A Survey on Mapping Semi-Structured Data and Graph Data to Relational Data

Yuan, Gongsheng

2023-02

Yuan , G , Lu , J & Yan , Z 2023 , ' A Survey on Mapping Semi-Structured Data and Graph Data to Relational Data ' , ACM Computing Surveys , vol. 55 , no. 10 . https://doi.org/10.1145/3567444

http://hdl.handle.net/10138/354263
https://doi.org/10.1145/3567444

A Survey on Mapping Semi-Structured Data and Graph Data to Relational Data

GONGSHENG YUAN, Zhejiang University, China and University of Helsinki, Finland
JIAHENG LU and ZHENGTONG YAN, University of Helsinki, Finland
SAI WU, Zhejiang University, China

The data produced by various services should be stored and managed in an appropriate format for gaining valuable knowledge conveniently. This leads to the emergence of various data models, including relational, semi-structured, and graph models, and so on. Considering the fact that the mature relational databases established on relational data models are still predominant in today's market, it has fueled interest in storing and processing semi-structured data and graph data in relational databases so that mature and powerful relational databases' capabilities can all be applied to these various data. In this survey, we review existing methods on mapping semi-structured data and graph data into relational tables, analyze their major features, and give a detailed classification of those methods. We also summarize the merits and demerits of each method, introduce open research challenges, and present future research directions. With this comprehensive investigation of existing methods and open problems, we hope this survey can motivate new mapping approaches through drawing lessons from each model's mapping strategies, as well as a new research topic - mapping multi-model data into relational tables.

CCS Concepts: • **Information systems** → **Relational database model**; **Semi-structured data**; **Hierarchical data models**; **Network data models;**

Additional Key Words and Phrases: Relational schema, relational storage, semi-structured data, JSON, XML, graph data, RDF, property graph, model mapping

ACM Reference format:
Gongsheng Yuan, Jiaheng Lu, Zhengtong Yan, and Sai Wu. 2023. A Survey on Mapping Semi-Structured Data and Graph Data to Relational Data. *ACM Comput. Surv.* 55, 10, Article 218 (February 2023), 38 pages.
https://doi.org/10.1145/3567444

1 INTRODUCTION

Since the emergence of **database management systems (DBMSs)**, the database community has been continuously exploring which kinds of data models are appropriate for such a system. This is because data modeling establishes the logical structure of a database, which determines how data is stored, organized, and manipulated in the databases. With the evolution of data models - from

The work is supported by the China Scholarship Council, the Zhejiang Provincial Natural Science Foundation (Grant No. LZ21F020007), and the National Natural Science Foundation of China (Grant No. 61872315).
Authors' addresses: G. Yuan, Zhejiang University, Hangzhou, China, 310027, University of Helsinki, Helsinki, Finland, 00014; email: gongsheng.yuan@helsinki.fi; J. Lu (corresponding author) and Z. Yan, University of Helsinki, Helsinki, Finland, 00014; emails: {jiaheng.lu, zhengtong.yan}@helsinki.fi; S. Wu, Zhejiang University, Hangzhou, China, 310027; email: wusai@zju.edu.cn.

0360-0300/2023/02-ART218
https://doi.org/10.1145/3567444

the relational model (relationships across records are predefined and normalized), semi-structured model (self-describing, constantly evolving, convenient for data exchange), to the graph model (capturing the inherent graph structure of data) - a wide variety of database systems have been developed. For example, an **RDBMS (relational DBMS)** (e.g., Oracle [98]) is based on the relational model [38]; **XML (eXtensible Markup Language)** databases (e.g., MarkLogic [87] and BaseX [23]) that manage data in XML format [30] are a flavor of document-oriented databases; A **JSON (JavaScript Object Notation)** document database (e.g., MongoDB [93]) that is designed to store and query data as JSON documents [119] is also a flavor of document-oriented databases. A graph database (e.g., AllegroGraph [9] and Neo4j [95]) uses graph structures (e.g., **RDF (Resource Description Framework)** [77] and **PG (property graph)** [60]) to represent and store data. The data models (key-value, column, document, and graph stores) [42] used by NoSQL databases are different from the relational model in RDBMSs, making some operations faster in NoSQL databases. However, compared to those NoSQL databases, RDBMSs still dominate today's business market since they possess mature and powerful capabilities to handle security, query optimization, transaction management, and so on. Therefore, there is increasing interest in storing and processing NoSQL data in RDBMSs. Storing those data in RDBMSs could enable NoSQL data applications to access and update legacy relational database tables easily while bridging the gap between structured and NoSQL information. Besides, this approach could make several data models survive together in a relational database, making it possible to constitute applications involving multi-model data (i.e., using relational tables to preserve structured tabular data, using semi-structured documents to record unstructured object-like data, and using graphs to store highly linked referential data) [83].

However, due to the mismatch between the complexity of NoSQL data structure and the simplicity of flat relational tables, it is a challenge to store these datasets in RDBMSs with a relational schema. To deal with this challenge, many researchers proposed a variety of approaches. As shown in Table 1, we classify those approaches into several categories based on their principal techniques and strategies. Each category will be discussed in-depth and compared in the following sections.

Scope: The goal of this survey is to perform a comprehensive study on transforming XML, JSON, RDF, and PG data into relational data. We temporarily do not consider key-value store and column store. This is because that key-value and column stores place emphasis on a data storage paradigm. For example, ArangoDB [13] (a graph database) is key-value stored internally. MonetDB [92] (an RDBMS) preserves data in vertical fragmentation (aka. column store). Besides, to the best of our knowledge, there are no relevant works about mapping key-value and column data into relational data after searching literature on the web.

Main Contributions: This survey revisits existing approaches of mapping semi-structured and graph data to relational data and summarizes their main features. The detailed categorization and comparative analysis enable the reader to capture the aerial view of this field and quickly locate the research field he/she is interested in. As we do not limit the survey to a specific data model, the reader can get a broader scope in this research area. Besides, we identify open problems and future directions to show that it is still a challenging and promising research area. The comprehensive review and analysis make this article useful in motivating new mapping techniques for storing semi-structured and graph data in RDBMSs, serving as a technical reference for choosing appropriate mapping methods under different scenarios, and providing an alternative way for implementing multi-model databases products. In particular, the main contributions are summed up as follows:

(1) We chronicle the approaches of mapping semi-structured or graph data into relational data and provide a detailed classification of these approaches.
(2) We provide a comprehensive overview of existing methods and a detailed description of their features for practitioners or organizations to choose which approaches suited them most.

Table 1. An Overview of Mapping Methods

		Approach	Literature	Description
Semi-Structured Document	XML	Structure-Based	[10, 11, 47, 117]	It takes advantage of XML schemas to generate a specific relational schema for each XML document.
		Model-Based	[59, 67, 71, 75]	It is a generic mapping regarding XML as a tree and designing mapping based on nodes, edges, paths, etc.
		Semantic Information-Based	[41, 78, 86, 114]	It adopts XML constraints (e.g., keys/foreign keys) to improve the generated relational schema's quality.
		Cost-Driven	[24, 25, 109, 141]	It uses a cost model to estimate cost of each schema to find/generate the "optimal" relational schema.
	JSON	Structure-Based	[66]	It extracts the JSON schema information from JSON data and use it to create a relational schema.
		Model-Based	[21, 34]	It shreds JSON documents into relational data with a fixed and generic relational schema.
		Unsupervised Learning-Based	[46]	It designs a relational schema for an input JSON document automatically.
		Cost-Based	[118]	It adapts the relational data layout dynamically to minimize cost for a given JSON document.
Graph Data	RDF	Triples Table	[31, 64, 89, 96]	The RDF data is preserved as a linearized list of triples and stored by a three-column schema.
		Property Table	[37, 81, 130, 131]	Based on the data's regularity (frequent patterns), it stores several related properties in the same table.
		Path-Based	[88]	As RDF data structure is a directed graph, it designs relational schemas with path information.
		Vertical Partitioning	[4]	It uses a fully decomposed storage model (DSM) to preserve RDF data in the relational tables.
		Entity-Oriented	[26]	It uses a mix of horizontal and binary tables.
		DRL-Based	[140]	It adopts Deep Reinforcement Learning (DRL) to design an adaptive relational structure to store RDF.
	PG	Column-Oriented	[121]	It utilizes the column group concept to create a flexible relational schema.

(3) We compare existing methods from various viewpoints and then present their cons and pros, which could make readers understand these methods clearly.
(4) We identify open problems, present future research directions, and indicate storing multi-model data in tabular format in RDBMSs forms a challenging and promising research area.

Related Work: So far, there exist some reviews involving how to map semi-structured or graph data into relational data. Bourret [28] (1999) provides an introduction to the table-based mapping approach that is one of the commonly used methods to map the XML schema to the relational

schema. Chaudhuri and Shim [35] (2003) hold a seminar to discuss how to represent XML data in the relational model. Gou et al. [63] (2007) begin with the introduction of XML query processing and then describe how to utilize RDBMSs to store and query XML data. Mlynkova and Pokorny [91] (2007) focus on adaptive or flexible mapping methods (e.g., [12, 22, 25]) to improve XML processing based on RDBMSs. Kolahi and Libkin [73] (2007) use an information-theoretic approach [14] to compare XML designs and corresponding designs of relational schema. Kader et al. [68] (2008) present advantages of hybrid storage combining structure mapping and XML data type. Vyas et al. [128] (2014) review several techniques for mapping XML data to relational data with supervised and unsupervised learning. Then Mourya et al. [94] (2015) simply demonstrate two mapping strategies: schema-aware and schema-less. Next, Qtaish et al. [106] (2015) review and compare some model-based mapping approaches. A more recent survey by Qtaish et al. [108] (2019) revisits the popular methods employing RDBMSs to manage XML data by relational schema, but this survey lacks a detailed discussion on those methods and open problems. Petković and Piriyaie (2021) [102] provide a simple comparison between the Argo/3 approach [34] and **Single Table Data Mapping (STDM)** [21]. As for storing graph data in RDBMSs, Velegrakis [127] (2010) and Sakr and Al-Naymat [112] (2010) review several approaches on mapping RDF data into relational data and indicate the advantages of managing RDF in RDBMSs. MahmoudiNasab and Sakr [85] (2010) give us an experimental evaluation of several relational representations of RDF data. Faye et al. [55] (2012) survey three strategies (i.e., triple table, property table, and vertical partitioning) to store RDF data in RDBMSs. After that, Wylot et al. [134] (2018) list several relevant works on RDF data storage and query processing. Unfortunately, previous surveys only focus on XML-To-Relational or RDF-To-Relational mappings. There exists no paper discussing JSON-To-Relational and PG-To-Relational mappings. Compared to current works, this survey covers all the most popular data models in NoSQL data. But this survey is not simply piled up by several data models; it could provide a clear dissection of this research field in extent (semi-structured and graph models) and depth (the detailed categorization and comparative analysis). The full review makes it a complete technical reference, and we hope readers could get some inspiration from this article.

Outline: The rest of this article is organized as follows: Section 2 presents a comprehensive introduction of mapping semi-structured documents into relational data. Section 3 offers a detailed description of mapping graph data into relational data. In Section 4, we identify open problems and present future research directions, and finally, we conclude this survey in Section 5.

2 MAPPING SEMI-STRUCTURED DATA INTO RELATIONAL DATA

Unlike the highly structured table instances in RDBMSs, semi-structured data is schema-less. This paradigm lacking predetermined schema upfront is a self-describing data model (i.e., it contains the data structure along with its actual values, and its data instances allow different objects to have different structures and properties). These flexible features relieve developers from the upfront schema design effort and let them more quickly get their applications up and running without worrying in advance about which attributes may appear in their raw data or about their domains, types, and dependencies [124]. However, the introduction of semi-structured data increases the difficulty of data management. For example, it may improve long-term development and maintenance complexity. Specifically, due to a lack of explicit entity and relationship tracking, it will burden new developers who are unfamiliar with the raw data [46]. We think that it is an excellent way to develop an efficient solution for storing semi-structured data based on firm theoretical foundations. Thus, RDBMSs, based on relational algebra [38], may be one of the best choices in providing a promising and economical solution to handle semi-structured data, which also has the following advantages:

- RDBMS is a mature system and scales very well by relational technologies (e.g., TiDB [2]).
- People could use many powerful capabilities of RDBMSs and do not need to spend decades developing native semi-structured systems.
- People could use a common and standard **Structured Query Language (SQL)**.

Besides, the advantages of mapping semi-structured data into relational data include but are not limited to the following points:

- This method could provide a way to combine the best features of both worlds: the flexibility of semi-structured data, consistency of the relational model, and efficiency of SQL;
- Storing semi-structured data in a good-designed relational schema is beneficial to the fast and efficient querying and avoids some long-term issues (e.g., sharing, performance) [124];
- We could obtain **ACID (atomicity, consistency, isolation, durability)** compatibility from the features of SQL when querying JSON instances from the relational schema [66].

Consequently, it has attracted considerable interest in leveraging RDBMSs' powerful and reliable data management services to store and query semi-structured data [11]. Generally, there are about three ways to store semi-structured data in RDBMSs. The first one is defining a data type, a built-in one, in RDBMSs for preserving the semi-structured data. For example, XML data type [1] could preserve the XML content of the data in an internal representation. This internal representation contains information such as document order and containment hierarchy. However, it does not support some column and table constraints, such as PRIMARY KEY/FOREIGN KEY, UNIQUE, and COLLATE. And it is cast or converted to [n]varchar(max), not supporting text. The second way is to store semi-structured data with SQL data types (e.g., [82]) such as large object storage, [n]varchar(max), varbinary(max), VARCHAR2, **Character Large Object (CLOB)**, and **National Character Set Large Object (NCLOB)**. However, it is not efficient to parse a large object for accessing an element or attribute. Even though it is appropriate for retrieving documents, it requires specific indices to facilitate processing. The last one is to shred the semi-structured data into relational tables. In this survey, we mainly focus on the third one and provide an overview of relevant works to guide practitioners or organizations to choose which approaches suit them most. If readers are interested in the previous two methods, Appendix A gives some related introductions in detail.

For the third approach, due to the mismatch between the relational and semi-structured data models, we need a "good" mapping for shredding and loading the semi-structured data into relational tables. The meaning of "good" depends on several factors, such as the nature of data, the application, and the query workload. Specifically, we give the following challenges:

(1) Keep the data accuracy and avoid data loss while shredding;
(2) Maintain the structure of semi-structured documents;
(3) Consider integrity constraint;
(4) Reduce storage consumption;
(5) Achieve efficiency for query and update operations;
(6) Support the semantic search;
(7) Handle dynamics of semi-structured data;
(8) Enable systems to reconstruct original semi-structured data.

Because the data model of semi-structured data is essentially different from that of relational data, it is not easy to define a "good" mapping. Firstly, the relational model is a flat, normalized, and unordered data representation with tables, records, and columns. Then, we not only need to address the hierarchical and ordered structure of semi-structured data with relational tables, but

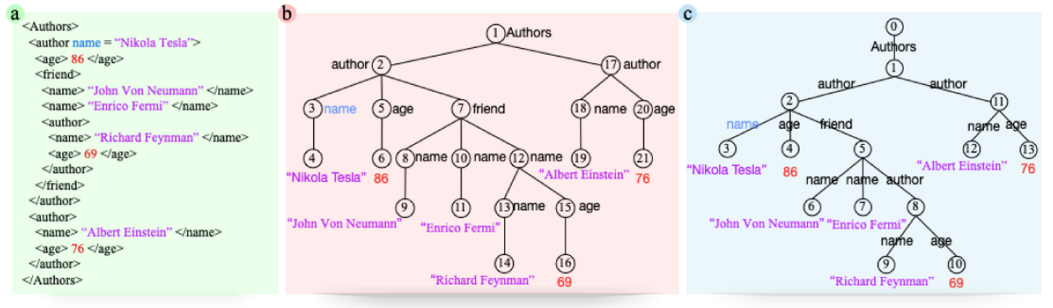

Fig. 1. An Example of XML. (a) XML document. (b) Node-labeled XML tree. (c) Edge-labeld XML tree.

sometimes we also need to take its nested and recursive elements into account. Although it is difficult, with the increasing popularity and amount of semi-structured data on the Web, this growth has prompted numerous researchers to propose various designs and strategies for mapping semi-structured data into rows and columns within tables. Therefore, this section reviews these works, summarizes their methods, and gives their limitations. In detail, we briefly introduce the most famous representatives of the semi-structured data model, XML, and JSON, firstly. Next, Sections 2.2 and 2.3 present the existing solutions of mapping XML and JSON documents into relational tables, respectively.

2.1 The Preliminaries of XML and JSON

2.1.1 XML. Figure 1 presents an example of an XML document, and it can be modeled as a labeled and ordered tree. According to label location, an XML tree can be depicted as a *node-labeled* tree or an *edge-labeled* tree. As one of the most important representatives of semi-structured data, XML has been widely applied to exchange and express data on the internet. Also, the self-describing feature of semi-structured data making XML describe data independent from platforms facilitates all kinds of applications and services supporting XML. These facts make the size of XML quickly increase, which leads researchers to consider storing XML in RDBMSs so that people could make better use of the properties from both XML and RDBMSs. To store XML in RDBMSs, many methods have been proposed. For those methods, people generally divide them into two categories based on whether XML schema (i.e., **document type descriptor (DTD)**) is known. When XML's schema exists, people collect structural constraint information from the schema file and use it to guide the mapping process making different XML documents have different relational schemas. Unfortunately, since XML's schema may not always be available, people propose using a generic mapping, a fixed and pre-defined relational schema, to store all XML documents. In Section 2.2, we will describe a more fine division based on the current classification and the paper's predominant technique while providing a review regarding mapping XML to a relational schema.

2.1.2 JSON. Considering that XML needs many rules to represent semi-structured data, this complexity makes it a less-than-ideal format for representing data-oriented semi-structured data. As the other most important data format of semi-structured data, JSON is proposed as a light-weight, schema-less, readable, writable, and language-independent data interchange format on the web. Nowadays, it has become a popular format since it is simple yet powerful, which not only supports hierarchical, nested, dynamic, and self-describing data structure but is easy to parse and generate by machines. Each JSON object consists of structured *key-value* pairs, where *key* denotes attribute name, *value* is the attribute value. Since value has four primitive types (i.e., String, Number, Boolean, and Null) natively supported by Javascript, it further improves JSON's popularity.

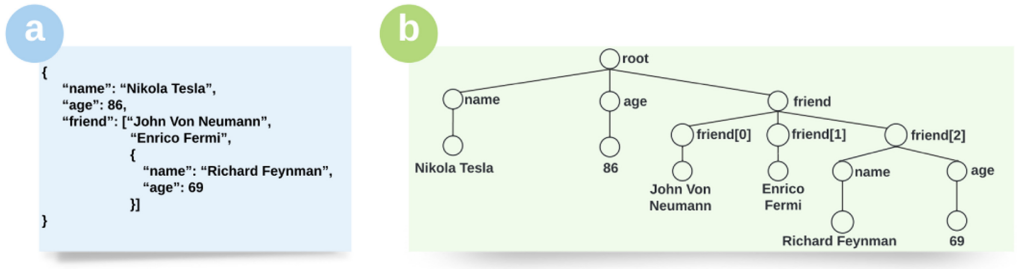

Fig. 2. An Example of JSON. (a) JSON document. (b) JSON tree.

Figure 2 shows a JSON document that can also be modeled as a tree. The growing popularity of JSON leads to the rapid growth of JSON data, which has fueled more and more interest in loading and processing JSON documents within RDBMS. Unfortunately, the essential properties of JSON format (e.g., schema-less and dynamic) present enormous challenges for mapping JSON documents to relational tuples. We have to consider the data sparseness caused by JSON's flexibility and the mismatch between the complexity of JSON's hierarchical and recursive structure and the simplicity of flat relational tables. To overcome these difficulties, researchers have made some attempts. We will review these mapping approaches to show their development and summarizes them in Section 2.3.

2.1.3 The Differences between XML and JSON. As the leading representatives of semi-structured data, XML and JSON have many similar features. However, since JSON is proposed as a lightweight, compact data interchange format to replace XML in some applications, their differences result in the difference between XML-to-Relational mapping and JSON-to-Relational mapping. These differences between XML and JSON include but are not limited to the following points.

(1) **Schema.** XML document has XML schema or DTD information to describe restrictions on its structure. In contrast to XML, JSON does not have a similar equivalent.
(2) **Order.** XML has a sibling concept while JSON has index order in its array.
(3) **Path.** Nodes (elements) of XML documents with the same tag may have the same path (parent), but JSON does not.
(4) **Interacts.** People could map JSON data to the native data types of Javascript. For XML, users need to use the programmatical way to interact with XML documents via **DOM (Document Object Model)** or **SAX (Simple API for XML)**.

The similarities between JSON and XML allow people to use some methods from the XML-to-Relational mappings to do JSON-to-Relational mappings, while some distinguishing characteristics of JSON make researchers need to reconsider how to do JSON-to-Relational mapping. Managing massive volumes of semi-structured data with RDBMSs is a challenge, but there are also enough benefits to use an SQL engine as the target query processor for semi-structured data operations.

2.2 Mapping XML Documents to Relational Data

2.2.1 Structure-Based Mapping. The first category is schema-based mapping, also called a structure-based approach, where the schema/structure means the XML schema (DTD) - describing the structure of XML data and facilitating the data exchange among different applications based on a consensus on the meaning of tags. Here, it could help design a more compact storage schema by eliminating redundancy and help improve query efficiency by significantly reducing the number of joins (e.g., inlining as many proper elements as possible into a single table). Therefore,

Table 2. Comparison among Structure-Based Mapping Methods

	Time	Works	Contributions	Order Preserved	Empirically Validated
Inlining	1999	*Inlining* [117]	Proposing three **Inlining** techniques (*Basic*, *Shared*, and *Hybrid*)	No	Yes
	2001	*Yan and Fu* [90]	Discovering FDs and the candidate keys to normalize the relational schema prototype gotten by the **Inlining** technology	No	No
	2003	*NewInlining* [84]	An improved ***Inlining*** [117] to deal with DTDs including arbitrary cyclic DTDs, to eliminate redundancy for shard nodes, and to reduce the number of relations	Yes	No
	2005	Balmin et al. [22]	Creating relational schema with technologies of binary-coded XML fragments and denormalized tables that feature **inlined** repeatable elements	Yes	Yes
	2007	*ODTDMap* [16]	An improved ***Inlining*** [117] with the ability to handle a DTD having cycles and set-valued XML attributes	Yes	Yes
	2013	*Suri and Sharma* [123]	Presenting an **Inlining** algorithm for handling recursion in an XML document	No	No
General Annotation	2004	*MDF* [12]	Proposing a mapping definition framework based on a declarative approach	Yes	No
	2004	*ShreX* [11, 47]	A modular and extensible mapping system	Yes	No
General Label	2004	*X-Ray* [69]	Proposing a generic approach for integrating XML with RDBMSs	Yes	No
	2005	*SPIDER* [10]	Proposing a labelling scheme	Yes	Yes

the primary purpose of this part is to comprehensively review the development of the structure-based approach, summarize them, and represent them in Table 2.

Inlining is proposed in [117], which uses a set of transformations to "simplify" the original DTD's complexity while preserving the semantics. Next, it utilizes a DTD graph to represent the simplified DTD and converts the DTD graph to relations. However, there is a high probability of causing excessive fragmentation of the document when directly mapping elements to relations. Hence, the *Basic* inlining is presented to solve the fragmentation problem by inlining as many descendants of an element as possible into a single relation. But, due to the *Basic* allowing an element node to appear in multiple tables repeatedly, the proposed *Shared* inlining technology identifies these element nodes and creates separate tables for these elements to share. Besides, to control the number of tables, the *Shared* provides some rules to decide whether or not to create a

relation. Finally, it proposes the *Hybrid* inlining technology to combine the *Basic* (join reduction properties) with the *Shared* (the sharing features) for improving query performance.

Yan and Fu [90] propose two algorithms (Global Schema Extraction and DTD-splitting Schema Extraction) to generate relational schemas based on the XML data and the DTD. Those two algorithms have the same framework. Firstly, they simplify DTD; Then they need to create schema prototype trees; Next, they form relational schema prototypes; After that, they find **functional dependencies (FDs)** and candidate keys; Finally, they normalize the formed prototypes. However, the global algorithm analyzes the XML data to discover FDs and the candidate keys. Next, they use them to normalize the prototypes. The DTD-splitting algorithm infers features of the XML data from the DTD and conducts schema decomposition (DTD split) before discovering FDs and keys.

NewInlining is proposed in [84] and inspired by the shared-inlining method [117]. It starts with simplifying DTDs by a set of new transformation rules. Next, it creates and inlines DTD graphs, where inlining rules eliminate the redundancy and deal with DTDs containing arbitrary cycles. Finally, it generates relational schemas based on the inlined graph.

Balmin et al. [22] propose a schema-driven decomposition framework, which firstly adopts the labeled tree notation to represent XML data. Next, it utilizes schema graphs to abstract the syntax of XML schema definitions, decomposes the schema graph into fragments (including non-**MVD (Multi-valued dependency)** fragments), and generates a relational table definition for each fragment. Finally, it decomposes the XML data and loads them into the corresponding tables.

ODTDMap is proposed in [16], which simplifies DTD, creates a DTD graph, does the inlining operation, and generates the database schema and $\delta-$mapping. Besides, two data mapping algorithms (*OXInsert* and *SDM*) are proposed. Both *OXInsert* and *SDM* utilize globe IDs of elements to help reconstruct XML documents.

Suri and Sharma [123] propose mapping an XML DTD into relations, which has three steps: (1) simplifying the complexities of DTD; (2) creating a DTD graph based on simplified DTD; and (3) using the proposed inlining algorithm to generate a relational schema from the DTD graph, where the algorithm will decide to create one or two relations for the two elements appearing in a cycle.

MDF is proposed in [12], which is a **mapping definition framework (MDF)**. MDF starts with annotating an input XML schema with a limited number of pre-defined annotations, then parses annotated XML schema, creates the relational schema, verifies mapping correctness and losslessness, and ends up with shredding the input XML document to tuples.

ShreX is proposed in [11, 47], which provides a comprehensive system for mapping, loading, and querying XML documents. Specifically, the mapping is specified by annotating an XML schema, which shows how elements and attributes are stored in tables. Furthermore, it makes mappings diversify through combining different annotations. That is, *ShreX* can use existing mapping strategies as well as potential new mapping techniques. The annotation processor's function is to parse an annotated XML schema, check the validity of the mappings, and form the corresponding relational schema. Finally, the document shredder shreds an XML document and generates the tuples.

X-Ray is proposed in [69], whose principal purpose is to support the existing schemas. *X-Ray* offers several basic mapping options and decides which kind of mapping is reasonable according to different situations. Reasonable mappings are served as mapping patterns to promote the mapping process at a syntactical level through analyzing the database schema and the DTD and suggesting potential mappings as well as preventing others due to syntactical conflicts. Since those mapping patterns are universally applicable, *X-Ray* employs them to represent mapping knowledge for mapping an XML to a relational schema.

SPIDER is proposed in [10], which uses **SPIDER (Schema based Path IDentifiER)** to identify paths from the root node to a node, adopts Sibling Dewey Order to identify multiple nodes appearing in the same path, and designs the following four relational tables to preserve the XML document.

(1) **Element** (*docID, nodeID, spider, sibling, parentID*);
(2) **Attribute** (*docID, nodeID, spider, sibling, parentID, value*);
(3) **Text** (*docID, nodeID, spider, sibling, parentID, value*);
(4) **Path** (*spider, pathExp*).

Discussion. With DTDs, the relational schema generated by a structure-based approach is tailored to specific XML documents. This is to say, structure-based approaches could use predefined rules to generate different relational schemas for different DTDs. These schemas usually tend to have a more compact storage representation and an excellent query performance [126]. However, both inlining and annotation techniques do not consider semantic constraints. Besides, due to a lack of path information, some queries require many joining operations in the relational schema generated by the above methods. What's more, a complex and large XML schema may generate a relational schema with many simple tables, although the XML document instance is simple. As for *X-Ray* [69], it is just a research prototype supporting the existing schemas. Finally, *SPIDER* [10] uses a pair of spider and Sibling Dewey Order to identify each node. With these labeled nodes, it creates a four-table schema according to *XRel* [115]. Although this schema could reduce the range of relabeling (spider is not affected) when updating documents and make retrieval more efficient, it cannot store node orders exactly by employing a pair of spider and Sibling Dewey Order if a DTD contains multiple components having the same name but appearing in different places [10]. Furthermore, XML documents do not require DTDs' existence. This fact would cause a problem that these methods may not be inapplicable when the absence of DTDs.

2.2.2 Model-Based Mapping. Contrary to the previous work, this part deals with mapping in the absence of XML schema. In fact, schema absence is a common phenomenon these days, which makes querying these schemaless XML documents difficult. Considering this, people propose a model-based approach to map XML documents without schema information into relational data as an alternative way to solve this difficulty. Generally, the model-based approach is a generic mapping that regards an XML document as a tree model and designs mapping based on nodes, edges, paths, and so on. Next, we will review the development of this generic mapping comprehensively.

(1) Fixed Schema. The work presented in this portion (and summarized in Table 3) is about mapping an XML document into relation tuples with a fixed number of tables.

Edge-Based Approach. This approach maintains the Parent-Child (using Source object - Target object) and Ancestor-Descendent (using self-join) relationships in the table.

Florescu and Kossmann [58, 59] regards an XML document as an ordered and labeled directed graph, where each XML element is a node, element-subelement relationships are edges, values of an XML document are leaves. Then it proposes three alternative ways to record edges of a graph:

(1) Store all edges of the graph in a single table (i.e., the edge approach):
Edge (*source, ordinal, name, flag, target*).
(2) Class every edge having an identical label to a table:
$\mathbf{B}_{name}$ (*source, ordinal, flag, target*);
(3) Use a single universal table to store all the edges (i.e., the universal table):
Universal ($source, ordinal_{n_1}, flag_{n_1}, target_{n_1}, \ldots, ordinal_{n_k}, flag_{n_k}, target_{n_k}$).

two alternative ways to preserve the leaves:

Table 3. Comparison among Model-Based Mapping Methods with Fixed Schema

Approaches	Time	Works	Order Preserved	Empirically Validated	No. of Tables
Edge-Based	1999	*Florescu and Kossmann* [58, 59]	Yes	Yes	1 or more
Path-Based	2001	*XRel* [71]	Yes	Yes	4
	2004	*SUCXENT* [103]	Yes	Yes	5
	2004	*SUCXENT++* [104]	Yes	Yes	5
	2010	*Xlight* [139]	Yes	Yes	5
	2010	*SMX/R* [6]	Yes	Yes	2
Edge- & Path- & Signature-Based	2002	*XParent* [67]	Yes	Yes	4
	2008 2009	*XPred* [132, 133]	Yes	Yes	3
	2012	*Wang et al.* [129]	No	No	2
	2012	*Ying et al.* [136]	Yes	Yes	4
Edge- & Path-Based	2001	*Khan and Rao* [70]	No	Yes	2
	2005	*XPEV* [105]	Yes	Yes	3
Path- & Signature-Based	2005	*LNV* [50]	Yes	Yes	2
Pointer-Based	2006	*XMLEase* [51]	No	Yes	1
Token-Based	2008	*Dweib et al.* [48]	Yes	Yes	2
	2009	*MAXDOR* [49]	Yes	Yes	2
Edge- & Signature-Based	2012	*XRecursive* [52, 53]	No	Yes	2
	2012	*Suri and Sharma* [122]	Yes	Yes	2
Labeling-Based	2012	*s-XML* [120]	Yes	Yes	2
Path- & Labeling-Based	2015	*XMap* [29]	Yes	No	3
	2016	*XAncestor* [107]	Yes	Yes	2
	2017	*Mini-XML* [142]	Yes	Yes	2

(1) Establish separate *Value* tables for each conceivable data type: $\mathbf{V}_{type}$ (*vID, value*).
(2) Store values together with edges (*Inlining*) to keep values and attributes in the same tables.

which leads to overall six different relational schemas for storing XML documents (i.e., graphs).

In the above tables, the attribute *source* keeps the source ids of each edge, the *target* preserves the target ids and utilizes the $flag$ to distinguish between internal nodes and leaves, the *ordinal* holds the orders of edges, and $n_1, \ldots, n_k$ in the table **Universal** are the label names.

Path-Based Approach. It preserves all available path expressions (from the root to each node in the XML tree) in a relational attribute.

XRel is proposed in [71], which decomposes an XML document into nodes based on its tree structure, stores the simple path expressions (from the root to node) of these nodes, and preserves these nodes in different relational tables according to their types.

(1) **Path** (*pathID, pathExp*);
(2) **Element** (*docID, pathID, start, end, index, reindex*);
(3) **Text** (*docID, pathID, start, end, value*);
(4) **Attribute** (*docID, pathID, start, end, value*).

XRel designs a schema containing four tables to store the combination of the path expression and the region of nodes in an XML tree as relational tuples. These could help record the topology information of the XML tree and the expanded names of nodes. The attributes *start* and *end* indicate start and end position of a region. The *index* represents the order of an element node among its siblings in the XML document order, and the *reindex* indicates the reverse document order.

SUCXENT is proposed in [103], which stores the information of paths and nodes in tables:

(1) **Document** (*docID, docName*);
(2) **Path** (*pathID, pathExp*);
(3) **PathValue** (*docID, pathID, leafOrder, siblingOrder, leftSibIxnLevel, leafValue*);
(4) **TextContent** (*docID, linkID, text*);
(5) **AncestorInfo** (*docID, siblingOrder, ancestorOrder, ancestorLevel*).

The table **Path** preserves paths of all the leaf nodes. **PathValue** stores all leaf nodes, where the column *leftSibIxnLevel* storing the level of the highest common ancestor of the leaf node is used to reconstruct the XML document, the column *leafValue* is used to record the textual content of the leaf node. However, for large textual data (e.g., DNA sequences), *LeafValue* only keeps a link. *SUCXENT* uses another table **TextContent** to hold such large data. As for **AncestorInfo**, it saves the ancestor information for each leaf node to quickly answer some queries.

SUCXENT++ is proposed in [104], which stores the leaf nodes and the associated paths together with new offered attributes to handle the recursive XML queries.

(1) **Document** (*docID, docName*);
(2) **Path** (*pathID, pathExp, cPathID*);
(3) **PathValue** (*docID, pathID, leafOrder, cPathID, branchOrder, branchOrderSum, leafValue*);
(4) **TextContent** (*docID, pathID, leafOrder, cPathID, branchOrder, branchOrderSum, text*);
(5) **DocumentRValue** (*docID, level, rValue*).

It introduces the attribute *cPathID* to convert any recursive path expression to a range query. Users could use the attributes *branchOrder*, *branchOrderSum*, and *rValue* to decrease the consumption of storage and the times of join operations.

Xlight is proposed in [139], whose schema has the following five relational tables:

(1) **Document** (*docID, docName*);
(2) **Path** (*pathID, pathExp*);
(3) **Data** (*docID, pathID, leafNo, leafGroup, linkLevel, leafValue, hasAttrib*);
(4) **Ancestor** (*docID, leafGroup, ancestorPre, ancestorLevel*);
(5) **Attribute** (*name, val, id, pre*).

In this schema, the table **Data** stores all the information of leaf nodes in the XML document. **Ancestor** preserves the ancestor information of each leaf node. The attribute *leafGroup* marks the same number for any leaf nodes having the same parent. The *linkLevel* indicates the level that each path is linked with its previous path. The *hasAttrib* records the number of attributes in each path.

SMX/R is proposed in [6], where *startPos/endPos* denotes the starting (pre-order) /end (post-order) location of the node.

(1) **Path** (*docID, pathID, startPos, endPos, nodeLevel, nodeType, nodeValue*);
(2) **PathIndex** (*pathID, pathExp, nodeName*).

Edge- & Path- & Signature-Based Approach. It preserves path expressions (path-based method), parent-child relationships (edge-based method) in the tables. Besides, this approach assigns a different signature (number) to each distinctive label (node).

XParent is proposed in [67], where the table **LabelPath** provides a global view of the XML documents. **DataPath** keeps parent-child relationships, which can be further materialized as ancestor-descendant relationships. The attribute *length* and *order* represent the number of edges in the label path and the order of the element among its siblings, respectively.

(1) **LabelPath** (*pathID, length, pathExp*);
(2) **DataPath** (*parentNodeID, childNodeID*); /**Ancestor** (*nodeID, ancestorID, level*);
(3) **Element** (*pathID, nodeID, order*);
(4) **Data** (*pathID, nodeID, order, value*).

XPred is proposed in [132, 133], which stores the structural information (e.g., parent-child relationship and order) distributively into nodes to reduce the number of joins when doing queries.

(1) **Path** (*pathID, length, labelPath*);
(2) **Node** (*nodeID, pathID, order, parentID*);
(3) **Data** (*nodeID, pathID, order, parentID, value*).

Wang et al. [129] propose the following schema, where **ValueTable** stores the leaf nodes with the value. **NoValueTable** stores the inner nodes. The attribute *nodeID* is the node identifier number assigned by pre-order traversal.

(1) **ValueTable** (*nodeID, name, value, pathExp, parentID, level*);
(2) **NoValueTable** (*nodeID, name, parentID, level*).

Ying et al. [136] keep the parent-child relationship, path, and level information to support structural queries, especially for the twig query.

(1) **File** (*docID, docName*);
(2) **Path** (*pathID, pathExp*);
(3) **LeafNode** (*docID, leafNodeID, pathID, parentID, leafValue*);
(4) **InnerNodes** (*docID, innerNodeID, nodeName, parentID, level, sibling*).

Edge- & Path-Based Approach. ***Khan and Rao*** [70] propose the following schema to keep parent-child relationships and path information, where the attribute *pathExp*, considered as the primary key, is the simple path expression (from root to node) of these nodes.

(1) **SampleTable** (*pathExp, dataItem, parentPathExp*);
(2) **AttributeTable** (*pathExp, attributeName attributeValue*).

XPEV is proposed in [105], whose schema is proposed by combining edge [59] with path [71]:

(1) **Path** (*pathID, pathExp*);
(2) **Edge** (*pathID, source, target, label, ordinal, flag*);
(3) **Value** (*pathID, source, target, label, ordinal, value*).

Path- & Signature-Based Approach. It preserves path expressions (path-based method) in the table and assigns a different signature (number) to each distinctive label (node).

LNV is proposed in [50], where the attribute *pathNode* (*pathSignature*) is a list of nodes (signatures of labels) in the path ordered from the root. The attribute *value* is the value associated with

the end of the path. The attribute *typeNode* denotes the leaf node's type that can be an element, attribute, comment, or text. The attribute *position* records where the element node is among its sibling.

(1) **LabelsSignatures** (*label, signature*);
(2) **Path** (*docID, pathSignature, pathNode, value, typeNode, position*).

Pointer-Based Approach. It preserves as many the pointers of nodes' ancestors as possible.

XMLEase is proposed in [51], where some redundant edges are introduced into the XML tree so that each node is connected to its ancestors instead of just its parent. How many ancestors be connected for each node will depend on the number of ancestor columns in the pre-defined table.

(1) **Table** (*identifier*, $ancestor_1$, $ancestor_2$, ...)

The attribute *identifier* denotes labels (values) for intermediate nodes (leaves) of the XML tree. Other columns keep *identifier*'s ancestors. In this way, it could speed up hierarchical data's retrieval.

Token-Based Approach. It uses a table to record XML document structure information and uses another table to preserve token (element, tag, attribute, or property) information.

Dweib et al. [48] keep the XML document structure in the attribute *docStructure* (a big text field containing a coded string). Any changes (e.g., adding a new tag or deleting an existing property) in the structure should be recorded in this attribute.

(1) **Documents** (*docID, docStructure*);
(2) **Tokens** (*docID, tokenID, tokenName, tokenValue*).

MAXDOR is proposed in [49], which adopts a global label approach for identifying each token in an XML document and assigns additional labels (parent, left and right sibling) to each token for facilitating future inserting and relocating a given token. Besides, *MAXDOR* uses the table **Documents** to keep document information.

(1) **Documents** (*docID, docName, docElement, totalTokens, schemaInfo*);
(2) **Tokens** (*doctID, tokenID, lSib, parentID, rSib, tokenLevel, tokenName, tokenVal, tokenType*).

Edge- & Signature-Based Approach. Each element or attribute is identified by a signature (number) and each path is identified by its parent from the root node in a recursive manner.

XRecursive is proposed in [52, 53], whose schema is:

(1) **LabelStructure** (*labelName, signature, parentID*);
(2) **LabelValue** (*signature, value, type*).

Suri and Sharma [122] design the following schema:

(1) **Node** (*nodeID, nodeName*);
(2) **Data** (*docID, nodeID, parentID, nodeValue, nodeType, position*).

Labeling-Based Approach. It uses a labeling technique to annotate each node.

s-XML is proposed in [120], which adopts the Persistent Labeling [62] to annotate each node in the XML tree and stores those labels in the attribute *selfLabel*. In the following schema, **ParentTable** preserves the non-leaf (internal) nodes. **ChildTable** records the leaf (external) nodes.

(1) **ParentTable** (*nodeID, parentNodeName, NodeName, level, parentNodeID, selfLabel*);
(2) **ChildTable** (*nodeID, level, parentNodeName, selfLabel, parentNodeID, value*).

Path- & Labeling-Based Approach. It preserves path expressions in the table and adopts a labeling technology to annotate each node.

Table 4. Comparison among Model-Based Mapping Methods with a Non-Fixed Number of Tables

Time	Works	Contributions	Order Preserved	Empirically Validated
1999	*STORED* [44, 45]	Exploiting the regularities inherent in the semi-structured data to design schemas by the data mining	No	Yes
2009	*Kyu and Nyunt* [75]	Automatically creating the schema	No	Yes

XMap is proposed in [29], which uses ORDPATH labeling [97] (conceptually similar to the Dewey technique) to materialize the parent-child relationship, stores it in the attribute *ordpath*, and uses it to reflect a numbered tree edge of the path from the root to a node.

(1) **Data** (*ordpath, value, order, numberofElements, numberofAttributes, pathID*);
(2) **Node** (*nodeID, nodeName*);
(3) **Path** (*pathID, pathExp*).

XAncestor is proposed in [107], where the table **AncestorPath** stores the ancestor paths (root-to-parent) of the leaf nodes in the XML tree. The attribute *ancesPos* is a position of the ancestor for the leaf node, whose value is obtained by Dewey order labeling.

(1) **AncestorPath** (*ancesPathID, ancesPathExp*);
(2) **LeafNode** (*nodeName, ancesPathID, ancesPos, nodeValue*).

Mini-XML is proposed in [142], which adopts a persistent labeling approach to annotate leaf nodes. The specific format is (Level, [P-pathID, S-order]) stored in the attribute *pos*, where Level is the depth of the current leaf node in the XML tree, P-pathID is the path id of the direct parent node, and S-order is the order among its sibling.

(1) **Path** (*pathID, pathExp, pos*);
(2) **Leaf** (*leafID, name, value, pos*).

(2) Non-Fixed Schema. Next, we will introduce some works that map an XML document into a relational schema with a non-fixed number of tables and summarize these methods in Table 4.

STORED is proposed in [44, 45], which takes data instances as input and uses a heuristic algorithm, data-mining, to generate complex storage patterns with high combined support for creating tables. Each storage pattern keeps a pointer back to its subpattern with the highest data support, which is used to find the required attributes. Each semi-structured object having all the required attributes for a relational table will be stored in it. And the remaining attributes in this table may be filled with nulls. *STORED* uses created relational schemas to store most of the data. As for the outliers, parts of the semi-structured data that do not fit the generated schema or the possible future inserted data, are stored in a self-describing structure (overflow graph) to guarantee that the mapping and storing are lossless. Besides, *STORED* could take several parameters as input (e.g., the maximum number of relations allowed) to control generated relational schema.

Kyu and Nyunt [75] utilizes a data extraction approach to get a table name list, a table element list, a table attribute list, and the primary key of each table. Next, it uses those lists to create a relational schema and presents a data mapping algorithm to store XML data into relational tables.

Discussion. Compared to structure-based mapping, model-based approaches are widely studied since they are typically simple to implement, do not require extra schema information basically, and could have a better performance. Moreover, most model-based approaches could handle

dynamic XML documents whose DTDs change from time to time and support XML documents without any extension of the relational model. And there are many methods (edge, path, signature, labeling, pointer, token or combinations among these methods, etc.) supporting the model-based approach to map XML documents into relational tuples. Depending on adopted methods, they will create a varying number of tables. But the works introduced in the former have in common that they have a fixed schema, regardless of how XML document instances change. However, these methods also have their limitations. According to different methods taken, they may cause different performance variations. This is because some approaches may generate very complex SQL queries involving many joins for complex path expressions. For example, the edge method possibly has many self-joins when reconstructing a large XML document. Besides, the path method needs high storage space to keep path information. The pointer method holds more ancestor columns in the pre-defined relational table, and there will be more chances for "Null" pointers, thus wasting storage. The token method could not handle the complex semantic searches. The labeling method needs a larger space to store labeling when dealing with a large XML document. Therefore, except for introducing new techniques or combining different strategies to improve query performance, researchers also attempt to create relational schemas with non-fixed tables to fit XML document instances better. Unfortunately, these methods also inevitably store much structure information to reconstruct the original XML data and do not consider the importance of semantic information toward the relational schema, which could help reduce space consumption.

2.2.3 Semantic Information Approach. Recently, studies in the constraints of XML (e.g., keys and foreign keys) have caused an interest in using the semantic information to improve the generated relational schema's quality. In this part, we will introduce current researches in this area, classify it as the third category of the mapping approach, and summarize those works in Table 5.

CPI is proposed in [78, 79], which discovers semantic constraints hidden in DTDs and then rewrites the discovered constraints in relational notations. Since finding and preserving semantic constraints is independent of transformation algorithms, one could use other transformations instead of only the hybrid inlining algorithm.

XSchema is proposed in [86], which provides two normal form representations of regular tree grammars - NF1 and NF2. NF1 representation is used for document validation and schema validation. NF2 forms the basis for mapping type definitions in XML schema to SQL. Besides, this paper defines *XSchema*, a language-independent formalism to specify XML schemas. Next, it starts with simplifying *XSchema* to get a simpler *XSchema*, which does not have constraints that cannot be captured in the relational model. Then, it uses inlining [117] to generate relational schemas, maps collection types, stores *IDREF* and *IDREFS* attributes, handles recursion, captures the order specified in the XML model, and keeps constraints such as key constraints and inclusion dependencies.

RRXS is proposed in [36], which presents **XML functional dependencies (XFDs)** to capture structural as well as semantic information. It offers a set of rewriting rules to obtain redundancy-reduced XFDs in polynomial time. Then, *RRXS* translates optimized XFDs to relational functional dependencies and creates a **third normal form (3NF)** decomposition to guide the design of the target relational schema, where the generated schema is redundancy-reduced and has a set of keys.

Xshrex is proposed in [80], which is an extended *ShreX* by adding more constraints like structural choice, unique, key & foreign key, and domain constraints. Although these constraints need to be checked when doing insertions, deletions, and updates, it does not yield prohibitive costs. On the contrary, queries could utilize the index created based on the user-defined primary and foreign key constraints to improve performance.

X2R-Xing is proposed in [135], which starts by using a data structure called a marked schema tree to store the mapping from the DTD to a relational schema, where the node grouping algorithm

Table 5. Comparison among Mapping Methods with Semantic Information

	Time	Works	Contributions	Order Preserved	Empirically Validated
With DTD	2000 2001	*CPI* [78, 79]	Designing a relational schema with the hybrid inlining algorithm [117] and constraints-preserving algorithm	No	Yes
	2002	*XSchema* [86]	Mapping based on the theory of regular tree grammars	Yes	No
	2003	*RRXS* [36]	Reducing redundancy by using XFDs	No	Yes
	2006	*Xshrex* [80]	Extending *ShreX* [47] by holding integrity constraints in the XML schema	Yes	Yes
	2007	*X2R-Xing* [135]	A mapping system with range indexing and XML key constraints	Yes	Yes
	2010	*Castro et al.* [33]	Proposing a mapping mechanism using the conceptual model to maximize the preservation of semantics	Yes	Yes
	2011	*X2R-Ahmad* [8]	Designing a non-redundant relational schema with XFDs and DTD	No	Yes
W/O-DTD	2000	*Monet-XML-Model* [114]	Decomposing XML into small, flexible, and semantically homogeneous tables based on the binary associations	No	Yes
	2007	*Davidson et al.* [41]	Refining the design of the relational schema based on XML key propagation	No	Yes

generates the schema tree. Then the schema tree is used to shred XML documents into relational tuples. In this process, it indexes XML node groups based on range indexing. And it propagates key constraints for XML to keys in a relational schema.

Castro et al. [33] propose using the conceptual model as the intermediate schema for achieving the mapping. For establishing parallelism between two data models (i.e., XML and relation), they use a class diagram in **UML (Unified Modeling Language)**. This is because of the simplicity with which schemas modeled in UML can be mapped to relational databases. In this intermediate schema, DTD constructors are mapped into classes, and the relationships between them are presented in the form of associations in the UML diagram. The attribute *level* represents the nested levels for the main elements. The number of the appearance of an element is stored in the attribute *cardinality*. The logical operators in DTD are preserved in the attribute *operator*.

X2R-Ahmad is proposed in [8], which first obtains the XML structure from DTD and generates the DTD schema for describing XML. The expression of form about functional dependency for XML (XFD) is: (C, Q : X → Y), where C is the downward context path (defined by an XPath expression), Q is a target path, X is an **LHS (Left-Hand-Side)**, and Y is an **RHS (Right-Hand-Side)**. Next, it uses a constraint-preserving algorithm to remove redundant paths in XFD. It then maps the paths to attributes for obtaining a relational schema and several functional dependencies over this schema.

Monet-XML-Model is proposed in [114], which offers a data model (Monet-XML-Model) based on a complete binary fragmentation of the document tree to represent, store, and query all related

Table 6. Comparison among Mapping Methods with Cost-Driven Strategy

Time	Works	Contributions	Order Preserved	Empirically Validated
2002	*LegoDB* [24, 25]	Proposing a cost-based approach to find the optimal mapping in the solution domain for a specific scenario	No	Yes
2003	*Zheng et al.* [141]	Proposing a cost-driven approach to generate a near-optimal relational schema for a given XML data and expected workload in the limit of space	No	Yes
2003	*FlexMap* [109]	Generating efficient relational configurations for XML applications that suit an XML Schema with cost-based methods	No	Yes

associations (e.g., parent-child relationships, attributes, and topological orders) in the document. It applies paths to group semantically related associations into the same relation. In this way, related data can be accessed directly in the form of a separate table for a given query, avoiding large scans.

Davidson et al. [41] develop algorithms to find minimum cover functional dependencies from a set of XML keys on XML data through a given mapping (transforming an XML document to relational tables). With the functional dependencies, one could normalize the relational schema into, e.g., 3NF, BCNF to obtain efficient relational storage for XML data.

Discussion. When creating a relational schema, it is quite natural to consider all kinds of normal forms and integrities. Therefore, we think mapping XML to a relational schema with semantic information is more in line with our perception. However, most works in this field need DTD information (e.g., [33, 78, 79, 86]). Some methods may increase space consumption to keep redundant information (e.g., [80, 135]). And several approaches (e.g., [8, 114]) may create many simple tables, which will increase efforts to reconstruct the original document. Besides, those methods do not consider the importance of workloads (queries and data updates) toward the relational schema.

2.2.4 Cost-Driven Approach. Given the flexibility of XML, and the variety and complexity of transactions processed by XML applications, it's hard to say which of a structure-based approach and a model-based approach is better. The structure-based approaches take advantage of DTD to generate a specific relational schema for each XML document. However, this method may not get a "good" schema for arbitrary XML data having different complexity. What's more, there are some applications needing to deal with XML documents without DTDs. Therefore, model-based mapping is proposed. But this generic mapping limits the performance of relational schema. This is because the target relational schema is pre-defined and fixed, regardless of the XML schema. As a result, it is unlikely to work well for all possible applications. Therefore, next, we will review a cost-driven approach in this section, which could generate a near-optimal relational schema. We classify this approach as the fourth category of the mapping approach and summarize current works in Table 6.

LegoDB is proposed in [24, 25], which is a cost-based XML mapping system that takes an XML schema, an XQuery workload, and a set of sample documents as input, and outputs an efficient relational schema for a given application. In detail, *LegoDB* starts with extracting statistical information from the given XML documents. This information is used to derive relational statistics that are needed by the relational optimizer to estimate the cost of the query workload. Then, *LegoDB*

utilizes the XML schema and XML statistics to generate an initial physical schema (*p-schema*). Next, a set of *p-schema* rewritings are applied to the generated *p-schema* for getting a space of alternative *p-schema*. Based on a greedy heuristic, *LegoDB* explores an interesting subset of this space to find the best relational schema. In this process, *LegoDB* derives a relational schema from the *p-schema*, transforms XML statistics into relational statistics for the corresponding relational schema, translates the XQuery workload into the corresponding SQL equivalent, and uses a relational optimizer to obtain cost estimates.

Zheng et al. [141] firstly use an annotated schema graph to represent the XML schema. Thus, all of the possible partitioning schemes on the annotated schema graph consist of the solution space. The selection problem of the XML mapping schema can be regarded as the problem of the graph's optimal partition. It could use the Hill-Climbing algorithm to find the optimal solution in this solution space for an expected workload at a reasonable time. The Hill-Climbing algorithm starts from an initial schema generated by three approaches (*Attribute* mapping [58], *Shared,* and *Hybrid* mapping [117]). If one mapping schema is a state in the solution space, the algorithm tries to visit all the neighboring states that can be reached from the current state through state transformation defined by four primitive operations and uses the cost model to estimate the cost of executing the workload at the new state. Finally, the final state with minimal cost is returned as the optimal partitioning scheme, i.e., target relational schema.

FlexMap is proposed in [109], which defines a schema tree by several type constructors to represent an XML schema. A relational configuration could be derived from a schema tree. Suppose there is a set of transformations like inline and outline, type split/merge, commutativity, and associativity, and union distribution/factorization. As transformations are applied and new configurations are derived, *FlexMap* uses a cost model to estimate the cost for the query workload under each relational configuration. To find a nice configuration, *FlexMap* designs three greedy algorithms (InlineGreedy, ShallowGreedy, and DeepGreedy) to study how the quality of the final configuration is influenced by the choice of transformations and the query workload. In the end, *FlexMap* optimizes the DeepGreedy to get GroupGreedy by a grouping transformation concept and uses a small threshold of δ to accelerate processing (early terminate the iteration).

Discussion. The cost-driven approach uses a cost model or a relational optimizer to obtain cost estimates for each storage schema to find or generate an "optimal" relational schema. However, the problem is that we need to guarantee the accuracy of the cost model, which has a significant influence on the results [109, 141]. Besides, another problem is the generated schema that does not preserve too many constraints [24, 25]. Therefore, how to combine the cost-driven approach with semantic information to design a "good" relational schema is an interesting topic.

2.2.5 Other Studies on Mapping XML Documents to Relational Data. The research on mapping XML documents to relational data has many sub-problems that include but are not limited to building XML view on the relational schema to improve query performance, saving XML order, and translating XML query to SQL. Due to the space limitation, we list a few examples in Table B.1 (see Appendix B) to show research on these points.

2.3 Mapping JSON Documents to Relational Data

Argo is proposed in [34], which uses a vertical table format (three-column table for the object id, key, and value) [7] to solve the problem of sparse data representations in relational tables and uses a key-flattening technique to handle the hierarchical structure of JSON (i.e., objects and arrays). In detail, it appends the keys of a nested object to their parent key for forming the *Argo*'s table keys, where *Argo* uses the "." as an interval separator character. For arrays, each value is identified

by the table key (arrayKey[position]) after shredding JSON arrays into tables. *Argo* presents two schemas:

(1) *Argo/1* uses a single table to store JSON document:
    ❶ **Argo** (*objID, keyStr, valStr, valNum, valBool*).
(2) *Argo/3* uses three tables to store key-value pairs according to the value types ((long) number, string, and boolean):
    ❶ **ArgoStr** (*objID, keyStr, valStr*);
    ❷ **ArgoNum** (*objID, keyStr, valNum*);
    ❸ **ArgoBool** (*objID, keyStr, valBool*).

Bahta-Atay [21] propose **Single-Table Data Mapping (STDM)** and **Multi-Table Data Mapping (MTDM)** algorithms to map JSON documents into relational tuples, which are inspired by the universal and binary approaches [59]. STDM and MTDM algorithms build a JSON tree derived from the JSON document, where nodes represent JSON elements and edges represent parent-child relationships (see Figure 2(b)). Then, they store the JSON tree in the following tables:

(1) The STDM maps the JSON tree into a table:
    ❶ **STDM** (*elementID, parentElementID, elementName, valText, valNumeric*).
(2) The MTDM creates two separate tables to store non-leaf and leaf nodes, respectively:
    ❶ **NonLeafTable** (*elementID, parentElementID*);
    ❷ **LeafNodeTable** (*elementID, parentElementID, value*).

Irshad et al. [66] propose transforming JSON schema into a relational schema by parsing the JSON schema file, preserving the extracted information in the following metadata tables, creating relational tables with a vertical table approach by reading these two metadata tables, and storing the JSON data in the created tables.

(1) **RelationalStructureMasterTable** (*RS_ID, level, objectName, tableName, attributeCategory, parentLevel, pkColumn*);
(2) **RelationalStructureChildTable** (*RS_CID, attributeName, columnName, attributeDataType, required, RS_ID*).

DiScala and Abadi [46] present a three-phase unsupervised **machine learning (ML)** algorithm to automatically design a relational schema for an input JSON document. The first phase starts with transforming the JSON data into a flat table similar to the universal relation model. Next, it identifies "soft" functional dependencies among attributes. After that, it leverages them to decompose the previous flat table into a collection of smaller tables joined by primary-to-foreign key relationships. Each small table consists of a group of attributes that exhibit similar functional relationships and are likely to correspond to an independent semantic entity. The second phase searches these entities to discover semantically equivalent entities with overlapping attributes and merge them into a single entity to eliminate redundant storage. The third phase combines the intermediate tables produced from the previous two phases to create a relational schema for avoiding excessive normalization.

DVP is proposed in [118], which presents **Dynamic Vertical Partitioning (DVP)** technology utilizing heuristics to adapt the data layout for the workload dynamically. DVP groups some attributes accessed together in queries into the same partition (smaller table) by an algorithm with polynomial complexity. This dynamic partition is based on two criteria: awareness of workload access patterns and data sparseness awareness. Specifically, when the *DVP* is invoked, it starts with the current layout (or an initial partitioning) and generates a new layout by incrementally refining the current schema. At each iteration, *DVP* examines all existing attributes and partitions. For each attribute-partition pair, *DVP* calculates the gain of moving the attribute to the specific partition. When there is no further cost improvement, *DVP* stops and returns a new layout.

Petković [101] proposes using the following schema to store JSON documents, where it assigns a unique ID for each element and preserves its corresponding parent ID.

(1) **Table** (*elementID, parentID, objectID, keyName, value, valueType*).

Discussion. The model-based approach is a generic mapping having a predefined fixed schema, which has been widely studied in the field of XML-to-Relation mapping. It is especially suitable for JSON data. This is because JSON data come without a schema [19]. The other advantage of this approach is it could handle the dynamic feature of JSON documents. However, it is inevitable to have limitations, just like in the domain of XML-to-Relation mapping. For example, works [21, 34] need recursive joins when it processes a complex query, which affects application performance. Recently, some people have concentrated on deducing a meaningful schema for JSON data [19, 20, 27, 32, 61]. We could apply this technique [18] to find schema information of JSON data and employ this information to guide the design of relational schemas. Then, we could store JSON data in the relational tables and empower JSON to use RDBMSs for analysis and complex queries (e.g., [66]). Based on this idea, we may be able to draw lessons from the structured-based approach in the field of XML-to-Relation mapping. We think this is an interesting research topic.

The unsupervised ML approach wants to transform JSON documents into relational data automatically. It expects to identify the structures implied in semi-structured documents and extracts them to create relational schemas. However, this is not easy to generate a good schema with matching algorithms discovering semantic entities or with analysis tools gaining a semantic understanding of complex data. For example, although the work of *DiScala and Abadi* [46] could simplify the cognitively tricky task of exploring new JSON documents by highlighting recurring structural and semantic patterns, the generated schema often contradicts original expectations. Besides, the method in [46] does not support functional dependencies with multiple attributes on the left-hand side, and it does not consider all structural information relevant to input data.

The cost-based approach provides a flexible way to adapt the data layout for the workload dynamically. Although this approach could use a cost model to evaluate each storage schema to find or generate an "optimal" relational schema that achieves predefined goals, the generated schema may not be a "good" one. For example, the approach of *Bahta-Atay* [46] is able to achieve a better cache utilization and TLB utilization, but it may create a large number of small tables.

XML and JSON are the main representatives of semi-structured data. Therefore we review how to map them into relational schemas in previous sections. Since mapping JSON documents to relational tables is a new research topic, it does not have many references (see Table 7). But in other words, this means there is still a lot of room for improvement on this topic.

3 MAPPING GRAPH DATA INTO RELATIONAL DATA

The graph model is a natural way of representing linked data. It gains more and more popularity in the database community as the growth of linked data on the web and the broad applications of social networks, web graphs, geographical networks, and so on. This is because people put more and more attention on the relationship among the objects. Graphs allow increasingly interconnected networks to be visualized in a straightforward way to catch crucial information. These benefits make various applications built on graph data. However, this leads to another problem, how to store and query increasing graph data efficiently. Considering the difficulty and cost of developing an new native graph database, many application developers resort to RDBMSs to store graphs. For graph data models, there are about two ways to store their data instances in RDBMSs. One is to adopt the external data type - **BLOB (Binary Large Objects)** - to keep unstructured binary large objects such as property graphs. BLOB data types have full transactional support. Its value manipulations can be committed or rolled back. However, the BLOB has a maximum limitation, i.e., 4 gigabytes of binary data. We have to reassemble and/or disassemble BLOB whenever accessing it. Another is to design a good schema layout for the storage of graph data in RDBMSs. Since the storage scheme based on RDBMSs is currently the primary storage method for the graph data, our emphasis is on this strategy. As for the first approach, interested readers may refer to Appendix C.

Table 7. Comparison among Methods of Mapping JSON into Relational Data

	Time	Works	Contributions	No. of Tables	Empirically Validated
Model-Based	2013	*Argo* [34]	Proposing a mapping layer to make RDBMSs support the flexibility of the JSON data model	1 or 3	Yes
	2019	*Bahta-Atay* [21]	Proposing two JSON-to-Relation mapping algorithms: STDM and MTDM, according to model-based approaches	1 or more	Yes
Structure-Based	2019	*Irshad et al.* [66]	Proposing using the descriptive nature of JSON schema to create a relational schema	-	Yes
ML-Based	2016	*DiScala and Abadi* [46]	Proposing an unsupervised machine learning algorithm to design relational schemas	-	Yes
Cost-Driven	2019	*DVP* [118]	Proposing an architecture-aware technique to adapt the relational data layout for workloads dynamically	-	Yes
Adjacency List	2020	*Petković* [101]	Proposing a general method for storing hierarchical data and comparing it with the approach of STDM [21]	-	Yes

To store graph data in relational tables, we need to create a relational schema. We know that the design of a relational schema is guided by finding the regularity or uniformity of datasets. Unfortunately, the *a priori* uniformity causes difficulties for modeling a dynamic scene (e.g., social networks) [55]. This is because the primary goal of the RDF/property graph is to handle non-regular or unstructured data. And the fundamental cause of this hardness is a conflict between *a priori* regularity demanded by the relational model and the irregularity of the graph data model. Due to the conflict between these two data models, it is essential to consider the following points when designing a "good" schema for storing the graph data in relational tables.

(1) Guarantee the information integrity;
(2) Handle the scalability for large graph stores;
(3) Support the dynamics of graph structure;
(4) Achieve efficiency for query and update operations;
(5) Accommodate multi-valued properties;
(6) Adapt to data sparsity for reducing space consumption.

The critical demand for storing graph data in RDBMSs is holding the whole relevant data to guarantee information integrity. As more and more large-scale datasets are linked together, some datasets may consist of billions of nodes or more. These data might be frequently updated online, mostly by adding new nodes and edges. We believe that an efficient relational storage scheme for graph data should offer scalability and support dynamics in its data management system. And we should keep the response time for updates, especially query operations, under the acceptable range on the available hardware to maintain excellent efficiency. Besides, we might meet a situation in a graph dataset where a subject is associated with several objects by the same property. That is, such property has distinct values. The designed schema should have the ability to handle this multi-valued case. Lastly, we should notice the data sparsity problem and avoid storing too many

Table 8. Methods about Mapping Graph into Relational Tables

		Time	Works	Contributions	No. of Tables
RDF Graph	Triples Table	2002	*Jena* [89]	Normalizing the triples table to store RDF data	3
		2003	*3store* [64]	Normalizing the triples table by a hash technology	4
		2003	*Sesame* [31]	Proposing an architecture independently from platforms to keep RDF data and schema information	13
		2010	*RDF-3X* [96]	Using a single "giant triples table" to store RDF data	1
	Property Table	2003 2006	*Jena2* [130, 131]	Introducing property tables and property-class tables to store RDF data for improving query performance	-
		2005	*Chong et al.* [37]	Proposing a compact storage format and using SP-MJVs to speed up specific types of queries	2
		2009	*Data-Centric* [81]	Presenting a two-phase algorithm consisting of clustering and partitioning to create schema	-
	Path	2005	*Matonoy et al.* [88]	Proposing a path-based relation schema	6
	VP	2007	*Abadi et al.* [4]	Dividing a triples table into several two-column tables to store RDF data	-
	Entity	2013	*DB2RDF* [26]	Using a mixed schema having k-ary and binary tables	4
	DRL	2021	*GSBRL* [140]	Learning an adaptive relational schema for various data and workloads	-
PG	Column	2015	*GRAPHITE* [99]	Proposing a framework (GRAPHITE) as a central graph processing component inside RDBMSs	2

NULL values in the relational tables for reducing space consumption. This section will review the current mapping approaches to show their development and summarize them in Table 8.

3.1 The Preliminaries of RDF Graph and Property Graph

3.1.1 RDF Graph. RDF is a schema-less and self-describing (the graph's labels within the graph describe the data itself) data format. It is common to use RDF to describe various types of metadata. One typical usage is to describe large-scale metadata, such as ontologies, dictionaries, and data dictionaries. RDF data is a collection of statements (i.e., triples) where each triple is defined as *subject-predicate-object (s-p-o)* and interpreted as "a subject s has a relationship p with object o, or a subject s has a predicate (or property) p with value o". From a formalized perspective, a triple is $(s, p, o) \in (U \cup B) \times U \times (U \cup B \cup L)$, where U (representing **Uniform Resource Identifier, URI)**, B (denoting blank node), and L (expressing literal) are disjoint infinite sets. This is, a subject must be a URI or a blank node; a predicate (property) is always a URI; an object can be any of these data types ($U/B/L$). Besides, a collection of triples can be represented as a directed graph connecting resource nodes and their property values by labeled arcs. The graph structure of RDF is called the *edge-labeled graph* [134] in which labels are added to edges to indicate the different types of relationships (see Figure 3). An edge-labeled graph G is a pair of (V, E), where V is a finite set of

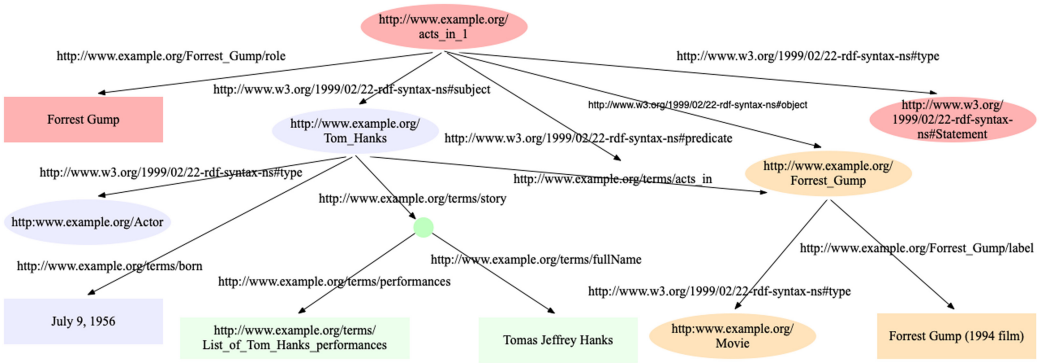

Fig. 3. An Example of RDF graph.

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:abbre="http://www.example.org/terms/"
  xmlns:ae="http://www.example.org/Forrest_Gump/">
  <rdf:Description rdf:about="http://www.example.org/Tom_Hanks"
                   abbre:born="July 9 1956">
    <rdf:type rdf:resource="http:www.example.org/Actor" />
   <abbre:story>
    <rdf:Description abbre:fullName="Tomas Jeffrey Hanks">
     <abbre:performances rdf:resource="http://www.example.org/terms/List_of_Tom_Hanks_performances/" />
    </rdf:Description>
   </abbre:story>
   <abbre:acts_in>
    <rdf:Description rdf:about="http:www.example.org/Forrest_Gump"
                     ae:label="Forrest Gump(1994 film)">
     <rdf:type rdf:resource="http:www.example.org/Movie" />
    </rdf:Description>
   </abbre:acts_in>
  </rdf:Description>
  <rdf:Description rdf:about="http://www.example.org/acts_in">
   <rdf:subject rdf:resource="http://www.example.org/Tom_Hanks" />
   <rdf:predicate rdf:resource="http://www.example.org/terms/acts_in" />
   <rdf:object rdf:resource="http://www.example.org/Forrest_Gump" />
   <rdf:type rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#Statement"/>
   <ae:role>Forrest Gump</ae:role>
  </rdf:Description>
</rdf:RDF>
```

Fig. 4. One possible serialization of the RDF graph (Figure 3) in XML syntax.

vertices (or nodes), and E is a finite set of edges, $E \subseteq V \times Lab \times V$, Lab is a set of labels. Syntactically, the RDF graph could also be represented by an XML syntax. One possible serialization of the RDF data (Figure 3) in XML syntax looks like the description of Figure 4. Structurally, we could parse the RDF into a series of triples and store them in RDBMSs. Therefore, many researchers dedicate themselves to designing a "good" relational schema for storing and querying RDF data.

3.1.2 Property Graph. As another commonly used graph-based data model, the property graph is defined as a directed labeled graph where each vertex or edge could have an arbitrary number of property-value pairs (see Figure 5). And the key-value pairs of a vertex (or an edge) could be encapsulated in an object, exhibiting an object-oriented view of graphs. Therefore, after being introduced by Rodriguez-Neubauer [111], the property graph has been extensively used by graph database systems like Neo4j [76] and Sparksee.[1] These specialized graph databases for graph analysis lie in a broader enterprise ecosystem where there are some already existing data processing platforms (e.g., RDBMSs) for carrying out "traditional" data analysis jobs [54]. Therefore, users

[1] http://sparsity-technologies.com/#sparksee.

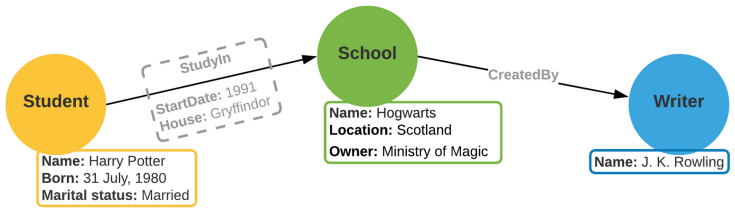

Fig. 5. An Example of property graph.

could directly use RDBMSs to manage graph data instead of spending decades developing a new database. Of course, the graph engines have their benefits (e.g., affording a vertex-centric form of graph programming, which is intuitive for the end (graph analytics) application developer to use). However, with a syntactic layer on top of SQL, RDBMSs could also provide much of this programmer's convenience [54]. Those, coupled with the powerful and mature data management services of RDBMSs, remind people it is time to reconsider using RDBMSs to manage graph data.

3.1.3 The Differences between RDF Graph and Property Graph.

(1) **Function.** RDF is more about data exchange and property graph about storage and query.
(2) **Definition.** Property graph has no concept of URIs or entailments. But, it allows direct association of properties (key-value pairs) with edges. RDF, by contrast, needs reification or a quad data model to associate properties with edges.
(3) **Structure.** The vertices and edges of the property graph could have an internal structure (key-value pairs). For RDF, neither vertices nor edges have this; they are purely unique labels.

Due to the structure complexity, the property graph faces more challenges in storing edge and vertex in RDBMSs. Maybe we could transform the property graph into an RDF graph [65], and then keep the new gotten graph in RDBMSs with RDF-to-Relational technologies.

3.2 Mapping RDF Graph to Relational Data

3.2.1 Triples Table. The first category approach of providing persistent storage for RDF data in RDBMSs is to store statements in triples tables. One of the straightforward implementations is using a giant triples table (i.e., a three-column table) to preserve RDF data as a linearized list of triples (*subject-predicate-object*). To avoid storing too much redundant information, there are many variations on this approach.

Jena is proposed in [89], which normalizes the triples table by storing literals and URIs in separate tables so that they are stored only once. In the following schema, the table **Literal** keeps all literal values, and **Resource** holds all URIs.

(1) **Statement** (*subject, predicate, uriID, literalID*);
(2) **Literal** (*literalID, value*);
(3) **Resource** (*resourceID, uri*).

3store is proposed in [64], which normalizes the triples table by hashing the resource URIs and literal values as foreign keys. In the following schema, the attributes *flagLiteral*, *flagUri* are boolean values to indicate whether the object is a literal value or a URI.

(1) **Triples** (*model, subject, predicate, object, flagLiteral, flagURI*);
(2) **Models** (*hash, model*);
(3) **Resources** (*hash, uri*);
(4) **Literals** (*hash, value*).

Sesame is proposed in [31], which presents the following architecture to preserve RDF data and schema information. To reduce storage cost, *Sesame* encodes resource and literal via an integer

value (the id field). The attribute *isDerived* is added into some tables, which is to encode whether a statement was explicitly asserted or derived from the schema information.

(1) **Triples** (*subject, predicate, object, isDerived*);
(2) **Property** (*propertyID, isDerived*);
(3) **Range** (*propertyID, class, isDerived*);
(4) **Domain** (*propertyID, class, isDerived*);
(5) **SubClassOf** (*subclass, superclass, isDerived*);
(6) **Class** (*classID, isDerived*);
(7) **Namespaces** (*namespaceID, prefix, name*);
(8) **Resources** (*resourceID, namespace, localName*);
(9) **Type** (*resourceID, class, isDerived*);
(10) **SubPropertyOf** (*subprop, superprop, isDerived*);
(11) **Labels** (*resourceID, literal, isDerived*);
(12) **Comment** (*resourceID, literal, isDerived*);
(13) **Literals** (*literalID, language, value*).

RDF-3X is proposed in [96], which is a workload-independent schema (i.e., a single "giant triples table" with appropriate indexes). Considering that triples may include long string literals, it uses a mapping dictionary to replace all literals with ids. As a triples table would incur expensive self-joins, *RDF-3X* address this problem by creating the "right" (appropriate) index set and using merge joins.

Discussion. The triples table could handle dynamic RDF by inserting statements directly into the table without considering RDF data types. Since URIs and literal values tend to be long strings, one efficient way to reduce space consumption is not to store the entire string but to keep shortened versions or keys in the table (e.g., *Oracle* [98], *Jena* [89], *3store* [64], *Sesame* [31], and *RDF-3X* [96] map strings to integer identifiers). Though flexible, this schema may cause a scalability issue as the quantity of RDF data grows fast. This is because it uses a giant triples table to store RDF data, and almost all interesting queries require many expensive self-joins over this table.

3.2.2 Property Table. Based on the RDF data's regularity (frequent patterns), the second category method introduces the property table concept to store several related properties together in a table. In this approach, a single tuple may include numerous RDF statements.

Jena2 is proposed in [130], which uses a property table to keep all the subject-object pairs related to a particular property (predicate). That is, this property would not appear in any other tables. *Jena2* clusters multiple properties about a common subject together to form a property table. Besides, a special property table (the property-class table) is proposed to hold all instances of a specified class and reserve properties of that class. The approach of [131] gives the following three property table formats. The table **SingleValuedPropertyTable** records values for one or more properties that have a maximum cardinality of one. **MultipleValuedPropertyTable** stores a single property that has a maximum cardinality greater than one (or unknown). And **PropertyClassTable** stores all members (single-valued properties) of a class together.

(1) **SingleValuedPropertyTable** (*subject,* $prop_1$, $prop_2$, . . . , $prop_n$);
(2) **MultipleValuedPropertyTable** (*subject, property*);
(3) **PropertyClassTable** (*subject,* $prop_1$, $prop_2$, . . . , $prop_n$, *type*).

Chong et al. [37] propose a compact storage format where RDF data is stored (after normalization) in the following two tables. In the table **IDTriples**, triples are recorded in the identifier format, which avoids storing URIs (or literals) repeatedly. The table **URIMap** holds the mappings from uriIDs to URIs (literals). Besides, a class of materialized views called **subject-property matrix materialized join views (SPMJVs)** is adopted to speed up specific types of queries over RDF triples, where the subject-property matrix is a property table-like data structure.

Table 9. The Features and Differences among Different types of Property Tables

Name	Features	Differences
Clustered Property Table	Each table includes a cluster of properties that tend to be accessed together frequently.	A particular property may only appear in at most one table.
Property-Class Table	Each table groups resembling sets of subjects based on the property *rdf:type* and stores them together.	A property may exist in multiple property-class tables. Any single-valued property may be stored in this table, i.e., not just those that declare the class in their domain. *Jena2* [130, 131] also uses it as the storage of reified statements.
Subject-Property Matrix	Each table consists of a set of single-valued properties that occur together. These properties can be direct properties of subjects or nested properties.	This table is used as an auxiliary data structure (i.e., a materialized view) instead of a primary storage structure.

(1) **IDTriples** (*modelID, subjectID, propertyID, objectID,...*);
(2) **URIMap** (*uriID, valueURI,...*).

Data-Centric is proposed in [81], which presents a two-phase algorithm consisting of clustering and partitioning to create relational schema. The clustering phase scans the whole RDF data to cluster all properties for generating several groups. Each group, which is made up of properties frequently appearing together, is a candidate *n*-ary table. Properties not in clusters may be stored in binary tables. The partitioning phase takes clusters as input and determines whether or not to remove some properties for balancing the trade-off between holding as many properties as possible in a table and reducing NULL storage to a minimum (i.e., below a given threshold). *Data-Centric* also handles the multi-valued properties problem and reification storage. The final relational schema is a balanced mix of binary (i.e., decomposed storage [39, 40]) and *n*-ary tables (i.e., property tables) based on the data structure.

Discussion. To distinguish the property tables that have been introduced, we summarize their features and differences in Table 9. The property table method offers several advantages over the triples table method. Initially, having multiple tables in a schema is more like a general relational schema, which makes access to legacy data stored in RDBMSs quite natural. Besides, it may improve performance through a better locality and caching. Next, the use of numerous tables may make better use of the query optimizer. Finally, this approach simplifies database administration since the different tables can be separately managed. For the property tables, they could derive from the ontology of the dataset (e.g., Sesame [31]). Of course, these tables could also be defined by the applications (e.g., Jena2 [130]). However, these definitions must be provided when the graph is initially created, which makes this method lose some flexibility. And data sparsity results in many NULL values in the property table method. Furthermore, multi-valued attributes are slightly inconvenient to present in a flattened format. But unfortunately, it is common to see multi-valued attributes in various RDF datasets, which causes the complexity of designing schema.

3.2.3 Path-Based Mapping. The approaches, storing RDF data in statement formats in RDBMSs, would require many join operations when doing path-based queries. Therefore, path-based

mapping is proposed for handling those queries efficiently by keeping schema information as well as path expressions of each resource in tables.

Matonoy et al. [88] divide the RDF graph into subgraphs and then extracts different information from these subgraphs to fill the following proposed path-based relation schema. In this schema, the attributes *pre*, *post*, and *depth* express node numbers created by the extended interval numbering scheme. The values of these attributes are calculated from the **Class Inheritance (CI)** graphs. The attributes *domain* and *range*, calculated from the **Domain-Range (DR)** graphs, define the domain and range of a property. The RDF instance data in the **Generic (G)** graph is recorded by using three tables **Resource**, **Path**, and **Triple**, where **Resource** collects every node in graph G, **Path** extracts and holds all absolute arc-path expressions for each node, and **Type** stores Type (T) graphs that associate the RDF schema with RDF instances.

(1) **Class** (*className, pre, post, depth*);
(2) **Property** (*propertyName, domain, range, pre, post, depth*);
(3) **Resource** (*resourceName, pathID, dataType*);
(4) **Triple** (*subject, predicate, object*);
(5) **Path** (*pathID, pathExp*);
(6) **Type** (*resourceName, className*).

Discussion. Since the RDF data structure is a directed graph, most of the queries for RDF data can be regarded as subgraphs matching or finding a set of nodes that can be reached via given path expressions. These queries, represented in path expressions, need many join operations when RDF is stored in statement formats. Path-based schema could efficiently reduce the number of join operations. However, this approach would increase space cost due to keeping path expressions.

3.2.4 Vertical Partitioning (VP) Approach. For taming the scalability problem and avoiding using clustering algorithms, the vertical partitioning approach uses a fully **decomposed storage model (DSM)** to preserve RDF data. We also call it a predicate-oriented approach.

Abadi et al. [4] divide a triples table into several two-column tables whose amounts are equal to the quantity of distinct RDF properties. For each table, one attribute preserves the subjects having that property, and the other attribute holds the corresponding object values. With this schema, listing each distinct value in a successive row could address the problem of multi-valued attributes. Besides, to locate specific subjects quickly and use fast merge joins, tuples in these tables are sorted by subjects. Of course, the value column could also be optionally indexed.

Discussion. The proposal of RDF is not equal to a kind of physical storage. As a logical data model, we do not need to store collections of triples on disks. The vertical-partitioning approach creates distinct two-column tables for each property. The advantage of this schema is able to support heterogeneous records, especially for non-well-structured data. Besides, due to a query only accessing the involved properties, I/O costs are greatly reduced. However, when a query involves one subject's numerous properties, it has to merge corresponding two-column tables. As more and more new predicates appear, this would result in a large number of small tables.

3.2.5 Entity-Oriented Approach. This approach could make relational schema have flexibility (handling dynamic RDF schemas) and scalability (handling most complex queries efficiently for a large of RDF data) by using a mix of horizontal tables and binary tables.

DB2RDF is proposed in [26], which attempts to preserve all the predicates for a given entity on a single row while handling the inherent variability of different entities. **DirectPrimaryHash** is a wide table where the attribute *entry* keeps the subject s, each pair of predicate$_i$ and value$_i$ ($0 \leq i \leq k$) preserves s' associated predicates and objects. If s has more than k predicates, new tuples are used to store the additional attributes until covering and storing all the predicates for s.

DirectSecondaryHash is a binary table that is used to store multi-valued predicates. The tables **DirectPrimaryHash** and **DirectSecondaryHash** hold the outgoing edges of an entity (from *s* to the predicates). For efficient access, *DB2RDF* also encodes the incoming edges of an entity (object values of the predicate to subject *s*) with **ReversePrimaryHash** and **ReverseSecondaryHash**.

(1) **DirectPrimaryHash** (*entry, spill, predicate*$_1$, *value*$_1$, . . . , , *predicate*$_k$, *value*$_k$);
(2) **DirectSecondaryHash** (*valueID, element*);
(3) **ReversePrimaryHash** (*entry, spill, predicate*$_1$, *value*$_1$, . . . , , *predicate*$_{k'}$, *value*$_{k'}$);
(4) **ReverseSecondaryHash** (*valueID, element*).

Discussion. Due to using the wide table, the entity-oriented approach could reduce the number of joins when queries look for multiple predicates for the same subject or object. Besides, using a column to store various predicates could efficiently save space. Otherwise, we have to use as many columns as predicates to keep the whole RDF data. Of course, if possible, storing all the instances of a predicate in the same column could take advantage of all the index benefits of relational representations. However, as new data flow in databases, the original *k* may be unsuitable.

3.2.6 Deep Reinforcement Learning-Based Approach. The sixth approach category is to use **deep reinforcement learning (DRL)** to design an adaptive storage structure that fits various datasets and workloads. This approach allows users to obtain an optimal relational schema by interacting with the environment without requiring prior experience.

GSBRL is proposed in [140], which takes a dataset (stored in a single triples table) and a workload (a set of SQL statements rewritten from SPARQL statements) as input to find the optimal schema in RDBMSs. Firstly, **GSBRL** vectorizes the data storage features to encode the storage state of the current tables. Then, it uses **Double Deep Q-Network (DDQN)** as a training model to interact with the environment (database). The DDQN selects actions (dividing or merging tables) to train the network. In the process of training, the database returns the query time to generate the reward. After the training, **GSBRL** could find the final schema according to the trained deep neural network.

Discussion. Compared to current work using fixed rules, DRL-Based approach could generate a more reasonable and adaptive storage structure. However, it may require a large number of queries to assist in training the deep neural networks for obtaining a reasonable relational schema. This is inevitable to result in time-consuming.

3.3 Mapping Property Graph to Relational Data

3.3.1 Column-Oriented Approach. This approach utilizes the column group concept to create a flexible relational schema for handling dynamic graph data.

GRAPHITE is proposed in [99], which is an extensible graph traversal framework, worked as a central graph processing unit inside RDBMSs. It provides an extensible set of logical graph traversal operators and their corresponding implementations. It also offers two traversal implementations (i.e., **level-synchronous (LS)** traversal and **fragmented-incremental (FI)** traversal) to support various graph topologies and different graph traversal queries efficiently. This framework operates on the following physical column group schema. In this schema, each table is a column group, which could handle the new attribute insertion problem by appending a new column to the column group.

(1) **Vertex** (*vID, attribute*$_1$, *attribute*$_2$, ...);
(2) **Edge** (*vID*$_{start}$, *vID*$_{terminate}$, *attribute*$_1$, *attribute*$_2$, ...).

Discussion. This approach could easily handle updates of the property graph. Furthermore, it could use run-length-based compression techniques [3] to compress NULL values in sparsely

populated columns for saving space consumption. However, this approach is not friendly for star queries (i.e., queries involve multiple attributes for the same vertex or edge).

3.4 Other Technologies for Storing Graph Data in RDBMSs

There are also other approaches to store the property graph data in the RDBMSs. For example, we could use an adjacency (i.e., entity-oriented) approach [26] to hold all the adjacency edges of a vertex on the same row as much as possible while using JSON to store attribute values together for eliminating joins [113, 121]. For more detail, interested readers may refer to Appendix D.

4 OPEN PROBLEMS AND FUTURE RESEARCH DIRECTIONS

Although various approaches have been proposed to enable an RDBMS to manage semi-structured data and graph data without extension, the landscape of efforts is fragmented, with no clear view of which approach is the best and what open problems we should address in this field. To help towards this direction, we identify and summarize the following open challenges:

- **The trade-off between space consumption and query performance.** It is hard to balance data sparsity and query complexity when storing semi-structured or graph data in RDBMSs. If relational tables consist of few columns that are highly correlated, these narrow tables would have a higher average value density. That is, these tables have fewer NULL values. But it may result in a single query involving multiple table join operations. In contrary, if tables are made wide, they may include many NULL values [5].
- **Adaptability to fit dynamic data and workload.** The evolving structure of semi-structured data and graph data and workload's variety and dynamics make storing these data in RDBMSs difficult. This is because there is a conflict between the fixed relational schema and new appearing properties (nonexistent attribute in tables).
- **Scalability on handling growing data size.** We use scalability to measure an RDBMS's ability to handle a growing amount of SQL operations by adding data to the designed relational schema. The goal of the research aims to provide a "good" schema to store semi-structured data and graph data in RDBMSs so that users could efficiently perform various SQL operations over this schema. But this scalability problem is compounded by ever-increasing volumes of data to be maintained in tables. Therefore, we need to notice the scalability problem when designing a relational schema to store semi-structured or graph data.

This survey presents a comprehensive review, analysis, and discussion of the existing approaches attempting to address the problems as mentioned above. Each approach outshines in one or more aspects, having its unique application scenarios. According to the existing works, we identify some future research directions:

(1) Without the knowledge of schema information (a common situation in practice), the model-based mapping approach is the most approachable for XML and JSON documents. Considering that graph data also has no schema, the model-based method might be applied to the field of mapping RDF or PG data into relational data. Therefore, the model-based approach is well worthy of further studying.
(2) The adaptive approaches, dynamically adjusting relational schema to ensure required performance, are more like what we need today. This is because such an approach can work well for any long-running workload. However, this approach heavily relies on the cost model. Thus, it introduces another research problem about how to define an appropriate cost function.
(3) **Artificial intelligence (AI)** could leverage computer science and data to handle tasks (e.g., schema design) that typically require human intelligence. It could relieve the users from

such a tedious task. Different from most approaches (e.g., structure-based, model-based, and cost-driven), AI is a data-driven technique, which could utilize data to generate an optimal schema for fitting various datasets and workloads. Therefore, we think this would be an attractive research direction in the future.

(4) As big data applications increase in size and complexity, one application may produce data having multiple formats. These data might have certain relationships instead of being independent of each other. For managing multi-model data in a unified platform, multi-model databases are proposed [83]. But they are still not as mature as traditional RDBMSs. Thus, this introduces a new research topic - how to use RDBMSs to manage multi-model data.
   - A viable approach is first to map multi-model data into a unified intermediate format (e.g., map RDF into XML, see Figure 3 and 4) and then leverage prior technologies to map this intermediate format to relational data.
   - Another feasible way is using AI technology to directly learn a relational schema from workloads to store multi-model data. Interested readers can refer to our latest work [137, 138] which employs the reinforcement learning method to generate a relational schema for multi-model datasets consisting of relational, RDF, and JSON.

5 CONCLUSION

Since RDBMS has many powerful artificial services, it fuels more and more interest in using mature RDBMSs to manage various data. This paper's primary goal is to review and introduce the existing literature on mapping semi-structured data and graph data into relational data. Instead of investigating a single specific data model, we cover the currently most popular data models and study how to map each of them into relational data model. With the development of research, we may expand it by applying the model-based method to the field of mapping RDF or PG data into relational data, exploring the adaptive approaches to dynamically adjust relational schema to ensure required performance, adopting AI techniques to generate relational schema, or attempting to map multi-model data into relational data, and so on. This review is essential because it provides useful insights into the current state of the art in this field, identifies open problems for both researchers and practitioners, and motivates new research topics towards this research direction.

APPENDICES

A ALTERNATIVE WAYS OF SUPPORTING SEMI-STRUCTURED DATA IN RDBMSS

The XML data type [1] is an internal representation in SQL Server, which is a bit like *int*, *varchar*, and other built-in types. Users could use it as a column type and define column-level or table-level constraints. They could also use XQuery or the **XML Data Manipulation Language (XML DML)** to update the XML instances, where the XML DML is the extension of XQuery. Besides, users could use an XML constant or an explicit cast to the XML type for assigning a default XML instance. Overall, the XML data type could make it convenient if XML data have complicated structures that are not easy to transform into tables.

Large objects (LOBs) [98] could store semi-structured data as a whole in RDBMSs instead of mapping a document into several smaller tables. Applications concerning semi-structured documents usually deal with massive volumes of character data. *LOBs* datatypes like *CLOB* and *NCLOB* are suitable for saving and operating such data. Oracle8i and later versions suggest processing semi-structured data with *LOBs*. This is because a table could keep multiple *LOBs*, and each *LOB* could preserve 4GB data or more.

Petković [100] investigates how RDBMSs like Oracle, PostgreSQL, and SQL Server integrate JSON documents. Oracle stores JSON documents by using data types *VARCHAR2*, *CLOB*, and *BLOB*. PostgreSQL supports two data types, *JSON* and *JSONB*, for storing JSON documents, where *JSON*

Table B.1. Other Studies on Mapping XML to Relational Schema

Time	Works	Contributions
2000 2001	*SilkRoute* [56, 57]	Proposing a general, dynamic, and efficient tool, *silkRoute*, for viewing and querying relational data in XML
2001	*Shanmugasundaram et al.* [116]	Reconstructing an XML view on the relational schema
2002	*Tatarinov et al.* [125]	Proposing three order encoding approaches to record XML orders and convert ordered XPath expressions into SQL statements
2003	*DeHaan et al.* [43]	Proposing dynamic intervals, an encoding based on an interval representation of XML data that enables relational engines to execute arbitrarily nested XQuery FLWR expressions
2003	*Krishnamurthy et al.* [74]	Formalizing the problem of finding optimal relational mapping for the XML workload and exploring the problem complexity
2005	*VLEI* [72]	Proposing the VLEI code and applying it to XML labeling to reduce the cost of the insertion operation
2007	*ID-XMLToSQL* [17]	Proposing an approach - translating XML query into SQL statements - that is suitable for both single- and multi-valued schema mappings

stores an exact copy of the input text, and *JSONB* stores data in a decomposed binary form. SQL Server keeps JSON documents with the *NVARCHAR* data type. Oracle integrates JSON according to the ANSI SQL/JSON standard and implements the most concepts specified in the standard. PostgreSQL has not implemented any features specified in the standard due to its implementation before the standard. SQL Server uses the standard to define the data type for JSON documents and realizes a few standardized features.

Liu et al. [82] extend RDBMSs to use the document-object-store model for storing JSON objects natively, where each JSON object instance is stored in an aggregated form without decomposition (i.e., it is self-contained). This aggregated storage not only facilitates the import/export of JSON data but eliminates the cost of reconstructing the original JSON data.

B OTHER STUDIES ON MAPPING XML TO RELATIONAL DATA

Table B.1 shows some other relevant researches on mapping XML documents into relational data.

C THE ALTERNATIVE WAYS OF SUPPORTING GRAPH DATA IN RDBMSS

Binary Large Objects (BLOB)[2] are designed to store large objects in RDBMSs efficiently. With *BLOB*, RDBMSs could directly store the property-value pairs of property graph as an attribute (e.g., [15, 54, 110]), which avoids dividing property-values of a vertex/edge into smaller pieces. However, the disadvantage is that we have to reassemble and/or disassemble *BLOB* whenever accessing one of these property-value pairs. For example, ***Linkbench*** [15] designs two tables (one for graph nodes and one for graph edges) to store property graph data. Instead of creating an attribute for each property, it uses the attribute *data* (*BLOB*) to store relevant properties. And it leaves the responsibility of extracting specific attributes from the *BLOB* to the application.

(1) **Vertex** (*vID, type, data, ...*);
(2) **Edge** (vID_{start}, $vID_{terminate}$, *type, data, ...*).

[2]https://docs.oracle.com/cd/E17276_01/html/api_reference/C/blob.html.

D OTHER TECHNOLOGIES FOR STORING GRAPH DATA INTO RELATIONAL TABLE

Sun et al. [121] design the following relational schema, inspired by [26], that combines relational storage for adjacency information and JSON storage for vertex and edge attributes.

(1) **OutgoingPrimaryAdjacency** (*vID, spill,* eID_1, $label_1$, $value_1$, . . . , eID_k, $label_k$, $value_k$);
(2) **IncomingPrimaryAdjaccency** (*vID, spill,* eID_1, $label_1$, $value_1$, . . . , eID_q, $label_q$, $value_q$);
(3) **OutgoingSecondaryAdjacency** (*valueID, eID, value*);
(4) **IncomingSecondaryAdjacency** (*valueID, eID, value*);
(5) **VertexAttributes** (*vID, attribute (JSON object)*);
(6) **EdgeAttributes** (*eID, inVertex, outVertex, label, attribute(JSON object)*).

ACKNOWLEDGMENTS

We would also like to thank all the reviewers and editors for their constructive comments and valuable suggestions. Their comments and suggestions have substantially helped us to improve the quality of this survey.

REFERENCES

[1] SQL Server 2019. Accessed November 27, 2020. SQL Docs. (Accessed November 27, 2020). https://docs.microsoft.com/en-us/sql/relational-databases/xml/xml-data-type-and-columns-sql-server?view=sql-server-ver15/.
[2] SQL Server 2022. Accessed February 6, 2022. TiDB Docs. (Accessed February 6, 2022). https://docs.pingcap.com/tidb/stable.
[3] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. 671–682.
[4] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. 2007. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd International Conference on Very Large Data Bases*. 411–422.
[5] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. 2009. SW-Store: A vertically partitioned DBMS for semantic web data management. *The VLDB Journal* 18, 2 (2009), 385–406.
[6] Fethi Abduljwad, Wang Ning, and Xu De. 2010. SMX/R: Efficient way of storing and managing XML documents using RDBMSs based on paths. In *2010 2nd International Conference on Computer Engineering and Technology*, Vol. 1. IEEE, V1–143.
[7] Rakesh Agrawal, Amit Somani, and Yirong Xu. 2001. Storage and querying of e-commerce data. In *VLDB*, Vol. 1. 149–158.
[8] Kamsuriah Ahmad. 2011. A comparative analysis of managing XML data in relational database. In *Asian Conference on Intelligent Information and Database Systems*. Springer, 100–108.
[9] AllegroGraph. Accessed February 19, 2022. AllegroGraph. (Accessed February 19, 2022). https://allegrograph.com/.
[10] T. Amagasa, Masatoshi Yoshikawa, Dao Dinh Kha, T. Shimizu, and K. Fujimoto. 2005. A mapping scheme of XML documents into relational databases using schema-based path identi. ers. In *International Workshop on Challenges in Web Information Retrieval and Integration*. IEEE, 82–90.
[11] Sihem Amer-Yahia, Fang Du, and Juliana Freire. 2004. A comprehensive solution to the XML-to-relational mapping problem. In *Proceedings of the 6th Annual ACM International Workshop on Web Information and Data Management*. 31–38.
[12] Sihem Amer-Yahia, Fang Du, and Juliana Freire. 2004. A generic and flexible framework for mapping XML documents into relations. In *VLDB'04: Proceedings of 30th International Conference on Very Large Data Bases*.
[13] ArangoDB. 2020. *What is a Multi-model Database and Why Use It?* [White Paper] (2020).
[14] Marcelo Arenas and Leonid Libkin. 2005. An information-theoretic approach to normal forms for relational and XML data. *Journal of the ACM (JACM)* 52, 2 (2005), 246–283.
[15] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. LinkBench: A database benchmark based on the Facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1185–1196.
[16] Mustafa Atay, Artem Chebotko, Dapeng Liu, Shiyong Lu, and Farshad Fotouhi. 2007. Efficient schema-based XML-to-Relational data mapping. *Information Systems* 32, 3 (2007), 458–476.
[17] Mustafa Atay, Artem Chebotko, Shiyong Lu, and Farshad Fotouhi. 2007. XML-to-SQL query mapping in the presence of multi-valued schema mappings and recursive XML schemas. In *International Conference on Database and Expert Systems Applications*. Springer, 603–616.

[18] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2019. Parametric schema inference for massive JSON datasets. *The VLDB Journal* 28, 4 (2019), 497–521.
[19] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2019. Schemas and types for JSON data: From theory to practice. In *Proceedings of the 2019 International Conference on Management of Data*. 2060–2063.
[20] Mohamed-Amine Baazizi, Houssem Ben Lahmar, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2017. Schema inference for massive JSON datasets. In *Extending Database Technology (EDBT)*.
[21] Rahwa Bahta and Mustafa Atay. 2019. Translating JSON data into relational data using schema-oblivious approaches. In *Proceedings of the 2019 ACM Southeast Conference*. 233–236.
[22] Andrey Balmin and Yannis Papakonstantinou. 2005. Storing and querying XML data using denormalized relational databases. *The VLDB Journal* 14, 1 (2005), 30–49.
[23] BaseX. Accessed November 27, 2020. BaseX. (Accessed November 27, 2020). https://basex.org/.
[24] Philip Bohannon, Juliana Freire, Jayant R. Haritsa, Maya Ramanath, Prasan Roy, and Jérôme Siméon. 2002. LegoDB: Customizing relational storage for XML documents. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 1091–1094.
[25] Philip Bohannon, Juliana Freire, Prasan Roy, and Jérôme Siméon. 2002. From XML schema to relations: A cost-based approach to XML storage. In *Proceedings 18th International Conference on Data Engineering*. IEEE, 64–75.
[26] Mihaela A. Bornea, Julian Dolby, Anastasios Kementsietsidis, Kavitha Srinivas, Patrick Dantressangle, Octavian Udrea, and Bishwaranjan Bhattacharjee. 2013. Building an efficient RDF store over a relational database. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 121–132.
[27] Pierre Bourhis, Juan L. Reutter, Fernando Suárez, and Domagoj Vrgoč. 2017. JSON: Data model, query languages and schema specification. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 123–135.
[28] Ronald Bourret et al. 1999. XML and databases. (1999).
[29] Zakaria Bousalem and Ilias Cherti. 2015. XMap: A novel approach to store and retrieve XML document in relational databases. *JSW* 10, 12 (2015), 1389–1401.
[30] Tim Bray, Jean Paoli, C. Michael Sperberg-McQueen, Eve Maler, Franois Yergeau, et al. 2000. Extensible markup language (XML) 1.0. (2000).
[31] Jeen Broekstra, Arjohn Kampman, and Frank Van Harmelen. 2003. Sesame: An architecture for storing and querying RDF data and schema information. *Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential* 197 (2003).
[32] Javier Luis Cánovas Izquierdo and Jordi Cabot. 2013. Discovering implicit schemas in JSON data. In *International Conference on Web Engineering*. Springer, 68–83.
[33] Elena Castro, Dolores Cuadra, and Manuel Velasco. 2010. From XML to relational models. *Informatica* 21 (2010), 505–519.
[34] Craig Chasseur, Yinan Li, and Jignesh M. Patel. 2013. Enabling JSON document stores in relational systems. In *WebDB*, Vol. 13. 14–15.
[35] S. Chaudhuri and Kyuseok Shim. 2003. Storage and retrieval of XML data using relational databases. In *Proceedings 19th International Conference on Data Engineering (Cat. No. 03CH37405)*. IEEE, 802–802.
[36] Yi Chen, Susan Davidson, Carmem Hara, and Yifeng Zheng. 2003. RRXS: Redundancy reducing XML storage in relations. In *Proceedings 2003 VLDB Conference*. Elsevier, 189–200.
[37] Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan. 2005. An efficient SQL-based RDF querying scheme. In *Proceedings of the 31st International Conference on Very Large Data Bases*. 1216–1227.
[38] Edgar F. Codd. 2002. A relational model of data for large shared data banks. In *Software Pioneers*. Springer, 263–294.
[39] George P. Copeland and Setrag N. Khoshafian. 1985. A decomposition storage model. *ACM SIGMOD Record* 14, 4 (1985), 268–279.
[40] John Corwin, Avi Silberschatz, Perry L. Miller, and Luis Marenco. 2007. Dynamic tables: An architecture for managing evolving, heterogeneous biomedical data in relational database management systems. *Journal of the American Medical Informatics Association* 14, 1 (2007), 86–93.
[41] Susan Davidson, Wenfei Fan, and Carmem Hara. 2007. Propagating XML constraints to relations. *J. Comput. System Sci.* 73, 3 (2007), 316–361.
[42] Ali Davoudian, Liu Chen, and Mengchi Liu. 2018. A survey on NoSQL stores. *ACM Computing Surveys (CSUR)* 51, 2 (2018), 1–43.
[43] David DeHaan, David Toman, Mariano P. Consens, and M. Tamer Özsu. 2003. A comprehensive XQuery to SQL translation using dynamic interval encoding. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. 623–634.
[44] Alin Deutsch, Mary Fernandez, and Dan Suciu. 1999. Storing semistructured data in relations. In *Proceedings of the Workshop on Query Processing for Semistructured Data and Non-standard Data Formats*.

[45] Alin Deutsch, Mary Fernandez, and Dan Suciu. 1999. Storing semistructured data with STORED. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*. 431–442.
[46] Michael DiScala and Daniel J. Abadi. 2016. Automatic generation of normalized relational schemas from nested key-value data. In *Proceedings of the 2016 International Conference on Management of Data*. 295–310.
[47] Fang Du, Sihem Amer-Yahia, and Juliana Freire. 2004. ShreX: Managing XML documents in relational databases. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases-Volume 30*. 1297–1300.
[48] Ibrahim Dweib, Ayman Awadi, Seif Elduola Fath Elrhman, and Joan Lu. 2008. Schemaless approach of mapping XML document into Relational Database. In *2008 8th IEEE International Conference on Computer and Information Technology*. IEEE, 167–172.
[49] Ibrahim Dweib, Ayman Awadi, and Joan Lu. 2009. MAXDOR: Mapping XML document into relational database. *Open Information Systems Journal* 3, 1 (2009), 108–122.
[50] Mohamed E. El-Sharkawi and N. A. El-Hadi El Tazi. 2005. LNV: Relational database storage structure for XML documents. In *The 3rd ACS/IEEE International Conference on Computer Systems and Applications, 2005.* IEEE, 49.
[51] Atilla Elçi and Behnam Rahnama. 2006. XMLEase: A novel access-and space-efficiency model for maintaining XML data in relational databases. In *SWWS*. 186–192.
[52] Mohammed Adam Ibrahim Fakharaldien, Khalid Edris, Jasni Mohamed Zain, and Norrozila Sulaiman. 2012. Mapping extensible markup language document with relational database management system. *International Journal of Physical Sciences* 7, 25 (2012), 4012–4025.
[53] Mohammed Adam Ibrahim Fakharaldien, Jasni Mohamad Zain, and Norrozila Sulaiman. 2012. XRecursive: An efficient method to store and query XML documents. *CoRR* abs/1203.6454 (2012). arXiv:1203.6454
[54] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M. Patel. 2015. The case against specialized graph analytics engines. In *CIDR*.
[55] David C. Faye, Olivier Cure, and Guillaume Blin. 2012. A survey of RDF storage approaches. *Revue Africaine de la Recherche en Informatique et Mathématiques Appliquées* 15 (2012), 11–35.
[56] Mary Fernandez, Atsuyuki Morishima, and Dan Suciu. 2001. Efficient evaluation of XML middle-ware queries. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*. 103–114.
[57] Mary Fernández, Wang-Chiew Tan, and Dan Suciu. 2000. SilkRoute: Trading between relations and XML. *Computer Networks* 33, 1-6 (2000), 723–745.
[58] Daniela Florescu and Donald Kossmann. 1999. A performance evaluation of alternative mapping schemes for storing XML data in a relational database. (1999).
[59] Daniela Florescu and Donald Kossmann. 1999. Storing and querying XML data using an RDMBS. *IEEE Data Engineering Bulletin* 22 (1999), 3.
[60] Thomas Frisendal. Accessed February 19, 2022. Metadata Recycling into Graph Data Models. (Accessed February 19, 2022). https://leanpub.com/metawrangling/.
[61] Angelo Augusto Frozza, Ronaldo dos Santos Mello, and Felipe de Souza da Costa. 2018. An approach for schema extraction of JSON and extended JSON document collections. In *2018 IEEE International Conference on Information Reuse and Integration (IRI)*. IEEE, 356–363.
[62] Alban Gabillon and Majirus Fansi. 2005. A persistent labelling scheme for XML and tree databases. In *SITIS*. 110–115.
[63] Gang Gou and Rada Chirkova. 2007. Efficiently querying large XML data repositories: A survey. *IEEE Transactions on Knowledge and Data Engineering* 19, 10 (2007), 1381–1403.
[64] Stephen Harris and Nicholas Gibbins. 2003. 3store: Efficient bulk RDF storage. (2003).
[65] Olaf Hartig. 2014. Reconciliation of RDF* and property graphs. *arXiv preprint arXiv:1409.3288* (2014).
[66] Lubna Irshad, Li Yan, and Zongmin Ma. 2019. Schema-based JSON data stores in relational databases. *Journal of Database Management (JDM)* 30, 3 (2019), 38–70.
[67] Haifeng Jiang, Hongjun Lu, Wei Wang, and Jeffrey Xu Yu. 2002. Path materialization revisited: An efficient storage model for XML data. In *Australasian Database Conference*, Vol. 5. Citeseer.
[68] Yasser Abdel Kader, Barry Eaglestone, and Siobhán North. 2008. An analysis of relational storage strategies for partially structured XML. In *WEBIST (1)*. 165–170.
[69] Gerti Kappel, Elisabeth Kapsammer, and Werner Retschitzegger. 2004. Integrating XML and relational database systems. *World Wide Web* 7, 4 (2004), 343–384.
[70] Latifur Khan and Yan Rao. 2001. A performance evaluation of storing XML data in relational database management systems. In *Proceedings of the 3rd International Workshop on Web Information and Data Management*. 31–38.
[71] Won Kim. 2001. XRel: A path-based approach to storage and retrieval of XML documents using relational databases. *ACM Transactions on Internet Technology (TOIT)* 1, 1 (2001), 110–141.
[72] Kazuhito Kobayashi, Wenxin Liang, Dai Kobayashi, Akitsugu Watanabe, and Haruo Yokota. 2005. VLEI code: An efficient labeling method for handling XML documents in an RDB. In *21st International Conference on Data Engineering (ICDE'05)*. IEEE, 386–387.

[73] Solmaz Kolahi and Leonid Libkin. 2007. XML design for relational storage. In *Proceedings of the 16th International Conference on World Wide Web*. 1083–1092.
[74] Rajasekar Krishnamurthy, Venkatesan T. Chakaravarthy, and Jeffrey F. Naughton. 2003. On the difficulty of finding optimal relational decompositions for XML workloads: A complexity theoretic perspective. In *International Conference on Database Theory*. Springer, 270–284.
[75] Zin Mar Kyu and Thi Thi Soe Nyunt. 2009. Storing DTD-independent XML data in relational database. In *2009 IEEE Symposium on Industrial Electronics & Applications*, Vol. 1. IEEE, 197–202.
[76] Mahesh Lal. 2015. *Neo4j Graph Data Modeling*. Packt Publishing Ltd.
[77] Ora Lassila, Ralph R. Swick, et al. 1998. Resource description framework (RDF) model and syntax specification. (1998).
[78] Dongwon Lee and Wesley W. Chu. 2000. Constraints-preserving transformation from XML document type definition to relational schema. In *International Conference on Conceptual Modeling*. Springer, 323–338.
[79] Dongwon Lee and Wesley W. Chu. 2001. CPI: Constraints-preserving inlining algorithm for mapping XML DTD to relational schema. *Data & Knowledge Engineering* 39, 1 (2001), 3–25.
[80] Qiuju Lee, Stéphane Bressan, and Wenny Rahayu. 2006. XShreX: Maintaining integrity constraints in the mapping of XML schema to relational. In *17th International Workshop on Database and Expert Systems Applications (DEXA'06)*. IEEE, 492–496.
[81] Justin J. Levandoski and Mohamed F. Mokbel. 2009. RDF data-centric storage. In *2009 IEEE International Conference on Web Services*. IEEE, 911–918.
[82] Zhen Hua Liu, Beda Hammerschmidt, and Doug McMahon. 2014. JSON data management: Supporting schema-less development in RDBMS. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. 1247–1258.
[83] Jiaheng Lu and Irena Holubová. 2019. Multi-model databases: A new journey to handle the variety of data. *ACM Computing Surveys (CSUR)* 52, 3 (2019), 1–38.
[84] Shiyong Lu, Yezhou Sun, Mustafa Atay, and Farshad Fotouhi. 2003. A new inlining algorithm for mapping XML DTDs to relational schemas. In *International Conference on Conceptual Modeling*. Springer, 366–377.
[85] Hooran MahmoudiNasab and Sherif Sakr. 2010. An experimental evaluation of relational RDF storage and querying techniques. In *International Conference on Database Systems for Advanced Applications*. Springer, 215–226.
[86] Murali Mani and Dongwon Lee. 2002. XML to relational conversion using theory of regular tree grammars. In *Efficiency and Effectiveness of XML Tools and Techniques and Data Integration over the Web*. Springer, 81–103.
[87] MarkLogic. Accessed February 19, 2022. MarkLogic. (Accessed February 19, 2022). https://www.marklogic.com/.
[88] Akiyoshi Matono, Toshiyuki Amagasa, Masatoshi Yoshikawa, and Shunsuke Uemura. 2005. A path-based relational RDF database. In *Proceedings of the 16th Australasian Database Conference-Volume 39*. Citeseer, 95–103.
[89] Brian McBride. 2002. Jena: A semantic web toolkit. *IEEE Internet Computing* 6, 6 (2002), 55–59.
[90] Yan Men-hin and Ada Wai-chee Fu. 2001. From XML to relational database. In *Proc. of the KRDB*.
[91] Irena Mlynkova and Jaroslav Pokorný. 2007. Adaptability of methods for processing XML data using relational databases-the state of the art and open problems. *Int. J. Comput. Sci. Appl.* 4, 2 (2007), 43–62.
[92] MonetDB. Accessed February 20, 2022. MonetDB. (Accessed February 20, 2022). https://www.monetdb.org/.
[93] MongoDB. Accessed February 19, 2022. MongoDB. (Accessed February 19, 2022). https://www.mongodb.com/.
[94] Maya Mourya and Preeti Saxena. 2015. Survey of XML to relational database mapping techniques. *Adv. Comput. Sci. Inf. Technol. (ACSIT)* 2 (2015), 162–166.
[95] Neo4j. Accessed February 19, 2022. Neo4j. (Accessed February 19, 2022). https://neo4j.com/.
[96] Thomas Neumann and Gerhard Weikum. 2010. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal* 19, 1 (2010), 91–113.
[97] Patrick O'Neil, Elizabeth O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. 2004. ORDPATHs: Insert-friendly XML node labels. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. 903–908.
[98] Oracle. Accessed November 27, 2020. Oracle Database Online Documentation 10g Release 2 (10.2). (Accessed November 27, 2020). https://docs.oracle.com/cd/B19306_01/appdev.102/b14249/adlob_intro.htm#i1009016/.
[99] Marcus Paradies, Wolfgang Lehner, and Christof Bornhövd. 2015. GRAPHITE: An extensible graph traversal framework for relational database management systems. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*. 1–12.
[100] Dušan Petković. 2017. JSON integration in relational database systems. *Int. J. Comput. Appl.* 168, 5 (2017), 14–19.
[101] Dušan Petković. 2020. Non-native techniques for storing JSON documents into relational tables. In *Proceedings of the 22nd International Conference on Information Integration and Web-based Applications & Services*. 16–20.
[102] Dusan Petkovic and Ali Piriyaie. 2021. Shredding JSON data into relational environment. *International Journal of Computer Applications* 174 (2021), 25–29.

[103] Sandeep Prakas, Sourav S. Bhowmick, and Sanjay Madria. 2004. SUCXENT: An efficient path-based approach to store and query XML documents. In *International Conference on Database and Expert Systems Applications*. Springer, 285–295.

[104] Sandeep Prakash, Sourav S. Bhowmick, and Sanjay Madria. 2004. Efficient recursive XML query processing in relational database systems. In *International Conference on Conceptual Modeling*. Springer, 493–510.

[105] Jie Qin, Shumei Zhao, Shuqiang Yang, and Wenhua Dou. 2005. Efficient storing well-formed XML documents using RDBMS. In *Proceedings of ICSSSM'05. 2005 International Conference on Services Systems and Services Management, 2005.*, Vol. 2. IEEE, 1075–1080.

[106] Amjad Qtaish and Kamsuriah Ahmad. 2015. Model mapping approaches for XML documents: A review. *Journal of Information Science* 41, 4 (2015), 444–466.

[107] Amjad Qtaish and Kamsuriah Ahmad. 2016. XAncestor: An efficient mapping approach for storing and querying XML documents in relational database using path-based technique. *Knowledge-Based Systems* 114 (2016), 167–192.

[108] Amjad Qtaish and Mohammad T. Alshammari. 2019. A narrative review of storing and querying XML documents using relational database. *Journal of Information & Knowledge Management* 18, 04 (2019), 1950048.

[109] Maya Ramanath, Juliana Freire, Jayant R. Haritsa, and Prasan Roy. 2003. Searching for efficient XML-to-relational mappings. In *International XML Database Symposium*. Springer, 19–36.

[110] Christine F. Reilly. 2020. Parallel traversal of graphs stored in RDBMSs. In *CIDR*.

[111] Marko A. Rodriguez and Peter Neubauer. 2010. Constructions from dots and lines. *Bulletin of the American Society for Information Science and Technology* 36, 6 (2010), 35–41.

[112] Sherif Sakr and Ghazi Al-Naymat. 2010. Relational processing of RDF queries: A survey. *ACM SIGMOD Record* 38, 4 (2010), 23–28.

[113] Matthias Schmid. 2019. On efficiently storing huge property graphs in relational database management systems. In *Proceedings of the 21st International Conference on Information Integration and Web-based Applications & Services*. 344–352.

[114] Albrecht Schmidt, Martin Kersten, Menzo Windhouwer, and Florian Waas. 2000. Efficient relational storage and retrieval of XML documents. In *International Workshop on the World Wide Web and Databases*. Springer, 137–150.

[115] Zülal Şevkli, Mine Mercan, and Atakan Kurt. 2004. A middleware approach to storing and querying XML documents in relational databases. In *International Conference on Advances in Information Systems*. Springer, 223–233.

[116] Jayavel Shanmugasundaram, Eugene Shekita, Jerry Kiernan, Rajasekar Krishnamurthy, Efstratios Viglas, Jeffrey Naughton, and Igor Tatarinov. 2001. A general technique for querying XML documents using a relational database system. *ACM SIGMOD Record* 30, 3 (2001), 20–26.

[117] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. 1999. Relational databases for querying XML documents: Limitations and opportunities. In *Proceedings of the 25th International Conference on Very Large Data Bases*. 302–314.

[118] Sahel Sharify, Alan Lu, Jin Chen, Arnamoy Bhattacharyya, Ali Hashemi, Nick Koudas, and Cristiana Amza. 2019. An improved dynamic vertical partitioning technique for semi-structured data. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 243–256.

[119] Manu Sporny, Dave Longley, Gregg Kellogg, Markus Lanthaler, and Niklas Lindström. 2014. JSON-LD 1.0. *W3C Recommendation* 16 (2014), 41.

[120] Samini Subramaniam, Su-Cheng Haw, and Poo Kuan Hoong. 2012. s-XML: An efficient mapping scheme to bridge XML and relational database. *Knowledge-Based Systems* 27 (2012), 369–380.

[121] Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guotong Xie. 2015. SQLGraph: An efficient relational-based property graph store. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1887–1901.

[122] Pushpa Suri and Divyesh Sharma. 2012. A model mapping approach for storing XML documents in relational databases. *International Journal of Computer Science Issues (IJCSI)* 9, 3 (2012), 495.

[123] Pushpa Suri and Divyesh Sharma. 2013. Schema based storage of XML documents in relational databases. *International Journal on Web Service Computing* 4, 2 (2013), 23.

[124] Daniel Tahara, Thaddeus Diamond, and Daniel J. Abadi. 2014. Sinew: A SQL system for multi-structured data. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. 815–826.

[125] Igor Tatarinov, Stratis D. Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. 2002. Storing and querying ordered XML using a relational database system. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*. 204–215.

[126] Feng Tian, David J. DeWitt, Jianjun Chen, and Chun Zhang. 2002. The design and performance evaluation of alternative XML storage strategies. *ACM SIGMOD Record* 31, 1 (2002), 5–10.

[127] Yannis Velegrakis. 2010. Relational technologies, metadata and RDF. In *Semantic Web Information Management*. Springer, 41–66.

[128] Saurabh Vyas and Shruti Kolte. 2014. A review paper for mapping of XML data to relational table. *International Journal of Computer Science and Mobile Computing* 3, 12 (2014), 327–332.
[129] Qi Wang, Zhongwei Ren, Liang Dong, and Zhongqi Sheng. 2012. Path-based XML relational storage approach. *Physics Procedia* 33 (2012), 1621–1625.
[130] Kevin Wilkinson, Craig Sayers, Harumi A. Kuno, Dave Reynolds, et al. 2003. Efficient RDF storage and retrieval in Jena2. In *SWDB*, Vol. 3. Citeseer, 131–150.
[131] Kevin Wilkinson and Kevin Wilkinson. 2006. Jena property table implementation. (2006).
[132] Jun Wu and Shang Yi Huang. 2008. XPred: A new model-mapping-schema-based approach for efficient access to XML data. In *Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services*. 123–130.
[133] Jun Wu and Shang-Yi Huang. 2009. An efficient mapping schema for storing and accessing XML data in relational databases. *International Journal of Web Information Systems* (2009).
[134] Marcin Wylot, Manfred Hauswirth, Philippe Cudré-Mauroux, and Sherif Sakr. 2018. RDF data storage and query processing schemes: A survey. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–36.
[135] Guangming Xing, Zhonghang Xia, and Douglas Ayers. 2007. X2R: A system for managing XML documents and key constraints using RDBMS. In *Proceedings of the 45th Annual Southeast Regional Conference*. 215–220.
[136] Jie Ying, Suyan Cao, and Yuan Long. 2012. An efficient mapping approach to store and query XML documents in relational database. In *Proceedings of 2012 2nd International Conference on Computer Science and Network Technology*. IEEE, 2140–2144.
[137] Gongsheng Yuan and Jiaheng Lu. 2021. MORTAL: A tool of automatically designing relational storage schemas for multi-model data through reinforcement learning. *ER2021 Demos and Posters* (2021).
[138] Gongsheng Yuan, Jiaheng Lu, Shuxun Zhang, and Zhengtong Yan. 2021. Storing multi-model data in RDBMSs based on reinforcement learning. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 3608–3611.
[139] Hasan Zafari, Keramat Hasani, and M. Ebrahim Shiri. 2010. XLight, an efficient relational schema to store and query XML data. In *2010 International Conference on Data Storage and Data Engineering*. IEEE, 254–257.
[140] Lei Zheng, Ziming Shen, and Hongzhi Wang. 2021. GSBRL: Efficient RDF graph storage based on reinforcement learning. *World Wide Web* (2021), 1–22.
[141] Shihui Zheng, Ji-Rong Wen, and Hongjun Lu. 2003. Cost-driven storage schema selection for XML. In *Eighth International Conference on Database Systems for Advanced Applications, 2003. (DASFAA 2003). Proceedings*. IEEE, 337–344.
[142] Huchao Zhu, Huiqun Yu, Guisheng Fan, and Huaiying Sun. 2017. Mini-XML: An efficient mapping approach between XML and relational database. In *2017 IEEE/ACIS 16th International Conference on Computer and Information Science (ICIS'17)*. IEEE, 839–843.

Received 4 October 2021; revised 22 September 2022; accepted 29 September 2022