Distributed and Parallel Databases (2024) 42:31–71
https://doi.org/10.1007/s10619-023-07433-1

Multi-model query languages: taming the variety of big data

Qingsong Guo[1,2] · Chao Zhang[3] · Shuxun Zhang[2] · Jiaheng Lu[2]

Accepted: 20 April 2023 / Published online: 31 May 2023

Abstract

A critical issue in Big Data management is to address the variety of data–data are produced by disparate sources, presented in various formats, and hence inherently involves multiple data models. Multi-Model DataBases (MMDBs) have emerged as a promising approach for dealing with this task as they are capable of accommodating multi-model data in a single system and querying across them with a unified query language. This article aims to offer a comprehensive survey of a wide range of multi-model query languages of MMDBs. In particular, we first present the SQL-based extensions toward multi-model data, including the standard SQL extensions such as SQL/XML, SQL/JSON, and GQL, and the non-standard SQL extensions such as SQL++ and SPASQL. We then study the manners in which document-based and graph-based query languages can be extended to support multi-model data. We also investigate the query languages that provide native support on multi-model data. Finally, this article provides insights into the open challenges and problems of multi-model query languages.

Keywords Multi-model data · Query language · Cross-model query

✉ Qingsong Guo
qingsongg@gmail.com

✉ Jiaheng Lu
jiaheng.lu@helsinki.fi

Chao Zhang
cycchao@mail.tsinghua.edu.cn

Shuxun Zhang
shuxun.zhang@helsinki.fi

1 School of Computer Science & Technology, North University of China, No.3 of Xueyuan Road, Taiyuan 030051, Shanxi, China

2 Department of Computer Science, University of Helsinki, P.O. Box 68, Pietari Kalmin katu 5, 00560 Helsinki, Finland

3 Department of Computer Science, Tsinghua University, 30 Shuangqing Road, Beijing 100084, China

1 Introduction

In past decades, we were witnessing the burst of heterogeneous data, where data may be produced by disparate sources, presented in various formats (structured, semi-structured, or unstructured), and hence inherently involves multiple data models. Take a healthcare dataset Mimic II [1] as an example, it encompasses 26,000 patients/days in the intensive care unit (ICU) of Beth Israel Hospital in Boston. This dataset includes data collected from disparate sources: (1) real-time data (time series from bedside monitoring devices); (2) a historical archive of waveform data (from previous patients); (3) patient metadata (relational data); (4) doctor's and nurse's notes (text); and (5) prescription information (semi-structured data). Relational data is only a small portion of this dataset. To make the right treatment decisions, doctors have to check the real-time diagnosis and request the historical treatments of the patients as well.

The demands for efficient management of massive multi-model data have triggered the development of Multi-Model DataBase (MMDB) systems [2, 3]. Conventionally, every database must adhere to a specific data model, which determines the logical structure of data, and the manner in which data is stored, organized, and manipulated by database systems. The relational model has become the dominant data model of databases since its inception in 1970 [4]. However, many of the relational DBMSs gradually evolved into their multi-model versions with support to the lately invented data models such as XML, JSON, graph, and key-value data. We have found 124 MMDB systems listed on the DB-Engines Ranking site[1] (414 DBMSs in total). An MMDB tightly integrates multiple storage engines together and accommodates data in the formats that fit the sources best, e.g., key/value pairs, relational tables, graphs, XML/JSON documents, etc. It also provides a unified query language to express users' interests across multiple data models.

In this article, we present a comprehensive investigation of the query languages of MMDBs, as shown in Table 1. Syntactically, these languages can be roughly divided into four categories: the SQL extensions, the XPath/XQuery extensions, the graph extensions, and the native ones. The most appealing feature of these query languages is the capability of expressing cross-model queries—a query spans multiple data models. Therefore, such language allows users to express relational queries, document queries, graph queries, key/value lookups, and arbitrary mixtures of them in a single query. By relational queries, we mean queries that manipulate relations in accordance with relational algebra/calculus. By document queries, we mean queries that navigate through a document from the root to any particular node. By graph query, we mean queries that involve the particular connectivity features coming from the edges, e.g., shortest path, graph traversal, and pattern matching.

Related work To the best of our knowledge, this is the first survey to discuss state-of-the-art research works and industrial products on multi-model query languages. A number of related surveys have been published in past years, but most of them focused on the general issues in multi-model data management or query languages for specific data models, especially for graph models. Scholl [5] investigates the restrictions on the relational model and the ways to extend the model to

[1] DB-Engines Ranking (Apr 2023). https://db-engines.com/en/ranking

Table 1 Multi-model query languages

Category	Languages	Data models	Description
SQL Extensions	SQL/XML	Relation, XML	A part of SQL:2003; Examples: PL/SQL, T-SQL, PL/pgSQL, Db2, MySQL, etc
	SQL/JSON	Relation, JSON	A part of SQL:2016; Examples: PL/SQL, T-SQL, PL/pgSQL, Db2, MySQL, etc
	SQL++	Relation, JSON	Extend SQL with path expressions; Examples: AsterixDB's SQL++, Couchbase's N1QL
	SQL/SPARQL	Relation, RDF	Non-standard extensions of SQL; Examples: Oracle's SQL/SPARQL, Virtuoso's SPASQL
	SQL/PGQ	Relation, property graph	A planned SQL extension for property graph; not implemented yet
	GQL	Relation, property graph	The latest planned standard for property graph; not implemented yet
XPath/XQuery Extensions	Core-XPath	XML, JSON	Navigational capability of XPath; not implemented yet
	GXPath$_{reg}$	XML, JSON, graph	XPath extension towards document and graph data; not implemented yet
	JSONPath	XML, JSON	A mimic of XPath for both XML and JSON; inspired the SQL/XML standard
	JSONiq	XML, JSON	A mimic of XQuery for both XML and JSON; Example: the Zorba XQuery processor
Graph Extensions	XSPARQL	RDF, XML	A combination of SPARQL and XQuery; Example: Marklogic's XSPARQL
	AgensQL	Graph, relation	A combination of Cypher and SQL; Example: AgensGraph's query language
	AgeQL	Graph, relation	A combination of SQL and openCypher; Example: PostgreSQL AGE's query language
Native	AQL	KV, JSON, property graph, GeoSpatial	A native multi-model query language; ArangoDB's query language
	OrientQL	Graph, JSON, KV	A native multi-model query language; OrientDB's query language
	KQL	Relation, Time series, GeoSpatial	A combination of SQL and Gremlin; Example: OrientDB's query language

more capable data models, including the nested relational model, the object model, and the object-relation model. Schweikardt et al. [6] give an in-depth investigation of the theories that underpin the relational query languages (relational algebra, relational calculus, and SQL), including the expressive power, query and data complexities, and manners to extend a query language with more expressiveness. Atzeni et al. [7] presents a comprehensive survey on NoSQL modeling, such as Key-value data, document data, and NoAM (NoSQL Abstract Model). Prior to Atzeni et al.'s work, Angles and Gutiérrez described the issues in graph data modeling such as graph schema, graph manipulation, and the integrity constraints enforcing consistency of graph data [8]. Wood [9] and Barceló [10] study several graph query languages from a theoretical point of view, focusing on their expressive power and the computational complexity of associated problems. Angles [11] makes a comprehensive survey of the fundamental concepts underpinning modern graph query languages, such as navigational queries, regular path queries, and graph pattern matching. Bondiombouy and Valduriez [12] analyzed a bunch of representative polystore systems on their architecture, data model, query languages, and query processing techniques. MMDBs can be roughly viewed as tightly-coupled polystores but has a significant difference within query language and query processing. Another related work is published by Lu and Holubová [3], they summarize a variety of data models widely adopted by database systems and discuss the general issues and challenges in multi-model data management. Compared to that work, this article has a different focus on multi-model query languages which are not well investigated in previous work.

Outline The survey is structured as follows: We first discuss the general concepts related to multi-model databases in Sect. 2. In the same section, we also summarize the essential queries demanded by each type of data model, including conjunctive queries for SQL, navigational queries for document and graph data, and graph pattern matching for graph data. We then step into the details of multi-model query languages. In Sect. 3, we present the SQL-based extensions toward multi-model data, including the standard SQL extensions such as SQL/XML, SQL/JSON, and GQL, and the non-standard SQL extensions such as SQL++ and SPASQL. In Sect. 4, we study the manners in which a document query language can be extended to support multi-model data. In Sect. 5, we give the graph-based extensions toward multi-model query. In Sect. 6, we describe the recent query languages that natively support multi-model data. Finally, we conclude with a brief summarization of the challenges related to designing a multi-model query language or extending from the existing base languages.

2 Preliminaries

In this section we briefly review the related concepts in multi-model data management.

2.1 Multi-model data

We concentrate on the relational model and 5 widespread NoSQL models in this article.

Relational data The relational model become the dominant data model since its invention by Codd in 1970 [4, 13]. It provides a tabular view in which data are represented as relations with columns and rows. The model has an elegant math foundation, but it is too strict to represent more complex data [14, 15]. For example, relations following the model must be flat (i.e., attribute values to be atomic) due to the First Normal Form (1NF). In recent years, various data models have been proposed to extend the relational model.

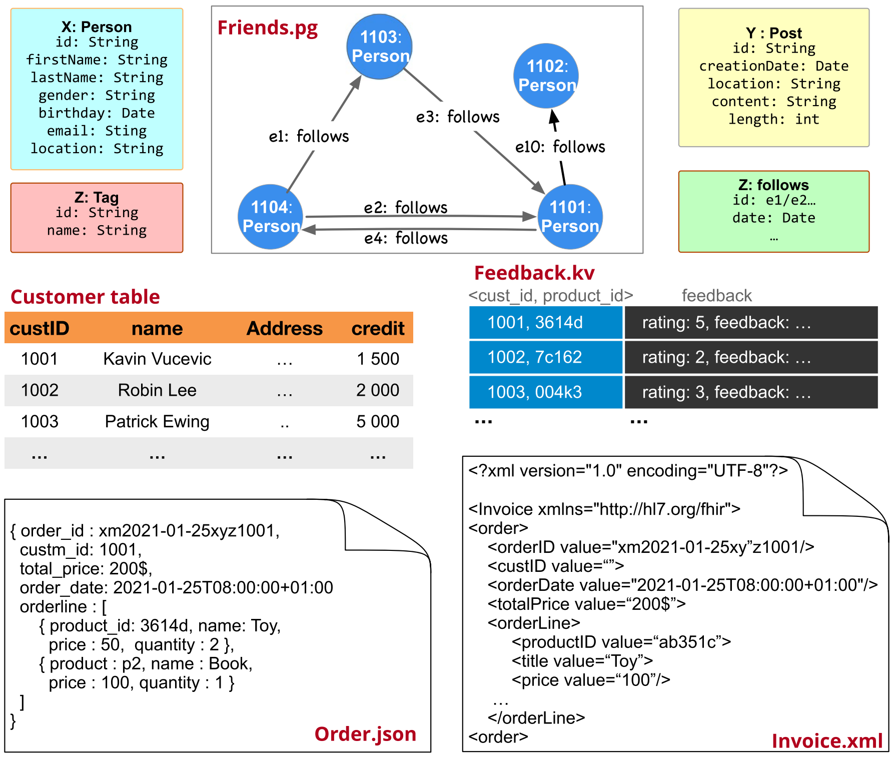
Semi-structured data Semi-structured models (or *documents*) are widely used to represent the web data and exchange over the Internet [16, 17]. It is a self-describing structure that associates semantic tags or markers and enforces hierarchies of records and fields by nesting elements within it. XML (eXtensible Markup Language) [18, 19] and JSON (JavaScript Object Notation) [20, 21] are the most widespread semi-structured data. XML is a typeless markup language that represents data using nested *elements* delimited by *tags*, which can contain *plain text*, *nested subelements*, or their *combination*. JSON [22, 23] is a human-readable open-standard format that is based on the idea of an arbitrary combination of three basic data types used in most programming languages—*key/value pairs*, *arrays*, and *objects*.

Graph data The graph model represents data as a graph $\langle V, E\rangle$ of vertices (or objects) V and edges E connecting the vertices in V. There are many variants of the graph model [8, 24, 25], including the *hypergraph model* [26], the *temporal graph* [27], the *edge-labeled graph* (ELG) [28], and the *property graph model* (PGM) [29]. The most popular representatives are the edge-labeled graph and property graph. A well-known example of the edge-labeled graph is the Resource Description Framework (RDF) [28, 30]. It represents named properties and their values as a collection of triples, $\{\ldots, \langle s, p, o\rangle, \ldots\}$. Each triple represents a relationship (labeled with a predicate p) between two nodes in the Semantic Web, where the subject s is a resource or an entity, and the object o is another node or a literal value. The *property graph* [31, 32] represents data as a directed, attributed multi-graph. Vertices and edges are objects with a set of labels and a set of *key-value* pairs, so-called properties.

Key-value data Key-Value (KV) data is the simplest data model that has been widely adopted by NoSQL databases such as BerkeleyDB and Redis.

Key-Value data consists of a collection of key-value pairs $\langle k, v\rangle$ that are viewed as individual records and each value is designated a unique key, with which we can quickly store, retrieve, or modify the value. The values in key-value pairs have no type and it is up to the application to determine what type of data is being used, i.e., it could be an integer, string, JSON, XML file, or binary data like images.

Figure 1 displays an example of multi-model data, including a relational table (*customers*), two semi-structured documents (*orders.json* and *invoices.xml*), a social network (*property graph* G), and a collection of key-value pairs (*feedback.kv*). The example is generated with UniBench [33–35]—a benchmark for testing multi-model databases and was developed by the UDBMS group from the University of Helsinki.



Customer table

custID	name	Address	credit
1001	Kavin Vucevic	…	1 500
1002	Robin Lee	…	2 000
1003	Patrick Ewing	..	5 000
…	…	…	…

Feedback.kv

<cust_id, product_id>	feedback
1001, 3614d	rating: 5, feedback: …
1002, 7c162	rating: 2, feedback: …
1003, 004k3	rating: 3, feedback: …
…	…

```
{ order_id : xm2021-01-25xyz1001,
  custm_id: 1001,
  total_price: 200$,
  order_date: 2021-01-25T08:00:00+01:00
  orderline : [
     { product_id: 3614d, name: Toy,
       price : 50,  quantity : 2 },
     { product : p2, name : Book,
       price : 100, quantity : 1 }
  ]
}
```

```
<?xml version="1.0" encoding="UTF-8"?>

<Invoice xmlns="http://hl7.org/fhir">
<order>
   <orderID value="xm2021-01-25xy"z1001/>
   <custID value="">
   <orderDate value="2021-01-25T08:00:00+01:00"/>
   <totalPrice value="200$">
   <orderLine>
       <productID value="ab351c">
       <title value="Toy">
       <price value="100"/>
    …
   </orderLine>
<order>
```

Fig. 1 An example multi-model dataset generated with UniBench [34]

UniBench simulates the application scenario within a commercial social network, in which a user can follow anyone else be interested in, leave comments on the posts from others he has already followed, navigate the marketplace for products posted by sellers and place orders on the site, and give feedback on the purchased product. In the example data, the social network is represented as a property graph G, where each node (edge) has a label describing its type (e.g., `:Person`, `:Post`, `:Tag`, and `:follows`) and associates with a set of key/value pairs as its properties. Nodes of other types, i.e., `:Post`, `:Tag`, are not presented in the graph. The JSON document records the order information, while invoices are kept in the XML document so that they can be presented to the customers in various styles. Customer feedback on the products is kept in key/value pairs (with a pair of keys, i.e., *custID* and *productID*). Finally, we use a relational table to record the information of customers.

2.2 Multi-model queries

The main purpose of managing massive multi-model data is to be able to query it. UniBench [34] also defines a collection of the workload of multi-model queries. Taking the query Q5 of UniBench as an example:

Q5: *Given a customer and a product category, list out all persons who are the customer's friends within 3-hop friendships in Follows.pg graph, besides, they have bought products in the given category. Finally, return the feedback with a 5-star rating review of those bought products.*

The query accesses data from heterogeneous sources, i.e., *Follows.pg*, *Orders.json*, and *Feedback.kv* listed in Fig. 1. As shown in Query 1, we can write Q5 in ArangoDB's query language (AQL) [36]. The query involves a variable-length path query on the property graph, an embedded array operation on the JSON document, and a composited key lookup on key-value pairs. Thereafter, we use two equi-joins (i.e., a graph-JSON join and a JSON-KV join) to assemble the partial records from the property graph, JSON document, and key-value pairs.

```
LET personCollection = (
 FOR person IN 1..1 OUTBOUND @id Friends
  FOR order IN Orders
   FILTER order.cust_id == person._key AND @brand IN order.orderline[*].brand
   RETURN distinct(person)
)
FOR person IN personCollection
 FOR post IN OUTBOUND person._id PersonHasPost
  FOR tag IN OUTBOUND post PostHasTag
   RETURN {person:person, tags:tag};
```

Query 1 UniBench Q5 in ArangoDB's AQL

Query 1 is a *multi-model query* (MMQ), which is a mixture of relational queries, document queries, graph queries, and key/value fetches. By document queries, we can navigate through the root to any particular node in a document. By graph query, we mean queries that involve the particular connectivity features coming from the edges, e.g., shortest path, graph traversal, and pattern matching. The query operations in key-value stores rely on low-level key lookups. Ideally, a multi-model query language is capable of expressing an arbitrary mixture of queries on all data models we discussed in the previous section. Nevertheless, we concentrate on three types of fundamental queries.

(1) Relational queries The essence of relational queries is *conjunctive queries* (CQs), which use a restricted form of the first-order logic expressions with only conjunction operator and existential quantifier. In particular, CQs are equivalent to the `SELECT-FROM-WHERE` queries in SQL in which the `WHERE` clause uses exclusively conjunctions of atomic equality conditions [6]. However, we cannot express the transitive closure and aggregate with CQ (likewise with the relational algebra and calculus) [37]. Normally, one can extend the expressive power of CQ in several ways: (1) adding more operations such as union and negation; (2) allowing recursion, which is essential for extending CQ to graph data; (3) counting, which is essential for aggregate queries; (4) introducing more powerful grammars (e.g., regular expression or context-free grammar) for formulating complex query expressions.

(2) Navigational queries Navigational queries (NQs) form the foundation for many semi-structured and graph query languages such as Lorel [38], OQL [39], XPath [40], XQuery [41], XSLT [42], and nSPARQL [43]. Most formalisms for NQs are based on the notion of *path expression* [44–47], which specify the way to navigate the underlying data [16, 39, 46, 48, 49]. Essentially, there are two types of NQs: *simple path query* (SPQ) and *regular path query* (RPQ) [9, 16, 50–54]. An SPQ is a complete sequence $\pi = l_1. \cdots .l_n$ of edges, where $l_1, \ldots, l_n$ are labels of edges between objects. It can be extended by introducing wildcards (?, *, %, or #) and more expressive grammar such as context-free grammar [55]. Formally, RPQs are expressions of the form (x, L, y), where L is a regular language over an alphabet Σ of edge labels. It generalizes SPQs to support transitive closure and can be further extended with more complex patterns, backward navigation, relations over paths, and mixing labels and data in nodes [45, 55], such as the *conjunctive regular path queries* (CRPQs) [56], *two-way regular path queries* (**2CRPQs**) [56]. The *property path queries* extend RPQs with a mild form of negation [57] and forms the conceptual core of the SPARQL 1.1 standard [58, 59].

(3) Graph pattern matching *Graph pattern matching* (GPM) is one of the foundations that underpin the graph query languages such as W3C's SPQAQL [59–61], Cypher [29, 62], TinkerPop's Gremlin [63, 64], Oracle's PGQL [65, 66], LDBC's G-CORE [67], and TigerGraph's GSQL [68, 69]. A *graph pattern* $P = (V_P, E_P, L_P)$ is a directed graph that specifies the structural and semantic requirements that matched subgraphs in the data graph G must satisfy. Tree pattern is a special form of graph pattern that has been extensively studied in the context of XML database [70, 71]. The task of GPM is to find the set M of subgraphs from data graph G that match a pattern graph P. The precise definition of a match varies among graph query languages but is generally based on the following semantics: (1) *subgraph isomorphism* (i.e., structural matching) or near isomorphism between P and $m \in M$ [72], and (2) *equality* or *similarity* between the types and attribute values of the vertices and edges in P and those in $m \in M$.

2.3 Cross-model query processing

So far we have stated examples of multi-model data and queries. We proceed to present *multi-model query languages* (MMQLs) that allow users to express MMQs in a declarative way. Ideally, an MMQL is capable of expressing any MMQs using an arbitrary mixture of relational, navigational, and graph queries that span over a collection of multi-model data. The problem of accessing multi-model data sources [73, 74], i.e., managed by various heterogeneous DBMSs such as relational, XML, or graph DBMSs, has been extensively studied in the context of multidatabase systems [75, Chapter 7] (also known as federated database systems) and data integration [76]. In general, an MMQ q is a mapping that spans a collection of multi-model data $D = \{d_1, \ldots, d_k\}$ and maps it to the query result $q(D)$. The evaluation of q is a challenging task and we say it as *cross-model query processing* [2, 12, 77] when an MMQ spans multiple data models. Roughly, we can identify two feasible approaches for cross-model query processing: (1) the *mediator-wrapper* fashion in Polystore systems, and (2) a *holistic evaluation* in MMDB systems.

Polystores The basic idea of Polystore (or Multistore) systems (e.g., Polybase[78] and BigDAWG [79, 80]) is to provide integrated access to a set of heterogeneous data stores (SQL or NoSQL). We can divide polystore systems into three categories, i.e., loosely-coupled, tightly-coupled, and hybrid, based on the level of coupling with the underlying data stores. Query processing in polystores is complex and commonly based on the **mediator-wrapper** architecture [12, 81], where the **mediator** provides a *global view* of multi-model data to users and each wrapper supports the functionality of translating subqueries to the particular store, and reformatting answers (partial results) appropriate to the mediator. The mediator has a catalog of data stores, and each wrapper has a local catalog of its data store. With the global schema, one can express MMQs over a polystore as if it is a single database. However, query evaluation in polystores involves massive data exchange, which consists of defining a global schema over the underlying multi-model data and mappings between the global schema and the local schemas.

Multi-model databases Polystores may result in disparate data silos and multiple interfaces that require expensive integration workflows. This poses a great challenge to query processing and encourages the development of Multi-Model DataBases (MMDBs) [2, 34]. A plethora of MMDBs emerged over the past years, such as Oracle [82, 83], PostgreSQL [84], MongoDB [85], ArangoDB [86], and OrientDB [87]. These MMDBs are developed in two ways: (1) native MMDBs built from scratch such as ArangoDB and OrientDB; (2) extensions of the existing relational, document, and graph DBMSs. For more details about MMDBs, we refer the readers to Lu and Holubová's survey paper [3]. Table 2 summarizes the representatives for each type of MMDBs. An MMDB system can be viewed as a variant of the tightly integrated polystores, in which multi-model data is accommodated in a set of native storage engines, a global execution engine is on the top of the local stores, and MMQs are expressed in a unified query language. There is a global execution engine that runs on top of local storage engines and the executor directly accesses local storage engines without any intervention of wrappers. Consequently, MMQs can be evaluated in a holistic manner instead of being divided into subqueries.

Cross-model query processing via schema mapping Cross-model query processing in both polystores and MMDBs may incur expensive data exchange [77, 88]. Figure 2 gives a visual representation of the MMQ Query 1, in which data fragments from the property graph, JSON document, and key-value pairs should be assembled into an intact query result with a graph-JSON join and a JSON-KV join. The two joins can be done through two data exchange schemes, i.e., a graph-to-JSON mapping and a KV-to-JSON mapping.

The main task in data exchange is to translate data between the source schema and the target schema under a set of source-to-target constraints known as schema mappings [89]. Query processing under schema mappings [90] has also been investigated extensively in the context of multidatabase systems [75, Chapter 7] and data integration [91]. It typically relies on the two cases where each target atom is mapped to a query over the source (called GAV, global-as-view), and where each source atom is mapped to a query over the target (called LAV, local-as-view) [76]. The general case, called GLAV, in which queries over the source are mapped to queries over the target, has attracted a lot of attention recently, especially for data exchange. The GAV and LAV are widely implemented in polystores. For more details about query processing via schema mapping, we refer the readers to Bondiombouy et al.' survey paper [12].

Table 2 Multi-model databases

Category	DBMSs	Query languages	Supported models	Description
Relation-based Extensions	AsterixDB	SQL++	Relation, JSON	Extend SQL with path expressions
	Oracle	PL/SQL	Relation, JSON, XML, RDF	Extended from SQL
	MySQL	Oracle MySQL	Relation, KV	Extended from SQL
	PostgreSQL	PL/pgSQL	Relation, JSON, XML, KV	Extended from SQL, Turing-complete
	SQL Server	T-SQL	Relation, JSON, XML	Extend SQL to support JSON and XML, Turing-complete
	IBM Db2	SQL	Relation, XML	Extend SQL to support trees, not Turing-complete
Document-based Extensions	MarkLogic	XQuery/XPath	XML, JSON, RDF, Relation	Extended from XQuery, Turing-complete
	Zorba	XQuery/JSONiq	XML, JSON	Extended from XQuery
Graph-based Extension	OrientDB	OrientQL	Graph, JSON, KV	Extended from graph model, not Turing-complete
	AgensGraph	AgensQL	Graph, Relation	Cypher on SQL or SQL on Cypher, not Turing-complete
	Aphache AGE	openCypher+SQL	Graph, Relation	a graph extension of PostgreSQL to use SQL along with openCypher
Native	ArangoDB	AQL	JSON, graph, KV	SQL-like query language
	OrientDB	OrientQL	Graph, JSON, KV	SQL-like
	Kusto	KQL	Relation, time series, GeoSpatial	SQL-like

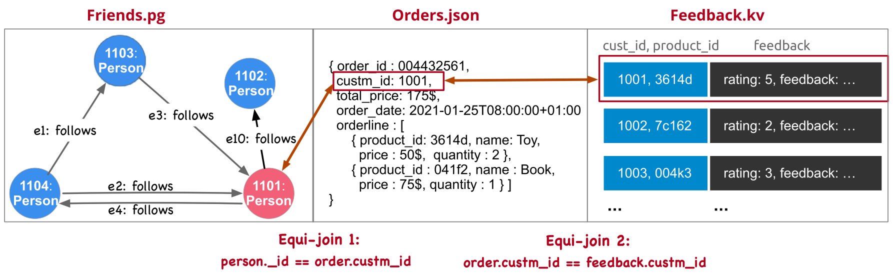


Fig. 2 A visual representation of the query Q5 in UniBench

3 Relational extensions

In this section, we present the relational extensions toward multi-model data.

3.1 Relational query languages and the SQL standards

In the early 1970s, Codd proposed two mathematical query languages, called relational algebra (RA) and relational calculus (RC, also known as the Data Sublanguage ALPHA) [92, 93], which form the basis for real languages like SQL. ALPHA inspired the design of subsequent query languages including SEQUEL [94, 95] and QUEL [96, 97]. The structured query language SEQUEL finally evolved to SQL. The core query syntax of SQL is the `SELECT-FROM-WHERE` (SFW) clauses, and we say a query language is SQL-like if it has similar clauses as well. By 1986, SQL had been formally adopted by the ANSI and ISO as a standard database query language, namely SQL-86 (also known as SQL1). The earliest versions of SQL lacked support for some aspects of the relational model, including subqueries, primary keys and referential integrity. All these problems were solved by the SQL:1992 standard (or SQL2) [98]. Procedural extensions SQL/PSM [99] formally became a part of the SQL:1996. The common table expressions (CTEs), `WITH [RECURSIVE ]` construct, was introduced into SQL:1999 (also called SQL3) to allow recursive queries [100, 101].

Recently, many traditional RDBMSs have been evolving to their multi-model versions, such as Oracle, MySQL, MS SQL Server, PostgreSQL, IBM Db2, and MariaDB. The query languages of these systems are naturally extended from the SQL standard and thus have certain compliance with each other. Table 2 lists several such extensions. Consequently, a number of extensions were proposed to handle multi-model data, and some of them have been accepted by the SQL standard, such as SQL/XML [102], SQL/JSON [103], SQL/PGQ [104], and GQL [105–107]. In this section, we present three standard extensions such as SQL/XML, SQL/JSON, and GQL, and two non-standard extensions such as SQL++ and SPQRSQL.

3.2 SQL extensions toward semistructured data

We mainly focus on the SQL extensions toward XML and JSON. In addition to the predefined data types (e.g., NUMERIC and CHAR), SQL/XML and SQL/JSON introduce the new data type `XML` and `JSON` into the SQL standard, where each data type associates with a set of constructors, functions, and rules for XML(JSON)-to-SQL mapping. Even though XML and JSON are somewhat similar—documents with nested structures—their integration into SQL is quite different. The most striking difference is that the standard does not define a native JSON type like it does for XML. Instead, the standard uses strings to store JSON data. Other than creating a new data type, SQL++ builds a unified data model to capture and manipulate both relational data and JSON documents.

3.2.1 SQL/XML and SQL/JSON

SQL/XML SQL/XML [102] is a collection of XML-related specifications and became part 14 of the ANSI/ISO SQL standard in 2003. It is SQL-centric and lets SQL queries create XML structures with a few powerful XML publishing functions. It introduces the predefined data type XML together with constructors, functions, and XML-to-SQL data type mappings to support the manipulation and storage of XML in a relational database. SQL/XML has substantially similar functionality to XQuery. The specification defines the data type XML, and functions working on XML, including element construction, mapping data from relational tables, combining XML fragments and embedding XQuery expressions in SQL statements. Functions that can be embedded include XMLQUERY (which extracts XML or values from an XML field) and XMLEXISTS (which predicates whether an XQuery expression is matched). The existing RDBMSs have their own implementations of SQL/XML. As shown in Table 3, they have different compliance with the SQL/XML standard.

In a nutshell, the SQL/XML extension consists of the following 4 parts: **(1) The XML data type and related constructors**. The SQL/XML standard defines a single data type named XML. Its possible values are either null or a document conforms to the definition in W3C's XML recommendation [19]. DB2, SQL Server, and PostgreSQL provide native support for the XML data type. MySQL doesn't support `XML` data types like SQL Server or PostgreSQL but just takes XML as a CLOB (Character Large Object) data type. Oracle entitles the `XMLType`' instead of '`XML`'. **(2) Support for XPath/XQuery.** In order to retrieve XML data within a SQL query, one must incorporate the XPath expressions or XQuery's `FLWOR` clauses into the standard SQL syntax. Excepting MySQL supports only XPath, other RDBMSs provide full supports for XPath and XQuery. **(3) Special functions for handling XML data.** The standard defines 9 functions that allow application to retrieve XML directly within SQL queries, including `XMLAGG`, `XMLCONCAT`, `XMLCOMMENT`, `XMLELEMENT`, `XMLFOREST`, `XMLPARSE`, `XMLPI`, `XMLROOT`, and `XMLSERIALIZE`. Oracle, DB2, and PostgreSQL provide full support for all the XML functions. MySQL provide two functions, `ExtractValue()` and `UpdateXML()`, to work

Table 3 Compliance to the SQL/XML standard

	Oracle 11g	IBM DB2 9.7	MS SQL Server 2008	MySQL 5.1.30	PostgreSQL 9.1
XML Datatype	XMLType	XML	XML	CLOB	XML
XPath/XQuery Syntax	Full	Full	Full	XPath	Full
Predicates	Full	Full	Partial	No	Full
Functions	Full	Full	Partial	Partial	Ful

with XPath. **(4) Rules for mapping XML to SQL tables.** The standard defines several rules for mapping SQL types to XML or in a reversed mapping. For example, mapping an SQL table to XML and an XML Schema document.

SQL/JSON SQL/JSON [103] was introduced into the standard (ISO/IEC 9075:2016) in 2016. The specification includes a SQL/JSON data model, a set of SQL/JSON functions working on the data model, and a SQL/JSON path language for navigating the nested data structure. The data model is used by the SQL/JSON functions and the path language. There are five standard SQL/JSON functions, such as JSON_CONCAT, JSON_KEYCOUNT, JSON_KEYS, JSON_CONTAINS, and JSON_EXISTENCE. The SQL/JSON path language use the dot notation and it is a query language used by the SQL/JSON operators (JSON_VALUE, JSON_QUERY, JSON_TABLE, and JSON_EXISTS) to query JSON text. Unfortunately, the current SQL/JSON standard does not define a native data type for JSON, but represents it as a VARCHAR or binary formats such as AVRO or BSON.

3.2.2 SQL++

SQL++ [108, 109] applies to both relational and JSON data. It is backward-compatible with the SQL standard and is extended with a small number of query properties for JSON. Differing from the previous extensions, SQL++ creates a unified data model for relational tables and JSON.

(1) SQL++ Data Model The SQL++ data model is a superset of both relational tables and JSON, based on three observations: (a) A SQL tuple corresponds to a JSON object literal; (b) a SQL string, integer, or boolean map to the respective JSON scalar; and (c) a JSON array is similar to a SQL table (bag) with the order. The model expands JSON with bags (as opposed to having JSON arrays only) and enriched values, i.e., atomic values that are not only numbers and strings. Vice versa, we can also think of SQL++ as expanding SQL with JSON features: arrays, heterogeneity, and the possibility that any value may be an arbitrary composition of the array, bag and tuple constructors, hence enabling arbitrary nested structures, such as arrays of arrays.

The data operated on and query results of SQL++ are JSON values, which can be either *primitive values* or *structured values*, as shown in Fig. 3. A primitive value is any of the following: (1) a string like "Hello", (2) a number like 42 or −3.14159, (3) a boolean value true or false, or (4) null. A structured value could be an *object* or *array*, where an object is a list of *name-value pairs*

Fig. 3 A JSON value and an array

```
-- a JSON value --
customers{
  "custid": "C13",
  "name": "T. Cruise",
  "address":
  { "street": "201 Main St.",
    "city": "St. Louis, MO",
    "zipcode": "63101"
  },
  "rating": 750
}
-- an array --
[1, 2.5, "Hello", true, null]
```

separated by a comma and enclosed in curly braces. The name-value pairs are also called *fields*. Each name is a string and each value can be any *JSON value*. An array is an ordered list of items separated by commas and enclosed in square brackets. Each item can be any JSON value.

(2) Query structure and path expressions SQL++ has been adopted by many database systems, such as Couchbase's N1QL-SQL++ alignment [110] and the SQL++ interface of AsterixDB [111, 112]. An SQL++ query is either an SFW query or an expression query. Unlike SQL expressions, which are restricted to outputting scalar and null values, SQL++ expression queries output arbitrary values, and are fully composable with SFW queries.

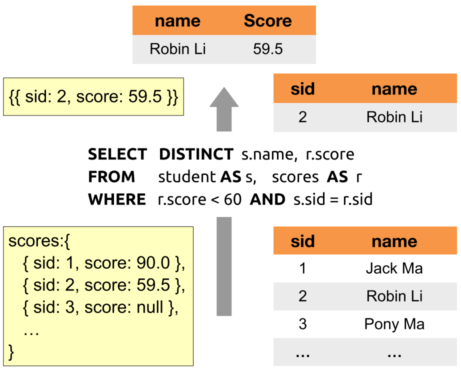
SQL++ supports simple path queries, which consist simply of an expression that evaluates to an object, followed by a dot and an identifier. For example, in the expression "`person.name`", `person` is a variable that evaluates to an object, and the identifier `name` is used to find a field in the object whose name matches that identifier. Since the result of a path expression is an object, it can serve as one *step* in a longer path expression such as "`person.name.first`". Since SQL++ supports boolean operators such as `NOT`, `AND`, and `OR`, we can issue more complex query patterns by binding several path expressions with such operators. Unfortunately, SQL++ doesn't support complex queries like RPQs, CRPQs, and 2CRPQs. Figure 4 shows a cross-model query of SQL++ accessing both the relational table and JSON document. The query is a composite of the relational query and path query. For more examples of SQL++, please refer to the technical papers [108, 109] and D. Chamberlin's tutorial [109].

3.3 SQL extensions toward graph data

3.3.1 SQL extension for RDF

The emergence of the Semantic Web is enabling new approaches to federated data queries. RDF promotes universally grounded identifiers for data, allowing the SPARQL query language for RDF to perform joins across different data sources. There is no standard SQL extension towards SPARQL. W3C defines a set of rules for mapping between relational data and RDF [113] and the RDBMS vendors can

Fig. 4 A SQL++ query



develop their own SQL/SPARQL extensions following these rules. We present two extensions, i.e., Oracle's SQL/SPARQL and OpenLink Virtuoso's SPASQL [114].

(1) Oracle's SQL/SPARQL Oracle extends SQL with full SPARQL 1.1 query constructs [115]. A SPARQL subquery is embedded within the SQL query following the SEM_MATCH statement. As shown in the following query, by integrating RDF with existing enterprise data, one can use Oracle's SQL/SPARQL to JOIN with relational data, to create tables/views for RDF data, and to allow SQL constructs/functions working on RDF data.

```
SELECT n1, n2
FROM TABLE(
  SEM_MATCH(
    'PREFIX foaf: <http://...>
    SELECT ?n1 ?n2
    FROM graph1
    WHERE {?p foaf:name ?n1
      OPTIONAL {?p foaf:knows ?f .
                ?f foaf:name  ?n2 }
      FILTER (REGEX(?n1, "^A")) }
    ORDER BY ?n1 ?n2',
SEM_MODELS('M1'), ...));
```

Query 2 A Graph Query in Oracle's SQL/SPARQL

(2) Virtuoso's SPASQL As a multi-model RDBMS, Virtuoso supports native stores for a bunch of data models such as relational tables, RDF, and XML. It also provides multiple ways to declaratively manipulate data, one of them is called SPASQL (SPARQL-in-SQL) [116], which delivers a built-in blend of SQL functionality with SPARQL functionality. SPASQL is a simple extension of the SQL standard, allowing execution of SPARQL queries within SQL statements, typically by treating them as a subquery or function clauses. The syntax of SPASQL is as follows:

```
SELECT {select-list}
FROM (SPARQL {sparql-query}) AS {sql-alias}
WHERE {sql-query-join-conditions}
```

In a SPASQL query, a SPARQL subquery is embedded within a SQL query following the keyword "`SPARQL`". The subquery draws only from relations represented as RDF statements and returns data in tabular form to an SQL interface. Virtuoso has a universal server, which is a middleware that combines the functionalities of a traditional RDBMS, an RDF engine, an XML engine, a web server, and even a file server in a single system. While processing a SPASQL query, the universal server will incur an RDF function call to the RDF engine for handling the SPARQL subquery and the result is then returned to the host SQL engine for further processing. In this way, one can retrieve both relational and RDF data with a SPASQL query like Query 3.

```
SELECT  X.customer AS "CUSTOMER", X.interest AS "INTEREST"
FROM (
  SPARQL
  SELECT ?customer ?interest
  FROM <http://www.example.com/customers>
  WHERE { ?customer foaf:interest ?interest }
) AS X;
```

Query 3 A SPASQL (SPARQL-in-SQL) Query

3.3.2 SQL extension for property graph

In recent years, a wealth of property graph query languages have been developed by database vendors, including Neo4j's Cypher/openCypher [117, 118], Oracle's PGQL [65, 66], LDBC's G-CORE [67], and TigerGraph's GSQL [68]. In 2019, the ISO SQL Committee initiated a new project GQL (Graph Query Language) [106] as the standard property query language to unify the previous languages. Prior to GQL, WG3, and SC32 have helped to define SQL/PGQ (SQL/Property Graph Queries) as a new planned Part 16 of the SQL standard [104], which allows a read-only graph query to be embedded inside a SQL SELECT statement and returning a table of data values as the result. SQL/PGQ is mainly used for matching a graph pattern using a syntax that is very close to Cypher, PGQL, and G-CORE. The GQL project coordinates closely with the SQL/PGQ and its project proposal [105] shows that GQL will in general be a superset of SQL/PGQ.

GQL is intended to be useable both as a declarative property graph query language as well as in conjunction with SQL [119, 120]. The existing property graph query languages and the new GQL standard can fit together with the existing SQL standard. One key goal is composability—the ability to return the result of a graph query as a graph that can be referenced in other graph queries. Query syntax follows the syntactic tradition of SQL, i.e., using the `SELECT-FROM-WHERE` clauses, and a query is represented as a tree of clauses, each introduced by a distinct keyword and optionally further qualified by sub-clauses, expressions, and patterns. According to the working draft [105] of the GQL specification in 2019, GQL reuses basic literal and expression syntax from SQL but is amended with the following features.

(1) New data types GQL provides new data types for handling graph elements. Graph types describe the structure of a graph in terms of nodes, edges, and their labels and properties. A graph type consists of the following definitions: (i) An (abstract) content type definition that defines a combination of required labels, required properties, and the required data types of those properties. (ii) A node type definition that defines a combination of required labels, required properties, and the required data types of those properties that a node shall have. Node types may be defined as an extension of other node types or content types. (iii) An edge type definition that defines a combination of required labels, required properties, and the required data types of those properties that an edge and its endpoints shall have. Edge types may be defined as an extension of other edge types or content types. (iv) Graph element constraint definitions such as key constraint definitions, where a graph element is either a node or an edge.

(2) Query syntax, structure, and clauses GQL borrows the syntax from Cypher and provides an SQL-like query structure over table or graph data that produces a tabular query result set. Specifically, GQL uses the `SELECT-(MATCH)-FROM-WHERE-(RETURN)` clauses in which the `MATCH` and `RETURN` clauses are optional when querying relational tables. Therefore, GQL can provide full compliance with the SQL standard. Given a pattern `MATCH (p:Person)-[:lives_in]->(c:City)`, the query result returned by the `SELECT` clause is represented in a tabular fashion. We have to use the optional `RETURN` statement if we need to construct a graph as output. The syntax of GQL's `RETURN` clause is given in Query 4. GQL follows the definition in Cypher and also provides multiple semantics for graph pattern matching [121, 122], i.e., isomorphism and homomorphism. In addition, GQL borrows the syntax of regular path expressions from PGQL [66], i.e., "`(a:Person)-/:friend_of*/->(a:Person)`".

```
<return statement> ::=
  [ <statement mode> ] RETURN {
        <asterisk>
        | [ <set quantifier> ] <return item list> <projection clauses>
}

<return item list> ::=
  <return item> [ { <comma> <return item> } ... ]

<return item> ::=
  <value expression> [ AS <identifier> ]
...
```

Query 4 Syntax of GQL's RETURN Clause

4 Document extensions

Many semi-structured data models have been proposed in practice, including the ODMG Data Model (ODM) [39, 123], the Object Exchange Model (OEM) [124], XML [19], JSON [21], and Google's Protocol Buffers (ProtoBuf) [125]. The former 3 models have standard query languages—OQL [126] for ODM data, Lorel for OEM data [38, 48], and XPath/XQuery for XML data [40, 127] respectively. Unfortunately, we do not have standard query languages for JSON and ProtoBuf. ProtoBuf is mainly used for serializing structured data and Google only provides low-level API for programming languages like Go and Java. In this section, we proceed to discuss document query languages and their extensions toward multi-model data.

4.1 XPath/XQuery extensions

Since the late-1990s, a bunch of XML query languages has been proposed, such as XPath/XQuery [40, 41], Xcerpt [128], XDuce [129], CDuce [130], and XTreeQuery [131], among which XPath/XQuery have already become the standards advocated by W3C. In this section, we examine the expressiveness of XPath/XQuery on path queries and graph pattern matching. In addition, we describe the extensions of XPath/XQuery toward multi-model data.

4.1.1 Core XPath/XQuery

(1) XPath XPath is a path language designed to both navigate nodes and query data from an XML document. It plays a prominent role as an XML navigational language and becomes a core component of a number of more expressive languages with variables such as XQuery. Essentially, an XPath query is a path expression that consists of a sequence of location steps, where each step, `/axis::node_label-[predicates]`, has three components: (i) an *axis*, (ii) a *node test*, and (iii) zero or more *predicate*. XPath supports a number of axes: *ancestor*, *ancestor-or-self*, *attribute*, *child*, *descendant*, *descendant-or-self*, *following*, *following-sibling*, *parent*, *preceding*, *preceding-sibling*, and *self*. All nodes in an XML document conform to a document order in which the data is represented as a hierarchy tree and a path query is evaluated in the order with respect to a context node. The axes can be divided into a forward axis and a reverse axis according to the direction from the context node. For example, `child` or `descendant` specifies the two directions for XML navigation respectively. A node test within an XPath query is to retrieve nodes and the predicates after the node test is to filter a sequence of values. Apart from using the name of a node or a wildcard (to select unknown nodes), we can also use other node tests such as `node()` and `text()`.

(2) XQuery XQuery shares the same data model with XPath and supports the same functions and operators. It is a superset of XPath and takes advantage of both XPath expressions and SQL-like FLWOR syntax [40, 132]. It extends XPath in many ways, in which the most important ones are the capability to specify output query nodes and an introduction of sequences in values. In addition to the path expressions, XQuery provides CRUD syntax like SQL. There are several versions of XQuery in use. XQuery 1.0 [41] became a W3C Recommendation in 2007, XQuery 3.0 [127] in 2014 and the revised version XQuery 3.1 became a W3C Recommendation in 2017. The overall design of XQuery is based on a language proposal called Quilt [133], which was significantly influenced by OQL [39, 123] and SQL [134], and by previous XML query language proposals such as XQL [135] and Lorel [38].

4.1.2 Document navigation and the core-XPath fragment

A simple path query of XPath is a sequence of axis steps (`A/B/C`). Query 5 presents 5 simple path queries of XPath. P1 lists the descriptions of all items offered for sale by Smith. P2 has the same purpose as P1 but uses a different axis. P3 defines the path P1 as a variable `description`. P4 reuses the variable in P3 to list the status attribute of the item that is the parent of a given description. P5 is to list all the items that are the parents of a given description but excluding the `@status` elements. We can use predicates in the steps of a path query to filter a sequence of values. For example, in the step `item[seller = "Smith"]`, the predicate (`seller = "Smith"`) is used to select certain item nodes and discard others. We will refer to the items in the sequence being filtered by a predicate as candidate items. The variable `description` given in P3 is used to bound a node in the document, and traverses will start at the nodes bound to the variable.

```
P1: document(''items.xml'')//description
P2: document(''items.xml'')/child::*/child::item[child::seller = ''Smith'']
P3: $description = document(''items.xml'')//description
P4: $description/../@status
P5: $description/../*[not(self::@status)]
```

Query 5 Simple XPath Queries

Path expressions can be also written in abbreviated syntax. Within a path expression, a single dot (`.`) refers to the context node (self-axis), and two consecutive dots (..) refer to the parent of the context node. These notations are abbreviated invocations of the child (`/`), descendant (`//`), and attribute axes (`@`), respectively.

The navigational capability of the XPath standard can be captured all by a fragment called Core-XPath [49, 136, 137]. The syntax of Core-XPath is given in Table 4. It supports all XPath's axes, except for the attribute and namespace axes, and allows sequencing and taking unions of path expressions and full booleans in the filter expressions. Formally, the semantics of Core-XPath queries can be modeled by binary relations on the nodes of an XML tree. For example, the path expression `descendant::p` (abbreviated as `.//p`) denotes in any XML tree T, the set of all pairs (m, n) with n a descendant node of m that has tag name p. Thus the binary relation is equivalent to a conjunctive query: $\phi(x, y) = \text{descendant}(x, y) \wedge \text{p}(y)$.

Conversely, the conjunctive query $\phi(x, y) = \exists z_1 \dots z_n \bigwedge_{i=1}^{n} \text{descendant}\left(x, z_i\right) \wedge \text{p}_i\left(z_i\right)$ defines a binary relation that can be expressed as a union of the XPath queries, i.e., `descendant::p(1)/.../descendant::p(n)/descendant::q`, for all permutations of $1 \dots n$. Therefore, conjunctive path queries and 2-way conjunctive path queries can be naturally supported by all versions of XPath.

XPath standards do not support RPQs, but it can be fixed by adding two operators, i.e., Kleene star and path equality. The Kleene star allows us to take the reflexive transitive closure of arbitrary path expressions and path equalities allow us to specify the equality conditions. Formally, the semantics of these operators is as follows [49]:

$$R^*_{\text{PExpr}} = \text{reflexive transitive closure of } R_{\text{PExpr}}$$
$$R \approx S_{\text{NExpr}} = \left\{x \mid \exists y \cdot (x, y) \in R_{\text{PExpr}} \cap S_{\text{PExpr}}\right\}$$

Thereafter, we can use the RPQ expression, `(child::p)*/child::q`, to list all child items or descendant items named `q`, where `(path)*` denotes the reflexive

Table 4 Syntax of Core-XPath

MMDBs	Data models	Query languages
Axis	:=	self ǀ child ǀ parent ǀ right ǀ left ǀ descendant
	:=	ǀ ancestor ǀ following ǀ preceding
	:=	ǀ following sibling ǀ preceding sibling
NameTest	:=	QName ǀ *
Step	:=	Axis::NameTest
PathExpr	:=	Step
	:=	ǀ PathExpr/PathExpr
	:=	ǀ PathExpr union PathExpr
	:=	ǀ PathExpr[NodeExpr]
	:=	PathExpr
	:=	ǀ not NodeExpr
	:=	ǀ NodeExpr and NodeExpr
	:=	ǀ NodeExpr or NodeExpr

transitive closure of the binary relation denoted by `path`. RPQ can be viewed as a mix between Core XPath and regular path expressions [16]: it has the filter expressions of the former and the Kleene star of the latter. With the extended operators, one can also easily issue CRPQ and 2CRPQ queries.

XML path query and tree pattern query Essentially, there are two types of structural XML queries [138]: (1) the no branching path query, and (2) the XML *tree pattern query* (also known as *twig query*). Query 6 shows three XML queries, where Q1 is a simple path query, but Q2 and Q3 are two twig queries formulated with XPath and XQuery respectively. Using the boolean operators provided by XPath/XQuery, one can issue more complex queries by a combination of XPath expressions. XPath provides full support to the SPQs and SPQs with wildcards, but unfortunately the current version of it does support any RPQ and its extensions.

```
Q1: //a/b/c/d
Q2: //a//b[.//c]/d

Q3: for $a in //a, $d in $a//b//d
    return <o>{$d,$a/b}</o>
```

Query 6 XPath and XQuery Query Examples

An XML tree pattern query is a special case for the graph pattern query. Formally, a twig query is a pair $Q = (T, F)$, where T is a node-labeled and edge-labeled tree with a distinguished node $x \in T$ and F is a boolean combination of constraints on nodes. Node labels are variables such as \$$x$ and \$$y$. Edge labels are one of PC, AD indicating parent–child or ancestor–descendant. Node constraints are of the form \$$x.tag = TagName$ or \$$x.datarelOpval$, where \$$x.data$ denotes the data content of node \$$x$, and `relOp` is one of $=, <, >, \leq, \geq, \neq$. Twig queries can be seen as an abstraction of a core fragment of XPath and XQuery [139]. Therefore, a substantial amount of work on XML query evaluation and optimization [70, 140, 141] using tree patterns as a basis.

4.1.3 Graph pattern matching with XPath/XQuery

An XML tree can be viewed as a special directed acyclic graph (DAG), so its query languages XPath/XQuery can be also used to query graph data. Several works have been done to extend the XPath to query graph data [142–144]. Path expressions of the core graph XPath, denoted by **GXPath**$_{core}$, are stated as:

$$\alpha, \beta := \varepsilon | \_ | a | a^{-} | a^{*} | a^{-*} | [\varphi] | \alpha \cdot \beta | \alpha \cup \beta | \bar{\alpha}$$

Similary, path expressions of *regular graph XPath*, denoted by **GXPath**$_{reg}$, are given

$$\alpha, \beta := \varepsilon | \_ | a | a^{-} | [\varphi] | \alpha \cdot \beta | \alpha \cup \beta | \bar{\alpha} \mid \alpha^{*}$$

We call this fragment “Core graph XPath” since it is natural to view edge labels (and their reverse) in data graphs as the single-step axes of the usual XPath on trees. For instance, a and a- could be similar to “child” and “parent”. Thus, in our core fragment, we only allow transitive closure over navigational single-step axes, as is done in Core XPath on trees. Note that we did not explicitly define the counterpart of node label tests in **GXPath** node expressions to avoid notational clutter, but all the results remain true if we add them. According to the investigation [142–144], **GXPath** provides full support to the navigational queries and graph pattern queries.

4.1.4 Querying relational data with XPath/XQuery

When XPath is used to query relational data, relational tables are treated as though they are XML documents, and path expressions work in the same way as they do for XML. Since relational data have a flat structure, path expressions used for tables are usually simple. XQuery extends XPath with the FLWOR clauses. To extend it to relational data, three essential perspectives must be taken into account: (i) The mapping between relational tables and XML data; (ii) The syntax difference between XQuery and SQL, especially the returned clause; (iii) The correspondence between the built-in functions for XQuery and SQL. The SQL/XML standard [102] defines the rules for mapping between relations and XML data, so an XPath/XQuery-based extension can follow these rules to provide support for relational data. Many database systems implement the SQL/XML mapping rules but have different built-in XQuery functions, so their supports for relational data vary a lot. For example, the XQuery engines for MarkLogic [145] and DataDirect [146] have very different supports for relational data.

DataDirect’s XQuery DataDirect treats all data to look like XML. Notice that while the syntax between the XQuery expression and the SQL statement differs, the semantics are the same—the FOR clause has been translated as part of the SELECT FROM statement; the where clause has been translated as the predicate; and so on. Another difference that must be taken into account when using XQuery to query relational data is structure—the output of a SQL statement is a table (a flat structure), but the typical XML value is a tree. To achieve the required transformation of the result from a flat structure to a tree structure, DataDirect XQuery translates the query into two parts: an XML construction part and a SQL part. Query 7 illustrates a such query, in which the XML construction part adds XML tags to the results retrieved from the database to create the hierarchy requested in the query.

```
<root>
{
  for $book in collection(''pub.dbo.bookshop'')/bookshop/publication/book
  where $book/@bookid = 1
  return $book/title
}
</root>
```

Query 7 DataDirect's XQuery-for-relation

4.2 JSON-oriented extensions

Navigational queries on JSON documents are similar to that on XML or graph databases. Similar to Core-XPath and Core-GXPath, we can also define a JSON navigation logic(JNL) to capturing the navigation capability [20]. We can reuse results devised for other semi-structured languages such as XPath/XQuery, but the nature of JSON and the functionalities present in query languages also demand new approaches or refinement of these techniques. A bunch of query languages has been proposed for JSON data, such as MongoDB's query language [85], JAQL [147], JSONPath [148], JSONiq [149], and SQL++ [108]. To note that there is no standard query language for JSON and most of these languages are inspired either by XPath/XQuery (e.g, JSONPath and JSONiq) or SQL (e.g., SQL++).

4.2.1 Document navigation with JSONPath

JSON documents can be also retrieved in the way of XPath. JSONPath [148] covers the essential parts of XPath 1.0 [40] and hence shares a lot of common characteristics with it. The correspondence of syntax elements between JSONPath and XPath can be found in the language specification [148]. JSONPath expressions always refer to a JSON structure in the same way as XPath expressions are used in combination with an XML document. The "root member object" in JSONPath is always referred to as $ regardless if it is an object or array.

JSONPath expressions can use the **dot-notation** "`.store.book[0].title`" and the **bracket-notation** "`['store']['book'][0]['title']`" to formulate path queries as well. Internal or output paths will always be converted to the more general bracket notation. JSONPath allows the wildcard symbol * for member names and array indices. It borrows the descendant operator '.' from E4X and the array slice syntax proposal `[start:end:step]` from ECMASCRIPT 4. Expressions of the underlying scripting language (<`expr`>) can be used as an alternative to explicit names or indices as in "`.store.book[(@.length-1)].title`". Filter expressions are supported via the syntax ?(<boolean expr>) as in "`.store.book[?(@.price < 10)].title`".

4.2.2 Querying document data with JSONiq

JSONiq [149–151] is a query language that mimics XQuery. It borrows ideas from XQuery, such as the structure and semantics of a `FLWOR` construct, the language's functional aspect, the semantics of comparisons in the face of data heterogeneity, and declarative, snapshot-based updates. For example, Query 8 is used to calculate the average score for each question in a test.

```
for $answer in collection("answers")
  let $qid := $answer.question_id
  groupby $qid
  where count($answer) gt 1
  return {
    "question": $qid,
    "count": avg($answer.score)
  }
```

Query 8 JSONiq FLWOR expression

5 Graph extensions

In this section, we present the graph extensions toward multi-model data.

5.1 Graph query languages

A variety of graph languages have been proposed for querying graph data in various applications such as knowledge graphs, social networks, and real-time road networks [9–11, 25, 32, 54, 61, 144, 152–154]. Most of the relatively early graph languages originated from the research community such as Lorel [38], StruQL [155], UnQL [156], G [157], and GraphLog [50]. The most recent ones are mainly from the industry, including W3C's SPARQL [59–61], Cypher/openCypher [29, 62, 117, 118, 158], TinkerPop's Gremlin [63, 64], Oracle's PGQL [65, 66], LDBC's G-CORE [67], and TigerGraph's GSQL [68, 69]. The modern graph query languages are proposed and implemented for interrogating specific graph data models—SPARQL for RDF graphs and the rest for property graphs respectively. SPARQL [59, 60] and Cypher [29, 62, 117] are the standard query languages for the RDF graph and property graph respectively, so we concentrate on the extensions of SPARQL and Cypher toward multi-model data.

The graph query languages are mainly based on the notation of *graph pattern matching*, with which we can express graph patterns and path queries against the data. Table 5 shows the core query syntax of property graph query languages—they all have SQL-like syntax and SQL-like functionalities. The queries given in this table are to list the pairs of persons who are friends or friend-of-friend along with the `:follows` relationship (of length 2) and one person in each pair had commented on the other's post. PGQL pioneered support for full regular path expressions [66].

Cypher was influenced by XPath [40] and SPARQL [43, 59], but it only supports a restricted form of RPQs: the concatenation and disjunction of single relationship types, as well as variable length paths (i.e., transitive closure). For example, `(a:Person)-[:knows*2]->(b:Person)` describes paths of fixed length of 2, `(a:Person)-[:knows*3..5]->(b:Person)` represents paths have variable lengths from 3 to 5, and `(a:Person)-[:knows*]->(b:Personb)` stands for paths of any length. GSQL and G-CORE combine the ASCII-art syntax from Cypher and the regular path expression syntax from PGQL. With the PGQL's regular path expression, we can express RPQ and CRPQ queries. By using the boolean operators such as conjunction ($A\&B$), disjunction ($A|B$), negation ($!A$), and grouping/nesting ($(A\&B)|C$), we can express even more complex graph queries. In addition, we can issue 2CRPQs by removing the direction of relationships, e.g., `(a)-[:knows]-(b)` implies `a` and `b` know each other.

5.2 SPARQL extensions

RDF is the standard data model for Semantic Web and is widely adopted by knowledge graphs. SPARQL [59, 60, 115] is the standard query language for RDF databases. We refer the readers to Karvounarakis et al.'s paper [159] and Haase et al.'s paper [160] for a comparison of RDF query languages. Syntactically, SPARQL adopts the SELECT-FROM-WHERE query structure and uses one or more PREFIX(s) to define the abbreviations of resource URIs. Particularly, the SELECT clause returns a table of variables and values that satisfy the query; each variable is a string starting with a question mark "?". The FROM clause states the targeted RDF graph to be queried. The WHERE clause specifies the query pattern to the targeted RDF graph; each pattern consists of a set of triple patterns. Since SPARQL 1.1 [58, 115], it can supports complex queries such as RPQs [11] and graph patterns matching.

So far, a handful of extensions SPARQL are available for multi-model data, including relational, key-value, GeoSpatial, and XML data. Table 6 summarizes supported data models, query languages, and new functionalities of these extensions. In the following, we introduce these extensions in detail.

(1) Extending SPARQL with SQL SPARQL allows one to call built-in SQL functions or stored procedures in the SELECT and WHERE clauses. The call starts with a prefix *sql:* before a function name. In this way, one can blend SQL and SPARQL queries for richer data access. As we can see from Query 9, the Virtuoso[2] data system allows to call a SQL function *ComposeInfo* in the SELECT clause for concatenating the bindings of first name and last name from the Berners–Lee person graph.

[2] https://virtuoso.openlinksw.com/.

Table 5 Core query syntax of PGQL, Cypher, G-CORE, and GSQL

Language	Core query syntax and Example query
PGQL	**SELECT-FROM-MATCH-WHERE**
	SELECT a.name, b.name, x.id
	FROM social_network
	MATCH (a:Person)-/:follows(1,2)/->(b:Person)
	WHERE (a)-[:hasPost]->(x:Post)<-[:comment]-(b)
Cypher	**FROM-MATCH-WHERE-RETURN**
	FROM social_network
	MATCH (a:Person)-[:follows*1..2]->(b:Person)
	WHERE (a)-[:hasPost]->(x:Post)<-[:comment]-(b)
	RETURN a.name, b.name, x.id
G-CORE	**SELECT-MATCH-ON-WHERE**
	SELECT a.name, b.name
	MATCH (a:Person)-[:follows*1..2]->(b:Person) ON social_network
	WHERE EXISTS (
	CONSTRUCT ()
	MATCH (a)-[:hasPost]->(x:Post)<-[:comment]-(b))
GSQL	**SELECT-FROM-WHERE**
	SELECT a.name, b.age, x.id
	FROM social_network
	WHERE (a:Person) -/:follows*1..2/-> (b:Person)
	(a) -[:hasPost]-> (x:Post)
	(b) -[:comment]-> (x)
	ORDER BY a.name

Table 6 A summary of SPARQL extensions

Multi-model data	Query language	Functionality
RDF+Relational	SPARQL+SQL	SQL procedures and aggregation
RDF+Text	SPARQL+Full-Text Search	Keyword search over RDF triples
RDF+XML	SPARQL+XQuery	FLWOR expressions on RDF
RDF+Geo-Spatial	SPARQL+Geo-Spatial	Operations on geometric objects

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns\#>
SELECT (sql:ComposeInfo (?fName,?lName))
FROM <http://www.w3.org/People/Berners-Lee/card>
WHERE{
  ?s rdf:type foaf:Person .
  optional{?s foaf:firstName ?fName }.
  optional{?s foaf:lastName ?lName }.
  filter (?fName like '%Tim%') .};
```

Query 9 Call a SQL function in the SPARQL query

The query results of SPARQL are tuple-like triples, so we can easily integrate them with the aggregation functionalities. Virtuoso's SPARQL supports GROUP BY, ORDER BY, LIMIT, and OFFSET. It also provides a set of aggregation functions (e.g., COUNT, MIN, MAX, AVG, and SUM) in the result clause. Query 10 is to list out the town or city in the UK that has the largest proportion of students, where the GROUP BY clause operates on two variables: town and graduate, and the ORDER BY clause sorts the number of graduates by town in descending order.

```
SELECT ?town, Count(*)
WHERE{
  ?uni dbpedia-owl-inst:country dbpedia:United_Kingdom;
  dbpedia-owl-uni:grad ?graduate;
  dbpedia-owl-inst:city ?town;
}
GROUP BY ?town ?grad
ORDER BY DESC
```

Query 10 A SPARQL Query with SQL-like Aggregation

(2) Extending SPARQL with full-text search The second extension to SPARQL is a full-text search, which has been supported by two RDF-based databases: Virtuoso and Semantic Server.[3] For example, Query 11 would match all subjects whose foaf:Name starts with Tim. In Virtuoso, we can add an index on the RDF graph by defining a rule *DB.DBA.RDF_OBJ_FT_RULE_ADD* over the target variable. The function takes three arguments that define the IRI's for the RDF graph, the target predicate, and the application name. If NULL is given then all graphs or predicates match. To invoke the created full-text index, the built-in function *bif:contains* needs to be used with target text for retrieving the wanted triples.

```
DB.DBA.RDF_OBJ_FT_RULE_ADD
  ('http://www.w3.org/people#', name, 'people');
SELECT *
FROM <http://www.w3.org/people#>
WHERE { ?s foaf:Name ?name . ?name bif:contains "Tim*'".}
```

Query 11 SPARQL+Full-text Search

[3] http://www.intellidimension.com/developers/library/sparql-extensions.aspx#functions.

Table 7 SPARQL extensions to GeoSpatial data

Supported data types	Extended functions
Point, Multipoint, LineString, MultiLineString, Polygon, MultiPolygon, GeometryCollection	equals, disjoint, intersects, touches, within, contains, overlaps, crosses, nearby, distance

(3) Extending SPARQL with Geo-Spatial data This extension comes from Virtuoso 7.1.[4] Table 7 summarizes the extension for Geo-Spatial data, including common geometric data types such as point, line, and polygon, and functions for querying these objects. Query 12 is to find the geometry objects in the dataset that intersects the given polygon object.

```
SELECT  ?s, ?p
FROM <http://linkedgeodata.org/>
WHERE {
  ?s  <http://www.w3.org/2003/01/geo/wgs84_pos#geometry>  ?p .
  FILTER
  ( bif:st_intersects
  ( bif:st_geomfromtext(
  "POLYGON((1-2, 6-1, 9-3, 8-5, 3-6, 1-2))" ), ?p ) )}
```

Query 12 SPARQL+Geo-Spatial

GeoSPARQL[5] is another geographic query language for RDF which defines a vocabulary for representing geospatial data in RDF. In particular, it introduces an extension to the SPARQL query language for processing geospatial data. Query 13 is to list the nearby objects within 1 km of a given location.

```
SELECT *
WHERE{
  ?object spatial:nearby(40.74, -73.989, 1 km).
  ?object rdfs:label ?label}
```

Query 13 SPARQL+Geo-Spatial

(4) Extending SPARQL with XQuery The extension comes from XSPARQL [161]—a query language that integrates XQuery to SPARQL for transformations between RDF and XML. As shown in the following abstractions of XSPARQL, the **DWMC** (Dataset, Where, Modifier, Construct) syntax of SPARQL is extended with **FLWOR** (For, Let, Where, Order, Return) expressions of XQuery. XSPARQL introduces two concepts of "lifting" and "lowering" which translate data from XML to RDF and vice versa. In particular, the construct (C) clause in XSPARQL transforms the XML data to an output RDF graph for the "lifting".

[4] Virtuoso 7.1. https://docs.openlinksw.com/virtuoso/virtclientrefintro/.

[5] GeoSPARQL. https://www.ogc.org/standards/geosparql.

Table 8 A summary of cypher extensions

Data models	Language	Functionality
Property graph + Relational	Cypher + SQL	SQL-in-Cypher, Cypher-in-SQL
Property graph + ML models	Cypher + UDF/Procedure	Regression, Classification, Graph embeddings

```
SPARQL with XQuery in XSPARQL
F  for var in XPath-expression
L let var := XPath-expression
W where XPath-expression
O order by expression or F' for varlist
 --  D from / from named <dataset-URI>
 --  W where { pattern }
 --  M order by expression limit
 --  C construct { template (with nested XSPARQL) }
R return XML+ nested XSPARQL
```

Query 14 SPARQL+XQuery

5.3 Cypher extensions

Cypher is designed for querying property graphs. Syntactically, it adopts the MATCH-WHERE-RETURN syntax, where the MATCH clause specifies the query pattern, the WHERE clause includes filters for labels and properties, and the RETURN clause retrieves the intended results. Regarding path queries, Cypher allows transitive closure (recursive operator $*$) over a single edge label in a property graph, as well as the shortest paths between two nodes. Semantically, Cypher follows a non-repeated-edges isomorphism semantics [11], where the edge variables must have one-to-one relationships but node variables can have repeated bindings in the results. Table 8 summarizes the up-to-date Cypher extensions toward relational data and machine learning models. We proceed to discuss the two extensions in detail.

(1) Extending cypher with SQL This extension originates from AgensGraph[6] and Apache AGE,[7] where both are built on top of PostgreSQL to provide graph database functionalities—mapping the graph to the vertex and edge tables through a Cypher query engine. Therefore, AgensGraph and Apache AGE can ingest Cypher queries, SQL queries, or a hybrid of them. Typically, a hybrid query performs aggregation and statistical processing on tables and columns by using SQL query, and the Cypher query to replace the relational Join operations. Specifically, Cypher and SQL are mixed together via the following two ways: (1) embedding Cypher in a SQL query (i.e., Cypher-in-SQL), or (2) embedding SQL in a Cypher query (i.e., SQL-in-Cypher). AgensGraph supports both manners but Apache AGE supports only the first one. The ways to extend Cypher with SQL in AgensGraph and Apache AGE are much simple than that used by SQL/PGQL and GQL (see Sect. 3.3 for details).

[6] AgensGraph. https://bitnine.net/agensgraph/.

[7] Apache AGE. https://age.apache.org/.

Cypher-in-SQL Since the result of a Cypher query is a relation, we can directly place it in the FROM clause of a SQL query. The syntax is shown in Query 15. In this way, we are able to use Cypher syntax inside the FROM clause to utilize a set of vertex or edges stored in a graph database as data in a SQL statement.

```
SELECT [column_name]
FROM ({table_name | SQLquery | CYPHERquery})
WHERE [column_name operator value];
```

Query 15 Cypher in SQL

SQL-in-Cypher The syntax is shown in Query 16, in which a SQL query is placed in the WHERE clause of a Cypher query as an input for the filters of pattern matching. When querying the content of a graph database with Cypher queries, one can use the SQL query to search specific data from a relational database. However, the usage of SQL-in-Cypher queries is restricted since the result of the SQL query can only be a single row of results.

```
MATCH [table_name]
WHERE [column_name operator {value | SQLquery | CYPHERquery}]
RETURN [column_name];
```

Query 16 SQL in Cypher

(2) Extending cypher with ML models As machine learning (ML) techniques become more and more important in making sense of data, an emerging extension of Cypher is to support ML models. This is done by defining user-defined procedures in Cypher queries and the procedures can be deployed to Neo4j as plugins. As shown in Query 17, a predefined logistic regression model *iris* is used to predict the unknown species of the flowers based on four features.

```
MATCH (iris:Iris) WHERE NOT EXISTS (iris.species)
SET iris.predicted_species =  regression.logistic.predict (‘iris’,
  {‘x1’:iris.petal_length, ‘x2’:iris.petal_width,
   ‘x3’:iris.sepal_length, ‘x4’:iris.sepal_width})
```

Query 17 Cypher+ML models

6 Native multi-model query languages

In recent years, there have emerged several native MMDBs [3]. The native MMDBs treat the supported data models as the “first-class citizen” and implement native multi-model query languages for issuing MMQs, as shown in Table 9. In contrast to single-model-based query languages like SQL, XQuery, and Cypher, they deliver new query syntax for manipulating multi-model data. Moreover, their processing diagrams are different from the aforementioned multi-model query languages, from the expressive power to query evaluation. In this section, we introduce three representatives. Namely, ArangoDB Query Language (AQL), OrientDB Query Language (OrientQL), and Kusto Query Language (KQL).

Table 9 A summary of native multi-model query languages

Query language	Supported data models	Query syntax
AQL (ArangoDB)	Key-Value, JSON, Graph, GeoSpatial	FFR (For, Filter, Return)
OrientQL (OrientDB)	Key-Value, JSON, Graph	SQL-like syntax with dot(.)
KQL (Kusto)	Relational, Time series, GeoSpatial	Dataflow operators with pipe(|)

Table 10 Multi-model functionalities of AQL

Data models	Syntax/Operator
Key-Value	Return (@collection, @key)
JSON	FOR @doc IN @collection FILTER Boolean_function(@doc) RETURN @doc
Graph	FOR @vertex,@edge,@path IN @min..@max OUTBOUND|INBOUND|ANY @ startVertex GRAPH @graph RETURN {@vertex, @edge, @path}
GeoSpatial	GEO_LINESTRING(), GEO_MULTILINESTRING(), GEO_MULTIPOINT(), GEO_POINT(), GEO_POLYGON(), GEO_MULTIPOLYGON(),DISTANCE(), GEO_CONTAINS(), GEO_DISTANCE(), GEO_AREA(),GEO_EQUALS(), GEO_ INTERSECTS(), IS_IN_POLYGON()

6.1 ArangoDB query language

ArangoDB Query Language (AQL) [36] is a declarative query language developed by ArangoDB, which is a multi-model database supporting key-value, document, and graph models. Particularly, it is implemented on a key-value storage engine, *RocksDB*. The documents in ArangoDB follow the JSON format, and they are organized and grouped into collections. Moreover, the graph data is stored in vertex and edge collections as well. Table 10 gives the multi-model functionalities supported by AQL.

AQL is a pure data manipulation language (DML) allowing for operations such as selection, filtering, projections, aggregation, and joining. The basic building block of AQL is the `FOR-FILTER-RETURN` (FFR) expressions where the *FOR* operation is used for data iteration, *FILTER* for data filtering and joining, and *RETURN* for data projection and returning. In particular, the results can be returned as JSON or table, or be visualized as graphs. In addition to advanced array operations, AQL also provides the syntax of graph traversal for named graphs and edge collections using the FFR expressions. Such a feature allows for navigational path queries, pattern matching queries, and shortest path queries expressed in AQL. Interestingly, AQL can also specify a single statement to retrieve and combine data from JSON, key-value, and graph. Query 18 shows an example of AQL query, in which we can naturally handle cross-model queries.

Table 11 Multi-model functionalities of OrientQL

Data models	Syntax/Operator
Key-Value	SELECT * FROM #rid
JSON	SELECT [<Projections>] [FROM <Target>] [WHERE <Condition>] [GROUP BY <Field>][ORDER BY <Fields>] [UNWIND <Field>]
Graph	TRAVERSE < relationship> [FROM <target>] [MAXDEPTH <number> | WHILE < condition>] [STRATEGY <strategy>]
Geo-Spatial	Point(), Line(), Polygon(), MultiPoint(), MultiLine(), MultiPolygon(), ST_AsText(), ST_GeomFromText(), ST_AsGeoJSON(), ST_GeomFromGeoJSON(), ST_Within(), ST_Contains(), ST_Disjoint(), ST_Intersects(), ST_AsBinary()

```
FOR friend IN 1..3 OUTBOUND PersonId/1 KnowsGraph
  FOR order IN Order
    FOR feedback IN Feedback
    FILTER order.customer_id==friend._id AND
           "Nike" IN order.items[*].brand AND
           friend._id==feedback.custID AND feedback.Rating==5 AND
           feedback.productId IN order.items[*].productId
    RETURN {person:friend, feedback:feedback}
```

Query 18 UniBench Q5 in AQL

The above query is from UniBench Q5, which finds a given person (id=1)'s 3-hop friends and feedback. These friends have bought products with a given brand b, i.e., `name="Nike"`, and the feedback should have 5-star ratings. Specifically, line 1–3 operates on the graph, JSON, and key-value data, respectively. In the graph traversal, the min and max length is 1 and 3, the direction is outbound from the starting person with Id 1 in the "knowsgraph" which is an edge collection. The filtering conditions and equijoin predicates are specified in the *Filter* clause accordingly during lines 4–6. In line 4, the JSON orders are joined with persons in the graph on Ids. line 5, it uses the *[*]* operator to access to the *brand* element in *items* array. In line 6, the persons in the graph are joined with feedback that has a 5-score rating. Finally, the filtered persons and feedback are returned as a JSON array.

6.2 OrientDB query language

OrientDB Query Language (OrientQL) [87] is a declarative multi-model query language developed by OrientDB, which utilizes the object records to persist the data and links the records by pointers. Particularly, an object record can be a document, a bytes record (BLOB), a vertex, and an edge. Consequently, it can support the data models that include document, graph, object, and key-value models simultaneously. Table 11 gives the multi-model functionalities supported by OrientQL.

OrientQL is a SQL-like language with a `SELECT-FROM-WHERE` structure. However, it does not completely follow the standard SQL syntax. For instance, the JOIN syntax is not supported while relationships are represented by links. Thus, it uses the dot (.) notation to navigate links or embedded documents. For the graph traversal, it provides a TRAVERSE command that recursively navigates the graph in either depth-first or breadth-first search. Query 19 illustrates an implementation of UniBench Q5 in OrientQL's syntax.

```
SELECT person, person.feedback
FROM (
   TRAVERSE Expand(Out("Knows"))
   FROM person
   WHERE PersonId=56 and $depth<3)
WHERE "Nike" in Order.items.brand and feedback.Rating==5
UNWIND Order.items
```

Query 19 An OrientDB SQL query

The query combines data from tabular, graph, and JSON. Specifically, in line 1, the SELECT clause returns the person and feedback. In lines 3–5, it leverages the TRAVERSE-FROM-WHERE syntax to traverse the graph with the start vertex (PersonId=56) within a max traversal depth of 3. In line 6, two filtering conditions are specified in the WHERE clause. In particular, the Order is associated with the person implicitly because the person is the source table in FROM clause. The nested brand is accessed via a chain of path-oriented dot operations. In line 7, it flattens the embedded JSON document to multiple single-row documents by using the UNWIND operator.

6.3 Kusto query language (KQL)

Kusto Query Language (KQL)[8] is a declarative query language introduced by Microsoft Azure Data Explorer. It is a scalable data analytics service in Microsoft Azure Cloud, aiming for interactive analysis of big data. KQL adopts a pipelined syntax with a sequence of operators to filter, transform, join, and aggregate the data. Particularly, KQL has a number of operators for tabular, time series, and Geo-Spatial data analysis. Table 12 lists the multi-model functionalities supported by KQL.

Query 20 is an example of KQL query for analyzing the data stored in a table called StormEvents. Specifically, line 1 declares the data source. Line 2 defines a function named `distance_1_to_100 m` to compute the shortest distance between two geospatial coordinates. Line 3 gives the range of the distance that should be between 1 and 100 m. Line 4 contains another filtering condition in the where operator that requires the events should take place between 2007-11-01 and 2007-12-01. Finally, the last line projects the results on three columns: distance_1_to_100m, State, and EventType. In addition to the single-source analysis, KQL supports cross-database and cross-cluster queries for complex data analysis.

[8] https://docs.microsoft.com/en-us/sharepoint/dev/general-development/keyword-query-language-kql-syntax-reference.

Table 12 Multi-model functionalities of KQL

Data models	Syntax/Operator
Relational	Source Table 1 | [Tabular operators*] [union | join] [Source Table n | [Tabular operators*][union | join]]*
Time series	make-series, avg(), count(), max(), min(), percentile(), stdev(), sum(), variance(), series_fir(), series_iir(), series_fit_line(), series_outliers(), series_periods_detect(), series_stats(), series_fill_backward(), series_fill_const(), series_fill_forward(),series_fill_linear()
Geo-Spatial	geo_distance_2points(),geo_distance_point_to_line(),geo_line_densify(), geo_point_in_circle(),geo_point_in_polygon(),geo_point_to_geohash(), geo_point_to_s2cell(),geo_polygon_to_s2cells(),geo_polygon_densify()

```
StormEvents
| EXTEND distance_1_to_100m = geo_distance_2points
(BeginLon, BeginLat, EndLon, EndLat)
| WHERE distance_1_to_100m between (1 .. 100)
| WHERE StartTime >= datetime(2007-11-01)
AND StartTime < datetime(2007-12-01)
| PROJECT distance_1_to_100m, State, EventType
```

Query 20 A KQL Multi-Model Query

7 Open challenges and problems

In contrast to the single-model-based query languages (e.g., SQL), most of the MMQLs we have investigated are built for practical purposes and thus lack of solid foundation. There are many open challenges and problems related to these languages. In this section, we show a compiled list of these challenges.

(1) Expressivity The most important feature of an MMQL is the capability to express multi-model queries. An open problem is to determine the expressivity of the MMQL—how much the mixtures of the fundamental queries can be expressed with the language. Unfortunately, none of the existing MMQLs are full-fledged to support all the fundamental queries listed in Sect. 2.2. Most of the existing MMQLs, including the native ones, provide support for multi-model data by introducing various informal operations, e.g., the *traversal operations* for document data and the *matching operations* for graph data, whose expressivities still need better investigation. Moreover, there is no accepted notion of completeness for document and graph query languages. The relational completeness was established by the relational algebra (or calculus) [93], but we do not have that counterpart in any existing MMQLs. It has a serious restriction on the application of these languages, and so far we haven't seen any measurement of the expressive power of MMQLs.

(2) Universal data model The second challenge is about designing a universal data model to encapsulate the difference between multi-model data. The existing NoSQL data models can be viewed as extensions or simplifications of the relational model. In general, we have several ways to extend the relational model [5]: (1) The simplest way is to add new data types into the primitive type set, such as literal, hypertext, and URI. (2) One can also include other type constructors such as lists, multisets, and arrays to generate other data models such as JSON [17, 21, 22]. (3) By repeatedly using the relation constructor, one can yield the *nested relational model* [162]. The nested relational model (or complex value model) contains nested type constructors that allow building nested relations from atomic types by using tuple constructors and set constructors [162, 163]. (4) By separating the set and tuple of the relation constructor, one can support the complex object and hence yields the *object-relation model* [123, 126, 164–166]. In this model, we can integrate structured data with semi-structured data [167].

By mixing the aforementioned methods we can create many of the existing data models. However, it is still a challenging task to create a universal data model for multi-model data. In the MMQLs we have investigated, only SQL++ [109] creates a unified model for both relational data and JSON. The consequence is that the underlying multi-model data cannot be processed in a unified logic and hence has a significant impact on query processing and optimization. In recent years, several research groups (e.g., Fleming et al. [168], Spivak et al. [169, 170], and Lu et al. [171, 172]) tried applying category theory as a unifying formalism (to create an abstraction from a higher level) to multi-model data. But these methods are still too complicated to be implemented in MMDBs.

(3) Cross-model query processing and optimization Query processing in most MMDB systems relies on data exchange, which can be done in two ways: (1) designating a local data model as the primary model and the rest are translated into the format as the primary model, or (2) creating a super-model to describe the local data models. Most MMDB systems implement the first solution and the relational model is typically taken as the primary model. For instance, in an MMDB extended from a relational DBMS, the NoSQL data involved in a query will be mapped into relational tables according to the specifications in the SQL standard (e.g., SQL/XML, SQL/JSON, and SQL/PGQ). Consequently, we may observe many types of data exchange schemes, i.e., XML-to-relation, JSON-to-relation, and graph-to-relation mappings, in a cross-model query in that database.

Alternatively, Bugiotti et al. were to create an abstract model like NoAM (NoSQL Abstract Model) [173], which is a super-model that specifies the underlying NoSQL databases. However, the proposal covers only a few types of NoSQL databases (i.e., key/value, column, and document). Similarly, Atzeni et al. [174, 175] leverage a meta-model to facilitate the data exchange for multi-model data. The meta-model is a formalism for the definition of data models and uses a small set of commonly used meta-constructs such as lexical types, abstract types, aggregation, and function to define data models. Data exchange in can be accomplished in two steps: (1) from the source model to the supermodel; and (2) from the meta-model to the target model. The super-model/meta-model approach can be used efficiently to facilitate schema mapping, but query processing under this approach still results in significant overheads. Recently, several approaches have been proposed to optimize query evaluation of MMQs, such as efficient enumeration of execution plan [176] and query augmentation that supports automatic enrichment of the answer [177, 178]. Cross-model query processing and optimization of MMQs are still open for MMDBs.

8 Conclusions

It is a challenging task to design a query language for MMDBs that allows users to express multi-model queries. Throughout the previous sections, we investigated a number of multi-model query languages from the syntactical perspective, which can be divided into four types: SQL extensions, XPath/XQuery extensions, graph extensions, and native ones. The investigation of existing multi-model query languages, from the syntactical and application perspectives, makes this article useful for motivating new multi-model query languages, as well as serving as a technical reference for formulating multi-model queries. This survey also shows that the existing MMQLs are still far from a mature query language compared to the single-model-based languages.

Acknowledgements This work was partially funded by the Finnish Academy Project 310321 and the Fundamental Research Program of Shanxi Province, China (Grant No. 20210302123025).

Funding Open Access funding provided by University of Helsinki including Helsinki University Central Hospital.

References

1. Saeed, M., et al.: Multiparameter intelligent monitoring in intensive care II: a public-access intensive care unit database. Crit. Care Med. **39**, 952–960 (2011)
2. Lu, J., Holubová, I.: Multi-model data management: what's new and what's next?, pp. 602–605 (OpenProceedings.org)
3. Lu, J., Holubova, I.: Multi-model databases: a new journey to handle the variety of data. ACM Comput. Surv. **52**, 1–38 (2019)
4. Codd, E.F.: A relational model of data for large shared data banks. Commun. ACM **13**, 377–387 (1970)
5. Scholl, M.H.: Extensions to the relational data model, pp. 163–182 (1992)
6. Schweikardt, N., Schwentick, T.: Database Theory: Query Languages, 2nd edn., p. 39. Chapman and Hall/CRC, Boca Raton (2010)
7. Atzeni, P., Bugiotti, F., Cabibbo, L., Torlone, R.: Data modeling in the NoSQL world. Comput. Stand. Interfaces **67**, 103149 (2020)
8. Angles, R., Gutierrez, C.: Survey of graph database models. ACM Comput. Surv. **40**, 1–39 (2008)
9. Wood, P.T.: Query languages for graph databases. SIGMOD Rec. **41**, 50–60 (2012)
10. Barceló, P.: Querying graph databases, pp. 175–187
11. Angles, R., et al.: Foundations of modern query languages for graph databases. ACM Comput. Surv. **50**, 1–40 (2017)
12. Bondiombouy, C., Valduriez, P.: Query processing in multistore systems: an overview. Int. J. Cloud Comput. **5**, 309–346 (2016)

13. Codd, E.F.: Derivability, redundancy and consistency of relations stored in large data banks. Research Report /RJ /IBM /San Jose, California **RJ599** (1969)
14. Codd, E.F.: Extending the database relational model to capture more meaning. ACM Trans. Database Syst. (TODS) **4**, 397–434 (1979)
15. Atzeni, P., Antonellis, V.D.: Relational Database Theory. The Benjamin/Cummings Publishing Company, San Francisco (1993)
16. Abiteboul, S., Buneman, P., Suciu, D.: Data on the Web: From Relations to Semistructured Data and XML. Morgan Kaufmann, San Francisco (1999)
17. IETF RFC 8259. The JavaScript Object Notation (JSON) Data Interchange Format. https://datatracker.ietf.org/doc/html/rfc7159 (2014)
18. Klarlund, N., Schwentick, T., Suciu, D.: XML: Model, Schemas, Types, Logics, and Queries, 1–41. Springer, Heidelberg (2003)
19. Extensible Markup Language (XML) 1.0 (Fifth Edition). https://www.w3.org/XML/
20. Bourhis, P., Reutter, J.L., Suárez, F., Vrgoc, D.: *JSON:* data model, query languages and schema specification, pp. 123–135
21. ECMA-404. The JSON Data Interchange Standard, 2nd Edition. https://www.json.org/json-en.html (2017)
22. Pezoa, F., Reutter, J.L., Suarez, F., Ugarte, M., Vrgoč, D.: Foundations of JSON Schema, pp. 263–273
23. Baazizi, M.-A., Colazzo, D., Ghelli, G., Sartiani, C.: Schemas and types for json data: From theory to practice, 2060–2063. ACM, New York (2019)
24. Ullman, J.D.: Principles of Database and Knowledge-Base Systems - Volume I: Classical Database Systems. Tech. Rep. (1988)
25. Güting, R.H.: GraphDB: Modeling and Querying Graphs in Databases, pp. 297–308
26. Beeri, C., Fagin, R., Maier, D., Yannakakis, M.: On the desirability of acyclic database schemes. J. ACM **30**, 479–513 (1983)
27. Moffitt, V.Z., Stoyanovich, J.: Temporal Graph Algebra, Vol. Part F1306 (2017)
28. Resource Description Framework (RDF). https://www.w3.org/RDF/ (2004)
29. Group, C.L.: Cypher query language reference version 9. https://s3.amazonaws.com/artifacts.opencypher.org/openCypher9.pdf (2011)
30. Erling, O. et al.: The LDBC social network benchmark: interactive workload, pp. 619–630 (2015)
31. Angles, R.: The property graph database model. Vol. 2100 of *CEUR* Workshop Proceedings (CEUR-WS.org)
32. Bonifati, A., Fletcher, G., Voigt, H., Yakovets, N.: Querying Graphs. Synthesis Lectures on Data Management, vol. 10. Morgan & Claypool Publishers, San Rafael (2018)
33. Lu, J.: Towards Benchmarking Multi-Model Databases
34. Zhang, C., Lu, J., Xu, P., Chen, Y.: UniBench: A Benchmark for Multi-model Database Management Systems, pp. 7–23. Springer, Heidelberg (2018)
35. Zhang, C., Lu, J.: Holistic evaluation in multi-model databases benchmarking. Distrib. Parallel Databases **39**, 1–33 (2021)
36. ArangoDB Query Language(AQL). https://www.arangodb.com/docs/stable/aql/index.html
37. Aho, A.V., Ullman, J.D.: Universality of data retrieval languages. **POPL 79**, 110–120 (1979)
38. Abiteboul, S., Quass, D., Mchugh, J., Widom, J., Wiener, J.L.: The Lorel query language for semistructured data. Int. J. Digit. Libr. **1**, 68–88 (1997)
39. Cattell, R.G.G., Barry, D.K.: The Object Data Standard: ODMG 3.0. Morgan Kaufmann, San Francisco (2000)
40. Clark, J., DeRose, S.: XML Path Language (XPath), Version 1.0, W3C Recommendation. https://www.w3.org/TR/xpath-datamodel-31/ (1999)
41. Boag, S. et al.: XQuery 1.0: An XML Query Language (Second Edition). https://www.w3.org/TR/2010/REC-xquery-20101214/ (2010)
42. XSL Transformations (XSLT): Version 1.0. https://www.w3.org/TR/1999/REC-xslt-19991116
43. Pérez, J., Arenas, M., Gutiérrez, C.: nSPARQL: a navigational language for RDF. J. Web Semant. **8**, 255–270 (2010)
44. Barceló, P., Hurtado, C.A., Libkin, L., Wood, P.T.: Expressive languages for path queries over graph-structured data. ACM Trans. Database Syst. **37**, 3–14 (2012)
45. Figueira, D.: Foundations of Graph Path Query Languages (Course Notes). Reasoning Web Summer School, Leuven, Belgium. hal-03349901v2 (2021)

46. Mendelzon, A.O., Wood, P.T.: Finding regular simple paths in graph databases. SIAM J. Comput. **24**, 1235–1258 (1995)
47. Wang, G., Liu, M.: Query Processing and Optimization for Regular Path Expressions, vol. 2681, pp. 30–45. Springer, Heidelberg (2003)
48. Abiteboul, S.: Querying Semi-Structured Data. Lecture Notes in Computer Science, vol. 1186, pp. 1–18. Springer, Heidelberg (1997)
49. ten Cate, B., Marx, M.: Navigational XPath: calculus and algebra. SIGMOD Rec. **36**, 19–26 (2007)
50. Cruz, I.F., Mendelzon, A.O., Wood, P.T.: A graphical query language supporting recursion. ACM SIGMOD Rec. **16**, 323–330 (1987)
51. Calvanese, D., Giacomo, G.D., Lenzerini, M., Vardi, M.Y.: Rewriting of regular expressions and regular path queries, pp. 194–204
52. Calvanese, D., Giacomo, G.D., Lenzerini, M., Vardi, M.Y.: Rewriting of regular expressions and regular path queries. J. Comput. Syst. Sci. **64**, 443–465 (2002)
53. Vardi, M.Y.: A Theory of Regular Queries, 1–9. ACM, New York (2016)
54. Barceló, P., Libkin, L., Reutter, J.L.: Querying regular graph patterns. J. ACM **61**, 8:1-8:54 (2014)
55. Martinez, P., Lopez, J., Rodriguez, F.J., Wiggins, J.B., Boyer, K.E.: An algorithm for context-free path queries over graph databases (2020)
56. Calvanese, D., Giacomo, G.D., Lenzerini, M., Vardi, M.Y.: Containment of Conjunctive Regular Path Queries with Inverse, pp. 176–185. Morgan Kaufmann, San Francisco (2000)
57. Kostylev, E.V., Reutter, J.L., Romero, M., Vrgoc, D.: SPARQL with Property Paths. Lecture Notes in Computer Science, vol. 9366, pp. 3–18. Springer, Heidelberg (2015)
58. SPARQL 1.1 Property Paths. https://www.w3.org/2009/sparql/docs/property-paths/Overview.xml#define-property-paths (2010)
59. Harris, S., Seaborne, A., Prud'hommeaux, E.: SPARQL 1.1 Query Language. W3C recommendation (2013)
60. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. W3C recommendation (2008)
61. Curé, O., Blin, G.: RDF Database Systems: Triples Storage and SPARQL Query Processing. Morgan Kaufmann, San Francisco (2015)
62. Francis, N. et al.: Cypher: An Evolving Query Language for Property Graphs, pp. 1433–1445 (2018)
63. Rodriguez, M.A.: The Gremlin Graph Traversal Machine and Language. https://arxiv.org/abs/1508.03843 (2015)
64. TinkerPop: The Gremlin Graph Traversal Machine and Language. https://tinkerpop.apache.org/gremlin.html (2021). Accessed Oct 2021
65. PGQL 1.1 specification. https://pgql-lang.org/spec/1.1/ (2017)
66. van Rest, O., Hong, S., Kim, J., Meng, X., Chafi, H.: PGQL: A Property Graph Query Language. ACM, New York (2016)
67. Angles, R. et al.: G-CORE: A Core for Future Graph Query Languages, pp. 1421–1432
68. Wu, M., Deutsch, A.: GSQL: An SQL-Inspired Graph Query Language. Tech. Rep. (2018)
69. Deutsch, A., Xu, Y., Wu, M., Lee, V.E. TigerGraph: A Native MPP Graph Database. https://arxiv.org/abs/1901.08248 (2019)
70. Amer-Yahia, S., Cho, S., Lakshmanan, L.V.S., Srivastava, D.: Minimization of Tree Pattern Queries, pp. 497–508 (2001)
71. Czerwinski, W., Martens, W., Niewerth, M., Parys, P.: Optimizing tree patterns for querying graph- and tree-structured data. SIGMOD Rec. **46**, 15–22 (2017)
72. Ullmann, J.R.: An algorithm for subgraph isomorphism. J. ACM **23**, 31–42 (1976)
73. Kelter, U., Däberitz, D.: An Assessment of Non-Standard DBMSs for CASE Environments, pp. 96–113. Springer, Heidelberg (1996)
74. Atzeni, P., Bugiotti, F., Rossi, L.: Uniform access to nosql systems. Inf. Syst. **43**, 117–133 (2014)
75. Özsu, M.T., Valduriez, P.: Principles of Distributed Database Systems, 4th edn. Springer, Heidelberg (2020)
76. Doan, A., Halevy, A.Y., Ives, Z.G.: Principles of Data Integration. Morgan Kaufmann, San Francisco (2012)
77. Ciucanu, R.: Cross-Model Queries and Schemas: Complexity and Learning. Ph.D. thesis, Lille University of Science and Technology, France (2015)
78. DeWitt, D.J., et al.: Split query processing in polybase, pp. 1255–1266

79. Elmore, A.J., et al.: A demonstration of the BigDAWG Polystore system. Proc. VLDB Endow. **8**, 1908–1911 (2015)
80. Duggan, J., et al.: The BigDAWG Polystore system. SIGMOD Rec. **44**, 11–16 (2015)
81. Bondiombouy, C.: Query Processing in Multistore Systems. (Traitement de requêtes dans les systèmes multistores). Ph.D. thesis, University of Montpellier, France (2017)
82. Multimodel Database, White Paper. *ORACLE CORPORATION* 16 (2019)
83. Oracle Database. https://www.oracle.com/database/ (2021)
84. PostgreSQL: The World's Most Advanced Open Source Relational Database. https://www.postgresql.org/ (2021)
85. MongoDB: Build faster! Build smarter!. https://www.mongodb.com/ (2021). Accessed Oct 2021
86. ArangoDB. https://www.arangodb.com/
87. OrientDB: The database designed for the modern world. https://orientdb.com/
88. Kaitoua, A., Rabl, T., Markl, V.: A distributed data exchange engine for polystores. it Inf. Technol. **62**, 145–156 (2020)
89. Fagin, R., Kolaitis, P.G., Miller, R.J., Popa, L.: Data Exchange: Semantics and Query Answering. Lecture Notes in Computer Science, vol. 2572, pp. 207–224. Springer, Heidelberg (2003)
90. Calvanese, D., Giacomo, G.D., Lenzerini, M., Vardi, M.Y.: Query processing under GLAV mappings for relational and graph databases. Proc. VLDB Endow. **6**, 61–72 (2012)
91. Lenzerini, M.: Data integration: a theoretical perspective, pp. 233–246
92. Codd, E.F.: A Data Base Sublanguage Founded on the Relational Calculus, SIGFIDET '71, pp. 35–68. ACM, New York (1971)
93. Codd, E.F.: Relational completeness of data base sublanguages. Research Report /RJ /IBM /San Jose, California **RJ987** (1972)
94. Chamberlin, D.D., Boyce, R.F.: *SEQUEL:* a structured english query language, pp. 249–264
95. Chamberlin, D.D., et al.: SEQUEL 2: A unified approach to data definition, manipulation, and control. IBM J. Res. Dev. **20**, 560–575 (1976)
96. Held, G., Stonebraker, M., Wong, E.: *INGRES:* a relational data base system, Vol. 44, pp. 409–416 (AFIPS Press)
97. Stonebraker, M., Held, G., Wong, E., Kreps, P.: The design and implementation of INGRES. ACM Trans. Database Syst. **1**, 189–222 (1976)
98. Melton, J., Simon, A.R.: Understanding the New SQL: A Complete Guide. Morgan Kaufmann, San Francisco (1993)
99. Committee, I.J.T.: ISO/IEC 9075-4:2011, Information technology–Database languages–SQL–Part 4: Persistent Stored Modules (SQL/PSM). https://www.iso.org/standard/53684.html (2011)
100. ISO/IEC 9075-2:1999 Information technology–Database languages–SQL–Part 2: Foundation (SQL/Foundation). https://www.iso.org/standard/26197.html (1999)
101. Melton, J.: Understanding the New SQL: A Complete Guide, vol. I, 2nd edn. Morgan Kaufmann, San Francisco (2000)
102. ISO/IEC 9075-4:2011, Information technology–Database languages–SQL–Part 4: Persistent Stored Modules (SQL/PSM). https://www.iso.org/standard/53684.html (2011)
103. Committee, I.J.T.: ISO/IEC TR 19075-6:2017(E), Part 6: SQL support for JavaScript Object Notation (JSON). https://www.iso.org/standard/67367.html (2017)
104. ISO/IEC CD 9075-16.2 Information technology–Database languages SQL–Part 16: SQL Property Graph Queries (SQL/PGQ). https://www.w3.org/TR/sparql11-query/ (2019)
105. Plantikow, S., Cannan, S.: GQL Early Working Draft v2.2. https://www.delltechnologies.com/en-us/blog/crap-and-crud-from-database-to-datacloud/ (2019)
106. ISO SC32/WG3: Graph Query Language (GQL) Standard. https://www.gqlstandards.org/home (2021)
107. W3C Workshop on Web Standardization for Graph Data. https://www.w3.org/Data/events/data-ws-2019/report.html (2019)
108. Ong, K.W., Papakonstantinou, Y., Vernoux, R.: The SQL++ semi-structured data model and query language: A capabilities survey of sql-on-hadoop, nosql and newsql databases. https://arxiv.org/abs/1405.3631 (2014)
109. Chamberlin, D.: SQL++ For SQL Users: A Tutorial. Couchbase Inc, Santa Clara (2018)
110. Database Query Language N1QL: Get the familiarity of SQL with the flexibility of JSON. https://www.couchbase.com/products/n1ql (2021)
111. Alsubaiee, S., et al.: AsterixDB: a scalable, open source BDMS. Proc. VLDB Endow. **7**, 1905–1916 (2014)

112. AsterixDB - The SQL++ Query Language. https://asterixdb.apache.org/docs/0.9.3/sqlpp/manual.html (2018)
113. A Direct Mapping of Relational Data to RDF. https://www.w3.org/TR/rdb-direct-mapping/ (2012)
114. Virtuoso: Data-driven agility without compromise. https://virtuoso.openlinksw.com/ (2021)
115. DuCharme, B.: Learning SPARQL: Querying and Updating with SPARQL 1.1. O'Reilly Media Inc, Sebastopol (2013)
116. OpenLink Virtuoso Blog. About linked data, data virtualization, and data flow. https://medium.com/virtuoso-blog (2021)
117. Group, C.L.: Cypher 10 Improvement Proposals. https://github.com/opencypher/openCypher/labels/cypher10 (2017)
118. openCypher query language. https://opencypher.org/ (2016)
119. Lindaaker, T.: An overview of the recent history of Graph Query Languages (2018)
120. Plantikow, S. Summary Chart of Cypher, PGQL, and G-Core. https://s3.amazonaws.com/artifacts.opencypher.org/website/materials/DM32.2/DM32.2-2018-00086r1-summary-chart-of-cypher-pgql-gcore.pdf (2018)
121. Gallagher, B.: Matching structure and semantics: a survey on graph-based pattern matching, Vol. FS-06-02, pp. 45–53
122. Junghanns, M., Kießling, M., Averbuch, A., Petermann, A., Rahm, E.: Cypher-based Graph Pattern Matching in Gradoop, pp. 3:1–3:8
123. Cattell, R.G.G.: The Object Database Standard: ODMG-93. Morgan Kaufmann, San Francisco (1994)
124. Papakonstantinou, Y., Garcia-Molina, H., Widom, J.: Object exchange across heterogeneous information sources, pp. 251–260
125. Protocol Buffers – Google's data interchange format. https://github.com/protocolbuffers/protobuf (2008)
126. Cluet, S.: Designing OQL: allowing objects to be queried. Inf. Syst. **23**, 279–305 (1998)
127. Robie, J., Chamberlin, D., Dyck, M.: XQuery 3.0: An XML Query Language, W3C Recommendation. https://www.w3.org/TR/xquery-30/ (2014)
128. Bry, F., Schaffert, S.: The XML Query Language Xcerpt: Design Principles, Examples, and Semantics, Vol. 2593 of *LNCS*, pp. 295–310 (Springer, Heidelberg, 2002)
129. Hosoya, H., Pierce, B.C.: XDuce: a statically typed XML processing language. ACM Trans. Internet Technol. **3**, 117–148 (2003)
130. Benzaken, V., Castagna, G., Frisch, A.: CDuce: an XML-centric general-purpose language. ACM SIGPLAN Not. **38**, 51–63 (2003)
131. Chen, Z., Ling, T.W., Liu, M., Dobbie, G.: Xtree for Declarative XML Querying. Lecture Notes in Computer Science, vol. 2973, pp. 100–112. Springer, Heidelberg (2004)
132. Berglund, A., et al.: XML Path Language (XPath) 2.0 (Second Edition), W3C Recommendation. https://www.w3.org/TR/xpath20/#XPath (2010)
133. Chamberlin, D.D., Robie, J., Florescu, D.: Quilt: An XML Query Language for Heterogeneous Data Sources. Lecture Notes in Computer Science, vol. 1997, pp. 1–25. Springer, Heidelberg (2000)
134. Kovse, J., Mahnke, W.: Introducing Custom Language Extensions to sql:1999, 193–208. Springer, Heidelberg (2003)
135. Ishikawa, H., Kubota, K., Kanemasa, Y.: A query language for XML data, XQL (1998)
136. Gottlob, G., Koch, C., Pichler, R.: Efficient Algorithms for Processing Xpath Queries, pp. 95–106. Morgan Kaufmann, Hong Kong (2002)
137. Marx, M., de Rijke, M.: Semantic characterizations of navigational xpath. SIGMOD Rec. **34**, 41–46 (2005)
138. Baca, R., et al.: Structural XML query processing. ACM Comput. Surv. **50**, 64:1-64:41 (2017)
139. Lakshmanan, L.V.S., Wang, W.H., Zhao, Z.J.: Answering tree pattern queries using views, pp. 571–582
140. Chen, Z., et al.: Counting Twig Matches in a Tree, pp. 595–604
141. Jagadish, H.V., Lakshmanan, L.V.S., Srivastava, D., Thompson, K.: TAX: A Tree Algebra for XML. Lecture Notes in Computer Science, vol. 2397, pp. 149–164. Springer, Heidelberg (2001)
142. Mendelzon, A.O., Wood, P.T.: Finding Regular Simple Paths in Graph Databases, pp. 185–193
143. Cassidy, S.: Generalizing xpath for directed graphs (2003)
144. Libkin, L., Martens, W., Vrgoč, D.: Querying graphs with data. J. ACM **63**, 14:1-14:53 (2016)

145. Marklogic XQuery. https://docs.marklogic.com/guide/xquery/langoverview#chapter (2021). Accessed Oct 2021
146. DataDirect XQuery. http://media.datadirect.com/download/docs/ddxquery/allddxq/reference/wwhelp/wwhimpl/common/html/wwhelp.htm?context=reference &file=quickstart2.html (2020). Accessed Oct 2021
147. Beyer, K.S., et al.: Jaql: a scripting language for large scale semistructured data analysis. Proc. VLDB Endow. **4**, 1272–1283 (2011)
148. Gössner, S., Frank, S.: JSONPath - XPath for JSON. https://goessner.net/articles/JsonPath/h
149. Robie, J., et al.: JSONiq: The JSON Query Language. https://www.jsoniq.org/ (2016)
150. Florescu, D., Fourny, G.: JSONiq: the history of a query language. IEEE Internet Comput. **17**, 86–90 (2013)
151. Fourny, G. Jsoniq: The sql of nosql (2013)
152. Barceló, P., Libkin, L., Reutter, J.L.: Querying graph patterns, pp. 199–210
153. Barceló, P., Libkin, L., Lin, A.W., Wood, P.T.: Expressive languages for path queries over graph-structured data. ACM Trans. Database Syst. **37**, 31:1-31:46 (2012)
154. Angles, R., Reutter, J., Voigt, H.: Graph query languages. Encyclopedia of Big Data Technologies, pp. 883–890 (2019)
155. Fernández, M.F., Florescu, D., Levy, A.Y., Suciu, D.: Declarative specification of web sites with strudel. VLDB J. **9**, 38–55 (2000)
156. Buneman, P., Fernandez, M.F., Suciu, D.: UnQL: a query language and algebra for semistructured data based on structural recursion. VLDB J. **9**, 76–110 (2000)
157. Consens, M.P., Mendelzon, A.O.: Expressing structural hypertext queries in graphlog, pp. 269–292
158. Marton, J., Szárnyas, G. & Varró, D. *Formalising openCypher Graph Queries in Relational Algebra*, Vol. 10509 LNCS, 182–196 (Springer, Heidelberg, 2017)
159. Karvounarakis, G., Alexaki, S., Christophides, V., Plexousakis, D., Scholl, M.: RQL: a declarative query language for RDF, pp. 592–603
160. Haase, P., Broekstra, J., Eberhart, A., Volz, R.: A comparison of RDF query languages. ISWC **3298**, 502–517 (2004)
161. XSPARQL Language Specification. https://www.w3.org/Submission/xsparql-language-specification/ (2009)
162. Scholl, M.H., Paul, H., Schek, H.: Supporting flat relations by a nested relational kernel, pp. 137–146
163. Wang, H., Zaniolo, C., Luo, C.: *ATLAS:* a small but complete SQL extension for data mining and data streams, pp. 1113–1116 (2003)
164. Abiteboul, S., Beeri, C.: The power of languages for the manipulation of complex objects. Tech. Rep. (1988)
165. Deux, O.: The O2 system. Commun. ACM **34**, 34–48 (1991)
166. Kifer, M., Kim, W., Sagiv, Y.: Querying Object-Oriented Databases, pp. 393–402
167. Lahiri, T., Abiteboul, S., Widom, J.: Ozone: Integrating structured and semistructured data, Vol. 1949, 297–323 (Springer, Heidelberg, 1999)
168. Fleming, M.W., Gunther, R., Rosebrugh, R.D.: A database of categories. J. Symb. Comput. **35**, 127–135 (2003)
169. Spivak, D.I.: Table manipulation in simplicial databases. https://aarxiv.org/abs/1003.2682 (2010)
170. Schultz, P., Spivak, D.I., Vasilakopoulou, C., Wisnesky, R.: Algebraic databases. https://arxiv.org/abs/1602.03501 (2016)
171. Uotila, V., et al.: Multi-model Query Processing Meets Category Theory and Functional Programming
172. Uotila, V., et al.: MultiCategory: multi-model query processing meets category theory and functional programming. Proc. VLDB Endow. **14**, 2663–2666 (2021)
173. Bugiotti, F., Cabibbo, L., Atzeni, P., Torlone, R.: Database design for nosql systems, Vol. 8824 of *LNCS*, pp. 223–231 (Springer, Heidelberg, 2014)
174. Atzeni, P., Torlone, R.: A metamodel approach for the management of multiple models and translation of schemes. Inf. Syst. **18**, 349–362 (1993)
175. Atzeni, P., Gianforme, G., Cappellari, P.: A universal metamodel and its dictionary. Trans. Large Scale Data Knowl. Centered Syst. **1**, 38–62 (2009)
176. Forresi, C., Francia, M., Gallinucci, E., Golfarelli, M.: Optimizing execution plans in a multistore, Vol. 12843 of LNCS, pp. 136–151 (Springer, Heidelberg, 2021)
177. Maccioni, A., Torlone, R.: Augmented access for querying and exploring a polystore, pp. 77–88

178. Maccioni, A., Torlone, R.: Learning How to Optimize Data Access in Polystores, Vol. 11721 of LNCS, pp. 115–127 (Springer, Heidelberg, 2019)