A Worst-Case Optimal Join Algorithm for SPARQL

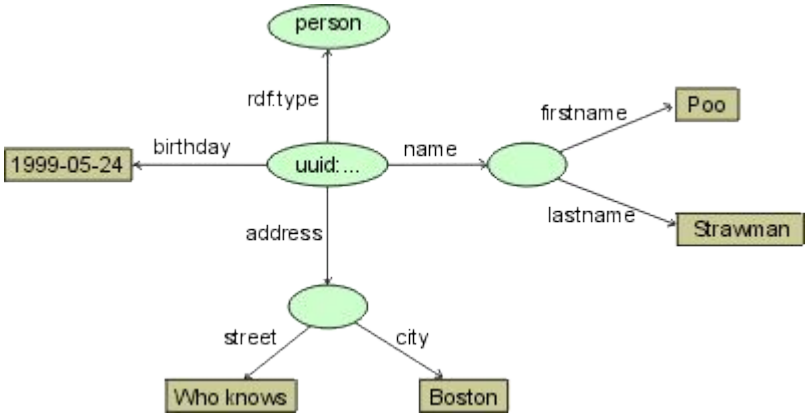
Integrantes:

- Gabriel Aguirre
- Benjamín Farías
- Juan Hernández
- Benjamín Lepe

Conceptos Preliminares - RDF

¿Qué es **RDF** (*Resource Description Framework*)?

- Modelo de datos que funciona de forma conceptual, utilizado para recursos de la web.
- Declaraciones en formato: **sujeto**, **predicado**, **objeto (S-P-O).**
- **RDF terms** son *Blank Nodes*, *Literals* y *IRIs.*



Conceptos Preliminares - SPARQL

¿Qué es **SPARQL**?

- Lenguaje de consultas utilizado en bases de datos de tipo **RDF**.
- Se basa en conjuntos de ***triple patterns*** que tienen la misma forma que los modelos **RDF**. Estos conjuntos se denominan ***basic graph patterns***.
- Ampliamente utilizado para obtener recursos desde la web.

```
PREFIX  dc: <http://purl.org/dc/elements/1.1/>
SELECT  ?title
WHERE   { <http://ejemplo.org/libros> dc:title ?title }
```

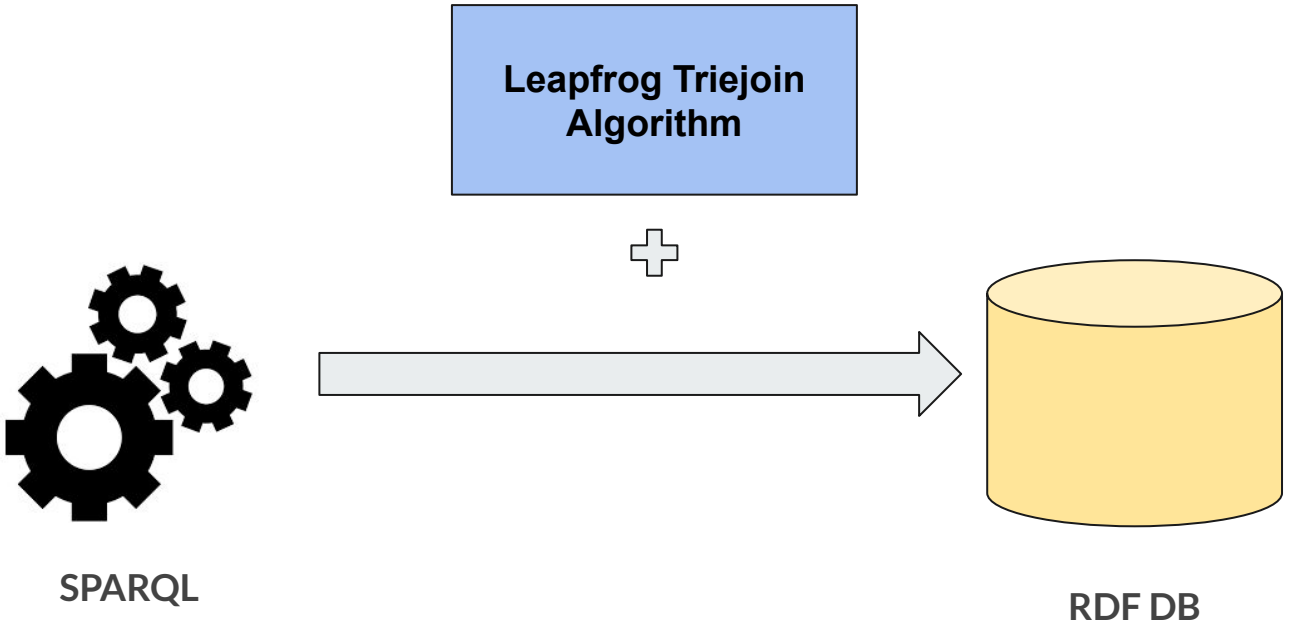
Conceptos Preliminares - SPARQL Semantics

Mapping Function: $\mu : \mathbf{V} \rightarrow \mathbf{IBL}$

Compatible Mappings: $\mu_1 \sim \mu_2 \leftrightarrow \mu_1(?x) = \mu_2(?x) \quad \forall ?x \in dom(\mu_2) \cap dom(\mu_1)$

Mapping Sets Join: $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \text{ and } \mu_1 \sim \mu_2\}$

Idea Principal



Trabajo Relacionado

- **Indexing**: SPARQL permite indexar *triples* de manera eficiente. Uso de *Nested Data Structures* o *B+Trees*.

Trabajo Relacionado

- **Indexing**: SPARQL permite indexar *triples* de manera eficiente. Uso de *Nested Data Structures* o *B+Trees*.
- **Pairwise Joins:** SPARQL normalmente utiliza variantes de algoritmos de join utilizados en bases de datos relacionales, como *nested-loop joins* o *hash joins*.

Trabajo Relacionado

- **Indexing**: SPARQL permite indexar *triples* de manera eficiente. Uso de *Nested Data Structures* o *B+Trees*.
- **Pairwise Joins:** SPARQL normalmente utiliza variantes de algoritmos de join utilizados en bases de datos relacionales, como *nested-loop joins* o *hash joins*.
- **Multiway Joins**: SMJoin es un algoritmo creado para SPARQL que utiliza multiway joins, sin embargo, pairwise joins son más eficientes en ciertas consultas.

Trabajo Relacionado

- **Indexing**: SPARQL permite indexar *triples* de manera eficiente. Uso de *Nested Data Structures* o *B+Trees*.
- **Pairwise Joins:** SPARQL normalmente utiliza variantes de algoritmos de join utilizados en bases de datos relacionales, como *nested-loop joins* o *hash joins*.
- **Multiway Joins**: SMJoin es un algoritmo creado para SPARQL que utiliza multiway joins, sin embargo, pairwise joins son más eficientes en ciertas consultas.
- **Worst-Case Optimal Joins:** Se han aplicado en diversas áreas incluyendo grafos, pero ningún trabajo se había enfocado en aplicarlo sobre SPARQL.

Leapfrog Join for Basic Graph Patterns (LFJ)

Se divide en **dos fases** principales:

- **Leapfrog:** Evalúa **una** variable.
- **Eliminación de Variables:** Evalúa **múltiples** variables.

Leapfrog

Calcular los ***non-trivial outputs*** de una sola variable:

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

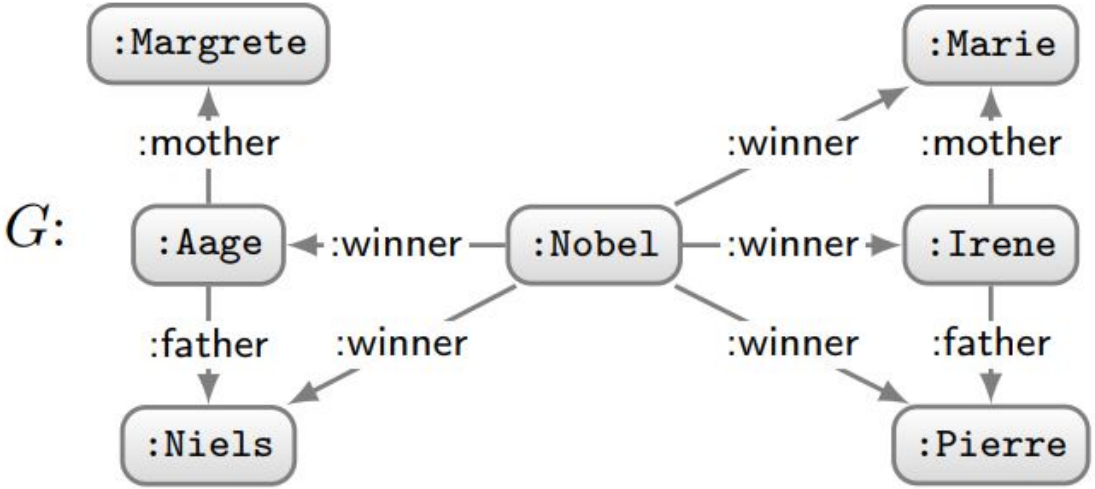
Eliminación de Variables

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     μ_0 ← ∅
3     foreach μ ∈ LF_G(μ_0(P), ?x_1) do
4         μ_1 ← μ_0 ∪ μ
5         foreach μ ∈ LF_G(μ_1(P), ?x_2) do
6             μ_2 ← μ_1 ∪ μ
7             ⋱
8                 foreach μ ∈ LF_G(μ_{n-1}(P), ?x_n) do
9                     Output μ_{n-1} ∪ μ    // write to output and continue
```

Ejemplo del Algoritmo LFJ



$$P: \begin{array}{l} (?x_1, :\text{winner}, ?x_2) \\ (?x_1, :\text{winner}, ?x_3) \\ (?x_1, :\text{winner}, ?x_4) \\ (?x_2, :\text{father}, ?x_3) \\ (?x_2, :\text{mother}, ?x_4) \end{array}$$

Operador Físico - Implementación

Framework:

- Motor de base de datos para RDF
- Utiliza *Nested-Loop Joins* sobre índices *B+Tree*

Implementación de Leapfrog Join (LFJ):

- Índices
- Orden de Variables
- Optimización

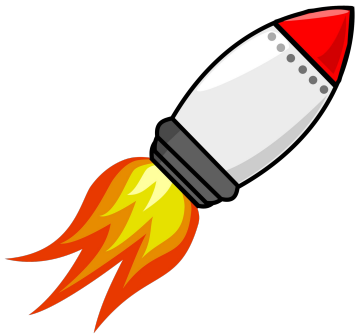
Operador Físico - Índices

Condición para que LFJ sea Worst-Case Optimal:

Theorem 1. *An implementation of Leapfrog Join is worst-case optimal if, for every RDF graph G, basic graph pattern P, and variable* ?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* ?x.

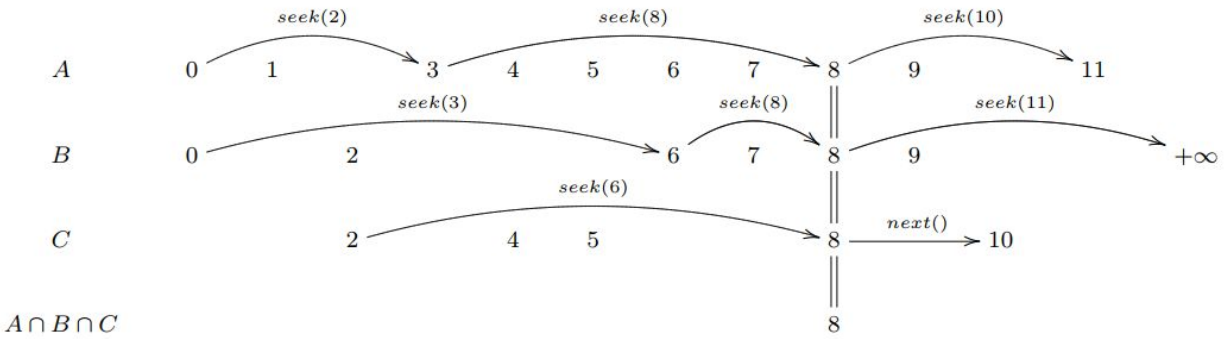


Operador Físico - Índices

Triple Patterns: (1, 1, ?x), (2, ?x, 0)

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

El mismo algoritmo de intersección utilizado por **Leapfrog Triejoin!**



Operador Físico - Índices

RDF (S: Sujeto, P: Predicado, O: Objeto)

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

Cambiar los *Tries* por **B+Trees:**

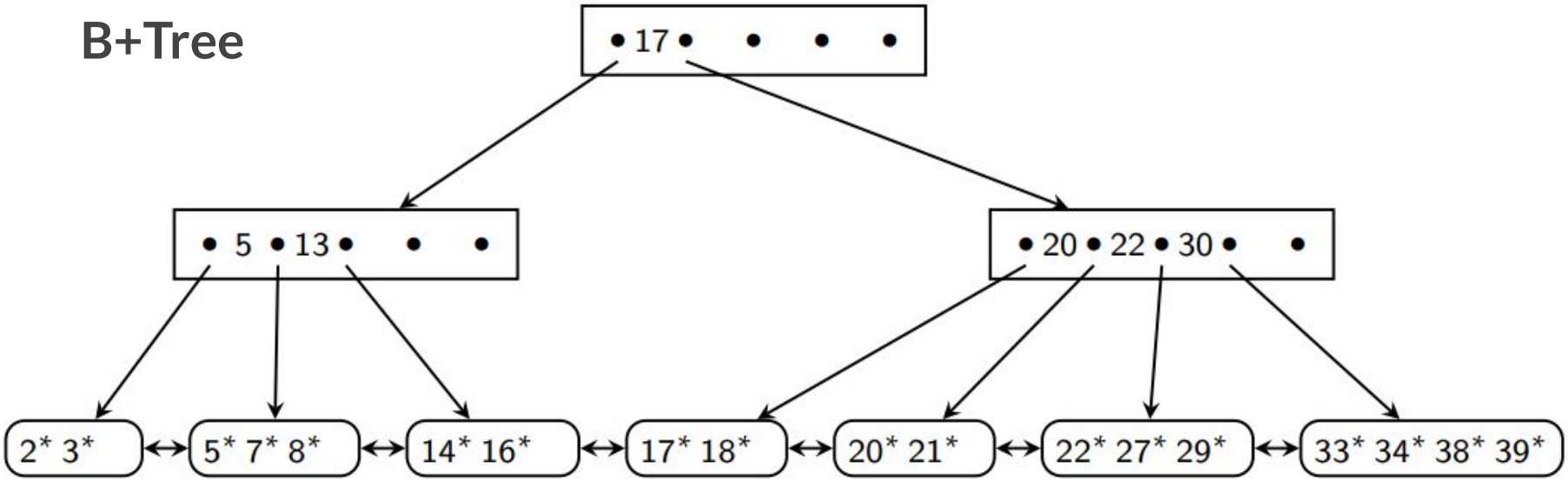
- Jena trae 3 árboles por default: **SPO, POS y OSP.**
- Se añaden 3 más: **SOP, PSO, OPS.**

Estos **6 árboles B+Tree** conforman el **nuevo índice** para LFJ!

- Recorrer los árboles de forma *top-down* + almacenar ubicación del **último resultado**
- En el peor caso toma tiempo **logarítmico**

Operador Físico - Índices

B+Tree



Operador Físico - Índices

Condición para que LFJ sea Worst-Case Optimal:

Theorem 1. *An implementation of Leapfrog Join is worst-case optimal if, for every RDF graph G, basic graph pattern P, and variable* ?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* ?x.

El nuevo Leapfrog Join es Worst-Case Optimal!

Operador Físico - Orden de Variables

- El **orden de eliminación de las variables** determina la **eficiencia** del algoritmo
- Buscamos **filtrar** los resultados lo antes posible

Orden Propuesto:

- Dado un orden de los *triple patterns* entregado por **Jena,** por ejemplo:

$$O_{\text{trip}} = (?z, :p3, ?u), (?x, :p2, ?z), (?x, :p1, ?y)$$

- Elegimos primero las **variables de join** y posteriormente las **variables solitarias:**

$$O_{\text{var}} = ?z, ?x, ?u, ?y$$

Operador Físico - Optimización

- Las variables **solitarias** sólo dependen de su propio *triple pattern*!
- **Es poco eficiente explorarlas utilizando el LFJ anidado**

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     μ_0 ← ∅
3     foreach μ ∈ LF_G(μ_0(P), ?x_1) do
4         μ_1 ← μ_0 ∪ μ
5         foreach μ ∈ LF_G(μ_1(P), ?x_2) do
6             μ_2 ← μ_1 ∪ μ
7             ⋱
8                 foreach μ ∈ LF_G(μ_{n-1}(P), ?x_n) do
9                     Output μ_{n-1} ∪ μ    // write to output and continue
```

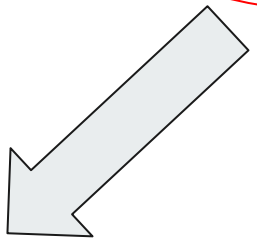
Operador Físico - Optimización

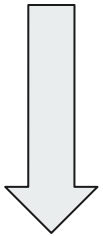
Mejora:

Var. de Join

Var. Solitarias

$$O_{\text{var}} = ?x_1, \ldots, ?x_m, ?x_{m+1}, \ldots ?x_n$$





Leapfrog Join Anidado

Producto Cartesiano

Experimentos - Motores de Prueba

- Apache Jena con LFJ
- Apache Jena sin LFJ [5]
- Virtuoso [3]
- Blazegraph [4]

Experimentos - Sets de Prueba

- Berlin SPARQL Benchmark (BSBM):
  - 10 500 queries
  - 30 predicados distintos
- WatDiv Benchmark:
  - 1 000 queries
  - 85 predicados distintos
  - 20 patterns distintos
  - Sin patrones cíclicos
- Wikidata Graph Pattern Benchmark (WGPB):
  - 850 queries
  - 2 101 predicados distintos
  - 9 patrones con resultados de única variable
  - 8 patrones con resultados de múltiples variables

Resultados - BSBM

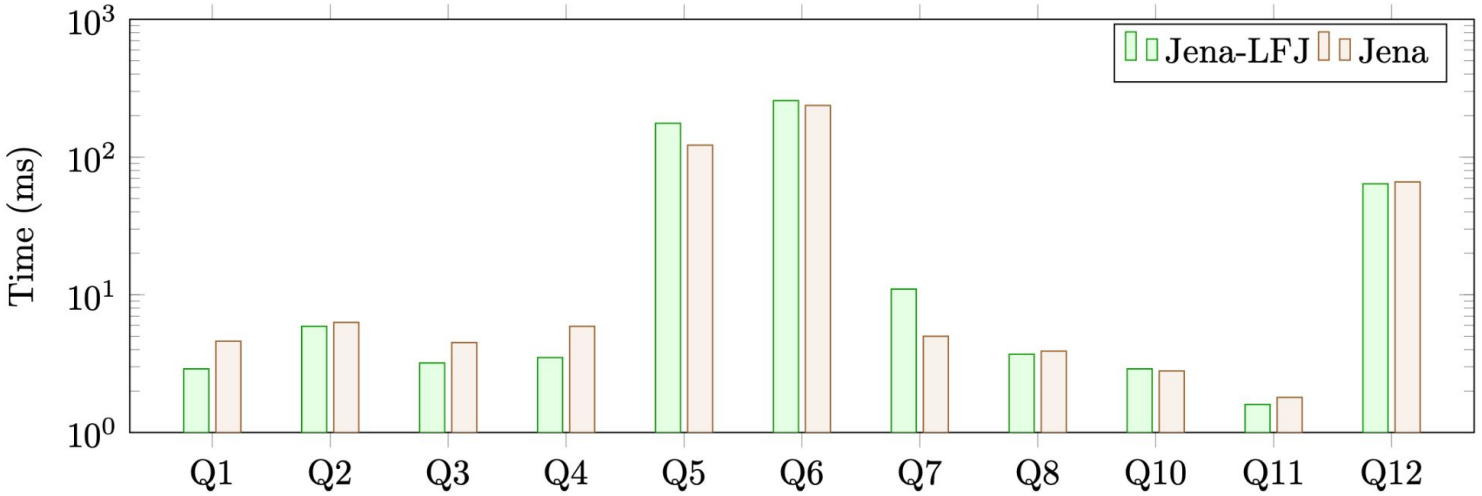


Fig. 6.2. Plot of runtimes for queries of the Berlin Benchmark with log y-axis.

Resultados - WatDiv Benchmark

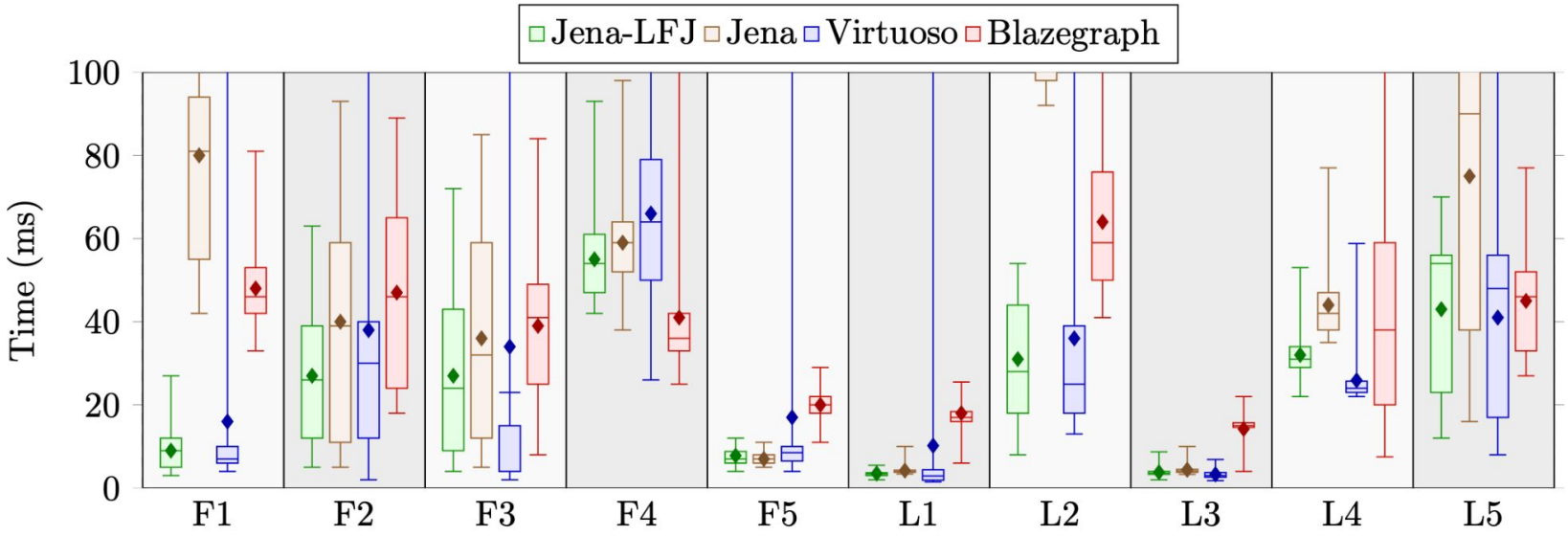


Fig. 6.3. Box plots of runtimes for queries L and F of the WatDiv Benchmark.

Resultados - WatDiv Benchmark

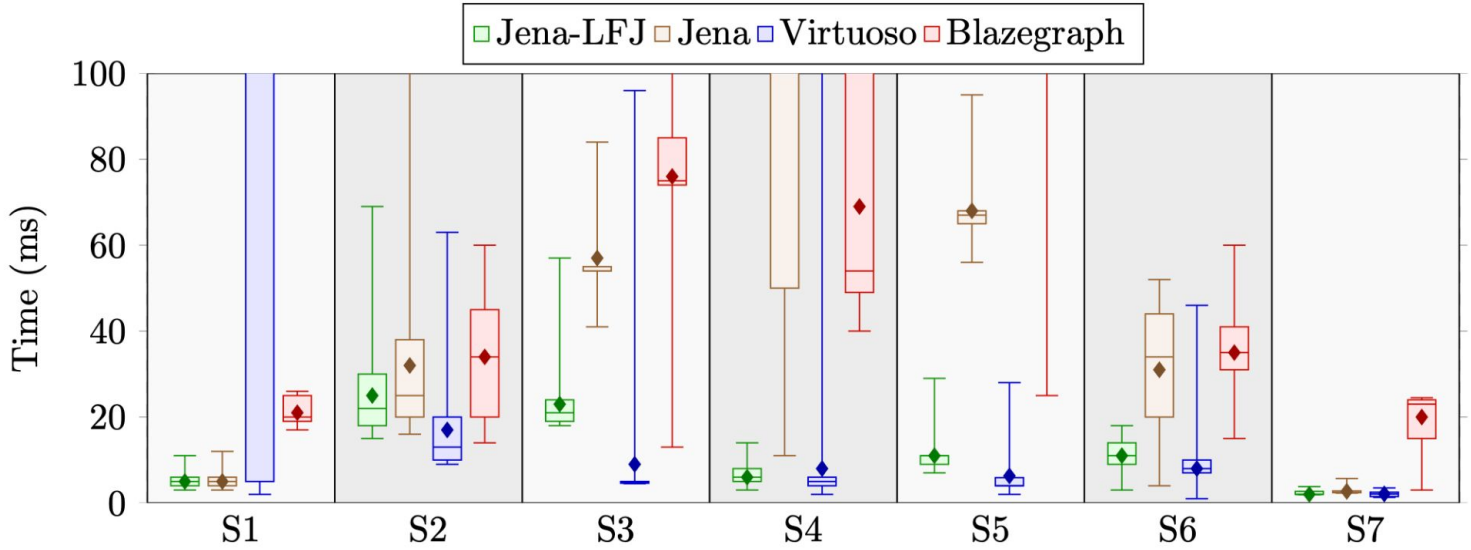


Fig. 6.4. Box plots of runtimes for queries S of the WatDiv Benchmark

Resultados - WGPB

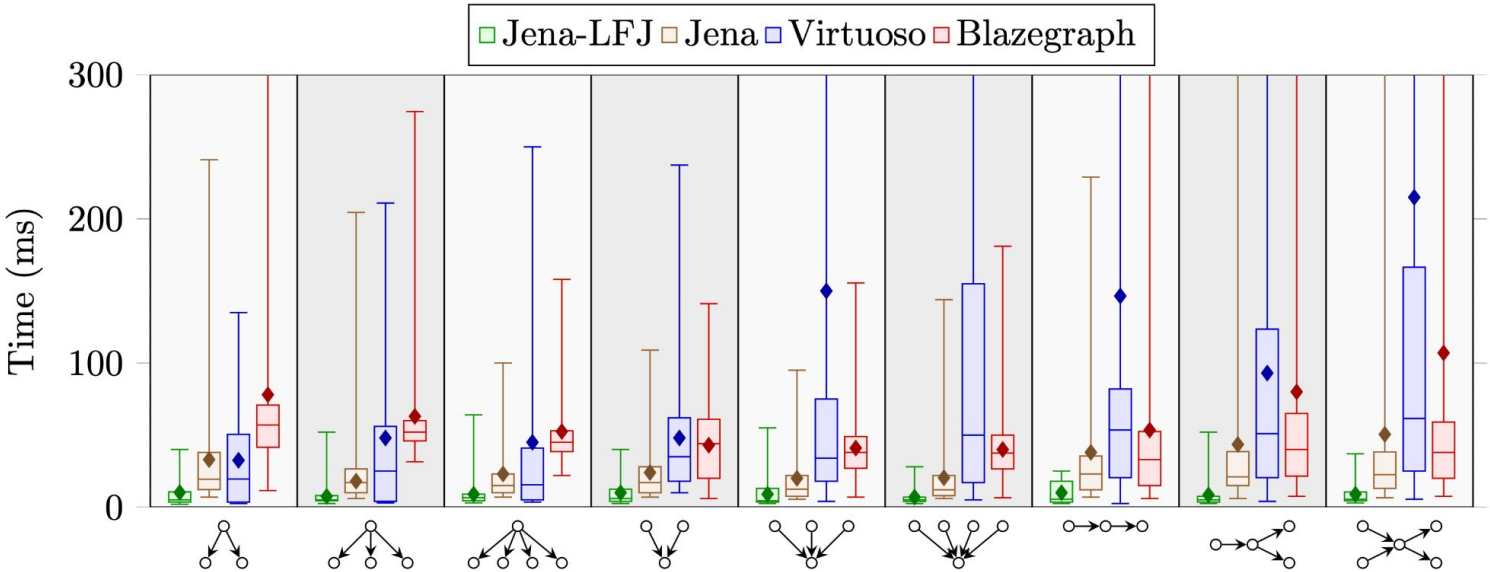


Fig. 6.6. Box plots of runtimes for queries with a single join variable

Resultados - WGPB

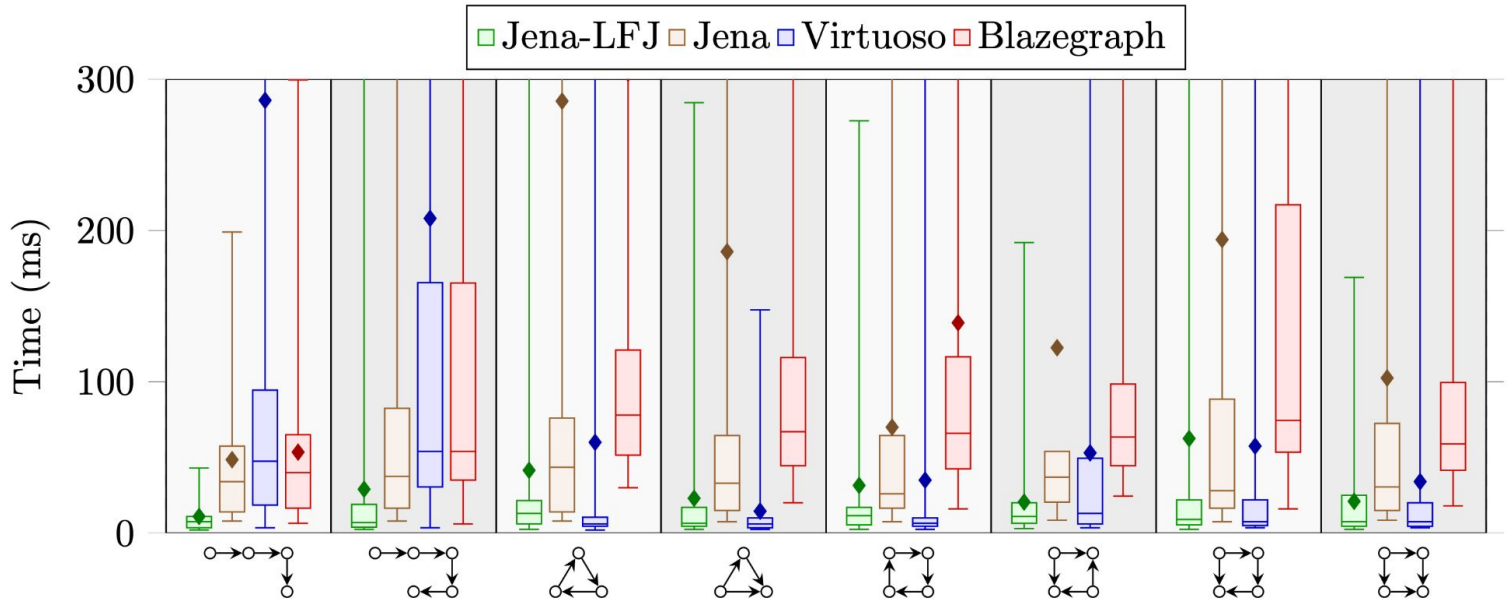


Fig. 6.7. Box plots of runtimes for queries with multiple join variables

Conclusiones

- Primera implementación de un algoritmo *worst-case optimal* en el contexto de SPARQL.
- Los resultados deben ser considerados puntos de partida en el área.
- Se identifican 3 líneas de posibles trabajos futuros:
  - Potenciales beneficios de otros algoritmos WCO sobre SPARQL
  - Formas efectivas de elegir el orden de variables
  - Optimizaciones de otros operadores de SPARQL (ej: Property Paths)

Bibliografía

[1] Hogan, Riveros, Rojas, Soto. (2019). *A Worst-Case Optimal Join Algorithm for SPARQL*.

[2] Veldhuizen. (2012). *Leapfrog Triejoin: A Simple, Worst-Case Optimal Join Algorithm*.

[3] Erling, Mikhailov. (2009). *RDF Support in the Virtuoso DBMS*.

[4] Thompson, Personick, Cutcher. (2014). *The Bigdata® RDF Graph Database*.

[5] Apache Jena. https://jena.apache.org/. Accessed on 2018-12-30.