural building blocks [BGK03].

Using a top-down strategy one would first look for book vertices in the index graph in which the book predicate is embedded and then progressively embed its adjacent vertices into adjacent predicates of the query graph. In the example provided there is only one book index vertex. It happens to comply with the required structural requirements, thus its extend forms the result of the structural part of the query, which is the set of the two vertex identities &5 and &6. However, only vertex &6 is a result for the complete query including the atomic value predicate. Since atomic values are excluded from the bisimulation, a validation process for every embedding of a path of the query graph containing a value predicate needs to be performed on the data graph. Here, the path starting at the book vertices &5 and &6 in the data graph of Figure 6.1 needs to be followed via a `title` vertex to check whether it contains an atomic vertex with the value "Databases". As a consequence, the original data graph needs to be maintained in addition to the index.

6.1.1.2 Query Evaluation Using Data Dictionaries

Approaching the task from the other end, i.e. starting the query evaluation at the atomic value predicate will be presented in terms of the DDOM approach discussed in Chapter 5. Figure 6.4 shows the fully indexed dictionaries and the structure array of a part of the example source. The format of the index entries differs slightly from the one described in Chapter 5. For the initial description this will be of no relevance and only the parts of the entries presented in bold are used to refer back to the entries of the structure array.

Using this approach on Query 6.1, one can quickly verify that there exist `title` vertices containing the atomic value "Databases", namely the entries at the addresses **8** and **24** in the structure array, corresponding to the vertices

#	Type	↗
0	Document	-
1	Element	1
2	Element	2
3	Element	3
4	Attribute	1
5	Text	1
6	/Attribute	1
7	Element	5
8	Text	1
9	/Element	5
10	/Element	3
11	Element	4
12	Attribute	2
13	Text	1
14	/Attribute	2
15	Element	5
16	Text	2
17	/Element	5
18	/Element	4
19	Element	4
20	Attribute	2
21	Text	2
22	/Attribute	2
23	Element	5
24	Text	1
25	/Element	5
26	/Element	4
27	/Element	2
	...	

↘	Element	Index Entries
1	bibliography	(1:54)
2	publications	(2:27)
3	proceedings	(3:10)
4	book	(11:18), (19:26)
5	title	(7:9), (15:17), (23:25)
6	people	...
7	person	...
8	name	...

↘	Attribute	Index Entries
1	editor	(4:6)
2	author	(12:14), (20:22)
3	id	...

↘	Text:title	Index Entries
1	Databases	8, 24
2	Programming	16

↘	Text:name	Index Entries
1	Miller	...
2	Smith	...
3	Stewart	...

↘	Text:author	Index Entries
1	p2	13
2	p1	21

↘	Text:editor	Index Entries
1	p3	5

Fig. 6.4: The structure array and indexed domain dictionaries as used by DDOM. The bold numbers in the index entries refer to the corresponding start positions of the entries in the structure array.

&19 and &21 of Figure 6.1. Equally one can use the index on the tag name dictionary to verify that there exist book and author vertices in the data graph. The book vertices are represented by entries starting at addresses 11 and 19 in the structure array, the entries for the author vertices start at the addresses 12 and 20. However in order to verify ancestor-descendant relationships between entries, one needs to scan linearly through the structure array. Starting at the start entry of the potential ancestor vertex the array is scanned until either the corresponding closing entry or the potential descendant entry has been found. Only the latter case represents a valid result. Finding parent or child entries follows a similar approach although here the nesting depth needs to be computed for every entry. In the example given, a scan for the first book entry starting at position 11 leads to a `title` entry at address 15, but none of the identified atomic value entries is encountered before the closing tags of the `title` and book entries are found at positions 17 and 18 respectively. Thus this entry, corresponding to vertex &5 of the data graph, does not represent a valid result. The similar scan starting at the book entry at position 19 matches all the required entries from the list of potential descendants, thus the result is valid.

The DDOM approach works on trees only rather than general data graphs. Because of this, entries are nested according to the hierarchy represented by the tree-view of the graph. In the original DDOM prototype, only the starting addresses of entries are contained in the index, corresponding to the aforementioned bold numbers in Figure 6.4. However, if one stores the complete range using the start and end addresses of the subtree rooted by a node as done in Figure 6.4, one can derive the ancestor-descendant relationship using this information alone. In the provided example one can determine that only the second "Database" value node can be part of a valid result, because its address 24 in the structure entry, falls within the range of a book entry, which is (16:26). The other "Databases" node with address 8 however is not contained by the range of any book index entry. This information is clearly useful, especially if the query is restricted to the use of such ancestor-descendant relationships between partial results. Here however a complete path is specified, i.e. parent-child relationships are used. Thus it is still necessary to verify that the `title` node containing the atomic value "Database" is a child of the book node. The same validation step must be performed for the author predicate.

The fact that identifiers can be used to indirectly encode structural relationships between nodes of a tree will be used by the hybrid representation discussed

next. Although this allows the validation of the structural constraints between individual nodes, it still does not allow the selection of a set of nodes based on their structural properties as can be achieved using the index graphs described in Section 6.1.1.1.

6.2 Bridging the Gap: Signatures Based on Numbering Schemes

The approaches described in Section 6.1.1.1 and 6.1.1.2 originate from different perspectives and lack a common element that could be used for their combination. Index graphs allow set-at-a-time operation and maintain structural relationships between vertex-sets but abstract away from the individual vertices of the data graph. Dictionary compression organises data into homogeneous domains and maintains the identity of individual vertices of the data graph but their structural relationships are not exposed directly.

The approach proposed in Section 6.2.3 is based on signatures as an exchange mechanism between both of these approaches. A signature is a compact representation of an important property of a given source. The signatures used here describe structural relationships between tree nodes. Before these are introduced in Section 6.2.2, numbering schemes for trees that serve as the individual entries of the signatures will be introduced in Section 6.2.1.

6.2.1 Numbering Schemes for Tree Nodes

Graph labellings were introduced in Definition 2.5 of Chapter 2 as a one-to-one mapping between the vertex-set of a graph and a set of identifiers. Here two particular schemes used to label the node-set of an ordered tree will be described. The definitions can be equally applied to unordered trees, in this case an arbitrary ordering of the child nodes of every node will suffice. The target domain of these labellings will be the set of natural numbers, thus they will be called *numbering schemes* in order to distinguish them from arbitrary labelling schemes. Note that this always implies an order on the node-set given by the order of the natural numbers identifying them, even if the data model is considered to be that of an unordered tree.

Example 6.1 (Preorder numbering scheme): One example of a numbering scheme for ordered trees is the *preorder numbering scheme* [AHU74]. The tree nodes are

traversed in preorder, i.e. the root is visited *first* followed by a recursive traversal of the subtrees rooted at its children in their given order. The order in which nodes are visited is used as their label. Thus the root node carries the label '0', its leftmost[2] child '1' and so on, as shown in the left hand number of the nodes shown in Figure 6.5(a).

Example 6.2 (Postorder numbering scheme): Another example of a simple numbering scheme for an ordered tree is the *postorder numbering scheme* [AHU74]. This is the opposite of the previous scheme, as tree nodes are visited in postorder, i.e. parent nodes are visited *after* all subtrees rooted by their children have been traversed recursively. Again the order in which nodes are visited is used as their label. Thus the root node carries the highest label, the leftmost leaf in the tree carries the label '0' and so on, as shown in the right hand number of the nodes shown in Figure 6.5(a).

Using the two numbering schemes together as shown in Figure 6.5(a), one can determine the ancestor-descendant relationship between any two nodes in constant time just by looking at their pre- and postorder codes. This fact was described by Dietz [Die82], from whose work the following proposition is taken:

Proposition 6.1: A node x is an ancestor of y iff x occurs before y in the preorder traversal of T and after y in the postorder traversal.[3]

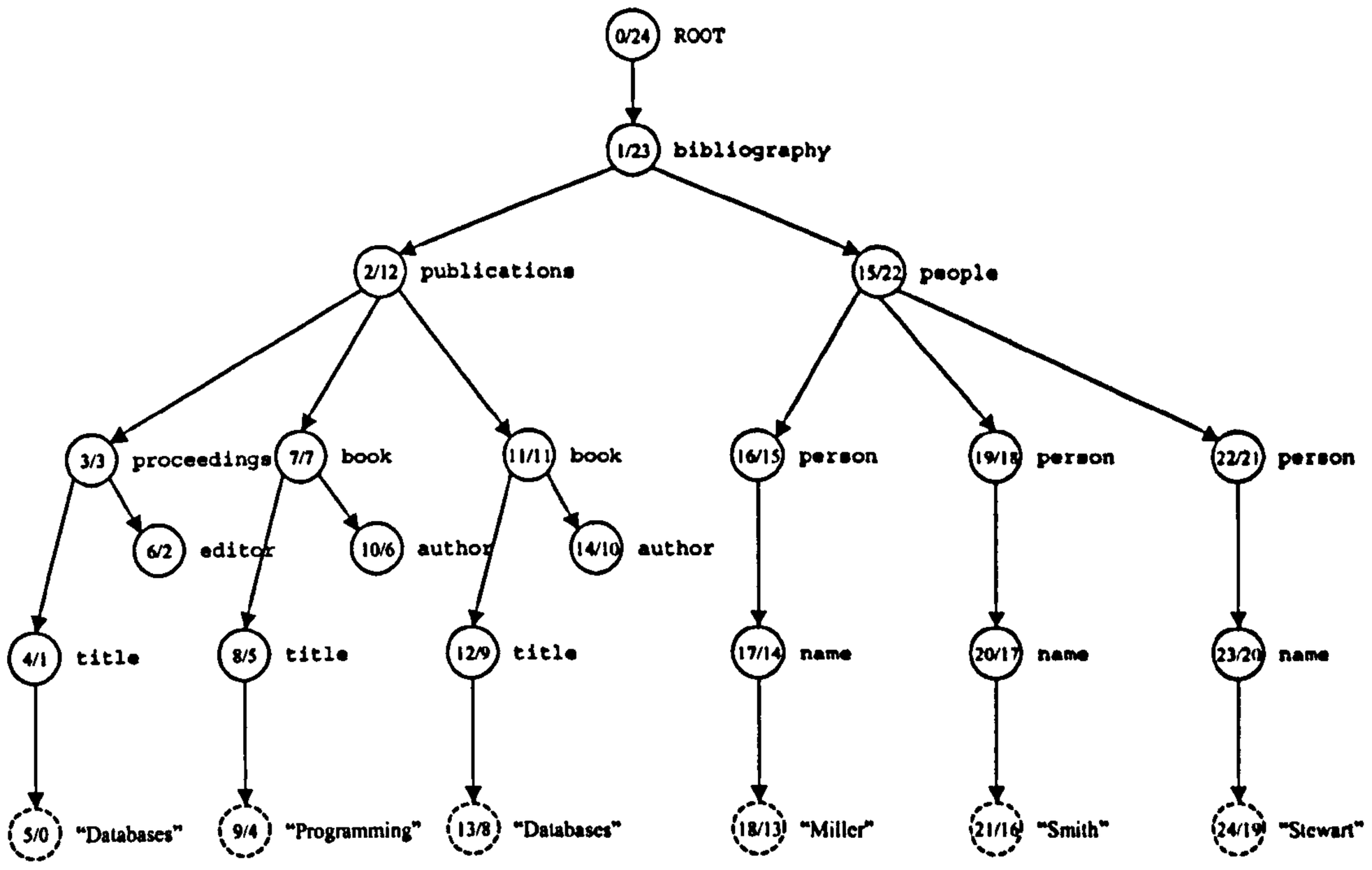
Several researchers [Gru02, ZA+03] use the combined (pre, post)-pairs as identifiers in labelling schemes specifically useful for XML processing. Such works and their extensions for enhanced XML query processing are described in 6.3.1. In order to keep the presentation of the following approach simple, such extensions, usually aimed at particular features of the XML standard like the distinction of attributes and elements, will be ignored for the scope of this thesis, though similar extensions could be applied.

6.2.2 Signatures for Data Trees

A *signature* of the tree-view of a data graph is a compact representation of the structural relationships of its nodes. Figure 6.5(b) shows a signature of the data

[2] assuming an ordering of child nodes from left to right

[3] This differs minutely from the definition provided in Section 2.2.1.3 in so far as that considers a node to be its own ancestor or descendant, whereas the proposition provided here only considers *true descendants*.



(a) An ordered data tree with preorder (left) and postorder (right) labels

SIGNATURE					
pre	post	value	pre	post	value
0	24	ROOT	13	8	"Databases"
1	23	bibliography	14	10	author
2	12	publications	15	22	people
3	3	proceedings	16	15	person
4	1	title	17	14	name
5	0	"Databases"	18	13	"Miller"
6	2	editor	19	18	person
7	7	book	20	17	name
8	5	title	21	16	"Smith"
9	4	"Programming"	22	21	person
10	6	author	23	20	name
11	11	book	24	19	"Stewart"
12	9	title			

(b) The signature of the data tree shown in (a)

Fig. 6.5: The tree-view of the example source together with its signature

tree presented in Figure 6.5(a) based on Dietz' labelling scheme. It is formed by the set of pre- and postorder codes for each node together with either its tag name or atomic value. The order of the individual nodes is encoded in the numbering scheme, thus the order of the representation in Figure 6.5(b) is irrelevant. However, the convention of ordering the tuples by their preorder number allows the removal of this attribute from the physical representation and aids the reconstruction of the original document.

The structure presented in Figure 6.5(b) can easily be stored and processed by a relational database. Grust [Gru02] describes how to translate the XPath axis first into the space of the signature and then into SQL queries. For example if one was to select all descendants of the `people` node in Figure 6.5(a), the equivalent query over the signature relation shown in Figure 6.5(b) would be

SELECT v_2.* FROM SIGNATURE v_1, SIGNATURE v_2
WHERE (v_1.value = "people")
AND (v_2.pre > v_1.pre) AND ($v2$.post < v_1.post)

In general the pre- and postorder codes divide the address space of the signature into four quadrants, one for descendants, one for ancestors, one for previous and one for following nodes. This is shown in Figure 6.6 for the `people` node of the example data from Figure 6.5. The bottom-right quadrant of this diagram forms the result to the previous query. This type of diagram was first used by Grust [Gru02].

6.2.3 A Motivation for a Hybrid Design

The numbering schemes provided above and thus the signatures based on them are restricted to trees. Such simple numbering schemes do not extend to general graphs. The approach described here will apply numbering schemes to the distinct spanning tree of a data graph in order to validate individual entries gained from set-based operations.

Section 6.1.1.2 has shown how the index entries of value and tag dictionaries can be used to validate structural queries more efficiently. In fact, the set of all index entries of that example shown in Figure 6.4 forms a signature that is equivalent to the one described by Zhang et al. [ZN+01]. However, rather than storing the complete signature in an array structure or relation, its individual entries can be grouped according to some data clustering as shown in the motivating example of Section 6.1.1.

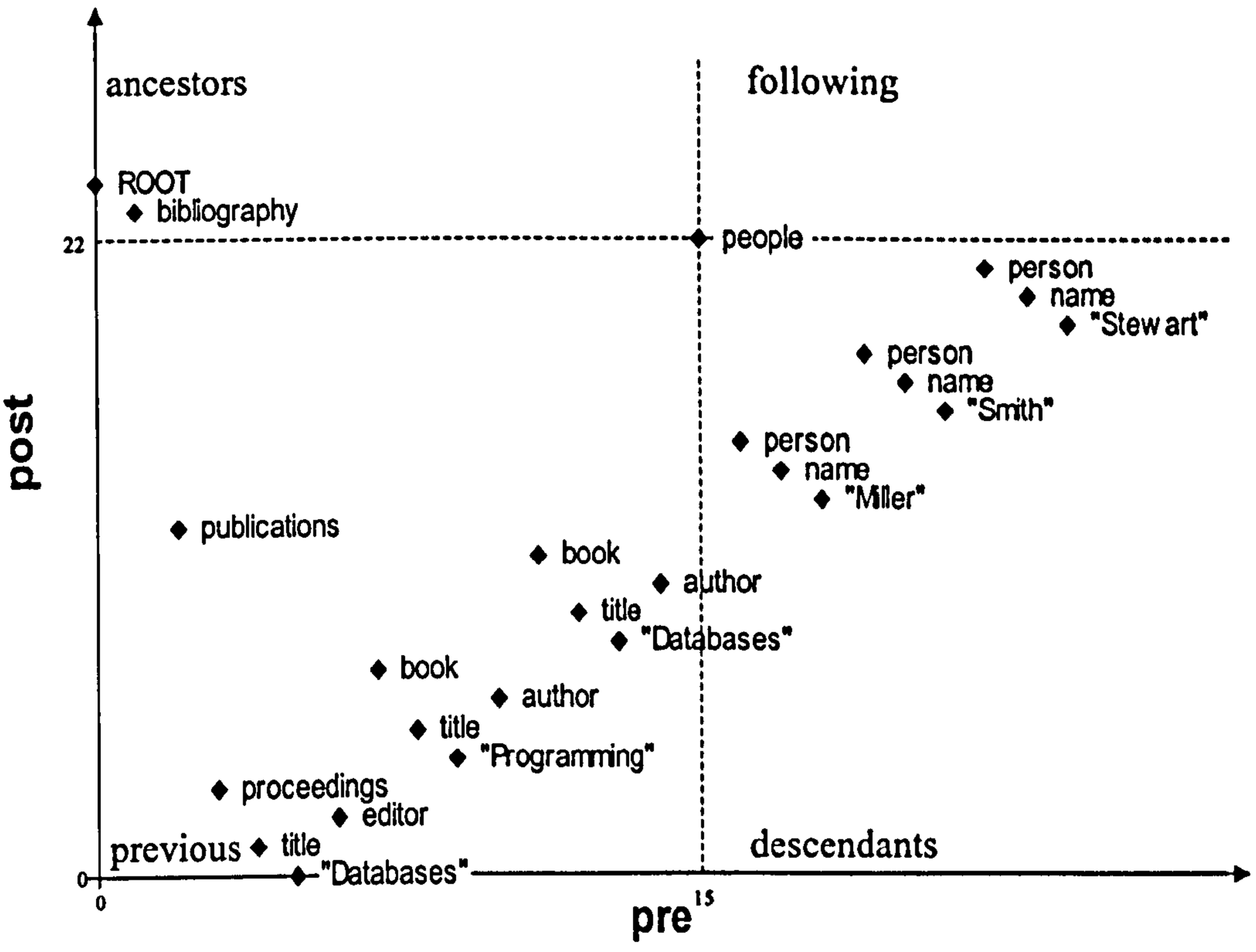


Fig. 6.6: The plane of pre- and postorder codes is divided into four quadrants

The domains implied by the DDOM approach are a special type of a parent domain, i.e. based on local backward-bisimilarity with $k = 1$. Thus a structural index graph based on vertex bisimilarity with $k_b = 1$ can be combined with the indexed dictionaries presented in Figure 6.4. At the same time, the vertex identifiers used in both the dictionaries and the extend of the index graph can be replaced with the entries based on Dietz'numbering scheme, creating a unique address space for validation purposes. The approach suggested here can be seen as the cross-product of an index graph with a signature with its leaf nodes being replaced by domain dictionaries. Figure 6.7 illustrates this using the (1,1)-F+B-index graph of the example data graph shown in Figure 6.1. In this illustration, the incorporated atomic value dictionaries are suppressed in order to simplify the diagram.

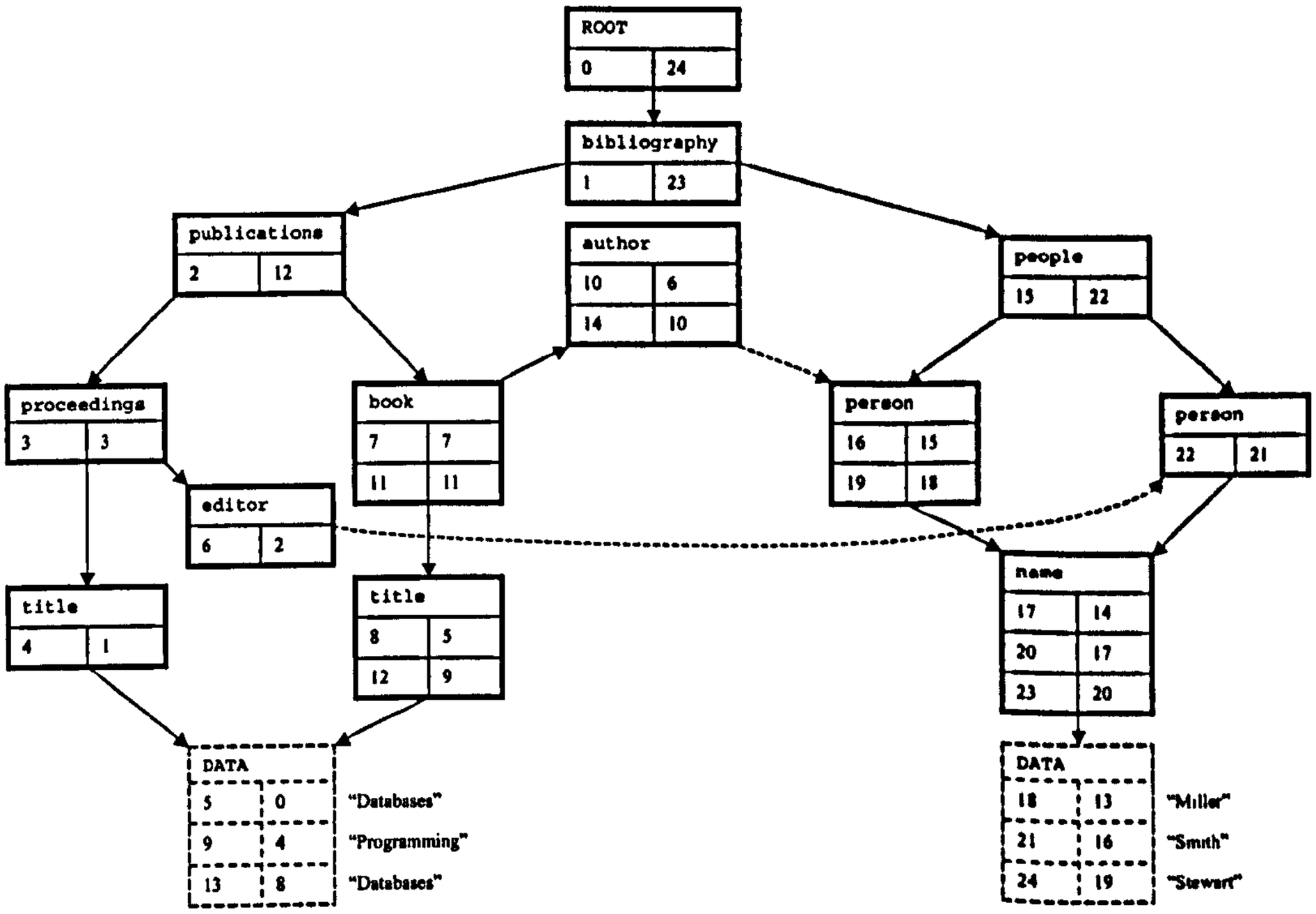


Fig. 6.7: The combination of signature information with a structural index. The atomic value dictionaries at the leaves are not shown for reasons of simplicity.

6.3 Related Work on Combining Structure and Value Querying

The use of numbering schemes and signatures for enhancing SSD processing has been investigated by several researchers. A brief review of the numbering schemes developed, their extensions and query strategies is presented in Section 6.3.1. More recently the shortcomings created by the lack of integration of structural indices on the one hand and atomic value based indexing on the other have been identified. First approaches to the solution of these limitations are described in Section 6.3.2.

6.3.1 Numbering Schemes and Signatures

Several researchers have investigated the benefits of using numbering schemes to answer queries on SSD. Most of these approaches are based on Dietz' numbering scheme using pre- and postorder codes. Extensions to this scheme and similar

schemes exist that are particularly adapted to the XML data model. Their main motivation lies in their extensibility.

6.3.1.1 Preorder and Postorder numbering Schemes

Dietz [Die82] describes a data structure for efficient presentation of linked lists based on trees. An application of this structure is the determination of the ancestor-descendant relationship based on the pre- and postorder numbering schemes described in Example 6.1 and 6.2.

Grust [Gru02] analysed the properties of Dietz's numbering scheme further and identified that the original pre- and postorder numbering scheme can also be used to answer queries along the previous/following axis of XPath. Furthermore he extended the scheme to include direct references to parent nodes and type information (tag name and element or attribute node type), which allows for the storage of the complete XML document in a single relation. All XPath axes operations can easily be performed on the resulting relation with the help of only relational indices. Algorithms exploiting the possibilities of B- and R-trees for XML processing are given.

Zezula et al. [ZA+03] also describes an XML document signature based on Dietz's numbering scheme. However, their querying mechanism is based on string matching over this signature rather than the relational query translation described by Grust. In addition to the ancestor-descendant and preceding-following sibling relationships directly encoded in Dietz' numbering scheme, they describe an extension to accelerate the determination of parent and child nodes using additional references to the first following and first ancestor element in the signature.

6.3.1.2 Range and Extensible Numbering Schemes

Zhang et al. [ZN+01] investigate the performance of relational database and information retrieval technology for the purpose of answering containment queries. In order to compute this class of queries efficiently, they use inverted list entries based on the start and end position of each node in the document tree as described in Section 6.1.1.2. Bruno et al [BKS02] makes use of this numbering scheme to develop stack based join algorithms for tree pattern matching that keep the size of intermediate results to a minimum.

Li and Moon [LM01] recognise the benefits of Dietz' scheme for the evaluation of regular path expressions but identify its weaknesses with respect to

updates. To allow for easier insertions into the tree, they replace the (preorder, postorder)-tuples with (order, size)-tuples, which allow the allocation of extra space for future insertions but maintain the property of constant-time ancestor determination. They decompose an XML source into relations and develop algorithms that perform joins based on this numbering scheme.

Chien et al. [CV+02] present a similar investigation to the one performed by Grust using a durable numbering scheme, but ignore a number of XML-specific features such as the distinction of attributes and elements. Their focus lies on the efficient implementation of ancestor-descendant-joins based on skipping irrelevant partial results.

Kha et al. [KYU02b] describe a numbering scheme based on unique identifiers [LY+96] that provides deterministic addresses for k-ary trees. The fixed fan-out degree assumed in the original work severely limits the possibility of updates and makes poor use of the address space. The extension devises a two level scheme, which allows different k values for different regions of the tree [KYU02a]. Kha et al. also show how this numbering scheme can be utilised for processing structure and keyword queries.

6.3.2 Hybrid Querying Systems

Work on integrating different kinds of index structures for SSD is a recent development. The only early contribution in this field is McHugh and Widom's work [MW99] on query optimisation for the Lore project. However this is mainly restricted to heuristics that determine when to use the four specific forms of indices (value, text, link and path index, cf. [MW+98]), provided by their experimental base. In particular it does not allow for complex structural indices as provided by index graphs based on bisimilarity.

Halverson et al. [HB+03] identify the need to combine pattern matching techniques based on inverted lists with the navigational approach typical for XML tree traversal algorithms and criticise the lack of integration between these two lines of research. They provide a cost model for query answering in each of these domains, identifying query classes that are better suited to either approach or to a combination of both. Based on this model they devise a query optimiser and evaluated its influence on the Niagara database system [ND+01].

Choi and Buneman [CB03] combine the XMill [LS00] approach for compact representation of atomic data with the approach for skeleton compression by

sharing subtrees [BGK03] to address XML join queries. Their fundamental assumption is that the skeleton of typical XML documents is small and thus can be kept in memory. The actual data is only used in the last stage of their join algorithm, avoiding unnecessary I/O operations. However for subqueries with low value predicate selectivity the inflexible approach requiring four sequential scans over the document structure is wasteful.

Kaushik et al. [KK+04a, KK+04b] extend their original work [KS+02, KB+02] on structural indices for path expressions to include keyword constraints on the contained atomic data. They propose a general strategy to combine structural indices with inverted lists in order to address this class of queries efficiently and test their approach using the Niagara system [ND+01]. As their value indices are based on techniques developed in the context of information retrieval systems, their resulting query system includes support for finding the k most relevant results. Such techniques however are beyond the scope of this thesis. The fundamental difference between their work and the work presented here lies in the integration of the different index structures. Within their inverted list they use signature entries based on the same numbering scheme proposed here, extended by an identifying label of the corresponding index node in the structural index. The approach presented here breaks the atomic data dictionaries, which replace their use of inverted lists, according to the structural domains. Consequently the part of the dictionary corresponding to a structural domain can be incorporated into the node of the index graph representing it. By doing this the secondary data structure of the index graph becomes a primary data structure that replaces the original data graph rather than summarises it.

6.4 Experimental System

An experimental system was designed and implemented based on the ideas outlined in Section 6.2.3. The decision about components to be used as structural and value indices and numbering scheme depends largely on the chosen query language and its associated expressive power.

6.4.1 Tree Pattern Expressions

Branching path expressions are used as a basic query language since they represent an important subset of the expressive power of selective query languages for

SSD. However, two constraints are placed upon this language in order to keep the prototype design simple. Firstly, only the edges in the tree view of the data graph are considered, i.e. additional graph arcs are not supported. As a consequence of this restriction to trees, the query language can also be restricted to allow tree patterns only, thus eliminating the need for backward directed axes. Secondly the semantics of query expressions are restricted to return the matches of the root predicate of the query tree rather than the matches of an arbitrary predicate. The resulting language allows the encoding of tree patterns, thus its expressions are called *tree* or *twig pattern expressions* [ZA+03, BKS02]. Query 6.1 shown in Figure 6.2(a) and used in the motivating example is a member of this query class.

Intuitively the result of a tree pattern expression e on a data graph DG is the set of nodes $N_e \subset N(T(DG))$ at which the query tree can be embedded successfully into the data graph, such that the query predicates are embedded into vertices with a corresponding tag label and the structural constraints between query predicates are satisfied by the corresponding paths in the data graph. A minor extension to the atomic value predicates specified in Section 2.3.1.3 was included, which represents the keyword matching functionality used by Kaushik et al. [KK+04a]. This solely allows the choice between exact and substring matching at the leaf nodes and does not affect the complexity of the query language as a whole. Keyword predicates are represented by the syntax `DATA="*`*keyword*`*"`, where *keyword* represent the substring sought.

6.4.2 The Components of the Data Structure

The developed prototype is based on index graphs utilising the concept of local bisimilarity [KB+02] introduced in Example 4.7, a variant of Dietz' numbering scheme additionally annotated by node level information [Die82, BKS02] and a dictionary structure as used in the HIBASE [CMW98] and DDOM (Chapter 5) approach. The reason for this choice and further details about these components are explained below.

The choice of the family of index graphs developed by Kaushik et al. is based on their clearly described and mathematically sound model. The complete F&B-index is the minimal covering index graph for all branching path expressions. For the restricted class introduced in Section 6.4.1, indexing can be restricted to outgoing arcs in the data graph and thus to a tree depth of one (cf. Appendix B.2). In addition only the arcs of the distinguished spanning tree need to be considered.

Thus the $(k, 0)$-F+B-index graph is covering for structural tree patterns with matching path lengths of up to k.

However, the prototype allows for the complete family of indices based on local bisimilarity so that the influence of different data groupings on the query performance can be investigated. In order to apply the approach of dictionary compression successfully, the data needs to be pre-organised in a way that combines values drawn from a homogeneous domain. In the original work on the DDOM model, data was grouped by the label of its parent node. The concept of bisimilarity generalises this approach by grouping vertices by their label first and then refining this organisation based on incoming or outgoing paths. The possibility for parameterising the bisimulation results in different refinements of the dictionary structure to be tested. An important special case will be the $(k, 1)$-F+B-index (Example 4.7), which combines the properties of the F(k)-index for structural constraints with the previously used atomic value dictionaries grouped by parent nodes.

The numbering scheme used to form the entries within the indices is a straightforward extension of Dietz' original scheme. In addition to the pre- and postorder codes of each node, it also stores their level information, i.e. their distance from the root node in the tree-view of the data graph. This is beneficial in order to resolve queries for parent or child nodes or other queries over restricted path length, as shown by the works of Zhang et al. [ZN+01] and Bruno et al. [BKS02]. For the linear path query `//bibliography/*/author` one only needs to validate that the difference of the tree level of all possible ancestor/descendant pairs equals two, without looking at possible intermediate nodes.

6.4.3 Querying System

Different classes of querying algorithm were developed to work on top of data structures like the one shown in Figure 6.7 and described in Section 6.4.2. These will be called NSGraph (for *N*umbering *S*cheme *Graph*) in the following description. As far as the query algorithms are concerned, these behave equally over data graphs and NSGraphs, with the vertices of data graphs only containing a single entry per node whereas NSGraphs can contain any number. The different querying algorithms are described in the following subsections. Fundamentally there exist two groups of query strategies, the merge-join algorithms and graph embedding algorithms. Each of these major strategies can either operate in a

top-down or bottom-up fashion as described in Section 2.3.3. Minor additional variations of the individual algorithms are described below. All algorithms can make use of a general tag index and an atomic value index. The former allows the location of a set of vertices based on a given tag name and the latter locates vertices whose value dictionary contains at least one matching atomic value. In addition, all algorithms may make use of a variant of the merge-join algorithm described by Chien et al. [CV+02] that computes the join of ancestor-descendant tuples with a specified path length between them in time linear in the sum of the size of its argument sets.

6.4.3.1 Pessimistically Validating Tree Embedding Algorithm

The top-down variant (PE-TD) of this algorithm uses the tag index to find all vertices of the NSGraph whose tag name matches the constraint of the query root predicate. Starting from these candidates all adjacent vertices of the NSGraph are matched against the set of adjacent predicates of the query graph. This is performed recursively in the case of the descendant-axis. On reaching a leaf in the query pattern tree, the remaining valid embeddings of this leaf are passed back to its parent node and validated at each predicate as correct embedding points for it using the merge-join algorithm. Twig predicates in the query pattern combine the results from their children before passing their own embedding upwards. On reaching the root predicate, all embeddings for this tree pattern are computed and can be output.

Conversely the bottom-up algorithm (PE-BU) starts by identifying matches of the leaf predicates and then embeds the structure represented by the query graph following its arcs in reverse direction. Entries from the corresponding NSGraph vertices are only added if they are ancestors of the already discovered possible result set. In both cases the embedding stops when either the query graph is completely embedded or an intermediate result set is empty.

6.4.3.2 Optimistically Validating Tree Embedding Algorithm

The algorithm described above works even if the query length exceeds the forward bisimilarity of the used NSGraph, because all entries are validated for every edge of the tree pattern. However, if the underlying bisimulation is covering this is wasteful and can be avoided. The optimised, lazy variants of the above algorithms only validate the entries of vertices in the NSGraph if the matched

path exceeds their bisimilarity length, at branching points of the query pattern and at vertices that have multiple incoming edges, i.e. at merging points of the NSGraph. Otherwise these algorithms behave identically to the one described above, i.e. they start at the root or leaf predicates respectively and iteratively embed predicates connected to them in either a top-down (OE-TD) or bottom-up (OE-BU) fashion.

6.4.3.3 Ancestor/Descendant Merge-Join Algorithm

This last group of algorithms does not make use of the structural information of the NSGraph directly, but simply uses a merge-join approach on candidate sets solely selected using the flat tag label and atomic data indices. The top-down variant (MJ-TD) uses the root predicate to select the first group of potential ancestor nodes and joins it with the group of entries complying with the tag constraint given by the first child predicate of the root. In the case of a linear query the result of this operation is used as set of ancestors for the next join, this time with nodes complying with the tag constraint of the root's grandchild. On reaching a twig predicate, the evaluation is performed following one outgoing path at a time, each time further restricting the initial candidate set used for matching the remaining paths. As in the top-down embedding strategies, the remaining leaf results are passed back up and collated at twig nodes before being passed on upwards. Again this strategy can be employed in a top-down (MJ-TD) or bottom-up (MJ-BU) fashion. The latter variant uses either the atomic data or tag label index to determine the initial set of candidates for each leaf predicate, depending on its type. In any case these algorithms perform one join per edge in the query graph, unless one intermediate result is empty in which case the query can terminate early. In practice this means very few joins, however the sets of potential ancestors and descendants taking part in the join are usually quite large, making the individual joins computationally more expensive.

6.5 Query Execution Performance Analysis

The foci of this analysis are the effects that variations in the data grouping have on the performance and complexity of the three querying algorithms and two strategies. External comparisons of the overall performance are of restricted validity because of the limitations and simplification of the prototype discussed

Query	Tree Pattern
K1	//item[/description//keyword/DATA="*attires*"]
K2	//open-auction[/bidder/date/DATA="*1999*"]
K3	//person[/profile/education/DATA="*Graduate*"]
K4	//closed-auction[/annotation//happiness/DATA="*10*"]
Q1a	//person[/name/DATA="Klemens Pelz" & /watches & /emailaddress & /creditcard]
Q1b	//person[/name/DATA="Klemens Pelz" & /watches & /emailaddress & /phone]
Q2a	//profile[/income & /education & /gender]
Q2b	//profile[/income & /education & /gender/DATA="male"]
Q2c	//profile[/income & /education & /gender/DATA="female"]
Q3a	//item[/location/DATA="United States" & /payment/DATA="*Creditcard*" & /quantity/DATA="1"]
Q3b	//item[/payment[/DATA="*Creditcard*" & /DATA="*Cash*"]]
Q4a	//description[//text[//keyword & //bold]]
Q4b	//description[//text/*[/keyword & /bold]]

Tab. 6.1: The benchmark tree pattern queries in BPE syntax

in Section 6.4. However the following results will identify whether or not the choice of a particular data grouping has a significant impact on particular query algorithms.

6.5.1 The Benchmark Queries and Data Source

The tree pattern queries collected for this experiment are designed to highlight the response of the implemented query algorithms to a number of distinct challenges within the designated query class. These are characterised in the following sections. Table 6.1 presents all queries in BPE syntax.

6.5.1.1 Linear Tree Patterns with Value Predicates

The first set of queries contains only linear path expressions, i.e. no predicate of the query tree pattern has more than one child. Such queries can be resolved using simpler data structures, e.g. path indices such as the DataGuide [GW97]. The performance of the hybrid system will be tested against this class of queries because it represents an important subclass of the general query class. The queries used are taken from Kaushik et al. [KK+04a] and are aimed at the 1 MB XMark dataset [SW+02] described in Appendix C.3 that models the data for an online auctioning system. Query K1 is a modification of the version of the original

work that returns all `item` entries containing the `keyword` "attires" rather than atomic "attires" vertices which occur below an `item` in order to comply with the restrictions of tree pattern expressions. Queries K2 – K4 are exactly as described in the original research.

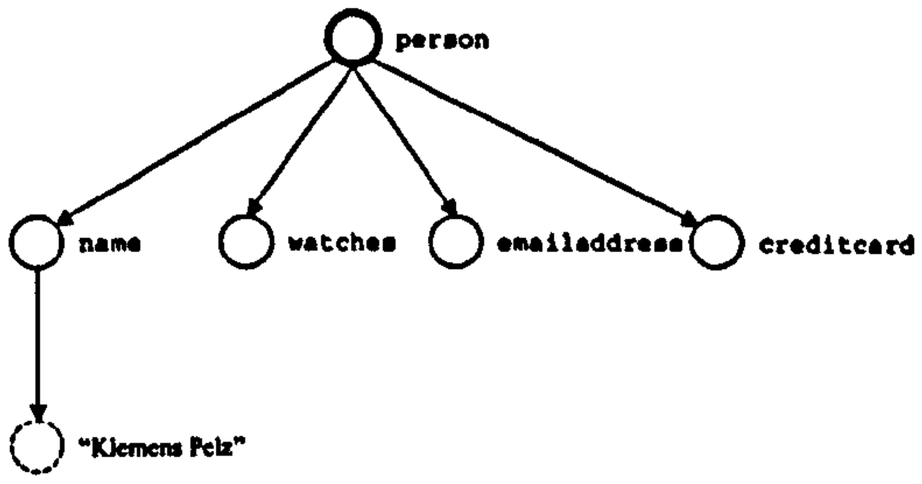
6.5.1.2 Branching Tree Patterns with Value Predicates

The next group of queries represents proper branching tree expressions, i.e. queries for which at least one structural predicate is connected to more than one child predicate. The queries are purpose-designed to evaluate different aspects of query processing and are resolved against the same XMark dataset used for the linear queries described above. Figure 6.8 shows the graph representations of these tree patterns.

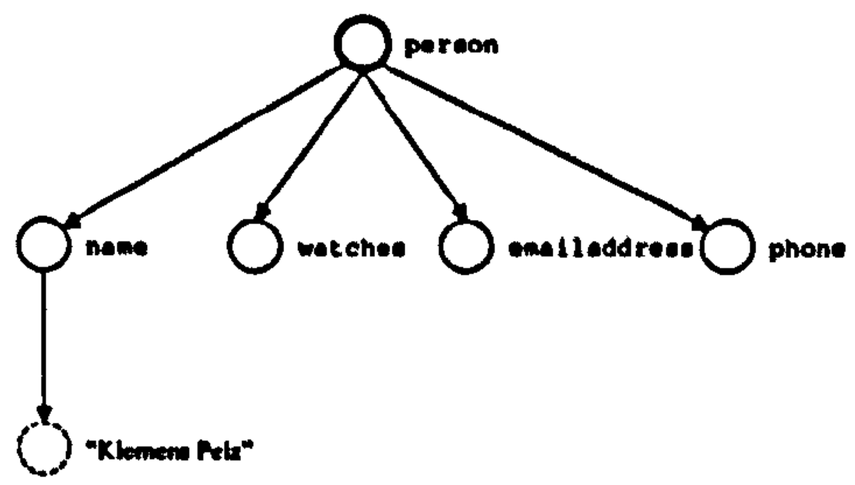
Point Queries Queries Q1a and Q1b are typical examples of the targeted query class, i.e. they combine both structural and atomic value predicates. In particular they represent *point queries*, i.e. they essentially ask whether the given pattern exists in the data or not. This is achieved by including an atomic value predicate on the node representing the `name` attribute attached to each `person` node of the source. Both queries search for a `person` named "Klemens Pelz". Only one such `person` exists in the database. However, the specific entry in the database only conforms with the structure of Query Q1a, but not with the structure of Query Q1b.

Queries with Varying Cardinality The Queries Q2a – Q2c all locate parts of the data graph with the same structure, a `profile` that contains at least `income`, `education` and `gender` information about the person it is describing. Query Q2a represents solely this structural constraint, whereas Query Q2b and Q2c restrict the results to the male and female subsets respectively by means of an atomic value predicate. In the XMark dataset, there are about twice as many male entries as there are female. Thus these patterns give an insight into the effect that the cardinality of value predicates has on the query evaluation.

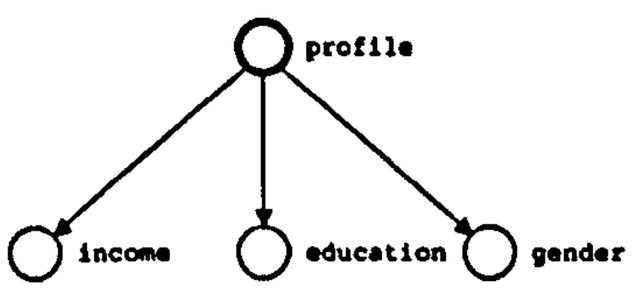
Conjunctive Value Queries Queries Q3a and Q3b both contain more than one atomic value predicate that needs to be true in combination with the structural predicates. Query Q3a looks for an `item` whose child nodes match three different, single valued atomic value predicates for its `location`, `payment` type and



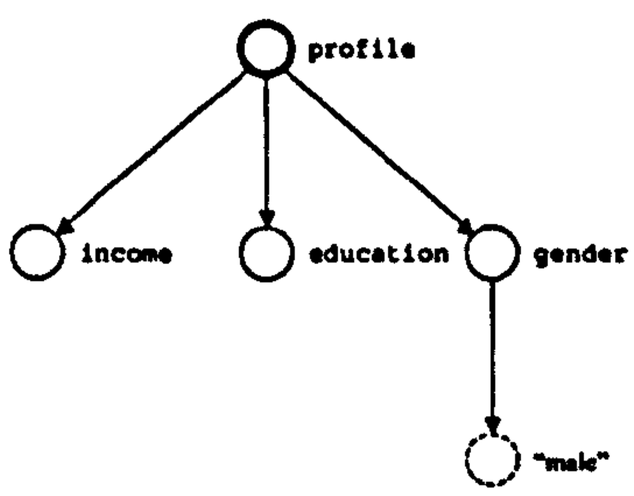
(a) Query Q1a



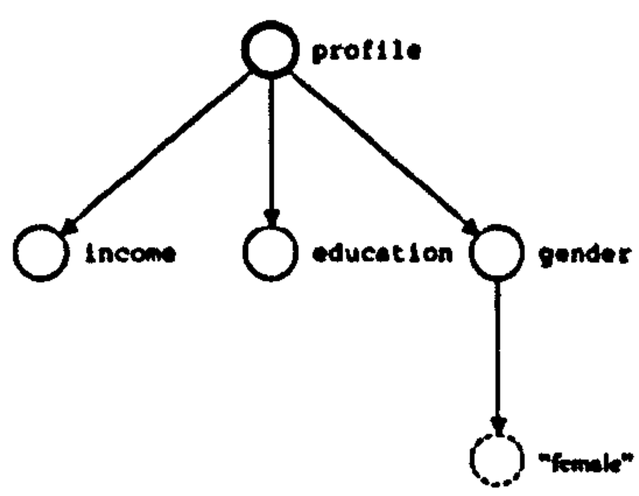
(b) Query Q1b



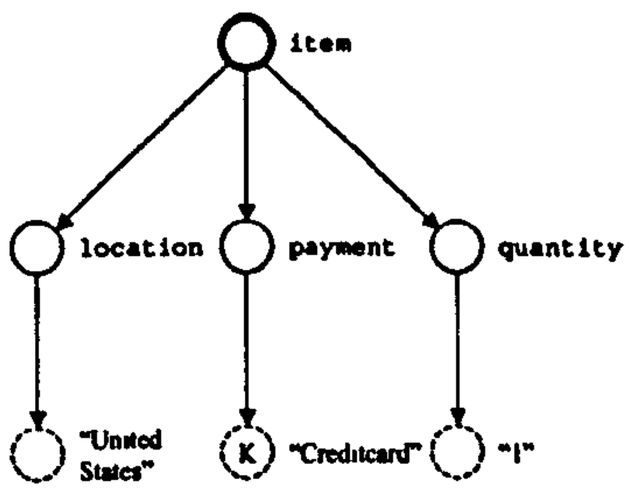
(c) Query Q2a



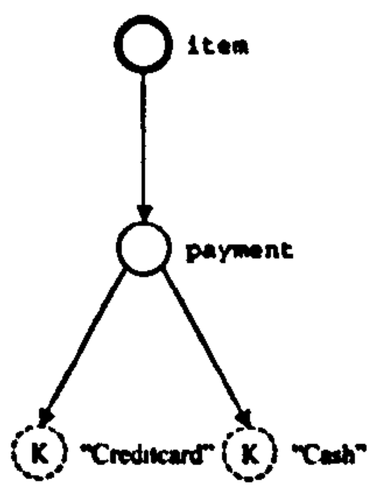
(d) Query Q2b



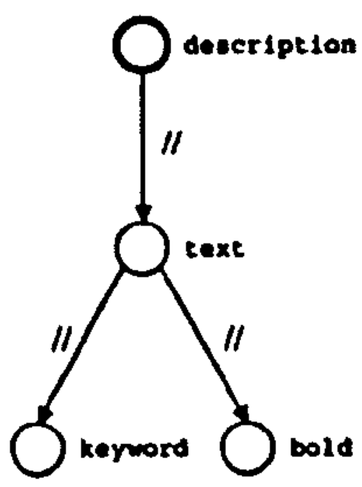
(e) Query Q2c



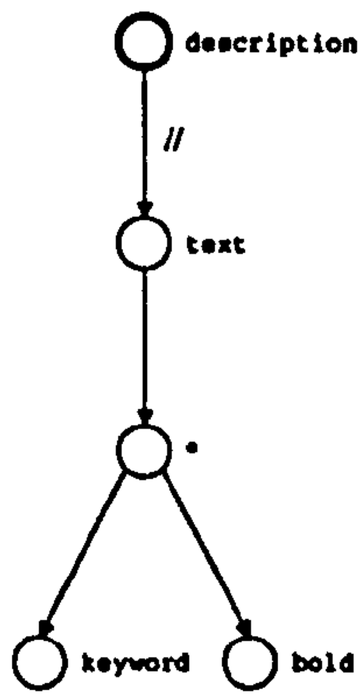
(f) Query Q3a



(g) Query Q3b



(h) Query Q4a



(i) Query Q4b

Fig. 6.8: The graph representations of the branching tree patterns

quantity. The item of Query Q3b has only one child predicate, but this predicate needs to comply with two atomic value predicates at the same time, here implemented by looking for the two distinct keywords "Creditcard" and "Cash" below a common payment node. The semantics of keyword queries represent a substring matching on the atomic value. Thus the two constraints can actually be matched by a single atomic value node, e.g. an atomic node with value "Cash, Creditcard".

Queries Containing Regular Expressions over Nested Parts of the Source The textual descriptions of items and categories of the XMark dataset can contain highly nested mark-up language and mixed content elements. Queries Q4a and Q4b query this part of the database for structural constraints with variable path lengths and tag label wildcards. Query Q4a simply asks for a descriptions node containing some text that contains at least one keyword and one bold descendant node, whereas Query Q4b requires the nodes matching the same leaf predicates to be the children of a common node with an unknown tag label.

6.5.1.3 Result Verification

The results computed by all of the query algorithms presented in Section 6.4.3 were verified using the eXist[4] open source native XML database. It provides a partial XQuery implementation that can resolve all the queries shown in Table 6.1. The results returned by the experimental system are identical to those returned by eXist.

6.5.2 Query Execution Performance of Data and NSGraphs

The first set of experiments keeps the used bisimilarity structure constant allowing the comparison of its influence on the different query strategies and contrasting it with the behaviour found on data graphs. The bisimulation is designed to be covering for the structural part of the most complex query of a query set not containing descendant operators, i.e. its forward bisimilarity k_f is set to the depth of the query tree pattern. Its backward bisimilarity k_b is set to one, in order to organise the atomic data by its parent's tag label. This represents the most common atomic data organisation, which was already used in Chapter 5.

[4] http://www.exist-db.org

Consequently the NSGraph used for the linear tree patterns described in Section 6.5.1.1 is based on (3,1)-bisimilarity and the NSGraph used for the branching query patterns of Section 6.5.1.2 is based on (2,1)-bisimilarity. The former reduces the size of the vertex set from 32,864 for the data graph to 3,170 vertices in case of the NSGraph. In the latter case the resulting NSGraph contains only 2,068 vertices, equivalent to approximately 6.3% of the original vertex-set.

6.5.2.1 Performance Metrics Used

Tables 6.2 – 6.4 show the measurements obtained by executing all the described query strategies for the set of benchmark tree patterns on the data and NSGraphs. Each table lists four different metrics for each run, the numbers of vertices of the graph visited, the number of joins performed using an ancestor-descendant join-merge algorithm based on the numbering scheme, the total number of all entry references taking part in such joins and the average query time. They are designed to highlight different aspects of the overall performance of a particular algorithm.

The number of vertices visited is an indication of the expected I/O costs. Although the prototype system works entirely in memory, a practical implementation working on larger datasets would need to load data from external storage. Assuming one I/O operation per vertex accessed is a reasonable base often used for performance modeling in database systems. It reflects the fact that external storage is usually block-based and thus the costs of loading a large but localised structure like a vertex of a NSGraph containing many entry references is usually cheaper than loading many small structures that might be spread across many different blocks.

The number of joins indicates how often each of the algorithms validates that the candidate set it is working on belongs to the proper intermediate result set. This becomes necessary since, although the NSGraph is designed to be covering for the structural part of a query, it is no longer covering for tree patterns containing data predicates. This metric indicates how appropriate the given strategy is for a particular query.

In addition the total number of entries joined in such a way is measured because the join algorithm used has a complexity that is linear in the size of its arguments. Consequently this gives an indication of the computational costs of a query. Particularly an algorithm employing three joins over very restricted candidate sets will be cheaper than an algorithm just using one join over a substantial

part of the total number of entries.

Lastly, the query time gives an estimate how these costs influence the overall query execution. Because an in-memory prototype is used, the computation costs here dominate over the I/O costs.

6.5.2.2 Results for the Linear Tree Patterns

As one can see from Table 6.2 there exists a strategy for queries K2 – K4 on the summarised data, which outperforms the best strategy on the data graph in terms of the overall performance. This does not hold true for Query K1 that contains a descendant operator, because the NSGraph is not covering for even the structural part of this query. For this query the algorithms performed on the potentially cyclic NSGraph need to check more potential embedding paths than in the restricted data tree.

More important though is the observation that for almost all algorithms and queries the NSGraph requires fewer vertex visits than the data graph. The only exception to this pattern is the application of the bottom-up embeddings over Query K1. This is due to two reasons. Firstly, the atomic value predicate of this query has a very low selectivity, returning only five hits on its own, of which only one appears in the right context. Thus only a very limited fraction of the data graph is actually searched and the NSGraph cannot offer a substantial saving on this account. The second reason has already been discussed above and results from the use of the descendant operator in this query.

In terms of the query algorithms' complexity and their computational costs the results are the inverse of what was said in the previous paragraph. Here clearly the data graph shows its strength due to its simpler structure. The number of joins is zero for all embedding based algorithms, as one never needs to validate that the vertices at which the individual predicates are embedded belong to the same subtree of the data graph. This is enforced by the embedding mechanism. For the merge-join algorithms, which use the tag label and value indices to find candidates rather than traversing the arcs of the data graph, a constant number of joins needs to be performed. These algorithm perform one join per edge in the query pattern, regardless whether the data graph or NSGraph is used.

In terms of overall performance top-down and bottom-up perform about equally well on the data graph, whereas the top-down embeddings have a small advantage on the NSGraphs. This appears to be surprising, given the fact that

Source	Data graph				(3,1)-NSGraph			
Query	K1	K2	K3	K4	K1	K2	K3	K4
Results	1	75	19	11	1	75	19	11
Number of vertices visited								
PE-BU	16	784	57	365	296	34	25	39
OE-BU	16	784	57	365	3223	48	27	41
PE-TD	6680	5930	3739	1261	4712	208	1620	74
OE-TD	6680	5930	3739	1261	4712	208	1620	74
MJ-BU	1559	2645	1728	852	295	63	305	75
MJ-TD	1559	2645	1728	852	295	63	305	75
Number of joins performed								
PE-BU	0	0	0	0	100	53	85	54
OE-BU	0	0	0	0	452	16	88	55
PE-TD	0	0	0	0	113	17	85	22
OE-TD	0	0	0	0	113	25	104	24
MJ-BU	3	3	3	3	3	3	3	3
MJ-TD	3	3	3	3	3	3	3	3
Number of entries joined								
PE-BU	0	0	0	0	1566	13709	575	1097
OE-BU	0	0	0	0	6530	3204	1431	1651
PE-TD	0	0	0	0	2433	3507	575	876
OE-TD	0	0	0	0	2433	3803	618	984
MJ-BU	1561	3110	1766	896	1561	3110	1766	896
MJ-TD	1561	3110	1766	896	1561	3110	1766	896
Average processing time in *ms*								
PE-BU	17.0	25.1	13.0	16.0	21.0	18.0	13.0	16.1
OE-BU	15.0	22.0	13.0	16.0	32.1	16.0	14.0	16.0
PE-TD	37.1	28.1	20.1	5.0	26.0	8.1	8.0	2.0
OE-TD	38.0	32.0	18.0	6.1	26.1	11.0	10.1	2.0
MJ-BU	23.0	27.1	23.1	18.0	21.1	26.0	21.0	17.0
MJ-TD	23.0	31.0	25.0	18.0	22.0	30.1	24.0	18.0

Tab. 6.2: Execution performance of the linear tree pattern queries on the data and NSGraph

the bottom-up algorithms visit fewer vertices and both types of algorithm exhibit the same complexity on linear tree patterns. It will become obvious, once the results of the branching patterns have been discussed that this is actually an artifact caused by the regular expression matching being performed against the atomic data nodes. All linear path queries of the benchmark suite use *keyword* predicates, i.e. they locate substring matches rather than exact hits. This is a consequence of using the queries presented by Kaushik et al. [KK+04a], which were based on a hybrid system using inverted lists rather than dictionaries. The cost for this substring matching dominates the overall cost of the atomic data matching procedure, but is not accounted for by the individual performance metrics.

Due to the restriction to non-branching query patterns, the performance metrics of top-down and bottom-up strategies of the same algorithms differ little or not at all in the case of merge-join strategies. This is mainly due to the fact that the same path must be followed during the matching process and that it does not matter in which direction this occurs. Things become more interesting for the general branching tree patterns, the performance metrics for which are shown in Table 6.3 and 6.4

6.5.2.3 Results for the Branching Tree Patterns

As in the linear case the most important observation is that there exists a strategy for every query that visits considerably fewer vertices of the hybrid NSGraph than the best strategy over the data graph. Only taking the best strategy for each query and data structure into account, this difference lies between a factor of just under two and up to 30. In fact every strategy visits fewer vertices on every query over the NSGraph with the exception of the bottom-up embedding algorithms for Query Q4a. This query contains three descendant operators, each of which result in a lengthy traversal of the hybrid structure.

There exists a query strategy for the hybrid data structure that outperform the best strategy on the data graph for all queries except the queries Q1a – Q1b and Q2b – Q2c. These queries have a very low selectivity based on their atomic value predicates and can thus be evaluated quickly on the data graph using the atomic value index used by the bottom-up embedding algorithms. In general the queries containing regular expressions over the nested description part of the source exhibit the slowest performance, both on the data graph and the hybrid

Query	Q1a	Q1b	Q2a	Q2b	Q2c	Q3a	Q3b	Q4a	Q4b
Results	1	0	40	27	13	78	55	159	6

Number of vertices visited

PE-BU	513	500	286	305	267	1458	273	16816	1748
OE-BU	513	500	286	305	267	1458	273	16816	1748
PE-TD	3031	3027	2433	2473	2473	5753	3091	17529	13525
OE-TD	3031	3027	2433	2473	2473	5753	3091	17529	13525
MJ-BU	6970	6957	700	745	726	2961	1520	4301	70029
MJ-TD	2488	2475	424	469	450	2093	869	2832	35696

Number of joins performed

PE-BU	0	0	0	0	0	0	0	0	0
OE-BU	0	0	0	0	0	0	0	0	0
PE-TD	0	0	0	0	0	0	0	0	0
OE-TD	0	0	0	0	0	0	0	0	0
MJ-BU	5	5	3	4	4	6	4	4	6
MJ-TD	5	4	3	4	4	6	3	3	4

Number of entries joined

PE-BU	0	0	0	0	0	0	0	0	0
OE-BU	0	0	0	0	0	0	0	0	0
PE-TD	0	0	0	0	0	0	0	0	0
OE-TD	0	0	0	0	0	0	0	0	0
MJ-BU	6971	6958	700	790	752	3628	1677	4987	70800
MJ-TD	2492	2359	639	729	691	2999	1040	3518	36382

Average processing time in *ms*

PE-BU	12.0	4.0	4.0	2.0	2.0	26.1	28.0	45.0	14.0
OE-BU	7.1	4.0	3.0	2.0	1.0	22.0	27.0	44.1	15.0
PE-TD	22.0	14.1	7.0	7.1	6.0	18.0	10.1	92.1	60.1
OE-TD	16.0	13.0	9.0	9.0	8.0	23.1	12.0	84.2	63.1
MJ-BU	44.1	48.1	3.0	3.0	3.0	30.0	33.0	25.1	727.0
MJ-TD	15.0	13.0	3.0	3.0	3.0	27.1	31.1	69.1	385.6

Tab. 6.3: Execution performance of branching tree patterns on the data graph

Query	Q1a	Q1b	Q2a	Q2b	Q2c	Q3a	Q3b	Q4a	Q4b
Results	1	0	40	27	13	78	55	159	6
Number of vertices visited									
PE-BU	341	343	30	31	30	58	9	24356	506
OE-BU	530	532	30	31	31	63	9	24356	506
PE-TD	1247	1243	1553	1557	1557	1222	530	13003	12056
OE-TD	1247	1243	1553	1557	1557	1222	530	13003	12056
MJ-BU	1226	884	399	400	400	218	138	556	4692
MJ-TD	704	706	153	154	154	118	70	352	2420
Number of joins performed									
PE-BU	530	532	256	264	262	191	58	1386	349
OE-BU	509	511	256	256	256	139	58	1386	343
PE-TD	174	172	228	232	232	143	56	1235	140
OE-TD	178	174	456	449	437	242	94	1235	140
MJ-BU	5	4	3	4	4	6	4	4	6
MJ-TD	5	4	3	4	4	6	3	3	4
Number of entries joined									
PE-BU	1705	1679	3679	3821	3465	18940	3739	28815	7281
OE-BU	1447	1421	3679	5779	4563	32393	3739	28815	7269
PE-TD	496	456	3316	3393	3155	10724	3532	48227	6132
OE-TD	540	478	3826	3879	3613	11594	4287	48227	6132
MJ-BU	6971	5209	700	790	752	3628	1691	4987	70800
MJ-TD	2492	2359	639	729	691	2999	1040	3518	36382
Average processing time in *ms*									
PE-BU	11.1	8.0	2.0	3.0	3.0	20.0	27.1	78.1	10.0
OE-BU	8.0	9.0	3.0	3.0	4.0	17.1	26.0	80.2	10.0
PE-TD	10.0	9.0	7.0	7.0	6.1	10.0	5.0	111.1	63.1
OE-TD	10.0	9.0	13.1	13.0	13.0	28.0	10.0	111.2	63.1
MJ-BU	40.1	28.0	3.0	4.0	4.0	27.1	31.1	21.1	515.8
MJ-TD	13.0	12.0	3.0	3.0	2.0	25.0	29.0	17.0	292.4

Tab. 6.4: Execution performance of branching tree patterns on the (2,1)-NSGraph

structure. The later contains cycles for this part of the source, which complicates the evaluation of regular expressions and requires all embedding algorithms to make use of arc- or vertex markers to avoid endless loops.

The embedding algorithms require to verify the entries representing vertex instances of the data graph at every branching point of the query and when the matched path exceeds the bisimilarity length of the NSGraph, whereas the merge-join algorithms have a fixed complexity of one join per edge in the query tree. However due to the fact that the candidate selection is solely based on tag labels, but not structure, the cardinality of the argument sets can be larger than those for the embeddings algorithms on the hybrid structure. This is the case for the Queries Q1a and b, which have a very low selectivity, and for Query 4b, whose tag label wildcard forces the merge-join algorithm to consider all vertices of the graph.

The data graph clearly favours the optimistically validating bottom-up (OE-BU) algorithm, which produces the best or near best performance for all but the very complicated Queries Q3a – Q4a. For the multi-valued predicate Queries Q3, the individual predicates do not restrict the nodes being visited sufficiently for the bottom-up algorithms as this is only achieved once the two predicate branches join. They thus favour the pessimistically validating top-down embedding (PE-TD) algorithm, which enables the predicate matching procedure on one branch to restrict the candidate set before it is used for matching the next branch. Query 4a contains three descendant operators and is thus best resolved using a merge-join algorithm that does not incur extra cost for this operator. Query 4b forbids this strategy due to the use of the tag label wildcard and favours a bottom-up approach that quickly narrows down the candidate set, before the descendant operator is reached. On the NSGraph the situation is similar. However here the optimistically validating top-down algorithms that does not filter out candidates entries before all structural constraints have been met, is outperformed on queries Q3a and Q3b by the pessimistic top-down embedding algorithm that continually reduces the set of candidate entries through validation.

6.5.3 Varying the Coarseness of the NSGraph Structure

In order to analyse the influence a particular data grouping has on the execution performance of the different query strategies, the parameters used to define the underlying bisimulation of the hybrid representation are varied. Both forward

and backward bisimilarity length are varied independently from zero to three. This defines a four-dimensional, discrete parameter space for every single metric measured, given by the forward and backward bisimilarity, the query algorithm used and the query executed. The measurements obtained in this experiment are too numerous to be presented here entirely but are listed in Appendix D for completeness. Here only a few interesting aspects can be highlighted, which were established using pivot tables and diagrams over the collected data.

6.5.3.1 Results for the Linear Tree Patterns

Figure 6.9 shows the minimal number of vertex visits taken to answer an individual linear query by any of the algorithms over the complete set of query strategies. This is presented as values over the bisimilarity parameter k_b and k_f, determining the backward and forward bisimilarity respectively. The number of visits for all queries are stacked up to give an estimate of the overall usefulness of a particular NSGraph.

Two observations can be made using this data. Firstly it becomes obvious that the number of vertices visited increases with the complexity of the bisimulation and the NSGraph based upon it. This trend is visible both for increasing backward and forward bisimilarity. The increase is monotonic for increasing forward bisimilarity. For increasing backward bisimilarity the results are less clear cut. There is a significant difference, about one order of magnitude for most queries, between those structures based on zero backward bisimilarity and NSGraphs based on bisimilarity greater than zero. However, the increase is not monotonic for the cases $k_f = 2$ and $k_f = 3$. In the first case there is a reduction in the number of required vertex visits for $k_b = 2$ and in the second case for $k_b = 3$. Closer inspection of the raw data shows that the discontinuity between $k_b = 0$ and $k_b = 1$ arises from the different set of query strategies used for optimal computation. Whereas for the case with no backward bisimilarity a combination of merge-join algorithms (for Query K1) and bottom-up embedding algorithms (for all other queries) requires least visits, NSGraphs based on backward bisimilarity with $k_b \geq 1$ are better addressed using a combination of top-down (for Query K4) and bottom-up (for all other Queries) embedding algorithms.

The second observation is that Query K1 dominates the processing costs for this set of queries. This is at least partly expected as its descendant operator means that no graph based on local bisimilarity is covering even for the structural

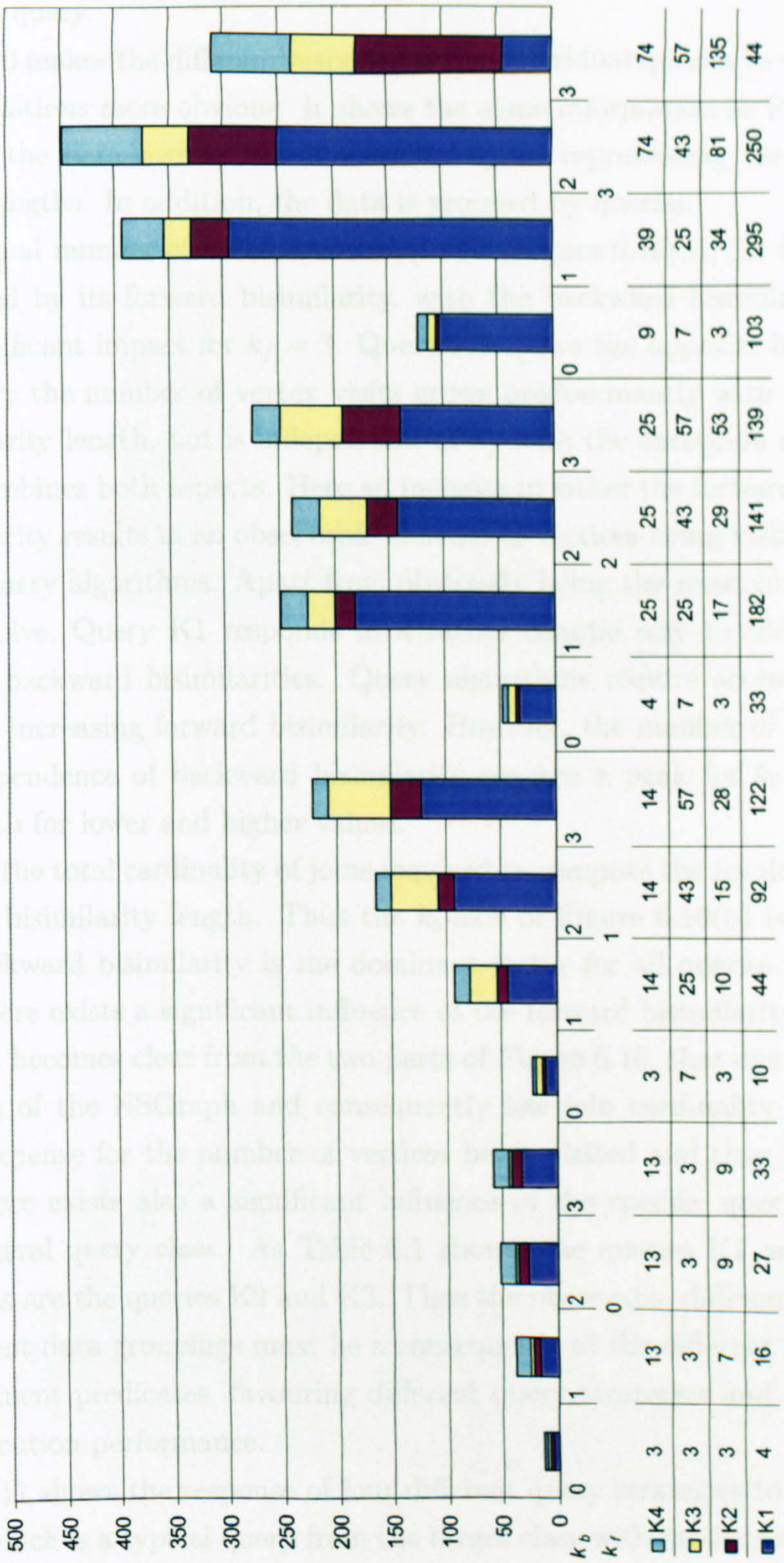


Fig. 6.9: The number of vertex visits used by the best query strategy for each of the linear query patterns over the bisimilarity of the NSGraph

aspect of this query.

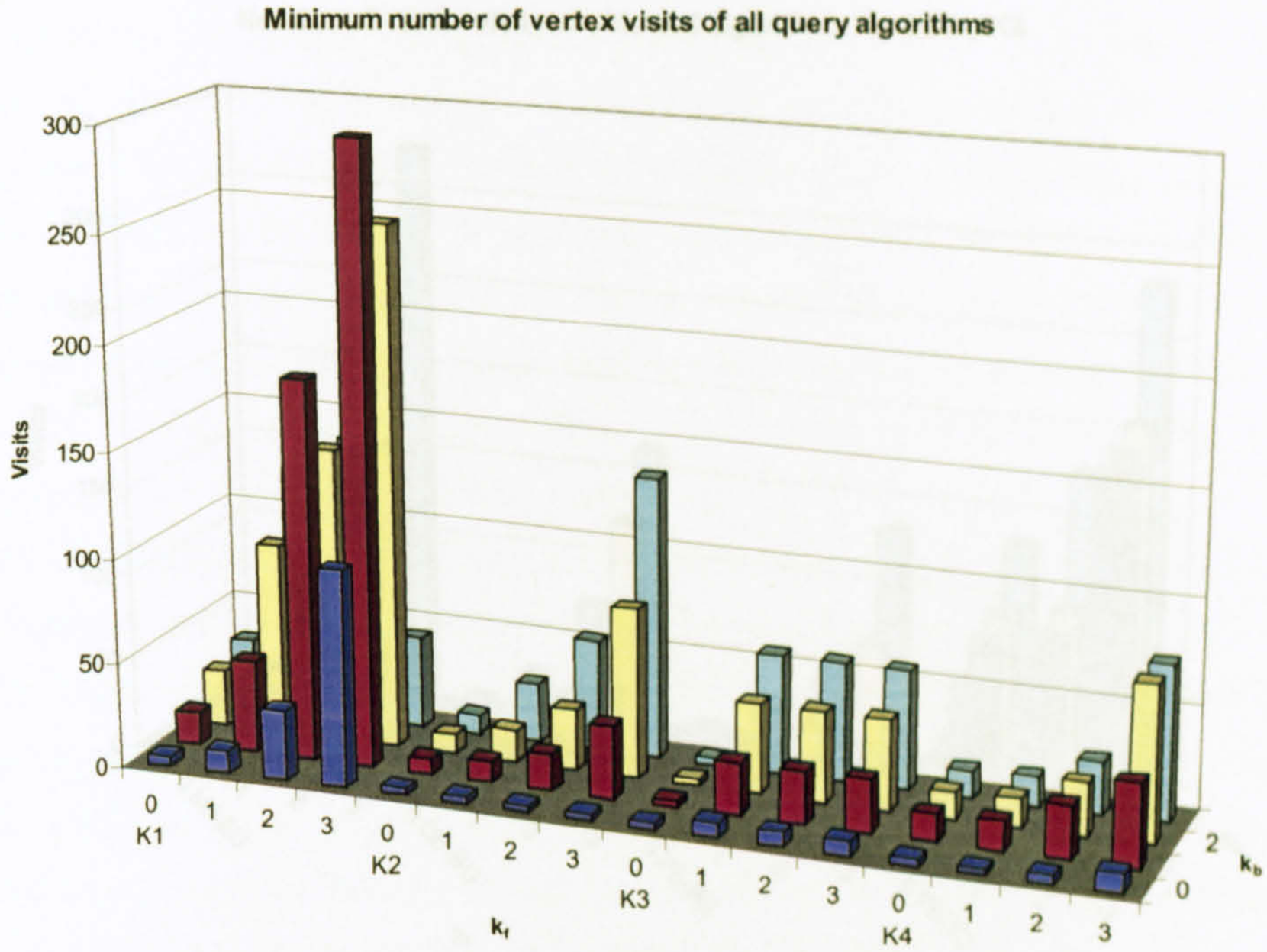
Figure 6.10 makes the different response of the individual queries to variations in the bisimulations more obvious. It shows the same information as Figure 6.9, but presents the data over a two-dimensional space representing the different bisimilarity lengths. In addition, the data is grouped by queries.

The minimal number of vertex visits required (Figure 6.10(a)) for Query K4 are dominated by its forward bisimilarity, with the backward bisimilarity only having a significant impact for $k_f = 3$. Query K3 shows the opposite behaviour. For this query the number of vertex visits grows predominantly with the backward bisimilarity length, but is independent of k_f with the exception of $k_f = 0$. Query K2 combines both aspects. Here an increase in either the forward or backward bisimilarity results in an observable increase of vertices being visited by the best suited query algorithms. Apart from obviously being the most complicated query to resolve, Query K1 responds in a rather chaotic way to the different forward and backward bisimilarities. Query algorithms require access to more vertices with increasing forward bisimilarity. However, the number of nodes accessed in dependence of backward bisimilarity reaches a peak for $k_b = 1$ and decreases both for lower and higher values.

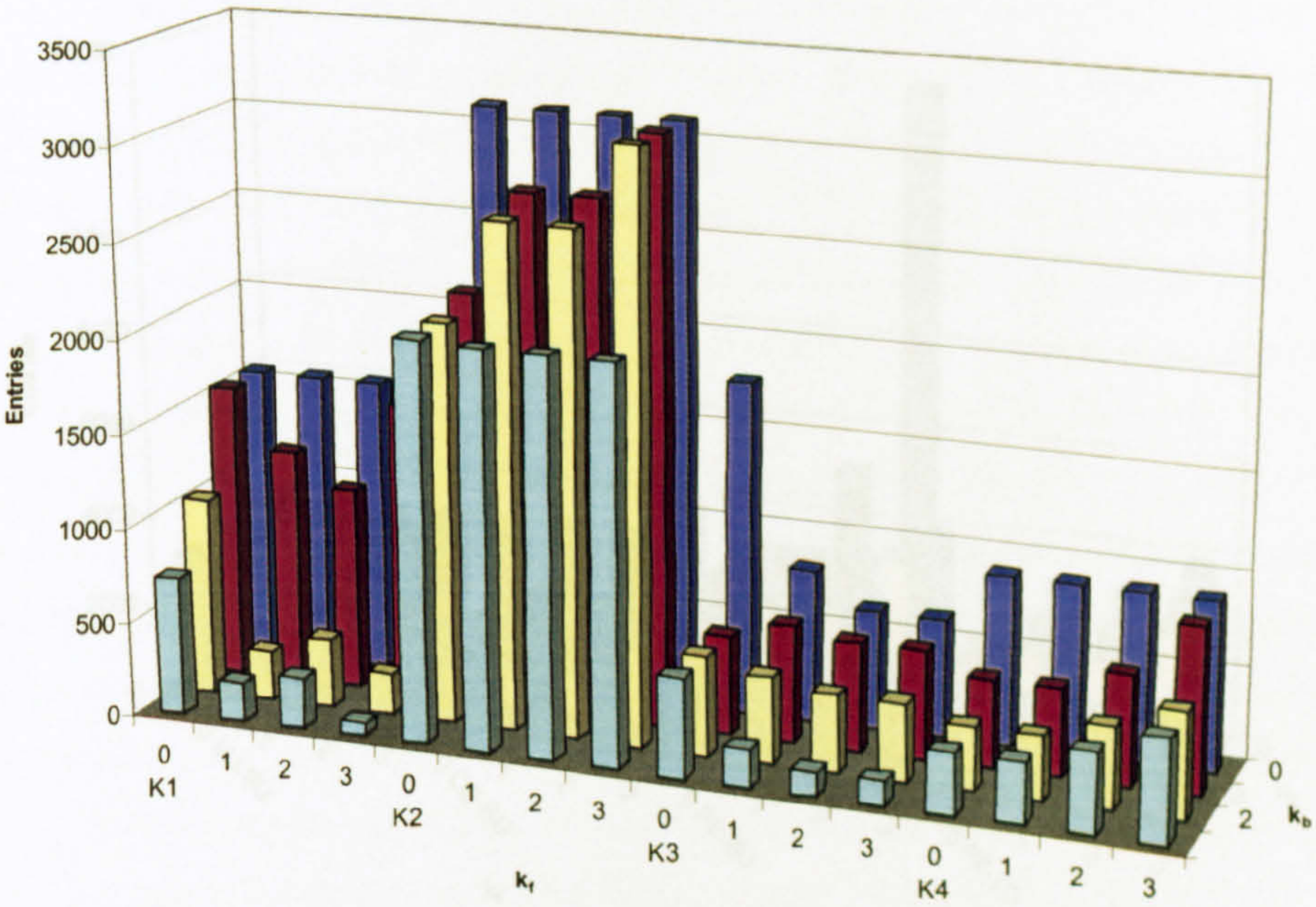
Inversely the total cardinality of joins required to compute the results increase with shorter bisimilarity length. Thus the k_b-axis of Figure 6.10(b) is reversed. Here the backward bisimilarity is the dominant factor for all queries. Only for Query K1 there exists a significant influence of the forward bisimilarity.

Overall it becomes clear from the two parts of Figure 6.10, that one is trading the precision of the NSGraph and consequently low join cardinality and computational expense for the number of vertices being visited and thus I/O costs. However, there exists also a significant influence of the specific query and not only the general query class. As Table 6.1 shows, the queries K1 and K4 are isomorphic as are the queries K2 and K3. Thus the observable different response to the different data groupings must be a consequence of the different selectivity of its constituent predicates, favouring different query strategies and leading to different execution performance.

Figure 6.11 shows the response of four different query strategies to the linear Query K2, which is a typical query from the target class with moderate predicate selectivity. The query algorithms not shown respond in a very similar way to the ones presented, i.e. top-down and bottom-up merge-join algorithm behave almost identically for this particular query, as do the pessimistic and optimistic

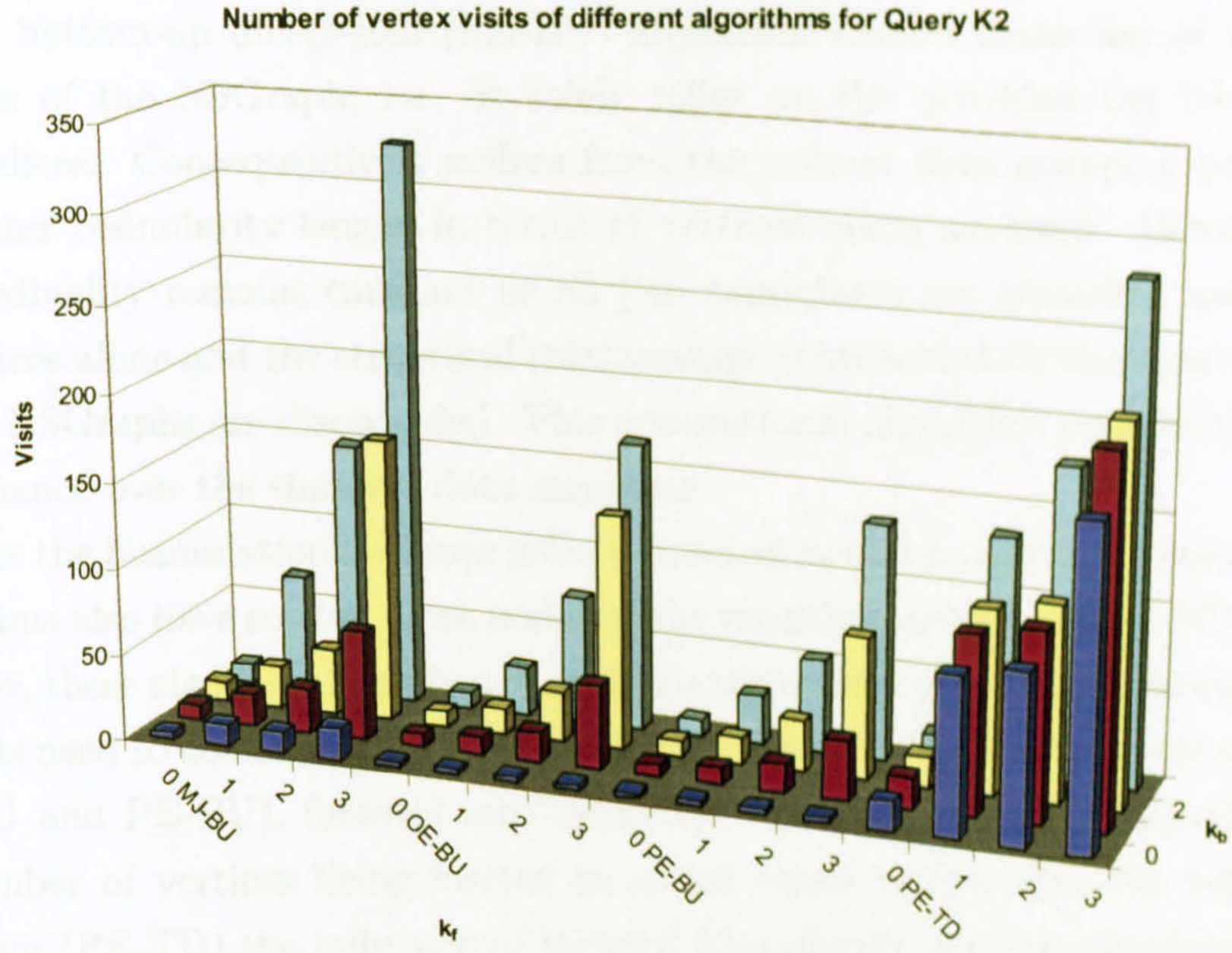


(a) Number of vertices visited

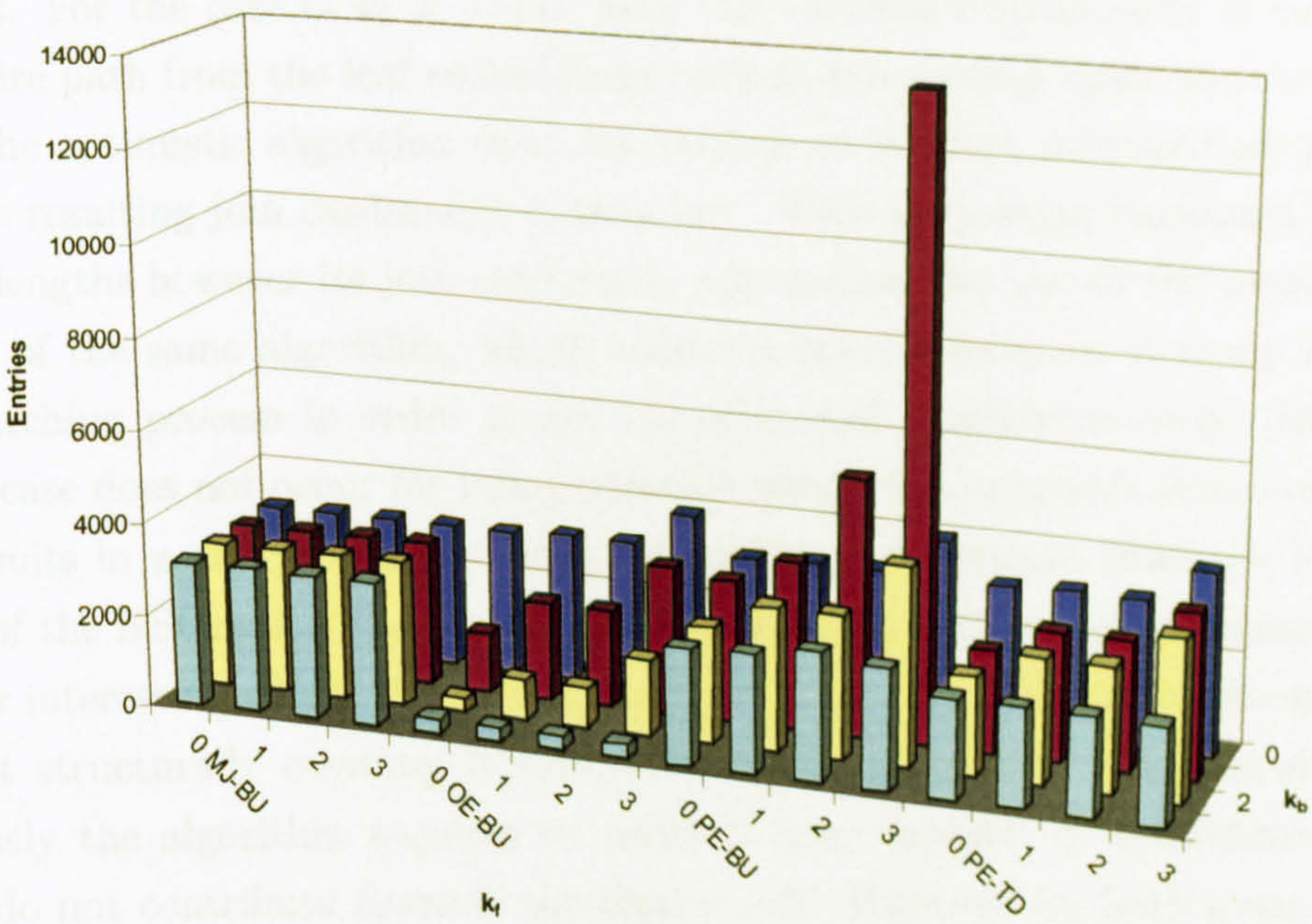


(b) Total cardinality of joins

Fig. 6.10: The response of the four different queries to different NSGraphs



(a) Number of vertices visited



(b) Total cardinality of joins

Fig. 6.11: The response of four different query algorithms to Query K2

top-down embedding algorithms.

The bottom-up merge-join (MJ-BU) algorithm cannot make use of the advantages of the NSGraph, i.e. it solely relies on the provided tag label and data indices. Consequently it suffers from the refined data grouping provided by greater bisimilarity length in terms of vertices being accessed. However its join cardinality remains constant as all join candidates are identified using the flat indices alone and the structural relationships represented by the shape of the various NSGraphs are disregarded. This conventional algorithm exposes the best performance over the simplest data structure.

Since the bisimulation becomes more refined as k_b and k_f grow, the embedding algorithms also have to visit more nodes of the resulting, more complex NSGraph. However, these algorithms can make use of the structural properties provided by it and thus need to consider less entries for their joins. For both bottom-up variants (OE-BU and PE-BU), forward and backward bisimilarity length contribute to the number of vertices being visited in about equal terms. For the top-down algorithm (PE-TD) the influence of forward bisimilarity is more significant than the influence of backward bisimilarity.

Again the k_b-axis was inverted for the graph shown in Figure 6.11(b). Here the difference between optimistic and pessimistic bottom-up embedding becomes obvious. For the case of $k_b = 3$, i.e. once the backward bisimilarity is covering the entire path from the leaf embeddings back to the rooting `open-auction` vertices, the optimistic algorithm does not require to validate intermediate results and the resulting join cardinality is very low. With decreasing backward bisimilarity lengths however its join cardinality approaches the one of the pessimistic variant of the same algorithm, which validates entry references at every step of the matching process in order to get rid of invalid candidates early. Because such a case does not occur for this particular query this approach is sub-optimal and results in a almost constant join cardinality, independent from the level of detail of the NSGraph. The pessimistic bottom-up algorithm also demonstrates another interesting effect. For $k_b = 1$ and $k_f = 3$, i.e. for a relatively fine-grained but not structurally covering NSGraph, the join cardinality increases sharply. Obviously the algorithm requires to verify a large number of candidates here, which do not contribute towards the final result. However for both greater and smaller backward bisimilarity this metric decreases. It shows that depending on the cardinality of individual candidate sets for parts of a query graph, the join cardinality does not need to increase monotonically with decreasing precision of

the data structure. The example chosen proves that there are cases where a coarser NSGraph will be more appropriate to a given query and require fewer candidates to be joined.

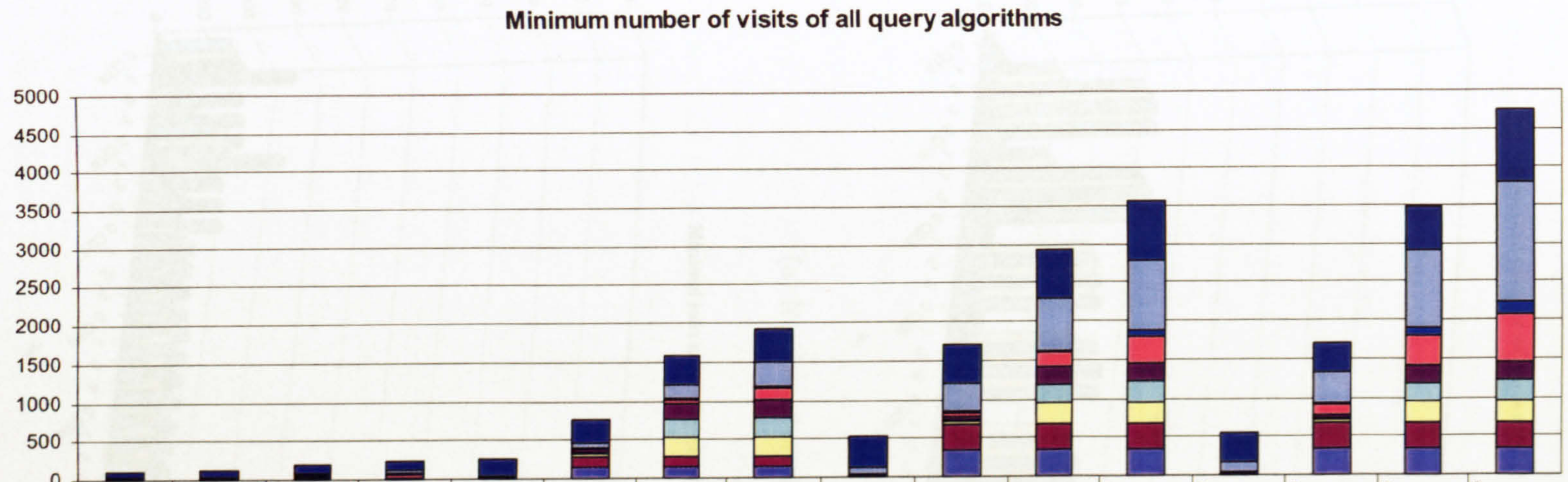
The pessimistic top-down embedding strategy also shows a strong correlation between decreased precision of the backward bisimilarity length and the cardinality of required joins. The forward bisimilarity length however has no noticeable effect on the join cardinality. Apparently many bids from 1999 belong to open auctions, but not necessarily vice versa.

6.5.3.2 Results for the Branching Tree Patterns

The results obtained using the set of branching queries are similar to those obtained for linear queries, which supports the claim that the provided design is equally suitable for each class of queries. In terms of absolute values, the results exceed those of the linear queries, which is expected given their higher complexity and expressive power. Figure 6.12 shows the minimum number of vertices being visited by any query algorithm for the set of benchmark queries over different NSGraphs. The data confirm the results from the linear case. Here the increase of vertices being visited is monotonic both for increasing backward and forward bisimilarity. Queries Q4a – b, which include regular expression over the recursive `description` part of the source dominate the number of visits required.

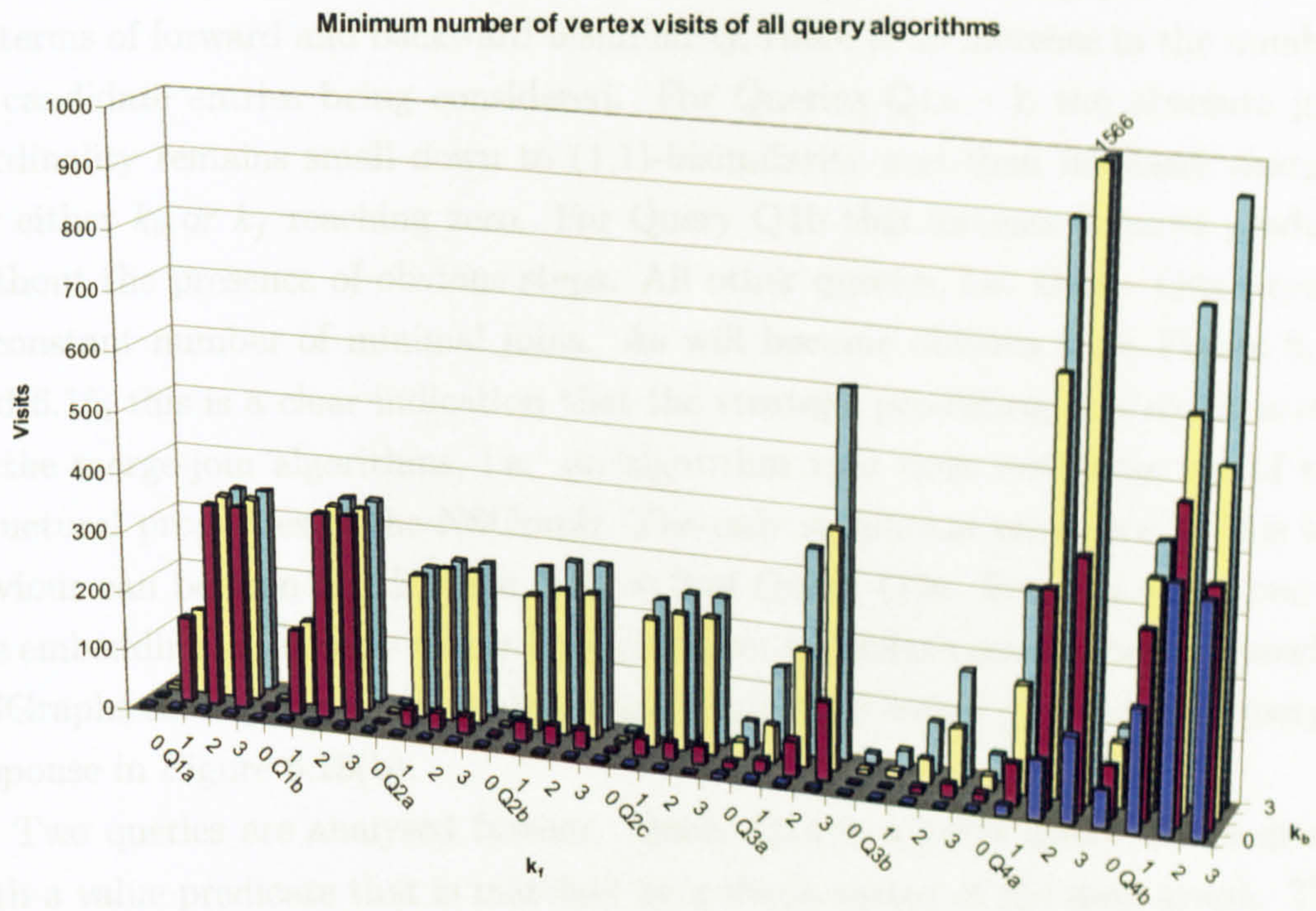
Figure 6.13 makes this more obvious and shows that not only the amount of data being considered grows for these queries, but also the computational power required to perform the required joins. There exist significant differences between the responses of the optimal number of query visits and total join cardinality for the different queries. For queries Q1a – Q1b the forward bisimilarity length dominates the number of vertices being visited (Figure 6.13(a)), whereas for queries Q2a – Q2c the backward bisimilarity dominates. For the remaining Queries Q3a – Q4b both factors notably influence the number of vertices being visited. Given that with increased precision of the bisimulation the total number of vertices also grows, this is actually the expected behaviour. However, queries Q1a – Q2c prove that even a continually growing NSGraph does not imply a continually growing number of vertices being visited.

The minimal total number of entries being joined during query execution (Figure 6.13(b)) only reveals two different classes of queries in this set. Queries Q1a – Q1b and Q4b can be resolved with fewest joins for the most precise NSGraph,

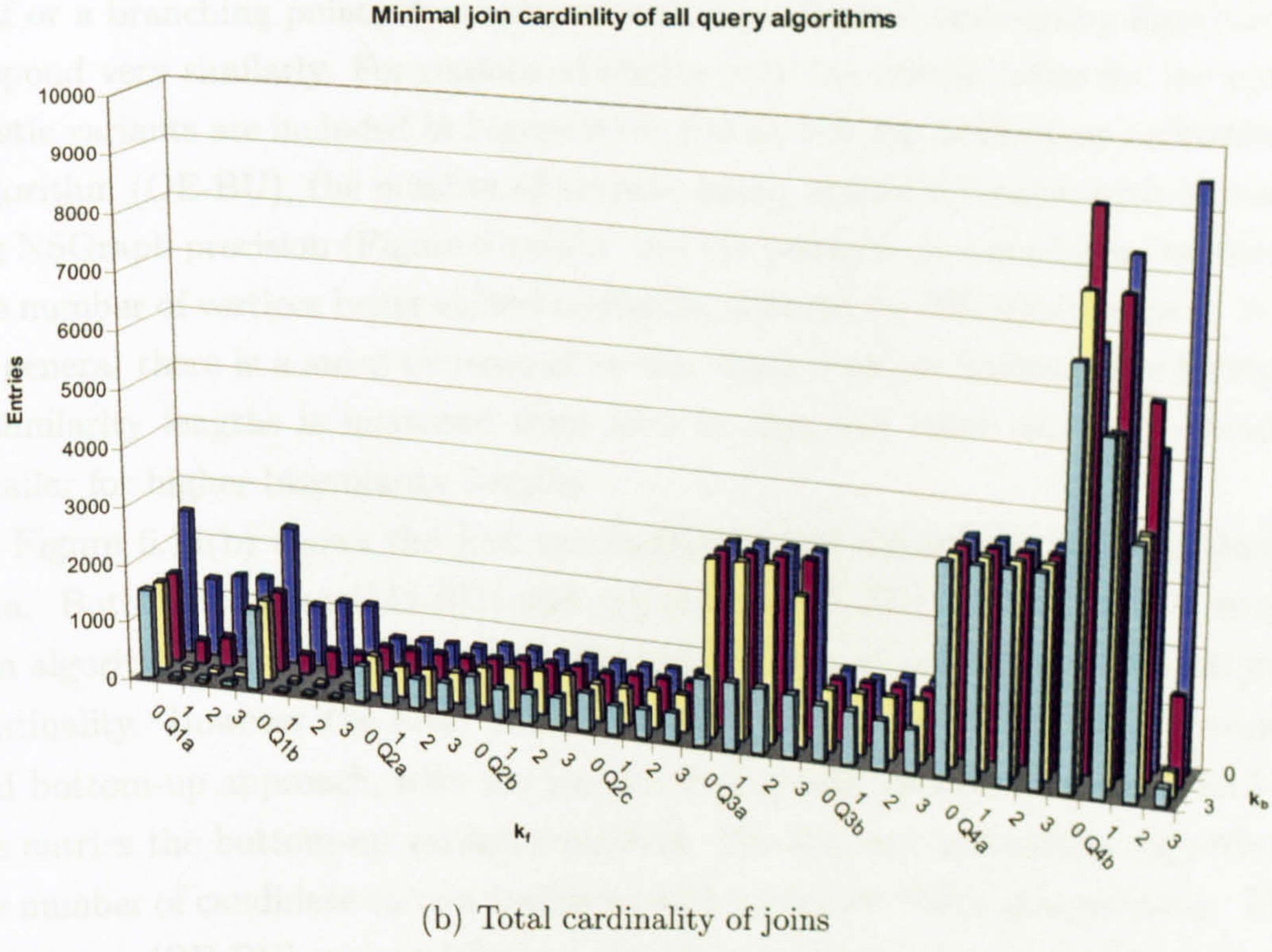


k_b	0				1				2				3			
k_f	0				1				2				3			
Q4b	61	84	104	111	196	305	368	416	397	506	628	783	376	384	553	951
Q4a	4	12	24	43	18	68	177	317	97	352	674	903	140	409	1024	1566
Q3b	3	3	8	18	3	4	13	29	3	9	39	82	3	21	95	172
Q3a	6	10	27	43	6	20	60	141	6	58	189	342	6	128	392	607
Q2c	4	4	4	4	4	30	211	229	4	30	225	244	4	30	225	244
Q2b	4	4	4	4	4	31	228	261	4	31	242	276	4	31	242	276
Q2a	3	3	3	3	3	30	239	239	3	30	256	256	3	30	256	256
Q1b	5	5	5	5	6	141	141	141	6	343	343	343	6	343	343	343
Q1a	5	5	5	5	6	141	141	141	6	341	341	341	6	341	341	341

Fig. 6.12: The number of vertex visits used by the best query strategy for each of the branching query patterns over the bisimilarity of the NSGraph



(a) Number of vertices visited



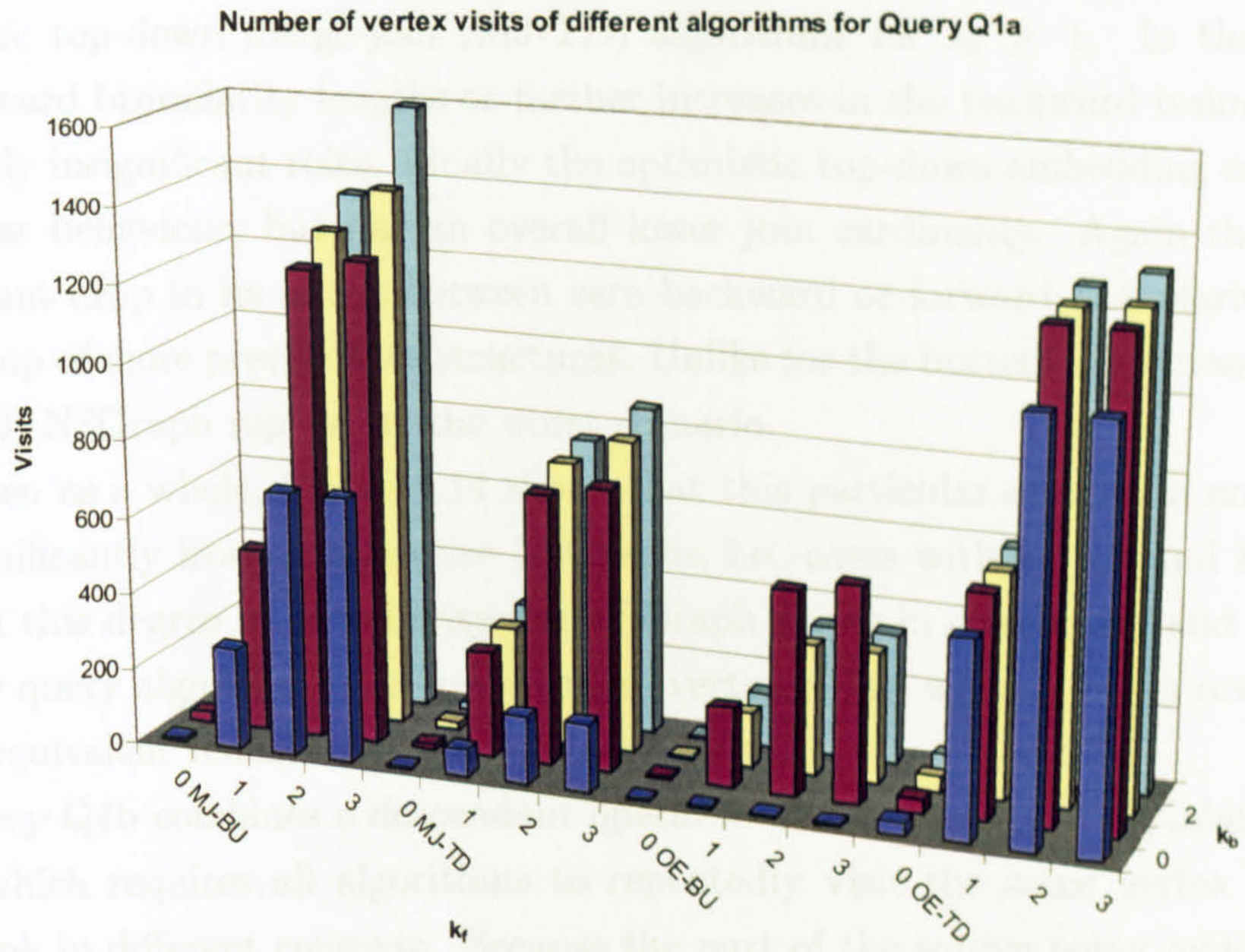
(b) Total cardinality of joins

Fig. 6.13: The response of the nine different queries to different NSGraphs

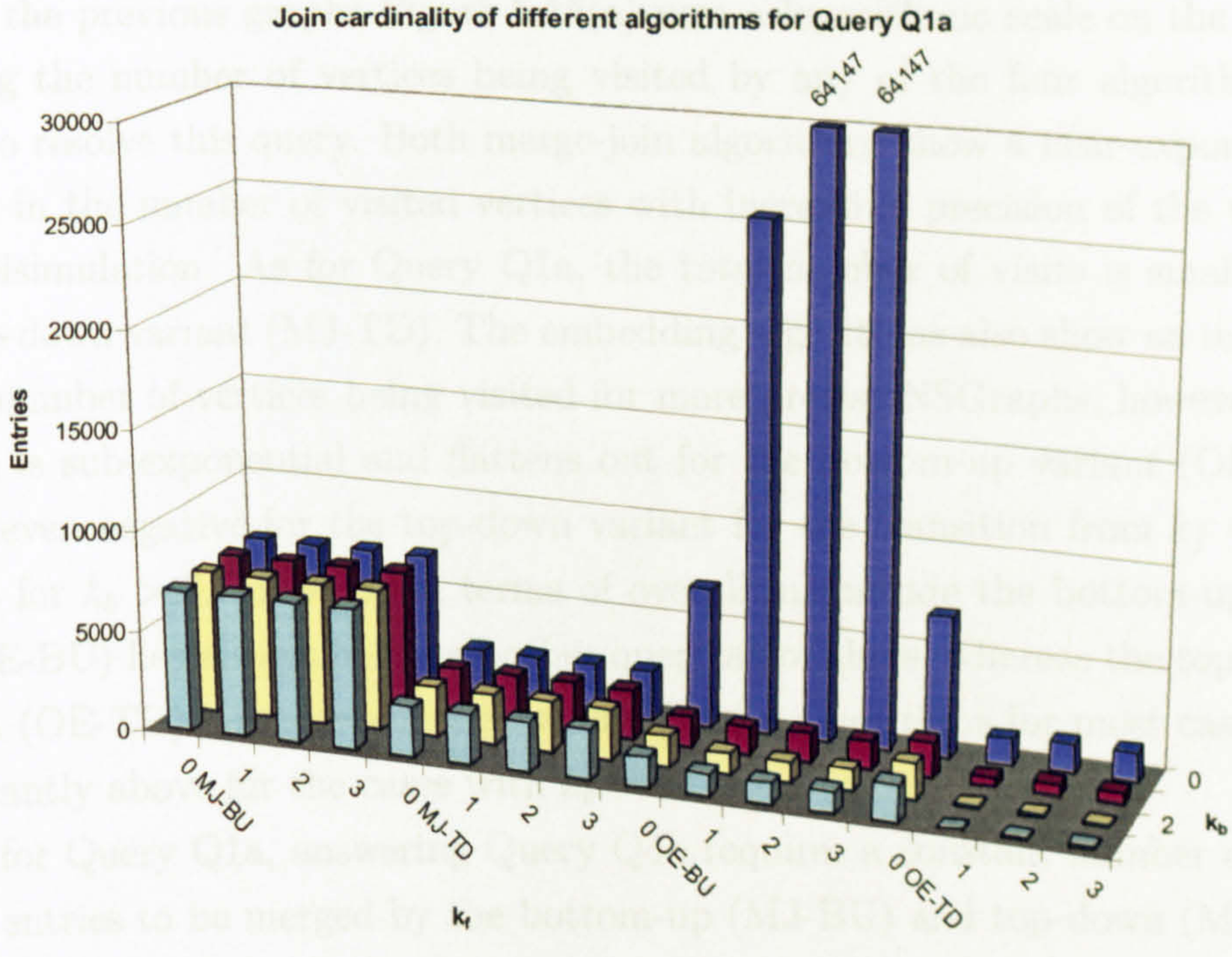
i.e. the graph based on the (3,3)-bisimulation. With decreasing precision, both in terms of forward and backward bisimilarity, there is an increase in the number of candidate entries being considered. For Queries Q1a – b the absolute join cardinality remains small down to (1,1)-bisimilarity and then increases sharply for either k_b or k_f reaching zero. For Query Q4b this increase is more gradual without the presence of obvious steps. All other queries, i.e. Q2a – Q4a expose a constant number of minimal joins. As will become obvious from Figure 6.14 and 6.15, this is a clear indication that the strategy producing this result is one of the merge-join algorithms, i.e. an algorithm that does not make use of the structural properties of the NSGraph. The only significant exception to this behaviour can be seen for the case of $k_b = 3$ of Query Q3a. For this query one of the embedding algorithms requires to join fewer candidate entries for very precise NSGraphs than the merge-join algorithm. Thus there exists a step in this query's response in Figure 6.13(b).

Two queries are analysed further. Query Q1a is a point query, i.e. a query with a value predicate that is matched by a single vertex of the data graph. The response of four of the different query strategies to this query is presented in Figure 6.14. Due to its structure every predicate of the query pattern is either a leaf or a branching point, thus optimistic and pessimistic embedding algorithms respond very similarly. For reasons of clarity only the results taken for the optimistic variants are included in Figure 6.14. For all but the bottom-up embedding algorithm (OE-BU), the number of vertices being visited increases with increasing NSGraph precision (Figure 6.14(a)). For the bottom-up embedding, however, the number of vertices being visited is slightly reduced for NSGraphs with $k_b > 1$. In general there is a steep increase of vertex visits if either backward or forward bisimilarity lengths is increased from zero to one, but these increases become smaller for higher bisimilarity lengths.

Figure 6.14(b) shows the join cardinality of the algorithms used for Query Q1a. Both bottom-up (MJ-BU) and top-down (MJ-TD) variants of the merge join algorithm are unaffected by the data structure used and have a constant join cardinality. However the total number of entries considers differs for top-down and bottom-up approach, with the top-down approach looking at only a third of the entries the bottom-up variant considers. For the two embedding algorithms the number of candidate entries decreases with increased NSGraph precision. The bottom-up (OE-BU) variant becomes almost unusable for the case of $k_b = 0$ and $k_f \geq 1$ because it requires more than an order of magnitude more candidates



(a) Number of vertices visited



(b) Total cardinality of joins

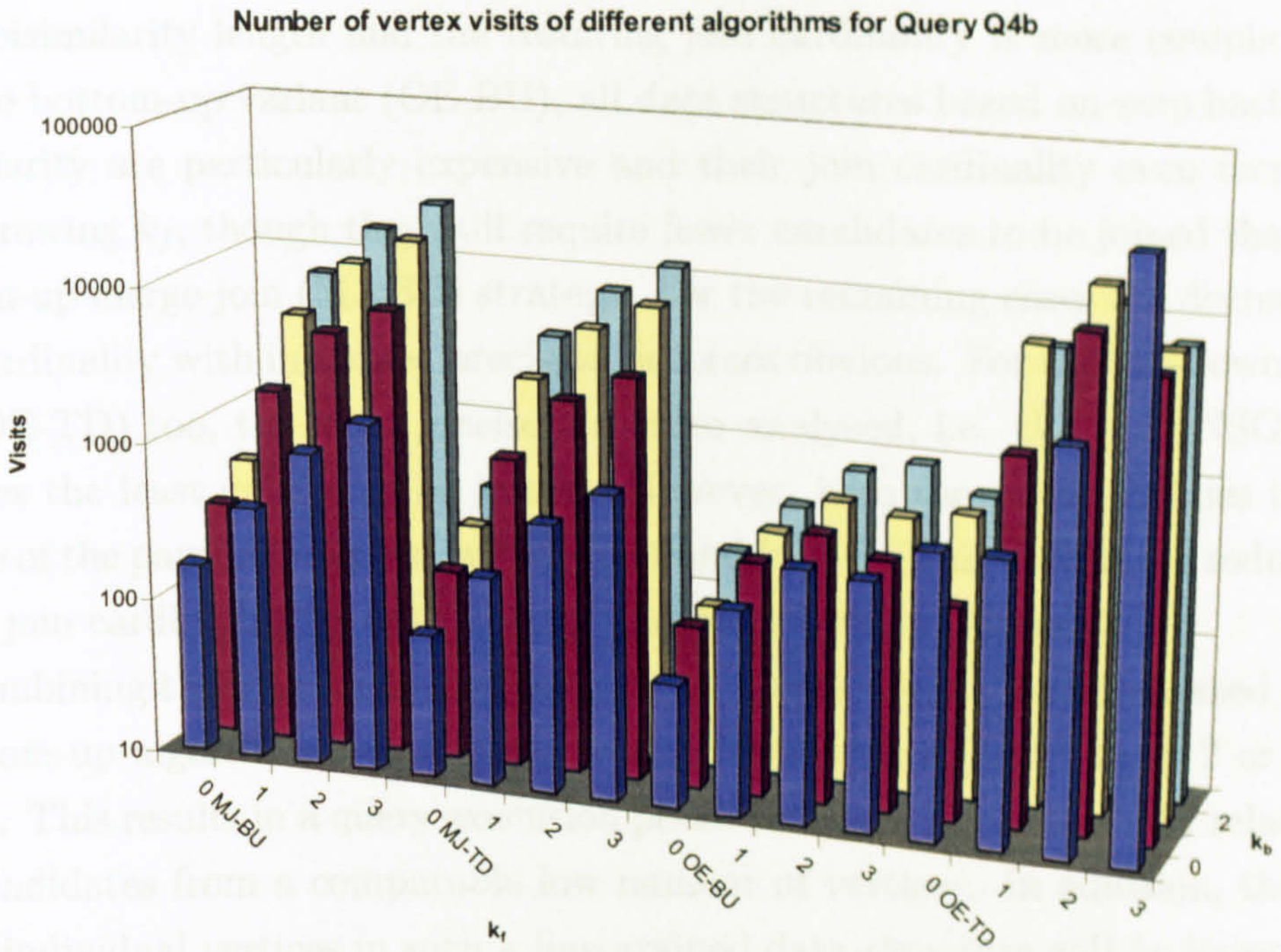
Fig. 6.14: The response of four different query algorithms to Query Q1a

to be joined. However this is drastically reduced to a number of joins smaller than the top-down merge-join (MJ-TD) algorithms for $k_b \geq 1$. In this case the forward bisimilarity lengths or further increases in the backward bisimilarity play only insignificant roles. Finally the optimistic top-down embedding exposes a similar behaviour, but has an overall lower join cardinality. Again the only significant drop in its size is between zero backward or forward bisimilarity and the group of more precise data structures. Unlike for the bottom-up variant, here the (0,0)-NSGraph represents the worst scenario.

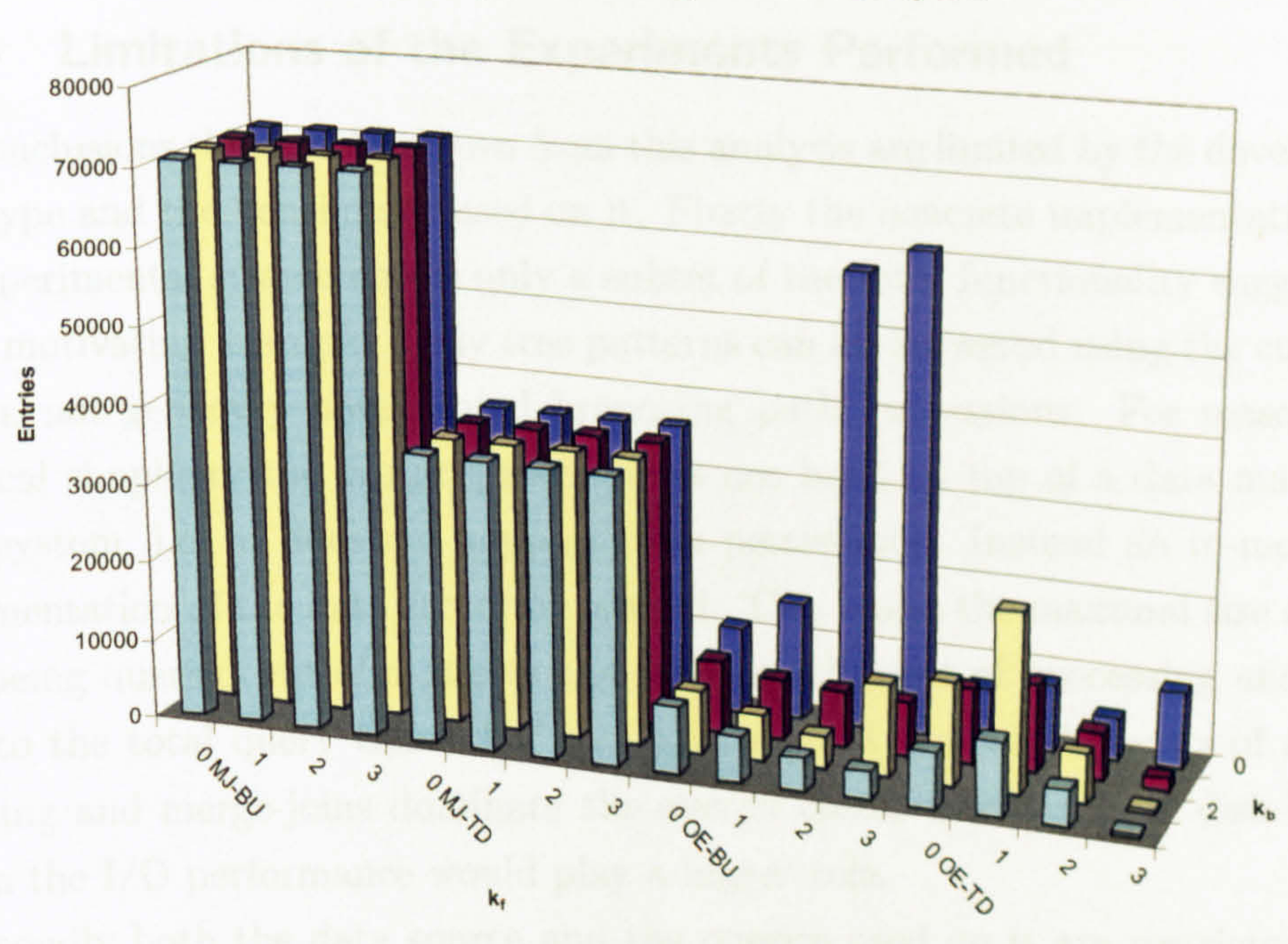
Taken as a whole, Figure 6.14 shows that this particular query can not benefit significantly from very precise NSGraphs, i.e. cases with $k_f > 1$ and $k_b > 1$. Beyond this degree of bisimilarity the NSGraph grows in complexity and consequently query algorithm need to visit more vertices of it, without being rewarded by an equivalent reduction in the join cardinality.

Query Q4b combines a descendant operator with the use of a tag label wildcard, which requires all algorithms to repeatedly visit the same vertex of the NSGraph in different contexts. Because the part of the source being matched is recursive in nature, its corresponding NSGraphs contain cycles. Thus the total number of vertices being visited exceeds the size of the vertex set in many cases. Unlike the previous graphs Figure 6.15(a) uses a logarithmic scale on the z-axis showing the number of vertices being visited by any of the four algorithms in order to resolve this query. Both merge-join algorithms show a near-exponential growth in the number of visited vertices with increasing precision of the underlying bisimulation. As for Query Q1a, the total number of visits is smaller for the top-down variant (MJ-TD). The embedding algorithms also show an increase in the number of vertices being visited for more precise NSGraphs, however this growth is sub-exponential and flattens out for the bottom-up variant (OE-BU) and is even negative for the top-down variant for the transition from $k_f = 2$ to $k_f = 3$ for $k_b > 0$. However in terms of overall magnitude the bottom-up variant (OE-BU) lies always below all other query algorithms, whereas the top-down version (OE-TD) lies between the two merge-join algorithms for most cases but significantly above for the cases with $k_b = 0$.

As for Query Q1a, answering Query Q4b requires a constant number of candidate entries to be merged by the bottom-up (MJ-BU) and top-down (MJ-TD) merge-join algorithms. Again the top-down strategy is the better choice, here by a about a factor of two. The two embedding algorithms produce a less regular result. Though the general trend confirms that more precise NSGraphs require



(a) Number of vertices visited



(b) Total cardinality of joins

Fig. 6.15: The response of four different query algorithms to Query Q4b

fewer entries to be validated, the relationship between the forward and backward bisimilarity length and the resulting join cardinality is more complicated. For the bottom-up variant (OE-BU), all data structures based on zero backward bisimilarity are particularly expensive and their join cardinality even increases with growing k_f, though they still require fewer candidates to be joined than the bottom-up merge-join (MJ-BU) strategy. For the remaining cases the decrease in join cardinality with increased precision becomes obvious. For the top-down variant (OE-TD) too, the most precise structure analysed, i.e. the (3,3)-NSGraph, requires the least entries to be joined. However, here the maximum lies in the middle of the parameter-space, with $k_f = 1$ and $k_b = 2$. This indicates a reduction in the join cardinality for both coarser and more refined NSGraphs.

Combining the observations of both parts, Query Q4b is best addressed using a bottom-up algorithm over a fairly precise NSGraph, e.g $k_f = k_b = 2$ or $k_f = k_b = 3$. This results in a query execution phase that only needs to verify relatively few candidates from a comparable low number of vertices. In addition, the size of the individual vertices in such a fine-grained data structure will be lower than those of a very coarse one such as the (0,0)-NSGraph. This, however, is not taken into account by the performance metrics used.

6.5.4 Limitations of the Experiments Performed

The conclusions that can be drawn from this analysis are limited by the developed prototype and the benchmark used on it. Firstly the concrete implementation of the experimental system covers only a subset of the total functionality suggested in the motivating example. Only tree patterns can be answered using the current system, not generally unrestricted branching path expressions. For reasons of technical simplicity the prototype system is not built on top of a data management system, i.e. it does not support data persistence. Instead an in-memory implementation of the data structure is used. This limits the maximal size of the data being queried but also affects the relative influence of processing and I/O costs to the total query time. For the experimental system the costs of string matching and merge-joins dominate the overall costs, whereas for a disk-based system the I/O performance would play a bigger role.

Secondly both the data source and the queries used on it are restricted. As the area of semistructured data processing is still immature in comparison with relational technology, there exists only few benchmarks. In addition, published

benchmarks are often biased to analyse very particular but limited aspects of SSD processing.

Thirdly the analysis of the execution metrics of the benchmark queries used in the experiments presented here produce complex data over a four-dimensional parameter space. A complete analysis of the data generated remains a task for a future investigation.

6.6 Summary

This chapter motivates and describes a design for a hybrid query system that is able to deal with queries containing both structural and atomic value predicates. It shows how to combine two different data groupings, each of which was previously shown to provide an efficient solution for one of these two aspects. A numbering scheme for the nodes of the distinct spanning tree serves as the glue combining the two partial solutions and allows for a easy transition between them.

The experimental system based on this design is able to resolve branching tree patterns with arbitrary value predicates using different query algorithms adapted to work on top of this data structure. Its evaluation has shown that both the type of the query and the cardinality of its constituting predicates have a significant influence on its execution performance.

Moreover the design and its implementation allow variation in the coarseness of the bisimulation on which it is based. This is crucial in order to show the impact that a particular grouping has on the evaluation of a query. It also makes obvious the trade-off between the precision of a used data structure with respect to a class of queries and the computational expense required to resolve them. The experiments performed clearly indicate that although the bisimulation grows uniformly with increased bisimilarity length (Section 4.3.2), important performance metrics do not closely follow this increase.

7. CONCLUSIONS

Scientific Results

"Science is always wrong.
It never solves a problem without creating ten more."

George Bernard Shaw, *The Doctor's Dilemma*, 1911

7.1 Results

This chapter combines the observations gained from looking at the issue of data groupings from various angles. The concept was originally introduced and used by the optimisation model presented in Chapter 3. Chapter 4 has formalised the abstract concept of data groupings into the more tractable problem of domains for graph structured data. Different definitions provided there were used through the following experimental chapters. Chapter 5 demonstrated the beneficial effects data groupings have on efficient SSD representation, and finally Chapter 6 analysed the influence of such groupings on a number of query algorithms.

7.1.1 Data Groupings as Explanatory Tool for Query Optimisation Techniques

As in the relational case, the success or failure of semistructured data management will be decided by its performance rather than its theoretical properties. Consequently much research has been devoted to optimising SSD management systems and was referred to throughout this thesis. There are, however, no widely accepted models that describe this process in general. It is the hypothesis of this thesis that a significant element of optimisation techniques is based on data groupings. To support this hypothesis, an abstract model of the general optimisation process has been designed and was detailed in Chapter 3.

The model developed was used to describe three different optimisation problems (Section 3.3). One of these was based on the author's work on SSD com-

pression, one on work performed by Kaushik et al. [KS+02, KB+02] on structural indices based on local bisimilarity and the last one addressed a purely hypothetical class of queries. The solutions to all of these problems could be described in terms of the model proposed and thus support its utility.

7.1.2 Domains for Graph-Structured Data

Domains, a concept fundamental to the relational paradigm, are noticeable absent from the definition of SSD models. Yet many optimisation methods employed by relational technology depend on this concept. If one is to transfer such optimisations to the semistructured case, the concept of domains needs to be re-established there first. This thesis presents a definition of the concept of a domain structure in general, which depends on the individual application semantics. This concept is refined into a more practical definition for application independent domain structures, which are based on graph properties alone. A subclass of these are the *equivalence domain structures*, which are based on some given equivalence relationship between vertices of a data graph.

Because multiple equivalence relationships can be attached to a single source, there exists a multitude of possible definitions of domain structures. Each such structure implies a different set of actual domains for a given source. Seven examples of application independent domain structures and one application dependent domain structure were introduced Section 4.2. They were taken from other research in the area of SSD processing, in particular from work on semistructured indexing. The seven application independent definitions were applied to a collection of example sources and the resulting domains were statistically analysed. For the first five examples that are fixed by definition, a rough classification of three different types of sources could be established (Section 4.3.1). Sources that are essentially a semistructured encoding of regular data result in simple domain structures, exposing their inherent regularity. A second group of sources still exposes structural coherence of the established domains, but members of these domains can be found in different contexts. A last group does not expose either structural or contextual homogeneity and thus results in very complicated domain structures. The last two application independent domain structures (Example 4.6 and 4.7) can be parameterised to reflect the desired degree of coherence between members of a discovered domain. This approach was applied to a single graph-structured source in order to find pathological points in the parameter space. The

normal distribution of the experimental results shows that pathological points do not exist for the specific source used (Section 4.3.2).

The even less restricted case of semantic domains is investigated further by the investigation of type projection of semistructured data. This work extends beyond the scope of this thesis and is included in Appendix A.

7.1.3 Compression of Semistructured Data

Compression algorithms for SSD can be improved by applying them to semantically homogeneous domains. Using Example 4.2 of Chapter 4, this work shows how dictionary compression previously used in the relational case can be transferred successfully to SSD.

The implemented DDOM prototype saves up to 80% of memory in comparison in other implementations of the DOM model for data-centric documents. If value indices are created on the resulting dictionaries, their total size in combination with the original data structure is still significantly smaller than the conventional, non-indexed implementations.

Access through the DOM interface prohibits an external query engine from making use of the developed data structure in general and the value indices in particular. Thus external query engines accessing the data structure through this interface perform poorly for many queries. However, accessing the DDOM data directly in a way that makes best use of the dictionaries allows more efficient querying. The experimental analysis suggests that the provided solution is beneficial for a subset of queries that are based on selective value predicates and can outperform conventional implementations on this type of query.

7.1.4 Hybrid Querying

While most previous research addresses either the structural or the atomic value part of a query, Chapter 6 provides an approach for integrating the two. This is important, as reducing the volume of data being considered is an important aspect of every query optimiser. Depending on the specific query, this might be achieved by either strutural or atomic value predicates and an integration at an early stage in the query process can thus help to filter out irrelevant information earlier in the process.

A hybrid data structure was developed, in which numbering schemes were

used as a bridging element between data groupings based on structural properties identified using local bisimilarity and atomic value dictionaries. The experimental implementation allows analysis of the influence that different data groupings have on a particular query algorithm. The performance metrics of several variants of subgraph embeddings vary greatly in response to the different data groupings parameterised by the bisimilarity length of the underlying bisimulation, whereas two variants of a merge-join algorithm, which do not make use of the hybrid data structure, are unaffected of different data groupings.

An important observation drawn from the experimental results is that the actual response is more specific to the actual instance of a query than the general query class as suggested in the general optimisation model. This issue is further discussed in Section 7.2.4. But even if one considers a single query the response to varying levels of precision of the data grouping is not uniform. This is in contrast to the findings of Section 4.3.2. Local minima in the parameter space suggest particularly useful data groupings, which capture the aspects of a source relevant to the query without increasing the complexity of the domain structure too far.

7.2 Limitations and Future Work

The abstract model proposed in this thesis is amenable to experimental investigation. Two such experiments were provided in chapters 5 and 6. Further support can be gained from other related work reviewed by this thesis in general and by the investigation included in Appendix A in particular. However, there exist particular aspects, whose relationship to data groupings deserve a further investigation.

7.2.1 Choice of Data Sources

The example data sources used to test the hypotheses presented are deficient in a number of ways. First and foremost, many of them originate from structured data sources and only exhibit a limited degree of irregularity.

Many of the used sources represent hierarchical structures rather than true graphs. Though many sources describe data that is intrinsically graph structured, e.g. the DBLP bibliographic database, the limited treatment of graphs by the XML standard prevents application developers from exposing this information in the physical representation. Instead it is encoded implicitly in their application

semantics. This is problematic for an automatic analysis of such data as required for this thesis. The two cross-referencing mechanisms, which are provided by the W3C, ID:IDREF-attributes and XPointer, are both too limiting for many applications. In addition, ID:IDREF references require the presence of a document schema and only work within one document, which is too limiting for many data-centric applications. XPointer is not part of the XML standard itself and has only received little attention.

Such shortcomings will be resolved as the technology matures. It would be of interest to repeat similar experiments to the ones described here, once large, truly graph structured, real world data sources are available. Such data is becoming available now. Examples are the IMDB movie database[1] or Amazon's book catalogue[2]. Because of their size and their economic value, the information contained in these sources is only available through querying for individual entries rather than as a complete database. Reconstructing, storing and querying a large proportion of such sources opens experiments to legal, technical and practical questions.

7.2.2 Extension of Query Language

The query language used throughout this thesis is that of branching path expression extended by atomic value predicates (Section 2.3.1). Like the original branching path expressions, it ignores the order of outgoing edges from a common vertex. Core XPath on the other hand supports queries depending on this property and thus extends beyond the expressive capacity investigated by this thesis. The numbering scheme used in Chapter 6 also encodes order. It would be worthwhile to investigate how this information can be exposed to and utilised by the query process. The origins of this situation were highlighted in Section 4.2. An investigation into how important the issue of order is for data centric-applications is still outstanding.

The experimental systems described in chapters 5 and 6 only cover tree patterns but not general graphs. The numbering scheme used as part of the data structure developed in Chapter 6 is also restricted to trees rather than graphs and prevents a straightforward extension to this general case. Finding numbering schemes for general graphs that at least encode local relationships between

[1] `http://www.imdb.com`
[2] `http://www.amazon.com`

vertices would support such an investigation.

7.2.3 Domain Statistics as Metrics for Semistructured Data

The classifications of data sources established in Section 4.3.1 using competing domain structure definitions does not closely follow what is known as the classification into data- and document-centric and hybrid documents [KM02]. Both the set of domain definitions used and the set of sources analysed was necessarily restricted for the scope of this thesis. It would be of interest to extend such an investigation to see whether the results could be used as an additional set of metrics to classify instances of SSD. Up to now such analysis was based directly on the particular instance [KSH02] or a provided document schema [AL02, Sah00].

7.2.4 Query Planning Based on Data Statistics

The response of different data groupings on the two sets of benchmark queries used in Section 6.5 have shown interesting results. Even for isomorphic queries, i.e. queries belonging to the same complexity class, the response could vary significantly. Whereas for some queries the forward bisimilarity length determined the query costs, others were more affected by the backward bisimilarity, a combination of both or neither. This different behaviour for similar queries can only be explained with the different selectivity of the individual predicates constituting an individual query. Such issues provide an interesting approach to query optimisation based on the statistical data gathered during data classification.

7.2.5 Comparison with Information Retrieval Systems

All work presented here was concerned with the processing of or querying for exact answers to given problems. This strict discipline might incur a severe performance penalty in comparison with laxer IR systems, which try to give reasonably approximate answers. Even if exact answers are sought, computing approximate answers first and verifying them in a later process might yield performance benefits. An extensive practical comparison, based on the analysis of application requirements, between such approximate and accurate systems would be of interest.

REFERENCES

[AB84] Serge Abiteboul and Nicole Bidoit. Non first normal form relations to represent hierarchically organized data. In *PODS 1984*, pages 191–200, 1984.

[ABS00] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kaufmann, San Francisco, CA, USA, 2000.

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Design and Analysis of Computer algorithms.* Addison-Wesley, 1974.

[AL02] Marcelo Arenas and Leonid Libkin. A normal form for XML documents. In Lucian Popa, editor, *PODS 2002*, pages 85–96, Madison, Wisconsin, USA, 2002. ACM.

[Ala96] Aladdin Enterprises. GZIP file format specification version 4.3. RFC 1952, May 1996. `http://www.ietf.org/rfc/rfc1952.txt`.

[AO+99] Malcolm Atkinson, Maria E. Orlowska, et al., editors. *Proceedings of the 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK.* Morgan Kaufmann, 1999.

[AQ+97] Serge Abiteboul, Dallan Quass, et al. The lorel query language for semistructured data. *Int. J. on Digital Libraries*, 1(1):68–88, Apr 1997.

[BB99] John Bosak and Tim Bray. XML and the second-generation web. *Scientific American*, May 1999.

[BC00] Angela Bonifati and Stefano Ceri. Comparative analysis of five XML query languages. *SIGMOD Record*, 29(1):68–79, 2000.

[BF00] André Bergholz and Johann Cristoph Freytag. Querying semistructured data based on schema matching. In Richard Connor and Alberto Mendelzon, editors, *DBPL'99*, volume 1949 of *LNCS*, pages 168–183. Springer, 2000.

[BGK03] Peter Buneman, Martin Grohe, and Christoph Koch. Path queries on compressed XML. In *VLDB 2003*, 2003.

[BI+02] Philip A. Bernstein, Yannis E. Ioannidis, et al., editors. *Proceedings of the 28th International Conference on Very Large Databases*, Hong Kong, China, 2002. Morgan Kaufmann.

[BKS02] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: Optimal XML pattern matching. In SIGMOD2002 [SIG02], pages 310–321.

[BL82] Dines Bjørner and Hans Henrik Løvengreen. Formalization of database systems - and a formal definition of IMS. In *VLDB 1982*, pages 334–347. Morgan Kaufmann, 1982.

[BLCG92] Tim Berners-Lee, Robert Cailliau, and Jean-Francois Groff. The world-wide web. *Computer Networks and ISDN Systems*, 25(4–5):454–459, 1992.

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, May 2001.

[Bra03] Chris Brandin. Xml data management: Native XML and XML-enabled database systems. In Akmal B. Chaudhri, Awais Rashid, and Roberto Zicari, editors, *XML Data Management*, chapter Information Modeling with XML, pages 3–17. Addison Wesley, 2003.

[Cat94] R. G. G. Cattell. *The Object Database Standard: ODMG-93 (Release 1.1)*. Morgan Kaufmann, 1994.

[CB03] Byron Choi and Peter Buneman. XML vectorization: A column-based XML storage model. Technical Report MS-CIS-03-17, University of Pennsylvania, 2003.

[Che01] James Cheney. Compressing XML with multiplexed hierarchical PPM models. In *DCC 2001*, pages 163–172. IEEE Computer Society, 2001.

[Cho02] Byron Choi. What are real DTDs like? In Mary F. Fernandez and Yannis Papakonstantinou, editors, *WebDB '02*, pages 43–48, Madison, WI, USA, Jun 2002. ACM PODS/SIGMOD. Informal proceedings.

[CL+01] Richard Connor, David Lievens, et al. Extracting typed values from XML data. In *OOPSLA Workshop on Objects, XML and Databases*, 2001.

[CLO03] Qun Chen, Andrew Lim, and Kian Win Ong. D(K)-index: An adaptive structural summary for graph-structured data. In Alon Haley, Zachary Ives, and AnHai Doan, editors, *SIGMOD 2003*, pages 134–144, 2003.

[CMW98] W. Paul Cockshott, Douglas McGregor, and John Wilson. High-performance operations using a compressed database architecture. *Computer J.*, 41(5):283–296, 1998.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *CACM*, 13(6):377–387, 1970.

[CRU02] Akmal Chaudri, Erhard Rahm, and Rainer Unland, editors. *Web Databases 2002 (NODe '02 Workshop)*, Erfurt, Thuringia, Germany, Oct 2002. `http://dbs.uni-leipzig.de/webdb/webdb02/index.html`.

[CU02] Akmal Chaudri and Rainer Unland, editors. *XML-Based Data Management (EDBT 2001 Workshop)*, Prague, Czech Republic, Mar 2002.

[CV+02] Shu-Yao Chien, Zografoula Vagena, et al. Efficient structural joins on indexed XML documents. In Bernstein et al. [BI+02].

[DBT71] Report of the CODASYL data base task group, Apr 1971.

[DF+98] Alin Deutsch, Mary F. Fernandez, et al. XML-QL. In *QL '98* [W3C98]. `http://www.w3.org/TandS/QL/QL98/`.

[DFS99] Alin Deutsch, Mary Fernandez, and Dan Suciu. Storing semistructured data with STORED. In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *SIGMOD 1999*, volume 28 of *SIGMOD Record*, pages 431–442. ACM Press, 1999.

[Die82] Paul F. Dietz. Maintaining order in a linked list. In *STOCS'82*, pages 122–127. ACM Press, 1982.

[DOM00] World Wide Web Consortium. *Appendix C: Java Language Binding*, 2000. http://www.w3.org/TR/2000/WD-DOM-Level-1-20000929/java-language-binding.html.

[FH+02] J. Freire, J. Haritsa, et al. StatiX: Making XML count. In SIGMOD2002 [SIG02], pages 181–192.

[FK99] Daniela Florescu and Donald Kossmann. Storing and querying XML data using an RDBMS. *Data Engineering Bulletin*, 22(3):27–34, Sep 1999.

[FL+03] Johann-Christoph Freytag, Peter C. Lockemann, et al., editors. *Proceedings of the 29th International Conference on Very Large Databases*, Berlin, Germany, 2003. Morgan Kaufmann.

[FTS00] Mary F. Fernández, Wang-Chiew Tan, and Dan Suciu. SilkRoute: Trading between relations and XML. *Computer Networks*, 33(1–6):723–745, 2000.

[GMW99] R. Goldman, J. McHugh, and J. Widom. From semistructured data to XML: Migrating the Lore data model and query language. In Sophie Cluet and Tova Milo, editors, *WebDB'99*, Philadelphia, Pennsylvannia, USA, Jun 1999. INRIA. Informal Proceedings.

[Gru02] Torsten Grust. Accelerating XPath location steps. In SIGMOD2002 [SIG02], pages 109–120.

[GS00] Marc Girardot and Neel Sundaresun. Millau: An encoding format for efficient representation and exchange of XML over the web. In *WWW 1999*, volume 33 of *Computer Networks*, pages 747–765. Elsevier, Jun 2000.

[GW97] Roy Goldman and Jennifer Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In Matthias Jarke, Michael J. Carey, et al., editors, *VLDB 1997*, pages 436–445. Morgan Kaufmann, 1997.

[HB+03] Alan Halverson, Josef Burger, et al. Mixed mode XML query processing. In Freytag et al. [FL+03], pages 225–236.

[HMW02] Abu Sayed M. L. Hoque, Douglas R. McGregor, and John N. Wilson. Database compression using of-line dictionary methods. Technical report, Department of Computer and Information Science, University of Strathclyde, Glasgow, Scotland, UK, 2002.

[Hoq03] Abu Sayed M. L. Hoque. *Compression of Structured and Semi-Structured Information.* PhD thesis, University of Strathclyde, Glasgow, UK, 2003.

[HTM99] World Wide Web Consortium. *HTML 4.01 Specification*, W3C recommendation 24 December 1999 edition, 1999. `http://www.w3.org/TR/html401`.

[ICD02] *Proceedings of the 18th IEEE International Conference on Data Engineering (ICDE), San Jose, USA, February 2002*, 2002.

[Ioa96] Yannis E. Ioannidis. Query optimization. *ACM Comput. Surv.*, 28(1):121–123, 1996.

[JK84] Matthias Jarke and Jürgen Koch. Query optimization in database systems. *ACM Comput. Surv.*, 16(2):111–152, 1984.

[KB+02] Raghav Kaushik, Philip Bohannon, et al. Covering indexes for branching path queries. In SIGMOD2002 [SIG02], pages 133–144.

[KC02] Thomas Kudrass and Matthias Conrad. Management of XML documents in object-relational databases. In Chaudri and Unland [CU02], pages 69–82.

[KK+04a] Raghav Kaushik, Rajasekar Krishnamurthy, et al. On the integration of structure indexes and inverted lists. In Gerhard Weikum, Arnd Christian König, and Stefan Deßloch, editors, *SIGMOD 2004*, pages 779–790. ACM, 2004.

[KK+04b] Raghav Kaushik, Rajasekar Krishnamurthy, et al. On the integration of structure indexes and inverted lists. In *ICDE 2004*, page 829. IEEE Computer Society, 2004.

[KM02] Meike Klettke and Holger Meyer. *XML & Datenbanken – Konzepte, Sprachen und Systeme.* xml.bibliothek. dpunkt.verlag, Dec 2002.

[KS+02] Raghav Kaushik, Pradeep Shenoy, et al. Exploiting local similarities for indexing paths in graph-structured data. In ICDE2002 [ICD02].

[KSH02] Meike Klettke, Lars Schneider, and Andreas Heuer. Metrics for XML document collections. In Chaudri and Unland [CU02], pages 162–176.

[KYU02a] Dao Dinh Kha, Masatoshi Yoshikawa, and Shunsuke Uemura. Application of rUID in processing XML queries on structure and keywords. In *DEXA 2002*, volume 2453 of *LNCS*, pages 279–289, 2002.

[KYU02b] Dao Dinh Kha, Masatoshi Yoshikawa, and Shunsuke Uemura. A structural numbering scheme for XML data. In Chaudri and Unland [CU02], pages 91–108.

[LH87] Debra A. Lelewer and Daniel S. Hirschberg. Data compression. *ACM Computing Surveys*, 19:261–296, 1987.

[LM01] Quanzhong Li and Bongki Moon. Indexing and querying XML data for regular path expressions. In Peter M. G. Apers, Paolo Atzeni, et al., editors, *VLDB 2001*, pages 361–370. Morgan Kaufmann, 2001.

[LS00] Hartmut Liefke and Dan Suciu. XMill: An efficient compressor for XML data. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *SIGMOD 2000*, volume 29 of *SIGMOD Record*, pages 153–164, 2000.

[LY+96] Yong Kyu Lee, Seong-Joon Yoo, et al. Index structures for structured documents. In *Proceedings of the first ACM international conference on Digital libraries*, pages 91–99. ACM Press, 1996.

[MA+97] J. McHugh, S. Abiteboul, et al. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, Sep 1997.

[Mei02] Wolfgang Meier. eXist: An open source native XML database. In Chaudri et al. [CRU02], pages 169–183. `http://dbs.uni-leipzig.de/webdb/webdb02/index.html`.

[MS99] Tova Milo and Dan Suciu. Index structures for path expressions. In Catriel Beeri and Peter Buneman, editors, *ICDT 1999*, volume 1540 of *LNCS*, pages 277–295, 1999.

[MW+98] Jason McHugh, Jennifer Widom, et al. Indexing semistructured data. Technical report, Computer Science Department, Stanford University, 1998.

[MW99] Jason McHugh and Jennifer Widom. Query optimization for XML. In Atkinson et al. [AO+99], pages 315–326.

[ND+01] Jeffrey F. Naughton, David J. DeWitt, et al. The Niagara internet query system. *IEEE Data Eng. Bull.*, 24(2):27–33, 2001.

[Neu01] Mathias Neumüller. Compression of XML data. Master's thesis, University of Strathclyde, Glasgow, Scotland, UK, Sep 2001.

[Neu02] Mathias Neumüller. Compact data structures for querying XML. In Wolfgang Lindner and Július Štuller, editors, *EDBT 2002 PhD Workshop*, pages 127–130, Prague, Czech Republic, Mar 2002. MATFYZPRESS.

[NU+97] Svetlozar Nestorov, Jeffrey Ullman, et al. Representative objects: Concise representations of semistructured, hierarchical data. In *ICDE 1997*, pages 79–90, 1997.

[NW02] Mathias Neumüller and John N. Wilson. Improving XML processing using adapted data structures. In Chaudri et al. [CRU02], pages 63–77. `http://dbs.uni-leipzig.de/webdb/webdb02/index.html`.

[NW04] Mathias Neumüller and John N. Wilson. A model for querying semi-structured data through the exploitation of regular sub-structures. In *PREP 2004*, pages 117–118, University of Hertfordshire, Hatfield, Apr 2004.

[OM+02] Dan Olteanu, Holger Meuss, et al. XPath: Looking forward. In Chaudri and Unland [CU02], pages 249–263.

[PGMW95] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object exchange across heterogeneous information sources. In Philip S. Yu and Arbee L. P. Chen, editors, *ICDE 1995*, pages 251–260. IEEE Computer Society, 1995.

[PT87] Robert Paige and Robert E. Tarjan. Three partitioning refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, 1987.

[Ram03] Prakesh Ramanan. Covering indices for XML queries: Bisimulation - simulation = negation. In Freytag et al. [FL+03], pages 165–176.

[RB01] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, 2001.

[RLS98] Jonathan Robie, Joe Lapp, and David Schach. XML query language (XQL). In *QL '98* [W3C98]. `http://www.w3c.org/TandS/QL/QL98/pp/xql.html`.

[Sah00] Arnaud Sahuguet. Everything you ever wanted to know about DTDs, but were afraid to ask (extended abstract). In *WebDB (Selected Papers)*, pages 171–183, 2000.

[Sch01] Harald Schöning. Tamino – a DBMS designed for XML. In *ICDE 2001*, pages 149–154, Heidelberg, Germany, 2001. IEEE Computer Society.

[SGM86] International Organization for Standardization, Geneva. *Standard Generalized Markup Language (SGML)*, first edition, Oct 1986. Information Processing – Text and Office Systems.

[SIG02] *Proceedings of the ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 2002*, 2002.

[SM01] Neel Sundaresan and Reshad Moussa. Algorithms and programming models for efficient representation of xml for internet applications. In *WWW 2001*, pages 366–375. ACM Press, 2001.

[ST+99] Jayavel Shanmugasundaram, Kristin Tufte, et al. Relational databases for querying XML documents: Limitations and opportunities. In Atkinson et al. [AO+99], pages 302–314.

[Sta01a] Kimbro Staken. Introduction to dbXML. *XML.com*, Nov 2001. `http://www.xml.com/pub/a/2001/11/28/dbxml.html`.

[Sta01b] Kimbro Staken. Introduction to native XML databases. *XML.com*, Oct 2001. `http://www.xml.com/pub/a/2001/10/31/nativexml.html`.

[SW00] Harald Schöning and Jürgen Wäsch. Tamino – an Internet database system. In Carlo Zaniolo, Peter C. Lockemann, et al., editors, *EDBT 2000*, volume 1777 of *LNCS*, pages 383–387. Springer, 2000.

[SW+01] Albrecht Schmidt, Florian Waas, et al. Why and how to benchmark XML databases. *SIGMOD Record*, 30(3):27–32, 2001.

[SW+02] A. R. Schmidt, F. Waas, et al. XMark: A benchmark for XML data management. In Bernstein et al. [BI+02], pages 974 – 985.

[TH02] Pankaj Tolani and Jayant R. Haritsa. XGRIND: A query-friendly XML compressor. In ICDE2002 [ICD02], pages 225–234.

[TL76] D. Tsichritzis and F. H. Lochovsky. Hierarchical data-base management: A survey. *ACM Computing Surveys*, 8(1):105–123, 1976.

[W3C98] W3C. *The Query Languages Workshop*, Boston, MA, USA, Dec 1998. `http://www.w3.org/TandS/QL/QL98/`.

[WBX01] WAP Forum, Ltd. *Binary XML Content Format Specification*, version 1.3 edition, Jul 2001. `http://www.wapforum.org`.

[Wel84] T. A. Welch. A technique for high performance data compression. *IEEE Computer*, 17(6):8–20, Jun 1984.

[Wil75] Robin J. Wilson. *Introduction to Graph Theory*. Longman, 1975. First published by Oliver & Boyd, 1972.

[XLi01] World Wide Web Consortium. *XML Linking Language (XLink) Version 1.0*, W3C recommendation 27 June 2001 edition, 2001. `http://www.w3.org/TR/2001/REC-xlink-20010627/`.

[XML00] World Wide Web Consortium. *Extensible Markup Language (XML) 1.0 (Second Edition)*, W3C recommendation 06 October 2000 edition, 2000. `http://www.w3.org/TR/2000/REC-xml-20001006`.

[XML01] World Wide Web Consortium. *XML Information Set*, W3C recommendation 24 October 2001 edition, 2001. `http://www.w3.org/TR/2001/REC-xml-infoset-20011024`.

[XPa99] World Wide Web Consortium. *XML Path Language (XPath) Version 1.0*, W3C recommendation 16 November 1999 edition, 1999. `http://www.w3.org/TR/1999/REC-xpath-19991116`.

[XPa03] World Wide Web Consortium. *XML Path Language (XPath) Version 2.0*, W3C working draft 12 November 2003 edition, 2003. `http://www.w3.org/TR/2003/WD-xpath20-20031112/`.

[XQu03] World Wide Web Consortium. *XQuery 1.0: An XML Query Language*, W3C working draft 12 November 2003 edition, 2003. `http://www.w3.org/TR/2003/WD-xquery-20031112/`.

[XSL99] World Wide Web Consortium. *XSL Transformations (XSLT) Version 1.0*, W3C recommendation 16 November 1999 edition, 1999. `http://www.w3.org/TR/1999/REC-xslt-19991116`.

[ZA+03] Pavel Zezula, Giuseppe Amato, et al. Tree signatures for XML querying and navigation. In *XSym 2003*, number 2824 in LNCS, pages 149–163, Berlin, Germany, Sep 2003.

[Zlo75] Moshé M. Zloof. Query by example. In *AFIPS NCC*, volume 44, pages 431–438, Anaheim, CA, USA, May 1975. AFIPS.

[ZN+01] Chun Zhang, Jeffrey F. Naughton, et al. On supporting containment queries in relational database management systems. In *SIGMOD 2001*, 2001.

APPENDIX

A. TYPEX: A TYPE BASED APPROACH TO XML STREAM QUERYING

The following paper was published in the proceedings of the *Sixth WebDB Workshop* held in conjunction with the 2003 ACM SIGMOD Conference in San Diego. It describes a query method based on application semantics, in this case provided by the type declaration. The paper is printed in its original format on the following pages.

TypEx: A Type Based Approach to XML Stream Querying

George Russell
Department of Computer and Information Science
University of Strathclyde
Glasgow, U.K.
george@cis.strath.ac.uk

Mathias Neumüller
Department of Computer and Information Science
University of Strathclyde
Glasgow, U.K.
mathias@cis.strath.ac.uk

Richard Connor
Department of Computer and Information Science
University of Strathclyde
Glasgow, U.K.
richard@cis.strath.ac.uk

ABSTRACT

We consider the topic of query evaluation over semistructured information streams, and XML data streams in particular. Streaming evaluation methods are necessarily event-driven, which is in tension with high-level query models; in general, the more expressive the query language, the harder it is to translate queries into an event-based implementation with finite resource bounds.

We consider an alternative model by introducing a two-phase evaluation strategy. A query Q is decomposed into an event driven primary filter query Q', which incrementally gathers relevant data from the input stream, and another query Q'' which consumes this data as it becomes available. Evaluation of $Q(s)$ is then equivalent to $Q''(Q'(s))$. The importance of the separation is that it allows the first phase Q' to be expressed in a non-Turing complete algebra which may therefore be generally amenable to event-based interpretation. The second phase Q'' may be expressed in an arbitrary higher-order language, so long as its execution takes no longer than the extraction of the next input instance from the input stream.

In this paper a type algebra is used to express the first-phase query. This builds on previous work, which shows how traditional programming language types may be given a semantics within XML, therefore allowing their projection onto XML resources. A side-effect of this definition of projection is that instances of a type may be extracted in a form available for computation within a traditional domain. The use of type projection in this context also allows Q'' to be statically typed according to the type filter used for Q', which itself may be deduced by inference over Q''. A mechanism for translating a type into a network of event driven automata, which has the effect of gathering all data captured by that type from a semistructured input stream, is described. Although at an early stage of investigation, initial results suggest this approach provides a credible alternative to stream-based querying in at least some application domains.

Copyright is held by the author/owner.
International Workshop on the Web and Databases (WebDB).
June 12-13, 2003, San Diego, California.

Categories and Subject Descriptors

H.2.4 [Database Management]: Languages—*Query Languages*; D.3.3 [Programming Languages]: Language Constructs and Features—*Data Types and Structures*

Keywords

Type projection, stream processing, query typing, language integration, semistructured data

1. INTRODUCTION

The requirement for efficient, stream-based XML querying is by now established. There are several applications of such systems, notably in *selective dissemination of information* (SDI) and publish / subscribe systems [1] and for very large data environments and data integration [8]. Another motivation is the efficient transformation of streaming XML data, in applications such as XSLT processing and continuous data streams [9]. Such applications require processing to occur in real time as data becomes available, rather than after the end of the input stream is reached as in most XML query models.

SAX[1] provides a fully general XML stream-processing abstraction, giving the programmer an event-based interface based on callbacks. This interface provides full real-time streaming functionality; however the nature of the interface makes programming challenging for many applications, as conversion from the event stream to the logical structure of the underlying data is entirely the burden of the programmer.

Given this difficulty, a number of authors have investigated translation from a higher-level expressive form into an event-based model, expressed most commonly as a deterministic finite state automata (DFA) network [2, 7, 9, 11, 12]. Partial translations from both XPath and XQuery have been discussed, and while there are still open issues, significant progress has been made. One of the major outstanding issues however is the identification of the level of expression which can be sensibly handled in this model. Both XPath and XQuery contain expressions which can not usefully be translated into a DFA model without compromising computational thresholds underlying the purpose of the transla-

[1] `http://www.saxproject.org`

tions; the same of course is true for any single query language with sufficient expressive power to handle any reasonable range of queries.

We propose a significant departure from this methodology which we show works well for certain classes of application. The query is coded as a function in a Turing-complete programming language, with the assumption that the execution of this function will occur within acceptable bounds if its input can be isolated and passed to it as a process separated from the parsing of the input stream. This function could represent the query in its entirety, but in this domain the whole query is more likely to be represented by the repeated application of the function to instances of its input as they are extracted from the XML stream.

The function is typed according to its input, and this type is used to generate a deterministic state automata based input filter which extracts corresponding values from the XML input stream. These values are then passed, as they occur, to the query. In this way fully general queries can proceed in parallel with the parsing of the input. This builds on our previous work [5, 14, 15, 6, 10], in which we show how traditional programming language types may be given a semantics within XML, therefore allowing their extraction from XML resources.

The method will only work well if the query function is short-lived, and its type represents a significantly small subset of the XML stream; however we believe there exist many instances of this pattern. The main properties of the method are as follows:

- only the type-based extraction requires to be translated into the event-based paradigm
- only the second part of the query requires to be expressed, as the filter can be automatically generated from it
- the two phases of the execution can proceed in parallel

Compared with single-phase deterministic automata translations from XPath or XQuery, the method seems likely to work relatively well for complex queries over core regular data within a loosely structured stream, while it is likely to work relatively badly for simple queries over data that is inherently unstructured. The tradeoffs with respect to simplicity of expression and efficiency are complex and require further investigation, but in this paper we have at least shown credibility in these domains with respect to some classes of application.

In more generality, not considered further in this paper, the same basic two-part query framework may be used entirely within the XML standards domain, by expressing the filter by a projection defined over XML Schema. Instances of XML would then be generated by the first phase, allowing the secondary part to be expressed in any XML query language. One particularly interesting aspect of this is that, if the entire data stream is known to be valid with respect to a given schema, then the soundness of the filter may be statically assessed, and furthermore user-level tools for generating it from the XML Schema stream description could be envisaged.

2. A MOTIVATING EXAMPLE

For motivation, an example streamed application is coded in Java using various alternative implementation techniques, namely SAX, XPath, and TypEx, the system based on the approach described. SAX only creates temporary data structures to report parsing events which need to be transferred manually into the application specific data model for processing. The XPath code uses an XPath expression to define a superset of the required data, in conjunction with DOM code to perform the required query; the TypEx code uses a Java class definition to define a superset of the required data, in conjunction with a method of that class to query it. In the cases of XPath and TypEx the initial extraction may be performed incrementally and processed in parallel with the code that uses the extracted values, whereas in the SAX application the entire processing must be performed within the parsing callbacks.

The example is a news ticker application, which extracts news items from an arbitrary XML stream. Schema information of the expected stream is incomplete and restricted to the relevant data. The application programmer knows that there are *item* elements which contain at least two direct child elements named *title* and *description*. Both these items contain only textual content and can occur only once within any *item*. This data may be embedded at any point in the source and additional content may appear beneath *item* elements. The application is to display the content of these two fields as they are detected within an unbounded stream of XML.

2.1 Parsing Event Based Model

An example of an approach in which the query is directly contained in the application code is the event-based abstraction model used by *SAX*. Mappings between parsing events and the data model used are spread over various callback methods, making it hard to understand and thus maintain. The mixture of selection and computation also increases the coupling between user-specified computation and parsing process and is thus undesirable.

Listing 1: The SAX handler for the news ticker

```
public class NewsHandler extends DefaultHandler {
  private StringBuffer _title , _description ;
  private Stack elements = new Stack();
  public void characters(char[] ch,
      int start , int length) {
    if (elements.search("item") == 2) {
        if (elements.search(" title ") == 1)
          _title .append(ch, start, length);
        if (elements.search("description") == 1)
          _description .append(ch, start, length);
    }
  }
  public void startElement(String namespaceURI,
      String localName, String qName, Attributes atts) {
    elements.push(qName);
    if (qName.equals("item")) {
```

```
        _description = new StringBuffer();
        _title = new StringBuffer();
      }
    }
    public void endElement(String namespaceURI,
        String localName, String qName) {
      elements.pop();
      if (qName.equals("item")
        System.out.println( _title +"\n"+_description);
    }
    public static void main(String[] args) {
      SAXParser parser =
        SAXParserFactory.newInstance().newSAXParser();
      parser.parse(new URL(args[0]).openStream(),
        new NewsHandler());
    }
}
```

Listing 1 shows that the required state information is maintained by using a stack containing all opened tags (line 15). String buffers are created once an opening *item* tag is found. Upon occurrence of character data the top two elements of the stack are checked. If they fit the structural constraints, the content is appended to the relevant buffers. The content of these buffers is printed when the end of a news item is detected. Note that this implementation does not verify order, multiplicity or even existence of required fields, but just checks that identified fields suffice the structural constraints. Other constraints would need to be checked using either a more complicated handler or a partial schema validation process, which is currently not part of any of the relevant standards.

2.2 XPath/DOM Based Model

The combination of *XPath* [16] and *DOM* allows a different approach. Programs specify the desired set of nodes using a path expression, and operate over the results using tree traversal code. XPath expressions are navigation expressions over tree structured data which are sent to an execution engine in a similar fashion as embedded SQL statements. Selection is mechanically performed by the system and returns a collection of trees, requiring tree traversal code in the user specified computation. Typically a superset of the data required is returned because XPath returns the entire subtrees rooted at the selected nodes, regardless of the data requirements of the subsequent computation. However, as there is no strong coupling between the two phases, it cannot be guaranteed that the returned data actually satisfies the computational needs.

Listing 2: The same application using XPath

```
public class BBCNewsXPath {
  public static void main(String[] args){
    DocumentBuilderFactory factory =
      DocumentBuilderFactory.newInstance();
    DocumentBuilder builder =
      factory.newDocumentBuilder();
    Node doc = builder.parse(
      new InputSource( new URL (
        args [0]). openStream()));
    NodeList results =
      XPathAPI.selectNodeList(doc,
        "//item[description and child::text()]"
        +"[title and child::text()]"
        +"[count(title)=1]"
        +"[count(description)=1]");
    for (int i=0; i<results.getLength(); i++) {
      Node result = results.item(i);
      Element altem = (Element) result;
      NodeList titles = altem
        .getElementsByTagName("title");
      String title =
        ((Element) titles.item(0))
        .getFirstChild().getNodeValue();
      NodeList descriptions = altem
        .getElementsByTagName("description");
      String desc =
        ((Element) descriptions.item(0))
        .getFirstChild().getNodeValue();
      System.out.println( title + "\n" + desc);
    }
  }
}
```

The program shown in Listing 2 uses the single XPath statement stretching from lines 12–15 to declare the navigational steps required to select the relevant data. The XPath execution engine returns a NodeList containing the selected nodes (line 10). The loop starting in line 16 iterates through this list and extracts the data from the relevant text nodes using the DOM API, and in particular the method *getElementsByTagName* which selects nodes based on their name. Structural checks upon the input format have been added to the XPath query in Listing 2, which allows them to be omitted from the result processing code which otherwise would contain explicit structural checking. The XPath implementation used is not streaming, but this does not affect the purpose of the example.

2.3 Type Projection Based Model

The approach suggested by this paper has been implemented in the *TypEx* system. The extraction phase identifies data that may be relevant for the query by means of a filter type and binds this data to an instance of this type for use in the second phase. Since the computation is specified in terms of the filter type, the returned instances of this type will always contain a superset of the information required.

Listing 3: The filter class *item*

```
public class item {
  String title , description;
  public String toString() {
    return title+"\n"+description;
  }
}
```

In our example application, the class used to store and print news items is also used as the filter type (Listing 3). This defines the data extraction in terms of the host programming language and acts as a data model for the following computation, ensuring a match between the two phases and a seamless integration with the language. The actual com-

Listing 4: The TypEx Newsticker

```
public class BBCNews implements Observer {
  Extractor stories;
  public BBCNews() {
    stories = new Extractor(item.class);
    stories.addObserver(this);
  }
  public void parse(InputStream in) {
    stories.parse(in);
  }
  public void update(Observable o, Object i) {
    System.out.println((item) i);
  }
  public static void main(String[] args){
    BBCNews p = new BBCNews();
    p.parse(new URL(args[0]).openStream());
  }
}
```

putation, i.e. the printing of the content is also defined by this type. Listing 4 shows the usage of this filter. It creates an **Extractor** parameterised by the type (line 4), which extracts **item** objects during parsing and passes them to the observer for display.

3. TYPE BASED EXTRACTION

In this section we outline the translation of a core type language into a network of co-operating, deterministic automata. The resulting network is capable of extracting values of those types.

The type language includes both record and list constructors, but does not allow anonymous lists to occur as they have no semantic projection in XML. In addition, lists cannot occur at the top level, as this would be incongruous with the purpose of the mechanism, which is to release typed data as soon as possible. A grammar for the type language is given below.

typedef ::= label : *type*
type ::= *record* | *scalar*
record ::= {label$_1$: *field*$_1$, ..., label$_n$: *field*$_n$}
field ::= *scalar* | *record* | **list** [*type*]
scalar ::= **int** | **string**

A type expression is translated into a set of named component types as shown by the following example:

p: {a: **int**, b: **list** [{d: **int**}], c: {d: **int**}}

is transformed into

p : T_1
T_1: {a: T_2, b: list[T_3], c: T_3}
T_2: **int**
T_3: {d: T_2}

Each type T_i in this representation is mapped to an automaton, whose job is to extract an instance of that type from the input stream and return it as a value. The topology produced by this example is shown in Figure 1. The size of the automata network equals the size of the corresponding type graph plus one for the additional **sink** machine.

The **root** and **sink** machines always occur as single instances; the others are generated according to the individual type components. Each internal connection is a two-way link, passing input events in one direction and results in the other. Only one machine at a time actively processes events; a machine become active when it receives an *init()* event, and remains so until either a *return()* or *fail()* event is passed back to its initiator. Non-active machines pass events on to the currently active machine. Events are propagated synchronously in that they are not passed to the active machine until the previous event has been processed; this avoids synchronicity problems with the return events being passed back up the chain.

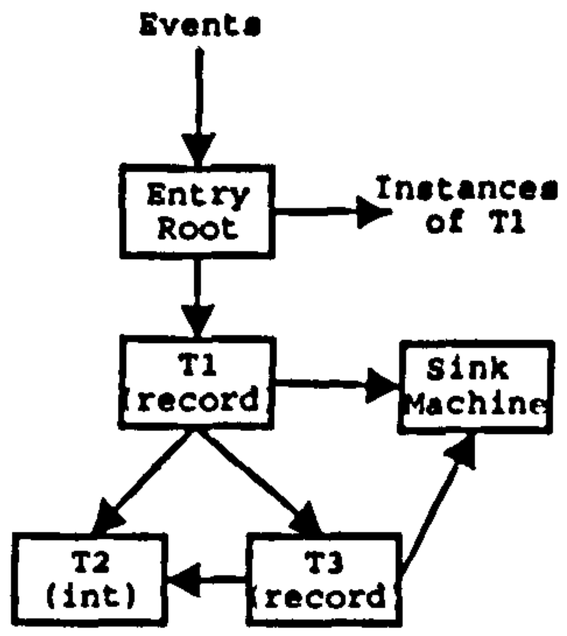


Figure 1: The example automata network

The following messages are defined: *open(l)*, *close(l)*, *text(v)*, *init(l)*, *return(v)*, and *fail()*. A parameter l stands for a label corresponding to an XML tag label, and a parameter v stands for a value. *open(l)*, *close(l)* and *text(v)* are events corresponding to a simplified input stream, and correspond to the textual form of the XML being processed. *init(l)*, as described, causes a machine to become active; its parameter is a label which, when subsequently received within a *close(l)* message, will cause it to pass control back to its initiator. This is either by means of a *return(v)* message, in the case it has been able to extract a value corresponding to its type from the input stream, and by means of a *fail()* message otherwise. There are four different classes of machines: **entry**, **sink**, **scalar** and **record**, as defined by the following behaviours:

- **Entry**: discard every event until an *open(l)* occurs, where l is the label corresponding to the name of the top-level *typedef*. At this point, *init(l)* is passed on to the machine corresponding to the type of the top-level *typedef*, which then becomes the active machine. Further events are passed to this machine until either a *return(v)* or *fail()* is received from it. On a *return(v)*, the parameter v is passed on to the network's receiver; on *fail()*, no further action occurs.

- **Sink**: the purpose of this machine is to discard an XML subtree, which cannot be of interest to the extraction. On receipt of *init(l)*, it becomes the active machine: its only required function is to discard events

until the corresponding *close(l)* event is received, ensuring that any contained subtrees using the same label l are also discarded. No value is returned.

- **Scalar:** on receipt of *init(l)*, a scalar machine requires the first event it receives to be a *text(v)* message, and its second to be a *close(l)* message. Depending on the particular scalar type, the string parameter v is examined to ensure it is structurally compatible, and if so is coerced and returned. Any other combination of events causes the machine to pass a *fail()* event back to its initiator.

- **Record:** a record machine starts with internal state variables corresponding to each of the field names occurring in its progenitor type description. For each one, the value is initialised to *undefined* if the field is a scalar or record type, and to an empty list if it is a list type. These variables are used to build up the state corresponding to the record value it attempts to extract. It may receive *text(v)*, *open(l)* and *close(l)* events from its initiator:

  - *text(v)*: the event is discarded
  - *open(l)*: the behaviour depends on whether there is an internal state variable corresponding to l, and if so what its value is. If there is no internal variable, an *init(l)* message is sent to the sink machine. If there is a variable which is a list, an *init(l)* message is passed to the machine corresponding to the field type; if this returns with a value, it is appended to the list, otherwise no action is taken. If there is a variable whose value is defined, this results in termination with *fail()*. Finally, if there is a variable whose value is not yet defined, an *init(l)* message is passed to the machine corresponding to the field type; if this returns with a result, it is assigned to the internal variable, otherwise no action is taken.
    This behaviour corresponds to a structural subtyping approach to extracting the corresponding data values in the XML whilst ignoring order fields and any extra fields that are not pertinent to the query.
  - *close(l)*: if all internal state fields are defined, a record value based on them is constructed and returned to the calling machine in a *return(v)* message; otherwise, a *fail()* message is passed back.

4. IMPLEMENTATION

The automata networks within *TypEx* form the middle layer of the system architecture. Below this layer an event handler translates parsing events generated by an underlying XML parser into automata events. Above the automata layer is the transformation layer, which generates networks from filter specifications and transforms extracted data graphs into instances of the specified type. The top layer is the programmer visible API, which allows the specification of an input source and associated filter types. It allows multiple listeners per filter and multiple filters per data stream, affording a degree of parallelism in the query process.

Each automaton is implemented as a Java class. Automata networks are generated using reflection to examine the field types and names of filter types. Reflection is also used during data instantiation process.

5. EXPERIMENTAL RESULTS

To determine the viability of our approach, we have processed a more complex example query than the one discussed in Section 2. The data set under consideration has been generated using the XMark benchmark [13] scalable data generation tool and describes auction site details containing items for sale, persons (bidders and sellers) and some more information not relevant for our purposes. In particular, we have queried large XML files (up to 11GB in size) for people (with attributes such as name, address, etc.). We compare programs based on SAX, DOM XPath and TypEx both in terms of the complexity of the code and their runtime performance. The number of lines have been taken as a simple measure of the complexity of the code (Table 1).

System	Lines of Code		Comment
	Select	Extract	
SAX	150		Selection and extraction
DOM	43		cannot be separated.
XPath	1	39	Uses DOM extraction code.
TypEx	6	17	

Table 1: Length of Query

System	1 MB	10 MB	15 MB	30 MB	100 MB
DOM	1.73	147.0	326.0	1296	N/A
XPath	0.38	4.6	7.7	45	N/A
SAX	0.19	1.1	1.5	3	13
TypEx	0.43	3.6	5.6	10	21

Table 2: Length of Query Execution (s)

Table 1 supports the observation gained from the news ticker example and illustrates the conciseness possible using the TypEx approach. The SAX program, contains a mix of high-level application code and low-level parsing callbacks and results in an order of magnitude increase of code. DOM and XPath approaches lie between these two extremes and are almost identical because of their common tree traversal code.

The SAX query execution time scales linearly with the size of the input data and its memory usage depends only the level of element nesting. DOM is unsuitable for streaming due to its inherent whole document approach. As the input size increases, the time taken to complete the query increases more than linearly, while the memory usage increases linearly. With an 100 MB source file, the query fails to complete due to an *OutOfMemory* error using a heap size of 512 MB. We were unable to obtain a complete implementation of XPath to operate upon streaming data. The implementation we used for this experiment, Xalan/J, is backed by a incrementally built tree structure, and thus scales similarly to a DOM implementation in space. It does not however, provide results in an incremental fashion and thus does not allow a direct comparison with either SAX or TypEx. TypEx, while slower to execute than the equivalent SAX program also scales linearly with the size of the input data. The memory requirements are determined by the size of the extracted type, i.e. are independent of the size of the document. We have used it to successfully query inputs of up to 11 GB in size.

6. RELATED WORK

To the best of our knowledge, this is the first attempt of using types as filters in a two stage stream query process. The requirements for stream processing have been identified in [3] and elsewhere.

Much of the effort concentrates on implementing the XPath [16] language over streams. A number of independent efforts are proceeding, such as XMLTK [2], SPEX [11], and XSQ [12]. Green et al. [7] describe the implementation of an XPath subset using a lazily generated network of DFA and provide a comparison with other XPath implementations over streams. The work concentrates on the construction of a single automata network representing a large number of XPath queries which are executed simultaneously. Olteanu et al. [11] describe the translation of a subset of XPath expression into equivalent expression using only forward axis, which can then be efficiently processed by their SPEX XPath system using a network of event driven automata. Finally the XSQ system [12] is based upon push-down transducers with associated buffers and aims at complete XPath support.

A more recent approach by Ludäscher et al. [9] describes the implementation of the computationally complete query language XQuery over streaming data, using transducer networks derived from a query expression. Their XSM system optimises the derived network based on static analysis and available schema information. Optimisation strategies used may prove useful for our approach.

Type projection is introduced in [5] and formalised in [15]. Language integration with Java is described in [14] in more depth, while its use within query languages for XML is also outlined [10, 6].

7. CONCLUSIONS

We have outlined a novel mechanism which allows XML stream queries to be decomposed into an extraction and a computation phase over the extracted data based on a single type description. While the concept requires further investigation, it has a number of advantages for some classes of application. In particular, it may allow the programmer to use a higher-level, and therefore more succinct, query without gravely affecting properties of efficiency. Queries are formulated within the domain of the host language and can thus be statically typed.

The mechanisms have been implemented and first results show some promise. However the investigation is at an early stage and a great deal of work remains to be done.

8. ACKNOWLEDGEMENTS

We would like to thank Fabio Simeoni, David Lievens, Steve Neely and the anonymous referees for making useful suggestions on the presentation of this work. It has been financially supported by EPSRC (GR/M72265) and BBSRC (17/BIO12052.) George Russell and Mathias Neumüller are supported by PhD studentships funded by the University of Strathclyde.

9. REFERENCES

[1] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In A. E. Abbadi, et al., editors, *VLDB 2000*, pages 53–64, 2000. Morgan Kaufmann.

[2] I. Avila-Campillo, T. J. Green, et al. XMLTK: An XML toolkit for scalable XML stream processing. In *PLAN-X: Programming Language Technologies for XML*, 2002.

[3] B. Babcock, S. Babu, and other. Models and issues in data stream systems. In L. Popa, editor, *PODS 2002*, pages 1–16, Madison, Wisconsin, USA, 2002. ACM.

[4] P. A. Bernstein, Y. E. Ioannidis, et al., editors. *Proceedings of the 28th International Conference on Very Large Databases*, Hong Kong, China, 2002. Morgan Kaufmann.

[5] R. Connor, D. Lievens, et al. Extracting typed values from XML data. In *OOPSLA Workshop on Objects, XML and Databases*, 2001.

[6] R. Connor, D. Lievens, et al. Projector – a partially typed language for querying XML. In *PLAN-X: Programming Language Technologies for XML*, 2002.

[7] T. J. Green, G. Mikklau, et al. Processing XML streams with deterministic automata. In D. Calvanese et al., editors, *ICDT 2003*, volume 2572 of *LNCS*, pages 173–189. Springer, 2002.

[8] T. Kiesling. Towards a streamed XPath evaluation. Diplomarbeit, Universität München, 2002.

[9] B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou. A transducer-based XML query processor. In Bernstein et al. [4].

[10] P. Manghi, F. Simeoni, et al. Hybrid applications over XML: Integrating the procedural and declarative approaches. In *Fourth International Workshop on Web Information and Data Management (WIDM'02)*, Virginia, USA, Nov 2002.

[11] D. Olteanu, T. Kiesling, and F. Bry. An evaluation of regular path expressions with qualifiers against XML streams. In *ICDE 2003*, Mar 2003.

[12] F. Peng and S. S. Chawathe. XPath queries on streaming data. In *SIGMOD 2003*, 2003.

[13] A. R. Schmidt, F. Waas, et al. XMark: A benchmark for XML data management. In Bernstein et al. [4], pages 974 – 985.

[14] F. Simeoni, D. Lievens, et al. Language bindings to XML. *IEEE Internet Computing*, 7(1), Jan/Feb 2003.

[15] F. Simeoni, P. Manghi, et al. An approach to high-level language bindings to XML. *Information & Software Technology*, 44(4):217–228, 2002.

[16] World Wide Web Consortium. *XML Path Language (XPath) Version 1.0*, W3C recommendation 16 November 1999 edition, 1999. `http://www.w3.org/TR/1999/REC-xpath-19991116`.

B. STRUCTURAL INDICES BASED ON BISIMILARITY

This appendix reproduces important results and some of the algorithms used by Kaushik et al. in their work on structural indices based on bisimilarity. It provides an overview of their definition of bisimilarity, the algorithms used to compute the resulting partition of the graph's vertex set and to generate the corresponding index graph, which is essential for understanding the design of the experimental system described in Chapter 6 and some of the examples being based on their notion of bisimilarity.

The work addressed in this appendix was developed over an extensive period of time and spans several publications. Their key contributions are summarised in the following sections. Details such as proofs or experimental results can be found in the original articles.

B.1 Exploiting Local Similarity for Indexing Paths in Graph-Structured Data

The initial work [KS+02] describes the fundamental concept of local similarity, which is used to reduce the complexity of index structures employed for answering regular, linear path expressions without value predicates over generally graph-structured data instances. This represents a class of queries that is covered by the 1-Index discussed by Milo and Suciu [MS99] and lead to the design of the DataGuide [GW97]. Such indices combine vertices of a data graph in a single vertex of an index graph if they form the result to a given query. Such a data grouping is provided by the *simulation* of the data graph, i.e. by finding vertex-sets that are *similar*. However, the exact partition can become very complex and is expensive to compute. The complexity is a result of the long and complex paths occurring in graph-structured sources that contribute disproportionately to the complexity of a covering index. Such paths are rarely the subjects of queries.

To get rid off these unwanted long paths in the structural summary, one has to restrict the accuracy of the index. The way chosen by Kaushik et al. is to reduce the index coverage to path expressions of limited lengths. The expense of computing the grouping can be reduced by replacing the ideal grouping with a *refinement* of it, i.e. by finding a grouping that is easier to compute, but never mixes members of the ideal grouping though it can split individual blocks of it even further. *Bisimulation* is such a refinement, which was already discussed by Milo and Suciu [MS99]. Kaushik et al. used it to define their A(k)-index, which is based on local backward bisimilarity of degree k. This means that each vertex of a data graph that belongs to a common bisimilarity class, has the same set of incoming paths up to length k. The following definition is taken directly from [KS+02].

Definition B.1 ($\approx^k$ [k-bisimilarity]): This is defined inductively.

1. For any two nodes[1], u and v, $u \approx^0 v$ iff u and v have the same label.

2. Node $u \approx^k v$ if $u \approx^{k-1} v$ and for every parent u' of u, there is a parent v' of v such that $u' \approx^{k-1} v'$ and vice versa.

The bisimilarity relationship defines an equivalence relationship on the node-set of a data graph, which is called *k-bisimulation*. From the definition the following properties can be derived, which essentially show that the index based on bisimilarity is covering for linear path expressions of length k. Again, the list of properties is taken directly from the original work.

Property B.1: (a) If node u and v are k-bisimilar, then the set of label-paths of length $\leq k$ into them is the same.

(b) The set of label-paths of length k into an A(k)-index node is the set of label-paths of length k into any node of its extend.

(c) The A(k)-index is *precise*[2] for any simple path expression of length less than or equal to k.

(d) The A(k)-index is *safe*, i.e., its result on a path expression always contains the graph result for that query.

---

[1] Kaushik et al use the term "node" instead of "vertex" as used in this thesis

[2] "Covering" in terms of this thesis.

```
procedure compute_k_bisim(G,k)
begin
1.  Q and X are each a list of node-sets
2.  Q = partition V_G by label
3.  X = (a copy of) Q
4.  for i = 1 to k do
5.    foreach X in X do  //stabilize Q w.r.t X
6.      compute Succ(X)
7.      foreach Q in Q do // split
8.        replace Q by Q ∩ Succ(X) and Q − Succ(X)
9.    if there was no split then
10.     break
11.   X = (a copy of) Q
end
```

```
procedure compute_A(k)_index(G, k)
begin
1.  compute_k_bisim(G,k)
2.  foreach equiv. class in k-bisimulation do
3.    create an index node I
4.    ext[I] = data nodes in the equiv. class
5.  foreach edge from u to v in G do
6.    I[u] = index node containing u
7.    I[v] = index node containing v
8.    if there is no edge from I[u] to I[v] then
9.      add an edge from I[u] to I[v]
end
```

Fig. B.1: A(k)-index computation (taken from [KS+02])

(e) The $(k+1)$-bisimulation is either equal to or is a refinement of the k-bisimulation.

The inductive definition of k-bisimilarity also gives rise to the algorithm that is used to compute it. The algorithm starts by creating a partition of the vertex-set of the graph according to their tag-labels. This partition is consecutively refined by splitting blocks of the partition according to increasingly longer common incoming path sets. It is based on a partitioning algorithm by Paige and Tarjan [PT87]. A node-set (or block of a partition) A is said to be *stable* with respect to another node-set B, if A is a subset of $Succ(B)$ or A and $Succ(B)$ are disjoint, where $Succ(B)$ is the set of nodes with an incoming arc from B. A partition P_1 of V is stable with respect to a partition P_2 of V if each block in P_1 is stable with respect to each block in P_2. If there are two nodes-sets A and B and one wished to make A stable with respect to B, one needs to split A into $A \cap Succ(B)$ and $A - Succ(B)$. Figure B.1 shows the resulting algorithms used to compute the k-bisimulation and the resulting A(k)-index graph. The algorithms runs in $\mathcal{O}(k|A|)$ time.

B.2 Covering Indexes for Branching Path Queries

This subsequent publication by Kaushik et al. [KB+02] extends the concept of local bisimilarity to outgoing paths in order to address a larger class of queries. The new query class is described by branching path expressions without value predicates and coincides with the definition of the query language presented in Section 2.3.1.

Definition B.1 is extended to make explicit the distinction between the edges of the spanning tree and other graph edges[3].

Definition B.2 ($\approx^k$ [k-bisimilarity]): This is defined inductively.

1. For any two nodes, u and v, $u \approx^0 v$ iff u and v have the same label.

2. Node $u \approx^k v$ if $u \approx^{k-1} v$, $par_u \approx^{k-1} par_v$ where par_u and par_v are respectively the parents of u and v, and for every v' that points to v through an idref edge, there is a v' that points to v to an idref edge such that $u' \approx^{k-1} v'$ and vice versa.

The index graph based on this refined definition of bisimilarity for branching path expressions with unbounded path lengths can be computed using the algorithm presented in Figure B.1. However, it must be applied repeatedly to the data graph G, with all edges reversed for every second run as explained in [KB+02] and shown in Figure B.2.

1. Reverse all edges in G.

2. Compute the bisimilarity partition (with the current partition as the initialization).

3. Set the current partition to what is output by the previous step.

4. Reverse edges in G again, obtaining the original G.

5. Compute the bisimilarity partition (again initializing the computation with the current partition).

6. Set the current partition to what is output by the previous step.

7. Repeat the above steps till the current partition does not change.

Fig. B.2: Algorithm for the computation of the F&B-index

The resulting index is called the F&B-index, which is the minimal covering index for all structural branching path expressions. Experimental results [KB+02] showed that the size of this index approaches the size of the original data graph for sources like the XMark dataset described in Section C.3 or the Open Directory Project[4]. Apart from restricting the length of the bisimilarity relation to lengths

[3] Called "idref edges" by Kaushik et al.

[4] `http://www.dmoz.org`

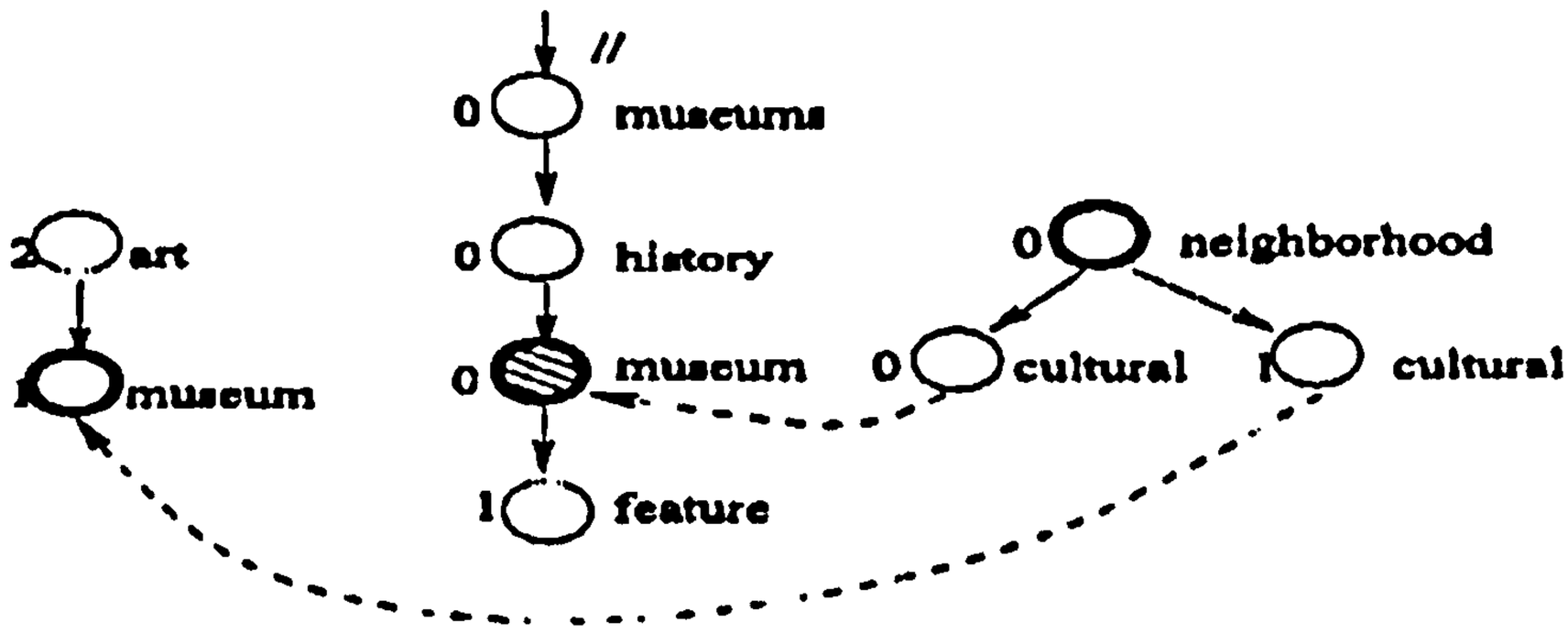


Fig. B.3: Example for tree-depth (taken from [KB+02])

k_{fwd} and k_{back} in forward and backward direction, it is also possible to restrict the number of iterations performed on the sequence of operations presented above, i.e. to replace step 7 by a fixed number of repetitions. This leads to the definition of the *tree-depth* [KB+02] of a query predicate.

> "The idea is that all nodes on the primary path have tree-depth 0. All nodes that do not have tree-depth 0 and have a path from some node in the primary path have tree-depth 1, nodes that do not have tree-depth 1 and have a path to some node of tree-depth 1 have tree-depth 2, nodes that do not have tree-depth 2 and have a path from some node of tree-depth 2 have tree-depth 3, and so on. Intuitively, odd tree-depths correspond to out-going path conditions, while even tree-depths correspond to in-coming path conditions."

The concept of tree-depth is illustrated using Figure B.3, which was taken from [KB+02] and illustrates Query B.1.

Query B.1: `//museums/history/museum[/featured&⇐cultural\neighborhood [/cultural⇒museum[\art]]]`

As becomes obvious from this query, whose tree-depth is only 2, tree-depth in practice will be of very limited size. In fact only a single iteration of the above procedure will suffice for most queries. The resulting index graph is called the F+B-index by Kaushik and was already introduced in Example 4.7.

The algorithm used to compute the partition of the vertex-set parameterised by the forward and backward bisimilarity k_{fwd} and k_{back} and the tree-depth td

procedure compute_partition(G,S)
$G \rightarrow$ data graph, $S \rightarrow$ index definition
begin
1. Convert all tags in G not in T into special tag other
2. Remove any occurrence of a node labeled other if it is not on some tree path from the root to any node with a label that is to be indexed
3. Let $\mathcal{P}$ be a list of sets of nodes
   //representing a partition of the nodes of G
4. $\mathcal{P} \leftarrow$ label-grouping of G
5. **for** $i = 1$ to td **do**
   //forward direction
6. Retain *idref* edges in ref_{fwd}
7. Reverse all edges in G
8. Compute the k_{fwd}-bisimulation on G initialising the computation with $\mathcal{P}$
9. $\mathcal{P} \leftarrow$ partition of nodes of G corresponding to the above k_{fwd}-bisimulation
   //backward direction
10. Restore G
11. Retain *idref* edges in ref_{back}
12. Compute the k_{back}-bisimulation on G initialising the computation with $\mathcal{P}$
13. $\mathcal{P} \leftarrow$ partition of nodes of G corresponding to the above k_{back}-bisimulation

end

Fig. B.4: Vertex-set partition computation (taken from [KB+02])

is shown in Figure B.4 [KB+02]. The algorithm also restricts the computation to a set of tag labels and idref edges. The algorithm for the computation of the corresponding index graph is identical to the algorithms shown in Figure B.1.

C. DESCRIPTION OF DATA SOURCES

This appendix describes and characterises the data sources that were used throughout the thesis in order to obtain the experimental results presented. Table C.1 lists the sources used, together with a short, informal description and Table C.2 shows some fundamental metrics of their corresponding data graphs. More details about the individual sources are given in the subsequent sections.

C.1 The Domain Name Server Database

This source is an XML encoding of the root DNS server routing table for the academic part of the Internet in Great Britain. It contains the mapping between IP addresses and all symbolic server names ending in `ac.uk`. The individual entries contain attributes for the server name, the four parts of its numeric IP address and up to six parts of its symbolic domain name. In the case of XML (Listing C.1) only those parts of the domain name that are present are stored, whereas in the original database null values are used to represent missing entries. Nevertheless it is a very regular source, being automatically generated from a table-like data source. It exhibits the typical properties of a data-centric source, a flat but wide tree with a comparatively small tag label alphabet.

Source	File Size	Description
DNS 100k	21 MB	Domain Name Server Database for `ac.uk` in XML format
DBLP	131 MB	Computer Science Bibliography in XML format
Nasa	24 MB	Astronomical Data converted from legacy flat-file format into XML
Macbeth	160 KB	Shakespeare's play "Macbeth" converted from SGML into XML format
XMark-1	1.1 MB	Synthetic dataset of the XMark benchmark project
XMark-10	11 MB	Synthetic dataset of the XMark benchmark project

Tab. C.1: Overview over the data sources used throughout the thesis

Source	$\lvert V_A\rvert$	$\lvert V_C\rvert$	$\lvert E\rvert/\lvert A\rvert^1$	$\lvert\Sigma\rvert$	$h(T)$	ϱ_{in}	ϱ_{out}
DNS-100k	851,419	951,421	1,802,839	15	4	1	100,000
DBLP	3,410,133	3,736,407	7,146,539	42	7	1	328,858
Nasa	1,005,205	532,964	1,538,168	72	9	1	4,871
Macbeth	3,287	3,976	7,262	18	7	1	71
XMark-1	12,414	20,450	32,863	78	3	153	255
			36,024				
XMark-10	122,221	200,106	322,326	78	13	148	2,550
			353,074				

Tab. C.2: Important metrics of the example sources: Number of atomic and complex, number of edges and arcs[1], size of label alphabet, height of distinguished spanning tree, maximal fan-in and fan-out factor of any vertex.
[1] identical for tree sources

```
<?xml version="1.0" ?>
<server-LIST>
  <server>
    <HOSTNAME>web0</HOSTNAME>
    <LEVEL0>uk</LEVEL0>
    <LEVEL1>ac</LEVEL1>
    <LEVEL2>strath</LEVEL2>
    <LEVEL3>cis</LEVEL3>
    <LEVEL4>www</LEVEL4>
    <IP0>130</IP0>
    <IP1>159</IP1>
    <IP2>196</IP2>
    <IP3>115</IP3>
  </server>
</server-LIST>
```

Listing C.1: Example entry of the DNS database in XML format

```
<?xml version="1.0"?>
<!DOCTYPE PLAY SYSTEM "play.dtd">

<PLAY>
  <TITLE>The Tragedy of Macbeth</TITLE>
  ...
  <SCNDESCR>SCENE  Scotland: England.</SCNDESCR>
  <PLAYSUBT>MACBETH</PLAYSUBT>
  <ACT>
    <TITLE>ACT I</TITLE>
    <SCENE>
      <TITLE>SCENE I.  A desert place.</TITLE>
      <STAGEDIR>Thunder and lightning. Enter three Witches</STAGEDIR>
      <SPEECH>
        <SPEAKER>First Witch</SPEAKER>
        <LINE>When shall we three meet again</LINE>
        <LINE>In thunder, lightning, or in rain?</LINE>
      </SPEECH>
      ...
    </SCENE>
    ...
  </ACT>
  ...
</PLAY>
```

Listing C.2: An excerpt from Shakespeare's Macbeth

C.2 Shakespeare's Macbeth Encoded in XML

The source representing the play "Macbeth" by William Shakespeare was taken from `http://www.ibiblio.org/xml/examples/shakespeare`. It is an XML encoded version of the play's manuscript including all stage directions. John Bosak took the text of the printed version and annotated it with semantic tags, such as `speaker`, `stagedir` or `title`. Due to the fact that the original is meant for human consumption, tags such as `stagedir` can occur in various contexts, e.g. in the middle of an actors `line`, leading to various mixed content elements in the XML encoding. It is an typical example of a document-centric data source. An excerpt of this source is shown in Listing C.2.

C.3 The XMark Benchmark Dataset

These sources are synthetic databases generated by the scalable XMark benchmark source generation tool *xmlgen*. Two databases were generated, one has a data volume of 1 MB, the other one is 10 MB large. Both sources model the data content of an electronic auction site. They represent multi-faceted documents, which contain both data- and document-centric components. Data-centric aspects of the source are used to define entities of the database that would be

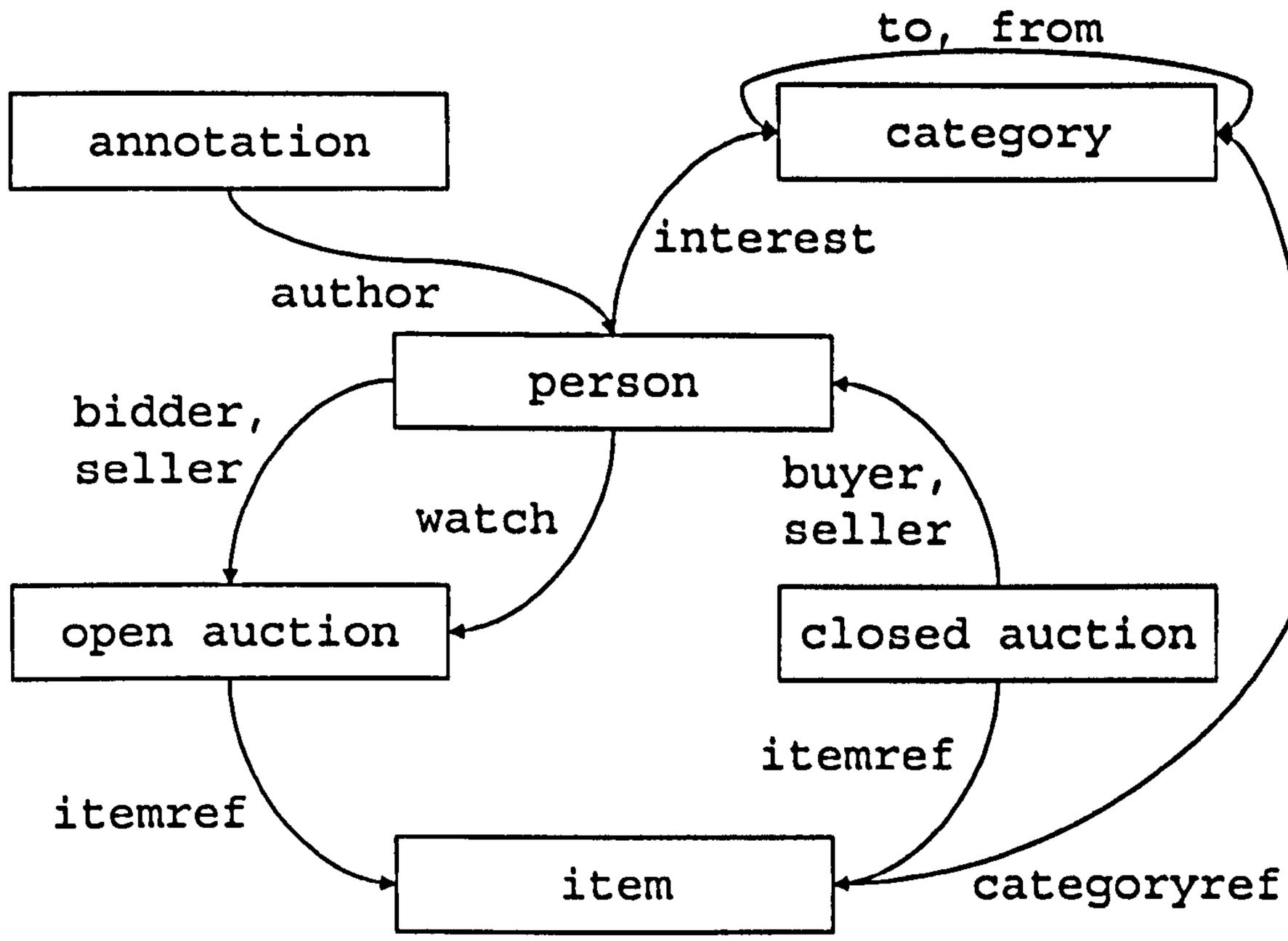


Fig. C.1: References within the XMark data source (after [SW+02])

stored in relational tables in a more traditional approach, i.e. the costs, locations and quantities of individual items, telephone numbers and address information of bidders and sellers and so on. However, the description of items or categories for example, exposes document-centric properties. It contains mixed content models, such as highlighted keywords within a description, and these are contained within a recursive structure, e.g. lists containing other lists as items. Another interesting property of these sources is the fact that they make a lot of use of XML's ID:IDREF-references, thus encoding proper graph rather than tree data structures. The sources, their schemata in the form of a DTD or even their complete set of entities are too large to be presented here. However, Figure C.1 shows the cross-references between the main entities of the source. It was taken from [SW+02] that also contains a more detailed description of the dataset.

C.4 The Nasa Astronomical Dataset

This source contains datasets converted from a legacy flat-file format into XML by the GSFC/NASA XML Project[1] and taken from the University of Washington's XML Data Repository[2]. The specific source used contains the Astronomical Data Center's public XML collection arranged by catalog category. Each dataset contains the article title, a source description, the lead author and the year of publication. Additional XLink references provide references to other parts of the project. However, these external references were not resolved by the work presented in this thesis, thus the source is parsed into a tree-shaped data structure. Because the source does not contain a document schema, the parsed structure contains a lot of unwanted atomic data vertices representing white-space within the document structure. Again this source presents a mixture of document-centric and data-centric elements.

C.5 The DBLP Bibliographic Database

The DBLP computer science bibliography of the Universität Trier provides bibliographic information on major computer science journals and proceedings. Their server indexes several hundred thousand articles and contains links to several thousand home pages of computer scientists. The snapshot of the XML version of the database used throughout this thesis was taken in January 2003, at which time it had a size of 131 MB. Since then its size has roughly doubled to 230 MB. Though the metadata of the individual publications possesses data-centric properties, their exact structure differs significantly from entry to entry. The database contains an increasing number of external links that connect the database entries with electronic versions of the articles themselves. As for the previous source, such external links were not resolved. Since the DBLP database does not make use of internal links between the entries, this source also encodes a document tree in terms of our model. An example entry of this source was already presented in Listing 2.2 of Chapter 2 of this thesis.

---

[1] http://xml.gsfc.nasa.gov/archive/index.html
[2] http://www.cs.washington.edu/research/xmldatasets/www/repository.html

D. NSGRAPH PERFORMANCE MEASUREMENTS RESULTS

This appendix lists the raw data obtained by running the different query algorithms over NSGraphs generated from the 1 MB XMark dataset with varying backward and forward bisimilarity lengths. This data is analysed in Section 6.5.3, which also list the benchmark queries.

D.1 Linear Path Patterns

			bw=0				bw=1			
Query	Algorithm	Data	fw=0	fw=1	fw=2	fw=3	fw=0	fw=1	fw=2	fw=3
K1	MJ-BU	Vertex visits	4	10	33	103	16	44	187	295
		Merge-joins	3	3	3	3	3	3	3	3
		Join cardinality	1561	1561	1561	1561	1561	1561	1561	1561
		Query time [ms]	27.1	27.1	28.1	31	26.1	22	20.1	19
	MJ-TD	Vertex visits	4	10	33	103	16	44	187	295
		Merge-joins	3	3	3	3	3	3	3	3
		Join cardinality	1561	1561	1561	1561	1561	1561	1561	1561
		Query time [ms]	26	29	28	32.1	28	22	20	20
	OE-BU	Vertex visits	69	100	356	848	170	882	5092	3223
		Merge-joins	5	14	66	296	54	126	996	452
		Join cardinality	2441	3293	2383	6845	4020	7106	12938	6530
		Query time [ms]	29	27	27.1	34.1	25	24.1	38	27
	OE-TD	Vertex visits	621	575	3060	45934	130	1078	8434	4712
		Merge-joins	140	61	288	4507	17	29	98	113
		Join cardinality	97698	5052	7759	169471	3798	3548	3095	2433
		Query time [ms]	78.1	20.1	29	288.4	7	10	36	23
	PE-BU	Vertex visits	69	100	356	848	58	109	182	296
		Merge-joins	5	14	66	296	21	20	77	100
		Join cardinality	2441	3293	2383	6845	2019	1245	1065	1566
		Query time [ms]	27.1	27.1	28	35.1	23.1	19	18.1	19
	PE-TD	Vertex visits	621	575	3060	45934	130	1078	8434	4712
		Merge-joins	140	61	288	4507	17	29	98	113
		Join cardinality	97698	5052	7759	169471	3798	3548	3095	2433
		Query time [ms]	83.1	21	30	291.4	7	9	36.1	21
K2	MJ-BU	Vertex visits	4	12	12	18	9	19	26	63
		Merge-joins	3	3	3	3	3	3	3	3
		Join cardinality	3110	3110	3110	3110	3110	3110	3110	3110
		Query time [ms]	28.1	29	28	31.1	20	23	23	23
	MJ-TD	Vertex visits	4	12	12	18	9	19	26	63
		Merge-joins	3	3	3	3	3	3	3	3
		Join cardinality	3110	3110	3110	3110	3110	3110	3110	3110
		Query time [ms]	31	33.1	32.1	34	24	25	26.1	27
	OE-BU	Vertex visits	3	3	3	3	7	10	17	48
		Merge-joins	3	6	6	10	2	8	8	16
		Join cardinality	3110	3178	3178	3938	1293	2104	2104	3204
		Query time [ms]	28	27	26	29.1	15.1	14	14	14
	OE-TD	Vertex visits	17	90	96	180	18	102	108	208
		Merge-joins	3	6	6	18	3	9	9	25
		Join cardinality	3110	3178	3178	4234	2191	2747	2747	3803
		Query time [ms]	22.1	20.1	20.1	23	9	9	9	10
	PE-BU	Vertex visits	3	3	3	3	7	10	17	34
		Merge-joins	3	6	6	10	5	11	18	53
		Join cardinality	3110	3178	3178	3938	3043	3599	5524	13709
		Query time [ms]	28.1	27.1	27.1	28	16	16	17	17.1
	PE-TD	Vertex visits	17	90	96	180	18	102	108	208
		Merge-joins	3	6	6	10	3	9	9	17
		Join cardinality	3110	3178	3178	3938	2191	2747	2747	3507
		Query time [ms]	21	20	21	22	8	8	7.1	8
K3	MJ-BU	Vertex visits	4	82	188	188	8	190	305	305
		Merge-joins	3	3	3	3	3	3	3	3
		Join cardinality	1766	1766	1766	1766	1766	1766	1766	1766
		Query time [ms]	25	30	30.1	31	18.1	19	19	18
	MJ-TD	Vertex visits	4	82	188	188	8	190	305	305
		Merge-joins	3	3	3	3	3	3	3	3
		Join cardinality	1766	1766	1766	1766	1766	1766	1766	1766
		Query time [ms]	27.1	31.1	31	33.1	19	21.1	21	22.1
	OE-BU	Vertex visits	3	7	7	7	3	27	27	27
		Merge-joins	3	63	67	67	2	84	88	88
		Join cardinality	1766	808	628	628	431	1423	1431	1431
		Query time [ms]	25	23	22	24	11	12	12	12
	OE-TD	Vertex visits	16	562	1123	1123	20	1018	1620	1620
		Merge-joins	3	63	67	86	3	81	85	104
		Join cardinality	1766	808	628	671	527	624	575	618
		Query time [ms]	18	15	16	20.1	2	5	8	10
	PE-BU	Vertex visits	3	7	7	7	3	25	25	25
		Merge-joins	3	63	67	67	3	81	85	85
		Join cardinality	1766	808	628	628	527	624	575	575
		Query time [ms]	24.1	23	22	24.1	12	12	12	12
	PE-TD	Vertex visits	16	562	1123	1123	20	1018	1620	1620
		Merge-joins	3	63	67	67	3	81	85	85
		Join cardinality	1766	808	628	628	527	624	575	575
		Query time [ms]	17	15.1	16.1	17	3	6	8	8

K4	MJ-BU	Vertex visits	4	4	5	13	19	26	34	75
		Merge-joins	3	3	3	3	3	3	3	3
		Join cardinality	896	896	896	896	896	896	896	896
		Query time [ms]	24.1	25.1	23	26.1	14	16	15	16.1
	MJ-TD	Vertex visits	4	4	5	13	19	26	34	75
		Merge-joins	3	3	3	3	3	3	3	3
		Join cardinality	896	896	896	896	896	896	896	896
		Query time [ms]	25	26	25.1	27	16	16.1	16	17
	OE-BU	Vertex visits	3	3	4	9	18	23	28	41
		Merge-joins	3	3	5	17	3	10	18	55
		Join cardinality	896	896	1015	1382	369	523	774	1651
		Query time [ms]	25	24.1	24	26.1	14	14	14.1	15
	OE-TD	Vertex visits	13	14	24	81	13	14	25	74
		Merge-joins	3	3	5	19	3	3	6	24
		Join cardinality	896	896	1015	1490	466	466	585	984
		Query time [ms]	14	14	14.1	17	1	1	1	2
	PE-BU	Vertex visits	3	3	4	9	18	23	28	39
		Merge-joins	3	3	5	17	4	11	20	54
		Join cardinality	896	896	1015	1382	608	762	865	1097
		Query time [ms]	24	24	25.1	26	14	14	14	15
	PE-TD	Vertex visits	13	14	24	81	13	14	25	74
		Merge-joins	3	3	5	17	3	3	6	22
		Join cardinality	896	896	1015	1382	466	466	585	876
		Query time [ms]	13	14	14	16	0	1	1	1

			bw=2				bw=3			
Query	Algorithm	Data	fw=0	fw=1	fw=2	fw=3	fw=0	fw=1	fw=2	fw=3
K1	MJ-BU	Vertex visits	27	112	374	485	33	142	429	679
		Merge-joins	3	3	3	3	3	3	3	3
		Join cardinality	1561	1561	1561	1561	1561	1561	1561	1561
		Query time [ms]	25	19	18	18	25	19 1	19 1	19
	MJ-TD	Vertex visits	27	112	374	485	33	142	429	679
		Merge-joins	3	3	3	3	3	3	3	3
		Join cardinality	1561	1561	1561	1561	1561	1561	1561	1561
		Query time [ms]	27.1	20	20	19	26	20	20	20
	OE-BU	Vertex visits	129	180	226	364	66	122	193	444
		Merge-joins	20	28	53	60	10	19	37	48
		Join cardinality	918	426	302	460	459	147	289	497
		Query time [ms]	23 1	17	14	14	19.1	15 1	14	14
	OE-TD	Vertex visits	453	4111	12499	4530	431	3220	7481	3466
		Merge-joins	39	59	80	43	27	38	28	21
		Join cardinality	10515	6702	1831	217	4366	1453	438	70
		Query time [ms]	14	24.1	59 1	24	7	15	37	20
	PE-BU	Vertex visits	55	92	141	250	66	122	139	44
		Merge-joins	13	22	42	49	11	20	30	34
		Join cardinality	1047	261	364	525	733	205	269	138
		Query time [ms]	23	15.1	17.1	14	20	14	14	14
	PE-TD	Vertex visits	453	4111	12499	4530	431	3220	7481	3466
		Merge-joins	39	59	80	43	27	38	28	21
		Join cardinality	10515	6702	1831	217	4366	1453	438	70
		Query time [ms]	16	24	59.1	25.1	8	15	39 1	20 1
K2	MJ-BU	Vertex visits	11	24	38	164	11	67	149	326
		Merge-joins	3	3	3	3	3	3	3	3
		Join cardinality	3110	3110	3110	3110	3110	3110	3110	3110
		Query time [ms]	19	23	23	23 1	21	23	23	23 1
	MJ-TD	Vertex visits	11	24	38	164	11	67	149	326
		Merge-joins	3	3	3	3	3	3	3	3
		Join cardinality	3110	3110	3110	3110	3110	3110	3110	3110
		Query time [ms]	24	26 1	26 1	27	25 1	27 1	27 1	26
	OE-BU	Vertex visits	9	15	29	134	9	28	73	166
		Merge-joins	1	4	4	8	1	4	4	8
		Join cardinality	310	866	866	1626	310	296	296	296
		Query time [ms]	14	14	13	14.1	13	14	14 1	14
	OE-TD	Vertex visits	18	105	111	215	18	135	178	282
		Merge-joins	3	12	12	32	3	12	12	24
		Join cardinality	2106	2662	2662	3718	2106	2092	2092	1490
		Query time [ms]	8	6 1	7.1	9	8	8	7	6
	PE-BU	Vertex visits	9	15	29	81	9	28	53	135
		Merge-joins	5	14	21	117	5	15	53	108
		Join cardinality	2493	3049	3049	4238	2493	2535	2739	2578
		Query time [ms]	17	16	17	18	16	16	17	16 1
	PE-TD	Vertex visits	18	105	111	215	18	135	178	282
		Merge-joins	3	12	12	24	3	12	12	24
		Join cardinality	2106	2662	2662	3422	2106	2092	2092	2092
		Query time [ms]	8	8	6	8	8	7 1	7	9
K3	MJ-BU	Vertex visits	9	260	380	395	10	285	411	456
		Merge-joins	3	3	3	3	3	3	3	3
		Join cardinality	1766	1766	1766	1766	1766	1766	1766	1766
		Query time [ms]	18	19.1	19.1	19	19	20 1	20	19
	MJ-TD	Vertex visits	9	260	380	395	10	285	411	456
		Merge-joins	3	3	3	3	3	3	3	3
		Join cardinality	1766	1766	1766	1766	1766	1766	1766	1766
		Query time [ms]	21.1	21	21	22	20.1	20	22	23 1
	OE-BU	Vertex visits	3	115	123	123	3	57	57	57
		Merge-joins	1	55	59	59	1	19	19	19
		Join cardinality	274	502	322	322	274	111	43	43
		Query time [ms]	11	13	12	12	11	12	12	12
	OE-TD	Vertex visits	21	1084	1691	1706	22	1095	1708	1753
		Merge-joins	3	93	97	116	3	57	57	57
		Join cardinality	527	459	408	451	527	201	129	129
		Query time [ms]	3	7	10	11	3	6	10	11
	PE-BU	Vertex visits	3	43	43	43	3	57	57	57
		Merge-joins	3	93	97	97	3	57	57	57
		Join cardinality	527	459	408	408	527	201	129	129
		Query time [ms]	12 1	13	12	12	12	12	13	11
	PE-TD	Vertex visits	21	1084	1691	1706	22	1095	1708	1753
		Merge-joins	3	93	97	97	3	57	57	57
		Join cardinality	527	459	408	408	527	201	129	129
		Query time [ms]	2	7	10	10 1	2	7	10	10 1

K4	MJ-BU	Vertex visits	22	66	88	169	23	96	122	225
		Merge-joins	3	3	3	3	3	3	3	3
		Join cardinality	896	896	896	896	896	896	896	896
		Query time [ms]	16	16	19	16	16.1	15	15	17
	MJ-TD	Vertex visits	22	66	88	169	23	96	122	225
		Merge-joins	3	3	3	3	3	3	3	3
		Join cardinality	896	896	896	896	896	896	896	896
		Query time [ms]	16 1	17	18	17.1	17	17	16	17.1
	OE-BU	Vertex visits	21	65	87	147	22	91	111	159
		Merge-joins	1	1	2	7	1	1	2	6
		Join cardinality	119	119	216	429	108	108	205	356
		Query time [ms]	14	14	14	15	14	15	14	15
	OE-TD	Vertex visits	13	14	25	74	13	14	25	74
		Merge-joins	3	3	6	21	3	3	6	14
		Join cardinality	335	335	432	661	324	324	421	556
		Query time [ms]	1	0	0	1	1	0	0	0
	PE-BU	Vertex visits	21	61	77	103	22	91	111	159
		Merge-joins	5	17	29	62	5	15	24	40
		Join cardinality	608	758	818	845	586	578	633	682
		Query time [ms]	14	15	15	16.1	15	15	15 1	16
	PE-TD	Vertex visits	13	14	25	74	13	14	25	74
		Merge-joins	3	3	6	19	3	3	6	18
		Join cardinality	335	335	432	553	324	324	421	540
		Query time [ms]	1	1	1	1	1	0	0	1

D.2 Branching Path Patterns

			bw=0				bw=1			
Query	Algorithm	Data	fw=0	fw=1	fw=2	fw=3	fw=0	fw=1	fw=2	fw=3
Q1a	MJ-BU	Vertex visits	9	266	690	690	27	479	1226	1257
		Merge-joins	5	5	5	5	5	5	5	5
		Join cardinality	6971	6971	6971	6971	6971	6971	6971	6971
		Query time [ms]	19	36.1	37	37.1	28.1	34.1	35	36.1
	MJ-TD	Vertex visits	6	71	177	177	12	275	704	735
		Merge-joins	5	5	5	5	5	5	5	5
		Join cardinality	2492	2492	2492	2492	2492	2492	2492	2492
		Query time [ms]	4	9	10	10	8	10	11.1	12
	OE-BU	Vertex visits	5	6	6	6	7	207	530	561
		Merge-joins	5	204	510	510	4	203	509	509
		Join cardinality	6971	25491	64147	64147	1532	1392	1447	1447
		Query time [ms]	17	7	13.1	13	3	4	9	9
	OE-TD	Vertex visits	34	514	1075	1075	38	580	1247	1247
		Merge-joins	5	68	178	178	5	68	178	178
		Join cardinality	6971	1329	1479	1479	1788	390	540	540
		Query time [ms]	20	6	9	7	5	4	9	8
	PE-BU	Vertex visits	5	6	6	6	5	141	341	341
		Merge-joins	5	204	510	510	7	207	530	561
		Join cardinality	6971	25491	64147	64147	2017	1633	1705	1736
		Query time [ms]	16.1	8	13	13	4	6	8	10
	PE-TD	Vertex visits	34	514	1075	1075	38	580	1247	1247
		Merge-joins	5	68	174	174	5	68	174	174
		Join cardinality	6971	1329	1435	1435	1788	390	496	496
		Query time [ms]	20	5	7	8	6	5	8.1	9
Q1b	MJ-BU	Vertex visits	5	133	345	345	15	236	629	660
		Merge-joins	3	3	3	3	3	3	3	3
		Join cardinality	3596	3596	3596	3596	3596	3596	3596	3596
		Query time [ms]	8	16	18	18.1	14	17	18.1	19
	MJ-TD	Vertex visits	6	71	177	177	12	275	706	737
		Merge-joins	4	4	4	4	4	4	4	4
		Join cardinality	2359	2359	2359	2359	2359	2359	2359	2359
		Query time [ms]	4	9	10	10	8	11	13	12
	OE-BU	Vertex visits	5	6	6	6	7	207	532	563
		Merge-joins	5	204	512	512	4	203	511	511
		Join cardinality	6958	25062	63238	63238	1519	1366	1421	1421
		Query time [ms]	17.1	7	11	12	2	4	8	8
	OE-TD	Vertex visits	26	510	1071	1071	30	576	1243	1243
		Merge-joins	4	66	174	174	4	66	174	174
		Join cardinality	5345	1067	1195	1195	1401	350	478	478
		Query time [ms]	16	4	7	7.1	4	5	9	8.1
	PE-BU	Vertex visits	5	6	6	6	5	141	343	343
		Merge-joins	5	204	512	512	7	207	532	563
		Join cardinality	6958	25062	63238	63238	2004	1607	1679	1710
		Query time [ms]	17	6	12.1	12	2	3	7	6
	PE-TD	Vertex visits	26	510	1071	1071	30	576	1243	1243
		Merge-joins	4	66	172	172	4	66	172	172
		Join cardinality	5345	1067	1173	1173	1401	350	456	456
		Query time [ms]	15	5	6	8	5	5	9	9
Q2a	MJ-BU	Vertex visits	6	48	48	48	6	372	399	399
		Merge-joins	3	3	3	3	3	3	3	3
		Join cardinality	700	700	700	700	700	700	700	700
		Query time [ms]	2	2	2	3	2	3	3	3
	MJ-TD	Vertex visits	4	18	18	18	4	144	153	153
		Merge-joins	3	3	3	3	3	3	3	3
		Join cardinality	639	639	639	639	639	639	639	639
		Query time [ms]	2	2	2.1	2	3	3	2	3
	OE-BU	Vertex visits	3	3	3	3	3	30	30	30
		Merge-joins	3	30	30	30	3	239	256	256
		Join cardinality	700	3463	3463	3463	700	3415	3679	3679
		Query time [ms]	3.1	1	1	1	2	2	3	2
	OE-TD	Vertex visits	19	167	167	167	19	1441	1553	1553
		Merge-joins	3	26	52	52	3	213	456	456
		Join cardinality	700	3148	3658	3658	700	3076	3826	3826
		Query time [ms]	2	2	6	7	2	6	13	12.1
	PE-BU	Vertex visits	3	3	3	3	3	30	30	30
		Merge-joins	3	30	30	30	3	239	256	256
		Join cardinality	700	3463	3463	3463	700	3415	3679	3679
		Query time [ms]	1	2	1	1	1	3	4	2
	PE-TD	Vertex visits	19	167	167	167	19	1441	1553	1553
		Merge-joins	3	26	26	26	3	213	228	228
		Join cardinality	700	3148	3148	3148	700	3076	3316	3316
		Query time [ms]	3	3	2	2	3	6	6	6

			bw=0				bw=1			
Query	Algorithm	Data	fw=0	fw=1	fw=2	fw=3	fw=0	fw=1	fw=2	fw=3
Q2b	MJ-BU	Vertex visits	7	49	49	49	7	373	400	400
		Merge-joins	4	4	4	4	4	4	4	4
		Join cardinality	790	790	790	790	790	790	790	790
		Query time [ms]	2	2	2	3	2	3	4	2
	MJ-TD	Vertex visits	5	19	19	19	5	145	154	154
		Merge-joins	4	4	4	4	4	4	4	4
		Join cardinality	729	729	729	729	729	729	729	729
		Query time [ms]	4	2	2	1	2	2	2	3
	OE-BU	Vertex visits	4	4	4	4	4	31	31	31
		Merge-joins	4	31	31	31	3	239	256	256
		Join cardinality	790	3371	3371	3371	674	5389	5779	5779
		Query time [ms]	2	1.1	2	2	1	3	4	2
	OE-TD	Vertex visits	20	168	168	168	20	1445	1557	1557
		Merge-joins	4	27	53	53	4	217	449	449
		Join cardinality	790	3160	3657	3657	790	3163	3879	3879
		Query time [ms]	2	2	6	7	3	5	12.1	13
	PE-BU	Vertex visits	4	4	4	4	4	31	31	31
		Merge-joins	4	31	31	31	4	247	264	264
		Join cardinality	790	3371	3371	3371	790	3579	3821	3821
		Query time [ms]	2	1	2	2	1	3	3	3
	PE-TD	Vertex visits	20	168	168	168	20	1445	1557	1557
		Merge-joins	4	27	27	27	4	217	232	232
		Join cardinality	790	3160	3160	3160	790	3163	3393	3393
		Query time [ms]	3	2	2	2	2	6 1	5	7
Q2c	MJ-BU	Vertex visits	7	49	49	49	7	373	400	400
		Merge-joins	4	4	4	4	4	4	4	4
		Join cardinality	752	752	752	752	752	752	752	752
		Query time [ms]	1	2	2	2	2	3	4	3
	MJ-TD	Vertex visits	5	19	19	19	5	145	154	154
		Merge-joins	4	4	4	4	4	4	4	4
		Join cardinality	691	691	691	691	691	691	691	691
		Query time [ms]	1	2	2	2	1	2	3	2
	OE-BU	Vertex visits	4	4	4	4	4	31	31	31
		Merge-joins	4	31	31	31	3	239	256	256
		Join cardinality	752	3200	3200	3200	655	4249	4583	4583
		Query time [ms]	2	1	1	1	1	2	4	2
	OE-TD	Vertex visits	20	168	168	168	20	1445	1557	1557
		Merge-joins	4	27	53	53	4	217	437	437
		Join cardinality	752	3065	3548	3548	752	2935	3613	3613
		Query time [ms]	2	2	7	6.1	2	6	12	12
	PE-BU	Vertex visits	4	4	4	4	4	30	30	30
		Merge-joins	4	31	31	31	4	245	262	262
		Join cardinality	752	3200	3200	3200	752	3233	3465	3465
		Query time [ms]	2	2	2	1	2	2	2	2
	PE-TD	Vertex visits	20	168	168	168	20	1445	1557	1557
		Merge-joins	4	27	27	27	4	217	232	232
		Join cardinality	752	3065	3065	3065	752	2935	3155	3155
		Query time [ms]	2	2	2	3	1	5	5	6 1
Q3a	MJ-BU	Vertex visits	9	16	64	145	31	57	218	459
		Merge-joins	6	6	6	6	6	6	6	6
		Join cardinality	3628	3628	3628	3628	3628	3628	3628	3628
		Query time [ms]	30.1	32	31	33.1	25.1	24 1	24	24 1
	MJ-TD	Vertex visits	7	10	26	53	17	33	118	257
		Merge-joins	6	6	6	6	6	6	6	6
		Join cardinality	2999	2999	2999	2999	2999	2999	2999	2999
		Query time [ms]	30	31	31	30	22	22	23.1	22
	OE-BU	Vertex visits	6	6	6	6	10	20	63	145
		Merge-joins	6	9	52	127	18	33	139	287
		Join cardinality	3628	3631	13710	31023	4653	7988	32393	65830
		Query time [ms]	30	28.1	28	29.1	14	15	17	19 1
	OE-TD	Vertex visits	31	78	437	1182	169	357	1222	2302
		Merge-joins	6	9	70	178	21	38	242	492
		Join cardinality	3628	3631	10288	21702	4801	5395	11594	22169
		Query time [ms]	22	20	33.1	49 1	7	8	26 1	42 1
	PE-BU	Vertex visits	6	6	6	6	10	20	58	128
		Merge-joins	6	9	52	127	23	48	191	408
		Join cardinality	3628	3631	13710	31023	5806	9182	18940	38421
		Query time [ms]	29.1	27	27 1	30	17.1	17.1	18 1	21
	PE-TD	Vertex visits	31	78	437	1182	169	357	1222	2302
		Merge-joins	6	9	39	97	21	38	143	303
		Join cardinality	3628	3631	9398	20833	4801	5395	10724	21366
		Query time [ms]	22.1	20	21	22	7	8	10	13

			bw=2				bw=3			
Query	Algorithm	Data	fw=0	fw=1	fw=2	fw=3	fw=0	fw=1	fw=2	fw=3
Q1a	MJ-BU	Vertex visits	36	520	1293	1408	40	564	1361	1596
		Merge-joins	5	5	5	5	5	5	5	5
		Join cardinality	6971	6971	6971	6971	6971	6971	6971	6971
		Query time [ms]	31	36.1	36	36	33	34	36	37.1
	MJ-TD	Vertex visits	18	292	744	814	19	303	761	861
		Merge-joins	5	5	5	5	5	5	5	5
		Join cardinality	2492	2492	2492	2492	2492	2492	2492	2492
		Query time [ms]	10	12	12	11	10	11	12	13
	OE-BU	Vertex visits	5	141	341	341	5	141	341	341
		Merge-joins	4	140	340	340	4	140	340	340
		Join cardinality	1532	1084	1033	1033	1532	1084	1033	1033
		Query time [ms]	2	3	7	6	3	3	4	7
	OE-TD	Vertex visits	39	588	1256	1271	40	599	1273	1318
		Merge-joins	5	5	8	8	5	5	8	8
		Join cardinality	1788	82	115	115	1788	82	115	115
		Query time [ms]	5	4	7.1	9.1	5	4	7	9
	PE-BU	Vertex visits	5	141	341	341	5	141	341	341
		Merge-joins	5	141	341	341	5	141	341	341
		Join cardinality	1788	1095	1044	1044	1788	1095	1044	1044
		Query time [ms]	3	5	4	8	3	6	7	8 1
	PE-TD	Vertex visits	39	588	1256	1271	40	599	1273	1318
		Merge-joins	5	5	5	5	5	5	5	5
		Join cardinality	1788	82	82	82	1788	82	82	82
		Query time [ms]	5	5	8	7	7.1	6.1	9	8
Q1b	MJ-BU	Vertex visits	22	261	678	763	24	283	712	857
		Merge-joins	3	3	3	3	3	3	3	3
		Join cardinality	3596	3596	3596	3596	3596	3596	3596	3596
		Query time [ms]	17	18	18	17	16	16	18	17.1
	MJ-TD	Vertex visits	18	292	746	816	19	303	763	863
		Merge-joins	4	4	4	4	4	4	4	4
		Join cardinality	2359	2359	2359	2359	2359	2359	2359	2359
		Query time [ms]	9	11	11	11	10	11	11	12
	OE-BU	Vertex visits	5	141	343	343	5	141	343	343
		Merge-joins	4	140	342	342	4	140	342	342
		Join cardinality	1519	1058	1007	1007	1519	1058	1007	1007
		Query time [ms]	2	3	4	5	2.1	3 1	5	6
	OE-TD	Vertex visits	31	584	1252	1267	32	595	1269	1314
		Merge-joins	4	3	4	4	4	3	4	4
		Join cardinality	1401	42	53	53	1401	42	53	53
		Query time [ms]	5.1	5	8	8	5	5	8	9
	PE-BU	Vertex visits	5	141	343	343	5	141	343	343
		Merge-joins	5	141	343	343	5	141	343	343
		Join cardinality	1775	1069	1018	1018	1775	1069	1018	1018
		Query time [ms]	3	3	4.1	4.1	2	3	5	5
	PE-TD	Vertex visits	31	584	1252	1267	32	595	1269	1314
		Merge-joins	4	3	3	3	4	3	3	3
		Join cardinality	1401	42	42	42	1401	42	42	42
		Query time [ms]	5	5	7	8	4	5	8	9
Q2a	MJ-BU	Vertex visits	6	581	625	625	6	581	625	625
		Merge-joins	3	3	3	3	3	3	3	3
		Join cardinality	700	700	700	700	700	700	700	700
		Query time [ms]	1	3	3	3	2	4	3	3
	MJ-TD	Vertex visits	4	353	379	379	4	353	379	379
		Merge-joins	3	3	3	3	3	3	3	3
		Join cardinality	639	639	639	639	639	639	639	639
		Query time [ms]	1	2	3	3	2	3	3	3.1
	OE-BU	Vertex visits	3	239	256	256	3	239	256	256
		Merge-joins	3	239	256	256	3	239	256	256
		Join cardinality	700	572	572	572	700	572	572	572
		Query time [ms]	2	4	3.1	3	2	4	4	4
	OE-TD	Vertex visits	19	1441	1553	1553	19	1441	1553	1553
		Merge-joins	3	213	456	456	3	213	456	456
		Join cardinality	700	510	1020	1020	700	510	1020	1020
		Query time [ms]	2	8	11	11	2	7.1	11.1	11
	PE-BU	Vertex visits	3	239	256	256	3	239	256	256
		Merge-joins	3	239	256	256	3	239	256	256
		Join cardinality	700	572	572	572	700	572	572	572
		Query time [ms]	2	3	3	3	2	3	4	4
	PE-TD	Vertex visits	19	1441	1553	1553	19	1441	1553	1553
		Merge-joins	3	213	228	228	3	213	228	228
		Join cardinality	700	510	510	510	700	510	510	510
		Query time [ms]	2	7	7	8	3	7	7	7

			bw=2				bw=3			
Query	Algorithm	Data	fw=0	fw=1	fw=2	fw=3	fw=0	fw=1	fw=2	fw=3
Q2b	MJ-BU	Vertex visits	7	589	633	633	7	622	667	667
		Merge-joins	4	4	4	4	4	4	4	4
		Join cardinality	790	790	790	790	790	790	790	790
		Query time [ms]	1	4.1	5	4	5	3	4	4
	MJ-TD	Vertex visits	5	361	387	387	5	394	421	421
		Merge-joins	4	4	4	4	4	4	4	4
		Join cardinality	729	729	729	729	729	729	729	729
		Query time [ms]	2.1	2	3	3	2	4	3	4
	OE-BU	Vertex visits	4	247	264	264	4	261	276	276
		Merge-joins	3	239	256	256	3	220	234	234
		Join cardinality	674	960	992	992	674	526	524	524
		Query time [ms]	2	3	4	4	1	4	3	4
	OE-TD	Vertex visits	20	1475	1589	1589	20	1475	1589	1589
		Merge-joins	4	238	470	470	4	229	434	434
		Join cardinality	790	813	1317	1317	790	546	972	972
		Query time [ms]	2	8	10	12	2	7	10	11
	PE-BU	Vertex visits	4	228	242	242	4	261	276	276
		Merge-joins	4	280	298	298	4	261	276	276
		Join cardinality	790	1056	1086	1086	790	622	618	618
		Query time [ms]	1	4	3	3	2	4	3	3
	PE-TD	Vertex visits	20	1475	1589	1589	20	1475	1589	1589
		Merge-joins	4	238	253	253	4	229	242	242
		Join cardinality	790	813	831	831	790	546	542	542
		Query time [ms]	2	7	7	8	2.1	7	7	8
Q2c	MJ-BU	Vertex visits	7	588	632	632	7	606	651	651
		Merge-joins	4	4	4	4	4	4	4	4
		Join cardinality	752	752	752	752	752	752	752	752
		Query time [ms]	1	4	5	3	4	4	4	4
	MJ-TD	Vertex visits	5	360	386	386	5	378	405	405
		Merge-joins	4	4	4	4	4	4	4	4
		Join cardinality	691	691	691	691	691	691	691	691
		Query time [ms]	2	4	3	3	4	2	3	3
	OE-BU	Vertex visits	4	244	261	261	4	229	244	244
		Merge-joins	3	237	254	254	3	204	218	218
		Join cardinality	655	766	788	788	655	488	486	486
		Query time [ms]	1	4	4	3	1	3	4	3
	OE-TD	Vertex visits	20	1475	1589	1589	20	1475	1589	1589
		Merge-joins	4	226	446	446	4	205	410	410
		Join cardinality	752	633	1099	1099	752	490	916	916
		Query time [ms]	2	7	11	11	2	8	11	12 1
	PE-BU	Vertex visits	4	211	225	225	4	229	244	244
		Merge-joins	4	262	280	280	4	229	244	244
		Join cardinality	752	824	844	844	752	546	542	542
		Query time [ms]	1	3	3	3	2	3	3 1	3
	PE-TD	Vertex visits	20	1475	1589	1589	20	1475	1589	1589
		Merge-joins	4	226	241	241	4	205	218	218
		Join cardinality	752	633	641	641	752	490	486	486
		Query time [ms]	2	8	7	7	2	8 1	7	8
Q3a	MJ-BU	Vertex visits	51	122	387	797	67	203	540	1012
		Merge-joins	6	6	6	6	6	6	6	6
		Join cardinality	3628	3628	3628	3628	3628	3628	3628	3628
		Query time [ms]	25	25.1	26.1	26	26.1	27	26.1	26.1
	MJ-TD	Vertex visits	35	82	271	565	51	163	424	780
		Merge-joins	6	6	6	6	6	6	6	6
		Join cardinality	2999	2999	2999	2999	2999	2999	2999	2999
		Query time [ms]	23	23	24	24 1	23	25 1	25	24
	OE-BU	Vertex visits	27	60	199	417	43	141	342	607
		Merge-joins	18	33	133	268	18	33	123	243
		Join cardinality	3483	3447	3113	2523	1123	1123	1089	1055
		Query time [ms]	15	14.1	16	17	15.1	16	16.1	18
	OE-TD	Vertex visits	185	391	1307	2446	185	391	1307	2446
		Merge-joins	36	64	302	567	36	64	198	378
		Join cardinality	4606	4518	4856	3841	2246	2203	1890	1702
		Query time [ms]	8	8	17	20.1	8	7	13	18
	PE-BU	Vertex visits	27	60	189	392	43	141	342	607
		Merge-joins	38	75	265	527	38	75	255	502
		Join cardinality	5018	4969	4614	3990	2658	2645	2590	2522
		Query time [ms]	17	16	19.1	20 1	23	18	19	22
	PE-TD	Vertex visits	185	391	1307	2446	185	391	1307	2446
		Merge-joins	36	64	206	398	36	64	198	378
		Join cardinality	4606	4518	3992	3078	2246	2203	2040	1798
		Query time [ms]	6	9	11	15	7	6.1	10	14 1