SMJoin: A Multi-way Join Operator for SPARQL Queries

Mikhail Galkin
University of Bonn & Fraunhofer IAIS
& ITMO University
Bonn, Germany
galkin@cs.uni-bonn.de

Kemele M. Endris
University of Bonn
Bonn, Germany
endris@cs.uni-bonn.de

Maribel Acosta
Karlsruhe Institute of Technology
Karlsruhe, Germany
acosta@kit.edu

Diego Collarana
University of Bonn & Fraunhofer IAIS
Sankt Augustin, Germany
collaran@cs.uni-bonn.de

Maria-Esther Vidal
Fraunhofer IAIS
Bonn, Germany
vidal@cs.uni-bonn.de

Sören Auer
Leibniz Universität Hannover / TIB
Information Center, Germany
soeren.auer@tib.eu

ABSTRACT

Join operators are particularly important in SPARQL query engines that collect RDF data using Web access interfaces. State-of-the-art SPARQL query engines rely on binary join operators tailored for merging results from SPARQL queries over Web access interfaces. However, in queries with a large number of triple patterns, binary joins constitute a significant burden on the query performance. Multi-way joins that handle more than two inputs are able to reduce the complexity of pre-processing stages and reduce the execution time. Whereas in the relational databases field multi-way joins have already received some attention, the applicability of multi-way joins in SPARQL query processing remains unexplored. We devise SMJoin, a multi-way non-blocking join operator tailored for independently merging results from more than two RDF data sources. SMJoin implements intra-operator adaptivity, i.e., it is able to adjust join execution schedulers to the conditions of Web access interfaces; thus, query answers are produced as soon as they are computed and can be continuously generated even if one of the sources becomes blocked. We empirically study the behavior of SMJoin in two benchmarks with queries of different selectivity; state-of-the-art SPARQL query engines are included in the study. Experimental results suggest that SMJoin outperforms existing approaches in very selective queries, and produces first answers as fast as compared adaptive query engines in non-selective queries.

CCS CONCEPTS

•Information systems →Join algorithms;

KEYWORDS

SPARQL; Join algorithms; Multi-way Join Operators

ACM Reference format:
Mikhail Galkin, Kemele M. Endris, Maribel Acosta, Diego Collarana, Maria-Esther Vidal, and Sören Auer. 2017. SMJoin: A Multi-way Join Operator for SPARQL Queries. In *Proceedings of Semantics2017, Amsterdam, Netherlands, September 11–14, 2017,* 8 pages.
DOI: 10.1145/3132218.3132220

Semantics2017, Amsterdam, Netherlands

DOI: 10.1145/3132218.3132220

1 INTRODUCTION

The increasing adoption of Linked Data has encouraged the publication of RDF datasets on the Web, as well as the development of Web query processing engines [1, 2, 8, 11, 12]. Web query processing requires the implementation of data management methods for selecting relevant datasets, and for collecting and merging data. Existing query engines rely on physical operators, e.g., binary joins, to access and combine data required for query answering [1, 2, 9]. However, Web data sources are autonomous, and data transfer rates can considerably vary among them. Thus, operators able to complete query processing schedulers and produce results as soon as possible, are mandatory for realizing efficient Web query processing. State-of-the-art query engines like nLDE [1] and ANAPSID [2], rely on adaptive query strategies and symmetric binary joins, to adjust execution schedulers to fluctuating data transfer rates.

Albeit effective, symmetric binary joins can drastically impact the performance of query processing whenever source answers need to be passed through multiple operators in a query plan. Multi-way symmetric joins have been proposed to overcome this problem, and propagate and generate (intermediate) results in a single step [4] during query execution. Nevertheless, multi-way joins tailored to exploit properties of SPARQL queries, e.g., query shape, have not yet been explored in existing unified and federated query processing engines.

We present *SMJoin*, a multi-way join operator customized for star-shaped SPARQL queries, i.e., triple patterns that share one variable. SMJoin employs specific data structures to process intermediate results; it is controlled by a router that maintains a *dynamic probing sequence* able to reduce the amount of steps required to generate a query answer. Having an arbitrary arity, SMJoin is able to reduce the height of a query tree plan, and effectively join multiple triple patterns of a star-shaped subquery in one step. In this article, performance of SMJoin is empirically evaluated on high- and low-selective queries. Observed results suggest that SMJoin speeds up query processing in high-selective star-shaped queries.

The remainder of the article is structured as follows: Section 2 motivates the importance of multi-way joins; Section 3 defines the SMJoin approach; Section 4 reports on the conducted experiments

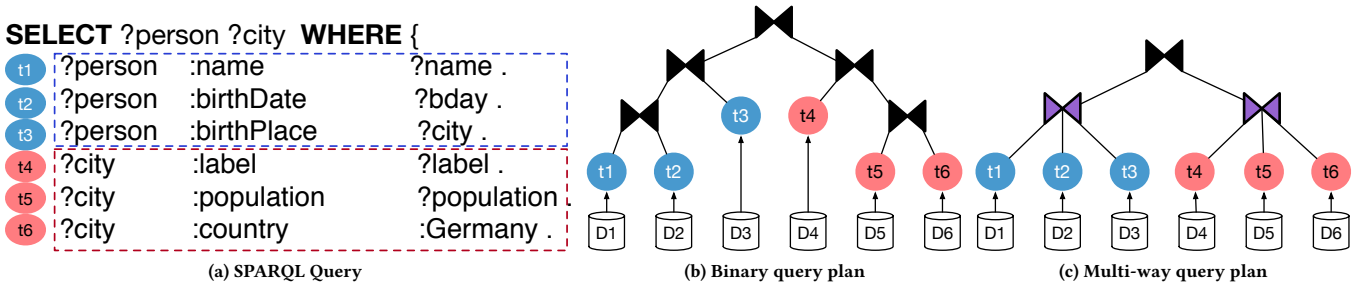


Figure 1: Motivating Example. (a) SPARQL query composed of six triple patterns grouped in two star-shaped blocks; every triple pattern resides in its own source. (b) An example query plan that employs only binary join operators. (c) A query plan with multi-way joins where the plan height and number of operators are reduced.

comparing SMJoin with binary join operators; Section 5 provides an overview of current state of the art in multi-way joins and semantic data management; and finally Section 6 concludes the article with the lessons learned and outlines a scope of the future work.

2 MOTIVATING EXAMPLE

SPARQL queries comprised of more than one triple pattern often share at least one variable. In certain queries one variable is shared among numerous triple patterns. Fig. 1a illustrates a query that consists of six triple patterns ($t_1, \ldots, t_6$) that are grouped into two star-shaped subqueries, i.e., the triple patterns share the same subject variable. Moreover, each triple pattern is answered only by one source ($D_1, \ldots, D_6$), respectively. Given the query and the description of sources, existing SPARQL query engines apply various query optimization strategies to construct the most efficient query plans. However, such engines employ only binary join operators in their plans. Consequently, the constructed plans, for instance, the one depicted on Fig. 1b, are variations of a binary or bushy tree. That is, for a subquery around the variable `?person`, t_1 has to be joined with t_2, an intermediate result is joined with t_3. Similarly, for a `?city` subquery, t_5 has to be joined with t_6, an intermediate result with t_4. Finally, intermediate results from two subgroups are joined to obtain final answers. Overall, five joins have to be performed. Complex plans of numerous joins negatively affect query performance. Additionally, more joins produce more intermediate results that in turn increase network traffic. Despite the variety of binary join operators [4, 9, 13], there is still room for improvement.

Supporting an arbitrary amount of inputs, multi-way join operators facilitate creation of simplified query plans, produce less intermediate results, reduce network costs, and therefore, improve query performance. Fig. 1c illustrates a query plan composed of two three-way join operators, where each multi-way operator performs a join of one star-shaped group (`?person` and `?city`), respectively.

In the relational database community multi-way join operators represent an established research field [7, 16, 17]. However, to the best of our knowledge, in semantic data management and SPARQL query processing using multi-way joins remain largely unexplored. In this article, we aim to bridge this gap and propose SMJoin (SPARQL Multi-way Join), a multi-way, non-blocking join operator tailored for star-shaped groups expressed in SPARQL.

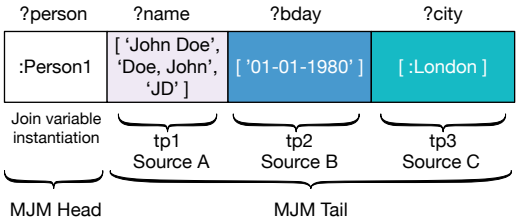


Figure 2: Example of an Multi Join Mapping (MJM). An MJM Head contains an instantiation *:Person1* of a join variable *?person*. An MJM Tail consists of instantiations of the triple patterns tp_1, tp_2, tp_3 over sources A, B, and C, respectively.

3 THE SMJOIN OPERATOR

3.1 Preliminaries

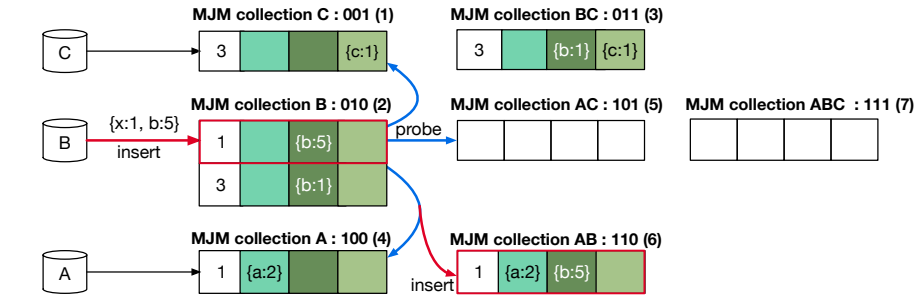
Star-Shaped Subqueries: In SPARQL queries, Basic Graph Patterns (BGPs) correspond to subqueries composed of triple patterns connected by the `AND` logical operator which represents a binary join operator. A star-shaped subquery is a BGP where all the triple patterns share exactly one join variable which is placed in the subject position in all the triple patterns of the BGP. A star-shaped subquery of triple patterns in a BGP S on a join variable $?X$, denoted as star(S, $?X$), is defined as follows [15]:

- star(S, $?X$) is a triple pattern tp in S, where tp is of the form $\{?X\ p\ o\}$, and $p \neq ?X$ and $o \neq ?X$.
- star(S, $?X$) is the union of two stars on the variable $?X$, star($S1$, $?X$) and star($S2$, $?X$), where triple patterns in $S1$ and $S2$ only share the variable $?X$, i.e., var(star($S1$, $?X$)) $\cap$ var(star($S2$, $?X$)) = $\{?X\}$.

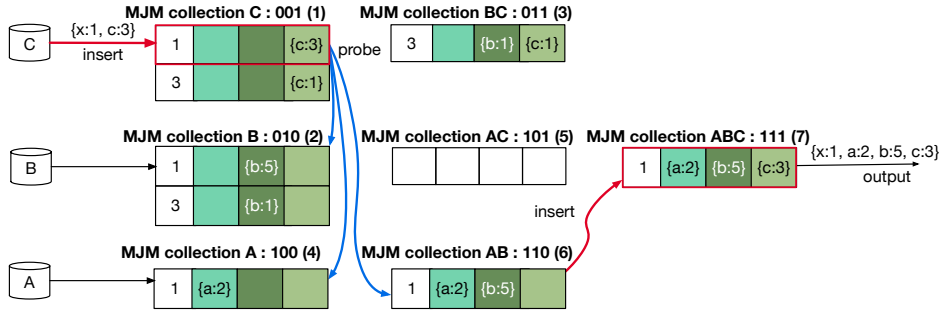
3.2 SMJoin

SMJoin resembles the *MJoin* [16] and *agjoin* [2] operators, but novel enhancements are developed to evaluate star-shaped subqueries.

Multi Join Mappings (MJMs): SMJoin utilizes Multi Join Mappings (MJMs) as inner data structures to store join intermediate results. Given a star-shaped subquery star(S, $?X$) of triple patterns $\{tp_1, \ldots, tp_n\}$ and an instantiation $\mu(?X)$ of the join variable $?X$,



(a) Generation of Intermediate Results



(b) Generation of Query Answers

Figure 3: SMJoin Intuition. (a) Current state: join variable is *x*, one tuple arrived from each of the sources *A*, *B*, *C*, and one intermediate result resides in the MJM collection *BC*. Then, a tuple arrives from the source *B*. The newly created MJM is probed against MJM collections *AC* (empty), *C*, and *A*. The instantiation of the join variable *x* in *A* is the same, a new intermediate join result is inserted into the MJM collection *AB*. Finally, the arrived MJM is inserted into the collection *B*. (b) A tuple arrives from the source *C*. Its MJM is probed first against the MJM collection *AB*. A join is identified so that collections *B* and *A* are omitted. The obtained result is inserted into the MJM collection *ABC* and considered as final. One result tuple is produced.

an MJM is defined as a pair as follows:

$$MJM = (\mu(?X), \{T_1, \ldots, T_n\}) \quad (1)$$

where T_i corresponds to a set of instantiations of the triple pattern tp_i in S such that for all the instantiations in T_i, the instantiation of the join variable $?X$ is $\mu(?X)$. In an MJM, the first argument of the pair, i.e., $\mu(?X)$, is called the MJM *Head*, while the second argument, i.e., $\{T_1, \ldots, T_n\}$, is named the MJM *Tail*.

Fig. 2 illustrates an MJM of the star-shaped subquery on the variable `?person` in Fig. 1a. The MJM *Head* is :Person1 which corresponds to an instantiation of the join variable *?person*. Moreover, the second argument of the MJM is $\{T_1, T_2, T_3\}$, with the instantiations of *?name*, *?bday*, *?city* of the triple patterns tp_1, tp_2, tp_3.

MJMs are indexed by MJM Head values, i.e., an instantiation of a join variable. In Fig. 2, *?person* is a join variable and the MJM Head value is `:Person1`. tp_1 after evaluation returns three tuples with *?name* of `:Person1`, while tp_2 and tp_3 produce instantiations of *?bday* and *?city* for `:Person1`, respectively. Thus, the MJM *Tail* consists of three sets with the tuples collected after executing tp_1, tp_2, and tp_3 against sources *A*, *B*, and *C*, respectively.

Fig. 2 presents a complete MJM for the star-shaped subquery of `?person`, i.e., an MJM with instantiations of the triple patterns tp_1, tp_2, tp_3 for the MJM *Head* :Person1, are part of the MJM *Tail*. Once an MJM is completed, the SMJoin operator produces three results as a Cartesian product of sets in the MJM Tail, e.g., {`:Person1,'John Doe','01-01-1980',:London`};{`:Person1,'Doe,John','01-01-1980', :London`}; and {`:Person1, 'JD','01-01-1980',:London`}. Referring to the plan in Fig. 1c those tuples are pushed forward as an input for a subsequent binary join operator in the query plan.

In order to join multiple answers from different sources, SMJoin aims at completing MJMs for the instantiations of a given join variable. Fig. 3 depicts the SMJoin pipeline, i.e., from the arrival of a first tuple until the generation of the final query answer.

MJM Collections: An MJM collection is a set of MJMs where the *Tails* are composed of sets of instantiations of the same group of triple patterns. For example, Fig. 3a depicts seven MJM collections. In particular, the MJM collection *B* stores two MJMs with various instantiations (i.e., {`x:1, b:5`}, {`x:3, b:1`}) of one triple pattern (e.g., `?x :property ?b`) that was answered by the source *B* .

Upon initialization of the SMJoin operator with n inputs, $2^n - 1$ MJM collections are created. Instantiations received from n sources are stored in n MJM collections. Combinations of these instantiations correspond to intermediate results and are kept in $((2^n-1)-n)$ MJM collections; only one MJM collection stores the complete results to be yielded. For instance, on Fig. 3a, SMJoin is initialized

with answers from sources *A*, *B*, and *C*; seven MJM collections are created where three MJM collections store incoming tuples from a respective source; three MJM collections accumulate intermediate results; and one MJM collection is reserved for the join results. We provide additional details how all these MJM collections are organized and ordered during a *probing sequence* in subsection 3.3.

Computing Join Results from MJM Collections: When an input triple - an instantiation of a triple pattern - arrives from a source, several actions are triggered. Firstly, to foster results generation and reduce response time, the input tuple is probed against relevant MJMs according to a *probing sequence*, i.e., MJMs are arranged in a way to yield join results faster. In case a join match occurs when probing, an intermediate result is stored in a relevant intermediate MJM collection. If an MJM Tail for a current $\mu(?X)$ is completed, i.e., the MJM contains input tuples from all sources, then this MJM exists in the output MJM collection and is ready to yield join results. Secondly, the tuple invokes its own MJM with its $\mu(?X)$ as MJM Head and tuple value in the corresponding section in MJM Tail. Finally, the new MJM is inserted into a corresponding MJM collection, i.e., MJM collection *A* puts together MJMs which join variable instantiations $\mu(?X)$ contain tuples from source *A*.

In Fig. 3a, a tuple `{x:1, b:5}` arrives from the source *B*. Given `'x'` as a join variable, at the current state, there exist intermediate MJMs in MJM collections: *A* (instantiated with a value `{x:1}` in its head), *B* with `{x:3}`, *C* (`{x:3}`), and *BC* (instantiated with a value `{x:3}` as a join between *B* and *C*). The arrived tuple from the source *B* generates its MJM and is subsequently probed against three MJM collections, namely *AC*, *A*, and *C*. As *AC* is empty, the arrived MJM `{1:[{b:5}]}` is probed against existing MJMs in *A* `{1:[{a:2}]}` and *C* `{3:[{c:1}]}`. $\mu(x) = 1$ occurs for a probing tuple in *A*; therefore, a join match is identified, a new MJM is created where the head remains the same, but the tail is a union of the tails of probed MJMs. The new intermediate MJM, i.e., `{1:[{a:2},{b:5}]}`, is inserted into the relevant MJM collection *AB* that accumulates joinable tuples from the sources *A* and *B*.

Furthermore, a tuple `{x:1, c:3}` arrives from the source *C* (cf. Fig. 3b). A new MJM `{1:[{c:3}]}` is created and probed against three MJM collections in the order *AB*, *B*, and *A*. The head of the intermediate result in *AB* obtained at the previous step, matches with the head of the MJM from the source *C*, i.e., $\mu(x) = 1$, and the join is identified. In turn, a new intermediate MJM `{1:[{a:2},{b:5},{c:3}]}` is created and inserted into the *ABC* MJM collection. As this MJM consists of the input tuples obtained from all three sources *A*, *B*, and *C*, the MJM is annotated as complete (similar to the one represented on Fig. 2). Thus, SMJoin is able to yield resulting joined tuples. Taking a Cartesian product of the MJM head and unique variables in the MJM tail, one tuple is pushed: `{x:1,a:2,b:5,c:3}`; it corresponds to the outcome of a three-way SMJoin operator. Note that after probing against an MJM in *AB* and discovering a join, there is no probing against MJM collections *A* and *B* separately; thus, all their possible join permutations are already in the *AB* collection. However, if probing of an MJM from *C* against an MJM in *AB* does not produce a new intermediate result, then the collections *A* and *B* are still present in the probing sequence. SMJoin is equipped with an intra-operator router for probing sequence; further, the router is able to avoid redundant probings.

3.3 SMJoin Intra-Operator Router

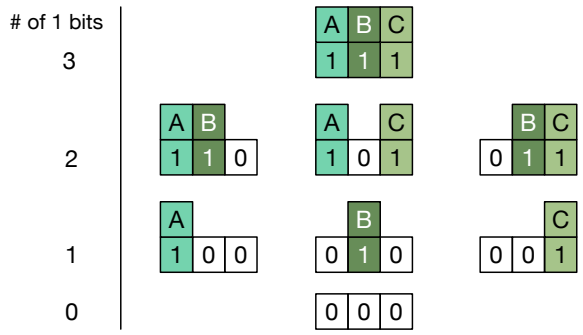
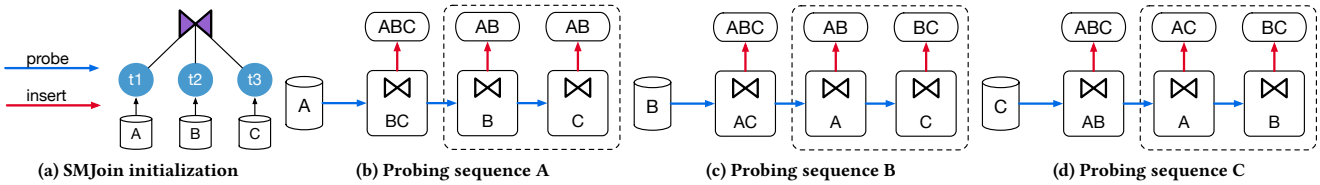


Figure 4: A lattice is comprised of three sources *A*, *B*, *C* that correspond to digits in the lattice. Total number of indices is $2^3 = 8$. The index 000 is redundant as it is not connected with any source or intermediate result. The components with indices 100, 010, 001 store incoming tuples from sources *A*, *B*, *C*, respectively. The components 110, 101, 011 store intermediate join results from *AB*, *AC*, *BC*, respectively. Index 111 denotes a component *ABC* with final SMJoin results.

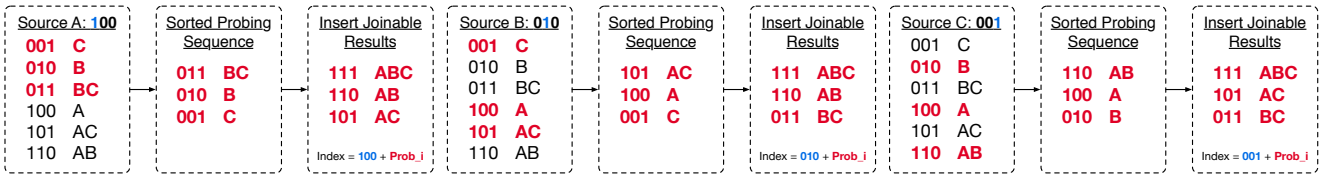
A Lattice of MJM Collections: Before describing the SMJoin router and its probing sequence generator, it is important to understand how the inner data structures, i.e., MJM collections, are organized. All the $2^n - 1$ MJM collections are ordered into a *lattice* structure (Fig. 4) where a binary index of an MJM collection corresponds to the contents of this MJM collection. Bit 1 at a certain position in the index number denotes sources which intermediate join results are stored in the given MJM collection. An MJM collection with index 0 becomes thus redundant as it does not contain any MJM from any source. Therefore, the overall amount of MJM collections equals to $2^n - 1$. Fig. 4 illustrates a lattice comprised of three sources *A*, *B*, *C*; lattice elements (MJM collection indices) consist of three digits equal to the number of sources. Indices that contain a single 1 bit (001, 010, 100) store MJMs received from the sources, i.e., MJM collection 001 (decimal index 1) accumulates MJMs from the source *C*, 010 (2) from *B*, and 100 (4) from *A*. Indices that include two 1 bits (011, 101, 110) store intermediate join results obtained from the respective sources. For instance, MJM collection with index 011 (3) consists of the intermediate join results between *B* (010) and *C* (001). Similarly, MJM collection 101 is a place for intermediate results between *A* (100) and *C* (001). Finally, index 110 shows that this MJM combines joins from *A* (100) and *B* (010). One of the natural benefits provided by a SMJoin lattice is that the index of an MJM collection to insert an intermediate result corresponds to a logical OR between two original indices, e.g., for *AB* 110 = 100 OR 010. The MJM collection with index 111 denotes eventual join operator results as three 1 bits represent that MJMs in this collection include results from all three sources *A*, *B*, and *C*. MJMs in this collection *ABC* are converted to the format of output tuples and pushed forward to the next level of a query plan.

SMJoin Probing Sequences: The SMJoin router takes advantage of the SMJoin binary indexation strategy to produce the best possible probing sequence for every attached source. An efficient



(a) SMJoin initialization (b) Probing sequence A (c) Probing sequence B (d) Probing sequence C

Figure 5: The SMJoin Router. (a) Initial state of SMJoin. (b) The probing sequence for the source *A* includes MJM collections *BC*, *B*, and *C* (blue arrows). Collections surrounded by a dashed line, *B* and *C*, are excluded from the probing sequence if the join occurs at the first stage when probing against *BC*. Successful probings are inserted into corresponding MJM collections *ABC* (output), *AB*, and *AC* (red arrows). (c) The probing sequence for the source *B* is comprised of *AC*, *A*, and *C* where two latter are excluded if there is a join between *B* and *AC*. (d) The probing sequence for the source *C* consists of *AB*, *A*, and *B*. If a join occurs between *C* and *AB* then *A* and *B* are excluded from the probing sequence.



(a) Probing sequence for the source A (b) Probing sequence for the source B (c) Probing sequence for the source C

Figure 6: Computing a probing sequence. (a) The incoming tuples from the source *A* are stored in the lattice component with index 100. At the first step three indices are selected as they do not contain any data from *A*, i.e., 001 (*C*), 010 (*B*), 011 (*BC*). At the second step the selected indices are sorted (desc.) by the amount of 1 bits in the binary representation, i.e., the sorted sequence is 011, 010, 001. If join is identified an index of the collection to insert an intermediate result into is computed as logical OR between source *A* index and probing index from the sequence, e.g., 100 OR 011 = 111. The same algorithm with selecting and sorting steps applies when computing a probing sequence for the sources *B* (b) and *C* (c).

probing sequence eliminates redundant comparisons between intermediate results and facilitates faster output generation. Fig. 5 illustrates probing sequences arranged by the SMJoin router for a given subquery of tp_1, tp_2, tp_3 joined by the SMJoin operator (cf. Fig. 5a). The Router is designed to follow two principles: *i*) Foster the join results output rate; *ii*) Eliminate redundant comparisons.

To tackle the first goal, the SMJoin router arranges a probing sequence in a specific order where the intermediate MJM collections are placed first in the queue iff they contain join results from all the sources except the incoming one. That is, given n sources, the first collection to probe is the one which MJMs already accumulate $n-1$ sources, e.g., for the source *A* the first collection to test is *BC* (cf. Fig. 6a). If a join is identified, then the MJM is directly passed to the output as the MJM contains tuples from all n sources and the insert index equals to 111 (*ABC*). Additionally, MJM collections *B* and *C* are removed from the probing sequence as redundant, i.e., there is no need to compare an MJM from *A* with *B* and *C* separately when it has already been probed against their joinable results in *BC*. If no join is identified at the current stage, the next MJM collection in the queue combines intermediate results from $n-2$ sources, e.g., as in Fig. 6a, for the source *A* those are collections *B* and *C*. If a join occurs, then a new intermediate result is inserted into a corresponding collection *AB* or *AC* whereas its index is computed as a logical OR between *A* and *B* or *A* and *C*, respectively; thus lattice properties are exploited. For n sources, the probing sequence is sorted in descending order by the number of sources that contributed to a given MJM collection with intermediate results, i.e., from $n-1$ to 1. The SMJoin router organizes probing sequences *B* (cf. Fig. 6b) and *C* (cf. Fig. 6c) similarly to the probing sequence *A* in Fig. 6a.

The second goal is required in high-dimensional SMJoin instantiations. Given n sources, SMJoin handles 2^n-1 MJM collections (2^n-2 excluding the output collection). Therefore, a straightforward approach to compare an incoming MJM with the rest of MJM collections has to probe this MJM against 2^n-3 collections that leads to exponential time complexity. In order to reduce time complexity, the SMJoin router introduces a *dynamic probing sequence* technique that prunes a probing sequence up to one MJM collection in the best case and to $2^{n-1}-1$ in the worst case. Fig. 6 illustrates how the SMJoin router identifies a probing sequence for each source. The generation of the sequence consists of two steps: *i*) Selection of the relevant MJM collections; and *ii*) Sorting the sequence.

SMJoin Router Steps: At the first step, the SMJoin router selects relevant MJM collections to probe against an MJM from a given source. In terms of binary indices of a lattice, the SMJoin router selects only those indices that have a 0 bit in the position of an incoming source, i.e., a 0 bit denotes that the given source has not yet been probed against the contents of an MJM collection. Exploiting the lattice structure, given n sources, each binary index has n binary digits and there exists $2^{n-1}-1$ indices that have a 0 bit at a certain position (excluding an empty all-zero index collection,

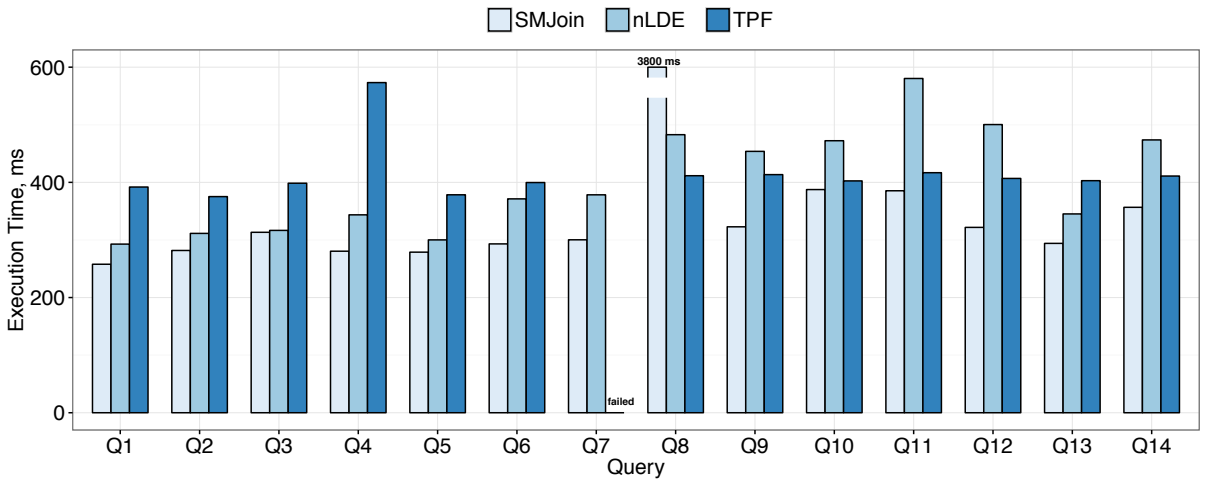


Figure 7: Benchmark 1. High selective queries. SMJoin demonstrates the best performance in all queries except Q8. Q8 takes almost four seconds as all its triple patterns are less selective compared to other queries. Q7 is not answered by TPF.

e.g., 000 or 0000). For instance, Fig. 6a shows the probing sequence generation for the source *A* that is connected to its MJM collection with index 100 (4). Among all MJM collections, the SMJoin router selects only those that have a 0 bit at the third position, e.g., 001 (1) that is bound to the source *C*, 010 (2) that belongs to *B*, and 011 (3), *BC*, that contains join results between *B* and *C*. The collections 100 (4), 101 (5), 110 (6) are discarded as all of them contain tuples from the source A, and they are not applicable for probing against *A*.

At the second step, the SMJoin router sorts the selected indices in descending order considering the amount of 1 bits in the binary representation of an index. Each 1 bit in the binary notation refers to a source whose tuples are presented in MJMs of the collection with this index. Therefore, the SMJoin router tackles the first goal of amplifying the output rate by placing MJM collections with intermediate results of higher join *readiness* at first places in the sequence. For the source *A* (cf. Fig. 6a), the probing sequence after applying a sorting algorithm is arranged as 011 (3), 010 (2), 001 (1). That is, if a join occurs when probing *A* with *BC*, the obtained result is immediately pushed as an output of the SMJoin whereas collections 010 and 001 are excluded by the SMJoin router from the probing sequence. The *dynamic probing sequence* technique relies on such exclusions in order to prune redundant probings. Generally, the SMJoin router examines a binary index of a successfully probed MJM collection; it also extracts from that index all permutations of binary numbers that contain 1 bits at the same position as in the given index. The extracted numbers are excluded from the probing sequence. For example, given the index 011 (3) two number are extracted: 010 (2) as it shares bit 1 in the second position, and 001 (1) as it shares 1 bit in the first position. In complex cases, e.g., queries over five sources, if a join occurred when probing an MJM from the collection 00001 against an MJM collection with index 10110 (22), then six numbers are extracted: 10100 (20), 10010 (18), 10000 (16), 00110 (6), 00100 (4), 00010 (2). Subsequently, those numbers are excluded from the probing sequence for the collection 00001.

Finally, the SMJoin router identifies an index of a relevant MJM collection to insert a new intermediate result, by evaluating a logical OR between the indices of the probing collections. As illustrated in Fig. 6a, an identified join between 100 (source *A*) and 010 (source *B*) is inserted into a collection with index 110 (*AB*) as 100 OR 010 = 110. The SMJoin router performs similarly to the described pipeline during selection, sorting, and pruning when computing a probing sequence for the source *B* (cf. Fig. 6b) and *C* (cf. Fig. 6c).

4 EMPIRICAL EVALUATION

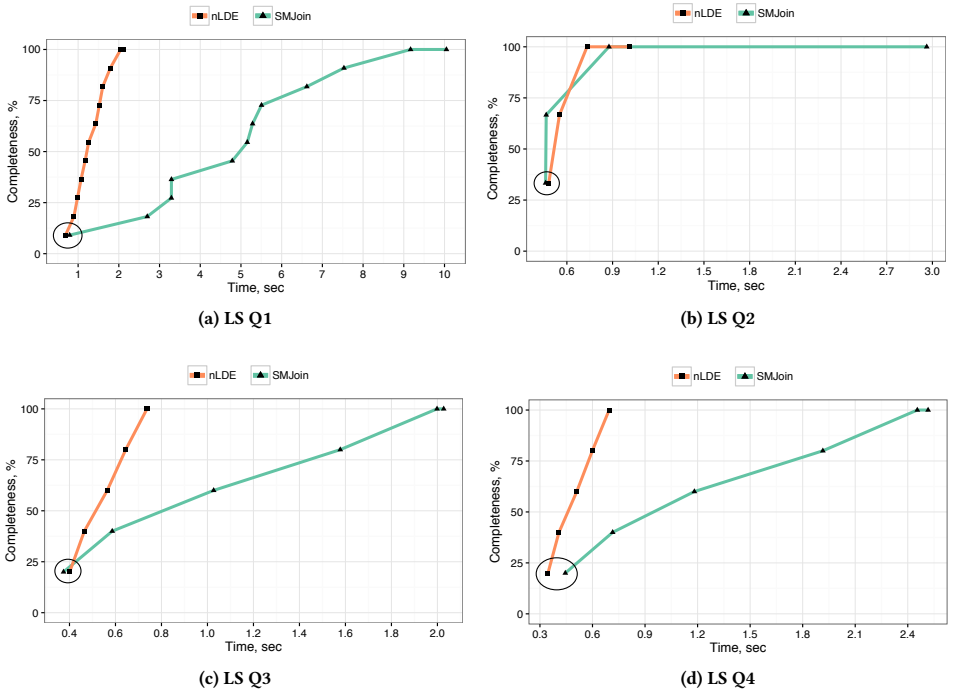
We empirically assess the performance of the SMJoin operator and benefits of a multi-way join setup compared to state-of-the-art SPARQL query processing engines, namely nLDE (Network of Linked Data Eddies) [1] and Triple Pattern Fragments Client [14]. The experimental configuration is as follows:

Benchmark and Queries: We applied two benchmarks of queries. Benchmark 1 contains 14 queries of high selectivity. Each query is composed of up to five triple patterns that share the same join variable, i.e., one star-shaped group is evaluated by a five-way SMJoin. Each triple pattern returns less than 100 intermediate results. Benchmark 2 consists of four queries that exhibit lesser selectivity, i.e., the number of intermediate results is two orders of magnitude larger than in Benchmark 1. Each query includes up to four triple patterns, i.e., a four-way SMJoin is invoked. In order to evaluate only the SMJoin operator performance not affecting query planning, all queries include one star-shaped subquery. Overall, 18 queries[1] are executed five times, an average value is reported.

Dataset: English version of DBpedia 2015 is converted into HDT format and hosted on top of the TPF Server. The total number of triples in the dataset is 837,257,959.

Metrics: *i*) *Execution Time*: Elapsed time between the submission of the query to an engine and the arrival of all the answers. Time corresponds to absolute wall-clock system time as reported by Python `time.time()`. Timeout limit is set to 600 seconds. *ii*) *Completeness*:

[1] https://github.com/migalkin/SMJoin-experiments/tree/master/queries/new_queries



(a) LS Q1 (b) LS Q2

(c) LS Q3 (d) LS Q4

Figure 8: Benchmark 2. Low-selective queries. Completeness over time. SMJoin yields the first answer at about the same time as nLDE. However, due to lesser selectivity SMJoin has to process more intermediate results, and therefore, it finishes later. SMJoin takes more time to produce complete results in Q1(a), Q3(c), Q4(d) sending triple patterns independently whereas nLDE employs Nested Loop Join. In Q2(b) SMJoin produces the complete result at the same rate as nLDE but has to finish processing of all intermediate results that takes two additional seconds.

Percentage of answers produced by a query engine when executing a query compared to the number of final query results.

Implementation: SMJoin operator is implemented in Python 2.7.10. We also created a decomposer of SPARQL Basic Graph Patterns (BGPs) into star-shaped subqueries. The query decomposer invokes SMJoin for each star-shaped subquery, and each triple pattern of the subquery is sent independently to a Triple Pattern Fragment (TPF) Server. For example, for a query composed of five triple patterns, five independent queries are posed. The experiments were executed on a Ubuntu 16.04 (64 bits) Dell PowerEdge R805 server, AMD Opteron 2.4 GHz CPU, 64 cores, 256 GB RAM.

Experiment 1: High Selective Queries Experiment 1 evaluates the SMJoin performance on the set of high selective queries from the Benchmark 1. Fig. 7 presents the observed results against 14 queries compared to nLDE and TPF engines. Each query is executed five times and the arithmetic mean is reported. The observed results show that SMJoin outperforms on average these two state-of-the-art engines in 13 queries. This suggests that one-stage multi-way (i.e., three- to five-way) joins outperform binary join plans for very selective queries composed of very selective triple patterns, i.e., triple patterns with one variable and around 100 instantiations. nLDE and TPF generally take in average more time to execute very selective queries compared to SMJoin, but vary in performance pairwise. For example, nLDE is faster than TPF in Q1-Q7, Q13 (Q7 is not answered by TPF and its time marked as zero), whereas TPF outperforms nLDE in Q8-Q12, Q14. SMJoin is less effective in Q8 producing complete results in about four seconds; this behavior is explained by the low selectivity of the query triple patterns. In SMJoin, triple patterns are posed independently; thus, the more results the TPF server returns, the higher the time SMJoin waits until all tuples arrive and joins are performed.

Experiment 2: Low-Selective Queries We assess the continuous SMJoin performance on low-selective queries from the Benchmark 2. Fig. 8 depicts the generation of query answers over time. SMJoin and nLDE are compared because as adaptive query processing techniques, they are able to produce results incrementally. Despite the higher amount of intermediate results, SMJoin is able to yield the first result at about the same time as nLDE. It is important to note that nLDE exploits information about the cardinality of the triple pattern fragments, and evaluates first the most selective patterns; nLDE also opportunistically places Nested Loop Joins to produce instantiations for low selective triple pattern fragments. In consequence, nLDE is able to drastically reduce the amount of intermediate results, and speed up execution time. In contrast, SMJoin works under the assumption of zero knowledge about the cardinality of the triple pattern fragments, and sends every triple pattern to the TPF server independently. These two different approaches of scheduling query execution allow for explaining why nLDE is faster than SMJoin over time. This is particularly important in queries like Q1 (cf. Fig. 8a), Q3 (cf. Fig. 8c), and Q4 (cf. Fig. 8d).

Fig. 8b shows the impact of a large number of intermediate results on the SMJoin performance. Thus, the complete query results are produced during the first 0.9 seconds. This is comparable to nLDE; however, SMJoin has to finish processing of all tuples received from the TPF server, thus taking two more seconds are produced.

Discussion: Observed results from Experiments 1 and 2 suggest that selecting SMJoin for high selective triple patterns and binary join operators for low-selective patterns allow for efficient execution of SPARQL query. To generate these plans, existing query decomposition and optimization techniques required to be extended to consider both join operators of different arity, i.e., binary and multi-way joins, as well as cardinality of triple pattern fragments.

5 RELATED WORK

Binary join operators are used by both Database and Semantic Web query processing engines. The operators are often classified along several dimensions, e.g., blocking and non-blocking, adaptive and non-adaptive. Federated query engines often employ a collection of join operators at hand to choose the best relevant and most efficient operator for a current condition.

Nested Loop Join [6], Hash Join [4], and Symmetric Hash Join (SHJ) [4] are traditional binary join operators that rely on hash tables as inner data structures to reduce the amount of necessary comparisons between tuples. Additionally, SHJ implements a non-blocking pipeline. XJoin [13] and Dependent join [5] are adaptive binary join operators that are able to adjust its behavior to address a case when a source becomes blocked or delayed.

SPARQL federated query processing engines extend traditional join operators. Firstly, ANAPSID [2] introduces the Adaptive Group Join (*agjoin*) algorithm on top of XJoin and Adaptive Dependent Join (*adjoin*) on top of Dependent join. Secondly, FedX [12] describes the Controlled Bound Worker Join (CBJ) operator. Symmetric Indexed Hash Join (*SIHJoin*) [9] has been proposed to facilitate querying of both remote and local Linked Data sources. However, all the mentioned operators are binary so that they share the disadvantages and problems posed in Section 2.

nLDE [1] is a novel approach for adaptive query processing. nLDE dynamically re-arranges a query plan in presence of data transfer delays and routes intermediate results to reduce execution on time. Nonetheless, nLDE constructs query plans of binary join operators, mainly Nested Loop Joins and *agjoin*.

Multi-way join operators are well-known in the relational database community. MJoin [16] is a classic multi-way join operator that emphasizes the importance of a correct probing sequence to avoid unnecessary tuple comparisons. Other approaches to multi-way joins, for instance, GrubJoin [7] or Theta-Join [17], often rely on MapReduce which makes them blocking by nature. However, such operators are not suited nor applicable for SPARQL.

SigMR [3] is a query processing system that tries to combine multi-way joins with SPARQL and MapReduce paradigm. Albeit proposing a multi-way join operator, it is bound to MapReduce and therefore blocking, and has to compute matrices of unnecessary comparisons and prune the results set. CQELS [10] implements a windowed multi-way join operator for RDF streams processing.

In contrast, SMJoin is a non-blocking multi-way join operator for one-time query processing that identifies only a relevant probing sequence of intermediate results. Moreover, SMJoin by design supports SPARQL star-shaped subqueries.

6 CONCLUSIONS AND FUTURE WORK

We presented SMJoin, a multi-way join operator that is tailored to support SPARQL queries and star-shaped groups. To the best of our knowledge, SMJoin is the first attempt to bring non-blocking multi-way joins to the semantic data management domain. Our future work aims at creating a query decomposition and planning approach that will efficiently combine multi-way joins with binary joins depending on the query selectivity. Additionally, we plan to employ SMJoin in heterogeneous federated query engines to support a variety of data sources and their formats. Moreover, SMJoin can be extended with a multi-way semantic similarity join condition to address the case when joinable tuples are semantically equivalent but encoded differently.

ACKNOWLEDGMENTS

This work is supported in part by the European Union under the Horizon 2020 Framework Program for the project BigDataEurope (GA 644564) and by the German Ministry of Education and Research with grant no. 13N13627 for the project LiDaKrA.

REFERENCES

[1] M. Acosta and M. Vidal. Networks of linked data eddies: An adaptive web query processing engine for RDF data. In *14th ISWC, USA*, pages 111–127, 2015.
[2] M. Acosta, M. Vidal, T. Lampo, J. Castillo, and E. Ruckhaus. ANAPSID: an adaptive query processing engine for SPARQL endpoints. In *10th ISWC, Germany*, pages 18–34, 2011.
[3] J. Ahn, D. Im, and H. Kim. Sigmr: Mapreduce-based SPARQL query processing by signature encoding and multi-way join. *The Journal of Supercomputing*, 71(10):3695–3725, 2015.
[4] A. Deshpande, Z. G. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.
[5] D. Florescu, A. Y. Levy, I. Manolescu, and D. Suciu. Query optimization in the presence of limited access patterns. In *SIGMOD 1999, USA.*, pages 311–322, 1999.
[6] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database System Implementation.* Prentice-Hall, 2000.
[7] B. Gedik, K. Wu, P. S. Yu, and L. Liu. Grubjoin: An adaptive, multi-way, windowed stream join with time correlation-aware CPU load shedding. *IEEE Trans. Knowl. Data Eng.*, 19(10):1363–1380, 2007.
[8] O. Görlitz and S. Staab. Splendid: SPARQL endpoint federation exploiting void descriptions. In *COLD*, pages 13–24. CEUR-WS.org, 2011.
[9] G. Ladwig and T. Tran. SIHJoin: Querying remote and local linked data. In *8th ESWC, Greece*, pages 139–153, 2011.
[10] D. L. Phuoc, H. N. M. Quoc, C. L. Van, and M. Hauswirth. Elastic and scalable processing of linked stream data in the cloud. In *12th ISWC, Australia, 2013*, pages 280–297, 2013.
[11] M. Saleem and A. N. Ngomo. Hibiscus: Hypergraph-based source selection for SPARQL endpoint federation. In *The Semantic Web: Trends and Challenges - 11th International Conference, ESWC 2014, Anissaras, Crete, Greece, May 25-29, 2014. Proceedings*, pages 176–191, 2014.
[12] A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. Fedx: Optimization techniques for federated query processing on linked data. In *10th ISWC, Germany*, pages 601–616, 2011.
[13] T. Urhan and M. J. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Eng. Bull.*, 23(2):27–33, 2000.
[14] R. Verborgh, M. V. Sande, O. Hartig, J. V. Herwegen, L. D. Vocht, B. D. Meester, G. Haesendonck, and P. Colpaert. Triple pattern fragments: A low-cost knowledge graph interface for the web. *J. Web Sem.*, 37-38:184–206, 2016.
[15] M. Vidal, E. Ruckhaus, T. Lampo, A. Martínez, J. Sierra, and A. Polleres. Efficiently joining group patterns in SPARQL queries. In *ESWC 2010*, pages 228–242, 2010.
[16] S. Viglas, J. F. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *29th VLDB, Germany*, pages 285–296, 2003.
[17] X. Zhang, L. Chen, and M. Wang. Efficient multi-way theta-join processing using mapreduce. *PVLDB*, 5(11):1184–1195, 2012.