A Worst-Case Optimal Join Algorithm for SPARQL

Aidan Hogan[2,3], Cristian Riveros[1,3], Carlos Rojas[1,3], and Adrián Soto[1,3]

[1] Pontificia Universidad Católica de Chile
[2] DCC, Universidad de Chile
[3] Millenium Institute for Foundational Research on Data

Abstract. Worst-case optimal multiway join algorithms have recently gained a lot of attention in the database literature. These algorithms not only offer strong theoretical guarantees of efficiency, but have also been empirically demonstrated to significantly improve query runtimes for relational and graph databases. Despite these promising theoretical and practical results, however, the Semantic Web community has yet to adopt such techniques; to the best of our knowledge, no native RDF database currently supports such join algorithms, where in this paper we demonstrate that this should change. We propose a novel procedure for evaluating SPARQL queries based on an existing worst-case join algorithm called Leapfrog Triejoin. We propose an adaptation of this algorithm for evaluating SPARQL queries, and implement it in Apache Jena. We then present experiments over the Berlin and WatDiv SPARQL benchmarks, and a novel benchmark that we propose based on Wikidata that is designed to provide insights into join performance for a more diverse set of basic graph patterns. Our results show that with this new join algorithm, Apache Jena often runs orders of magnitude faster than the base version and two other SPARQL engines: Virtuoso and Blazegraph.

1 Introduction

Since its initial standardisation over a decade ago, the SPARQL query language has enjoyed broad adoption, having been implemented in a wide variety of engines (e.g., [1,16,23,30]) and supported by hundreds of public endpoints on the Web [8], the most prominent of which receive thousands or even millions of queries per day [14,22]. Despite these successes, however, there remains room for improvement. Though current SPARQL implementations now work well for processing large workloads of relatively simple queries [22], as we show in later experiments, they still struggle when evaluating queries with more complex joins; we argue that this is due, in part, to the fact that prominent SPARQL engines rely on traditional join algorithms that have not changed for over a decade.

On the other hand, a new family of join algorithms has received much attention in the recent database literature: the state-of-the-art for join evaluation has moved away from pairwise join evaluation [29], towards multiway join evaluation where an arbitrary number of joins can be evaluated at once. One of the

main benefits of the multiway approach is to minimise the number of intermediate results generated. In fact, a variety of modern multiway join algorithms – including, for example, Leapfrog Triejoin [31], Minesweeper [25], Tetris [21], CacheTrieJoin [19], etc. – have been proven to be *worst-case optimal* [26,27], meaning that the runtime of the algorithm is bounded by the worst-case cardinality of the query result (i.e. the AGM bound [11]); this theoretical guarantee implies that no other join algorithm can exist that is asymptotically faster for all database instances. Several systems (e.g. Logicblox [9] and Emptyheaded [4]) have further implemented these worst-case optimal strategies and demonstrated their superior performance in practice for evaluating queries with complex joins.

A natural idea, then, is to leverage worst-case optimal join algorithms for evaluating basic graph patterns, which form the core of SPARQL queries. However, though work has been done on adopting such algorithms for graph queries and analytics [28,4,20], to the best of our knowledge, no such work has addressed the evaluation of SPARQL basic graph patterns.

In this paper, we aim to fill this gap by investigating the benefits of worst-case optimal join algorithms for evaluating basic graph patterns. Given our goal that worst-case optimal join algorithms be widely adopted on the Semantic Web in the near future, we select Leapfrog Triejoin (LFTJ) [31] as our base algorithm since it is relatively straightforward to adapt to the case of SPARQL while still providing worst-case optimal guarantees. We propose some adaptations of the LTFJ algorithm for the SPARQL setting, proving that these adaptations do not affect the theoretical guarantees of the algorithm. We discuss how the resulting algorithm can be integrated and optimised within a native RDF store that supports multiple index orders and cardinality-based join ordering, reducing the cost of adoption. Analogously, we create a fork of Apache Jena (TDB) [1] that supports worst-case join evaluation, and proceed to evaluate its performance against the unmodified version of the engine, as well as two other prominent SPARQL engines: Virtuoso [16] and Blazegraph [30]. We run experiments on the Berlin [13] and WatDiv [6] SPARQL benchmarks, and thereafter on a novel benchmark based on Wikidata [32] from which we generate a large set of SPARQL basic graph patterns exhibiting a variety of increasingly complex join patterns. Our results show that our fork of Apache Jena can reduce the runtimes of queries with non-trivial joins by orders of magnitude versus the baseline systems.

2 Preliminaries

We introduce some brief preliminaries for RDF and SPARQL used throughout and thereafter discuss the central notion of worst-case optimal joins.

RDF: RDF is the graph-based data model at the heart of the Semantic Web. RDF terms can be IRIs ($\mathbf{I}$), literals ($\mathbf{L}$) or blank nodes ($\mathbf{B}$). A triple $(s, p, o) \in \mathbf{IB} \times \mathbf{I} \times \mathbf{IBL}$ is called an RDF triple, where s is called the subject, p the predicate, and o the object.[4] An RDF graph is a set of RDF triples.

[4] We use $\mathbf{IB}$ as a shortcut for $\mathbf{I} \cup \mathbf{B}$, etc.

SPARQL: SPARQL is the standard query language for RDF [3]. Let $\mathbf{V}$ be a set of variables. A tuple $t \in \mathbf{ILV} \times \mathbf{IV} \times \mathbf{ILV}$ is called a triple pattern. Blank nodes in triple patterns can be considered as query variables for our purposes. A set of triple patterns is called a basic graph pattern. We denote by $\mathrm{var}(t)$ and $\mathrm{var}(P)$ the set of variables found in a triple pattern t and basic graph pattern P, respectively. We call a variable $?\mathtt{x} \in \mathrm{var}(P)$ a *join variable* if it appears in two or more triple patterns of P, and a *lonely variable* otherwise.

The semantics of SPARQL queries is defined in terms of mappings. A mapping μ is a partial function $\mu : \mathbf{V} \to \mathbf{IBL}$. The domain of μ, denoted $\mathrm{dom}(\mu)$, is the set of variables on which μ is defined. Given a triple pattern t, we denote by $\mu(t)$ the image of the triple pattern t under μ: the triple obtained by replacing the variables in t according to μ. We say that two mappings μ_1 and μ_2 are compatible, denoted $\mu_1 \sim \mu_2$, iff $\mu_1(?\mathtt{x}) = \mu_2(?\mathtt{x})$ for every $?\mathtt{x} \in \mathrm{dom}(\mu_1) \cap \mathrm{dom}(\mu_2)$. Given sets of mappings Ω_1 and Ω_2, we then define their join as $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \text{ and } \mu_1 \sim \mu_2\}$.

We can now define the evaluation of a triple pattern and a basic graph pattern over an RDF graph G (the latter being defined as a join over its triple patterns):

$$\begin{aligned} [\![t]\!]_G &= \{\mu \mid \mathrm{var}(t) = \mathrm{dom}(\mu) \text{ and } \mu(t) \in G\} \\ [\![\{t_1, \ldots, t_n\}]\!]_G &= [\![t_1]\!]_G \bowtie \ldots \bowtie [\![t_n]\!]_G \end{aligned}$$

Letting $\mu(P)$ denote the image of P under μ, with respect to the latter definition, we can equivalently say that $[\![P]\!]_G = \{\mu \mid \mathrm{dom}(\mu) = \mathrm{var}(P) \text{ and } \mu(P) \subseteq G\}$.

SPARQL further offers a wide range of query operators that can be used to combine or modify the results of basic graph patterns, such as union, optional, filters, aggregates, property paths, etc. In this paper, we focus on optimising the evaluation of basic graph patterns, which form the core of SPARQL queries; other SPARQL operators can be supported by applying standard techniques over the mappings generated from the query's basic graph patterns.[5] However, there is the possibility for bespoke methods that merge the evaluation of some of these operators – in particular optional, property paths, named graphs, etc. – with the evaluation of basic graph patterns by the proposed worst-case join algorithm. We leave the exploration of such embedded optimisations for future work. Furthermore, SPARQL assumes a default bag semantics, which preserves duplicates [7]; though we evaluate sets of solutions for basic graph patterns, such patterns alone never generate duplicate mappings, and thus our proposal is compatible with bag semantics being applied in higher-level query operators.

Worst case optimality: A join algorithm is called worst-case optimal if it satisfies the AGM bound [11], namely, if the running time over an instance G is bounded by the worst-case output size over all instances of the same size as G. Specifically, let P be a BGP and G an RDF graph. Consider the following linear

[5] Other features like `BIND`, `VALUES`, `SERVICE`, etc., that generate or extend mappings can be evaluated in the standard way.

program [11] adapted for the case of RDF and basic graph patterns:

$$
\begin{array}{lll}
\text{minimize} & \sum_{t \in P} x_t \cdot \log(|[\![t]\!]_G|) & \\
\text{subject to} & \sum_{t : ?\mathtt{x} \in \mathrm{var}(t)} x_t \geqslant 1 & \text{for each } ?\mathtt{x} \in \mathrm{var}(P) \\
 & x_t \geqslant 0 & \text{for each } t \in P
\end{array}
$$

where x_t is a variable for each $t \in P$. If $\mathrm{MIN}(P, G)$ is the minimum for the above optimization problem, then the AGM bound states that $|[\![P]\!]_G| \leqslant 2^{\mathrm{MIN}(P,G)}$ and this bound is tight: there exists an RDF graph G' of the same size as G where $|[\![P]\!]_{G'}|$ is equal to $2^{\mathrm{MIN}(P,G)}$ up to a logarithmic factor. We call an evaluation algorithm for a basic graph pattern *worst case optimal* if its running time is at most $2^{\mathrm{MIN}(P,G)}$ up to a logarithmic factor. All of our algorithmic analysis is done in data complexity where the size of the query is considered as fixed.

3 Related Work

Our goal is to optimise the evaluation of basic graph patterns in SPARQL. Here we first discuss the standard evaluation methods used in popular SPARQL engines, proposals of multiway joins for SPARQL, works on worst-case optimal join algorithms, and a summary of the novelty of our present work.

Indexing: In order to efficiently evaluate triple patterns, SPARQL engines employ indexes that offer optimised access to the underlying data; such engines will often build a complete index that can efficiently evaluate a triple pattern with any combination of constants and variables [18]. A complete index is comprised of multiple index orders, where a single index order with prefix lookups can be used to evaluate multiple forms of triple pattern;[6] for example, the index order **pos** allows for directly evaluating triples patterns of the form $(?, ?, ?)$, $(?, p, ?)$, $(?, p, o)$ and (s, p, o) without filtering, but not $(s, ?, ?)$, which would require reading all triples from the **pos** index and filtering those whose subject does not match the triple pattern (a better choice would be an index order like **spo** or **sop**). Some SPARQL engines build complete indexes for triples [33,23,10], while others directly support named graphs by indexing quads [18,16]. In terms of indexing implementations, one option is to apply standard data structures known from relational databases, such as B+Tree indexes [18,23,16]; another option is to develop RDF-specific techniques, such as nested data structures [33], bit matrices [10], etc., that take advantage of the fixed arity of triples.

Pairwise joins: While a complete index allows individual triple patterns to be evaluated efficiently, the evaluation of basic graph patterns requires applying join algorithms over the mappings generated from triple patterns. The most popular strategy for evaluating basic graph patterns is to use pairwise evaluation joining two sets of mappings at a time. In left-deep plans, the results of a triple

[6] Following [18], we use the notation $(s|?, p|?, o|?)$ to denote eight forms of triple patterns where, for example, $(?, p, o)$ refers to the set of triple patterns with variable subject, constant predicate and constant object: $\mathbf{V} \times \mathbf{I} \times \mathbf{IL}$.

pattern are joined with the current results of all joins thus far; for example, taking a basic graph pattern with four triple patterns, an example left-deep evaluation would be $(((t_1 \bowtie t_2) \bowtie t_3) \bowtie t_4)$ [18]. In bushy plans, two sets of join results can also be joined, leading to more balanced query plans; for example, $((t_1 \bowtie t_2) \bowtie (t_3 \bowtie t_n))$ is an instance of a bushy plan [23]. To implement such joins, SPARQL engines often use variants of well-known algorithms for join evaluation in relational databases, such as nested-loop joins [18,23], hash joins [23], and sort-merge joins [23]. An important aspect of optimising SPARQL query plans is then to exploit the commutativity and associativity of joins to find a query plan that minimises the number of intermediate results generated; a common strategy is to rely on cardinality estimates [18,23,16].

Multiway joins: Multiway join algorithms perform joins over two or more sets of mappings at once; a common strategy is to group, evaluate and join triple patterns sharing a given variable as a single operation. Multiway join evaluation can thus reduce the number of intermediate results that are generated. To the best of our knowledge, few works have investigated multiway joins in the context of SPARQL. One exception is the recent work of Galkin et al. [17], who propose a join algorithm for SPARQL queries called SMJoin that groups blocks of star-shaped joins (where a common join variable is present in the subject position) and applies multiway joins over each block. Experimental results show that the multiway join performs well for selective query patterns, but is outperformed by a pairwise-join baseline for other types of queries (due to the latter applying selectivity-based join reordering not available to SMJoin).

Worst-case optimal joins: Various works in the database literature have focused on worst-case optimal join algorithms [31,25,28,21,19], which have also been implemented as part of commercial databases [9,4]. A subset of such works have looked at the benefits of such algorithms for answering queries over graphs, incorporating experiments for evaluating queries based on graph patterns including cliques, trees, paths, etc. [28,4,19]; Aberger et al. [4] further provide experiments for analytical queries on graphs, such as Pagerank and shortest paths. While these works have provided evidence as to the value of worst-case optimal join algorithms for graphs, they do not address the SPARQL setting.

Novelty: We propose a multiway join algorithm for evaluating basic graph patterns in SPARQL based on Leapfrog Triejoin [31], modifying how it accesses indexes to ensure better compatibility with current SPARQL implementations. We prove that the adapted algorithm remains worst-case optimal, discuss its implementation in Jena, and provide experimental results analysing its runtime performance. Unlike the work of Galkin et al. [17], our multiway join algorithm is agnostic to the position of a join variable in a triple pattern. More generally, and to the best of our knowledge, this is the first work to explore the application of a worst-case join algorithm for evaluating SPARQL basic graph patterns.

4 Leapfrog Join for Basic Graph Patterns

Our goal is to investigate the potential benefits of using a worst-case optimal join algorithm on SPARQL query performance. Surveying the state-of-the-art algorithms in the database literature [31,25,28,21,19,24], we opted to base our algorithm on Leapfrog Triejoin algorithm (LFTJ) [31], mainly because it is the most concise among all such algorithms [24], and thus a good starting point for implementation within a SPARQL engine. We first present here a logical version of LFTJ that we call *Leapfrog Join* (LFJ), which includes only the core evaluation strategy on which LFTJ is based. LFJ can be divided into two main phases: *Leapfrog* and *variable elimination*. We begin by discussing both phases and give a running example of the algorithm. Later we propose a physical version of Leapfrog Join, designed to be easily integrated with existing SPARQL engines, mostly requiring adaptations at the index layer (see the discussion in Section 5).

Leapfrog: Unlike traditional join algorithms that evaluate triple pattern by triple pattern, Leapfrog Join rather proceeds by evaluating variable by variable. An important procedure in Leapfrog Join is to compute all *non-trivial outputs of a single variable*; more formally, given an RDF graph G, a basic graph pattern P and a variable $?\mathtt{x}$ in $\mathrm{var}(P)$ we want to compute the following set:

$$\mathrm{LF}_G(P, ?\mathtt{x}) \;=\; \{\mu \mid \mathrm{dom}(\mu) = \{?\mathtt{x}\} \text{ and } [\![\mu(t)]\!]_G \neq \varnothing \text{ for all } t \in P\}\,.$$

In other words, we want to identify all single variable mappings μ such that, for every $t \in P$, the output of $\mu(t)$ over G is non-empty when $?\mathtt{x}$ is replaced by $\mu(?\mathtt{x})$. Intuitively, if $\mu \in \mathrm{LF}_G(P, ?\mathtt{x})$, then μ is a good candidate for a partial mapping that can be extended to form an output mapping in $[\![P]\!]_G$. Note also that if $?\mathtt{x}$ is the only variable used in P (i.e., $\mathrm{var}(P) = \{?\mathtt{x}\}$), then the set $\mathrm{LF}_G(P, ?\mathtt{x})$ is the same as computing the intersection of all sets $[\![t]\!]_G$. In Section 5, we will show how to implement this function for one or more variables by exploiting standard B+tree indexes while maintaining worst-case optimality.

Variable elimination: While the Leapfrog phase evaluates a single variable, the variable elimination phase evaluates multiple variables. Given a basic graph pattern P with $\mathrm{var}(P) = \{x_1, \ldots, x_n\}$, an RDF graph G, and a variable order $O_{\mathrm{var}} = ?\mathtt{x}_1, \ldots, ?\mathtt{x}_m$, Algorithm 1 shows the nested structure of the variable elimination procedure, which constitutes the overall Leapfrog Join process. The procedure iterates over each variable $?\mathtt{x}_i$ in order, extending the mapping μ_i with a mapping $\mu \in \mathrm{LF}_G(\mu_i(P), ?\mathtt{x}_{i+1})$. Variable $?\mathtt{x}_i$ is fixed by extending μ with μ_i (i.e. $\mu_{i+1} = \mu_i \cup \mu$; note that $\mu_i \sim \mu$, so μ_{i+1} is also a mapping); in this way, variable $?\mathtt{x}_i$ is "eliminated" from P. The procedure moves on to eliminate the next variable $?\mathtt{x}_{i+1}$ analogously. After all variables $?\mathtt{x}_1, \ldots, ?\mathtt{x}_m$ are eliminated, the mapping $\mu_{m-1} \cup \mu$ is output, and the search for the next output is continued.

Figure 4.1 provides an example of variable elimination for a basic graph pattern over an RDF graph. We assume the order $?\mathtt{x}_1 \ldots ?\mathtt{x}_4$; how such an order is decided will be discussed later in Section 5.[7] Pairwise evaluation with this

[7] Such an order would be produced by SPARQL engines in practice if we had a graph with many `:father` and `:mother` relations, outnumbering `:winner` relations.

Algorithm 1: Variable elimination for basic graph patterns

input : RDF graph G, BGP P, variable order $O_{\text{var}} = ?\mathtt{x}_1 \ldots ?\mathtt{x}_n$
output: All mappings $[\![P]\!]_G$.

1 **Function** `LFTJ-Eval` (G, P, O_{var})
2 $\quad \mu_0 \leftarrow \varnothing$
3 $\quad$ **foreach** $\mu \in \text{LF}_G(\mu_0(P), ?\mathtt{x}_1)$ **do**
4 $\quad\quad \mu_1 \leftarrow \mu_0 \cup \mu$
5 $\quad\quad$ **foreach** $\mu \in \text{LF}_G(\mu_1(P), ?\mathtt{x}_2)$ **do**
6 $\quad\quad\quad \mu_2 \leftarrow \mu_1 \cup \mu$
7 $\quad\quad\quad \ddots$
8 $\quad\quad\quad\quad$ **foreach** $\mu \in \text{LF}_G(\mu_{n-1}(P), ?\mathtt{x}_n)$ **do**
9 $\quad\quad\quad\quad\quad$ **Output** $\mu_{n-1} \cup \mu$ `// write to output and continue`

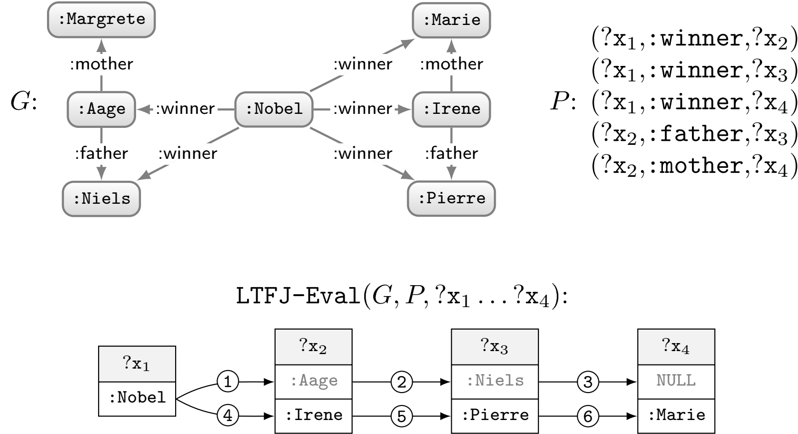

Fig. 4.1. Example of Leapfrog join for evaluating a SPARQL basic graph pattern

triple-pattern order would naively produce $5^3 = 125$ intermediary results containing the Cartesian product of all five winners of the Nobel prize (as would the multiway star-shaped join algorithm of Galkin et al. [17]). On the other hand, under Leapfrog Join, variable elimination ensures that when, e.g., $?\mathtt{x}_2$ is evaluated, only those winners that have some father and some mother are considered. The lower graph then shows the recursion order producing the final result(s).

5 A Physical Operator for Leapfrog Join

We implement Leapfrog Join (LFJ) in Apache Jena TDB version 3.9.0, which implements nested-loop joins on top of B+tree indexes. We choose Jena as it is one of the most widely-deployed (fully) open source SPARQL engines; however

the methods described can be generalised to other SPARQL engines. We now explain the main modifications required to support LFJ in Jena.

Indexes for LFJ: The first modification needed to run LFJ was to extend the index layer in Jena. Recall that a major phase in LFJ is to compute the set $\mathrm{LF}_G(P, ?\mathtt{x})$ given an RDF graph G, a basic graph pattern P and a variable $?\mathtt{x}$. The next result shows that Leapfrog Join is a worst-case optimal join algorithm whenever the computation of $\mathrm{LF}_G(P, ?\mathtt{x})$ is done in a reasonable time.

Theorem 1. *An implementation of Leapfrog Join is worst-case optimal if, for every RDF graph G, basic graph pattern P, and variable $?\mathtt{x}$, the computation of $\mathrm{LF}_G(P, ?\mathtt{x})$ is done in time at most:*

$$O\big(\max(\min_{t \in P : ?\mathtt{x} \in \mathrm{var}(t)} |\pi_{?\mathtt{x}}([\![t]\!]_G)|, 1) \cdot \log(|G|)\big)$$

where $\pi_{?\mathtt{x}}([\![t]\!]_G)$ is the projection of $[\![t]\!]_G$ over $?\mathtt{x}$.

The proof of Theorem 1 is given in Appendix A.

Calculating $\mathrm{LF}_G(P, ?\mathtt{x})$ is the same as computing the intersection of all sets $[\![t]\!]_G$; hence, one can use any adaptive intersection algorithm over n sets [15,12], which satisfies the time restriction of Theorem 1. In particular, our implementation of LFJ uses the intersection algorithm proposed by Veldhuizen [31].

The algorithm of adaptive intersection assumes that each set $\pi_{?\mathtt{x}}([\![t]\!]_G)$ can be navigated in increasing order. For this, we need an index I_G such that for every triple pattern t and every variable $?\mathtt{x}$ in t, it provides a seek method $\mathrm{I}_G[t, ?\mathtt{x}].\mathtt{seek}(\mathtt{:a})$ that outputs the least $\mathtt{:b}$ such that $\mathtt{:b} \geqslant \mathtt{:a}$ and $[\![\mu(t)]\!]_G \neq \varnothing$ for $\mu = \{?\mathtt{x} \rightarrow \mathtt{:b}\}$, or `NULL` if no such $\mathtt{:b}$ exists; in other words, the seek method jumps to the next non-trivial output for $?\mathtt{x}$ in the order. To satisfy the bound of Theorem 1, the seek method is required to take time logarithmic in the size of G. Although the original LFTJ algorithm proposes to use tries for I_G, such a seek method can be supported using B+Trees adding all six orders over s, p and o. Hence to Jena's three default orders **spo**, **pos**, and **osp**, our implementation adds three more orders: **sop**, **pso**, and **ops**. This roughly doubles the size of the on-disk index and the number of update operations required to add/remove triples, but (as shown later) offers gains in query performance with LFJ.[8]

Each index order is assigned a B+tree, where the seek method could then be implemented by traversing the B+tree top-down from root to leaf in the standard way. However, given that the seek method requests values in sequential order, we use a stack to store the current node in the iteration, its leaf, and its parents; when the next value is requested, we can read the next value in the order from the leaf or, starting from there, search the B+tree upwards and then back down in case that the next value is in another leaf. This bottom-up seek method offers

[8] We currently consider querying over a single RDF graph; if we were to consider a complete index on quads in order to support named graphs, the number of required indexes would jump to 24. In such a case, however, practical steps can be taken to reduce the number of indexes where, for example, some such orders will be rarely accessed by real-world queries and can thus be removed.

constant amortized time when only one variable is unbound [31], logarithmic time when two variables are unbound, and is more efficient in practice.

LFJ operator: We add a new LFJ join operator to Jena that takes a basic graph pattern and evaluates it using our implementation of LFJ (per Algorithm 1). Note that the original LFTJ algorithm applies some restrictions: (1) each relation symbol must appear only once, (2) the order of attributes of the relations (triple patterns in our case) must follow the global attribute order, (3) constants cannot appear within the join query and (4) each attribute can appear at most once in each relation (triple pattern in our case). The first restriction does not apply for our implementation. Restrictions (2) and (3) are not required to maintain worst-case optimality and are addressed by our indexes. The case of variables occurring twice in a triple pattern requires some extra care, but can be addressed with special indexes for triples repeating the same term in the given positions (which are typically uncommon in RDF data), or using a fresh variable and applying a low-level filter/intersection; we omit these details for brevity.

Variable order: The performance of LFJ is dependent on the chosen variable order [5]; referring back to Figure 4.1, for example, a more efficient order would be to swap $?\mathtt{x}_2$ and $?\mathtt{x}_4$, which would allow for more quickly rejecting the incomplete mapping involving `:Aage` in G. In principle, the goal of finding a variable ordering is similar to that for ordering triple patterns: in both cases, we wish to evaluate highly-selective triple patterns/variables that help to filter mappings early on. Along these lines, while specialised variable orderings have been proposed for worst-case optimal join algorithms [5], we propose a solution based on Jena's existing triple ordering; this has the additional advantage of making experiments between the baseline version of Jena and Jena with LFJ more comparable.

Given a triple-pattern order O_{trip} returned by Jena, we first choose join variables in order of appearance, and then select lonely variables in order of appearance; for example, if Jena gives $O_{\text{trip}} = (\mathtt{?z}, \mathtt{:p3}, \mathtt{?u}), (\mathtt{?x}, \mathtt{:p2}, \mathtt{?z}), (\mathtt{?x}, \mathtt{:p1}, \mathtt{?y})$, we will choose the variable order $O_{\text{var}} = \mathtt{?z}, \mathtt{?x}, \mathtt{?u}, \mathtt{?y}$ since `?z` is the first join variable that appears in O_{trip}, and `?x` is the second join variable that appears in O_{trip}; given that `?u` and `?y` are lonely variables (appearing in one triple pattern), they come after the join variables, again based on order of appearance. In fact, as we now discuss, the order of lonely variables will not affect performance.

Enumerating mappings: Early experiments comparing Jena with and without LFJ found that the performance of the former was sometimes orders of magnitude *worse* than the latter. We identified the issue as relating to how lonely variables are handled. To illustrate this issue, consider a graph pattern P' containing only the first three triple patterns of P in Figure 4.1 such that $?\mathtt{x}_2$, $?\mathtt{x}_3$ and $?\mathtt{x}_4$ are lonely variables. Applying the procedure of Algorithm 1, after assigning $?\mathtt{x}_1 \rightarrow$ `:Nobel`, we still require $5 \times 5 \times 5$ steps through the recursion, repetitively evaluating the same partially-bound triple patterns. This final recursion is unnecessary: since lonely variables are evaluated last, we know that the final mappings must be extended by the Cartesian product of the non-trivial outputs of the remaining lonely variables. To address this, assume a variable order $O_{\text{var}} = \mathtt{?x}_1, \ldots, \mathtt{?x}_m, \mathtt{?x}_{m+1}, \ldots \mathtt{?x}_n$ where $\mathtt{?x}_1, \ldots, \mathtt{?x}_m$ are join variables

and $?x_{m+1}, \ldots, ?x_n$ are lonely variables. Assume also that $t_1, \ldots, t_k$ are the triple patterns where $?x_{m+1}, \ldots, ?x_n$ are mentioned (each such triple pattern may mention one or more lonely variables). We eliminate $?x_1, \ldots, ?x_m$ per Algorithm 1, and for each partial solution μ_m generated, we compute the Cartesian product $\mu_m \times [\![\mu_m(t_1)]\!]_G \times \ldots \times [\![\mu_m(t_k)]\!]_G$, requiring k (note that $k < n - m$) additional calls to $[\![\mu_m(\cdot)]\!]_G$ for each μ_m (rather than having to call LF a total of $1 + \sum_{i=m}^{n-2} \prod_{j=m}^{i} |\,\mathrm{LF}_G(\mu_m(P), ?x_{j+1})|$ times for each μ_m).

6 Experiments and Results

We now compare the performance of query evaluation for Apache Jena (TDB) v.3.9.0 with LFJ, Apache Jena v.3.9.0 without LFJ, Virtuoso v.OS-7.2.7 [16] (one of the most deployed engines in practice [8]), and Blazegraph v.2.1.4 [30] (used by the Wikidata Query Service [22]). We run three sets of experiments using the Berlin SPARQL Benchmark [13], the WatDiv Benchmark [6], and a novel Wikidata Benchmark with complex graph patterns that we propose. We run all experiments on a single machine with Ubuntu 16.04.5, Intel Xeon CPU E5-2609 v4@1.70GHz, Seagate 1TB Enterprise Capacity 2.5-Inch HDD, and 32GB RAM. Code and configurations can be found online for reproducibility purposes [2].

6.1 Experiments on the Berlin SPARQL Benchmark

We first ran experiments over the Berlin SPARQL Benchmark (BSBM) [13], comparing query runtimes for Jena with (denoted Jena-LFJ) and without (denoted Jena) the LFJ modifications. We run the Explore Use-Case of BSBM, consisting of 12 queries using a mix of SPARQL 1.0 features, including optional, union, filter, graph, etc. In Figure 6.2 we show the average time of each query in logarithmic scale. These experiments were done by running 10,500 queries; we found that on average each query took 49.3 ms for Jena-LFJ and 41.6 ms for Jena. We conclude that the BSBM results show no clear trend to suggest that one implementation outperforms the other. BSBM queries do not contain large intermediary results and, thus, Jena-LFJ offers no improvement. Furthermore, given that BSBM queries contain other features of SPARQL, the baseline of Jena can use optimisations for other operators not currently available for Jena-LFJ (in particular, pushing range filters, which appear in many BSBM queries).

6.2 Experiments on the WatDiv Benchmark

After reviewing the BSBM results, we still foresaw the need to run experiments on queries with more complex and diverse basic graph patterns. We chose the WatDiv benchmark [6] which is designed for this purpose. We generate 50 queries for each of the 20 abstract patterns proposed in the benchmark. Executing the $50 \times 20 = 1000$ query instances and taking the average over all of them, Virtuoso takes 64 s, Jena-LFJ takes 77 s, Blazegraph takes 99 s, and Jena takes 198 s. Boxplots of runtimes for each specific query pattern are shown in Figures 6.3 and

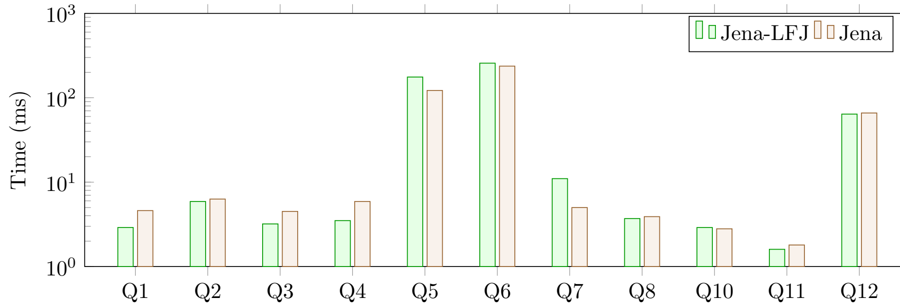

Fig. 6.2. Plot of runtimes for queries of the Berlin Benchmark with log y-axis.

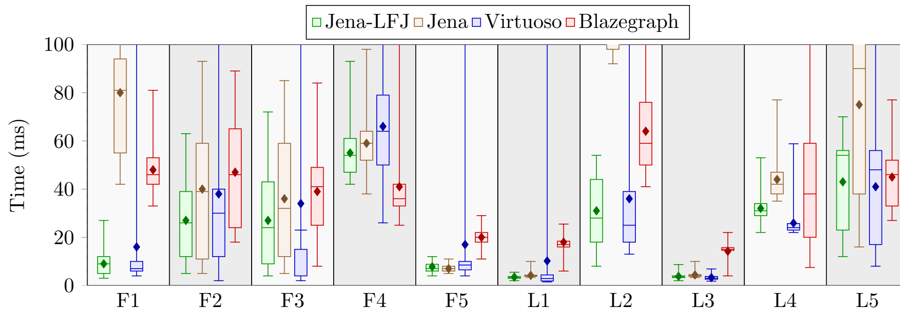

Fig. 6.3. Box plots of runtimes for queries L and F of the WatDiv Benchmark.

6.4. Unlike in the BSBM experiments, here Jena-LFJ is at least twice as fast as Jena in terms of the overall query runtime and it also outperforms Blazegraph. Indeed, these plots suggest that the running time of Jena-LFJ is much more stable than other implementations; the interquartile difference is at most 40 ms. Since this benchmark is oriented towards testing basic graph patterns, we can see here that our implementation is competitive with respect to the other engines, being slightly outperformed by Virtuoso. Despite this analysis, the runtimes of these queries are still in the order of less than 100 ms, making it difficult to claim that Jena-LFJ or Virtuoso is the best approach.

6.3 Experiments on the Wikidata Graph Pattern Benchmark

Though WatDiv contains more complex graph patterns than Berlin, it does not contain (for example) graph patterns with cycles; furthermore, both benchmarks are based on synthetic data with relatively simple schemata (e.g., BSBM and WatDiv have 30 and 85 distinct predicates, respectively). In order to compare the four engines for real data and a more diverse set of both acyclical and cyclical graph patterns, we thus developed a new benchmark that we call the Wikidata Graph Pattern Benchmark (WGPB).

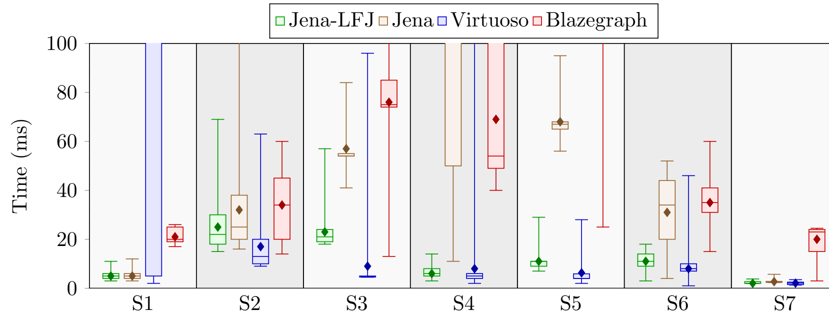

Fig. 6.4. Box plots of runtimes for queries S of the WatDiv Benchmark

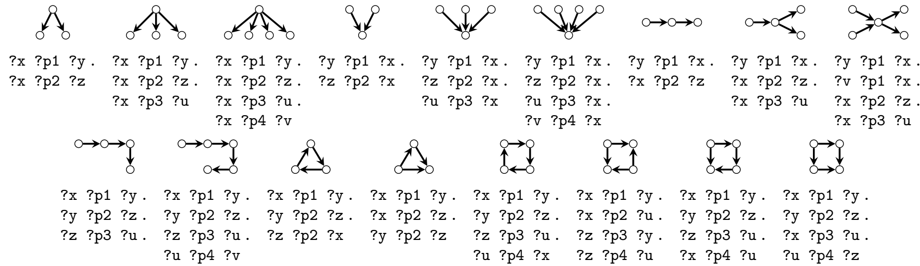

Fig. 6.5. Basic graph patterns and their associated diagram

Dataset: To generate the WGPB dataset, we take the Wikidata "truthy" dump from 2018/11/15. This dump contains 3,303,288,386 triples. Given that our goal is to develop queries on the graph structure of Wikidata, we remove labels, aliases, and descriptions, leaving 969,496,651 triples with 5,419 unique predicates. Given that we will later apply random sampling, we removed triples whose predicate appeared fewer than 1,000 times to ensure that we avoid generating trivial query instances. Finally, we also remove triples whose predicates appear in more than 1,000,000 triples. The result, which we call the Wikidata Core Graph (WCG), contains 82,923,234 triples with 2,101 distinct predicates.

Queries: To achieve a set of queries with diverse graph patterns, we create instances of the 17 abstract basic graph patterns shown in Figure 6.5; we focus on joins between subjects and objects as common in real-world queries [14]. For each abstract pattern we instantiate 50 queries using random walks in WCG per the given pattern; each instance replaces the predicate variables by the IRIs found on the walk; for example, the first pattern `?x ?p1 ?y . ?x ?p2 ?z` may be instantiated as `SELECT * WHERE{ ?x wdt:P57 ?y . ?x wdt:P166 ?z }`.

Results with single join variable: We first present results for queries with a single join variable (the top row of Figure 6.5), analysing the performance of

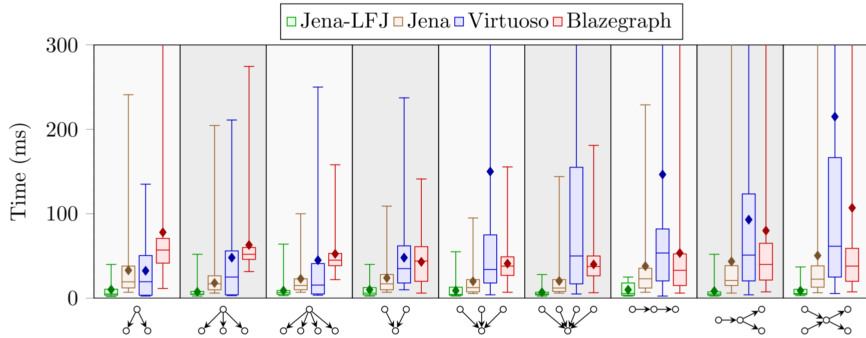

Fig. 6.6. Box plots of runtimes for queries with a single join variable

the Leapfrog procedure of LFJ. Executing the $50 \times 9 = 450$ query instances, in terms of overall query runtime across all patterns, Jena-LFJ takes 4.0 s, Jena takes 14.0 s, Blazegraph takes 27.9 s, and Virtuoso takes 64.8 s. Figure 6.6 then shows the detailed results per query pattern, where we focus the y-axis in on the range of 0–300 ms for clarity. Here we see that Jena-LFJ is at least twice as fast as Jena in terms of median or mean times, and can be 10–20 times faster than the slowest engine for some queries. The most notable speedup occurs when join variables appear in the object position, which may lead to many intermediate results when a node with high in-degree (e.g., a country) is involved; in such cases, LFJ performs better than other engines. One might consider that this speedup may be attributable to the lack of the three additional orders of **s**, **p** and **o** in the other engines. However, in the case of the best gains – i.e., joins in the object position – Jena-LFJ is using the **pos** index, which is already included in Jena; more generally, Jena uses index nested loop joins, which cannot benefit from further index orders when evaluating BGPs/equijoins.

We further observe that the runtimes for Jena-LFJ are more stable, with the maximum runtime never exceeding 55 ms; furthermore, within the 50 queries of each abstract pattern, the standard deviation in runtimes for Jena-LFJ is consistently around 9 ms, while Jena's standard deviation is always over 20 ms, and that of Virtuoso and Blazegraph is even higher, sometimes over 100 ms.

Results with multiple join variables: We now present results for queries with multiple join variables (the bottom row of Figure 6.5). Given that the previous experiments test the performance of Leapfrog for intersecting results for join variables in up to four patterns, our focus now is on the performance of variable elimination. We thus select abstract graph patterns where each variable appears in at most two triple patterns; such queries put as much emphasis as possible on the performance of the variable elimination phase versus the Leapfrog phase tested previously. Executing the $50 \times 8 = 400$ query instances, in terms of the overall query runtime across all patterns, Jena-LFJ takes 12 s, Virtuoso takes 37 s, Jena takes 112 s, and Blazegraph takes 35 s. Figure 6.7 again shows the detailed results focusing on the same y-axis range for clarity. We again see

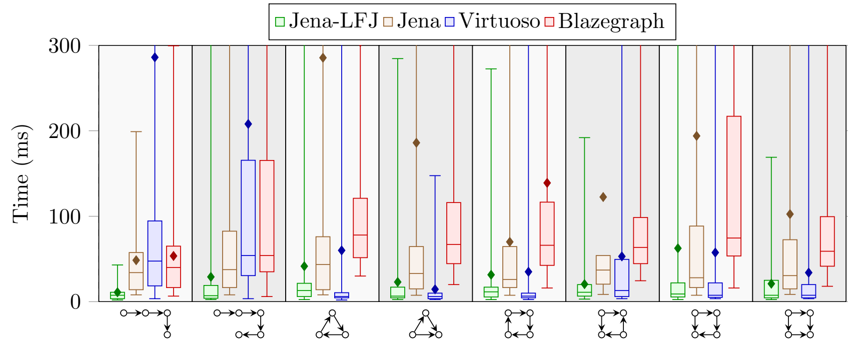

Fig. 6.7. Box plots of runtimes for queries with multiple join variables

that Jena-LFJ generally exhibits the most stable runtimes, clearly outperforming Jena and Blazegraph for all patterns and Virtuoso for the first two patterns. Comparing Jena-LFJ and Virtuoso for the latter six patterns (those with cycles), Virtuoso is competitive with and sometimes even outperforms Jena-LFJ; analysing further, we found that Virtuoso often chooses a better execution order than Jena(-LFJ), where manually optimising the variable order in Jena-LFJ for such cases results in much better performance than Virtuoso; this suggests that the variable ordering of Jena-LFJ could be improved. Even with the current variable ordering of Jena-LFJ, however, the clear gains in the first two patterns vs. Virtuoso outweigh slight gains by Virtuoso in some of the latter six patterns, as evidenced by the total runtimes mentioned previously (12 s vs. 37 s).

7 Conclusions

To the best of our knowledge, this is the first work to look at the benefits of worst-case optimal join algorithms in a SPARQL setting. Based on our results, we believe that worst-case optimal joins should become widely adopted by SPARQL engines in the near future; we also firmly believe that our results are only a starting point in this line of research, and that there is still much room left for maximising the potential benefits of such algorithms in a SPARQL setting. Along these lines, we have released an open source fork of Apache Jena implementing LFJ that can serve as a baseline for future experiments, and a novel benchmark based on Wikidata that can be used for testing future developments in a real-world setting. In terms of future work, we identify three main lines of research, investigating: (i) the potential benefits of other worst-case optimal join algorithms for SPARQL [31,25,28,21,19,24]; (ii) effective ways to optimise the variable order [21,5]; (iii) optimisations that push the evaluation of other SPARQL operators – particularly optional patterns, property paths, difference, and named graphs – into the worst-case optimal process.

Acknowledgements This work was supported by the Millennium Institute for Foundational Research on Data (IMFD) and by Fondecyt Grant No. 1181896.

References

1. Apache Jena. https://jena.apache.org/. Accessed on 2018-12-30.
2. Github project. https://gqgh5wfgzt.github.io/benchmark-leapfrog/.
3. SPARQL 1.1 Query Language. https://www.w3.org/TR/sparql11-query/. Accessed on 2018-12-30.
4. C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. *ACM Transactions on Database Systems (TODS)*, 42(4):20, 2017.
5. M. Abo Khamis, H. Q. Ngo, and A. Rudra. FAQ: questions asked frequently. In *Principles of Database Systems (PODS)*, pages 13–28. ACM, 2016.
6. G. Aluç, O. Hartig, M. T. Özsu, and K. Daudjee. Diversified stress testing of RDF data management systems. In *International Semantic Web Conference (ISWC)*, pages 197–212. Springer, 2014.
7. R. Angles and C. Gutiérrez. The Multiset Semantics of SPARQL Patterns. In *International Semantic Web Conference (ISWC)*, pages 20–36. Springer, 2016.
8. C. B. Aranda, A. Hogan, J. Umbrich, and P. Vandenbussche. SPARQL web-querying infrastructure: Ready for action? In *International Semantic Web Conference (ISWC)*, pages 277–293. Springer, 2013.
9. M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn. Design and implementation of the LogicBlox system. In *SIGMOD International Conference on Management of Data*, pages 1371–1382. ACM, 2015.
10. M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "Bit" loaded: a scalable lightweight join query processor for RDF data. In *World Wide Web (WWW)*, pages 41–50, 2010.
11. A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. In *Foundations of Computer Science (FOCS)*, pages 739–748. IEEE, 2008.
12. J. Barbay and C. Kenyon. Adaptive intersection and t-threshold problems. In *Symposium on Discrete Algorithms (SODA)*, pages 390–399. Society for Industrial and Applied Mathematics, 2002.
13. C. Bizer and A. Schultz. The Berlin SPARQL Benchmark. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(2):1–24, 2009.
14. A. Bonifati, W. Martens, and T. Timm. An analytical study of large SPARQL query logs. *PVLDB*, 11(2):149–161, 2017.
15. E. D. Demaine, A. López-Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In *Symposium on Discrete Algorithms (SODA)*. Citeseer, 2000.
16. O. Erling and I. Mikhailov. RDF Support in the Virtuoso DBMS. In *Networked Knowledge – Networked Media*. Springer, 2009.
17. M. Galkin, K. M. Endris, M. Acosta, D. Collarana, M. Vidal, and S. Auer. SMJoin: A Multi-way Join Operator for SPARQL Queries. In *International Conference on Semantic Systems (SEMANTICS)*, pages 104–111, 2017.
18. A. Harth and S. Decker. Optimized Index Structures for Querying RDF from the Web. In *Latin American Web Congress (LA-Web 2005)*, pages 71–80, 2005.
19. O. Kalinsky, Y. Etsion, and B. Kimelfeld. Flexible Caching in Trie Joins. In *International Conference on Extending Database Technology (EDBT)*, pages 282–293. Springer, 2017.
20. O. Kalinsky, O. Mishali, A. Hogan, Y. Etsion, and B. Kimelfeld. Efficiently charting RDF. *CoRR*, abs/1811.10955, 2018.

21. M. A. Khamis, H. Q. Ngo, C. Ré, and A. Rudra. Joins via geometric resolutions: Worst case and beyond. *ACM Transactions on Database Systems (TODS)*, 41(4):22, 2016.
22. S. Malyshev, M. Krötzsch, L. González, J. Gonsior, and A. Bielefeldt. Getting the Most Out of Wikidata: Semantic Technology Usage in Wikipedia's Knowledge Graph. In *International Semantic Web Conference (ISWC)*, pages 376–394. Springer, 2018.
23. T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *PVLDB*, 1(1):647–659, 2008.
24. H. Q. Ngo. Worst-case optimal join algorithms: Techniques, results, and open problems. In *Principles of Database Systems (PODS)*, pages 111–124. ACM, 2018.
25. H. Q. Ngo, D. T. Nguyen, C. Re, and A. Rudra. Beyond worst-case analysis for joins with minesweeper. In *Principles of Database Systems (PODS)*, pages 234–245. ACM, 2014.
26. H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms. In *Principles of Database Systems (PODS)*, pages 37–48. ACM, 2012.
27. H. Q. Ngo, C. Ré, and A. Rudra. Skew strikes back: New developments in the theory of join algorithms. *arXiv preprint arXiv:1310.3314*, 2013.
28. D. Nguyen, M. Aref, M. Bravenboer, G. Kollias, H. Q. Ngo, C. Ré, and A. Rudra. Join processing for graph patterns: An old dog with new tricks. In *GRADES*, page 2. ACM, 2015.
29. R. Ramakrishnan and J. Gehrke. *Database management systems*. McGraw Hill, 2000.
30. B. B. Thompson, M. Personick, and M. Cutcher. The Bigdata®RDF Graph Database. In *Linked Data Management.*, pages 193–237. 2014.
31. T. L. Veldhuizen. Leapfrog Triejoin: A simple, worst-case optimal join algorithm. In *ICDT*, pages 96–106, 2014.
32. D. Vrandečić and M. Krötzsch. Wikidata: A free collaborative knowledgebase. *Comm. ACM*, 57:78–85, 2014.
33. C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.

A Proof of Theorem 1

Fix a basic graph pattern P, an RDF graph G, and $?\mathtt{x}_1, \ldots, ?\mathtt{x}_n$ the chosen variable order. Further assume that the computation of $\mathrm{LF}_G(P', ?\mathtt{x})$ takes time at most $\min_{t \in P' : ?\mathtt{x} \in \mathrm{var}(t)} |\pi_{?\mathtt{x}}(\llbracket t \rrbracket_G)| \cdot \log(|G|)$ for every basic graph pattern P' (for simplicity, we will omit the trivial empty case where there exists $t \in P'$ such that $?\mathtt{x} \in \mathrm{var}(t)$ and $|\pi_{?\mathtt{x}}(\llbracket t \rrbracket_G)| = 0$ since the time taken will be simply $\log(|G|)$). Finally, for every RDF graph G' we will say that G' is of size less than G whenever $|\llbracket t \rrbracket_{G_i}| \leqslant |\llbracket t \rrbracket_G|$ for all $t \in P$ (recall that P is fixed).

The proof of Theorem 1 goes in two steps. First, we will bound the time of Leapfrog Join by bounding the time T_i of each for-loop $?\mathtt{x}_i$ of Algorithm 1. Then for each level $?\mathtt{x}_i$ we define a new RDF graph G_i of size less than G such that $T_i = |\llbracket P \rrbracket_{G_i}| \leqslant 2^{\mathrm{MIN}(P,G)}$. The proof will follow by taking the sum over all T_i.

Fix a variable $?\mathtt{x}_i$ and denote by $\bar{x}_{i-1} = ?\mathtt{x}_1, \ldots, ?\mathtt{x}_{i-1}$ the order of variables before $?\mathtt{x}_i$ (for the sake of simplification, in the sequel we consider $\bar{x}_i$ also as a

set). We start by bounding the time of the for-loop in Algorithm 1 corresponding to $?\mathtt{x}_i$. For this, consider the following extension of LF_G over $\bar{x}_{i-1}$:

$$\mathrm{LF}_G(P, \bar{x}_{i-1}) \;=\; \{\mu \mid \mathrm{dom}(\mu) = \bar{x}_{i-1} \text{ and } [\![\mu(t)]\!]_G \neq \varnothing \text{ for all } t \in P\}$$

Clearly, the number of times that the for-loop of $?\mathtt{x}_i$ will be called is given by $|\,\mathrm{LF}_G(P, \bar{x}_{i-1})|$. Then for each $\mu \in \mathrm{LF}_G(P, \bar{x}_{i-1})$ the Leapfrog procedure $\mathrm{LF}_G(\mu(P), ?\mathtt{x}_i)$ is called taking time at most $\min_{t \in \mu(P) : ?\mathtt{x}_i \in \mathrm{var}(t)} |\pi_{?\mathtt{x}_i}([\![t]\!]_G)|$ (omitting the $\log(|G|)$ factor for the moment). If we call T_i the number of steps that Algorithm 1 spends in the for-loop of $?\mathtt{x}_i$, we have that:

$$T_i \;=\; \sum_{\mu \in \mathrm{LF}_G(P, \bar{x}_{i-1})} \; \min_{t \in \mu(P) : ?\mathtt{x}_i \in \mathrm{var}(t)} |\pi_{?\mathtt{x}_i}([\![t]\!]_G)|$$

One can easily check that the total time of Algorithm 1 is given by $(\sum_{i=1}^{n} T_i) \cdot \log(G)$. Therefore, if we bound T_i by the AGM bound of P and G, then the worst case optimality of Leapfrog Join will be proven (recall that our analysis is in data complexity, omitting factors that depend on the size of P).

To bound the size of T_i, we build an RDF graph G_i such that $T_i = |[\![P]\!]_{G_i}|$ and the size of G_i is less that the size of G. Let $\perp$ be a dummy value. To build G_i define the set of mappings U_i such that $\mu \in U_i$ if and only if there exists $\mu' \in \mathrm{LF}_G(P, \bar{x}_{i-1})$ such that:

1. $\mu(?\mathtt{x}) = \mu'(?\mathtt{x})$ for every $?\mathtt{x} \in \bar{x}_{i-1}$,
2. $1 \leqslant \mu(?\mathtt{x}_i) \leqslant \min_{t \in \mu'(P) : ?\mathtt{x}_i \in \mathrm{var}(t)} |\pi_{?\mathtt{x}_i}([\![t]\!]_G)|$, and
3. $\mu(?\mathtt{x}) = \perp$ for every $?\mathtt{x} \in \{?\mathtt{x}_{i+1}, \ldots, ?\mathtt{x}_n\}$.

In other words, U_i contains all mappings built from mappings of $\mathrm{LF}_G(P, \bar{x}_{i-1})$ and extended by assigning to $?\mathtt{x}_i$ any value less than the time for computing $\mathrm{LF}_G(\mu'(P), ?\mathtt{x}_i)$. From U_i we can build the RDF graph G_i as follows:

$$G_i \;=\; \bigcup_{\mu \in U_i} \mu(P).$$

By construction, note that the size of G_i is less than the size of G. Furthermore, we have that $T_i = |[\![P]\!]_{G_i}|$. Indeed, for each $\mu' \in \mathrm{LF}_G(P, \bar{x}_{i-1})$ we will have $\big(\min_{t \in \mu'(P) : ?\mathtt{x}_i \in \mathrm{var}(t)} |\pi_{?\mathtt{x}_i}([\![t]\!]_G)|\big)$ different mappings in $[\![P]\!]_{G_i}$ and vice versa.

To finish the proof, recall the linear program associated to P and G, and its minimum value $\mathrm{MIN}(P, G)$. Consider also the same linear program but now for P and G_i. Given that G_i is of size less than G, then the minimization function associated to the linear program of P and G_i always satisfies:

$$\sum_{t \in P} x_t \cdot \log(|[\![t]\!]_{G_i}|) \leqslant \sum_{t \in P} x_t \cdot \log(|[\![t]\!]_G|).$$

Therefore, we can conclude that $\mathrm{MIN}(P, G_i) \leqslant \mathrm{MIN}(P, G)$ and thus:

$$T_i \;=\; |[\![P]\!]_{G_i}| \;\leqslant\; 2^{\mathrm{MIN}(P, G_i)} \;\leqslant\; 2^{\mathrm{MIN}(P, G)}$$

where the second inequality follows by the AGM bound. Given that each T_i is bounded by $2^{\mathrm{MIN}(P,G)}$ we conclude that the overall time is bounded by $n \cdot 2^{\mathrm{MIN}(P,G)} \cdot \log(G)$ and that Leapfrog Join is worst-case optimal.