Accelerating SPARQL Queries by Exploiting Hash-based Locality and Adaptive Partitioning

Razen Harbi · Ibrahim Abdelaziz · Panos Kalnis · Nikos Mamoulis · Yasser Ebrahim · Majed Sahli

Abstract State-of-the-art distributed RDF systems partition data across multiple computer nodes (workers). Some systems perform cheap hash partitioning, which may result in expensive query evaluation. Others try to minimize inter-node communication, which requires an expensive data pre-processing phase, leading to a high startup cost. Apriori knowledge of the query workload has also been used to create partitions, which however are static and do not adapt to workload changes.

In this paper, we propose AdPart, a distributed RDF system, which addresses the shortcomings of previous work. First, AdPart applies lightweight partitioning on the initial data, that distributes triples by hashing on their subjects; this renders its startup overhead low. At the same time, the locality-aware query optimizer of AdPart takes full advantage of the partitioning to (i) support the fully parallel processing of join patterns on subjects and (ii) minimize data communication for general queries by applying hash distribution of intermediate results instead of broadcasting, wherever possible. Second, AdPart monitors the data access patterns and dynamically redistributes and replicates the instances of the most frequent ones among workers. As a result, the communication cost for future queries is drastically reduced or even eliminated. To control replication, AdPart implements an eviction policy for the redistributed patterns. Our experiments with synthetic and real data verify that AdPart: (i) starts faster than all existing systems; (ii) processes thousands of queries before other systems become online; and (iii) gracefully adapts to the query load, being able to evaluate queries on billion-scale RDF data in sub-seconds.

R. Harbi · I. Abdelaziz · P. Kalnis · M. Sahli
King Abdullah University of Science & Technology, Thuwal, Saudi Arabia
E-mail: {first}.{last}@kaust.edu.sa

N. Mamoulis
University of Ioannina, Greece
E-mail: nikos@cs.uoi.gr

Y. Ebrahim
Microsoft Corporation, Redmond, WA 98052, United States
E-mail: yaelsa@microsoft.com

1 Introduction

The RDF data model does not require a predefined schema and represents information from diverse sources in a versatile manner. Therefore, social networks, search engines, shopping sites and scientific databases are adopting RDF for publishing web content. Large public knowledge bases, such as Bio2RDF[1] and YAGO[2] have billions of facts in RDF format. RDF datasets consist of triples of the form $\langle$`subject`, *predicate*, `object`$\rangle$, where *predicate* represents a relationship between two entities: a `subject` and an `object`. An RDF dataset can be regarded as a long relational table with three columns. An RDF dataset can also be viewed as a directed labeled graph, where vertices and edge labels correspond to entities and predicates, respectively. Figure 1 shows an example RDF graph of an academic network.

SPARQL[3] is the standard query language for RDF. Each query is a set of RDF triple patterns; some of the nodes in a pattern are variables which may appear in multiple patterns. For example, the query in Figure 2(a) returns all professors who work for CS with their advisees. The query corresponds to the graph pattern in Figure 2(b). The answer is the set of ordered bindings of $(?p, ?s)$ that render the query graph isomorphic to subgraphs in the data. Assuming the data is stored in a table $D(s, p, o)$, the query can be answered by first

[1] http://www.bio2rdf.org/
[2] http://yago-knowledge.org/
[3] http://www.w3.org/TR/rdf-sparql-query/

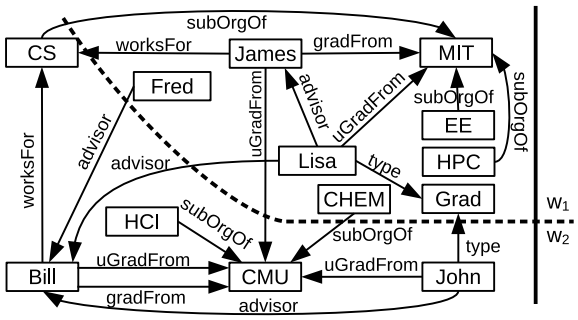
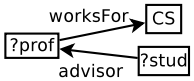


Fig. 1 Example RDF graph. An edge and its associated vertices correspond to an RDF triple; e.g., ⟨`Bill`, *worksFor*, `CS`⟩.

```
SELECT ?prof ?stud WHERE {
 ?prof    worksFor  CS .
 ?stud    advisor   ?prof .
}
```

(a) SPARQL



(b) Graph

Fig. 2 A query that finds CS professors with their advisees.

decomposing it into two subqueries, each corresponding to a triple pattern: $q_1 \equiv \sigma_{p=worksFor \wedge o=CS}(D)$ and $q_2 \equiv \sigma_{p=advisor}(D)$. The subqueries can be answered independently by scanning table D; then, we can join their intermediate results on the subject and object attribute: $q_1 \bowtie_{q_1.s=q_2.o} q_2$. By applying the query on the data of Figure 1, we get `(?prof, ?stud)` $\in$ {`(James, Lisa),(Bill, John), (Bill, Fred),(Bill, Lisa)`}.

Early research efforts on RDF data management resulted in efficient centralized RDF systems; like RDF-3X [25], HexaStore [33], TripleBit [36] and gStore [40]. However, centralized data management does not scale well for complex queries on web-scale RDF data. As a result, distributed RDF management systems were introduced to scale-out by partitioning RDF data among many compute nodes (workers) and evaluating queries in a distributed fashion. A SPARQL query is decomposed into multiple subqueries that are evaluated by each node independently. Since data is distributed, the nodes may need to exchange intermediate results during query evaluation. Consequently, queries with large intermediate results incur high communication cost, which is detrimental to the query performance [16,19].

Distributed RDF systems aim at minimizing the number of decomposed subqueries by partitioning the data among workers. The goal is that each node has all the data it needs to evaluate the entire query and there is no need for exchanging intermediate results. In such a *parallel* query evaluation, each node contributes a partial result of the query; the final query result is the union of all partial results. To achieve this, some triples may need to be replicated across multiple partitions. For example, in Figure 1, assume the data graph is divided by the dotted line into two partitions and assume that triples follow their subject placement. To answer the query in Figure 2, nodes have to exchange intermediate results because triples ⟨`Lisa`, *advisor*, `Bill`⟩ and ⟨`Fred`, *advisor*, `Bill`⟩ cross the partition boundary. Replicating these triples to both partitions allows each node to answer the query without communication. Still, even sophisticated partitioning and replication cannot guarantee that arbitrarily complex SPARQL queries can be processed in parallel; thus, expensive *distributed* query evaluation, with intermediate results exchanged between nodes, cannot always be avoided.

Challenges. Existing distributed RDF systems are facing two limitations. (*i*) *Partitioning cost:* balanced graph partitioning is an NP-complete problem [22]; thus, existing systems perform heuristic partitioning. In systems that use simple hash partitioning heuristics [17, 26,29,38], queries have low chances to be evaluated in parallel without communication between nodes. On the other hand, systems that use sophisticated partitioning heuristics [16,19,23,34] suffer from high preprocessing cost and sometimes high replication. More importantly, they pay the cost of partitioning the entire data regardless of the anticipated workloads. However, as shown in a recent study [28], only a small fraction of the whole graph is accessed by typical real query workloads. For example, a real workload consisting of more than 1,600 queries executed on DBpedia (459M triples) touches only 0.003% of the whole data. Thus, we argue that distributed RDF systems should leverage query workloads in data partitioning. (*ii*) *Adaptivity:* WARP [18] and Partout [13] do consider the workload during data partitioning and achieve significant reduction in the replication ratio, while showing better query performance compared to systems that partition the data blindly. Nonetheless, both these systems assume a representative (i.e., *static*) query workload and do not adapt to changes. Aluç et al. [1] showed that systems need to continuously adapt to workloads in order to consistently provide good performance.

In this paper, we propose AdPart, a distributed in-memory RDF engine. AdPart alleviates the aforementioned limitations of existing systems by capitalizing on the following key principles:

Lightweight Initial Partitioning: AdPart uses an initial hash partitioning that distributes triples by hashing on their subjects. This partitioning has low cost and does not incur any replication. Thus, the preprocessing time is low, partially addressing the first challenge.

Hash-based Locality Awareness: AdPart exploits hash-based locality to process in parallel (i.e., without data communication) the join patterns on subjects included in a query. In addition, intermediate results can potentially be hash-distributed to single workers

instead of being broadcasted everywhere. The locality-aware query optimizer of AdPart considers these properties to generate an evaluation plan that minimizes intermediate results shipped between workers.

Adapting by Incremental Redistribution: A hierarchical heat-map of accessed data patterns is maintained by AdPart to monitor the executed workload. Hot patterns are redistributed and potentially replicated in the system in a way that future queries that include them are executed in parallel by all workers without data communication. To control replication, AdPart operates within a budget and employs an eviction policy for the redistributed patterns. By adapting dynamically to the workload, AdPart overcomes the limitations of static partitioning schemes.

In summary, our contributions are:

- We introduce AdPart, a distributed SPARQL engine that does not require expensive preprocessing. By using lightweight hash partitioning, avoiding the upfront cost, and adopting a pay-as-you-go approach, AdPart executes tens of thousands of queries on large graphs within the time it takes other systems to conduct their initial partitioning.
- We propose a locality-aware query planner and a cost-based optimizer for AdPart to efficiently execute queries that require data communication.
- We propose the monitoring and indexing workloads in the form of hierarchical heat maps. Queries are transformed and indexed using these maps to facilitate the adaptivity of AdPart. We introduce an Incremental ReDistribution (IRD) technique for data portions that are accessed by hot patterns, guided by the workload. IRD helps processing future queries without data communication.
- We evaluate AdPart using synthetic and real data and compare with state-of-the-art systems. AdPart partitions billion-scale RDF data and starts up in less than 14 minutes, while other systems need hours or days. On billion-scale RDF data, AdPart executes complex queries in sub-seconds and processes large workloads orders of magnitude faster than existing approaches.

The rest of the paper is organized as follows. Section 2 reviews existing distributed RDF systems. Section 3 presents the architecture of AdPart and provides an overview of the system's components. Section 4 discuses our locality-aware query planning and distributed query evaluation, whereas Section 5 explains the adaptivity feature of AdPart. Section 6 contains the experimental results and Section 7 concludes the paper.

2 Related Work

In this section, we review recent distributed RDF systems related to AdPart. Table 1, summarizes the main characteristics of these systems.

Lightweight Data Partitioning: Several systems are based on the MapReduce framework [8] and use the Hadoop Distributed File System (HDFS), which applies horizontal random data partitioning. SHARD [29] stores the whole RDF data in one HDFS file. Similarly, HadoopRDF [20] uses HDFS but splits the data into multiple smaller files. SHARD and HadoopRDF solve SPARQL queries using a set of MapReduce iterations.

Trinity.RDF [38] is a distributed in-memory RDF engine that can handle web scale RDF data. It represents RDF data in its native graph form (i.e., using adjacency lists) and uses a key-value store as the backend store. The RDF graph is partitioned using vertex id as hash key. This is equivalent to partitioning the data twice; first using subjects as hash keys and second using objects. Trinity.RDF uses graph exploration for SPARQL query evaluation and relies heavily on its underlying high-end InfiniBand interconnect. In every iteration, a single subquery is explored starting from valid bindings by all workers. This way, generation of redundant intermediate results is avoided. However, because exploration only involves two vertices (source and target), Trinity.RDF cannot prune invalid intermediate results without carrying all their historical bindings. Hence, workers need to ship candidate results to the master to finalize the results, which is a potential bottleneck of the system.

Rya [27] and H2RDF+ [26] use key-value stores for RDF data storage which range-partition the data based on keys such that the keys in each partition are sorted. When solving a SPARQL query, Rya executes the first subquery using range scan on the appropriate index; it then utilizes index lookups for the next subqueries. H2RDF+ executes simple queries in a centralized fashion, whereas complex queries are solved using a set of MapReduce iterations.

All the above systems use lightweight partitioning schemes, which are computationally inexpensive; however, queries with long paths and complex structures incur high communication costs. In addition, systems that evaluate joins using MapReduce suffer from its high overhead [16, 34]. Although our AdPart system also uses lightweight hash partitioning, it avoids excessive data shuffling by exploiting hash-based data locality. Furthermore, it adapts incrementally to the workload to further minimize communication.

Table 1 Summary of state-of-the-art distributed RDF systems

System	Partitioning Strategy	Partitioning Cost	Replication	Workload Awareness	Adaptive
TriAD-SG [16]	Graph-based (METIS) & Horizontal triple Sharding	High	Yes	No	No
H-RDF-3X [19]	Graph-based (METIS)	High	Yes	No	No
Partout [13]	Workload-based horizontal fragmentation	High	No	Yes	No
SHAPE [23]	Semantic Hash	High	Yes	No	No
Wu et al. [34]	End-to-end path partitioning	Moderate	Yes	No	No
TriAD [16]	Hash-based triple Sharding	Low	Yes	No	No
Trinity.RDF [38]	Hash	Low	Yes	No	No
H2RDF+ [26]	H-Base partitioner (range)	Low	No	No	No
SHARD [29]	Hash	Low	No	No	No
AdPart	Hash	Low	Yes	Yes	Yes

Sophisticated Partitioning Schemes and Replication: Several systems employ general graph partitioning techniques for RDF data, in order to improve data locality. EAGRE [39] transforms the RDF graph into a compressed entity graph that is partitioned using a MinCut algorithm, such as METIS [22]. H-RDF-3X [19] uses METIS to partition the RDF graph among workers. It also enforces the so-called k-hop guarantee so any query with radius k or less can be executed without communication. Other queries are executed using expensive MapReduce joins. Replication increases exponentially with k; thus, k must be small (e.g., $k \leq 2$ in [19]). Both EAGRE and H-RDF-3X suffer from the significant overhead of MapReduce-based joins for queries that cannot be evaluated locally. For such queries, sub-second query evaluation is not possible [16], even with state-of-the-art MapReduce implementations, like Hadoop++ [9] and Spark [37].

TriAD [16] employs lightweight hash partitioning based on both subjects and objects. Since partitioning information is encoded into the triples, TriAD has full locality awareness of the data and processes large number of concurrent joins without communication. However, because TriAD shards one (both) relations when evaluating distributed merge (hash) joins, the locality of its intermediate results is not preserved. Thus, if the sharding column of the previous join is not the current join column, intermediate results need to be resharded. The cost becomes significant for large intermediate results with multiple attributes. TriAD-SG [16] uses METIS for data partitioning. Edges that cross partitions are replicated, resulting in $1-hop$ guarantee. A summary graph is defined, which includes a vertex for each partition. Vertices in this graph are connected by the cross-partition edges. A query in TriAD-SG is evaluated against the summary graph first, in order to prune partitions that do not contribute to query results. Then, the query is evaluated on the RDF data residing in the partitions retrieved from the summary graph. Multiple join operators are executed concurrently by all workers, which communicate via an asynchronous message passing protocol. Sophisticated partitioning techniques, like MinCut, reduce the communication cost significantly. However, such techniques are prohibitively expensive and do not scale for large graphs, as shown in [23]. Furthermore, MinCut does not yield good partitioning for dense graphs. Thus, TriAD-SG does not benefit from the summary graph pruning technique in dense RDF graphs because of the high edge-cut. To alleviate METIS overhead, an efficient approach for partitioning large graphs was introduced [32]. However, queries that cross partition boundaries result in poor performance.

SHAPE [23] is based on a semantic hash portioning approach for RDF data. It starts by simple hash partitioning and employs the same k-hop strategy as H-RDF-3X [19]. It also relies on URI hierarchy, for grouping vertices to increase data locality. Similar to H-RDF-3X, SHAPE suffers from the high overhead of MapReduce-based joins. Furthermore, URI-based grouping results in skewed partitioning if a large percentage of vertices share prefixes. This behavior is noticed in both real as well as synthetic datasets (See Section 6).

Recently, Wu et al. [34] proposed an end-to-end path partitioning scheme, which considers all possible directed paths in the RDF graph. These paths are merged in a bottom-up fashion. While this approach works well for star, chain and directed cyclic queries, other types of queries result in significant communication. For example, queries with object-object joins or queries that do not associate each query vertex with the type predicate require inter-worker communication. Note that our adaptivity technique (Section 5) is orthogonal to and can be combined with end-to-end path partitioning as well as other partitioning heuristics to efficiently evaluate queries that are not favored by the partitioning.

Workload-Aware Data Partitioning: Most of the aforementioned partitioning techniques focus on minimizing communication without considering the workload. Partout [13] is a workload-aware distributed RDF engine. First, it extracts representative triple patterns from the query load. Then, it uses these patterns to partition the data into fragments and collocates data

fragments that are accessed together by queries in the same worker. Similarly, WARP [18] uses a representative query workload to replicate frequently accessed data. Partout and WARP adapt only by applying expensive re-partitioning of the entire data; otherwise, they incur high communication costs for dynamic workloads. On the contrary, our system adapts incrementally by replicating only the data accessed by the workload which is expected to be small [28].

SPARQL on Vertex-centric: Sedge [35] solves the problem of dynamic graph partitioning and demonstrates its partitioning effectiveness using SPARQL queries over RDF. The entire graph is replicated several times and each replica is partitioned differently. Every SPARQL query is translated manually into a Pregel [24] program and is executed against the replica that minimizes communication. Still, this approach incurs excessive replication, as it duplicates the entire data several times. Moreover, its lack of support for ad-hoc queries makes it counter-productive; a user needs to manually write an optimized query evaluation program in Pregel.

Materialized views: Several works attempt to speed up the execution of SPARQL queries by materializing a set of views [6,15] or a set of path expressions [10]. The selection of views is based on a representative workload. Our approach does not generate local materialized views. Instead, we redistribute the data accessed by hot patterns in a way that preserves data locality and allows queries to be executed with minimal communication.

Relational Model: Relevant systems exist that focus on data models other than RDF. Schism [7] deals with data placement for distributed OLTP RDBMS. Using a sample workload, Schism minimizes the number of distributed transactions by populating a graph of co-accessed tuples. Tuples accessed in the same transaction are put in the same server. This is not appropriate for SPARQL because some queries access large parts of the data that would overload a single machine. Instead, AdPart exploits parallelism by executing such a query across all machines in parallel without communication. H-Store [31] is an in-memory distributed RDBMS that uses a data partitioning technique similar to ours. Nevertheless, H-Store assumes that the schema and the query workload are given in advance and assumes no ad-hoc queries.

Eventual indexing: Idreos et al. [21] introduced the concept of reducing the data-to-query time for relational data. They avoid building indices during data loading; instead, they reorder tuples incrementally during query processing. In AdPart, we extend eventual indexing to dynamic and adaptive graph partitioning. In our problem, graph partitioning is very expensive; hence, the potential benefits of minimizing the data-to-query time are substantial.

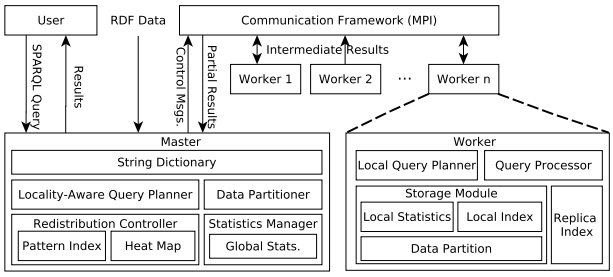


Fig. 3 System architecture of AdPart

3 System Architecture

Overview: AdPart employs the typical master-slave paradigm and is deployed on a shared-nothing cluster of machines (see Figure 3). This architecture is used by other systems, e.g., Trinity.RDF [38] and TriAD [16]. AdPart uses the standard Message Passing Interface (MPI) [12] for master-worker communication. In a nutshell, the master begins by encoding the data and partitioning it among workers. Each worker loads its triples and collects local statistics. Then, the master aggregates these statistics and becomes ready for answering queries. Each query is submitted to the master, which decides whether the query can be executed in *parallel* or *distributed* mode. In parallel mode, the query is evaluated concurrently by all workers without communication. Queries in distributed mode are also evaluated by all workers but require communication. AdPart monitors the submitted queries in the form of a heat map to detect hot patterns. Once such a pattern is detected, AdPart redistributes and potentially replicates the data accessed by the pattern among workers. Consequently, AdPart adapts to the query load and can answer more queries in parallel mode.

3.1 Master

String Dictionary. RDF data contain long strings in the form of URIs and literals. To avoid the storage, processing, and communication overheads, we follow the common practice [16,25,26,38] and encode RDF strings into numerical IDs and build a bi-directional dictionary.
Data Partitioner. A recent study [14] showed that 60% of the joins in a real workload of SPARQL queries are on the subject column. Hence, AdPart uses lightweight node-based partitioning using subject values. Given W workers, a triple t is assigned to worker w_i, where i is

the result of a hash function applied on $t.subject$.[4] This way all triples that share the same subject go to the same worker. Consequently, any star query joining on subjects can be evaluated without communication cost. We do not hash on objects because they can be literals and common types; this would assign all triples of the same type to one worker, resulting in load imbalance and limited parallelism [19].

Statistics Manager. It maintains statistics about the RDF graph, which are used for global query planning and during adaptivity. Statistics are collected in a distributed manner during bootstrapping.

Redistribution Controller. It monitors the workload in the form of heat maps and triggers the adaptive Incremental ReDistribution (IRD) process for hot patterns. Data accessed by hot patterns are redistributed and potentially replicated among workers. A redistributed hot pattern can be answered by all workers in parallel without communication. Replicated hot patterns are indexed in a structure called Pattern Index (PI). Patterns in the PI can be combined for evaluating future queries without communication. Further, the controller implements replica replacement policy to keep replication within a threshold (Section 5).

Locality-Aware Query Planner. Our planner uses the global statistics from the statistics manager and the pattern index from the redistribution controller to decide if a query, in whole or partially, can be processed without communication. Queries that can be fully answered without communication are planned and executed by each worker independently. On the other hand, for queries that require communication, the planner exploits the hash-based data locality and the query structure to find a plan that minimizes communication and the number of distributed joins (Section 4).

Failure Recovery. The master does not store any data but can be considered as a single-point of failure because it maintains the dictionaries, global statistics, and PI. A standard failure recovery mechanism (log-based recovery [11]) can be employed by AdPart. Assuming stable storage, the master can recover by loading the dictionaries and global statistics because they are read-only and do not change in the system. The PI can be recovered by reading the query log and reconstructing the heat map. Workers on the other hand store data; hence, in case of a failure, data partitions need to be recovered. Shen et al. [30] proposes a fast failure recovery solution for distributed graph processing systems. The solution is a hybrid of checkpoint-based and log-based recovery schemes. This approach can be used by AdPart to recover worker partitions and reconstruct the replica index. Reliability is outside the scope of this paper and we leave it for future work.

[4] For simplicity, we use: $i = t.subject \mod W$.

3.2 Worker

Storage Module. Each worker w_i stores its local set of triples D_i in an in-memory data structure, which supports the following search operations, where s, p, and o are subject, predicate, and object:

1. given p, return set $\{(s, o) \mid \langle s, p, o\rangle \in D_i\}$.
2. given s and p, return set $\{o \mid \langle s, p, o\rangle \in D_i\}$.
3. given o and p, return set $\{s \mid \langle s, p, o\rangle \in D_i\}$.

Since all the above searches require a known predicate, we primarily hash triples in each worker by predicate. The resulting *predicate* index (simply P-index) immediately supports search by predicate (i.e., the first operation). Furthermore, we use two hash maps to repartition each bucket of triples having the same predicate, based on their subjects and objects, respectively. These two hash maps support the second and third search operation and they are called *predicate-subject* index (PS-index) and *predicate-object* index (PO-index), respectively. Given the number of unique predicates is typically small, our storage scheme avoids unnecessary repetitions of predicate values. Note that when answering a query, if the predicate itself is a variable, then we simply iterate over all predicates. Our indexing scheme is tailored for typical RDF knowledge bases and their workloads, being orthogonal to the rest of the system (i.e., alternative schemes, like indexing all SPO combinations [25] could be used at each worker). Finally, the storage module computes statistics about its local data and shares them with the master after data loading.

Replica Index. Each worker has an in-memory *replica index* that stores and indexes replicated data as a result of the adaptivity. This index initially contains no data and is updated dynamically by the incremental redistribution (IRD) process (Section 5).

Query Processor. Each worker has a query processor that operates in two modes: (*i*) *Distributed Mode* for queries that require communication. In this case, the locality-aware planner of the master devises a global query plan. Each worker gets a copy of this plan and evaluates the query accordingly. Workers solve the query concurrently and exchange intermediate results (Section 4.1). (*ii*) *Parallel Mode* for queries that can be answered without communication. In this case, the master broadcasts the query to all workers. Each worker has all the data needed for query evaluation; therefore it generates a local query plan using its local statistics and executes the query without communication.

Table 2 Matching result of q_1 on workers w_1 and w_2.

w_1

?prof
James

w_2

?prof
Bill

Table 3 The final query results $q_1 \bowtie q_2$ on both workers.

w_1

?prof	?stud
James	Lisa

w_2

?prof	?stud
Bill	Lisa
Bill	John
Bill	Fred

Local Query Planner. Queries executed in parallel mode are planned by workers autonomously. For example, star queries joining on the subject are processed in parallel due to the initial partitioning. Moreover, queries answered in parallel after the adaptivity process are also planned by local query planners.

4 Query Evaluation

A basic SPARQL query consists of multiple subquery triple patterns: $q_1, q_2, \ldots, q_n$. Each subquery includes variables or constants, some of which are used to bind the patterns together, forming the entire query graph (e.g., see Figure 2(b)). A query with n subqueries requires the evaluation of $n-1$ joins. Since data are memory resident and hash-indexed, we favor hash joins as they prove to be competitive to more sophisticated join methods [3]. Our query planner devises an ordering of these subqueries and generates a left-deep join tree, where the right operand of each join is a base subquery (not an intermediate result). We do not use bushy tree plans to avoid building indices for intermediate results.

4.1 Distributed Query Evaluation

In AdPart, triples are hash partitioned among many workers based on subject values. Consequently, subject star queries (i.e., all subqueries join on the subject column) can be evaluated locally in parallel without communication. However, for other types of queries, workers may have to communicate intermediate results during join evaluation. For example, consider the query in Figure 2 and the partitioned data graph in Figure 1. The query consists of two subqueries q_1 and q_2, where:

- q_1: ⟨`?prof`, *worksFor*, `CS`⟩
- q_2: ⟨`?stud`, *advisor*, `?prof`⟩

The query is evaluated by a single subject-object join. However, neither of the workers has all the data needed for evaluating the entire query; thus, workers need to communicate. For such queries, AdPart employs the Distributed Semi-Join (DSJ) algorithm. Each worker scans the PO-index to find all triples matching q_1. The results on workers w_1 and w_2 are shown in Table 2. Then, each worker creates a projection on the join column $?prof$ and exchanges it with the other worker. Once the projected column is received, each worker computes the semi-join $q_1 \ltimes_{?prof} q_2$ using its PO-index. Specifically, w_1 probes $p =$ `advisor`, $o =$ `Bill` while w_2 probes $p =$ `advisor`, $o =$ `James` to their PO-index. Note that workers also need to evaluate semi-joins using their local projected column. Then, the semi-join results are shipped to the sender. In this case, w_1 sends ⟨`Lisa`, *advisor*, `Bill`⟩ and ⟨`Fred`, *advisor*, `Bill`⟩ to w_2; no candidate triples are sent from w_2 because `James` has no advisees on w_2. Finally, each worker computes the final join $q_1 \bowtie_{?prof} q_2$. The final query results at both workers are shown in Table 3.

Table 4 The final query results $q_2 \bowtie q_1$ on both workers.

w_1

?prof	?stud
James	Lisa
Bill	Lisa
Bill	Fred

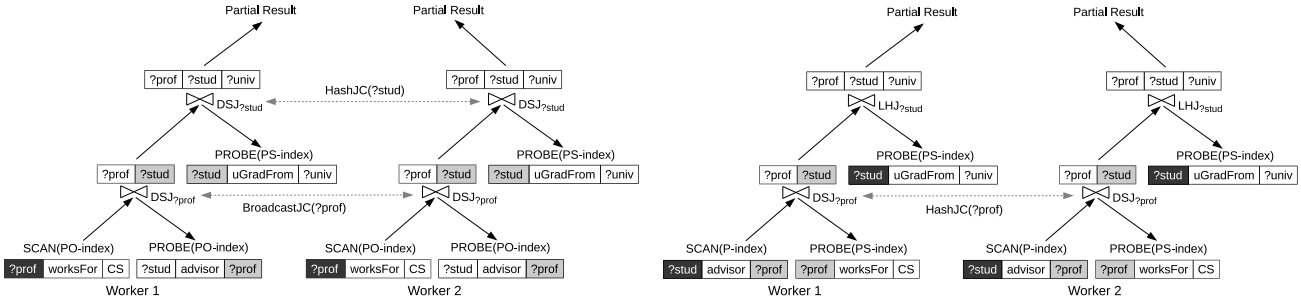
w_2

?prof	?stud
Bill	John

4.1.1 Hash-based data locality

Observation 1 *DSJ can benefit from subject hash locality to minimize communication. If the join column of the right operand is subject, the projected column of the left operand is hash distributed by all workers, instead of being broadcast to every worker.*

In our example, since the join column of q_2 is the object column ($?prof$), each worker sends the entire join column to the other worker. However, based on Observation 1, communication can be minimized if the join order is reversed (i.e., $q_2 \bowtie q_1$). In this case, each worker scans the P-index to find triples matching q_2 and creates a projection on $?prof$. Then, because $?prof$ is the subject of q_1, both workers exploit the subject hash-based locality by partitioning the projection column and communicating each partition to the respective worker, as opposed to broadcasting the entire projection column to all workers. Consequently, w_1 sends `Bill` to only w_2 because of Bill's hash value. The final query results are shown in Table 4. Notice that the final results are the same for both query plans; however, the results reported by each worker are different.



(a) q_1, q_2, q_3 (b) q_2, q_1, q_3

Fig. 4 Executing query Q_{prof} using two different subquery orderings.

4.1.2 Pinned subject

Observation 2 *Under the subject hash partitioning, combining right-deep tree planning and the DSJ algorithm, causes the intermediate and final results to be local to the subject of the first executed subquery pattern p_1. We refer to this subject as* `pinned_subject`.

In our example, executing q_1 first causes ?*prof* to be the *pinned_subject* because it is the subject of q_1. Hence, the intermediate and final results are local (pinned) to the bindings of ?*prof*, `James` and `Bill` in w_1 and w_2, respectively. Changing the order by executing q_2 first made ?*stud* to be the *pinned_subject*. Accordingly, the results are pinned at the bindings of ?*stud*.

AdPart leverages Observations 1 and 2 to minimize communication and synchronization overhead. To see this, consider Q_{prof} which extends the query in Figure 2 with one more triple pattern, namely q_3: ⟨`?stud`, *uGradFrom*, `?univ`⟩. Assume Q_{prof} is executed in the following order: q_1, q_2, q_3. The query execution plan is pictorially shown in Figure 4(a). The results of the first join (i.e., $q_1 \bowtie q_2$) are shown in Table 3 (?*prof* is the *pinned_subject*). The query continues by joining the results of ($q_1 \bowtie q_2$) with q_3 on ?*stud*, the subject of q_3. Both workers project the intermediate results on ?*stud* and hash distribute the bindings of ?*stud* (Observation 1). Then, all workers evaluate semi-joins with q_3 and return the candidate triples to the other workers where the final query results are formulated.

Notice that the execution order q_1, q_2, q_3 requires communication for evaluating both joins. A better ordering is q_2, q_1, q_3. The execution plan is shown in Figure 4(b). The first join (i.e., $q_2 \bowtie q_1$) already proved to incur less communication by avoiding broadcasting the entire projection column. The result of this join is pinned at ?*stud* as shown in Table 4. Since the join column of q_3 (?*stud*) is the *pinned_subject*, joining ($q_2 \bowtie q_1$) with q_3 can be processed by each worker without communication using Local Hash Join (LHJ). Therefore, the ordering of the subqueries affects the amount of communication incurred during query execution.

Table 5 Communication cost for different join types

Subject Pinning	SS	SO/OO	OS
Pinned	No Communication	Broadcast	Direct Communication
Unpinned	Direct Communication	Broadcast	Direct Communication

4.1.3 The four cases of a join

Formally, joining two subqueries, say p_i (possibly an intermediate pattern) and p_j, has four possible scenarios: the first three assume that p_i and p_j join on columns c_1 and c_2, respectively. (*i*) If c_2 = *subject* AND c_2 = *pinned_subject*, then the join is processed in parallel without communication. (*ii*) If c_2 = *subject* AND $c_2 \neq$ *pinned_subject*, then the join is evaluated using DSJ, but the projected join column of p_i is hash distributed. (*iii*) If $c_2 \neq$ *subject*, then the join is executed using DSJ and the projected join column of p_i is sent from all workers to all other workers. Finally, (*iv*) if p_i and p_j join on multiple columns, we opt to join on the subject column of p_j, if it is a join attribute. This allows the join column of p_i to be hash distributed as in *(ii)*. If the subject column of p_j is not a join attribute, the projection column is broadcast to all workers, as in *(iii)*. Verifying on the other columns is carried out during the join finalization. Table 5 summarizes these scenarios.

Based on the above four scenarios, we introduce our Locality-Aware Distributed Query Execution algorithm (see Algorithm 1). The algorithm receives an ordering of the subquery patterns. For each join iteration, if the second subquery joins on the pinned subject, the join is executed without communication (line 7). Otherwise, the join is evaluated by the DSJ algorithm (lines 9-28). In the first iteration, p_1 is a base subquery pattern; however, for the subsequent iterations, p_1 is a pattern of intermediate results. If p_1 is the first subquery to be matched, each worker finds the local matching of p_1 (line 2) and projects on the join column c_1 (line 5). If

Algorithm 1: Locality-Aware Distributed Execution

```
   Input: Query Q with n ordered subqueries {q1, q2, ... qn}
   Result: Answer of Q
1  p1 ← q1;
2  pinned_subject ← p1.subject;
3  for i ← 2 to n do
4      p2 ← qi;
5      [c1, c2]← getJoinColumns(p1, p2);
6      if c2 == pinned_subject AND c2 is subject then
7          p1 ← JoinWithoutCommunication (p1, p2, c1, c2);
8      else
9          if p1 NOT intermediate pattern then
10             RS1 ← answerSubquery(p1);
11         else
12             RS1 is the result of the previous join
13         RS1[c1] ← πc1(RS1); // projection on c1
14         if c2 is subject then
15             Hash RS1[c1] among workers;
16         else
17             Send RS1[c1] to all workers;
18         Let RS2 ← answerSubquery(p2);
19         foreach worker w, w : 1 → N do
20             // RS1w[c1] is the RS1[c1] received from w
21             // CRS2w are candidate triples of RS2 that
               join with RS1w[c1]
22             CRS2w ← RS1w[c1] ⋈RS1w[c1].c1=RS2.c2 RS2;
23             Send CRS2w to worker w;
24         foreach worker w, w : 1 → N do
25             // RS2w is the CRS2w received from worker w
               // RESw is the result of joining with worker w
               RESw ← RS1 ⋈RS1.c1=RS2w.c2 RS2w;
26         p1 ← RES1 ∪ RES2 ∪ .... ∪ RESN;
```

the join column of q_2 is subject, then each worker hash distributes the projected column (line 7); or sends it to all other workers otherwise (line 9). To avoid the overhead of synchronization, communication is carried out using non-blocking MPI routines. All workers perform semi-join on the received data (line 14) and send the results back to w (line 15). Finally, each worker finalizes the join (line 19) and formulates the final result (line 20). Lines 14 and 19 are implemented as local hash-joins using the local index in each worker. The result of a DSJ iteration becomes p_1 in the next iteration.

Algorithm 1 can solve star queries that join on the subject in parallel mode. Traditionally, the planning is done by the master using global statistics. We argue that allowing each worker to plan the query execution autonomously would result in a better performance. For example, using the data graph in Figure 1, Table 6 shows triples that match the following star query:

- q_1: ⟨?s, *advisor*, ?p⟩
- q_2: ⟨?s, *uGradFrom*, ?u⟩

Any global plan (i.e., $q_1 \bowtie q_2$ or $q_2 \bowtie q_1$) would require a total of four index lookups to solve the join. However, w_1 and w_2 can evaluate the join using 2 and 1 index lookup(s), respectively. Therefore, to solve such queries, the master sends the query to all workers; each worker utilizes its local statistics to formulate the execution plan, evaluates the query locally without communication, and sends the final result to the master.

Table 6 Triples matching ⟨?s, *advisor*, ?p⟩ and ⟨?s, *uGradFrom*, ?u⟩ on two workers.

Worker 1

advisor	?s	?p
	Fred	Bill
	Lisa	Bill
	Lisa	James

uGradFrom	?s	?u
	Lisa	MIT
	James	CMU

Worker 2

advisor	?s	?p
	John	Bill

uGradFrom	?s	?u
	Bill	CMU
	John	CMU

4.2 Locality-Aware Query Optimization

Our locality-aware planner leverages the query structure and hash-based data distribution during query plan generation to minimize communication. Accordingly, the planner uses a cost-based optimizer, based on Dynamic Programming (DP), for finding the best subquery ordering (the same approach is used by other systems [16,25,38]). Each state S in DP is identified by a connected subgraph ϱ of the query graph. A state can be reached by different orderings on ϱ. Thus, we maintain in each state the ordering that results in the least estimated communication cost ($S.cost$). We also keep estimated cardinalities of the variables in the query. Furthermore, instead of maintaining the cardinality of the state, we keep the cumulative cardinality of all intermediate results that led to this state. (Although the cardinality of the state will be the same regardless of the ordering, different orderings result in different cumulative cardinalities.)

We initialize a state S for each subquery pattern (subgraph of size 1) p_i. $S.cost$ is initially zero because a query with a single pattern can be answered without communication. Then, we expand the subgraph by joining with another pattern p_j, leading to a new state S' such that:

$$S'.cost = min(S'.cost,\ S.cost + cost(S, p_j))$$

If we reach a state using different orderings with the same cost, we keep the one with the least cumulative cardinality. This happens for subqueries that join on the *pinned_subject*. To minimize the DP table size, we maintain a global minimum cost ($minC$) of all found plans. Because our cost function is monotonically increasing, any branch that results in a cost $> minC$ is pruned. Moreover, because of Observation 1, we start the DP process by considering subqueries connected to

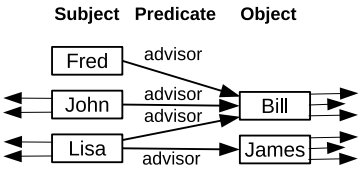


Fig. 5 Statistics calculation for p=*advisor*, based on Figure 1.

the subject with the highest number of outgoing edges; this increases the chances for converging to the optimal plan faster. The space complexity of the DP table is $O(s)$ where s is the number of connected subgraphs in the query graph. Since each state can be extended by multiple edges, the number of updates applied to the DP table (i.e., the time complexity) is $O(sE)$, where E is the number of edges in the query graph.

4.3 Cost Estimation

We first describe the statistics used for cost calculation. Recall that AdPart collects and aggregates statistics from workers for global query planning and during the adaptivity process. Keeping statistics about each vertex in the RDF data graph is too expensive. Therefore, we focus on predicates rather than vertices; this way the storage complexity of statistics is linear to the number of unique predicates, which is typically small compared to the data size. For each unique predicate p, we calculate the following: (i) The *cardinality* of p, denoted as $|p|$, is the number of triples in the data graph that have p as predicate. (ii) $|p.s|$ and $|p.o|$ are the numbers of *unique subjects and objects* using predicate p, respectively. (iii) The *subject* score of p, denoted as $\overline{p_S}$, is the average degree of all vertices s, such that $\langle \mathtt{s}, p, \mathtt{?x} \rangle \in D$. (iv) The *object* score of p, denoted as $\overline{p_O}$, is the average degree of all vertices o, such that $\langle \mathtt{?x}, p, \mathtt{o} \rangle \in D$. (v) *Predicates Per Subject* $P_{ps} = |p|/|p.s|$ is the average number of triples with predicate p per unique subject. (vi) *Predicates Per Object* $P_{po} = |p|/|p.o|$ is the average number of triples with predicate p per unique object.

For example, Figure 5 illustrates the computed statistics for predicate *advisor* using the data graph of Figure 1. Since *advisor* appears four times with three unique subjects and two unique objects, $|p| = 4$, $|p.s| = 3$ and $|p.o| = 2$. The subject score $\overline{p_S}$ is $(1+3+4)/3 = 2.67$ because *advisor* appears with four unique subjects: Fred, John and Lisa, whose degrees (i.e., in-degree plus out-degree) are 1, 3 and 4, respectively. Similarly, $\overline{p_O} = (6 + 4)/2 = 5$. Finally, the number of predicates per subject P_{ps} is $4/3 = 1.3$ because Lisa is associated with two instances of the predicate (i.e., two advisors).

We set the initial communication cost of DP states to zero. Cardinalities of subqueries with variable subjects and objects are already captured in the master's global statistics. Hence, we set the cumulative cardinalities of the initial states to the cardinalities of the subqueries and set the size of the subject and object bindings to $|p.s|$ and $|p.o|$. Furthermore, the master consults the workers to update the cardinalities of subquery patterns that are attached to constants or have unbounded predicates. This is done locally at each worker by simple lookups to its PS- and PO- indices to update the cardinalities of variables bindings accordingly.

We estimate the cost of expanding a state S with a subquery p_j, where c_j and P are the join column and the predicate of p_j, respectively. If the join does not incur communication, the cost of the new state S' is zero. Otherwise, the expansion is carried out through DSJ and we incur two phases of communication: (i) transmitting the projected join column and (ii) replying with the candidate triples. Estimating the communication in the first phase depends on the cardinality of the join column bindings in S, denoted as $B(c_j)$. In the second phase, communication depends on the selectivity of the semi-join and the number of variables ν in p_j (constants are not communicated). Moreover, if c_j is the subject column of p_j, we hash distribute the projected column. Otherwise, the column needs to be sent to all workers. The cost of expanding S with p_j is:

$$cost(S, p_j) = \begin{cases} 0, \text{if } c_j \text{ is subject \& } c_j = pinned\_subject \\ \\ S.B(c_j) + (\nu \cdot S.B(c_j) \cdot P_{ps}), \\ \text{if } c_j \text{ is subject \& } c_j \neq pinned\_subject \\ \\ (S.B(c_j) \cdot N) + (\nu \cdot N \cdot S.B(c_j) \cdot P_{po}), \\ \text{if } c_j \text{ is not subject} \end{cases}$$

Next, we need to re-estimate the cardinalities of all variables $\overline{v} \in p_j$. Let $|p.\overline{v}|$ denote $|p.s|$ or $|p.o|$ if $\overline{v}$ is subject or object, respectively. Similarly, let $P_{p\overline{v}}$ denote $|P_{ps}|$ if $\overline{v}$ is subject or $|P_{po}|$ if $\overline{v}$ is object. We re-estimate the cardinality of $\overline{v}$ in the new state S' as:

$$S'.B(\overline{v}) = \begin{cases} min(S.B(\overline{v}), |P|), & \text{if } \nu = 1 \\ min(S.B(\overline{v}), |p.\overline{v}|), & \text{if } \overline{v} = c_j \ \& \ \nu > 1 \\ min(S.B(\overline{v}), S.B(\overline{v}) \cdot P_{p\overline{v}}, |p.\overline{v}|), & \text{if } \overline{v} \neq c_j \ \& \ \nu > 1 \end{cases}$$

We use cumulative cardinality when we reach the same state by two different orderings. Thus, we also re-estimate the cumulative state cardinality $|S'|$. If P_{pc_j} denotes $|P_{ps}|$ or $|P_{po}|$ depending on the position of c_j, $|S'| = |S| \cdot (1 + P_{pc_j})$. Note that we use an upper bound estimation for cardinalities. A special case of the last equation is when a subquery has a constant; then, we assume that each tuple in the previous state connects to

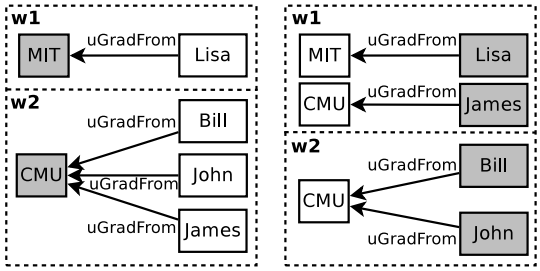
this constant by setting P_{pc_j}=1. Note that a more accurate cardinality estimation like the one used in Trinity.RDF [38] is orthogonal to our optimizer.

5 AdPart Adaptivity

Studies show that even minimal communication results in significant performance degradation [19,23]. Thus, data should be redistributed to minimize, if not eliminate, communication and synchronization overheads. AdPart redistributes only the parts of data needed for the current workload and adapts as the workload changes. The incremental redistribution model of AdPart is a combination of hash partitioning and k-hop replication, guided by the query load rather than the data itself. Specifically, given a hot pattern Q (hot pattern detection is discussed in Section 5.4), our system selects a special vertex in the pattern called the *core* vertex (Section 5.1). The system groups the data accessed by the pattern around the bindings of this core vertex. To do so, the system transforms the pattern into a redistribution tree rooted at the core (Section 5.2). Then, starting from the core vertex, first hop triples are hash distributed based on the core bindings. Next, triples that match the second level subqueries are collocated and so on (Section 5.3). AdPart utilizes redistributed patterns to answer queries in parallel without communication.

5.1 Core Vertex Selection

For a hot pattern, the choice of the core vertex has a significant impact on the amount of replicated data as well as on query execution performance. For example, consider query $Q_1 = \langle$`?stud`, $uGradFrom$, `?univ`$\rangle$. Assume there are two workers, w_1 and w_2, and refer to the graph of Figure 1; MIT and CMU are the bindings of ?*univ*, whereas Lisa, John, James and Bill bind to ?*stud*. Assume that ?*univ* is the core, then triples matching Q_1 will be hashed on the bindings of ?*univ* as shown in Figure 6(a). Note that every binding of ?*stud* appears in one worker only. Now assume that ?*stud* is the core and triples are hashed using the bindings of ?*stud*. This causes binding ?*univ*=CMU to exist on both workers (see Figure 6(b)). The problem becomes more pronounced when the query has more triple patterns. Consider $Q_2 = Q_1$ `AND` $\langle$`?prof`, $gradFrom$, `?univ`$\rangle$ and assume that ?*stud* is chosen as core. Because CMU exists on both workers, all its graduates (i.e., triples matching $\langle$`?prof`, $gradFrom$, `CMU`$\rangle$ will also be replicated. Replication grows exponentially with the number of triple patterns [19].



(a) Core is ?*univ* (b) Core is ?*stud*

Fig. 6 Effect of choice of core on replication. In (a) there is no replication. In (b) CMU is both workers.

Intuitively, if random walks start from two random vertices (e.g., students), the probability of reaching the same well-connected vertex (e.g., university) within a few hops is higher compared to other nodes. In order to minimize replication, we must avoid reaching the same vertex when starting from the core. Hence, it is reasonable to select a well-connected vertex as the core. Although, well-connected vertices can be identified by complex data mining algorithms in the literature, for the sake of minimizing the computational cost, we employ a simple approach. We assume that connectivity is proportional to degree centrality (i.e., in-degree plus out-degree edges). Recall from Section 4.3 that we maintain statistics $\overline{p_S}$ and $\overline{p_O}$ for each predicate $p \in P$, where P is the set of all predicates in the data. Let P_s and P_o be the set of all $\overline{p_S}$ and $\overline{p_O}$, respectively. We filter out predicates with extremely high scores and consider them outliers.[5] Outliers are detected using Chauvenet's criterion [4] on P_s then P_o. If a predicate p is detected as an outlier, we set: $\overline{p_S} = \overline{p_O} = -\infty$; otherwise we use $\overline{p_S}$ and $\overline{p_O}$ as computed in Section 4.3. Now, we can compute a score for each vertex in the query as follows:

Definition 1 (Vertex score) *For a query vertex v, let $E_{out}(v)$ be the set of outgoing edges and $E_{in}(v)$ be the set of incoming edges. Also, let A be the set of all $\overline{p_S}$ for the $E_{out}(v)$ edges and all $\overline{p_O}$ for $E_{in}(v)$ edges. The vertex score $\overline{v}$ is defined as: $\overline{v} = \max(A)$.*

Figure 7 shows an example for vertex score assignment. For vertex ?*prof*, E_{in}(?*prof*) = {`advisor`} and E_{out}(?*prof*) = {`gradFrom`}. Both predicates (i.e., advisor and gradFrom) contribute a score of 5 to ?*prof*. Therefore, $\overline{?prof} = 5$.

Definition 2 (Core vertex) *Given a query graph $G = (V, E)$ such that V and E are the set of vertices and*

[5] In many RDF datasets, vertex degrees follow a power-law distribution, where few ones have extremely high degrees. For example, vertices that appear as objects in triples with *rdf:type* have very high degree centrality. Treating such vertices as cores results in imbalanced partitions and prevents the system from taking full advantage of parallelism [19].

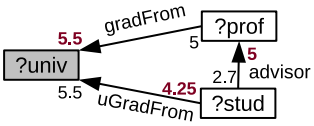


Fig. 7 Example of vertex score: numbers correspond to $\overline{p_S}$ and $\overline{p_O}$ values. Assigned vertex scores $\overline{v}$ are shown in bold.

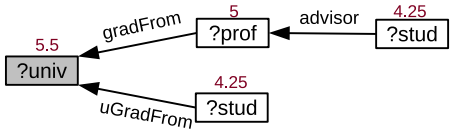


Fig. 8 The query in Figure 7 transformed into a tree using Algorithm 2. Numbers near vertices define their scores. The shaded vertex is the core.

Algorithm 2: Pattern Transformation

Input: $G = \{V, E\}$; a vertex-weighted, undirected graph, the core vertex v'
Result: The redistribution tree T

```
1  Let edges be a priority queue of pending edges
2  Let verts be a set of pending vertices
3  Let core_edges be all incident edges to v'
4  visited[v'] = true;
5  T.root=v';
6  foreach e in core_edges do
7      edges.push(v', e.nbr, e.pred);
8      verts.insert(e.nbr);
9      T.add(v', e.pred, e.nbr);
10 while edges notEmpty do
11     (parent, vertex, predicate) ← edges.pop();
12     visited[vertex] = true;
13     verts.remove(vertex);
14     foreach e in vertex.edges do
15         if e.nbr NOT visited then
16             if e.nbr ∉ verts then
17                 edges.push(vertex, e.nbr, e.pred);
18                 verts.insert(e.nbr);
19                 T.add(vertex, e.pred, e.nbr);
20             else
21                 T.add(vertex, e.pred, duplicate(e.nbr));
```

edges, respectively. Let $f(v)$ be a scoring function that assigns a score to each $v \in V$. We define the core vertex of Q as v' such that $f(v') = \max_{v \in V} f(v)$.

In Figure 7, ?*univ* has the highest score, hence, it is the core vertex for this pattern.

5.2 Generating the Redistribution Tree

Let Q be a hot pattern that AdPart decides to redistribute and let D_Q be the data accessed by this pattern. Our goal is to redistribute (partition) D_Q among all workers such that D_Q can be evaluated without communication. Unlike previous work that performs static MinCut-based partitioning [22], we eliminate the edge cuts by replicating edges that cross partitions. Since the balanced partitioning is an NP-complete problem, we introduce a heuristic for partitioning D_Q with two objectives in mind: (i) the redistribution of D_Q should benefit Q as well as other patterns. (ii) Because replication is necessary for eliminating communication, redistributing D_Q should result in minimal replication.

To address the first objective, we transform the pattern Q into a tree T by breaking cycles and duplicating some vertices in the cycles. The reason is that cycles constrain the data grouped around the core to be also cyclic. For example, the query pattern in Figure 7 retrieves students who share the same alma mater with their advisors. Grouping the data around universities without removing the cycle is not useful for retrieving professors and their advisees who do not share the same university. Consequently, the pattern in Figure 7 can be transformed into a tree by breaking the cycle and duplicating the ?*stud* vertex as shown in Figure 8. We refer to the result of the transformation as *redistribution tree*.

Our goal is to construct the redistribution tree that minimizes the expected amount of replication. In Section 5.1, we explained why starting from the vertex with the highest score has the potential to minimize replication. Intuitively, the same idea applies recursively to each level of the redistribution i.e., every child node in the tree has a lower score than its parent. Obviously, this cannot be always achieved; for example in a path pattern where a lower score vertex comes between two high score vertices. Therefore, we use a greedy algorithm for transforming a hot pattern Q into a redistribution tree T. Specifically, using the scoring function discussed in the previous section, we first transform Q into a vertex weighted, undirected graph G, where each node has a score and the directions of edges in Q are disregarded. The vertex with the highest score is selected as the core vertex. Then, G is transformed into the redistribution tree using Algorithm 2.

Algorithm 2 is a modified version of the Breadth-First-Search (BFS) algorithm, which has the following differences: (i) unlike BFS trees which span all vertices in the graph, our tree spans all edges in the graph. Each of the edges in the query graph should appear exactly once in the tree while vertices may be duplicated. (ii) During traversal, vertices with high scores are identified and explored first (using a priority queue). Since our traversal needs to span all edges, elements in the priority queue are stored as edges of the form (*parent*, *vertex*, *predicate*). These elements are ordered based on the vertex score first then on the edge label (predicate). Since the exploration does not follow the traditional BFS ordering, we maintain a pointer to the parent so edges can be inserted properly in the tree. As

Table 7 Triples from Figure 1 matching patterns in Figure 8.

Worker 1		Worker 2	
t_1	⟨Lisa, $uGradFrom$, MIT⟩	t_3	⟨Bill, $uGradFrom$, CMU⟩
		t_4	⟨James, $uGradFrom$, CMU⟩
		t_5	⟨John, $uGradFrom$, CMU⟩
t_2	⟨James, $gradFrom$, MIT⟩	t_6	⟨Bill, $gradFrom$, CMU⟩
t_7	⟨Lisa, $advisor$, James⟩	t_8	⟨Fred, $advisor$, Bill⟩
		t_9	⟨John, $advisor$, Bill⟩
		t_{10}	⟨Lisa, $advisor$, Bill⟩

an example, consider the query in Figure 7. Having the highest score, *?univ* is chosen as core, and the query is transformed into the tree shown in Figure 8. Note that the nodes have weights (scores) and the directions of edges have been moved back.

5.3 Incremental Redistribution

Incremental ReDistribution (IRD) aims at redistributing data accessed by hot patterns among all workers in a way that eliminates communication while achieving high parallelism. Given a redistribution tree, AdPart distributes the data along paths from the root to leaves using depth first traversal. The algorithm has two phases. First, it distributes triples containing the core vertex to workers using hash function $\mathcal{H}(\cdot)$. Let t be such a triple and let $t.core$ be its core vertex (the core can be either the subject or the object of t). Let $w_1, \ldots, w_N$ be the workers. t will be hash-distributed to worker w_j, where $j = \mathcal{H}(t.core) \mod N$. Note that if $t.core$ is a subject, t will not be replicated by IRD because of the initial subject-based hash partitioning.

In Figure 8, consider the first-hop triple patterns ⟨?prof, $uGradFrom$, ?univ⟩ and ⟨?stud, $gradFrom$, ?univ⟩. The core $?univ$ determines the placement of t_1-t_6 (see Table 7). Assuming two workers, t_1 and t_2 are hash-distributed to w_1 (because of MIT), whereas t_3-t_6 are hash-distributed to w_2 (because of CMU). The objects of triples t_1-t_5 are called their *source* columns.

Definition 3 (Source column) *The source column of a triple (subject or object) determines its placement.*

The second phase of IRD places triples of the remaining levels of the tree in the workers that contain their parent triples, through a series of distributed semijoins. The column at the opposite end of the source column of the previous step becomes the *propagating* column, i.e., $?prof$ in our previous example.

Definition 4 (Propagating column) *The propagating column of a triple is its object (resp. subject) if the source column of the triple is its subject (resp. object).*

At the second level of the redistribution tree in Figure 8, the only subquery pattern is ⟨?stud, $advisor$, ?prof⟩. The propagating column $?prof$ from the previous level becomes the source column for the current pattern. Triples $t_{7...10}$ in Table 7 match the sub-query and are joined with triples $t_{1...6}$. Accordingly, t_7 is placed in worker w_1, whereas t_7, t_9 and t_{10} are sent to w_2.

Algorithm 3: Incremental Redistribution

```
   Input: P = {E}; a path of consecutive edges, C is the core
          vertex.
   Result: Data replicated along path P
   // hash-distributing the first (core-adjacent) edge
 1 if e_0 is not replicated then
 2     coreData = getTriplesOfSubQuery(e_0);
 3     foreach t in coreData do
 4         m = B(C) mod N; // N is the number of workers
 5         sendToWorker(t, m);
   // then collocate triples from other levels
 6 foreach i : 1 → |E| do
 7     if e_i is not replicated then
 8         candidTriples = DSJ(e_0, e_i);
 9         IndexCandidateTriples(candidTriples);
10     e_0 = e_i;
```

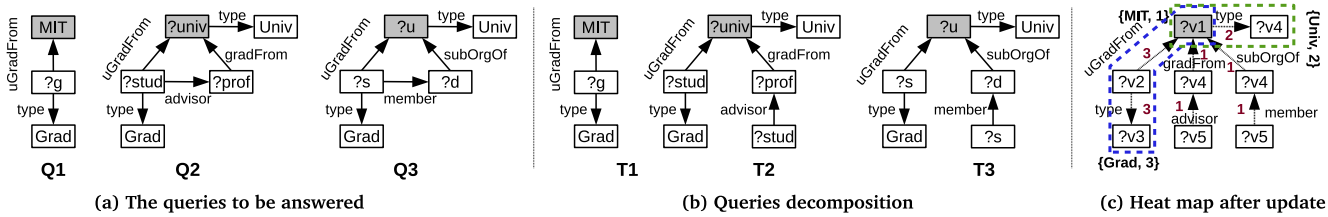
The IRD process is formally described in Algorithm 3. For brevity, we describe the algorithm on a path input since we follow depth-first traversal. The algorithm runs in parallel on all workers. Lines 1-5 hash distribute triples that contain the core vertex $\mathcal{C}$, if necessary.[6] Then, triples of the remaining levels are localized (replicated) in the workers that contain their parent. Replication is avoided for each triple which is already in the worker. This is carried out through a series of DSJ (lines 6-10). We maintain candidate triples at each level rather than final join results. Managing replicas in raw triple format allows us to utilize the RDF indices when answering queries using replicated data.

5.4 Queryload Monitoring

To effectively monitor workloads, systems face the following challenges: (*i*) the same query pattern may occur with different constants, subquery orderings, and variable names. Therefore, queries in the workload need to be deterministically transformed into a representation that unifies similar queries. (*ii*) This representation needs to be updated incrementally with minimal overhead. Finally, (*iii*) monitoring should be done at the level of patterns not whole queries. This allows the system to identify common hot patterns among queries.

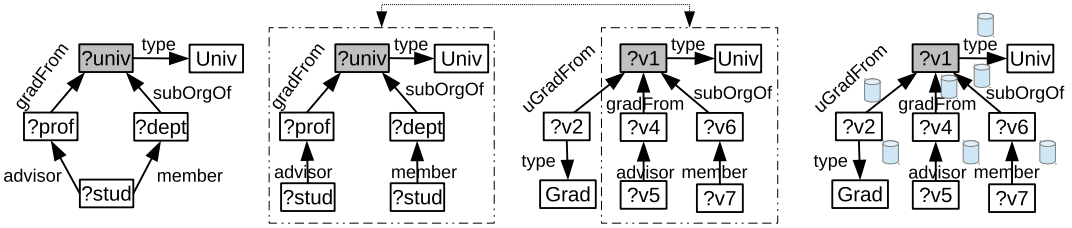
Heat map. We introduce a hierarchical heat map representation to monitor workloads. The heat map is main-

[6] Recall if a core vertex is a subject, we do not redistribute.



(a) The queries to be answered (b) Queries decomposition (c) Heat map after update

Fig. 9 Updating the heat map. Selected areas indicate hot patterns.



(a) Query (b) Redistribution Tree (c) Pattern Index (d) Replica Index

Fig. 10 A query and the pattern index that allows execution without communication.

tained by the redistribution controller. Each query Q is first decomposed into a redistribution tree T using Algorithm 2 (see Section 5.2), with the core vertex as root. To detect overlap among queries, we transform T to a tree template $\mathcal{T}$ in which all the constants are replaced with variables. To avoid losing information about constant bindings in the workload, we store the constants and their frequencies as meta-data in the template vertices. After that, $\mathcal{T}$ is inserted in the heat map which is a prefix-tree like structure that includes and combines the tree templates of all queries. Insertion proceeds by traversing the heat map from the root and matching edges in $\mathcal{T}$. If the edge does not exist, we insert a new edge in the heat map and set the edge count to 1; otherwise, we increment the edge count. Furthermore, we update the meta-data of vertices in the heat map with the meta-data in $\mathcal{T}$'s vertices. For example, consider queries Q_1, Q_2 and Q_3 and their decompositions T_1, T_2 and T_3, respectively in Figure 9(a) and (b). Assume that each of the queries is executed once. The state of the heat map after executing these queries is shown in Figure 9(c). Every inserted edge updates the edge count and the vertex meta-data in the heat map. For example, edge $\langle ?v_2, uGradFrom, ?v_1\rangle$ has edge count 3 because it appears in all $\mathcal{T}$'s. Furthermore, $\{MIT, 1\}$ is added to the meta-data of v_1.

We now describe the implementation details of the heat map. We use a dual tree representation for storing the heat map, where a tree node corresponds to an entire triple pattern. An edge denotes the existence of a common variable between any combination of subjects and objects in the connected triples. Note that this representation results in a tree forest. Whenever no confusion arises, we simply refer to both representations as heat map. The root node of the heat map is a dummy node that is connected to all core-adjacent edges from all patterns seen before. Figure 11 shows the dual representation of the heat map in Figure 9(c).

Algorithm 4: Update Heat Map

```
   Input: HeatMap dual representation T_hm, query tree dual
          representation T_q
   Result: T_hm updated
1  foreach QueryNode N_q → T_q.root.childs do
2      updateFreq (T_hm.root, N_q);
3  Procedure updateFreq(HeatNode N_hm, QueryNode N_q)
5      newParent ← NULL;
6      newParent ← findNode (N_hm.children, N_q);
7      if newParent is NULL then
8          newParent ← N_hm.insert (N_q);
9          newParent.count ← 1;
10     else
11         newParent.count ++;
12     updateMetaData (newParent, N_q);
13     foreach QueryChild C_q → N_q.children do
14         updateFreq (newParent,C_q);
16     return;
```

To update the heat map given a query Q, the tree template $\mathcal{T}$ is also transformed into its dual representation. This typically results in multiple independent trees. The heat map is updated using the dual of $\mathcal{T}$ level by level in a depth first manner. Algorithm 4 shows how the heat map is updated with a new query tree. Initially, a search process is started from the heat map root for each node in the first level of the query tree (line 1-2). The algorithm calls a procedure which takes as input both the heat map node and the query node (lines 3-20). The find function (line 6) is used to match the query node in the current level of the heat map. Recall that triple patterns in the heat map and $\mathcal{T}$ have variable subjects and objects. Therefore, a heat map node

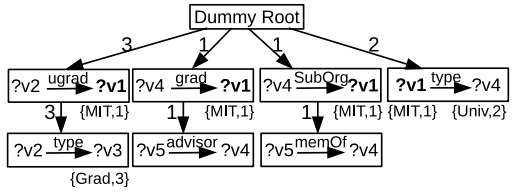


Fig. 11 Dual Tree Representation of the heat map shown in Figure 9(c).

matches the query node if they share the same predicate and direction. If no match is found, a new node is inserted in the heat map as a child of the current node (lines 7-9) with frequency 1. Otherwise, the count of the matched heat map node is incremented (lines 10-11). In both cases, we update the metadata (i.e., the occurrences of the target vertices and their frequencies) of the heat map node (line 12). Then, the procedure is recursively called for each child of the query node (lines 13-14). The find function is implemented using hash lookup based on the predicate and direction of the triple pattern. Hence, the complexity of updating the heat map is $O(|E|)$, where E is the number of edges in the query graph.

Hot pattern detection. The redistribution controller monitors queries by updating the heat map using Algorithm 4. Currently, we use a hardwired frequency threshold[7] for identifying hot patterns. Recall that while updating the heat map, we also update the frequency (count) of its nodes. A pattern in the heat map is considered to be hot if the update process makes its frequency greater than the threshold. As the heat map update process is carried out in a top-down fashion, we guarantee that a lower node in the heat map cannot have a frequency greater than its ancestors. Once a hot pattern is detected, the redistribution controller triggers the IRD process for that pattern. Recall that patterns in the heat map are templates in which all vertices are variables. To avoid excessive replication, some variables are replaced by dominating constants stored in the heat map. For example, assume the selected part of the heat map in Figure 9(c) is identified as hot. We replace vertex $?v_3$ with the constant Grad because it is the dominant value. On the other hand, $?v_1$ is not replaced by MIT because MIT does not dominate other values in query instances that include the hot pattern. We use the Boyer-Moore majority vote algorithm [5] for deciding the dominating constant.

[7] Auto-tuning the frequency threshold is a subject of our future work.

5.5 Pattern and Replica Index

Pattern index. The pattern index is created and maintained by the replication controller at the master. It has the same structure as the heat map, but it only stores redistributed patterns. For example, Figure 10(c) shows the pattern index state after redistributing all patterns in the heat map (Figure 9(c)). The pattern index is used by the query planner to check if a query can be executed without communication. When a new query Q is posed, the planner transforms Q into a tree T. If the root of T is also a root in the pattern index and all of T's edges exist in the pattern index, then Q can be answered in parallel mode; otherwise, Q is answered in distributed fashion. For example, the query in Figure 10(a) can be answered in parallel because its redistribution tree (Figure 10(b)) is contained in the pattern index. Edges in the pattern index are time-stamped at every access to facilitate our eviction policy.

Replica index. The replica index at each worker is identical to the pattern index at the master and is also updated by the IRD process. However, each edge in the replica index is associated with a storage module similar to the one that stores the original data. Each module stores only the replicated data of the specified triple pattern. In other words, we do not add the replicated data to the main indices nor keep all replicated data in a single index. There are four reasons for this segregation. (*i*) As more patterns are redistributed, updating a single index becomes a bottleneck. (*ii*) Because of replication, using one index mandates filtering duplicate results. (*iii*) If data is coupled in a single index, intermediate join results will be larger, which will affect performance. Finally, (*iv*) this hierarchical representation allows us to evict any part of the replicated data quickly without affecting the overall system performance. Notice that we do not replicate data associated with triple patterns whose subjects are core vertices. Such data are accessed from the main index directly because of the initial subject-based hash partitioning. Figure 10(d) shows the replica index that has the same structure as the pattern index in Figure 10(c). The storage module associated with $\langle$`?v7`, *member*, `?v6`$\rangle$ stores replicated triples that match the triple pattern. Moreover, these triples qualify for the join with the triple pattern of the parent edge.

Searching and updating the pattern and replica indices is carried in the same way as for the heat map (see Algorithm 4). However, the findNode function (line 6) is changed to account for triple patterns with bounded subject/objects. Such triple patterns can have at most two matches: (*i*) an exact match, where all constants

Table 8 Datasets Statistics in millions (M)

Dataset	Triples (M)	#S (M)	#O (M)	#S∩O (M)	#P	Indegree (Avg/StDev)	Outdegree (Avg/StDev)
LUBM-10240	1,366.71	222.21	165.29	51.00	18	16.54/26000.00	12.30/5.97
WatDiv	109.23	5.21	17.93	4.72	86	22.49/960.44	42.20/89.25
WatDiv-1B	1,092.16	52.12	179.09	46.95	86	23.69/2783.40	41.91/89.05
YAGO2	295.85	10.12	52.34	1.77	98	10.87/5925.90	56.20/71.96
Bio2RDF	4,287.59	552.08	1,075.58	491.73	1,714	8.64/21110.00	16.83/195.44

are matched; or (*ii*) a superset match, where both subject and object in the matching pattern are variables. If a triple pattern has two matches, the findNode function proceed with the superset matching branch because it will potentially benefit more queries in the future. This process is also implemented using hash lookups and hence has a complexity of $O(E)$, where E is the number of triple patterns in the query.

Conflicting Replication and Eviction. Conflicts may arise when a subquery appears at different levels in the pattern index. This may cause some triples to be replicated by the hot patterns that include them. This is not a correctness issue for AdPart as conflicting triples (if any) are stored separately using different storage modules. This approach avoids the burden of any housekeeping and existence of duplicates at the cost of memory consumption. Therefore, AdPart employs an *LRU* eviction policy that keeps the system within a given replication budget at each worker.

Recall that, each time an edge in the pattern index is accessed, its timestamp is updated. The search process in the pattern index is carried out in a top-down fashion. This means that the leaf nodes of the tree have the oldest timestamps. We store the leaves in a priority queue organized by timestamp. When eviction is required, the least recently used leaf and its matching replica index are deleted. Then, the parent of the evicted leaf is updated accordingly.

6 Experimental Evaluation

We evaluate AdPart against existing systems. We also include a non-adaptive version of our system, referred to as AdPart-NA, which does not include the features described in Section 5. In Section 6.1, we provide the details of the data, the hardware setup, and the competitors to our approach. In Section 6.2, we demonstrate the low startup and initial replication overhead of AdPart compared to all other systems. Then, in Section 6.3, we apply queries with different complexities on different datasets to show that (*i*) AdPart leverages the subject-based hash locality to achieve better or similar performance compared to other systems and (*ii*) the adaptivity feature of AdPart renders it several orders of magnitude faster than other systems. In Section 6.4, we conduct a detailed study of the effect and cost of AdPart's adaptivity feature. The results show that our system adapts incrementally to workload changes with minimal overhead without resorting to full data repartitioning. Finally, in Section 6.5, we study the data and machine scalability of AdPart.

6.1 Setup and Competitors

Datasets: We conducted our experiments using real and synthetic datasets of variable sizes. Table 8 describes these datasets, where #S, #P, and #O denote respectively the numbers of unique subjects, predicates, and objects. We use the synthetic LUBM[8] data generator to create a dataset of 10,240 universities consisting of 1.36 billion triples. LUBM and its template queries are used for testing most distributed RDF engines [16,23,26,38]. However, LUBM queries are intended for semantic inferencing and their complexities lie in semantics not structure. Therefore, we also use WatDiv[9] which is a recent benchmark that provides a wide spectrum of queries with varying structural characteristics and selectivity classes. We used two versions of this dataset: WatDiv (109 million) and WatDiv-1B (1 billion) triples. As both LUBM and WatDiv are synthetic, we also use two real datasets. YAGO2[10] is a real dataset derived from Wikipedia, WordNet and GeoNames containing 300 million triples. Bio2RDF[11] dataset provides linked data for life sciences and contains 4.64 billion triples connecting 24 different biological datasets. The details of all queries and workloads used in this section are available in the Appendix.

Hardware Setup: We implemented AdPart in C++ and used a Message Passing Interface library (MPICH2) for synchronization and communication. Unless otherwise stated, we deploy AdPart and its competitors on a cluster of 12 machines each with 148GB RAM and two 2.1GHz AMD Opteron 6172 CPUs (12 cores each). The machines run 64-bit 3.2.0-38 Linux Kernel and are connected by a 10Gbps Ethernet switch.

[8] http://swat.cse.lehigh.edu/projects/lubm/
[9] http://db.uwaterloo.ca/watdiv/
[10] http://yago-knowledge.org/
[11] http://download.bio2rdf.org/release/2/

Table 9 Partitioning Configurations

	LUBM-10240	WatDiv	Bio2RDF	YAGO2
SHAPE	2 forward	3 undirected	2 undirected	2 forward
H-RDF-3X	2 undirected	3 undirected	2 undirected	2 undirected

Competitors: We compare AdPart against TriAD [16], a recent in-memory RDF system, which is shown to have the fastest query response times to date. We compare to TriAD and TriAD-SG; the former uses lightweight hash partitioning while the later uses graph summaries for join-ahead pruning. We also compare against two Hadoop-based systems that employ lightweight partitioning: SHARD [29] and H2RDF+ [26]. Furthermore, we compare to two systems that rely on static replication by prefetching and use RDF-3X as underlying data store: SHAPE [23] and H-RDF-3X [19]. We configure SHAPE with full level semantic hash partitioning and enable the type optimization (see [23] for details). For H-RDF-3X, we enable the type and high degree vertices optimizations (see [19] for details). We compare with distributed systems only, because they outperform state-of-the-art centralized RDF systems.

6.2 Startup Time and Initial Replication

Our first experiment measures the time it takes all systems for preparing the data prior to answering queries. We exclude the string-to-id mapping time for all systems. For fair comparison, SHAPE and H-RDF-3X were configured to partition each dataset such that all its queries are processable without communication. Table 9 shows the details of these partitioning configurations. Using 2-hop forward guarantee for H-RDF-3X (which minimizes its replication [19]), we cannot guarantee that all queries can be answered without communication. This is mainly due to the high degree vertices optimization. For TriAD-SG, we used the same number of partitions reported in [16] for partitioning LUBM-10240 and WatDiv. Determining a suitable number of summary graph partitions requires empirical evaluation of some workload on the data or a representative sample. While generating a representative sample from these real data might is tricky, empirical evaluation on the original big data is costly [16]. Therefore, for fair comparison, we do not evaluate TriAD-SG on Bio2RDF and YAGO2.

As Table 10 shows, systems that rely on METIS for partitioning (i.e., H-RDF-3X and TriAD-SG) have significant startup cost. This is because METIS does not scale to large RDF graphs. To apply METIS, we had to remove all triples connected to literals; otherwise, METIS takes several days to partition LUBM-10240 and YAGO2. For LUBM-10240, SHAPE incurs less preprocessing time compared to METIS-based systems. However, for WatDiv and YAGO2, SHAPE performs worse because of data imbalance, causing some of the RDF-3X engines to take more time in building the databases. Partitioning YAGO2 and WatDiv using 2-hop forward and 3-hop undirected, respectively, placed all the data in a single partition. The reason is that all these datasets have uniform URI's, hence SHAPE could not utilize its semantic hash partitioning. SHAPE and H-RDF-3X did not finish partitioning Bio2RDF and were terminated after 24 hours.

Table 10 Preprocessing time (minutes)

	LUBM-10240	WatDiv	Bio2RDF	YAGO2
AdPart	**14**	**1.2**	**29**	**4**
TriAD	72	4	75	11
TriAD-SG	737	63	N/A	N/A
H-RDF-3X	939	285	>24h	199
SHAPE	263	79	>24h	251
SHARD	72	9	143	17
H2RDF+	152	9	387	22

Table 11 Initial replication

	LUBM-10240	WatDiv	YAGO2
SHAPE	42.9%	(1 worker) 0%	(1 worker) 0%
H-RDF-3X	19.5%	1090%	73.7%

SHARD and H2RDF+ employ lightweight partitioning, random and range-based, respectively. Therefore, they require less time compared to other systems. However, since they are Hadoop-based, they suffer from the overhead of storing the data first on Hadoop File System (HDFS) before building their data stores. TriAD and AdPart use lightweight hash partitioning and avoid the upfront cost of sophisticated partitioning schemes. As Table 10 shows, both systems start 4X up to two orders of magnitude faster than other systems. TriAD takes more time because it hash partitions the data twice (on the subject and object columns). Furthermore, TriAD requires extra time for sorting its indices and computing statistics.

Initial replication: We report only the initial replication of SHAPE and H-RDF-3x, since AdPart, TriAD, SHARD and H2RDF+ do not incur any initial replication (the replication caused by AdPart's adaptivity is evaluated in the next section). For LUBM-10240, H-RDF-3X results in the least replication (see Table 11) as LUBM is uniformly structured around universities (high degree vertices). Because of the high degree optimization, all entities of type university and their edges are removed before partitioning the graph using METIS. The resulting partitions are fully disconnected with zero edge cut. The extra replication is mainly because of the ownership triples used for duplicate elimination (see [19] for details). With full level semantic

Table 12 Query runtimes for LUBM-10240 (ms)

LUBM-10240	L1	L2	L3	L4	L5	L6	L7	Geo-Mean
AdPart	**317**	**120**	**6**	**1**	**1**	**4**	**220**	**15**
AdPart-NA	2,743	**120**	320	**1**	**1**	40	3,203	75
TriAD	6,023	1,519	2,387	6	4	114	17,586	369
TriAD-SG	5,392	1,774	4,636	9	5	10	21,567	333
SHAPE	25,319	4,387	25,360	1,603	1,574	1,567	15,026	5,575
H-RDF-3X	7,004	2,640	7,957	1,635	1,586	1,965	7,175	3,412
H-RDF-3X (in-memory)	6,841	2,597	7,948	1,596	1,594	1,926	7,551	3,397
H2RDF+	285,430	71,720	264,780	24,120	4,760	22,910	180,320	59,275
SHARD	413,720	187,310	aborted	358,200	116,620	209,800	469,340	261,362

hash partitioning and type optimization, SHAPE incurs almost double the replication of H-RDF-3X. For WatDiv, METIS produces very bad partitioning because of the dense nature of the data. Consequently, partitioning the whole data blindly using k-hop guarantee would result in excessive replication because of the high edge-cut. H-RDF-3X [19] replicated the data almost 11 times, i.e., each partition has almost the whole original graph. Because of the URI's uniformity of WatDiv and YAGO2, SHAPE places the data on a single partition. Therefore, it incurs no replication but performs as good as a single machine RDF-3X store.

6.3 Query Performance

In this section, we compare AdPart on individual queries against state-of-the-art distributed RDF systems. We demonstrate that even the AdPart-NA version of our system (which does not include the adaptivity feature) is competitive to systems that employ sophisticated partitioning techniques. This shows that the subject-based hash partitioning and the distributed evaluation techniques proposed in Section 4 are very effective. When AdPart adapts, its performance becomes even better and our system consistently outperforms its competitors by a wide margin.

LUBM dataset: In the first experiment (Table 12), we compare the performance of all systems using the LUBM-10240 dataset and queries L1-L7 defined in [2]. Queries L1-L7 can be classified based on their structure and selectivities into simple and complex. L4 and L5 are simple selective star queries whereas L2 is a simple yet non-selective star query that generates large final results. L6 is a simple query because it is highly selective. L1, L3 and L7 are complex queries with large intermediate results but very few final results.

SHARD and H2RDF+ suffer from the expensive overhead of MapReduce-based joins; hence, their performance is significantly worse than all other systems. On the other hand, SHAPE and H-RDF-3X perform better than SHARD and H2RDF+ because they do not require communication. H-RDF-3X performs better than SHAPE because it has less replication. However, as both SHAPE and H-RDF-3X use MapReduce for dispatching queries to workers, they still suffer from the non-negligible overhead of MapReduce (around 1.5 seconds on our cluster). Without this overhead, both systems would perform well for simple selective queries. Even for complex queries, these systems still perform reasonably well because queries run in parallel without any communication overhead. For example, for query L7 which requires excessive communication, H-RDF-3X and SHAPE perform better than TriAD and TriAD-SG. Note that this performance comes at a high preprocessing cost. Obviously, with a low hop guarantee, the preprocessing cost for SHAPE and H-RDF-3X is reduced but the query performance becomes worse because of the MapReduce-based joins [16]. AdPart and AdPart-NA outperform SHAPE and H-RDF-3X for three reasons: (*i*) managing the original and replicated data in the same set of indices results in large and duplicate intermediate results, rendering the cost of join evaluation higher. (*ii*) To filter out duplicate results, H-RDF-3X requires and additional join with the ownership triple pattern. (*iii*) TriAD and AdPart are designed as in-memory engines while RDF-3X is disk-based. For fairness, we also stored H-RDF-3X databases in a memory-mounted partition; still, it did not affect the performance significantly.

In-memory RDF engines, AdPart and TriAD, perform equally for queries L4 and L5 due to their high selectivities and star-shapes. AdPart exploits the initial hash distribution and solves these queries without communication, which explains why both versions of AdPart have the same performance. Similarly, L2 consists of a single subject-subject join; however, it is highly non-selective. TriAD solves the query by two distributed index scans (one for each base subquery) followed by a merge join. The merge join utilizes binary search for finding the beginning of the sorted runs from the left and right relations. These searches perform well for selective queries but not for L2. AdPart performs better than TriAD-SG by avoiding unnecessary scans.

Table 13 Query runtimes for WatDiv (ms)

WatDiv-100	Machines	L1-L5	S1-S7	F1-F5	C1-C3
AdPart	5	**2**	**2**	**7**	**22**
AdPart-NA	5	9	7	160	111
TriAD	5	4	15	45	170
SHAPE	12	1,870	1,824	1,836	2,723
H-RDF-3X	12	1,662	1,693	1,778	1,929
H2RDF+	12	5,441	8,679	18,457	65,786

Table 14 Query runtimes for YAGO2 (ms)

YAGO2	Y1	Y2	Y3	Y4	Geo-Mean
AdPart	**3**	**19**	**11**	**2**	**6**
AdPart-NA	19	46	570	77	79
TriAD	16	1,568	220	18	100
SHAPE	1,824	665,514	1,823	1,871	8,022
H-RDF-3X	1,690	246,081	1,933	1,720	6,098
H2RDF+	10,962	12,349	43,868	35,517	21,430
SHARD	238,861	238,861	aborted	aborted	238,861

In other words, utilizing its hash indexes and the right deep tree planning, AdPart requires a single scan followed by hash lookups. As a result, AdPart is faster than TriAD by more than an order of magnitude. The pruning technique of TriAD-SG eliminates the communication required for solving L6. Therefore, it outperforms TriAD and AdPart-NA. However, once AdPart adapts, L6 is executed without communication resulting in better performance than TriAD-SG, which incurs a slight overhead for summary plan generation.

Although AdPart-NA and TriAD have a similar partitioning scheme (with the difference in TriAD's full locality awareness on both subjects and objects), AdPart-NA achieves better performance than TriAD and TriAD-SG for all complex queries, L1, L3 and L7. There are three reasons for this: (*i*) When executing distributed merge/hash joins, TriAD needs to shard one/both relations among workers. On the contrary, AdPart only exchanges the unique values from the projected join column. The effect becomes more prominent in TriAD when concurrent joins at the lower level of the execution plan generate large and unnecessary intermediate results. These results need to be asynchronously sharded before executing joins at higher levels. (*ii*) AdPart exploits the subject-based locality and the locality of the intermediate results (pinning strategy) during planning to decide which join operators can run without communication regardless of their location in the execution tree. On the other hand, in TriAD, once a relation is sharded the locality of the intermediate results is destroyed which mandates further shardings at higher join operators. Finally, (*iii*) as in L2, if the join being executed is not selective, merge join performs worse than the hash joins used by AdPart-NA. The pruning technique of TriAD-SG was effective in reducing the overall slaves query execution time. However, the cost for summary graph processing in L3 and L7 was high; hence, the high query execution times compared to TriAD.

For L3, AdPart-NA is 7x to 14x faster than Triad and TriAD-SG, respectively. AdPart-NA evaluates the join that gives an empty intermediate result early, which avoids subsequent useless joins. However, the first few joins cannot be eliminated during query planning time. On the other hand, AdPart can detect queries with empty results during planning. As each worker makes its local parallel query plan, it detects the zero cardinality of the subquery in the replica index and terminates.

WatDiv dataset: The WatDiv benchmark defines 20 query templates[12] classified into four categories: linear (L), star (S), snowflake (F) and complex queries (C). Similar to TriAD, we generated 20 queries using the WatDiv query generator for each query category C, F, L and S. We deployed AdPart on five machines to match the setting of TriAD in [16]. Table 13 shows the performance of AdPart compared to other systems. For each query class, we show the geometric mean of each system. H2RDF+ performs worse than all other systems due to the overhead of MapReduce. SHAPE and H-RDF-3X, under 3-hop undirected guarantee, do not perform better than a single-machine RDF-3X. SHAPE placed all the data in one machine while H-RDF-3X replicated almost all the data everywhere. AdPart and TriAD, on the other hand, provide significantly better performance than MapReduce-based systems. TriAD performs better than AdPart-NA for L and F queries as these queries require multiple subject-object joins. TriAD can utilize the subject-object locality to answer these joins without communication whereas AdPart needs to communicate. Note that utilizing subject-object locality as in TriAD is orthogonal to our work. For complex queries with large diameters AdPart-NA performs better as a result of its locality awareness. When AdPart adapts, it performs better than all systems.

YAGO dataset YAGO2 does not provide benchmark queries, therefore we created a set of representative test queries (Y1-Y4). We show in Table 14 the performance of AdPart against all other systems. AdPart-NA solves most of the joins in Y1 and Y2 without communication; three out of four in Y1 and four out of six in Y2. This explains the comparable to superior performance of AdPart-NA compared to TriAD for Y1 and Y2, respectively. On the other hand, Y3 and Y4 require object-object joins on which AdPart-NA needs to broadcast the join column. As a results, TriAD performed better than AdPart-NA for these queries. Our adaptive version, AdPart, is up to several orders of magnitude faster than all other systems.

[12] `http://db.uwaterloo.ca/watdiv/basic-testing.shtml`

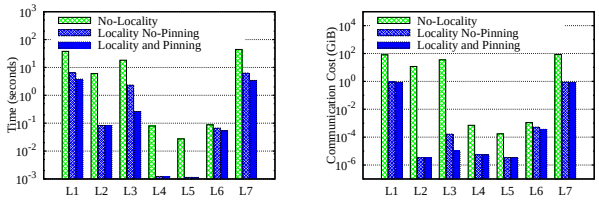
Table 15 Query runtimes for Bio2RDF (ms)

Bio2RDF	B1	B2	B3	B4	B5	Geo-Mean
AdPart	**3**	**2**	**2**	**3**	**1**	**2**
AdPart-NA	17	16	32	89	1	15
TriAD	4	4	5	N/A	2	4
H2RDF+	5,580	12,710	322,300	7,960	4,280	15,076
SHARD	239,350	309,440	512,850	787,100	112,280	320,027

Bio2RDF dataset: Similar to YAGO2, the Bio2RDF dataset does not have benchmark queries; therefore, we defined five queries (B1-B5) with different structures and complexities. B1 requires an object-object join which contradicts our initial partitioning. B2 and B3 are star queries with different number of triple patterns that require subject-object joins. Therefore, it is expected that TriAD would perform better than AdPart-NA (see Table 15). However, when AdPart adapts it, performs equally or better than TriAD. B4 is a complex query with 2-hop radius. AdPart-NA incur communication and utilize subject-based locality during sharding. TriAD, on the other hand, crashed during the query evaluation; hence marked as N/A. AdPart outperforms AdPart-NA by one order of magnitude. B5 is a simple star query with only one triple pattern in which all in-memory systems provide the same performance. H2RDF+ and SHARD perform worse than other systems due to the MapReduce overhead. Overall, AdPart outperforms all other systems by orders of magnitude.

6.3.1 Impact of Locality Awareness

In this experiment, we show the effect of locality aware planning on the distributed query evaluation of AdPart-NA (non-adaptive). We define three configurations of AdPart-NA: (*i*) We disable the *pinned_subject* optimization and hash locality awareness. (*ii*) We disable the *pinned_subject* optimization while maintaining the hash locality awareness; in other words, workers can still know the locality of subject vertices but joins on the pinned subjects are synchronized. Finally, (*iii*) we enable all optimizations. We run the LUBM (L1-L7) queries on the LUBM-10240 dataset on all configurations. The query response times and the communication costs are shown in Figures 12(a) and 12(b), respectively. Disabling hash locality resulted in excessive communication which drastically affected the query response times. Enabling the hash locality affected all queries except L6 because of its high selectivity. The performance gain for other queries ranges from 6X up to 2 orders of magnitude. In the third configuration, the pinned subject optimization does not affect the amount of communication because of the hash locality awareness. In other words, since the joining subject is local, AdPart does not communicate intermediate results. However, performance is affected by the synchronization overhead. The performance gain ranges from 26% in case of L6 to more than 90% for L3. Queries like L2, L4 and L5 are not affected by this optimization because they are star queries joining on the subject. The same behavior is also noticed in the WatDiv-1B dataset.



(a) Execution Time (b) Communication Cost

Fig. 12 Impact of locality awareness on LUBM-10240.

6.4 Workload Adaptivity by AdPart

In this section, we evaluate AdPart's adaptivity. For this purpose, we define different workloads on two billion-scale datasets that have different characteristics, namely, LUBM-10240 and WatDiv-1B.

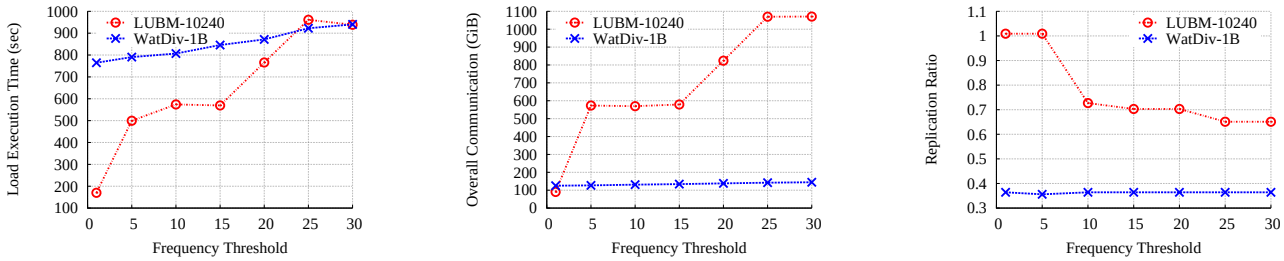
WatDiv-1B workload: We used the benchmark query generator to create a 5K-query workload from each query category (i.e., L, S, F and C), resulting in a total of 20K queries. Also, we generate a random workload by shuffling the 20K queries.

LUBM-10240 workload: As AdPart and the other systems do not support inferencing, we used all 14 queries in the LUBM benchmark without reasoning[13]. From these queries, we generated 10K unique queries that have different constants and structures. We shuffled the 10K queries to generate a random workload which we used throughout this section. This workload covers a wide spectrum of query complexities including simple selective queries, star queries as well as queries with complex structures and low selectivities.

6.4.1 Frequency Threshold Sensitivity Analysis

The frequency threshold controls the triggering of the IRD process. Consequently, it influences the execution time and the amount of communication and replication in the system. In this experiment, we conduct an empirical sensitivity analysis to select the frequency threshold value based on the two aforementioned query workloads. We execute each workload while varying the frequency threshold values from 1 to 30. Note that our frequency monitoring is not on a query-by-query basis

[13] Only query patterns are used. Classes and properties are fixed so queries return large intermediate results.



(a) Execution time (b) Communication cost (c) Replication cost

Fig. 13 Frequency threshold sensitivity analysis.

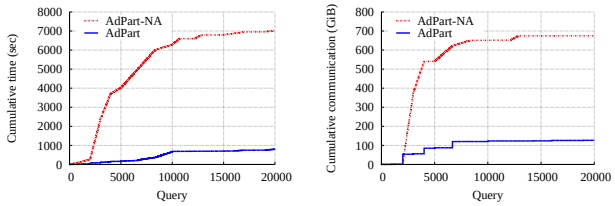
as our heat map monitors the frequency of the subquery pattern in a hierarchical manner (see Section 5.4). The workload execution times, the communication costs and the resulting replication ratios are shown in Figures 13(a), 13(b) and 13(c), respectively.

We observe that LUBM-10240 is very sensitive to slight changes in the frequency threshold because of the complexity of its queries. As the frequency threshold increases, the redistribution of hot patterns is delayed causing more queries to be executed with communication. Consequently, the amount of communication and synchronization overhead in the system increases, affecting the overall execution time, while the replication ratio is low because fewer patterns are redistributed.

On the other hand, WatDiv-1B is not as sensitive to this range of frequency thresholds because most of its queries are solved in subseconds using our locality-aware DSJ, without excessive communication. Nevertheless, as the frequency threshold increases, the synchronization overhead affects the overall execution time. Furthermore, due to our fine-grained query monitoring, AdPart captures the commonalities between the WatDiv-1B query templates for frequency thresholds 5 to 30. Hence, for all these thresholds the replication ratio remains almost the same. However, the system converges faster for lower threshold values, reducing the overall execution time. In all subsequent experiments, we use a frequency threshold of 10; this results in a good balance between time and replication. We plan to study the auto-tuning of this parameter in the future.

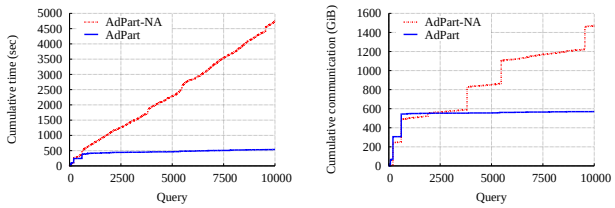
6.4.2 Workload Execution Cost

To simulate a change in the workload, queries of the same WatDiv-1B template are run consecutively while enforcing a replication threshold of 20%. Figure 14(a) shows the cumulative time as the execution progresses with and without the adaptivity feature. After every sequence of 5K query executions, the type of queries changes. Without adaptivity (i.e., AdPart-NA), the cumulative time increases sharply as long as complex queries



(a) Execution time (b) Communication cost

Fig. 14 AdPart adapting to workload (WatDiv-1B).

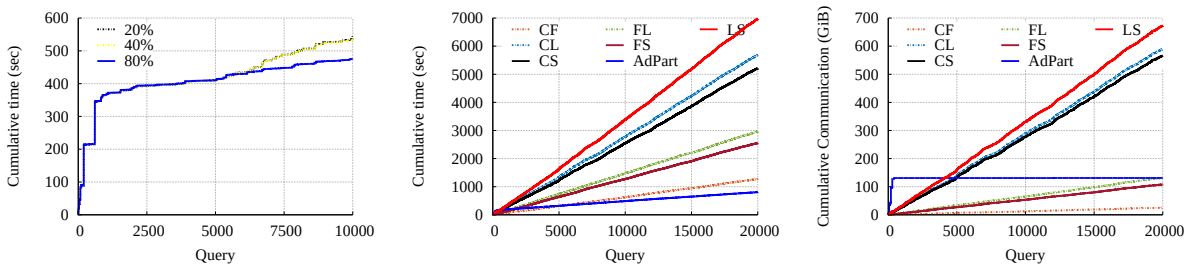


(a) Execution time (b) Communication cost

Fig. 15 AdPart adapting to workload (LUBM-10240).

are executed (e.g., from query 2K to query 10K). On the other hand, AdPart adapts to the workload change with little overhead causing the cumulative time to drop significantly by almost 6 times. Figure 14(b) shows the cumulative communication costs of both AdPart and AdPart-NA. As we can see, the communication cost exhibits the same pattern as that of the runtime cost (Figure 14(a)), which proves that communication and synchronization overheads are detrimental to the total query response time. The overall communication cost of AdPart is more than 7X lower compared to that of AdPart-NA. Once AdPart starts adapting, most of future queries are solved with minimum or no communication. The same behavior is observed for the LUBM-10240 workload (see Figures 15(a) and 15(b)).

Partitioning based on a representative workload: We tried to use Partout [13] to partition the LUBM-10240 and WatDiv-1B datasets based on a representative workload. However, it could not finish within reasonable time (<3 days) even for small workloads.



(a) Representative workload (b) Workload changes (time) (c) Workload changes (Communication)

Fig. 16 Comparison with workload-based partitioning.

Thus, in this experiment, we simulate two scenarios for workload-based data partitioning using AdPart. First, we assume the availability of a representative workload and measure how the training workload size affects performance. Second, we assume the data is partitioned using a workload that does not fully represent future queries. In both scenarios, there are two phases: training and testing. In the training phase, the adaptivity feature is enabled and the system can perform data redistribution for detected hot patterns. In the test phase, the adaptivity feature is disabled.

In the first scenario, we use a random workload of 10K LUBM queries where the first N% queries are used for training. The remaining queries are used for testing. Figure 16(a) shows how AdPart's performance changes as the size of the training window increases from 20% to 80%. With larger window sizes, more hot patterns are detected and redistributed in the training phase. Consequently, more queries in the test phase are solved without communication. Notice that, even with 20% queries, AdPart could detect most of the hot patterns in the workload and adapt accordingly. As a result, there is no significant difference between the total workload execution time when using 80% and only 20% training queries. This concludes that when a representative workload is available, systems that perform static workload-based partitioning like, Partout and WARP, can perform reasonably well for all workload queries.

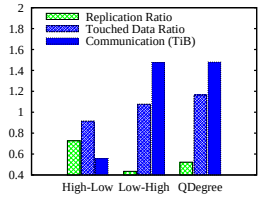
We further investigate another scenario where future queries are not well represented by the partitioning workload. The test set includes query patterns from the training query set as well as new queries that were not seen before. To do so, we train AdPart using different combinations of the workload categories defined by WatDiv-1B (C, F, S, and L). Each combination is made of two categories (10K queries); effectively producing six combinations, mainly CF, CL, CS, FL, FS, and LS. The test set includes 20K random queries made up from the four query categories. This way, some of the queries in the test workload would run in parallel while others (not in the representative workload) would require communication.

Figures 16(b) and 16(c) show the cumulative execution time and communication, respectively, for the test workloads (i.e., excluding the training time). For example, we train the system with the adaptivity feature enabled using 10K queries from two categories, like CF. Then, we test the system using 20K random queries while adaptivity is disabled. Obviously, the performance of the test workload highly depends on the complexity of the queries used in the training phase. For example, the complex (C) and snowflake (F) queries are the most expensive queries in the benchmark. Therefore, when the system is trained using the CF training workload, it performs much better than when trained using the LS workload. CF workload requires less communication because the L and S queries (not in the training workload) do not require excessive data exchange. Nonetheless, the CF execution time keeps increasing due to the existence of communication and synchronization overheads. In the same figures, we show the performance of AdPart without training, but the adaptivity is enabled all the time. Allowing the system to adapt incrementally and dynamically (without training) resulted in better performance when compared to all cases. AdPart incurs more communication at the beginning because of the IRD process; it then converges to almost constant communication.

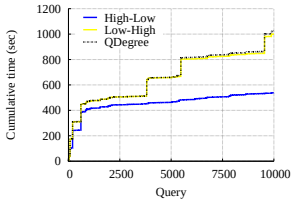
We also show another experiment with a more realistic workload where a given percentage of the workload queries are repeatedly executed while other new queries are constantly taken into account (see Appendix A).

6.4.3 Redistribution Tree Generation

In this experiment, we evaluate our query transformation heuristic (Section 5.1) against two alternative approaches. Recall that when transforming a hot query pattern into a redistribution tree, we select the vertex with the highest score to be the tree root. Then,

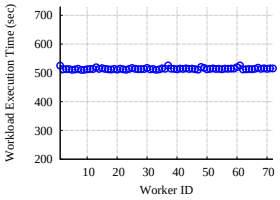


(a) Replication and Communication cost

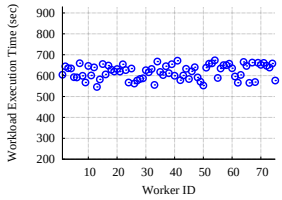


(b) Execution time

Fig. 17 Effect of hot pattern transformation.



(a) LUBM-10240



(b) WatDiv-1B

Fig. 18 Workload balance.

the query is traversed from high score vertices to lower score ones. We now compare our heuristic (referred to *High-Low* hereafter) to two different heuristics: (*i*) in *Low-High*, the vertex with the least vertex score is selected as core; then the query pattern is traversed by exploring vertices with lower scores first. The (*ii*) *QDegree* approach uses a different vertex scoring function where the score of a vertex in the hot query pattern is its out-degree. The pattern is then traversed from high score vertices to lower score ones. Note that the latter approach aims at minimizing the replication in a greedy manner by fully exploiting the initial hash partitioning. Recall that data that binds to triple patterns whose subject is a core are not replicated.

We evaluated all these heuristics by running the LUBM-10240 workload. In Figure 17(a), we show the resulting replication, the communication cost and the amount of data touched by the IRD process. Low-High and QDegree resulted in slightly less replication compared to High-Low. The reason is that both heuristics benefit from the initial hash partitioning by selecting cores with larger number of outgoing edges. However, the amount of data touched by IRD (i.e., data in the main and replica indices) in Low-High and QDegree is significantly higher. This affects adaptivity's performance because IRD is carried out using a series of DSJ iterations. Furthermore, as the data touched by the process is actually used for evaluating parallel queries, the performance of parallel queries is eventually affected.

Consequently, the cumulative workload execution time using High-Low is 1.9X faster than the other heuristics as shown in Figure 17(b). Since QDegree and Low-High touch and communicate almost the same amount of data, their cumulative execution times are also the same. Besides, note that QDegree does not use any statistical information from the data and only relies on the structure of the hot query pattern. Therefore, a redistributed pattern would not benefit other future queries with a slightly different structure. We repeated the experiment on WatDiv-1B and all heuristics resulted in almost the same communication cost, wall time, and touched data. This time, QDegree resulted in the least replication because its exploits best the initial subject-based hash partitioning.

Table 16 Load Balancing in AdPart

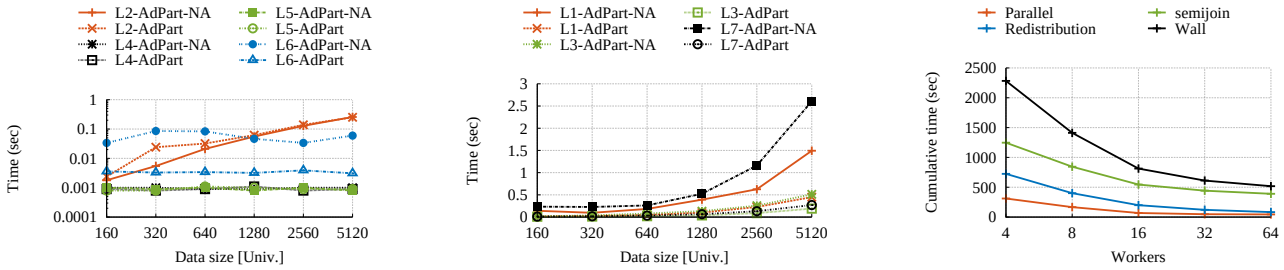
Dataset	Percentage of triples				Replication
	Max	Min	Average	StDev (σ)	Ratio
LUBM-10240	1.43%	1.35%	1.39%	0.02	0.73
WatDiv-1B	1.58%	1.20%	1.33%	0.07	0.36

6.4.4 Replication and Load Balance

In this experiment, we evaluate the load balance of AdPart from two different perspectives: (*i*) *data balance*, i.e., how balanced is the initial partitioning as well as the replication that results from the IRD process; (*ii*) *work balance*, i.e., how the evaluation cost is balanced among all workers in the system, during the execution of the workload. In Table 16, we report some statistics that characterize the data load balance in AdPart. Particularly, we report the average and standard deviation (σ) of the percentage of triples stored at each worker. As shown in the table, AdPart achieves very good data balance for both workloads because of the initial subject-based hash partitioning as well as the hashing used during the IRD process. Work is also well balanced among workers; i.e., the amount of work contributed by each worker is almost the same as shown in Figures 18(a) and 18(b) for the LUBM-10240 and WatDiv-1B, respectively.

6.5 Scalability

Data Scalability. We use the LUBM benchmark data generator to generate six datasets: LUBM-160, LUBM-320, LUBM-640, LUBM-1280, LUBM-2560 and LUBM-5120. We keep the number of workers fixed to 72 (6 workers per machine). Figures 19(a) and 19(b) show the data scalability of AdPart and AdPart-NA for simple and complex queries respectively. L4, L5, L6 are simple queries that are very selective and touch the same amount of data regardless of the data size. This describes the steady performance of both AdPart and



(a) Data scalability (simple) (b) Data scalability (complex) (c) Strong Scalability

Fig. 19 AdPart scalability using LUBM dataset.

AdPart-NA for these queries. Because L2 is not selective and returns massive final results, it is inevitable for its scalability to degrade as data size increases. Figure 19(b) shows the scalability of AdPart for complex queries. Queries L1 and L7 generate large number of intermediate results causing high communication cost, which explains their poor scalability of AdPart-NA. Nevertheless, as AdPart adapts to the workload, many queries are evaluated in parallel mode much faster.

Strong Scalability. In the last experiment, we use the 10K workload of LUBM-10240 to demonstrate the strong scalability of AdPart. We fix the workload while increasing the number of workers. Figure 19(c) shows the wall time for executing the workload. The time is split into the three constituents of AdPart execution, i.e, distributed execution (semijoin), redistribution (adaptivity) and parallel queries. All components of AdPart scale very well for up to 32 workers, afterwards the overhead of the semijoin communication starts dominating. Note that solving complex queries, like L1, L2, and L7 in parallel mode scale almost optimally. On the other hand, selective queries that touch very few data or are executed by a single worker do not scale. For future work, we will investigate the possibility of exploiting subjects and objects locality to further scale the distributed semijoin to more workers.

7 Conclusion

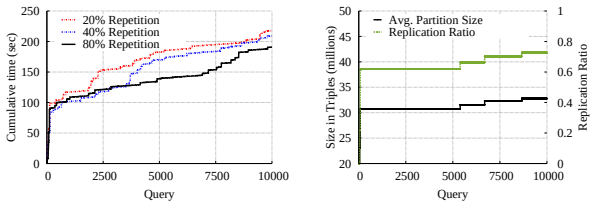
In this paper, we presented AdPart, an adaptive distributed RDF engine. Using lightweight partitioning that hashes triples on the subjects, AdPart exploits query structures and the hash-based data locality in order to minimize the communication cost during query evaluation. Furthermore, AdPart monitors the query workload and incrementally redistributes parts of the data that are frequently accessed by hot patterns. By maintaining and indexing these patterns, many future queries are evaluated without communication. The adaptivity feature of AdPart complements its excellent performance on queries that can benefit from its hash-based data locality. Frequent query patterns that are not favored by the initial partitioning (e.g., star joins on an object) can be processed in parallel due to adaptivity.

Our experimental results verify that AdPart achieves better partitioning and replicates less data than its competitors. More importantly, AdPart scales to very large RDF graphs and consistently provides superior performance by adapting to dynamically changing workloads. Currently, we are investigating the possibility of utilizing AdPart for general (i.e., non-RDF) graphs, and operators such as graph traversals, or reachability queries.

References

1. Aluç, G., Özsu, M.T., Daudjee, K.: Workload Matters: Why RDF Databases Need a New Design. PVLDB **7**(10) (2014)
2. Atre, M., Chaoji, V., Zaki, M.J., Hendler, J.A.: Matrix "Bit" loaded: a scalable lightweight join query processor for RDF data. In: WWW (2010)
3. Blanas, S., Li, Y., Patel, J.M.: Design and evaluation of main memory hash join algorithms for multi-core CPUs. In: SIGMOD (2011)
4. Bol'shev, L., Ubaidullaeva, M.: Chauvenet's Test in the Classical Theory of Errors. Theory of Probability & Its Applications **19**(4), 683–692 (1975)
5. Boyer, R.S., Strother Moore, J.: MJRTY: A Fast Majority Vote Algorithm. In: R.S. Boyer (ed.) Automated Reasoning: Essays in Honor of Woody Bledsoe, pp. 105–118. Kluwer, London (1991)
6. Chong, Z., Chen, H., Zhang, Z., Shu, H., Qi, G., Zhou, A.: RDF pattern matching using sortable views. In: CIKM (2012)
7. Curino, C., Jones, E., Zhang, Y., Madden, S.: Schism: a workload-driven approach to database replication and partitioning. PVLDB **3**(1-2) (2010)
8. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. In: OSDI (2004)
9. Dittrich, J., Quiané-Ruiz, J.A., Jindal, A., Kargin, Y., Setty, V., Schad, J.: Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). PVLDB **3**(1-2) (2010)
10. Dritsou, V., Constantopoulos, P., Deligiannakis, A., Kotidis, Y.: Optimizing query shortcuts in RDF databases. In: ESWC (2011)

11. Elnozahy, E.N.M., Alvisi, L., Wang, Y.M., Johnson, D.B.: A Survey of Rollback-recovery Protocols in Message-passing Systems. ACM Comput. Surv. **34**(3), 375–408 (2002)
12. Forum, M.P.: Mpi: A message-passing interface standard. Tech. rep., Knoxville, TN, USA (1994)
13. Galarraga, L., Hose, K., Schenkel, R.: Partout: A Distributed Engine for Efficient RDF Processing. CoRR **abs/1212.5636** (2012)
14. Gallego, M.A., Fernández, J.D., Martínez-Prieto, M.A., de la Fuente, P.: An empirical study of real-world SPARQL queries. In: USEWOD (2011)
15. Goasdoué, F., Karanasos, K., Leblay, J., Manolescu, I.: View selection in Semantic Web databases. PVLDB **5**(2) (2011)
16. Gurajada, S., Seufert, S., Miliaraki, I., Theobald, M.: TriAD: A Distributed Shared-nothing RDF Engine Based on Asynchronous Message Passing. In: SIGMOD (2014)
17. Harth, A., Umbrich, J., Hogan, A., Decker, S.: YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In: ISWC/ASWC, vol. 4825 (2007)
18. Hose, K., Schenkel, R.: WARP: Workload-aware replication and partitioning for RDF. In: ICDEW (2013)
19. Huang, J., Abadi, D., Ren, K.: Scalable SPARQL Querying of Large RDF Graphs. PVLDB **4**(11) (2011)
20. Husain, M., McGlothlin, J., Masud, M., Khan, L., Thuraisingham, B.: Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing. TKDE **23**(9) (2011)
21. Idreos, S., Kersten, M.L., Manegold, S.: Database Cracking. In: CIDR (2007)
22. Karypis, G., Kumar, V.: A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. SIAM J. Sci. Comput. **20**(1), 359–392 (1998)
23. Lee, K., Liu, L.: Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning. PVLDB **6**(14) (2013)
24. Malewicz, G., Austern, M., Bik, A., Dehnert, J., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a System for Large-scale Graph Processing. In: SIGMOD (2010)
25. Neumann, T., Weikum, G.: The rdf-3x engine for scalable management of rdf data. VLDB J. **19**(1), 91–113 (2010)
26. Papailiou, N., Konstantinou, I., Tsoumakos, D., Karras, P., Koziris, N.: H2rdf+: High-performance distributed joins over large-scale rdf graphs. In: IEEE Big Data (2013)
27. Punnoose, Roshan and Crainiceanu, Adina and Rapp, David: Rya: A Scalable RDF Triple Store for the Clouds. In: Cloud-I (2012)
28. Rietveld, L., Hoekstra, R., Schlobach, S., Guéret, C.: Structural Properties as Proxy for Semantic Relevance in RDF Graph Sampling. In: ISWC (2014)
29. Rohloff, K., Schantz, R.E.: High-performance, massively scalable distributed systems using the MapReduce software framework: the SHARD triple-store. In: PSI EtA (2010)
30. Shen, Y., Chen, G., Jagadish, H.V., Lu, W., Ooi, B.C., Tudor, B.M.: Fast Failure Recovery in Distributed Graph Processing Systems. PVLDB **8**(4) (2014)
31. Stonebraker, M., Madden, S., Abadi, D., Harizopoulos, S., Hachem, N., Helland, P.: The end of an Architectural Era: (It's Time for a Complete Rewrite). In: PVLDB (2007)
32. Wang, L., Xiao, Y., Shao, B., Wang, H.: How to partition a billion-node graph. In: ICDE (2014)
33. Weiss, C., Karras, P., Bernstein, A.: Hexastore: sextuple indexing for semantic web data management. PVLDB **1**(1) (2008)
34. Wu, Buwen and Zhou, Yongluan and Yuan, Pingpeng and Liu, Ling and Jin, Hai: Scalable SPARQL Querying using Path Partitioning. In: ICDE (2015)
35. Yang, S., Yan, X., Zong, B., Khan, A.: Towards effective partition management for large graphs. In: SIGMOD (2012)
36. Yuan, P., Liu, P., Wu, B., Jin, H., Zhang, W., Liu, L.: TripleBit: A Fast and Compact System for Large Scale RDF Data. PVLDB **6**(7), 517–528 (2013)
37. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: Cluster Computing with Working Sets. In: USENIX (2010)
38. Zeng, K., Yang, J., Wang, H., Shao, B., Wang, Z.: A distributed graph engine for web scale RDF data. PVLDB **6**(4) (2013)
39. Zhang, X., Chen, L., Tong, Y., Wang, M.: EAGRE: Towards scalable I/O efficient SPARQL query evaluation on the cloud. In: ICDE (2013)
40. Zou, L., Özsu, M.T., Chen, L., Shen, X., Huang, R., Zhao, D.: gStore: A Graph-based SPARQL Query Engine. VLDB J. **23**(4), 565–590 (2014)



(a) Effect of query repetition (b) Evolution of partition size

Fig. 20 AdPart adapting to workload (LUBM-10240).

A Workload Queries Repetition

In this experiment, we test AdPart's performance using a real scenario workload where a certain percentage of the queries is repeated while other new queries are taken into account. We use three workloads, each workload contains 10K LUBM random queries out of which a certain percentage is repeated. Figure 20(a) shows AdPart's performance while varying the amount of repeated queries between 20%, 40% and 80%. As the results suggest, the more the repeated queries, the less the workload execution time. Since AdPart monitors the query patterns and not the individual queries, it could capture most of the patterns in the workload even with only 20% of its queries repeated.

B Average Partition Size

In this experiment, we report how the average partition size changes during the workload execution. Using the 10K queries LUBM workload, Figure 20(b) shows how the partition size increases as more queries are executed. Initially, each partition contains around 19M triples. This corresponds to a 0% replication ratio as AdPart loads only the original dataset. As the system adapts, the size of each partition slightly increases till reaching an average size of around 33M triples; which counts for a 72% replication ratio after executing the whole 10K workload queries.

C LUBM Benchmark Queries

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/ zhp2/2004/0401/univ-bench.owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
Q1: SELECT ?X WHERE{
?X rdf:type ub:GraduateStudent .
?X ub:takesCourse <http://www.Department0.University0.edu/GraduateCourse0> .
}
Q2: SELECT ?X ?Y ?Z WHERE{
?X rdf:type ub:GraduateStudent .
?Y rdf:type ub:University .
?Z rdf:type ub:Department .
?X ub:memberOf ?Z .
?Z ub:subOrganizationOf ?Y .
?X ub:undergraduateDegreeFrom ?Y .
}
Q3: SELECT ?X WHERE{
?X rdf:type ub:Publication .
?X ub:publicationAuthor <http://www.Department0.University0.edu/AssistantProfessor0> .
}
Q4: SELECT ?X, ?Y1, ?Y2, ?Y3 WHERE
?X rdf:type ub:AssociateProfessor .
?X ub:worksFor <http://www.Department0.University0.edu> .
?X ub:name ?Y1 .
?X ub:emailAddress ?Y2 .
?X ub:telephone ?Y3 .
}
Q5: SELECT ?X WHERE{
?X rdf:type ub:UndergraduateStudent .
?X ub:memberOf <http://www.Department0.University0.edu> .
}
Q6: SELECT ?X WHERE{
?X rdf:type ub:UndergraduateStudent .
}
Q7: SELECT ?X, ?Y WHERE{
?X rdf:type ub:UndergraduateStudent .
?Y rdf:type ub:Course .
?X ub:takesCourse ?Y .
<http://www.Department0.University0.edu/AssociateProfessor0> ub:teacherOf ?Y .
}
Q8: SELECT ?X, ?Y, ?Z WHERE{
?X rdf:type ub:UndergraduateStudent .
?Y rdf:type ub:Department .
?X ub:memberOf ?Y .
?Y ub:subOrganizationOf <http://www.University0.edu> .
?X ub:emailAddress ?Z .
}
Q9: SELECT ?X, ?Y, ?Z WHERE{
?X rdf:type ub:GraduateStudent .
?Y rdf:type ub:AssociateProfessor .
?Z rdf:type ub:GraduateCourse .
?X ub:advisor ?Y .
?Y ub:teacherOf ?Z .
?X ub:takesCourse ?Z .
}
Q10: SELECT ?X WHERE{
?X rdf:type ub:TeachingAssistant .
?X ub:takesCourse <http://www.Department0.University0.edu/GraduateCourse0> .
}
Q11: SELECT ?X WHERE{
?X rdf:type ub:ResearchGroup .
?X ub:subOrganizationOf ?Z .
?Z ub:subOrganizationOf <http://www.University0.edu> .
}
Q12: SELECT ?X, ?Y WHERE{
?Y rdf:type ub:Department .
?X ub:headOf ?Y.
?Y ub:subOrganizationOf <http://www.University0.edu> .
}
Q13: SELECT ?X WHERE{
?X rdf:type ub:GraduateStudent .
?X ub:undergraduateDegreeFrom <http://www.University0.edu> .
}
Q14: SELECT ?X WHERE{
?X rdf:type ub:GraduateStudent .
}

D LUBM Workload

We generated a workload of 20,000 queries from LUBM benchmark queries shown in C. For queries that do not have constants (Q2 and Q9), we generate different query patterns by removing some triples and mutating the node types. For example, in Q2, we generated 18 different patterns by alternating student type between UndergraduateStudent and GraduateStudent (see Table 17). Similarly, other query patterns are generated by removing different combinations of the query triple patterns. We did not generate variations of Q6 and Q14 as they have only one triple pattern (*rdf:type*) with a single constant. For the rest of the queries, we generated 1000 different patterns from each query by varying the values of the query constants. For example, in Q1, we generate different query patterns by varying the values of both student type (UndergraduateStudent or GraduateStudent) and graduate courses.

Table 17 LUBM Workload

	Patterns	**Changes**
Q1	1000	Constants
Q2	18	Structure/Constants
Q3	1000	Constants
Q4	1000	Constants
Q5	1000	Constants
Q6	1	No Changes
Q7	1000	Constants
Q8	1000	Constants
Q9	30	Structure/Constants
Q10	1000	Constants
Q11	1000	Constants
Q12	1000	Constants
Q13	1000	Constants
Q14	1	No Changes

E YAGO2 Queries

PREFIX y: <http://yago-knowledge.org/resource/>
Y1: SELECT ?GivenName ?FamilyName WHERE{
?p y:hasGivenName ?GivenName .
?p y:hasFamilyName ?FamilyName .
?p y:wasBornIn ?city .
?p y:hasAcademicAdvisor ?a .
?a y:wasBornIn ?city .
}
Y2: SELECT ?GivenName ?FamilyName WHERE{
?p y:hasGivenName ?GivenName .
?p y:hasFamilyName ?FamilyName .
?p y:wasBornIn ?city .
?p y:hasAcademicAdvisor ?a .
?a y:wasBornIn ?city .
?p y:isMarriedTo ?p2 .
?p2 y:wasBornIn ?city .
}
Y3: SELECT ?name1 ?name2 WHERE{
?a1 y:hasPreferredName ?name1 .
?a2 y:hasPreferredName ?name2 .
?a1 y:actedIn ?movie .
?a2 y:actedIn ?movie .
}
Y4: SELECT ?name1 ?name2 WHERE{
?p1 y:hasPreferredName ?name1 .
?p2 y:hasPreferredName ?name2 .
?p1 y:isMarriedTo ?p2 .
?p1 y:wasBornIn ?city .
?p2 y:wasBornIn ?city .
}

F Bio2RDF

PREFIX pharmkb: <http://bio2rdf.org/pharmgkb_vocabulary>
PREFIX irefindex: <http://bio2rdf.org/irefindex_vocabulary>
PREFIX pubmd: <http://bio2rdf.org/pubmed_vocabulary>
PREFIX pubmdrc: <http://bio2rdf.org/pubmed_resource>
PREFIX omim:<http://bio2rdf.org/omim_vocabulary>
PREFIX drug:<http://bio2rdf.org/drugbank>
PREFIX uniprot:<http://bio2rdf.org/uniprot>
B1: SELECT ?o WHERE{
pubmdrc:1374967_INVESTIGATOR_1 pubmd:last_name ?o .
pubmdrc:1374967_AUTHOR_1 pubmd:last_name ?o .
}
B2: SELECT ?articleToMesh WHERE{
<http://bio2rdf.org/pubmed:126183> pubmd:mesh_heading ?articleToMesh .
?articleToMesh pubmd:mesh_descriptor_name ?mesh .
}
B3: SELECT ?phenotype WHERE{
?phenotype rdf:type omim:Phenotype .
?phenotype rdfs:label ?label .
?gene omim:phenotype ?phenotype .
}
B4: SELECT ?pharmgkbid WHERE{
?pharmgkbid pharmkb:xref drug:DB00126 .
?pharmgkbid pharmkb:xref ?pccid .
?DDIassociation pharmkb:chemical ?pccid .
?DDIassociation pharmkb:event ?DDIevent .
?DDIassociation pharmkb:chemical ?drug2 .
?DDIassociation pharmkb:p-value ?pvalue .
}
B5: SELECT ?interaction WHERE{
?interaction irefindex:interactor_a uniprot:O17680 .
}