Understanding RDF Data Representations in Triplestores*

Matteo Lissandrini[1], Tomer Sagi[1], Torben Bach Pedersen[1] and Katja Hose[1]

[1]Aalborg University, Denmark

Abstract

Because of the flexibility and expressiveness of their model, Knowledge Graphs (KGs) have attracted increasing interest. These resources are usually represented in RDF and stored in specialized data management systems called triplestores. Yet, while there exists a multitude of such systems, exploiting varying data representation and indexing schemes, it is unclear which of the many design choices are the most effective for a given database and query workload. Thus, first, we introduce a set of 20 access patterns, which we identify within 6 categories, adopted to analyze the needs of a given query workload. Then, we identify a novel three-dimensional design space for RDF data representations built on the dimensions of subdivision, redundancy, and compression of data. This design space maps the trade-offs between different RDF data representations employed to store RDF data within a triplestore. Thus, each of the required access patterns is compared against its compatibility with a given data representation. As we show, this approach allows identifying both the most effective RDF data representation for a given query workload as well as unexplored design solutions.

Keywords

RDF Data Management, Knowledge Graphs, Data Management Systems Design

1. Introduction

Knowledge Graphs (KGs) are nowadays already in widespread use because of their ability to represent entities and their relationships in many domains [2, 3]. KGs are usually represented as RDF data[1]. RDF models a graph as a set of subject-predicate-object triples, such that nodes serve as subjects and objects and edge labels as predicates. RDF data is usually stored in specialized data management systems called triplestores supporting declarative queries written in the SPARQL language. In parallel to the growing interest in KGs, we have seen the emergence of many different triplestore implementations, either from academic prototypes (e.g., RDF-3X), community projects (e.g., JENA TDB), or commercial products (e.g., Virtuoso, and GraphDB). Nonetheless, each system employs its own peculiar set of design choices, each appearing equally compelling at a first glance. Here, we provide a better understanding of the pros and cons of the

SEBD 2022: The 30th Italian Symposium on Advanced Database Systems, June 19-22, 2022, Tirrenia (PI), Italy
∗The full paper was originally published in the VLDB journal [1].
matteo@cs.aau.dk (M. Lissandrini); tsagi@cs.aau.dk (T. Sagi); tbp@cs.aau.dk (T. B. Pedersen); khose@cs.aau.dk (K. Hose)
0000-0001-7922-5998 (M. Lissandrini); 0000-0002-8916-0128 (T. Sagi); 0000-0002-1615-777X (T. B. Pedersen); 0000-0001-7025-8099 (K. Hose)

[1]The *property graph* [4] model has also been proposed, but this model has still not converged in a common standard and is not usually adopted to store KGs.

various design choices w.r.t. the needs of a given use case and query workload. We specifically focus on the existing solutions for storing, indexing, and querying RDF data. Our analysis complements existing surveys [5, 6], which merely provide a taxonomy of the features offered by the systems (e.g., API and data loading facilities) or classify them according to their underlying technology (e.g., relational versus native graph), but fail short in providing a detailed analysis of the design space in which their internal architectures reside. At the same time, this study, also complements existing efforts in improving the performances of these system by optimizing their query processing techniques [7]. Hence, we first characterize the *feature space for query access patterns*, i.e., the logical operations in a query that, given some input values, establish what kind of output values must be returned. Then, we uniquely identify the defining characteristics of existing RDF data representations within a *unifying three-dimensional design space* across three dimensions: *Subdivision, Compression*, and *Redundancy* (SCR). The design space defines the dimensions along which any storage system for RDF data must be designed. Thus, our approach identifies the most suitable data representation for a given access pattern, as well as access patterns that are currently not optimally supported by the existing representations. We thereby make the following contributions: (i) a feature space of 20 access patterns within 6 categories for SPARQL queries (Table 1); (ii) a design space for RDF data representations based on data subdivsion, redundancy, and compression (Table 2), which allows analyzing the trade-offs of each design choice; and finally (iii) a thorough methodology to analyze how different design choices impact system performance when answering access patterns with specific features. In the following, we first introduce the core concepts regarding the RDF data model (Section 2) and the features of the SPARQL query language (Section 3). Then, we present our SCR design space (Section 4) and conclude explaining how it allows deriving a number of important findings about the data representations adopted by existing systems (Section 5).

2. Preliminaries: RDF & SPARQL

The RDF data model revolves around the concepts of RDF triples and RDF graph patterns. An RDF triple, which corresponds to an edge in the graph, is a factual statement consisting of a subject (s), a predicate (p), and an object (o). Subjects and objects, which represent nodes, can be resources identified by International Resource Identifiers (IRI), anonymous nodes identified by internal IDs (called blank nodes), or literals (which can appear only as objects). Thus, an *RDF graph* $\mathcal{G}$ is a set of RDF triples. An RDF graph is queried by issuing a SPARQL query whose answers are computed based on the subgraphs matching the structural and logical constraints it prescribes. SPARQL queries contain one or more *basic graph patterns* (BGPs), which are sets of triples with zero or more of their components replaced with variables from an infinite set of variables $\mathcal{X}$. Thus, triple patterns are (s, p, o) triples, where any position may be replaced by a variable $x \in \mathcal{X}$. We refer to each value in each position as an *atom*. Moreover, there are special types of patterns that match pairs of atoms connected through sequences of edges satisfying a specific set of predicates, e.g., all nodes connected by an arbitrary sequence of edges with `ex:childOf` predicates. These patterns are called *property paths*. They are defined via a specialized form of expressions called *path expressions* (similar to regular expressions) and offer a succinct way to extend matching of triple patterns to paths of arbitrary length [8]. Finally,

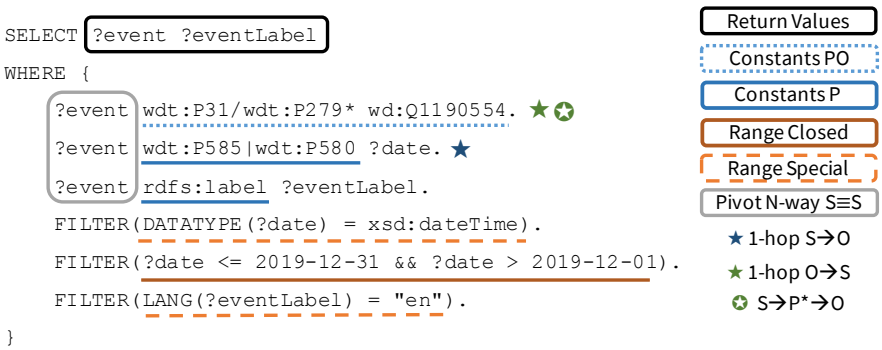

Figure 1: SPARQL query with annotated access patterns.

solutions of a query are the variables *bindings*, i.e., the mappings from each variable to a given node or edge in the graph that satisfies the conditions in the BGP.

In its simplest form, a SPARQL query has the form "SELECT $\vec{V}$ WHERE P", with $P = \{t_1, \dots, t_n\}$ being a set of triple patterns (as in Figure 1, the annotations are explained in the next section). Optionally, one or more FILTER clauses further constrain the variables in P. Let $\mathcal{X}_P$ denote the finite set of variables occurring in P, i.e., $\mathcal{X}_P \subset \mathcal{X}$, then $\vec{V}$ is the vector of variables returned by the query such that $\vec{V} \subseteq \mathcal{X}_P$. Additional operators such as UNION or OPTIONAL allow more than one BGP in a query by defining the non-conjunctive semantics of their combination. Finally, SPARQL queries can also make use of GROUP BY and aggregate operators. SPARQL queries are declarative and are therefore designed to be decoupled from the physical data access methods, i.e., the low-level algorithms and data structures for organizing and accessing data. This decoupling allows specific triplestore implementations to use different data representations and query processing techniques to execute a given query.

3. SPARQL Access Patterns

To identify the best internal representation for an RDF dataset and a given SPARQL query, we decompose the query's triple patterns into atomic operators and identify how they access a given data representation. In particular, we identify a set of *access patterns* that enable the required translation from the declarative query language to its logical query plan. Thus, an *access pattern* is the set of logical operations that, given a set of variable bindings over a graph, determine what data to access (and optionally to change) and what output to produce. The *feature space of access patterns* (summarized in Table 1) is comprised of 6 dimensions, each containing a set of alternative features, namely: **Constants**, the presence of constant values as atoms (instead of variables); **Filter**, the presence (or absence) of a filter on a range of values; **Traversal**, the connection of two nodes by means of one or more subsequent triple patterns; **Pivot**, the atom connecting multiple triple patterns in a BGP; **Return**, the information expected to be returned; **Write,** whether and how a change in the content of the graph is needed.
Consider the example in Figure 1, representing a query to retrieve location coordinates of archaeological sites, and compare it to the access patterns in Table 1. We can recognize, for instance, the property path `wdt:P31/wdt:P279*`, of the form `p1/p2* o`, meaning that it

Table 1
Feature space of access patterns for a SPARQL query.

Dimension	Features				
Constants	S, P, O *1 Constant*		SP, SO, PO *2 Constants*		SPO *3 Constants*
Filter	$<\&>$ *Closed range*		$<\|>$ *Open range*		$\mathbb{T}$ *Special type range*
Traversal	$s \rightarrow o$ *1-hop over* p		$s \xrightarrow{k} o$ *k-hop over* $p_1, \ldots, p_k$		$p^{*/+}$ *?-hop over* p
Pivot	$s \equiv s$ / $o \equiv o$ / $p \equiv p$ binary; same S/P/O position		$o \equiv s$ / $o \equiv p$ / $s \equiv p$ binary; different S/P/O positions		$v_0 \equiv v_1 \wedge v_0 \equiv v_2 \wedge \ldots v_0 \equiv v_N$ N-way; any S/P/O positions
Return	Values (all)	Values (distinct)	Values (sorted)	Existence	Aggregate
Write	Update		Insert		Delete

would match two alternative access patterns: (1) 1-hop traversals over `p1 = wdt:P31` reaching the target node `o=wd:Q1190554` directly, and (2) *-hop traversals starting with one edge for `wdt:P31` and reaching the object through a sequence of arbitrarily long paths matching the `p2 = wdt:P279` triple pattern. On the other hand, the `?event` variable is a 3-way pivot, which can also be executed as a set of binary $s \equiv s$ pivot access patterns. Similarly, we can also identify range filters and the presence of constant values. Each of these access patterns can be effectively implemented and answered by specific RDF data representations and indexes.

4. RDF Data Representations in the SCR Space

To support the data access patterns, triplestores implement a set of different data representations storing all or a subset of the graph $\mathcal{G}$. A data representation $\mathcal{D}$ to answer the given access pattern $\mathcal{A}$ needs to provide a way to retrieve the values for the bindings of the variables $\vec{V}$ from the information stored in $\mathcal{G}$ (or to modify $\mathcal{G}$ for write operations). Therefore, given an access pattern $\mathcal{A}$ and a data representation $\mathcal{D}$ of $\mathcal{G}$, we evaluate the performance of $\mathcal{D}$ in providing the correct instantiations of $\vec{V}$ for $\mathcal{A}$. Moreover, we also evaluate how effective, in terms of time, it is to execute $\mathcal{A}$ over $\mathcal{D}$, i.e., its cost. Answering such a question allows, given two distinct data representations, to select the most appropriate one, i.e., the one with the lowest cost.

Cost model. Answering a query using a particular data representation incurs a cost expressed in terms of the time it takes to retrieve all such answers. Several such cost models have been proposed for RDF [9, 10], but were tied to a simple two-tiered memory hierarchy and assumed a relational data representation. While such assumptions are no longer applicable, it still holds across novel (existing or future) machine architectures and memory types that random seek operations incur a different cost than sequential read operations. Additionally, we identify another set of costs derived from the difference between accessing compressed and uncompressed data. That is, while the use of compression is employed to minimize memory requirements and speed up the movement of large result sets, it incurs additional costs in compression and decompression. Therefore, we follow the recent convention of employing a

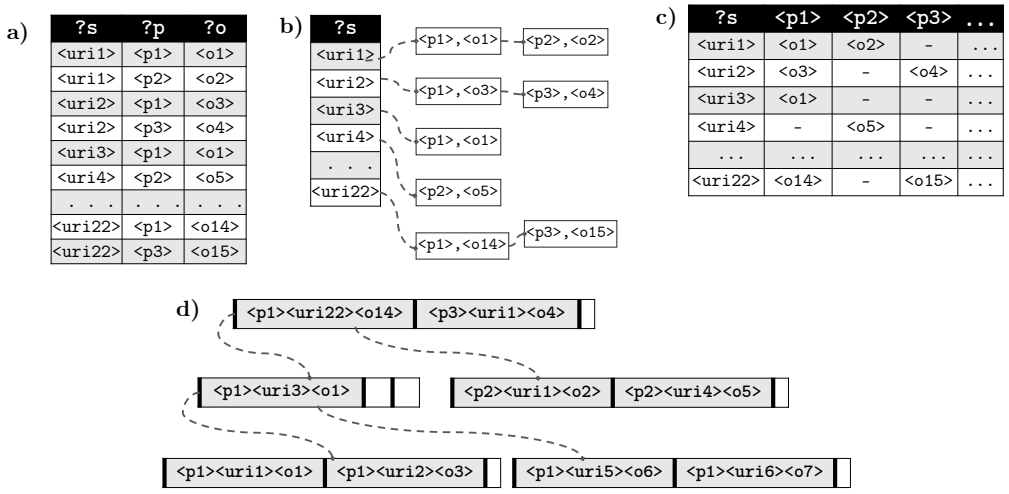

Figure 2: Common representations: a) sorted file, b) hash map, c) table, and d) B+ tree

set of cost constants [11] differentiating between the cost of random and sequential access and between the cost of compressed and uncompressed access.

Data Representation Compatibility. To assess if a data representation can efficiently support a specific access pattern, we define the notion of *compatibility*. We define this notion specifically for read operations, under the assumption that a read operation is required before deletion (to locate the data to be deleted) and before insertion (to locate where to insert the data and to avoid duplicates) and is therefore required for every type of access pattern. Therefore, given an access pattern $\mathcal{A}$ and a data representation $\mathcal{D}$ able to answer $\mathcal{A}$, we assume that there is a sequence of operations specified by an algorithm Γ defined over $\mathcal{D}$ that can compute an answer $\mathcal{S}$, such that $\mathcal{S} = \Gamma(\mathcal{A}, \mathcal{D})$. The number and cost of the operations specified by Γ directly determine whether $\mathcal{D}$ is a representation suitable for efficiently computing the answers to $\mathcal{A}$. When measuring the number of operations required to be executed over $\mathcal{D}$, we distinguish between *random seek* and *sequential* operations. In particular, an RDF data representation can be *seek compatible*, *sequence compatible*, or *selection compatible*.

For instance, one way to store $\mathcal{G}$ is as a list of 3-tuples (one per edge) sorted by subject and then by predicate and object (Figure 2.a and 2.d) with a clustered B+ tree index over them. In this representation, the query processing cost would be analogous to that of a relational table with three attributes (s, p, o), all part of a primary index. This representation is *sequence compatible* with any 1-hop access pattern that binds s, both s and p, or all three positions. That is, the algorithm $\Gamma(\langle s1,p1,?o\rangle, \{B{+}tree,sorted(\mathcal{G})\})$, would first find the first tuple performing $log(\mathcal{G})$ steps traversing the B+ tree looking for *s1, p1*, and then perform a linear scan over the file to retrieve the remaining tuples. However, the B+ tree is not *seek compatible* because the time to find the first result depends on the size of the graph, i.e., it involves $log(|\mathcal{G}|)$ seek operations to reach the first result. A different data representation is to employ a key-value data structure (similar to Figure 2b) and use the pair subject-predicate as the key, and the object as the value. In this data structure, triples sharing the same s and p will store the list of objects contiguously. This data structure is both *seek and sequence compatible* for a traversal that, given s and p, retrieves all corresponding objects. Nevertheless, this representation is neither *seek compatible*

Table 2
Summary of data representation design space axes.

Axis	Minimal Extreme	Maximal Extreme	Positive Effect	Negative Effect
Subdivision	Unsorted file	Pointers between every related data item	↓ # unneeded data items	↑ # random seeks
Compression	No compression	Compress all subdivisions in all representations	↓ # read bytes	↑ Decompression cost
Redundancy	Single representation	One representation for each possible BGP	↓ # random seeks	↑ maintenance cost, storage cost, query optimization time

nor *sequence compatible* if the query requires all edges for predicate p regardless of s. Thus, the definitions of cost and compatibility presented above allow analyzing the advantages and drawbacks of the different choices in each dimension of the design space.

The Design Space Dimensions. When designing data representations for an RDF store, one can model the design decisions over three axes: *Subdivision*, *Compression*, and *Redundancy*. Each of these orthogonal axes, whose properties are summarized in Table 2, represents a continuum along which the data representation adopted by a system can be positioned.

The *Subdivision* axis determines how fragmented the data is. At one extreme, all data is stored contiguously in a single structure; at the other extreme, each edge and node is stored as a separate object with pointers to its neighbors. In between, there are various data structures such as B+ trees or hash maps. This axis contains design decisions such as sorting, grouping, and hashing. Each of these decisions creates an additional subdivision in the data. Increasing the extent of subdivision, allows us to minimize the number of *unneeded* data items accessed to answer an access pattern (resulting in fewer operations that return items not in the result set). For instance, in a sorted file (Figure 2a), the sorting keys act as the simplest subdivision by collecting all triples with the same value together. On the other hand, with a hash table (Figure 2b), given the target IRI, the hash function separates all relevant triples sharing the same key. Therefore, decisions across the subdivision axis easily determine whether a data structure is selection compatible, i.e., if it binds all and only the answers within a specific subdivision, but it also impacts random and sequence compatibility.

The goal of *Compression* is to minimize the number of bits read to reach the first tuple in the result set and the number of bits required to read and potentially store the result set for further processing. The potentially negative impact of compression is, of course, the decompression required to evaluate a predicate in the access pattern if the access pattern is incompatible or partially incompatible. For example, consider a compressed data representation tuned for queries of the form (s, $?p$, $?o$) and $s{\equiv}s$ access patterns (e.g., BitMat [12]). Using this representation to answer an $s\ p^{*/+}\ o$ pattern (i.e., is o reachable from s through edges labeled with p) would require a potentially large number of row decompression operations to identify the edges that satisfy the traversal condition. Moreover, decisions across the compression axis heavily impact the selection compatibility when data that does not contribute to the answer are compressed together with data relevant for an access pattern.

The third axis is *Redundancy* which causes (redundant) copies of the data to be stored in the system. By adding redundant data representations and indexes, it is possible to define ideal (seek,

sequence, and selection compatible) data representations for each access pattern. However, this comes at the cost of having to store the same information multiple times. For example, by holding both an SPO clustered index and a PSO clustered index, each triple is stored twice. Thus, this hinders the compatibility with write access patterns. Moreover, design decisions including full/partial replication need to find a trade-off between storage space and efficient support of query access patterns. Hence, the cost of maintaining multiple representations is three-fold: (i) Increased latency for delete and insert operations with possibly reduced performance of read operations as well. (ii) An increase in space requirements, subsequently straining the limited space in main memory and causing an increase in all read costs. (iii) An increase in query optimization time because alternative structures to access the data result in a higher number of query execution plans that have to be considered.

5. The Capabilities of the SCR Space Analysis

Thanks to the feature space for analyzing query workload access patterns, and the Subdivision-Compression-Redundancy (SCR) design space of data representations for RDF databases, we enable the analysis of RDF store design decisions, specifically, which data representations can effectively support a given workload. Previous workload analyses are centered around those characteristics that would impact mainly the query optimizer. Instead, *we present an analysis of the access patterns specific to the storage layer.* The effect of various choices along the three axes is summarized in Table 2. We performed an analysis of popular RDF store benchmarks under these assumptions and identified a number of workload requirements for which systems rarely provide effective optimizations. Considering the *Subdivision* dimension, many systems implement dedicated tables for triples sharing a specific predicate. Alternatively, queries that match triples with the same subject can usually exploit the clustered representation where several properties for the same subject (or object) are stored together in a single row [13]. Furthermore, a constant in s or o can be exploited by a hash index of the form $s \mapsto po$. Yet, no representation exploits subdivision for a pair of constants, e.g., $sp \mapsto o$. Overall, we see that there is a strong incompatibility between filters on literals and the fact that all representations store literals and special types contiguously. We also perform a similar analysis also across the *Redundancy* dimension. For instance, specialized indexes have been proposed to speed up reachability and multi-hop path expressions (e.g., Sparqling kleene [14]). Yet, in existing systems, data representations supporting the traversal operations required by k-hop and *-hop access patterns are rarely employed. Our survey also reveals the absence of index *Compression* solutions (e.g., [15]). For instance, we do not find any application of effective compression solutions for secondary representations, such as counting indexes (e.g., sp→#) and structures like bloom-filters [16] or compact LSM-tries like SuRF [17].

Thus, researchers can use the SCR design space to design novel solutions in an informed manner. Such designs can be achieved either manually or even semi-automatically as proposed by the recent effort in self-designing data structures [11] and self-organizing database systems [18]. By cross-referencing the access patterns in a query workload with SCR design options, future RDF stores will be able to add components of the system to match the expected workload.

Acknowledgments

This research is supported by the European Union H2020 research and innovation program under the Marie Skłodowska-Curie grant agreement No 838216 and the Poul Due Jensen Foundation.

References

[1] T. Sagi, M. Lissandrini, T. B. Pedersen, K. Hose, A design space for rdf data representations, The VLDB Journal (2022) 1–27.

[2] N. Noy, Y. Gao, A. Jain, A. Narayanan, A. Patterson, J. Taylor, Industry-scale knowledge graphs: Lessons and challenges, ACM Queue 17 (2019) 48–75.

[3] M. Lissandrini, D. Mottin, K. Hose, T. B. Pedersen, Knowledge graph exploration systems: are we lost?, in: CIDR, 2022.

[4] M. Lissandrini, M. Brugnara, Y. Velegrakis, Beyond macrobenchmarks: Microbenchmark-based graph database evaluation, Proc. VLDB Endow. 12 (2018) 390–403.

[5] K. Nitta, I. Savnik, Survey of RDF Storage Managers, in: Proceedings of the International Conference on Advances in Databases, Knowledge, and Data Applications, 2014.

[6] I. Abdelaziz, R. Harbi, Z. Khayyat, P. Kalnis, A survey and experimental comparison of distributed sparql engines for very large rdf data, Proc. VLDB Endow. 10 (2017) 2049–2060.

[7] K. Rabbani, M. Lissandrini, K. Hose, Optimizing SPARQL queries using shape statistics, in: EDBT 2021, OpenProceedings.org, 2021, pp. 505–510.

[8] S. Harris, A. Seaborne, SPARQL 1.1 Query Language. W3C Recommendation, 2012.

[9] R. Cyganiak, A relational algebra for SPARQL, Technical Report, HP Laboratories Bristol, Bristol, UK, 2005.

[10] F. Frasincar, G.-J. Houben, R. Vdovjak, P. Barna, RAL: An Algebra for Querying RDF, WWW 7 (2004) 83–109.

[11] S. Idreos, K. Zoumpatianos, B. Hentschel, M. S. Kester, D. Guo, The data calculator: Data structure design and cost synthesis from first principles and learned cost models, in: SIGMOD, 2018, pp. 535–550.

[12] M. Atre, J. Srinivasan, J. A. Hendler, Bitmat: A main-memory bit matrix of RDF triples for conjunctive triple pattern queries, in: ISWC (Posters & Demonstrations), 2008, pp. 1–2.

[13] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, B. Bhattacharjee, Building an efficient RDF store over a relational database, in: SIGMOD, 2013.

[14] A. Gubichev, S. J. Bedathur, S. Seufert, Sparqling kleene: fast property paths in RDF-3X, in: Workshop on Graph Data Management Experiences and Systems, GRADES, 2013.

[15] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, R. Shen, Reducing the storage overhead of main-memory OLTP databases with hybrid indexes, in: SIGMOD, 2016.

[16] D. Ficara, S. Giordano, G. Procissi, F. Vitucci, Multilayer compressed counting bloom filters, in: Proceedings of the 27th Conference on Computer Communications, IEEE, 2008.

[17] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, A. Pavlo, Surf: Practical range query filtering with fast succinct tries, in: SIGMOD, 2018, pp. 323–336.

[18] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. C. Mowry, M. Perron, I. Quah, S. Santurkar, A. Tomasic, S. Toor, D. V. Aken, Z. Wang, Y. Wu, R. Xian, T. Zhang, Self-driving database management systems, in: CIDR, 2017.