Love-at-First-Sight: First Answers Without the Awkward Silence in Big Knowledge Graphs

Giannis Vassiliou
ECE, HMU
Heraklion, Greece
vassil@hmu.gr

Haridimos Kondylakis
CSD, UOC & FORTH-ICS
Heraklion, Greece
kondylak@ics.forth.gr

ABSTRACT

The increasing number of large knowledge graphs (KGs) now available online requires methods for their efficient exploration. Most of these KGs offer online SPARQL endpoints for querying and exploring their data. In a typical scenario, the users issue coarse, exploratory queries at the beginning, refining them further in the sequel in order to find the answer to the question in mind. However, those coarse exploratory queries are costly to evaluate as they usually involve many results and take too much time to be answered, or even worse, they time out, limiting the exploration potential of the data they expose. In this paper, we present the LFS (Love-at-First-Sight) system, offering a unique solution to the aforementioned problem, enabling users to efficiently get the first answers to their queries. More specifically, we are the first to define the problem of constructing first-sight summaries (FSS), i.e., summaries able to provide rapidly, first answers to user queries, relying on existing query logs. We provide effective algorithms for constructing both exact and approximate FSS under budget constraints with theoretical guarantees. We analytically and experimentally demonstrate latency reductions of up to two orders of magnitude over SPARQL endpoints and one order of magnitude over relevant baselines.

PVLDB Reference Format:
Giannis Vassiliou and Haridimos Kondylakis. Love-at-First-Sight: First Answers Without the Awkward Silence in Big Knowledge Graphs. PVLDB, 19(7): 1544 - 1557, 2026.
doi:10.14778/3801059.3801068

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/giannisvassiliou/LFS-2025N-main.

1 INTRODUCTION

Knowledge Graphs (KGs) are vast collections of interconnected entities used extensively for data integration and exploration [8]. As users increasingly leverage SPARQL endpoints to harvest information [8], efficiency is critical. In a typical usage scenario of a SPARQL endpoint, a user queries the data and gradually refines her query until the desired result is obtained [8].

The problem. In the aforementioned typical scenario, in the first coarse, exploratory phase, the returned results are usually too many, and as such, the users have to wait for a significant amount of time before they see the first results. Public endpoints, such as DBpedia or Wikidata, in order to prevent long-running queries that require a lot of CPU and memory resources, try to ensure stable and responsible services to the user community by setting up quotas on the execution time and arrival rate [2]. This certainly limits the exploration potential of the information that the SPARQL endpoints expose. This first-answer latency problem is especially harmful in interactive settings such as iterative query refinement, debugging query patterns, or exploratory analytics, where users rely on early examples to validate query intent and decide whether to continue, abort, or reformulate their query.

Existing approaches. To mitigate long delays at online SPARQL endpoints, users often rely on the `LIMIT` clause. However, `LIMIT` only restricts the number of returned results and typically does not reduce the amount of data processed during query evaluation, as the engine must still scan the full dataset to identify matches.

For substantial performance improvements, recent research on the area focuses on restricted SPARQL servers [6, 30, 40], which ensure termination by fragmenting query execution or paginating results. These methods, however, require intelligent clients and incur additional overhead due to client-side processing. Progressive query processing [7], on the other hand, returns results incrementally but provides no guarantees on the number or availability of early answers. Another prominent approach focusing on improving the efficiency of query answering is view selection and materialization. Given a KG and a query workload, materialize an appropriate set of views to improve query efficiency such that the views fit into a pre-specified storage constraint. However, the problem of selecting the appropriate views to materialize, *is hard* [27], and does not scale for workloads containing thousands of queries. Semantic summarization, on the other hand, extracts compact representations of large KGs for tasks such as visualization, exploration, and query answering [9, 32]. Workload-based summaries further identify frequently accessed graph regions [38, 39], but have not been designed to support rapid retrieval of the first few query answers.

LFS. Our system, Love-at-First-Sight (LFS), combines ideas from semantic summarization and view materialization to construct a compact subgraph of the original KG that enables rapid retrieval of the first answers to SPARQL queries.

To rapidly return the first few answers for a specific user query, a small, relevant part of the original graph should be extracted and used for query answering. However, for returning the first few answers to *any user query*, in essence, the entire graph should be available. Already, this shows the conflicting properties that such a summary should have and the difficulties entailed in optimally selecting the proper part of the graph to extract and use.

Proceedings of the VLDB Endowment, Vol. 19, No. 7 ISSN 2150-8097.
doi:10.14778/3801059.3801068

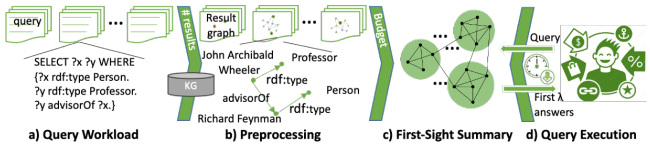

Figure 1: LFS main ideas.

LFS addresses this challenge by exploiting user query logs: each valid query is transformed into a CONSTRUCT query, and the resulting answer-induced subgraphs are merged into a compact summary, as illustrated in Figure 1. An optional budget constraint retains only the most informative triples, yielding a first-sight summary (FSS) that can be queried directly to retrieve the first λ answers without scanning the full KG. This approach scales to large workloads while keeping the summary compact (at most 205 MB), supports incremental updates as new queries arrive, and generalizes well to future queries by trading completeness for efficiency. Overall, our contributions within the LFS system, are the following:

- We introduce the problem of constructing first-sight summaries (FSS), compact subgraphs that enable rapid retrieval of the first query answers and satisfy modularity and monotonicity.
- We propose an efficient log-driven algorithm for FSS construction with linear complexity in the workload size and low memory footprint, allowing summaries to reside in main memory.
- We propose a budgeted greedy algorithm for approximate FSS with quality guarantees under space constraints.
- We show FSS is naturally incremental and propose a budget-constrained incremental algorithm for summaries.
- We experimentally evaluate LFS on YAGO, DBpedia, and Wikidata, demonstrating compact summaries and speedups of up to 210× over SPARQL endpoints for known queries.
- We further show that FSS generalize to unseen queries, covering up to 93% of the workload and achieving order-of-magnitude latency reductions over existing baselines.

To the best of our knowledge, we are the first to define the problem of constructing a first-sight summary for knowledge graphs and to provide effective, incremental algorithms for their construction. While our approach is inspired by classical materialized view selection and semantic summarization, LFS targets a fundamentally different optimization objective: minimizing first-answer latency. Unlike traditional view selection, which aims to reduce the total evaluation cost of queries, LFS focuses specifically on the earliest obtainable answers and on constructing lightweight, answer-centric mini-graphs. Existing SPARQL view-selection methods do not address (i) early-answer semantics, (ii) strict memory budgets requiring partial graph materialization, or (iii) operation under real-world query-log workloads at scale.

2 FIRST-SIGHT SUMMARIES

Preliminaries. We consider RDF Knowledge Graphs (KGs), the W3C standard for representing structured data on the Web. An RDF KG G is a set of triples (s, p, o), where a subject s is connected to an object o via a property p. Let $\mathcal{U}$, $\mathcal{L}$, and $\mathcal{B}$ denote the sets of URIs, literals, and blank nodes, respectively, which are pairwise disjoint. Blank nodes represent anonymous or unknown resources. Let $T = \mathcal{U} \cup \mathcal{B} \cup \mathcal{L}$ be the set of RDF terms. An RDF triple belongs to $(\mathcal{U} \cup \mathcal{B}) \times \mathcal{U} \times T$. Further, let V be a set of variables. A solution mapping is a partial function $\mu : V \to T$, with domain $dom(\mu) \subseteq V$.

For querying, we use SPARQL. SPARQL [1] is the W3C standard query language for RDF and follows a graph-matching semantics. A SPARQL query Q specifies a graph pattern P that is matched against an RDF graph G. The basic building block is a triple pattern, and a set of triple patterns forms a basic graph pattern (BGP). The evaluation of a BGP Q over G, denoted $Q(G)$, yields a set of solution mappings. We write $|Q(G)|$ for the number of answers and $|G|$ for the number of triples in G. SPARQL also provides operators such as *OPTIONAL*, *FILTER*, and *EXISTS*. To simplify presentation, we focus on BGPs in our formal development; however, our approach applies to the full monotonic fragment of SPARQL [20], excluding aggregates. This choice is empirically well justified: in the query logs used in our evaluation (Section 3), aggregate queries account for only 0.9%, 1.3%, and 8% of the queries over YAGO, DBpedia, and Wikidata, respectively, while non-monotonic queries account for 0.2%, 1.3%, and 6.1%.

High-Level Architecture. Figure 2 depicts how the LFS server can be used in practice and the components of the LFS engine. Users visit the corresponding SPARQL endpoint and can select either to retrieve the complete answers or the first few answers to their queries. In the online part, in case the users select the complete answers, their queries are forwarded to be answered by the corresponding traditional SPARQL endpoint of the KG. Otherwise, users select to trade efficiency for completeness, and as such, their queries are forwarded to be rapidly answered by the corresponding FSS. In the offline part, query logs are used along with their corresponding SPARQL endpoint in order to construct the first-sight summaries. As more queries are accumulated in the query logs, FSS can be incrementally updated offline to include the missing information.

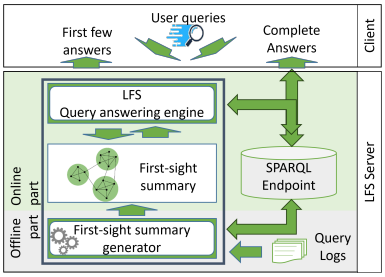

Figure 2: LFS high-level architecture.

Informal problem statement. We begin with an example in order to allow the readers to fully understand the complexity of the problem under discussion.

Example 1. Consider the KG in Figure 3, representing a university domain with persons and organizations. A user begins exploration by querying for advisors of *Richard Feynman*:

```
Q0: SELECT * WHERE {?x advisorOf ?y}.
```

Although the KG may contain many matching instances, the user primarily wants to verify that the query is valid and inspect initial results. If $Q0$ was known in advance, retaining only

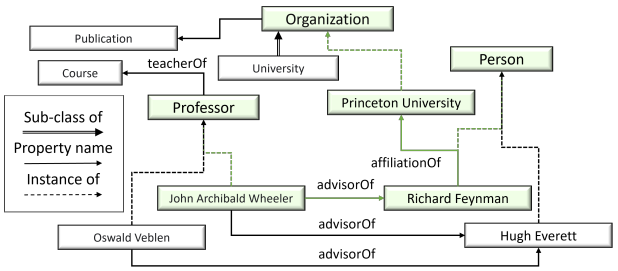

Figure 3: An example RDF KG and an FSS ($\lambda = 1$) for $Q1$.

the triple (*John Archibald Wheeler*, *advisorOf*, *Richard Feynman*) would suffice to return a first answer and enable further exploration, while additional answers could provide useful context.

Based on the aforementioned example, we are now ready to informally describe the problem of constructing a first-sight summary: *Given a KG G and a query workload $\mathcal{Q}$, construct a summary $S \in G$ that can be queried instead of G returning rapidly the first λ answers for all input queries $Q \in \mathcal{Q}$.*

Note that we are not interested in the top-k answers satisfying a selection criterion or maximizing a function. We are just interested in *any* λ results.

Summary Size vs Query Efficiency: This very first problem statement also makes it clear that the first-sight summary S should have the following properties:

- *Compactness.* Ideally, the first-sight summary should be able to provide the first few answers rapidly. In order to do this, it should contain minimal data that will enable efficient query answering.
- *Completeness.* On the other hand, it should contain minimal data for *all queries* in $\mathcal{Q}$. As the queries under consideration are numerous, in essence, a significant portion of the graph G should be kept in summary, which in turn leads to execution times similar to the standard execution time for querying the entire graph.

This, in essence, demonstrates that compactness and completeness are two conflicting properties that such a first-sight summary should possess, and the complexity entailed in deciding which triples from the original graph to keep as a summary. As we show in the sequel, as queries tend to search for similar information, achieving both is feasible.

Formal problem statement. We are now ready to formally define the problem of first-sight summary construction.

Definition 1 (First-Sight Summary)**.** Given a KG G, a query workload $\mathcal{Q}_n$ with n queries, and a number λ, a first-sight summary $FSS(\mathcal{Q}_n, G, \lambda)$, is a subgraph $FSS \subseteq G$ such that for every $Q \in \mathcal{Q}_n$: (i) $Q(FSS) \subseteq Q(G)$ and (ii) $|Q(FSS)| \geq min\{\lambda, |Q(G)|\}$

When n is irrelevant, we will use $\mathcal{Q}$ instead of $\mathcal{Q}_n$. Based on the aforementioned definition, we then could use FSS instead of G to return at least λ answers (when they exist) to any $Q \in \mathcal{Q}$. An FSS is not unique, as many different subsets of the original graph might exist with at least λ answers for each one of the queries in $\mathcal{Q}$.

It can be easily shown that there are always solutions of FSS for any set of arguments for which FSSs enjoy modularity and monotonicity.

Lemma 1 (Modularity). *Given a set of queries $\mathcal{Q}_n = \{Q_1, ..., Q_n\}$ targeting a knowledge graph G, and a number λ, there exist a solution $FSS(\mathcal{Q}_n, G, \lambda)$ such that:*

$$FSS(\mathcal{Q}_n, G, \lambda) \subseteq \bigcup_{i=1}^{n} FSS(Q_i, G, \lambda)$$

Proof. (sketch) As an FSS should contain at least λ answers for all queries in $\mathcal{Q}_n$, in essence, it should contain at least λ for each one of the $\{Q_1, ..., Q_n\}$. ■

Lemma 2 (Monotonicity).

$$|FSS(\mathcal{Q}_n, G, \lambda)| \leq |FSS(\mathcal{Q}_n, G, \lambda + 1)|$$

and also

$$|FSS(\mathcal{Q}_n, G, \lambda)| \leq |FSS(\mathcal{Q}_{n+1}, G, \lambda)|$$

Proof. $|FSS(\mathcal{Q}_n, G, \lambda)| \subseteq |\bigcup_{i=1}^{n} FSS(Q_i, G, \lambda)| \leq |\bigcup_{i=1}^{n} FSS(Q_i, G, \lambda + 1)| = |FSS(\mathcal{Q}_n, G, \lambda + 1)|$. Further, $|FSS(\mathcal{Q}_n, G, \lambda)| = |\bigcup_{i=1}^{n} FSS(Q_i, G, \lambda)| \leq |\bigcup_{i=1}^{n+1} FSS(Q_i, G, \lambda)| = |FSS(\mathcal{Q}_{n+1}, G, \lambda)|$ ■

In essence, λ is used in order to tune the size of the summary based on the number of answers that a user would like to be returned, given that the original graph has at least that number of answers. Obviously, the larger the λ, the larger the summary size will be, as it would need to store more triples.

2.1 The LFS algorithm

Next, we present an algorithm for constructing an FSS, capitalizing on the modularity property. The LFS algorithm (Algorithm 1) receives as input a KG G, a number λ denoting the number of results to be stored in the summary for each query, and a query log with BGP queries $\mathcal{Q}$. In addition, it receives the SPARQL endpoint SP of the knowledge graph. The algorithm initializes the feature-based semantic summary (FSS) as an empty set of triples (line 1). It then processes the query workload sequentially (lines 2–4). For each query Q, the triple patterns in the WHERE clause are extracted and rewritten into a CONSTRUCT query that retrieves the corresponding subgraph, including all involved variables, while limiting the number of returned answers to λ (line 3). The constructed query is executed against the SPARQL endpoint SP of the knowledge graph G, and the resulting triples are added to the summary (line 4). After all queries have been processed, the algorithm returns the final FSS.

Algorithm 1 LFS

Input: Queries $\mathcal{Q}$; limit λ; Endpoint SP.
Output: Summary S.

1: $S \leftarrow \emptyset$
2: **for all** $Q \in \mathcal{Q}$ **do**
3: $\quad Q' \leftarrow \texttt{construct}(Q, \lambda)$ ▷ Extract BGP and apply limit
4: $\quad S \leftarrow S \cup \texttt{execute}(Q', SP)$
5: **return** S

The following theorem holds, proved directly by construction:

Theorem 1 (Correctness)**.** The LFS algorithm constructs a first-sight summary for $G, \mathcal{Q}$ and λ.

The generated FSS enjoys *completeness* in the sense of Definition 1. For every query in $\mathcal{Q}$, the FSS contains all triples belonging to the mini-graphs of the first λ answers retrieved during construction. Further, it enjoys partial *compactness* as each individual query introduces at most λ results in the summary graph. It is partial, as queries are treated independently, and eventually there might be more than λ answers in the summary for a specific query, as allowed also by Definition 1. We will focus more on the notion of compactness in Section 2.2, respecting budget constraints and providing guarantees on the size of the result.

Example 2. Assume a query log over the KG of Figure 3 containing the following queries:

```
Q1. SELECT ?x WHERE { ?y rdf:type Professor.
      ?x rdf:type Person. ?y advisorOf ?x.}
Q2. SELECT ?y WHERE { ?y rdf:type Organization.
      ?x rdf:type Person. ?y affiliationOf ?x.}
```

Let LFS be executed with $\lambda = 1$. For each query, LFS extracts the `WHERE` clause, issues a corresponding `CONSTRUCT` query with `LIMIT 1`, and adds the resulting answer-induced mini-graph to the summary. For $Q1$, this yields the triples $(John\ Arch.\ Wheeler, rdf : type, Professor)$, $(Richard\ Feynman, rdf : type, Person)$, and $(John\ Arch.\ Wheeler, advisorOf, Richard\ Feynman)$. For $Q2$, the summary is extended with $(Princ.\ Univ., rdf : type, Organization)$, $(Richard\ Feynman, affiliationOf, Princ.\ Univ.)$, and the shared type triple for *Richard Feynman*. The resulting first-sight summary for $\mathcal{Q} = \{Q1, Q2\}$ and $\lambda = 1$ is the union of these six triples, shown in light green in Figure 3.

Due to the modularity property, the algorithm is highly parallelizable, as queries can be evaluated in parallel in order to construct the FSS. However, we leave the exploration of the parallelization aspect of our algorithm for future work.

Incremental Maintenance. Furthermore, the first-sight summary (FSS) can be efficiently maintained over time as new queries arrive. Thanks to the modularity of FSS construction (Lemma 1), each query contributes an independent mini-graph to the summary, enabling incremental updates without requiring reconstruction from scratch. When a new query appears in the log, LFS extracts its WHERE clause, issues the corresponding CONSTRUCT query, retrieves up to λ results, and simply appends the resulting triples to the existing summary. This preserves correctness by construction and increases the summary size monotonically, consistent with Lemma 2. Section 3.3 empirically confirms the efficiency of this incremental behavior: as more queries are incorporated, both coverage and first-answer latency improve smoothly.

Complexity. To identify the complexity of the algorithm, we should first identify the complexity of its components. The algorithm, in essence, should visit each query once, extract the where clause, and execute the corresponding query. As query answering of BGP queries is NP-complete, the problem remains in that complexity class. However, we have available really good query engines able to provide answers to the corresponding queries in a reasonable time (offline and without users waiting online), and our algorithm is linear to the number of queries in the query log that are needed for constructing the FSS.

Summary Size. Similarly, we can estimate the size of an FSS in terms of triples, as in essence, the LFS algorithm introduces for each one of the n queries in $\mathcal{Q}_n$ at worst λ times the mini-graph induced by the triple patterns. Assuming that each query has T_a triple patterns at most, then the size of the FSS is at most $(n \times \lambda \times T_a)$ triples. Of course, as many queries might include the same triple, and most of the queries usually include less than 5 triples, the summary size is expected to be relatively small, which is also confirmed from our experiments (refer to Section 3.1).

2.2 The LFS-Budget-Greedy algorithm

To address the theoretical unbounded growth of n, we introduce a hard budget B. Although this may yield an *approximate FSS*, our strategy ensures the summary maintains λ answers per query with high probability. In this direction, we formalize the construction of approximate FSSs under a triple *budget* as an instance of the well-known *set cover* problem (more specifically, a *set multicover*) [14]. Let each query $Q_i \in \mathcal{Q}_n$ have a set of sampled answers $A_i = \{a_{i1}, a_{i2}, \ldots, a_{im_i}\}$, where each answer $a_{ij} \subseteq G$ is a set of RDF triples (i.e., a mini-graph). Let $T = \bigcup_{i,j} a_{ij}$ be the set of all triples appearing in any answer to any query. We define binary variables $x_t \in \{0, 1\}$ for each triple $t \in T$, where $x_t = 1$ if t is selected in FSS and 0 otherwise. We also define binary indicator variables for each answer a_{ij}:

$$y_{ij} = \begin{cases} 1 & \text{if all triples in } a_{ij} \text{ are included in FSS } (x_t = 1, \forall t \in a_{ij}), \\ 0 & \text{otherwise.} \end{cases}$$

We now formulate the problem as follows:

$$\textbf{Minimize} \quad \sum_{t \in T} x_t \tag{P2}$$

$$\textbf{subject to} \quad \sum_{j=1}^{m_i} y_{ij} \geq \lambda, \qquad \forall i = 1, \ldots, n, \tag{1}$$

$$y_{ij} \leq x_t, \qquad \forall i, j \text{ and } t \in a_{ij}, \tag{2}$$

$$x_t \in \{0, 1\}, \quad y_{ij} \in \{0, 1\}, \forall t \in T,\ \forall i, j.$$

Constraint (1) ensures that for each query Q_i, at least λ of its sampled answers are fully retained (i.e., all triples in those answers are included). Constraint (2) enforces that an answer a_{ij} can only be marked as retained ($y_{ij} = 1$) if all of its constituent triples are selected ($x_t = 1$ for all $t \in a_{ij}$). This formulation captures the semantic requirement that query answers must be preserved in full (not partially), and provides a combinatorial foundation for optimizing approximate FSS construction under a strict triple budget. As we will demonstrate in Section 3.2, this budget-aware approach not only respects strict triple constraints but also achieves *coverage* (i.e., the percentage of queries that the approximate FSS can answer with at least λ results -when such results exist) close to the optimal, with only marginal performance loss even at aggressive budget reductions.

Greedy Approximation Algorithm. A natural approach for addressing the aforementioned problem is a greedy heuristic that iteratively picks triples to cover as many remaining query requirements as possible. In this direction, we present Algorithm 2. The algorithm begins (line 1) by initializing the summary set and the universe of all candidate triples. In lines 2–7, it processes each query in the workload by initializing its answer set, required coverage count, and extracting and executing an oversampled SPARQL query to retrieve candidate answers. Each answer is stored along with a coverage flag, and all encountered triples are accumulated into

Algorithm 2 LFS-Budget-Greedy

Input: Q; λ; SP; κ is the total number of results to consider from each query; the target summary size B.
Output: An approximate FSS for G, λ and Q of size B.

1: $S \leftarrow \emptyset; U \leftarrow \emptyset$
2: **for all** $Q_i \in Q$ **do**
3: $A_i \leftarrow \emptyset; r_i \leftarrow \lambda$ ▷ initialize answers & answer requirements
4: $Q_i' \leftarrow \texttt{construct}(Q_i, \kappa)$
5: $\mathcal{R}_i \leftarrow \texttt{executeQuery}(Q_i', SP)$ ▷ Retrieve answers
6: **for all** results $a_{ij} \in \mathcal{R}_i$ **do** ▷ Answer coverage indicators
7: $y_{ij} \leftarrow 0; A_i \leftarrow A_i \cup \{a_{ij}\}; U \leftarrow U \cup a_{ij}$
8: **while** some $r_i > 0$ and $|S| < B$ **do**
9: **for all** $t \in U \setminus S$ **do**
10: $gain(t) \leftarrow 0$
11: **for all** queries Q_i with $r_i > 0$ **do**
12: **for all** $a_{ij} \in A_i$ with $y_{ij} = 0$ and $t \in a_{ij}$ **do**
13: **if** $a_{ij} \subseteq S \cup \{t\}$ **then** $gain(t) \leftarrow gain(t) + 1$
14: Select $t^* \in U \setminus S$ with highest $gain(t)$ (break ties arbitrarily)
15: $S \leftarrow S \cup \{t^*\}$
16: **for** queries Q_i with $r_i > 0$ **do**
17: **for all** $a_{ij} \in A_i$ such that $y_{ij} = 0$ and $a_{ij} \subseteq S$ **do**
18: $y_{ij} \leftarrow 1; r_i \leftarrow r_i - 1$
19: **return** S

the candidate pool. The core greedy loop (lines 8–18) continues as long as there are queries needing answers and the budget has not been exhausted. For each triple not yet selected, the algorithm computes a gain score reflecting how many new full answers that triple could help complete. It then selects the triple with the highest gain (line 14), adds it to the summary (line 15), and updates the coverage status of answers and queries that are now satisfied (lines 16–18). Once all query requirements are met or the budget is used up, the final summary is returned (line 19).

Example 3. Now consider Q1 from Example 2 and Q3

```
Q3. SELECT ?x WHERE { ?x rdf:type Person. ?x worksWith ?y.
      ?x affiliationOf <Princeton University>.}
```

Assume we invoke Algorithm 2 with $\lambda = 1, \kappa = 2$ and a triple budget of $B = 5$. The result CONSTRUCT queries retrieve the following answers for Q_1:

```
a11 = {t1 = (John Arch. Wheeler, rdf:type, Professor),
t2 = (Richard Feynman, rdf:type, Person),
t3 = (John Arch. Wheeler, advisorOf, Richard Feynman)}
a12 = {t4 = (Albert Einstein, rdf:type, Professor),
t6 = (Nathan Rosen, rdf:type, Person),
t5 = (Albert Einstein, advisorOf, Nathan Rosen)}
```

For Q_3:

```
a21 = {t2 = (Richard Feynman, rdf:type, Person),
t8 = (Richard Feynman, affiliationOf, Princ. Univ.),
t9 = (Richard Feynman, worksWith, John Arch. Wheeler)}
a22 = {t6 = (Nathan Rosen, rdf:type, Person),
t7 = (Nathan Rosen, affiliationOf, Princ. Univ.),
t10 = (Nathan Rosen, worksWith, Albert Einstein)}
```

The candidate universe U consists of the triples t_1–t_{10} appearing in the above answer-induced mini-graphs. The greedy process first selects the shared high-gain triple t_2, contributing to both a_{11} and a_{21}. It then completes a_{21} by adding t_8 and t_9, satisfying Q_2. Finally, to satisfy Q_1, the algorithm selects t_3 and t_1, exhausting the budget. The resulting summary is $S = \{t_1, t_2, t_3, t_8, t_9\}$, which satisfies both queries by prioritizing high-utility shared triples.

By formulating the summary construction problem as a set multicover problem, we also gain formal performance guarantees. In particular, the greedy strategy provides a tractable way to approximate the optimal summary size within a logarithmic factor.

Theorem 2 (Approximation Guarantee)**.** Let OPT be the maximum number of queries that can be covered under a budget B. Then the greedy algorithm achieves:

$$\text{CoveredQueries} \geq \left(1 - e^{-B/\text{OPT}_{size}}\right) \cdot \text{OPT}$$

where OPT_{size} denotes the minimum number of triples required to fully satisfy OPT queries.

Proof. (sketch) For the proof, the problem is modeled as a quota-based coverage task, where each query must be fully satisfied (i.e., λ complete answers covered). At each greedy step, we select the triple that maximizes the number of newly satisfied queries. By averaging over the optimal solution, there always exists a triple contributing at least a $1/OPT_{size}$ fraction of the remaining optimal coverage. This leads to a recurrence showing exponential decay of the gap to the optimum, which solves to the aforementioned bound, generalizing the classic analysis of greedy submodular maximization under cardinality constraints [18]. ∎

This triple bound improves over the classical $(1 - 1/e)$ as the available budget B becomes large relative to the cost of the optimal solution. Intuitively, this theorem ensures that even though we cannot compute the exact optimal set of triples under the budget in a reasonable time, our greedy approach will always find a set whose coverage is provably close to the best possible.

Example 4. For example, in our DBpedia workload with $\lambda = 5, \kappa = 50, B = 10^4$, the optimal budget-constrained coverage of the queries would require an exhaustive search over more than 10^{40} candidate triple subsets, which is computationally infeasible. Theorem 2 guarantees that LFS-Budget-Greedy achieves a solution within a provable factor of this unreachable optimum. In practice, our experiments show (Fig. 8) that the achieved coverage is within 95–98% of the best coverage observed among all tested methods, confirming that the theoretical bound translates into high-quality summaries while keeping a reasonable runtime.

Budget Estimation. Similarly, we can estimate the required summary size B that ensures, with high probability, that each query Q_i has at least λ of its sampled answers retained. We assume each query Q_i contributes a set $A_i = \{a_{i1}, \ldots, a_{i\kappa}\}$ of κ sampled answers, where each a_{ij} is a set of maximum T_a triples. Our objective is to retain at least λ fully-covered answers for each query. The total number of sampled answer triples is at most $n \times \kappa \times T_a$. Let p be the probability that a randomly selected triple from the universe of candidate triples T contributes to the completion of a required answer (i.e., it belongs to some a_{ij} such that $y_{ij} = 0$ and $r_i > 0$). Since each answer requires all of its T_a triples to be included, we assume that an answer is successfully covered only if all T_a of its triples are included in S. The expected number of triple insertions needed to cover one answer is approximately $m/\bar{g}$, where $\bar{g}$ is the average gain per triple (i.e., number of uncovered answers it can

help complete). Since the greedy strategy always selects the triple with the maximum gain, we approximate:

$$B \approx \lambda \times n \times \frac{T_a}{\bar{g}}.$$

In realistic graphs, $\bar{g} \geq 1$ due to shared triples among answers (e.g., entity types, affiliations), especially when variable bindings overlap across queries. To make the estimation robust, we include a safety factor $\rho > 1$:

$$B \approx \rho \times \lambda \times n \cdot \frac{T_a}{\bar{g}}.$$

The factor ρ can be empirically tuned based on graph redundancy, e.g., $\rho = 1.2$ for graphs with moderate overlap, and $\rho = 2.0$ in sparse graphs with low triple reuse.

Example 5. In Example 3, each query has two answers of $T_a = 3$ triples. Setting $\lambda = 1$, $n = 2$, and assuming average gain $\bar{g} \approx 1.5$, we get: $B \approx 1.5 \cdot 2 \cdot \frac{3}{1.5} = 6$. As shown in the example, the greedy algorithm succeeds with $B = 5$, confirming that the estimation is slightly conservative, which is desirable under uncertainty. This analysis provides a principled way to anticipate resource requirements for summary construction in budget-constrained environments.

Complexity. The complexity of the LFS-Budget-Greedy algorithm is dominated by two main components: the preprocessing phase, where each of the n queries in the workload is executed against the SPARQL endpoint to retrieve κ oversampled answers of at most T_a triples each, and the greedy selection phase, where triples are iteratively chosen to satisfy the λ-coverage requirements under the budget B. The preprocessing incurs a cost of $O(n \cdot \text{SPARQL}(\kappa))$, where $\text{SPARQL}(\kappa)$ denotes the cost of retrieving κ answers for a BGP query. The greedy loop runs for at most B iterations, and in each iteration, it computes the gain of every remaining candidate triple, requiring scanning all uncovered answers, yielding $O(B \cdot |U| \cdot n\kappa T_a)$ time in the worst case, where U is the candidate universe of triples. Consequently, the overall running time of LFS-Budget-Greedy is $O(n \cdot \text{SPARQL}(\kappa) + B \cdot |U| \cdot n\kappa T_a)$, with the SPARQL executions typically dominating in practice, while space complexity is $O(n\kappa T_a)$.

Incremental Maintenance. When the summary has already reached the target budget B, new queries may still require additional triples in order to satisfy their λ required answers. In this case, Algorithm 3 extends the summary incrementally by allowing *swaps* between existing and newly discovered triples. After integrating the new queries and their sampled answers (lines 1–6), the algorithm recomputes for each query how many answers are currently covered by the existing summary (lines 7–12). In the greedy improvement phase (lines 13–29), it selects a triple $t^+ \notin S$ with the highest marginal gain (i.e., the number of additional answers that would become covered if t^+ were inserted; lines 14–19) and a triple $t^- \in S$ with the smallest loss (i.e., the number of currently covered answers that depend on t^-; lines 21–25). If the gain of inserting t^+ exceeds the loss of removing t^-, the algorithm performs a swap while keeping $|S| = B$ (lines 26–29). Coverage is then updated accordingly (lines 28–33), and the process repeats until no beneficial swap exists. In this way, the summary remains within the fixed budget while continuously adapting to newly arrived queries by evicting low-utility triples and incorporating higher-impact ones.

Algorithm 3 Incremental-LFS-Budget-Greedy with Eviction

Input: Summary S (with $|S| = B$); existing candidate triple universe U; existing answer sets A_i for past queries; $\mathcal{Q}_{new}$; λ; κ; SP; B.
Output: An approximate FSS for G, λ and $\mathcal{Q} \cup \mathcal{Q}_{new}$ of size B.

1: **for all** $Q_i \in \mathcal{Q}_{new}$ **do** ▷ Phase 1: process only the new queries
2: $A_i \leftarrow \emptyset$ ▷ initialize answers for new query
3: $Q_i' \leftarrow \texttt{construct}(Q_i, \kappa)$
4: $\mathcal{R}_i \leftarrow \texttt{executeQuery}(Q_i', SP)$ ▷ Retrieve answers
5: **for all** results $a_{ij} \in \mathcal{R}_i$ **do**
6: $A_i \leftarrow A_i \cup \{a_{ij}\}$; $U \leftarrow U \cup a_{ij}$
7: **for all** Q_i **do** ▷ Phase 2: recompute coverage for all queries
8: $c_i \leftarrow 0$ ▷ number of covered answers for Q_i
9: **for all** $a_{ij} \in A_i$ **do**
10: **if** $a_{ij} \subseteq S$ **and** $c_i < \lambda$ **then** $y_{ij} \leftarrow 1$; $c_i \leftarrow c_i + 1$
11: **else** $y_{ij} \leftarrow 0$
12: $r_i \leftarrow \lambda - c_i$ ▷ remaining answer requirements
13: **while** some $r_i > 0$ **do** ▷ Phase 3: greedy local improvement
14: **for all** $t \in U \setminus S$ **do** ▷ compute gain
15: $gain(t) \leftarrow 0$
16: **for all** queries Q_i with $r_i > 0$ **do**
17: **for all** $a_{ij} \in A_i$ with $y_{ij} = 0$ and $t \in a_{ij}$ **do**
18: **if** $a_{ij} \subseteq S \cup \{t\}$ **then** $gain(t) \leftarrow gain(t) + 1$
19: Select $t^+ \in U \setminus S$ with highest $gain(t)$
20: **if** $gain(t^+) = 0$ **then break** ▷ no triple improves coverage
21: **for all** $t \in S$ **do** $loss(t) \leftarrow 0$ ▷ compute loss of removing t
22: **for all** queries Q_i **do**
23: **for all** $a_{ij} \in A_i$ with $y_{ij} = 1$ **do**
24: **for all** $t \in a_{ij} \cap S$ **do** $loss(t) \leftarrow loss(t) + 1$
25: Select $t^- \in S$ with smallest $loss(t)$
26: **if** $gain(t^+) \leq loss(t^-)$ **then break** ▷ no beneficial swap under B
27: $S \leftarrow (S \setminus \{t^-\}) \cup \{t^+\}$ ▷ perform swap
28: **for all** Q_i **do** ▷ update coverage after swap
29: $c_i \leftarrow 0$
30: **for all** $a_{ij} \in A_i$ **do**
31: **if** $a_{ij} \subseteq S$ **and** $c_i < \lambda$ **then** $y_{ij} \leftarrow 1$; $c_i \leftarrow c_i + 1$
32: **else** $y_{ij} \leftarrow 0$
33: $r_i \leftarrow \lambda - c_i$
34: **return** S

In the incremental setting, only the newly arrived queries n_{new} are issued against the endpoint, giving a preprocessing cost of $O(n_{new} \cdot \text{SPARQL}(\kappa))$. These queries contribute at most $n_{new}\kappa T_a$ additional triples to the candidate universe U. Since the summary already contains B triples and eviction is allowed, each local-improvement iteration performs both (i) a full recomputation of gains for all triples in $U \setminus S$, costing $O(|U| \cdot n_{new}\kappa T_a)$, and (ii) a recomputation of losses for all triples in S, costing $O(B \cdot n\kappa T_a)$ over the combined old and new workloads. At most B swaps can be performed under the fixed budget, leading to a total incremental complexity of $O\Big(n_{new} \cdot \text{SPARQL}(\kappa) + B \cdot (|U| \cdot n_{new}\kappa T_a + B \cdot n\kappa T_a)\Big)$.

Construction Cost and Trade-offs. Although the construction of the FSS requires issuing SPARQL queries for the workload, this process is executed entirely offline and is linear in the workload size. In practice, as shown in Section 3.1, even for real logs containing tens of thousands of queries, construction completes on commodity hardware within a few hours. Moreover, LFS provides

multiple mechanisms that allow practitioners to trade construction cost for summary size or accuracy. The parameter λ controls how many results per query are materialized, while the oversampling parameter κ in LFS-Budget-Greedy bounds the number of answers retrieved, directly reducing endpoint load. The budget parameter B enables the user to specify any target summary size, and the greedy procedure guarantees high coverage even under aggressive budget reductions. Finally, because FSSs are incrementally maintainable (Section 3.3), only newly observed queries need to be processed as logs evolve, avoiding recomputation from scratch. Together, these mechanisms provide flexible and effective trade-offs for balancing construction overhead and summary quality.

Sampling and Cost–Coverage Trade-offs. It is important to note that neither LFS nor LFS-Budget-Greedy requires the processing of the full results of the queries in the log. In LFS, each query is evaluated with a LIMIT λ clause, and we only materialize up to λ answer graphs per query. In LFS-Budget-Greedy, we introduce an oversampling parameter $\kappa \geq \lambda$ that bounds the number of answers per query that are ever retrieved and used to populate the candidate universe U. As a result, the size of U is at most $n \times \kappa \times T_a$ triples rather than the potentially much larger full result sets. This naturally induces a trade-off between construction cost and coverage: smaller κ reduces the offline query-execution time and memory footprint but may miss some answers, whereas larger κ increases the likelihood of selecting high-utility triples in the greedy phase. In Section 4.2, we empirically study this trade-off and show that, for our workloads, $\kappa = 50$ already achieves coverage very close to that obtained with $\kappa = 100$ or 200, while keeping construction costs significantly lower.

Forgetting and Summary Reduction. While the FSS can be incrementally expanded as new queries arrive, long-running deployments may require the summary to adapt when storage budgets are fixed, or when previously important regions of the graph become less relevant. Our budget-aware formulation naturally supports both offline and online forgetting. Offline forgetting is achieved by reducing the target budget B and reapplying the greedy construction, which automatically retains high-utility triples while discarding those that contribute little to satisfying the λ answer requirements. In addition, Algorithm 3 provides an online forgetting mechanism: when $|S| = B$ and new queries introduce higher-utility triples, the algorithm evaluates local gain–loss trade-offs at the answer level, evicting low-impact triples and inserting more informative ones while keeping the summary size fixed. This enables the FSS to evolve over time in a cache-like manner, remaining compact, adaptive, and aligned with the evolving workload, while more sophisticated eviction heuristics remain an interesting direction for future exploration.

Query answering using FSSs. In essence, the problem of generating an FSS is a specialization of the materialized view selection problem for conjunctive queries. Chirkova et al. [10] showed that the problem of view materialization for conjunctive queries is NP-complete, indicating that exhaustive solutions to the aforementioned problem cannot be practically applied to large query workloads. However, in our approach, we require that for each query in the query workload *only λ results* be stored (when those are available in the original graph), and we *select all views* to be materialized as they only introduce minimal space overhead. In fact, FSSs require only a hundred MBs, as we will show in the experimental section, which even commodity computers now have. On top of that, using LFS-Budget-Greedy, we can further set specific budget requirements, approximating the best quality FSS that respects the given constraints. Further, we enjoy one additional and very significant benefit in query answering. Instead of rewriting the input queries to be answered by the views, in our case, *no query rewriting is required*, and we can directly forward the queries to the FSS to be answered. In fact, in FSS summaries, we can guarantee *completeness* by construction, i.e., that all queries used for constructing the FSS can be answered by the summary, returning the first few answers to those queries.

3 EXPERIMENTAL EVALUATION

Implementation. All algorithms were developed using Python. A local Virtuoso Triple Store was used for storing all datasets, whereas the summaries were stored and queried using an RDFLib SimpleMemory on the same machine. The evaluation was performed using a commodity computer, i.e., Windows 10 with an Intel® Core™ i3 10100 CPU @ 3.60GHz (4 cores) and 16 GB RAM.

Datasets. For our experiments, we used three knowledge graphs, i.e., YAGO, DBpedia and Wikidata. *YAGO* [36] is a KG that augments WordNet with common knowledge facts extracted from Wikidata, converting WordNet from a primarily linguistic resource to a common knowledge base. YAGO has knowledge of more than 10 million entities and contains more than 120 million triples about these entities. The query workload was provided by the YAGO endpoint and consists of 30,000 queries. Then we use *DBpedia* [5], along with the corresponding query workload. DBpedia aims to extract structured content from the information created in the Wikidata project. It includes 9.5 billion RDF triples. The available query workload acquired from the DBpedia endpoint is 16.3 MB, including 58,610 queries. *Wikidata* [41] is an open knowledge base that can be read and edited by both humans and machines. Wikidata contains 100 million items, and 1.4 billion triples, covering many general topics for knowledge exploration and data science applications. The query workload was retrieved from [28] and includes 192,325 queries.

In some cases, the queries were syntactically wrong and were removed. Further, several queries did not return results and were also ignored. Removing aggregate and non-monotonic queries, in addition, resulted in actually using 17,819 queries for YAGO, 30,800 queries for DBpedia, and 90,696 queries for Wikidata.

Metrics. We split the workload into 80% of the queries (14,255 queries for YAGO, 24,640 queries for DBpedia, and 72,556 queries for Wikidata) for constructing the first-sight summaries (for various λ and B), and we keep the remaining 20% for testing the summaries. For evaluation, we use the following metrics:

Construction time. We evaluate the construction time for the generation of the summary using the two algorithms.

FSS size. Further, we examine the space required for storing the uncompressed version of the FSS.

Query answering efficiency. We measure query efficiency when asking for the first answers. We contrast query efficiency when users search for the first few answers from the original graph using the LIMIT clause in their queries.

Coverage. Further, for approximate summaries, we measure the percentage of the queries that can return at least λ answers –when they exist– from the summary.

Competitors. As we are the first to define the problem of generating exact and approximate FSSs, a direct competitor that returns the first few answers to user queries does not exist. Nevertheless, we compare our approach with SAGE [30], which, although it requires a smart client, is able to return a first answer for input queries, as it splits SPARQL execution into chunks of time, returning each time a subset of the results. Further, we compare our approach with PING [7], which is also able to answer progressively SPARQL queries, where we base our evaluation of the first set of answers retrieved.

3.1 Evaluating construction time & size

Construction time. For warming up, we present the time needed to construct summaries using both the LFS and the LFS-Budget-Greedy (LFSB) using 80% of the queries of each dataset. The LFS algorithm for each query has to extract the query mini-graph, execute the corresponding SPARQL query, and then extract the first λ results. As such, by far the dominant time is query execution. We only present LFS for $\lambda = 1$, i.e., 1LFS, as the times for LFS for $\lambda = 1, 5, 10$ are exactly the same –we only keep more results for the summary. The LFSB, on the other hand, first selects κ results ($\kappa > \lambda$) and then uses the greedy loop to finally keep the ones with the highest gain. As such, we present the LFSB for $\lambda = 1, 5, 10$ out of 500 and 1000 results ($\kappa = 500, 100$) and B=100% (for B<100% the LFSB will have to search for fewer triples and will be faster).

The results are presented in Figure 4 (a, c, e). As shown, in essence, the time required for constructing the summaries is the time to query the endpoints with the corresponding LIMIT. As expected, the cost of LFSB is higher than the cost of LFS since more results are being retrieved and due to the greedy step in the sequel. However, still in LFSB, the dominant times are the times to execute the queries from the endpoints and fetch the results. Regarding the LFSB, the more answers are retained (as λ increases), the larger the execution time of the greedy step. Finally, we can see that the time required for constructing the summaries is larger in the cases where we have more queries to execute (14,255 queries for YAGO, 24,640 queries for DBpedia, and 72,556 queries for Wikidata). Nevertheless, summary construction is a task that is usually executed once and offline before starting to answer queries (in Section 3.3 we examine incremental update as well), so the construction time, in essence, does not affect the main purpose of summary construction, which is to facilitate rapid retrieval of the first answers.

Size. Next, we present the size of the summaries generated by our algorithms. The results are shown in Figure 4 (b, d, f). As the overall size depends on the number of times the query mini-graphs are instantiated, in essence, the size varies by the λ. The summary size is almost doubled when the λ is doubled, and this linear increase is also expected theoretically (refer to Section 2.1). Now, when more answers are sampled in the LFSB case, the algorithm retains the answers with the triples with the maximum gain. The bigger the sampling, the more chances that we find triples with bigger gain, which in turn result in a slightly smaller size since those triples cover more queries. Overall, the largest summary size of the three reasonably big knowledge graphs used in our experiments is 205MB, which can be stored in the main memory of commodity computers (even of mobile phones), introducing minimal requirements for answering rapidly the first few answers of users' queries.

Convergence. An interesting property of the incremental FSS construction is that, in practice, the summary exhibits a clear tendency toward stabilization as more queries are processed. Figure 4 (g) shows that the number of triples increases rapidly during the early stages (e.g., up to 40–50% of the workload), but the rate of growth drops sharply thereafter. For example, in DBpedia, the summary grows from 1,018 to 10,892 triples between 5% and 50% of the workload, but only from 18,140 to 20,885 triples between 80% and 100%. Wikidata and YAGO show the same pattern, with increasingly small increments beyond 70–80% of the workload. This diminishing-return effect suggests that the FSS gradually captures the core, high-utility triples that recur across the workload, after which additional queries contribute only limited new information. These empirical results indicate that the FSS becomes increasingly stable as the workload grows.

3.2 Train workload

LFS Query Efficiency. Next, we report query efficiency for returning the first results when we execute the queries in the train set, over our FSSs. The results are shown in Figure 4 (h). In the figure, we present the time required for getting the first few answers from our LFS summaries for λ=10, 5, and 1. We also include the time required to do the same from the SPARQL endpoints using the LIMIT clause. As shown in most cases, returning the first few answers from the FSS *is at least two orders of magnitude faster, showing the remarkable benefits that our summaries enjoy. At worst, our summaries return answers one order of magnitude faster than SPARQL endpoints* (Wikidata, YAGO 10LFS, DBpedia 10/5LFS), whereas *at best our summaries return answers three orders of magnitude faster* (YAGO 5/1LFS, DBpedia 1LFS). As the λ is decreased, the summary size is decreased as well, and this results in a decreased query execution time as a smaller graph has to be searched, and fewer results should be returned to the user.

LFSB Query Coverage. Next, we introduce a budget limitation, as a percentage of the FSS size (from x1 to x0.2), to the generated summaries, and we use LFSB. We present the cases for visiting $\kappa = 50, 100, 200$ results only for $\lambda = 5$ (summaries storing 5 results), as results are similar for $\lambda = 1$, and $\lambda = 10$. The results are shown in Figure 5. As shown, the more restrictive the budget, the more the coverage drops, reaching around 55% for all three datasets when we only keep 20% of the distinct triples for the summary. This corresponds to 2,879 triples for YAGO, 7,560 triples for DBpedia, and 23,653 triples for Wikidata, and is in line with our theoretical expectations. These results align with the logarithmic-factor guarantee in Theorem 2. When the budget B is reduced to just 20% of the full summary size, LFSB still preserves roughly half of the query coverage. This slow drop in coverage is expected from the $(1 - e^{-B/OPT_{size}})$ bound, which predicts that greedy selection retains a substantial fraction of the optimal coverage even under tight budgets. The reason is that the gain-based triple prioritization front-loads high-utility triples shared across many answers, so the most coverage-critical information remains in the summary despite aggressive pruning.

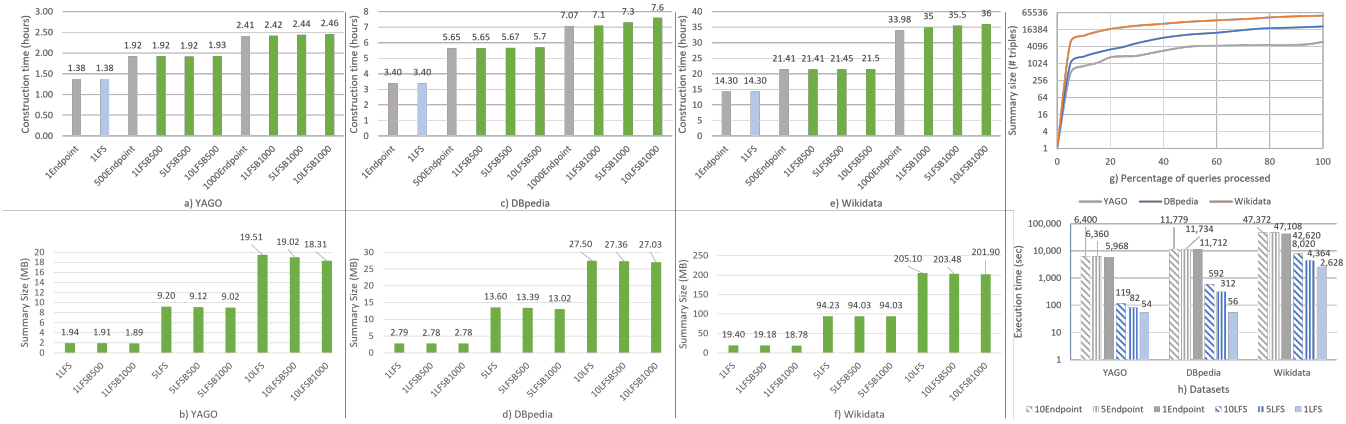

Figure 4: Construction time and summary size for LFS and LFSB for YAGO (a, b), DBpedia (c, d) and Wikidata (e, f). Convergence results as more queries are processed (g), and query efficiency for the train queries (h).

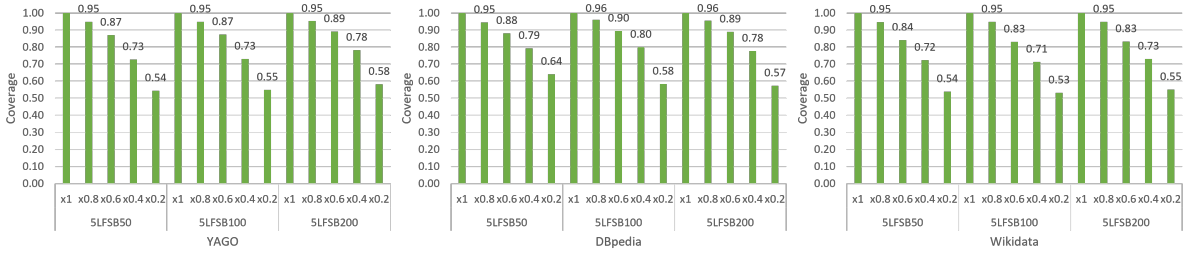

Figure 5: Coverage for different budget fractions and κ values.

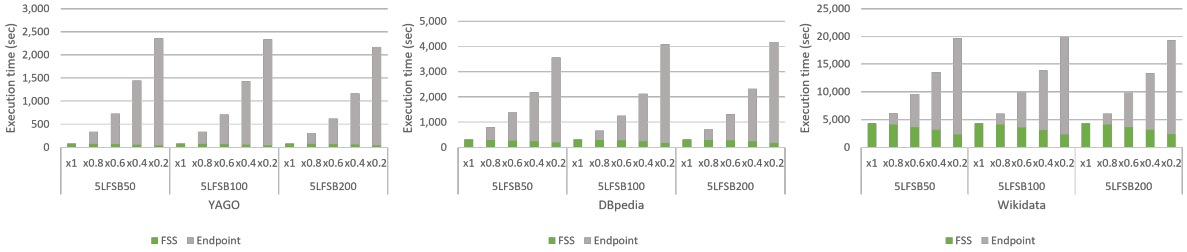

Figure 6: Time to retrieve 5 answers under budget limits.

LFSB Query Efficiency. Next, we identify how the budget affects query efficiency for retrieving the 5 first five answers (if they exist) when we have budget constraints. The results are shown in Figure 6. For the queries that do not have all answers in the LFSB summary, we resort to the SPARQL endpoint to retrieve all results, and the corresponding execution time is shown in gray. As shown, the more restrictive the budget constraint, the more queries we need to answer through the SPARQL endpoints, and this leads to a significant increase in total execution time. However, even with a 20% budget, the combined execution time is two times faster than simply going to the endpoints (refer to Figure 4h), showing the high benefits of our approach. Further, we identify that visiting more triples (50, 100, or 200) essentially only marginally improves coverage and improves efficiency. As such, for the rest of the experiments, we stick to $\kappa = 50$ for $\lambda = 5$.

3.3 Test workload

Next, we focus on checking how our summaries can be used to answer queries that appear in the test 20% part of the queries.

Coverage. First, we evaluate the coverage of the LFS summaries for the various λ. The results are shown in Figure 7 (left). We observe that the summaries already cover a large percentage of the YAGO and Wikidata queries (85–93%), while coverage for DBpedia stabilizes around 44–45%. This behavior is closely tied to the structure of real SPARQL workloads, which are known to exhibit strong repetition and overlap across queries [8]. In our datasets,

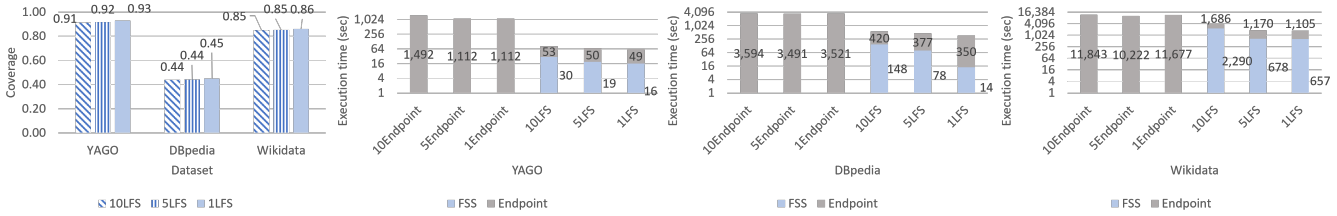

Figure 7: Coverage (left) and query efficiency (right) on test queries over exact summaries.

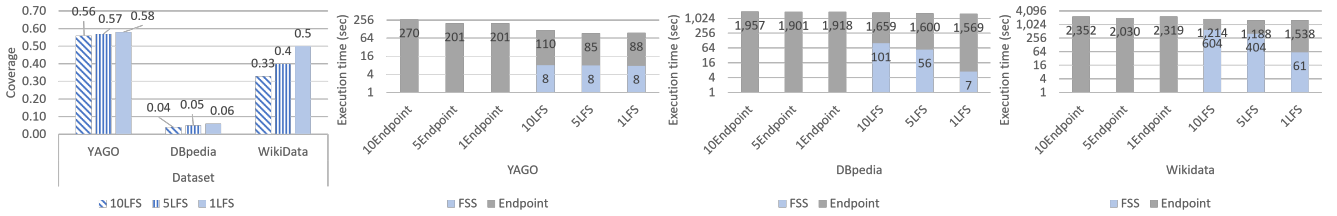

Figure 8: **Coverage (left) and query efficiency (right) on unknown queries over exact summaries.**

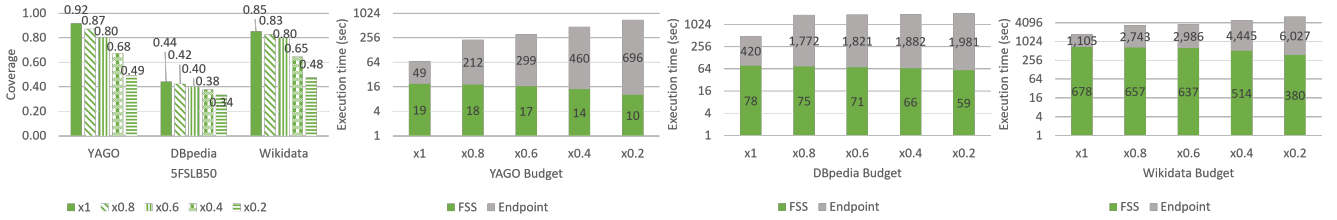

Figure 9: Coverage (left) and query efficiency (right) on test queries over approximate summaries (5LFSB).

75% of the YAGO test queries, 40% of the DBpedia test queries, and 75% of the Wikidata test queries also appear in the training workload. However, the achieved coverage is significantly higher than these repetition rates, indicating that the FSS does not only benefit from exact query reuse (as in caching), but also from structural and answer-level overlap across different queries that target similar regions of the KG. This provides empirical support for the underlying assumption that future queries share information with past workloads and explains why the summaries generalize well beyond the exact queries used for their construction.

Efficiency. Next, we focus on identifying how coverage translates into execution time, again resorting to SPARQL endpoints when we cannot get the correct number of results from the FSS. The results are shown in Figure 7 (right). For YAGO, using our summaries, the first few answers to user queries are answered *two orders of magnitude* faster than just going to the endpoints using the LIMIT clause. Note that FSSs can answer 91-93% of the queries here, but the remaining percentage requires double the time to be answered. For DBpedia, in all cases, the query execution times are *one order of magnitude faster* than directly going to the DBpedia endpoint. Again, the dominant time is the time required by the endpoint, whereas in this case, the FSSs can answer only 44-45% of the queries. Finally, for Wikidata, again the total execution time is *one order of magnitude faster* than going to the endpoints with slight fluctuations based on the quality of the summary, whereas the FSSs can cover 85-86% of the queries. Overall, in all cases, using the FSSs *is at least one order of magnitude faster* than going to the SPARQL endpoints, showing the promising results of our approach.

Unseen queries. Next, we evaluate the ability of LFS summaries to generalize to unknown queries, i.e., queries not observed during summary construction. Figure 8 reports the results. LFS shows strong generalization to unseen workloads on YAGO and Wikidata, where even small summaries answer a substantial fraction of unknown queries. With 10LFS, coverage reaches about half of unseen queries, and reducing the summary size leads only to moderate degradation. In contrast, DBpedia achieves consistently low coverage across all settings, indicating higher structural diversity and limited overlap between training and unseen queries. Execution-time results further highlight the benefits of LFS when summaries are applicable. For YAGO, the 56-58% of the queries answered from the FSS are processed in 8s, compared to 160–113s at the endpoint, *yielding total improvements up to 20×*. Similar trends are observed for Wikidata: although FSS evaluation is more expensive due to larger summaries, it remains substantially faster than endpoint execution. DBpedia shows a different pattern: while FSS execution is higher (from 7s to 101s) due to lower coverage, endpoint times remain dominant (≈1600s), meaning that even partial coverage translates into noticeable end-to-end latency reductions. Overall,

these results demonstrate that LFS summaries retain strong generalization capabilities for unknown queries, especially in datasets with high structural regularity such as YAGO and Wikidata.

Introducing budget constraints. Next, we introduce budget constraints, and we visualize coverage and the impact on the combined execution time. The results are shown in Figure 9. For YAGO and Wikidata, coverage gradually drops to 48-49% when we only retain 20% of the triples in the summary, whereas in DBpedia the coverage gradually drops from 44% to 34%. It is impressive that with only 20% of the triples of the full summary, we can still answer roughly 50% of the test queries still in YAGO and Wikidata. Translating this into query execution time, we again see that the smaller summaries lead to an increased execution time, as we need to resort to the endpoints for the queries which we cannot return the correct number of answers. *In all cases, however, using the summary is at least 40% faster than going to the SPARQL endpoints.*

Incremental Update. In the next experiment, we focus on the incremental version of our algorithms. As such, we keep 20% of the test queries, and we build an FSS for 10%, 50%, and 100% of the train queries. We show the experiment for only $\lambda = 10$, which is the largest summary due to a lack of space. The results are shown in Figure 10. The construction-time results show a clear and predictable scaling behavior for both approaches as the workload grows incrementally. Exact LFS exhibits near-linear growth with the number of processed queries, confirming that summary construction is dominated by issuing LIMIT queries to the endpoint and appending the resulting mini-graphs. LFSB consistently requires more construction time at each increment due to the additional greedy selection and eviction steps needed to enforce the budget constraint; however, this overhead remains proportional to the workload size and is incurred entirely offline. Overall, both methods scale smoothly with increasing history, while LFSB trades higher offline construction cost for bounded summary size and improved long-term sustainability under strict memory budgets. The incremental results highlight a clear trade-off between coverage and execution time when comparing exact LFS with its budget-aware variant, LFSB. As more training queries are incorporated, both methods exhibit monotonic improvements in coverage and corresponding reductions in total query execution time, confirming the effectiveness of workload accumulation. Exact incremental LFS consistently achieves higher coverage at each increment, which translates into fewer fallbacks to the SPARQL endpoint and thus lower endpoint execution time, particularly for YAGO and Wikidata. However, this comes at the cost of growing summary size and increased in-memory query time. In contrast, incremental LFSB enforces a fixed budget, resulting in slightly lower coverage across all increments, especially under tight budgets, but maintains bounded summary size and more stable FSS execution times. Despite the reduced coverage, LFSB still yields substantial end-to-end latency improvements over endpoint-only execution, and its coverage gap with LFS narrows as more queries are processed due to the eviction-aware greedy updates that prioritize high-utility triples. Overall, these results show that LFS is preferable when memory is not a constraint and maximum coverage is desired, whereas LFSB provides a robust and scalable alternative for long-running systems operating under strict storage budgets, preserving most of the latency benefits while offering predictable resource usage.

Beyond Encyclopaedic KGs. In order to test our solution beyond encyclopaedic KGs, we focused next on the DrugBank [19], a curated biomedical knowledge graph that integrates detailed information on drugs, drug–target interactions, enzymes, transporters, pathways, and associated diseases. We extracted 20K queries for our experiment from LSQ [34]. We again split the queries into test/-train (80/20), and we only present the results for the test queries. The results are presented in Figure 11. As shown, LFS achieves consistently high coverage across all summary sizes, indicating strong generalization to test queries. 1LFS, covers 60% of unknown queries, while 5LFS and 10LFS exhibit only marginally lower coverage (0.59 and 0.58, respectively), showing that coverage remains stable. In terms of execution time, summary-based combined execution (FSS) is around two times faster than endpoint evaluation. Queries answered directly from the summary execute in 6–16 seconds. Even when queries must partially fall back to the endpoint, the use of LFS leads to substantial latency reductions, demonstrating that DrugBank strongly benefits from summary-based query answering.

Comparison with baselines. Finally, we compare with SAGE and PING. SAGE and PING are the only systems we were able to identify that return a first answer to SPARQL queries. Although SAGE requires a smart client in order to be able to run and does not particularly address the FSS generation problem, it still, according to the authors, highly improves the time for the first results. PING, on the other hand, returns the answers to a given query progressively; however, without allowing for the configuration of the number of returned results in each step - we retain the results from the first step here. We installed both systems on our machine and ensured that they both exploit the full memory of the machine. We only tested WikiData and DBpedia, as they were already pre-configured for SAGE. In addition, we compare with the SPARQL endpoint (using LIMIT 10) on the same machine, which corresponds to the cold start (0% bars) of LFS, and LFSB with a budget of 0.5x, for $\lambda = 10$. Again, our summaries are constructed using 80% of the queries of each dataset, and the results presented in Figure 12 concern the rest 20% of the queries. The reported times for LFS/LFSB are the total combined execution time of the queries.

For DBpedia, at 0% (cold start), LFS and LFSB resort to the endpoint for returning first answers (3,590s). SAGE and PING are faster (2.093s and 1,953s). With 10% of the workload, LFS already overtakes both baselines, whereas with 50%, LFS reaches 1,273s, 1.6× faster than PING. With the full workload, LFS reaches 525s, *giving a 4× speedup over SAGE/PING*. LFSB mirrors the trend but remains above full-budget LFS. In the case of WikiData, the gains are even more pronounced because of the high endpoint cost. At 0%, LFS/LFSB resolve to the endpoints and require 11,843s. With 10% of the workload, LFS drops to 3,582s, outperforming both PING (7,054s) and SAGE (10,722s). With 50% and 100% of the workload, LFS reaches 3,460s and 3,007s, *giving 3.5× speedup over SAGE and 2.3× over PING*. LFSB improves similarly, being faster than both PING and SAGE after 10%. Overall, LFS and LFSB are competitive with existing systems even under limited workload history and become substantially faster as more queries are incorporated. This validates the central idea: workload-derived first-sight summaries provide major latency benefits at negligible storage cost, outperforming both SAGE and PING even with modest logs.

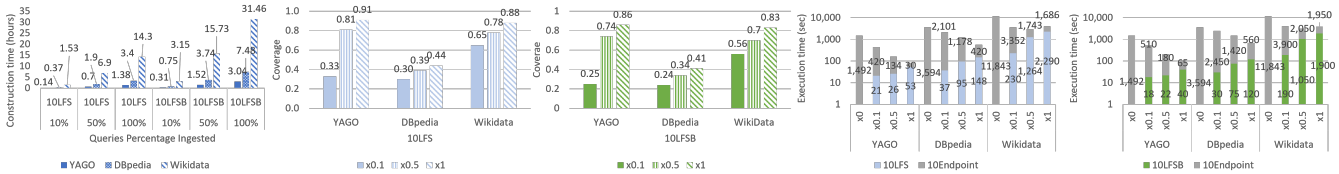

Figure 10: Construction time, coverage and query efficiency for the incremental versions.

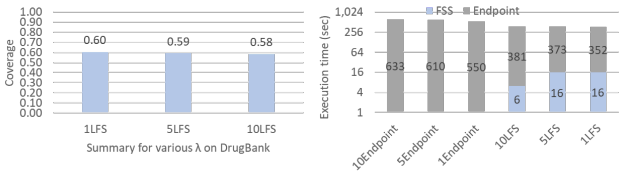

Figure 11: Coverage and query execution for DrugBank

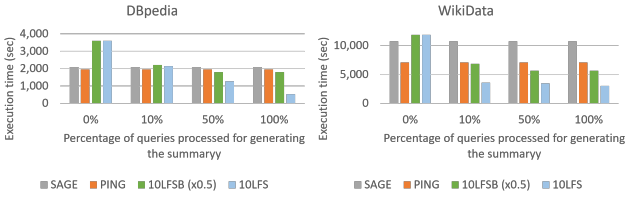

Figure 12: Query execution time for baselines, 10LFS and 10LFSB, as the number of processed queries increases.

4 RELATED WORK

Several works aim to optimize response time over large online triple stores, addressing quotas, timeouts, and high query latencies [3].

Query decomposition. A common approach decomposes queries into smaller subqueries that can be executed within endpoint limits and recombined client-side [4]. While effective in avoiding timeouts, this strategy requires a smart client and does not guarantee that all subqueries will terminate efficiently.

Restricted SPARQL server approaches. Systems such as TPF [40], SAGE [2, 30], and SmartKG [6] guarantee termination by restricting the SPARQL fragment or execution model. These approaches shift significant processing to the client, often increasing data transfer or client-side overhead, and may still delay first answers. In contrast, our approach requires no smart client and returns early answers directly from a precomputed summary.

Flexible and approximate query answering. Several methods relax or approximate queries to improve responsiveness. RELAX [15], preference-based approaches [11, 29], and APPROX [13] generate relaxed or alternative queries, while sampling-based techniques [33] approximate aggregates. These methods trade accuracy for speed, whereas we return exact answers without query relaxation or approximation.

Progressive and top-k query processing. PING [7] progressively returns answers by reducing intermediate results but provides no guarantees on the size or feasibility of early answers. Top-k approaches [16, 17, 26, 42, 43, 46] focus on ranked retrieval using specialized operators or indexes. In contrast, we aim to return any λ answers quickly, without ranking or query specialization.

Caching. Caching techniques store results or subqueries to accelerate future executions [21, 25, 31, 35, 44, 45, 47]. These methods face challenges in selection, eviction, and validity. Our approach differs by retaining answer-supporting triples from processed queries in a workload-driven summary: exact LFS grows monotonically without eviction, while the budget-aware variant performs principled gain–loss swaps at the triple level, ensuring fast delivery of the first λ answers under strict space constraints.

Exploratory search and KG exploration. Exploration systems [23] such as SPARKLIS [12], X2Q [24], Re2xOLAP [22], and RDF Explorer [37], focus on guiding users through large KGs via interaction and visualization. These systems do not address first-answer latency, which is critical in early exploration. LFS is complementary, enabling the retrieval of initial answers rapidly.

Overall, unlike existing approaches, LFS explicitly targets first-answer latency by materializing only minimal, answer-supporting subgraphs under a strict memory budget. To our knowledge, no prior system constructs budget-aware mini-graphs directly from query workloads to guarantee early availability of exact and approximate first answers.

5 CONCLUSIONS

We introduce first-sight summaries, compact semantic subgraphs that enable low-latency retrieval of the first answers to SPARQL queries. By combining ideas from materialized views and semantic summaries, we construct budget-constrained subgraphs that fit in main memory and can be queried directly. We propose two practical construction algorithms, LFS and LFS-Budget-Greedy, and show that both effectively reduce first-answer latency. Experiments on YAGO, DBpedia, and Wikidata show linear construction time in the query log size, modest storage requirements (as low as 205 MB), and substantial speedups—up to three orders of magnitude for known queries and two for unseen queries—outperforming relevant baselines. To our knowledge, this is the first approach to achieve such benefits with low construction cost and minimal storage overhead. Future directions include further compression, support for aggregate and non-monotone queries, and fully online, parallel construction methods for real-time summary updates.

ACKNOWLEDGMENTS

This work was supported by the MULTIPULM (GA 101226783) and PATH (101253520) Horizon EU projects. Also, the authors would like to thank Vassilis Karatzias for the interesting discussions on the mathematical foundations of the paper.

REFERENCES

[1] [n.d.]. W3C Recommendation, SPARQL Query Language for RDF. https://www.w3.org/TR/rdf-sparql-query/. Accessed: 2019-10-09.

[2] Julien Aimonier-Davat, Hala Skaf-Molli, Pascal Molli, Arnaud Grall, and Thomas Minier. 2022. Online approximative SPARQL query processing for COUNT-DISTINCT queries with web preemption. *Semantic Web* 13, 4 (2022), 735–755. https://doi.org/10.3233/SW-222842

[3] Waqas Ali, Muhammad Saleem, Bin Yao, Aidan Hogan, and Axel-Cyrille Ngonga Ngomo. 2022. A survey of RDF stores & SPARQL engines for querying knowledge graphs. *VLDB J.* 31, 3 (2022), 1–26. https://doi.org/10.1007/s00778-021-00711-3

[4] Carlos Buil Aranda, Axel Polleres, and Jürgen Umbrich. 2014. Strategies for Executing Federated Queries in SPARQL1.1. In *The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part II (Lecture Notes in Computer Science)*, Peter Mika, Tania Tudorache, Abraham Bernstein, Chris Welty, Craig A. Knoblock, Denny Vrandecic, Paul Groth, Natasha F. Noy, Krzysztof Janowicz, and Carole A. Goble (Eds.), Vol. 8797. Springer, 390–405. https://doi.org/10.1007/978-3-319-11915-1_25

[5] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary G. Ives. 2007. DBpedia: A Nucleus for a Web of Open Data. In *The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007 (Lecture Notes in Computer Science)*, Karl Aberer, Key-Sun Choi, Natasha Fridman Noy, Dean Allemang, Kyung-Il Lee, Lyndon J. B. Nixon, Jennifer Golbeck, Peter Mika, Diana Maynard, Riichiro Mizoguchi, Guus Schreiber, and Philippe Cudré-Mauroux (Eds.), Vol. 4825. Springer, 722–735. https://doi.org/10.1007/978-3-540-76298-0_52

[6] Amr Azzam, Javier D. Fernández, Maribel Acosta, Martin Beno, and Axel Polleres. 2020. SMART-KG: Hybrid Shipping for SPARQL Querying on the Web. In *WWW '20: The Web Conference 2020, Taipei, Taiwan, April 20-24, 2020*, Yennun Huang, Irwin King, Tie-Yan Liu, and Maarten van Steen (Eds.). ACM / IW3C2, 984–994. https://doi.org/10.1145/3366423.3380177

[7] Angela Bonifati, Stefania Dumbrava, Haridimos Kondylakis, Georgia Troullinou, and Giannis Vassiliou. 2025. Progressive Querying on Knowledge Graphs. In *Proceedings 28th International Conference on Extending Database Technology, EDBT 2025, Barcelona, Spain, March 25-28, 2025*, Alkis Simitsis, Bettina Kemme, Anna Queralt, Oscar Romero, and Petar Jovanovic (Eds.). OpenProceedings.org, 106–118. https://doi.org/10.48786/EDBT.2025.09

[8] Angela Bonifati, Wim Martens, and Thomas Timm. 2020. An analytical study of large SPARQL query logs. *VLDB J.* 29, 2-3 (2020), 655–679. https://doi.org/10.1007/s00778-019-00558-9

[9] Sejla Cebiric, François Goasdoué, Haridimos Kondylakis, Dimitris Kotzinos, Ioana Manolescu, Georgia Troullinou, and Mussab Zneika. 2019. Summarizing semantic graphs: a survey. *VLDB J.* 28, 3 (2019), 295–327. https://doi.org/10.1007/s00778-018-0528-3

[10] Rada Chirkova, Alon Y. Halevy, and Dan Suciu. 2002. A formal perspective on the view selection problem. *VLDB J.* 11, 3 (2002), 216–237. https://doi.org/10.1007/s00778-002-0070-0

[11] Peter Dolog, Heiner Stuckenschmidt, Holger Wache, and Jörg Diederich. 2009. Relaxing RDF queries based on user and domain preferences. *J. Intell. Inf. Syst.* 33, 3 (2009), 239–260. https://doi.org/10.1007/s10844-008-0070-7

[12] Sébastien Ferré. 2017. Sparklis: An expressive query builder for SPARQL endpoints with guidance in natural language. *Semantic Web* 8, 3 (2017), 405–418. https://doi.org/10.3233/SW-150208

[13] Riccardo Frosini, Andrea Calì, Alexandra Poulovassilis, and Peter T. Wood. 2017. Flexible query processing for SPARQL. *Semantic Web* 8, 4 (2017), 533–563.

[14] Qiang-Sheng Hua, Yuexuan Wang, Dongxiao Yu, and Francis C. M. Lau. 2010. Dynamic programming based algorithms for set multicover and multiset multicover problems. *Theor. Comput. Sci.* 411, 26-28 (2010), 2467–2474. https://doi.org/10.1016/J.TCS.2010.02.016

[15] Carlos A. Hurtado, Alexandra Poulovassilis, and Peter T. Wood. 2008. Query Relaxation in RDF. *J. Data Semant.* 10 (2008), 31–61. https://doi.org/10.1007/978-3-540-77688-8_2

[16] Sun-Young Ihm, Ki-Eun Lee, Aziz Nasridinov, Jun-Seok Heo, and Young-Ho Park. 2014. Approximate convex skyline: A partitioned layer-based index for efficient processing top-k queries. *Knowl. Based Syst.* 61 (2014), 13–28. https://doi.org/10.1016/j.knosys.2014.01.022

[17] Tao Jiang, Bin Zhang, Dan Lin, Yunjun Gao, and Qing Li. 2015. Incremental evaluation of top-k combinatorial metric skyline query. *Knowl. Based Syst.* 74 (2015), 89–105. https://doi.org/10.1016/j.knosys.2014.11.009

[18] Samir Khuller, Anna Moss, and Joseph Seffi Naor. 1999. The budgeted maximum coverage problem. *Information processing letters* 70, 1 (1999), 39–45.

[19] Craig Knox, Mike Wilson, Christen M Klinger, Mark Franklin, Eponine Oler, Alex Wilson, Allison Pon, Jordan Cox, Na Eun (Lucy) Chin, Seth A Strawbridge, Marysol Garcia-Patino, Ray Kruger, Aadhavya Sivakumaran, Selena Sanford, Rahil Doshi, Nitya Khetarpal, Omolola Fatokun, Daphnee Doucet, Ashley Zubkowski, Dorsa Yahya Rayat, Hayley Jackson, Karxena Harford, Afia Anjum, Mahi Zakir, Fei Wang, Siyang Tian, Brian Lee, Jaanus Liigand, Harrison Peters, Ruo Qi (Rachel) Wang, Tue Nguyen, Denise So, Matthew Sharp, Rodolfo da Silva, Cyrella Gabriel, Joshua Scantlebury, Marissa Jasinski, David Ackerman, Timothy Jewison, Tanvir Sajed, Vasuk Gautam, and David S Wishart. 2023. DrugBank 6.0: the DrugBank Knowledgebase for 2024. *Nucleic Acids Research* 52, D1 (11 2023), D1265–D1275. https://doi.org/10.1093/nar/gkad976 arXiv:https://academic.oup.com/nar/article-pdf/52/D1/D1265/55042351/gkad976.pdf

[20] Roman Kontchakov and Egor V. Kostylev. 2016. On Expressibility of Non-Monotone Operators in SPARQL. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifteenth International Conference, KR 2016, Cape Town, South Africa, April 25-29, 2016*, Chitta Baral, James P. Delgrande, and Frank Wolter (Eds.). AAAI Press, 369–379. http://www.aaai.org/ocs/index.php/KR/KR16/paper/view/12889

[21] Tomas Lampo, Maria-Esther Vidal, Juan Danilow, and Edna Ruckhaus. 2011. To Cache or Not To Cache: The Effects of Warming Cache in Complex SPARQL Queries. In *On the Move to Meaningful Internet Systems: OTM 2011 - Confederated International Conferences: CoopIS, DOA-SVI, and ODBASE 2011, Hersonissos, Crete, Greece, October 17-21, 2011, Proceedings, Part II (Lecture Notes in Computer Science)*, Robert Meersman, Tharam S. Dillon, Pilar Herrero, Akhil Kumar, Manfred Reichert, Li Qing, Beng Chin Ooi, Ernesto Damiani, Douglas C. Schmidt, Jules White, Manfred Hauswirth, Pascal Hitzler, and Mukesh K. Mohania (Eds.), Vol. 7045. Springer, 716–733. https://doi.org/10.1007/978-3-642-25106-1_22

[22] Matteo Lissandrini, Katja Hose, and Torben Bach Pedersen. 2023. Example-Driven Exploratory Analytics over Knowledge Graphs. In *Proceedings 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece, March 28-31, 2023*, Julia Stoyanovich, Jens Teubner, Nikos Mamoulis, Evaggelia Pitoura, Jan Mühlig, Katja Hose, Sourav S. Bhowmick, and Matteo Lissandrini (Eds.). OpenProceedings.org, 105–117. https://doi.org/10.48786/EDBT.2023.09

[23] Matteo Lissandrini, Davide Mottin, Katja Hose, and Torben Bach Pedersen. 2022. Knowledge Graph Exploration Systems: are we lost?. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org. https://www.cidrdb.org/cidr2022/papers/p40-lissandrini.pdf

[24] Matteo Lissandrini, Davide Mottin, Themis Palpanas, and Yannis Velegrakis. 2018. X2Q: Your Personal Example-based Graph Explorer. *Proc. VLDB Endow.* 11, 12 (2018), 2026–2029. https://doi.org/10.14778/3229863.3236251

[25] Johannes Lorey and Felix Naumann. 2013. Caching and Prefetching Strategies for SPARQL Queries. In *The Semantic Web: ESWC 2013 Satellite Events - ESWC 2013 Satellite Events, Montpellier, France, May 26-30, 2013, Revised Selected Papers (Lecture Notes in Computer Science)*, Philipp Cimiano, Miriam Fernández, Vanessa López, Stefan Schlobach, and Johanna Völker (Eds.), Vol. 7955. Springer, 46–65. https://doi.org/10.1007/978-3-642-41242-4_5

[26] Sara Magliacane, Alessandro Bozzon, and Emanuele Della Valle. 2012. Efficient Execution of Top-K SPARQL Queries. In *The Semantic Web - ISWC 2012 - 11th International Semantic Web Conference, Boston, MA, USA, November 11-15, 2012, Proceedings, Part I (Lecture Notes in Computer Science)*, Philippe Cudré-Mauroux, Jeff Heflin, Evren Sirin, Tania Tudorache, Jérôme Euzenat, Manfred Hauswirth, Josiane Xavier Parreira, Jim Hendler, Guus Schreiber, Abraham Bernstein, and Eva Blomqvist (Eds.), Vol. 7649. Springer, 344–360. https://doi.org/10.1007/978-3-642-35176-1_22

[27] Theofilos Mailis, Yannis Kotidis, Stamatis Christoforidis, Evgeny Kharlamov, and Yannis E. Ioannidis. 2021. View Selection over Knowledge Graphs in Triple Stores. *Proc. VLDB Endow.* 14, 13 (2021), 3281–3294. https://doi.org/10.14778/3484224.3484227

[28] Stanislav Malyshev, Markus Krötzsch, Larry González, Julius Gonsior, and Adrian Bielefeldt. 2018. Getting the Most Out of Wikidata: Semantic Technology Usage in Wikipedia's Knowledge Graph. In *The Semantic Web - ISWC 2018 - 17th International Semantic Web Conference, Monterey, CA, USA, October 8-12, 2018, Proceedings, Part II (Lecture Notes in Computer Science)*, Denny Vrandecic, Kalina Bontcheva, Mari Carmen Suárez-Figueroa, Valentina Presutti, Irene Celino, Marta Sabou, Lucie-Aimée Kaffee, and Elena Simperl (Eds.), Vol. 11137. Springer, 376–394. https://doi.org/10.1007/978-3-030-00668-6_23

[29] Xiangfu Meng, Zong Min Ma, and Li Yan. 2008. Providing Flexible Queries over Web Databases. In *KES (2) (Lecture Notes in Computer Science)*, Vol. 5178. Springer, 601–606.

[30] Thomas Minier, Hala Skaf-Molli, and Pascal Molli. 2019. SaGe: Web Preemption for Public SPARQL Query Services. In *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, Ling Liu, Ryen W. White, Amin Mantrach, Fabrizio Silvestri, Julian J. McAuley, Ricardo Baeza-Yates, and Leila Zia (Eds.). ACM, 1268–1278. https://doi.org/10.1145/3308558.3313652

[31] Nikolaos Papailiou, Dimitrios Tsoumakos, Panagiotis Karras, and Nectarios Koziris. 2015. Graph-Aware, Workload-Adaptive SPARQL Query Caching. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 1777–1792. https://doi.org/10.1145/2723372.2723714

[32] Tara Safavi, Caleb Belth, Lukas Faber, Davide Mottin, Emmanuel Müller, and Danai Koutra. 2019. Personalized Knowledge Graph Summarization: From the

Cloud to Your Pocket. In *2019 IEEE International Conference on Data Mining, ICDM 2019, Beijing, China, November 8-11, 2019*, Jianyong Wang, Kyuseok Shim, and Xindong Wu (Eds.). IEEE, 528–537. https://doi.org/10.1109/ICDM.2019.00063

[33] Arnaud Soulet and Fabian M. Suchanek. 2019. Anytime Large-Scale Analytics of Linked Open Data. In *The Semantic Web - ISWC 2019 - 18th International Semantic Web Conference, Auckland, New Zealand, October 26-30, 2019, Proceedings, Part I (Lecture Notes in Computer Science)*, Chiara Ghidini, Olaf Hartig, Maria Maleshkova, Vojtech Svátek, Isabel F. Cruz, Aidan Hogan, Jie Song, Maxime Lefrançois, and Fabien Gandon (Eds.), Vol. 11778. Springer, 576–592. https://doi.org/10.1007/978-3-030-30793-6_33

[34] Claus Stadler, Muhammad Saleem, Qaiser Mehmood, Carlos Buil-Aranda, Michel Dumontier, Aidan Hogan, and Axel-Cyrille Ngonga Ngomo. 2024. LSQ 2.0: A linked dataset of SPARQL query logs. *Semantic Web* 15, 1 (2024), 167–189. https://doi.org/10.3233/SW-223015

[35] Heiner Stuckenschmidt. 2004. Similarity-Based Query Caching. In *Flexible Query Answering Systems, 6th International Conference, FQAS 2004, Lyon, France, June 24-26, 2004, Proceedings (Lecture Notes in Computer Science)*, Henning Christiansen, Mohand-Said Hacid, Troels Andreasen, and Henrik Legind Larsen (Eds.), Vol. 3055. Springer, 295–306. https://doi.org/10.1007/978-3-540-25957-2_24

[36] Fabian M. Suchanek, Mehwish Alam, Thomas Bonald, Lihu Chen, Pierre-Henri Paris, and Jules Soria. 2024. YAGO 4.5: A Large and Clean Knowledge Base with a Rich Taxonomy. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2024, Washington DC, USA, July 14-18, 2024*, Grace Hui Yang, Hongning Wang, Sam Han, Claudia Hauff, Guido Zuccon, and Yi Zhang (Eds.). ACM, 131–140. https://doi.org/10.1145/3626772.3657876

[37] Hernán Vargas, Carlos Buil-Aranda, Aidan Hogan, and Claudia López. 2019. RDF Explorer: A Visual SPARQL Query Builder. In *The Semantic Web - ISWC 2019 - 18th International Semantic Web Conference, Auckland, New Zealand, October 26-30, 2019, Proceedings, Part I (Lecture Notes in Computer Science)*, Chiara Ghidini, Olaf Hartig, Maria Maleshkova, Vojtech Svátek, Isabel F. Cruz, Aidan Hogan, Jie Song, Maxime Lefrançois, and Fabien Gandon (Eds.), Vol. 11778. Springer, 647–663. https://doi.org/10.1007/978-3-030-30793-6_37

[38] Giannis Vassiliou, Fanouris Alevizakis, Nikolaos Papadakis, and Haridimos Kondylakis. 2023. iSummary: Workload-Based, Personalized Summaries for Knowledge Graphs. In *The Semantic Web - 20th International Conference, ESWC 2023, Hersonissos, Crete, Greece, May 28 - June 1, 2023, Proceedings (Lecture Notes in Computer Science)*, Catia Pesquita, Ernesto Jiménez-Ruiz, Jamie P. McCusker, Daniel Faria, Mauro Dragoni, Anastasia Dimou, Raphaël Troncy, and Sven Hertling (Eds.), Vol. 13870. Springer, 192–208. https://doi.org/10.1007/978-3-031-33455-9_12

[39] Giannis Vassiliou, Georgia Troullinou, Nikolaos Papadakis, and Haridimos Kondylakis. 2021. WBSum: Workload-based Summaries for RDF/S KBs. In *SSDBM 2021: 33rd International Conference on Scientific and Statistical Database Management, Tampa, FL, USA, July 6-7, 2021*, Qiang Zhu, Xingquan Zhu, Yicheng Tu, Zichen Xu, and Anand Kumar (Eds.). ACM, 248–252. https://doi.org/10.1145/3468791.3468815

[40] Ruben Verborgh, Miel Vander Sande, Olaf Hartig, Joachim Van Herwegen, Laurens De Vocht, Ben De Meester, Gerald Haesendonck, and Pieter Colpaert. 2016. Triple Pattern Fragments: A low-cost knowledge graph interface for the Web. *J. Web Semant.* 37-38 (2016), 184–206. https://doi.org/10.1016/j.websem.2016.03.003

[41] Denny Vrandecic and Markus Krötzsch. 2014. Wikidata: a free collaborative knowledgebase. *Commun. ACM* 57, 10 (2014), 78–85. https://doi.org/10.1145/2629489

[42] Dong Wang, Lei Zou, and Dongyan Zhao. 2015. Top-k queries on RDF graphs. *Inf. Sci.* 316 (2015), 201–217. https://doi.org/10.1016/j.ins.2015.04.032

[43] Yuxiang Wang, Xiaoliang Xu, Qifan Hong, Jiahui Jin, and Tianxing Wu. 2021. Top-k star queries on knowledge graphs through semantic-aware bounding match scores. *Knowl. Based Syst.* 213 (2021), 106655. https://doi.org/10.1016/j.knosys.2020.106655

[44] Gregory Todd Williams and Jesse Weaver. 2011. Enabling Fine-Grained HTTP Caching of SPARQL Query Results. In *The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I (Lecture Notes in Computer Science)*, Lora Aroyo, Chris Welty, Harith Alani, Jamie Taylor, Abraham Bernstein, Lalana Kagal, Natasha Fridman Noy, and Eva Blomqvist (Eds.), Vol. 7031. Springer, 762–777. https://doi.org/10.1007/978-3-642-25073-6_48

[45] Gang Wu and Mengdong Yang. 2012. Improving SPARQL query performance with algebraic expression tree based caching and entity caching. *J. Zhejiang Univ. Sci. C* 13, 4 (2012), 281–294. https://doi.org/10.1631/jzus.C1101009

[46] ZhiBang Yang, Xu Zhou, Kenli Li, Guoqing Xiao, Yunjun Gao, and Keqin Li. 2019. Efficient processing of top k group skyline queries. *Knowl. Based Syst.* 182 (2019). https://doi.org/10.1016/j.knosys.2019.06.003

[47] Wei Emma Zhang, Quan Z. Sheng, Kerry Taylor, and Yongrui Qin. 2015. Identifying and Caching Hot Triples for Efficient RDF Query Processing. In *Database Systems for Advanced Applications - 20th International Conference, DASFAA 2015, Hanoi, Vietnam, April 20-23, 2015, Proceedings, Part II (Lecture Notes in Computer Science)*, Matthias Renz, Cyrus Shahabi, Xiaofang Zhou, and Muhammad Aamir Cheema (Eds.), Vol. 9050. Springer, 259–274. https://doi.org/10.1007/978-3-319-18123-3_16