

Workload Matters: Why RDF Databases Need a New Design

Güneş Aluç, M. Tamer Özsu, Khuzaima Daudjee

University of Waterloo David R. Cheriton School of Computer Science
 {galuc,tozsu,kdaudjee}@uwaterloo.ca

ABSTRACT

The Resource Description Framework (RDF) is a standard for conceptually describing data on the Web, and SPARQL is the query language for RDF. As RDF is becoming widely utilized, RDF data management systems are being exposed to more diverse and dynamic workloads. Existing systems are workload-oblivious, and are therefore unable to provide consistently good performance. We propose a vision for a workload-aware and adaptive system. To realize this vision, we re-evaluate relevant existing physical design criteria for RDF and address the resulting set of new challenges.

1. INTRODUCTION

RDF is a schema-free data model in which data are represented as subject-predicate-object (s, p, o) statements called *triples* [16]. The model does not *explicitly* enforce a schema on the data even when the data may be *implicitly* structured. This flexibility makes publishing and linking data across heterogeneous domains much easier, which is an important reason why the Linked Open Data (LOD) cloud has been able to achieve data integration at a very large scale [4]. On the downside, RDF’s flexibility comes at a price: without the schema of the data known upfront, physical RDF database design is difficult, and there is no consensus regarding how RDF data should be represented. For relational implementations, one option is to represent data in a single large table [8] (Fig. 2a) and maintain copies with different sort orders [18, 23]. It has long been argued that grouping data can improve performance for different workloads [5, 21, 24]. Consequently, two other representations were introduced: (i) **grouping by predicates**, where data are partitioned such that there is one table per predicate, and the tables are stored in a column-store [1] (Fig. 2b); and (ii) **grouping by entities**, where *implicit* structures within the data are either manually or automatically discovered, and based on that information a relational schema is computed and data are mapped to multiple relational tables [5, 24] (Fig. 2c). Another alternative is to natively represent the graph structure of RDF data (Fig. 1a) [6, 12, 25]. In this case, **grouping**

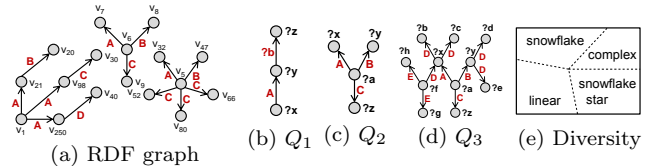


Figure 1: Sample graph, queries and query distribution

by graph vertices can be employed, whereby edges can be grouped by their incidence on a vertex (Fig. 2d).

Although each representation may be suitable for a different type of SPARQL workload, our experiments with state-of-the-art systems demonstrate that no single solution fits all (Table 1). These results show that (i) there is a different winner by a large margin for each query type, and (ii) the winner in one query may unexpectedly timeout in another. These two observations raise questions about the robustness of existing approaches, which is now a well-understood problem for SQL and has led to the introduction of row-stores for OLTP and column-stores for OLAP. Current solutions for RDF and SPARQL fall short for two key reasons: (i) structural diversity and (ii) dynamism in SPARQL workloads. First, (i) web applications that are supported by RDF data management systems are far more varied than conventional relational applications [4], (ii) data that are being handled are far more heterogeneous [10], and (iii) SPARQL is far more flexible in how triple patterns (i.e., the atomic query unit) can be combined [3], which all contribute to *structural diversity*: as shown in Fig. 1e, different parts of the database are queried with different query structures, which can be (i) linear (e.g., Fig. 1b), (ii) star-shaped (e.g., Fig. 1c), (iii) snowflake-shaped (e.g., Fig. 1d), or (iv) an even more complex combination of structures. Second, hotspots in RDF, which denote the RDF resources that are frequently queried, are prominent and have fluctuating phases of popularity. For example, an analysis over real SPARQL query logs reveal that during one week intervals before, during and after a conference, the popularity of the RDF resources related to that conference can significantly peak and then drop [15]. Under these typical web scenarios, queries for which a system performs poorly are inevitable and they may

	linear:L3	star:S3	snowflake:F5	complex:C3
<i>RDF-3x</i>	2.9ms	8.9ms	56.4ms	> 15min
<i>gStore</i>	103.9ms	6.2ms	> 15min	162.7ms
<i>MonetDB</i>	6.0ms	12.9ms	7.8ms	1547.8ms
<i>Virtuoso</i>	3.1ms	347.2ms	9.4ms	103623.7ms
<i>our prototype</i>	0.7ms	8.2ms	3.4ms	9.7ms

Table 1: Snapshot from our experimental results on Waterloo SPARQL Diversity Test Suite [2] at 100 million triples.

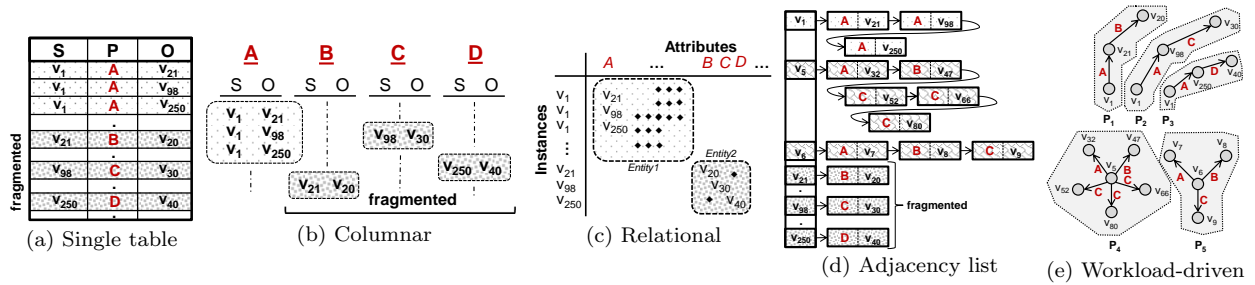


Figure 2: Alternative representations in physical RDF database design

become so frequent that the overall performance of the system for that workload is significantly reduced, even to the extent that the system times out.

In this paper, we propose a vision for a workload-aware and adaptive RDF data management system that addresses the following two problems. First, existing systems are designed and implemented using one of the representations in Fig. 2; however, all of these representations are *workload-oblivious*. As we explain in Section 2, for problematic queries, this can easily lead to (i) data fragmentation, (ii) poor data localization, and (iii) unnecessarily large intermediate results, which are significant performance barriers. Second, none of the existing systems can dynamically switch to a better representation. For systems like *RDF-3x* [18] and *gStore* [25], switching to a new layout is not an option because that would require redesigning and reimplementing the DBMS code from scratch. For systems like *Virtuoso* [11] and *DB2-RDF* [5], which rely on the RDF-to-relational representation, dynamically mapping the data to a new schema is not practical due to the overhead of physical design, where conventional self-tuning techniques are not sufficient. Our experience was that these systems consumed *hours* for loading data and indexing them even for small datasets.

2. PROPOSED APPROACH

We acknowledge that no single system will ever be the winner for every possible SPARQL query; nevertheless, a much more robust solution can be developed. In particular, we envision a system that can *automatically* and *continuously* adapt to changing workloads. To achieve our vision, we (i) argue for a **group-by-query** representation of RDF data that is purely *workload-driven*, and (ii) propose ways of **partially tuning** the database for a lightweight physical design that is easy to update and maintain.

2.1 Group-by-query Representation

In the group-by-query representation, (i) the content of each database record, as well as (ii) the way records are serialized on the storage system are dynamically determined based on the workload (Fig. 2e). Furthermore, the conventions that records must be (i) of fixed-length, and (ii) grouped into tables are relaxed. Consider the dataset and queries in Fig. 1. In the group-by-query representation in Fig. 2, different parts of the database are structured based on the different types of queries in the workload (i.e., Q_1, Q_2, Q_3). For example, P_1-P_3 are the three results of the linear query Q_1 , and P_4-P_5 are structured based on the results of the star-shaped query Q_2 . These records are serialized on disk in the order P_1-P_5 . As we demonstrate below, this can (i) reduce *data fragmentation*, (ii) improve *data localization* (which means that indexes are more efficient in localizing query processing to only the relevant parts of the database)

and (iii) reduce the size of *intermediate results*. Experiments with a preliminary prototype show that these benefits can provide up to $50\times$ improvement in performance [2].

Defragmentation: Consider Q_1 . In all four workload-oblivious representations, random I/O due to fragmentation is inevitable. In the single table representation, the results of the triple pattern $(?y, ?b, ?z)$ are fragmented across the table (Fig. 2a). Since this table is clustered on disk according to attribute S, data are also physically fragmented, which would also be true for any possible clustering proposed by RDF-3x. In the columnar representation, these data are fragmented across multiple columns such that they would actually be stored in different files (Fig. 2b). In the adjacency list representation, vertices v_{21}, v_{98}, v_{250} are fragmented across the hash buckets (Fig. 2d), hence, similar arguments can be made about physical layout. Whether data relevant to this query are fragmented in the relational representations depends on the schema of the database (Fig. 2c), which is determined based on the entities in the dataset but not the workload [5]. In contrast, for the layout in Fig. 2e, evaluating the query requires only a single disk access.

Data Localization: How data are grouped together impacts the natural choice of indexing. For example, in the adjacency list representation, triples that are incident on the same vertex are clustered. In other words, if two triples can be joined on their subject-subject (SS) or object-object (OO) attributes, they must also be placed within the same list. Since SS and OO joins are the only two building blocks of star-shaped queries, an index built across the adjacency list will have a clear advantage for star-shaped queries. As a case study, consider *gStore*, which creates a signature entry for each vertex in the adjacency list based on the corresponding values in the list, and stores these signatures in a VS-tree for efficient lookup. Then, given a star-shaped query such as Q_2 in Fig. 1c, which has three bound predicate patterns (i.e., A, B and C), *gStore* can easily locate those vertices whose incident edges satisfy *all* three patterns—in this case, v_5 and v_6 (Fig. 2d). Contrast this to *RDF-3x* that indexes the large SPO table on six combinations of attributes (i.e., $s-p, s-o, p-s, p-o, o-s, o-p$), but these indexes can tell only which triples contain A or B or C . To find out that only the stars centered at v_5 and v_6 satisfy *all* three patterns in the query, the joins need to be computed (Fig. 3a). With the group-by-query representation, we are proposing a layout that can be customized for a given workload. Therefore, depending on the query patterns in that workload, not only SS and OO joins but also subject-object (SO) and object-subject (OS) joins will be precomputed and clustered, enabling better indexing opportunities for the other query structures.

Intermediate Result Elimination: In the relational representations, depending on how data are organized into ta-

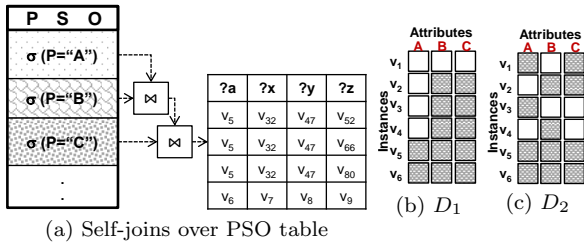


Figure 3: Self-joins are problematic in the single table layout, potentially generating unnecessary intermediate tuples.

bles, some queries need to be decomposed, resulting in the processing of unnecessary intermediate tuples. In case of the single table data organization (and its variants used in *RDF-3x*), every query needs to be decomposed into its triple patterns, where each triple pattern is evaluated against the large table and the results are self-joined. In this process, some intermediate tuples may not be needed for the computation of the final results. Although runtime optimizations exist [17, 22], they are not always effective. Consider query Q_2 (Fig. 1c) and the two datasets in Fig. 3b and Fig. 3c, where a shaded cell indicates that the value of an attribute (i.e., column) exists for the corresponding instance (i.e., row). For D_1 , join operations can be ordered such that the most selective triple pattern (i.e., A) is executed first. This technique, combined with sideways information passing [17], can significantly reduce the intermediate result set. In contrast, for D_2 , these optimizations do not work. First, join reordering does not make sense because each triple pattern is equally selective. Second, sideways information passing will not be effective unless the results from all selection operations are fully processed and the joins are computed. The advantage of the proposed representation is that data will be grouped such that the results of most queries in the workload are not segmented, thereby eliminating the need for query decomposition altogether.

2.2 Partial Tuning

For the group-by-query optimizations to be feasible, the system needs to dynamically update its physical design in a process that does not disrupt normal query execution. In practice, this means that the overhead of each incremental update on the physical design should be very small—given that most queries normally take sub-seconds to be executed, the updates should be much faster, perhaps on the order of microseconds. However, these types of updates are supported by existing self-tuning methods [7, 9] in a scalable way only if the physical schema changes are minor. This may not generally be true in RDF systems. Consider the single table layout in Fig 2a, and assume that data are initially sorted and indexed on attribute O, and that this physical design is suitable for only the first few queries. Hence, at runtime, a tuning advisor decides to switch to the columnar representation in Fig. 2b; however, to achieve this, the whole physical design needs to be updated from scratch, which is prohibitively expensive. In fact, the table will be partitioned on P and then each partition will be sorted and indexed on S. The former physical design efforts are wasted.

This is where *partial tuning* would be beneficial: one can partially cluster and index the database for only the relevant queries, thus preventing a waste of effort when the workload changes (Fig. 4). For example, one possible design optimization to reduce the I/O overhead in an RDF database can be

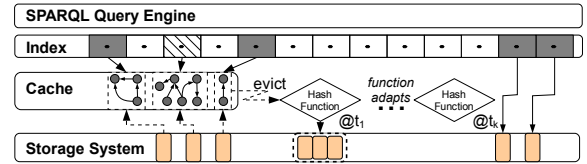


Figure 4: Partial indexing and re-structuring of the database

to co-cluster records on secondary storage based on query access patterns. If this were to be done partially, one could cluster only those records that correspond to the hotspots in the database, leaving the remaining ones unclustered. Then, hot records can be represented in main-memory using adjacency lists or even more sophisticated graph data structures, but for cold records, much less sophisticated data structures would suffice. Indexes can be constructed across the records in such a way that for frequent queries, they return a small number of false-positives (i.e., records that are not relevant to the query) while for the remaining queries they can be less efficient.

The only mature work in self-tuning databases that can be considered as a partial tuning technique is *database cracking* [13]. However, database cracking does not provide a complete solution for our vision because it works only with arrays in the context of in-memory column stores. In contrast, data structures and algorithms are needed that can support partial tuning for far more varied aspects of physical design, which are discussed next.

3. CHALLENGES

Physical data layout: As the workload changes, the way data are grouped into records may no longer be suitable for the new workload (due to data fragmentation, poor data localization and/or unnecessary intermediate results), therefore, they need to be updated. Second, access patterns over the storage system will change, indicating that records need to be re-clustered to reduce random I/O and cache stalls. Clustering algorithms used in conventional database design are not suitable for runtime execution—clustering is NP-hard and approximations have quadratic complexity [14]. Techniques are needed with similar objectives to these clustering algorithms, but with constant running time.

Consider one possible approach that combines hashing with caching (Fig 4). Let us assume that initially the database is structured such that each record contains exactly one triple. Likely, this is not a good representation for many workloads; however, as queries are executed, there is an opportunity for partially re-structuring the database based on the actual workload by predicting the future relevance of cached records. This is an open research problem (we discuss predictive models below). However, assuming that a good prediction algorithm can be designed, there are various opportunities: (i) multiple records may be packaged/merged into a new record, (ii) a record may be split into multiple records, (iii) records that are co-accessed across multiple queries may be placed contiguously on the storage system, or (iv) records that are no longer co-accessed may be distributed. As shown in Fig. 4, to quickly determine which records go together, a hash function is used (i.e., records that have the same hash value are assumed to have similar access patterns). The challenge is in designing an adaptive hashing scheme that can auto-tune as the query access patterns change.

Indexes: Typically, the total number of attributes is much larger in RDF than enterprise relational data. Furthermore, as argued in Section 2, depending on the workload, in a “good” RDF representation, records can be (i) variable sized, and (ii) arbitrarily structured. These properties make indexing a non-trivial problem. Irrespective of the actual implementation, the index should have the following properties. Consider the abstract index structure in Fig 4. Given a query Q , let $\mathbb{I}(Q)$ denote the set of records that are returned by the index (i.e., shaded and striped pointers) and let $\mathbb{R}(Q)$ denote the records that are *truly* relevant to Q (i.e., shaded pointers). As long as, for every possible query Q , the index satisfies $\mathbb{I}(Q) \supseteq \mathbb{R}(Q)$ (i.e., the index may return false-positive records but is guaranteed not to return any false-negatives) correct query results can be produced because false-positives can be eliminated in a validation step during query execution. There is clearly a trade-off between (i) index construction and maintenance overhead and (ii) validation overhead. The challenge is in designing an index such that for any query that lies within the window of interest it returns as few false-positives as possible whereas for old queries that are no longer frequent, it deliberately returns false-positives. The advantage of the partial indexing scheme is that it is easy to update due to its small size while being efficient for queries that are currently in use.

Dynamic tuning: An important issue is to decide (i) when and (ii) based on what information to tune the physical design of an RDF system. A reasonable approach can be to devise a predictive model from the last k queries and keep updating both the window size (k) and the predictive model as more queries are executed. While similar models exist for OLTP and OLAP workloads in SQL, the situation in SPARQL is different. Specifically, the following questions need to be answered: What is a good cost model for tuning, that is, what is the overhead of tuning versus the expected improvement in query execution time? This is closely related to query optimization in SPARQL, which is inherently different from SQL [19, 20]. When is it advantageous to tune the database? What are various coefficients in this cost model? Can the system adaptively learn these coefficients? How can “hotspots” be determined? That is, what is a good way to measure the popularity of an RDF resource—what about fluctuations in popularity [15]? Is there a way to model *volatility*, which refers to the variation in popularity over time? Given the linked nature of RDF [4], how can one decide on the scope of a query? For example, if a query is about the album THRILLER, should that also be counted towards the popularity of MICHAEL JACKSON, even though the latter is not explicitly stated in the query?

4. CONCLUSION

It is important to build robust RDF database systems to deal with the diversity and dynamism that are inherent in SPARQL workloads. While this requires existing RDF systems to be redesigned to incorporate self-tuning capabilities, the solution is not trivial. Specifically, due to the fuzzy notion of a schema in RDF, existing self-tuning methods in relational databases are not sufficient. Therefore, we propose a vision that contained key ingredients of a workload-driven data layout and a partial tuning scheme. While these ideas can be extended to distributed RDF systems [12, 20], in this paper we focus on single-node solutions. We have developed a prototype system that incorporates these basic premises

of the proposed system, and this gives us confidence for the feasibility of the vision. However, further research is needed to address the challenges in Section 3.

5. REFERENCES

- [1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. SW-Store: a vertically partitioned DBMS for semantic web data management. *VLDB J.*, 18:385–406, 2009.
- [2] G. Aluç, M. T. Özsu, K. Daudjee, and O. Hartig. chameleon-db: a workload-aware robust RDF data management system. Technical Report CS-2013-10, University of Waterloo, 2013.
- [3] M. Arias, J. D. Fernández, M. A. Martínez-Prieto, and P. de la Fuente. An empirical study of real-world SPARQL queries. In *Proc. 1st Int. Workshop on Usage Analysis and the Web of Data*, 2011.
- [4] C. Bizer, T. Heath, and T. Berners-Lee. Linked data - the story so far. *Int. J. Semantic Web Inf. Syst.*, 5(3):1–22, 2009.
- [5] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee. Building an efficient RDF store over a relational database. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 121–132, 2013.
- [6] M. Bröcheler, A. Pugliese, and V. S. Subrahmanian. DOGMA: A disk-oriented graph matching algorithm for rdf databases. In *Proc. 8th Int. Semantic Web Conference*, pages 97–113, 2009.
- [7] N. Bruno and S. Chaudhuri. To tune or not to tune? a lightweight physical design alerter. In *Proc. 32nd Int. Conf. on Very Large Data Bases*, pages 499–510, 2006.
- [8] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: implementing the semantic web recommendations. In *Proc. 13th Int. World Wide Web Conf. - Alternate Track Papers & Posters*, pages 74–83, 2004.
- [9] S. Chaudhuri and V. Narasayya. Self-tuning database systems: a decade of progress. In *Proc. 33rd Int. Conf. on Very Large Data Bases*, pages 3–14, 2007.
- [10] S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea. Apples and oranges: a comparison of RDF benchmarks and real RDF datasets. In *SIGMOD Conference*, pages 145–156, 2011.
- [11] O. Erling. Virtuoso, a hybrid RDBMS/graph column store. *IEEE Data Eng. Bull.*, 35(1):3–8, 2012.
- [12] J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL querying of large RDF graphs. *Proc. VLDB*, 4(11):1123–1134, 2011.
- [13] S. Idreos, S. Manegold, H. A. Kuno, and G. Graefe. Merging what’s cracked, cracking what’s merged: Adaptive indexing in main-memory column-stores. *PVLDB*, 4(9):585–597, 2011.
- [14] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Comput. Surv.*, 31:264–323, 1999.
- [15] M. Kirchberg, R. K. L. Ko, and B. S. Lee. From linked data to relevant data – time is the essence. In *Proc. 1st Int. Workshop on Usage Analysis and the Web of Data*, 2011.
- [16] G. Klyne and J. J. Carroll. Resource Description Framework (RDF): Concepts and abstract syntax. <http://www.w3.org/TR/rdf-concepts/>, 2004.
- [17] T. Neumann and G. Weikum. Scalable join processing on very large RDF graphs. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 627–640, 2009.
- [18] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB J.*, 19(1):91–113, 2010.
- [19] M. Schmidt, M. Meier, and G. Lausen. Foundations of SPARQL query optimization. In *Proc. 13th Int. Conf. on Database Theory*, pages 4–33, 2010.
- [20] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. FedX: Optimization techniques for federated query processing on linked data. In *Proc. 10th Int. Semantic Web Conference*, pages 601–616, 2011.
- [21] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold. Column-store support for RDF data management: not all swans are white. *Proc. VLDB*, 1(2):1553–1563, 2008.
- [22] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. Sparql basic graph pattern optimization using selectivity estimation. In *Proc. 17th Int. World Wide Web Conf.*, pages 595–604, 2008.
- [23] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proc. VLDB*, 1(1):1008–1019, 2008.
- [24] K. Wilkinson. Jena property table implementation. Technical Report HPL-2006-140, HP-Labs, 2006.
- [25] L. Zou, J. Mo, D. Zhao, L. Chen, and M. T. Özsu. gStore: Answering SPARQL queries via subgraph matching. *Proc. VLDB*, 4(1):482–493, 2011.