TriAD: A Distributed Shared-Nothing RDF Engine based on Asynchronous Message Passing

Sairam Gurajada[†] Stephan Seufert[†] Iris Miliaraki[†] Martin Theobald[‡]

[†]{gurajada, sseufert, miliaraki}@mpi-inf.mpg.de
Max-Planck Institute for Informatics
Saarbrücken, Germany

[‡]martin.theobald@ua.ac.be
University of Antwerp
Antwerp, Belgium

ABSTRACT

We investigate a new approach to the design of distributed, shared-nothing RDF engines. Our engine, coined "TriAD", combines *join-ahead pruning* via a novel form of RDF graph summarization with a *locality-based, horizontal partitioning* of RDF triples into a grid-like, distributed index structure. The multi-threaded and distributed execution of joins in TriAD is facilitated by an *asynchronous Message Passing protocol* which allows us to run multiple join operators along a query plan in a fully parallel, asynchronous fashion. We believe that our architecture provides a so far unique approach to join-ahead pruning in a distributed environment, as the more classical form of sideways information passing would not permit for executing distributed joins in an asynchronous way. Our experiments over the LUBM, BTC and WSDTS benchmarks demonstrate that TriAD consistently outperforms centralized RDF engines by up to two orders of magnitude, while gaining a factor of more than three compared to the currently fastest, distributed engines. To our knowledge, we are thus able to report the so far fastest query response times for the above benchmarks using a mid-range server and regular Ethernet setup.

Categories and Subject Descriptors

H.2.4 [**Systems**]: Distributed Databases, Query Processing

Keywords

Distributed RDF Indexing & SPARQL Processing, Asynchronous Message Passing, Parallel Join Evaluation, Join-Ahead Pruning

1. INTRODUCTION

The Resource Description Framework (RDF) and the SPARQL query language[1] are two recent standards recommended by the W3C for representing and querying linked data on the Web. RDF has become the main standard for semantic data and meanwhile found a wide adoption in the Database as well as the Semantic Web communities. With the increasing number of both commercial and non-commercial organizations, which actively publish RDF data, the amount and diversity of openly available RDF repositories is growing at an unprecedented pace. DBpedia[2], for example, which serves as the main hub for the Linked Open Data[3] (LOD) initiative, currently consists of more than 1 billion RDF triples. As of 2011, the entire LOD cloud already consisted of more than 31 billion RDF triples which are distributed across more than 300 LOD sources.

Consequently, and in response to this explosion of RDF data that is available on both the surface and the deep Web, much research effort has been invested recently in the development of scalable, both centralized and distributed, techniques for indexing RDF data and for processing SPARQL queries. Among the centralized approaches, native RDF stores like Jena, Sesame, HexaStore [26], SW-Store [1], MonetDB-RDF [22], RDF-3X [14, 15], BitMat [2] and TripleBit [28] have been carefully designed to keep up pace with the growing scale of RDF collections. Efficient centralized architectures either employ various forms of horizontal [14, 26] and vertical [1, 22] partitioning schemes, or apply sophisticated encoding schemes over the subjects' properties [2, 28].

With the increasing popularity of shared-nothing architectures based on the MapReduce paradigm [3], systems like SHARD [17], [8] (an offspring of SW-Store, in the following referred to as "H-RDF-3X"), and EAGRE [31] have been proposed for the scalable, distributed evaluation of SPARQL queries. While MapReduce allows for an easy adaptation of parallel (both Map- and Reduce-side [10]) join algorithms on top of RDF-specific index structures, MapReduce frameworks are known to incur a non-negligible overhead due to their iterative, synchronous communication protocols and fault-tolerant job scheduling strategies. Even with the currently fastest, openly available MapReduce implementations, such as Hadoop++ [4] and Spark [29], this typically renders sub-second query response times for distributed joins infeasible. Systems like H-RDF-3X [8] and EAGRE [31] thus make use of aggressive data replication to avoid iterative joins in Hadoop and to restrict query executions to the local RDF stores as much as possible. However, with longer-diameter queries or unexpected workloads, there is no alternative to running joins via Hadoop, which often slows down query response times by two or more orders of magnitude.

Trinity.RDF [30] is the first distributed RDF engine that employs a custom communication protocol based on the Message Passing Interface (MPI) standard [6]. Instead of joining index lists, Trinity.RDF follows a graph-exploration strategy on top of a distributed, in-memory key-value store. Although Trinity.RDF is only single-threaded in its final join phase, it often allows for faster response times compared to Hadoop-based RDF engines, especially when queries are selective and the graph exploration starts from just a few initial nodes. For non-selective queries, however, the generic architecture of Trinity.RDF, which is based on the Trinity graph engine [21], does not allow for the integration of parallel join techniques, as they are common, on the other hand, in Hadoop [8, 31].

[1] http://www.w3.org/TR/rdf-sparql-query/

[2] http://dbpedia.org

[3] http://linkeddata.org

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.
Copyright 2014 ACM 978-1-4503-2376-5/14/06 ...$15.00
http://dx.doi.org/10.1145/2588555.2610511.

We summarize this analysis by highlighting the following two limitations that all existing, distributed RDF engines currently face.

Problem 1: Synchronous vs. Asynchronous Join Executions. Although Hadoop-based joins allow for the execution of multiple join operators in parallel, they need to *synchronize at each level of the query plan* before they can continue to process the plan with the next iteration of joins. These synchronization steps are heavily dominated by a few stragglers or imbalanced query plans.

Problem 2: Graph Exploration vs. Relational Joins. Parallel graph exploration is very efficient for queries that aim to select just a few subgraphs out of the RDF data graph. For a row-oriented output format, as it is required by the SPARQL 1.0 and 1.1 standards, *graph exploration is not sufficient to generate the final join results*. Thus, the parallel execution of joins remains a crucial factor for the efficiency and scalability of a SPARQL engine.

1.1 TriAD Overview

To address the above problems, we propose a novel, shared-nothing, main-memory architecture in combination with an asynchronous Message Passing [6] protocol. Our engine, coined TriAD (for "Triple-Asynchronous-Distributed"), aims at closing the gap between current shared-nothing Hadoop engines [8, 17, 31], on the one hand, and pure graph-exploration strategies based on Message Passing [21, 30], on the other hand. TriAD is designed to achieve higher parallelism and less synchronization overhead during query executions than Hadoop engines by adding an additional layer of multi-threading for entire paths of a query plan that can be executed in parallel. TriAD is the first RDF engine that employs asynchronous join executions (using a custom MPI protocol) which are coupled with a lightweight join-ahead pruning technique for the distributed processing of SPARQL queries. Specifically, TriAD builds on the following principles.

Parallel and Asynchronous Join Executions. TriAD in principle follows a classical master-slave architecture. During query execution, however, the slave nodes operate largely autonomously and communicate directly via *asynchronously exchanged messages* to run multiple join operators along the query plan in parallel. Our form of communication is asynchronous because sibling execution paths of a query plan can be processed in a freely multi-threaded fashion and only need to be merged (i.e., be synchronized) once the intermediate results of entire such execution paths are joined.

Distributed RDF Indexes with Join-Ahead Pruning. We employ six primary SPO *permutation indexes* which are encoded into a distributed main-memory data structure that consists only of simple integer structs and vectors. Each SPO permutation list is first hash-partitioned ("sharded") according to its join key and then locally sorted in lexicographic order. Thus, even in its basic configuration without any multi-threaded execution of the query plan, TriAD can perform efficient, distributed merge-joins over the hash-partitioned permutation lists. In addition to the primary SPO indexes, we employ a form of join-ahead pruning via an additional RDF *summary graph* at the master node, in order to prune entire partitions of triples from the SPO lists that cannot contribute to the results of a given SPARQL query.

Distribution-Aware Query Optimizer. Similar to [14], TriAD employs a bottom-up dynamic programming (DP) algorithm for *join-order enumeration*. In addition to [14], we also consider the locality of the index structures at the slave nodes, the shipping cost of intermediate join results, and the option to execute sibling paths of the query plan in a multi-threaded fashion, in order to determine the plan with the overall least cost estimate. This enables the optimizer to take much better advantage of the actual hardware capabilities, by taking the network latency and bandwidth, the CPU capacity for merging and hashing, and parallel query executions via multi-threading and distribution into account.

1.2 Contributions

We summarize the novel aspects of our work as follows.

- We investigate a new approach to the design of distributed RDF engines. TriAD exploits both *intra-node multi-threading* and *asynchronous inter-node communication* to run multiple join operators of a query plan in a distributed and parallel way.
- We propose a novel form of RDF graph summarization to facilitate *join-ahead pruning* in a distributed environment. In contrast to sideways information passing, the graph summary is directly merged into the RDF-specific SPO indexes and thus allows us to perform this kind of join-ahead pruning in combination *with an asynchronous execution* of the join operators.
- TriAD employs two stages of query optimization (and execution) over both the RDF summary graph and the RDF data graph. Our distribution-aware query optimizer employs detailed summary- and data-graph statistics to determine the *best exploration-order* for the summary graph and the *best join-order* for the data graph, respectively. Both optimization steps are implemented via an efficient DP algorithm.
- Each individual join operator runs against a *distributed, horizontally partitioned RDF index*, such that even for a single join or path-like queries TriAD benefits from the distributed evaluation of these joins. In addition, for a more "bushy" query plan, consisting of multiple root-to-leaf paths (called "execution paths"), the execution of the joins runs in multiple threads at each compute node, which allows us to evaluate multiple operators in the query plan in parallel and asynchronously along these execution paths.
- We provide an *extensive experimental comparison* of TriAD to no less than nine state-of-the-art RDF, DBMS and Hadoop engines. We achieve the—to our knowledge—so far fastest query response times for the LUBM, BTC and WSDTS benchmarks reported for a mid-range server and regular Ethernet setup.

2. RELATED WORK

We next discuss a selection of RDF engines, which we believe are most related to our approach, and briefly discuss their differences to our architecture. We refer the reader to [5, 18, 22] for a comprehensive overview of recent approaches.

Relational Approaches. The majority of the existing RDF stores, both centralized and distributed, follow a relational approach towards storing and indexing RDF data. Recent approaches, such as SW-store by Abadi et al. [1], vertically partition RDF triples into multiple property tables. Hexastore [26] and RDF-3X [14, 15] employ index-based solutions by storing triples directly in B^+-trees over multiple, redundant SPO permutations. Including all permutations and projections of the SPO attributes, this may result in up to 15 such B^+-trees [14]. Coupled with sophisticated statistics and query-optimization techniques, these centralized, index-based approaches still are very competitive as recently shown in [24].

Join-Order Optimization. Determining the optimal join-order for a query plan is arguably the main factor that impacts query processing performance. RDF-3X [14] thus performs an exhaustive plan enumeration in combination with a bottom-up DP algorithm and aggressive pruning in order to identify the best join-order. In TriAD, we adopt the DP algorithm as it is described in [14], and we adapt it to finding both the best exploration-order for the summary graph and the best join-order for the subsequent processing against the SPO indexes. Moreover, by including detailed distribution information and the ability to run multiple joins in parallel into the underlying cost model of the optimizer, we obtain query plans that are more specifically tuned towards parallel execution than with a pure selectivity-based cost model.

Join-Ahead Pruning. Join-ahead pruning is a second main factor that influences the performance of a relational query processor. In join-ahead pruning, triples that might not qualify for a join are pruned even before the actual join operator is invoked. This pruning of triples ahead of the join operators may thus save a substantial amount of computation time for the actual joins. Instead of the sideways information passing (SIP) strategy used in RDF-3X [14, 15], which is a runtime form of join-ahead pruning, TriAD employs a similar kind of pruning via graph summarization [12, 16, 32]. Graph summarization serves as a preprocessing step to the actual query executions and thus has the crucial advantage that it can be adapted to an asynchronous execution of the join operators.

MapReduce. Based on the MapReduce paradigm, distributed engines like H-RDF-3X [8] and SHARD [17] horizontally partition an RDF collection over a number of compute nodes and employ Hadoop as a communication layer for queries that span multiple nodes. H-RDF-3X [8] partitions an RDF graph into as many partitions as there are compute nodes via METIS [9]. Then, a one- or two-hop replication is applied to index each of the local graphs via RDF-3X [15]. Query processing in both systems is performed using iterative Reduce-side joins, where the Map phase performs selections and the Reduce phase performs the actual joins [10]. Although such a setting works well for queries that scan large portions of the RDF data graph, for less data-intensive queries the overhead of iteratively running MapReduce jobs and scanning all—or large amounts—of the RDF tuples during the Map phase is significant. Even recent approaches like EAGRE [31] that focus on minimizing I/O costs by carefully scheduling Map tasks and utilizing extensive data replication cannot completely avoid Hadoop-based joins in the case of longer-diameter queries or unexpected workloads. Our experimental evaluation clearly shows that running joins via Hadoop should be avoided if interactive query response are desired.

Native Graph Approaches. Recently, a number of approaches were proposed to store RDF triples in native graph format. These approaches typically employ adjacency lists as a basic building block for storing and processing RDF data. Moreover, by using sophisticated indexes, like gStore [32], BitMat [2] and TripleBit [28], or by using graph exploration, like in Trinity.RDF [30], these approaches prune many triples before invoking relational joins to finally generate the row-oriented results of a SPARQL query. We believe that with Trinity.RDF [30], we provide a detailed experimental comparison to such graph approaches for RDF, which thus also represents a wider family of more generic graph engines such as Pregel [11] or Neo4j [25]. Other kinds of graph queries, such as reachability, shortest-paths or random walks, are partly already included in the SPARQL 1.1 standard and required for RDF/S-style inferences. Such queries are targeted by various graph engines, such as FERRARI [19] or GraphX [27], but we consider these to be beyond the scope of this work. Also beyond our current scope are workload awareness [20] and incremental updates [15].

Graph Partitioning. Apart from centralized engines like gStore [32], BitMat [2] and TripleBit [28], Huang et al. [8] also follow a graph partitioning approach over a distributed setting, where triples are assigned to different machines using the METIS graph partitioner. Graph partitioning allows triples that are close to each other in the RDF data graph to be stored at the same machine, thus overcoming the randomness issued by the purely hashing-based partitioning schemes used in systems like SHARD [17].

Graph Exploration vs. Joins. To avoid the overhead of Hadoop-based joins, Trinity.RDF [30] is based on a custom protocol based on the Message Passing Interface (MPI) [6]. In Trinity.RDF, however, intermediate variable bindings are computed among all slave nodes via graph exploration, while the final results need to be enumerated at the single master node using a single-threaded, left-deep join over the intermediate bindings. As an example, consider a SPARQL query with 3 variables `?x`, `?y`, `?z`, which each become bound to 10 distinct constants during graph exploration. Assuming that each combination of the bindings generates a valid SPARQL result, the 30 bindings lead to 1,000 rows that need to be generated for the join. Thus, the ability to evaluate joins in parallel remains a crucial factor for scaling-out an RDF engine.

3. BACKGROUND & PRELIMINARIES

In this section, we briefly review the key concepts that form the basis for the design of TriAD. We also establish the notation used throughout the rest of the paper.

3.1 RDF & SPARQL: Data & Query Model

An RDF collection consists of a set of triples of the form ⟨*subject*, *predicate*, *object*⟩ (or $\langle s, p, o\rangle$, for short), where *subject* denotes a globally unique resource, *object* may denote either a unique resource or a literal (i.e., a string or a number), and *predicate* denotes a relationship between the subject and object. RDF data can be represented as a directed, labeled multi-graph as defined next.

DEFINITION 1. *An* ***RDF data graph*** $G_D(V_D, E_D, L, \phi_D)$ *is a directed, labeled multi-graph where* V_D *is the set of data nodes,* E_D *is the set of directed edges connecting the nodes in* V_D*,* L *is the set of edge and node labels, and* ϕ_D *is a labeling function with* $\phi_D : V_D \cup E_D \rightarrow L$ *s.t.* $\forall v_i, v_j \in V_D$*,* $v_i \neq v_j$*, it holds that* $\phi_D(v_i) \neq \phi_D(v_j)$*.*

An RDF example in triplet form (TTL/N3 format) is given below.

```
Barack_Obama <bornIn> Honolulu .
Barack_Obama <won> Peace_Nobel_Prize .
Barack_Obama <won> Grammy_Award .
Honolulu <locatedIn> USA .
```

A conjunctive SPARQL query can again be represented by a set of triple patterns that together form the query graph. Each query triple is of the form $\langle s, p, o\rangle$, where each of the s, p, o components may denote either a query variable in *Vars* or a constant in L.

DEFINITION 2. *A* ***SPARQL query graph*** $G_Q(V_Q, E_Q, L, \mathit{Vars}, \phi_Q)$ *is a labeled, directed multi-graph where* V_Q *is the set of query nodes,* E_Q *is the set of edges connecting nodes in* V_Q*,* L *is the set of edge and node labels, and* ϕ_Q *is a labeling function with* $\phi_Q : V_Q \cup E_Q \rightarrow \mathit{Vars} \cup L$*.*

An example query (which retrieves all the people who are born in a city that is located in "`USA`" and who won some prize) is expressed in SPARQL as follows.

```
SELECT ?person, ?city, ?prize WHERE {
  ?person <bornIn> ?city .
  ?city <locatedIn> USA .
  ?person <won> ?prize . }
```

Processing a SPARQL query graph G_Q against an RDF data graph G_D thus resolves to finding all subgraph isomorphisms between G_Q and G_D. The result of a SPARQL query, however, is not itself a graph but—in analogy to SQL—a set of rows, each containing a distinct set of bindings of query variables in *Vars* to constants in L. For example, the result of the above SPARQL query over our RDF data snippet is the following.

```
Barack_Obama, Honolulu, Peace_Nobel_Prize .
Barack_Obama, Honolulu, Grammy_Award .
```

3.2 Graph Summarization

Graph summarization is an effective approach to prune dangling triples prior to the actual query processing. In graph summarization, a large data graph is first summarized into a smaller graph that

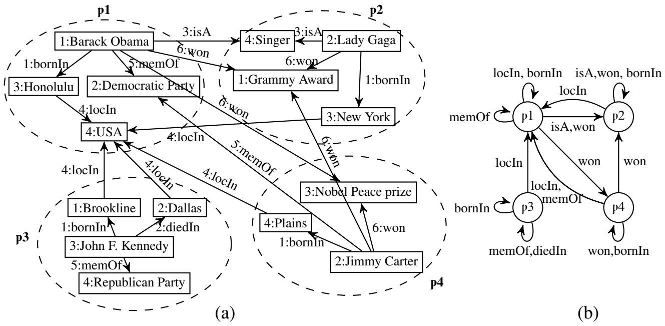


Figure 1: (a) RDF data graph G_D with locality-based partitioning; and (b) summary graph G_S for G_D

retains the principal characteristics of the original RDF data graph in a compact way. The main intuition behind graph summarization is that processing a query over the summary graph allows us to remove large parts of the data graph that contain no relevant triples with respect to the query. Running a complex query against both the summary graph and subsequently against the pruned data graph may thus be faster than running the query against the unpruned data graph. We formally define an RDF summary graph as follows.

DEFINITION 3. *An **RDF summary graph** $G_S(V_S, E_S, L, \phi_S)$ for a given RDF data graph $G_D(V_D, E_D, L, \phi_D)$ again is a labeled, directed multi-graph where each node $v \in V_S$, with $v \subseteq V_D$, called **supernode**, is a subset of nodes in V_D, and each edge $e \in E_S$, called **superedge**, connects two supernodes in V_S, and $\phi_S : E_S \rightarrow L$ maps each superedge $e \in E_S$ to a label in L.*

Generating Graph Summaries. A few centralized RDF stores have so far been proposed to perform join-ahead pruning via graph summarization [16, 32]. These extend the idea of bisimulation [12], which was originally employed for XML tree summarization, to RDF graphs. *Bisimulation-based summaries* [16] are particularly effective for join-ahead pruning if only the predicates of the query triple patterns are labeled with constants, such that multiple, possibly disconnected components of the data graph are merged into compact synopses for indexing. *Locality-based summaries* [32], on the other hand, are similar to graph clustering, in which nodes of the data graph are partitioned such that the nodes within each partition share more neighbors than the nodes that are spread across the partitions. Since SPARQL typically involves finding connected components of the data graph, locality-based approaches are particularly effective in pruning if one or more of the subjects or objects in the query graph are labeled with constants. Such queries are very common in SPARQL. An example of an RDF data graph and a corresponding locality-based summary graph is shown in Figure 1.

EXAMPLE 1. *Running our SPARQL query against the summary graph G_S of Figure 1 binds partitions p_1, p_2, p_4 to* `?person`*, p_1, p_2, p_4 to* `?city` *and p_2, p_4 to* `?prize`*. Thus, all RDF triples in G_D, which are associated with p_3, can safely be pruned when processing the query against the data graph without introducing false negatives to the result. By processing the query against G_D, we replace these supernode bindings of the query variables with their actual RDF constants and thus remove also false positives from the results. Often, this form of join-ahead pruning allows us to detect empty join results without even touching the data graph at all.*

4. SYSTEM ARCHITECTURE

An overview of the TriAD system architecture is depicted in Figure 2. TriAD resembles a typical master-slave, shared-nothing model, in which each compute node manages its own main memory area and stores disjoint partitions of the RDF index structures. One designated compute node, the *master node*, stores all metadata about the indexed RDF facts and serves as the initial point of contact for all indexing and query processing tasks. The remaining *slave nodes* hold the local index structures and exchange intermediate query results via a direct, asynchronous communication protocol among each other. All communication is based on the Message Passing Interface (MPI) using the MPICH2[4] API.

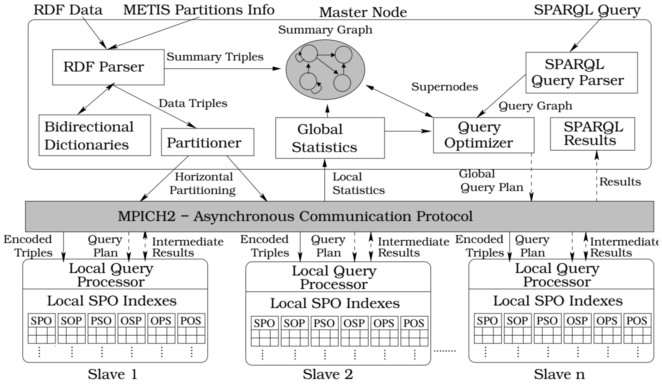


Figure 2: TriAD system architecture

Master Node

RDF Parser & Partitioner. This component takes care of parsing RDF files (provided in TTL/N3 format) and partitioning the complete set of incoming RDF triples into the summary graph and the local SPO index structures (Section 5).

SPARQL Parser. The SPARQL parser is responsible for preprocessing incoming queries. Queries are turned into a graph representation, before the query optimizer compiles the query into a global join plan which is then sent to all slaves (Section 6).

Summary Graph. The initial processing of a SPARQL query pattern against the summary graph facilitates join-ahead pruning (using a locality-based summarization strategy) at the slaves by removing graph partitions that contain no matching triples for the graph pattern denoted by the query (Section 5.1).

Bidirectional Dictionaries. The RDF parsing step involves building bidirectional mappings for the incoming RDF triples in order to quickly convert strings to integer ids and vice versa. To accommodate our graph partitioning scheme for the summary graph, the forward dictionary maintains the combination of partition identifier (a node in the summary graph) and component id (Section 5.2).

Global Statistics. When indexing finishes, the master receives the local index statistics from the slaves and merges these into its own global statistics to be used for query optimization (Section 5.5).

Query Optimizer. In a second processing step, the query optimizer (Section 6.3) builds the global query plan based on the global statistics, the locality of the SPO indexes, and cardinality re-estimations after processing the query against the summary graph.

Slave Nodes

Local SPO Indexes. At each slave, a local indexer receives the id-formatted triples and builds its local index structures for each of the six primary SPO permutations (Sections 5.3 & 5.4).

Local Query Processors. Each slave receives a copy of the global query plan from the master, whereupon the local query processors initialize their own instances of the physical query operators in the plan. The slaves concurrently start executing the same plan but scan different partitions of their local SPO indexes. Along with the global plan, the master also communicates the join-head pruning information from the summary graph to the slaves (Section 6.4).

[4] `http://www.mpich.org/`

5. INDEX ORGANIZATION

In this section, we provide a detailed description of the data partitioning and indexing strategies employed by TriAD.

5.1 Global Summary Graph

In order to avoid processing unnecessarily large SPO permutation lists at query time, we pursue a join-ahead pruning technique at the master node. Specifically, we employ a summary graph, denoted as $G_S(V_S, E_S, L, \phi_S)$, for this purpose, which is stored at the master and serves as a concise summary of the actual RDF data graph $G_D(V_D, E_D, L, \phi_D)$ (see Definitions 1 & 3).

RDF Graph Partitioning. Incoming triples, as they are produced by the RDF parser, are of the form $\langle s, p, o\rangle$, where $s, o \in V_D$ and p is a label in L. Each distinct $\langle s, p, o\rangle$ triple is mapped to an edge $v \in E_D$ with $\phi_D(v) = p$. In order to create the summary graph, we first consider this set of RDF facts as one large graph G_D (using an intermediate dictionary for mapping node and edge labels to integer ids) and apply a non-overlapping graph-partitioning algorithm like METIS [9] to it. In the resulting partitioning scheme, each distinct subject s or object o that occurs in an RDF triple is assigned to exactly one graph partition (i.e., supernode) $\mathrm{p} \in V_S$.

The resulting summary graph is treated as a new set of triples of the form $\langle \mathrm{p}_1, p, \mathrm{p}_2\rangle$, where $\mathrm{p}_1, \mathrm{p}_2 \in V_S$ are supernodes. For each original $\langle s, p, o\rangle$ triple that lies in the cut between two supernodes p_1,p_2, a new superedge $v \in E_S$ with the same label $\phi_S(v) = p$ as the original edge in V_D is introduced. Within each $\mathrm{p}_i \in V_S$, the original edges of the RDF data graph form self-loop edges of p_i. Moreover, among each such pair of supernodes p_i, $\mathrm{p}_j \in V_S$, the summary graph only stores edges with distinct labels p. Altogether, this reduces the size of the summary graph in comparison to the data graph drastically (see Figure 1).

Indexing the Summary Graph. After partitioning the data graph, summary triples of the form $\langle \mathrm{p}_1,\ p,\ \mathrm{p}_2\rangle$ are indexed at the master node. To support an efficient exploratory search over the summary graph, we index edges in G_S in an adjacency-list-like format. These are stored as two large in-memory vectors holding the PSO and POS permutations of the summary triples for both forward (outgoing links) and backward (incoming links) lookups. Each of the two vectors is sorted in lexicographical order and processed via a combination of binary search and direct pointer accesses.

Optimal Number of Partitions. Determining the number of partitions that minimizes the combined query cost over both the summary and the (pruned) data graph purely empirically may be a very tricky and costly procedure by itself. In order to obtain an estimation of the best summary graph size, we formulate the following cost model as an optimization problem that takes both the centralized query execution at the summary graph and the subsequent distributed execution at the pruned data graph into account.

Let $|V_D|$ and $|E_D|$ be the number of nodes and edges in the data graph, respectively, and let $d := \frac{|E_D|}{|V_D|}$ be the average degree of a node in the data graph. Further, let c_D denote the cost of executing a query against the data graph in a centralized setting. Ideally, the cost $c_{D,n}$ for processing a query in a distributed setting linearly scales with the number of slaves n, i.e., $c_{D,n} = \frac{c_D}{n}$. Similarly, let $|V_S|$ be the targeted number of nodes in the summary graph. Then it is reasonable to assume that the cost c_S of processing the query against the summary graph is proportional to the summary graph size, i.e., $c_S := \frac{|E_S|}{|E_D|} \cdot c_D = \frac{d\,|V_S|}{|E_D|} \cdot c_D$. Finally, let $|V_P|$ and $|E_P|$ be the number of nodes and edges in the data graph pruned by preprocessing the query against the summary graph. Then the cost $c_{P,n}$ of processing the query against the pruned graph in a distributed setting is $c_{P,n} := \frac{|E_P|}{|E_D|} \cdot c_{D,n}$. Assuming further that the size of the pruned data graph—at least for selective queries—is inversely proportional to the size of the summary graph, we can rewrite the latter cost as $c_{P,n} = \frac{\lambda}{|V_S|} \cdot c_{D,n}$. Putting all these costs together, we obtain the total cost $c_{Q,n}$ of processing a query against the summary and subsequently against the data graph as follows.

$$\begin{aligned} c_{Q,n} &:= c_S + c_{P,n} \\ &= \frac{d\,|V_S|}{|E_D|} \cdot c_D + \frac{\lambda}{|V_S|} \cdot \frac{c_D}{n} \end{aligned} \tag{1}$$

This yields a cost function that is convex in $|V_S|$. Minimizing $c_{Q,n}$ thus gives an optimal number of nodes when $|V_S| := \sqrt{\frac{\lambda |E_D|}{d\,n}}$. We remark that this result coincides with information-theoretic results for determining the optimal number of clusters in a data set [23].

Although this makes the number of summary graph partitions (e.g., for METIS) easy to compute, in practice, the best choice of partitions certainly depends on a multitude of parameters, including the particular characteristics of the given data set, the query workload, the hardware configuration, as well as the network bandwidth and latency. We project all these latent parameters into a single parameter λ in our cost model, which we need to measure (only once) empirically for a given hardware and benchmark setting.

EXAMPLE 2. *We empirically verified how well a measured value of λ generalizes to different scales of a given data set and query workload as follows. Based on the LUBM-160 benchmark with queries Q1–Q7 (see Section 7), we first stepwisely adjusted the number of summary graph partitions to find the value of $|V_S|$ that minimized the geometric mean of the queries' runtimes. LUBM-160 consists of $|E_D| = 27.9 \cdot 10^6$ triples with an average node degree of $d = 3.6$, and by varying $|V_S|$, we determined the best number of summary graph partitions to lie at around $|V_S| = 17k$ partitions. Thus, plugging the above values into Equation (1) for a cluster of $n = 5$ slaves, we obtain a value of $\lambda = 187$. We next use this value of λ to predict the best number of partitions for the LUBM-10240 setting (using the same queries), which consists of $|E_D| = 1.7 \cdot 10^9$ triples. Equation (1) predicts $|V_S| = 136k$ partitions, which is very well within the range of the actual best number of partitions, which we again manually determined to lie in between $100k$–$200k$ partitions (see Figure 6.A.4).*

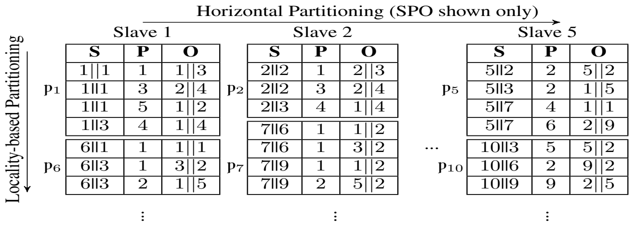
5.2 Encoding Triples

After determining the summary graph partitions that each distinct subject s and object o in the RDF data graph belongs to, the master node encodes the partitioning information directly into these triples. For this, let $\langle s, p, o\rangle$ denote a triple in the RDF data graph, and let $\langle \mathrm{p}_1, p, \mathrm{p}_2\rangle$ be its corresponding triple in the summary graph. We then obtain the final encoding of triples in the RDF data graph as $\langle \mathrm{p}_1||s, p, \mathrm{p}_2||o\rangle$. The integer ids of s and o are obtained by maintaining one separate dictionary (a hash map) per summary graph partition; the ids of p and p_1, p_2 are available from the intermediate dictionary and the summary graph itself.

EXAMPLE 3. *Following the summary graph shown in Figure 1, the triple* $\langle$`Barack_Obama`, `bornIn`, `Honolulu`$\rangle$ *is encoded as follows. The subject* `Barack_Obama` *is encoded as* $1||1$*, the predicate as* 1*, and finally the object as* $1||2$*, thus yielding* $\langle 1||1, 1, 1||2\rangle$ *(stored as a struct of integers) as the final encoding for this triple.*

5.3 Horizontal Partitioning of Data Triples

As with any distributed system, we partition the set of encoded RDF data triples across the slaves. Our horizontal partitioning scheme aims to preserve the locality information obtained from the summary graph by hashing entire summary graph partitions into the grid-like distribution scheme shown in Figure 3. Since each combined $\mathrm{p}_1||s$ and $\mathrm{p}_2||o$ identifier contains information about both the summary graph partition and the actual subject and object identifiers, we can now "shard" these triples as follows. Let $\langle \mathrm{p}_1||s,\ p,$


Horizontal Partitioning (SPO shown only)
Locality-based Partitioning
Slave 1
Slave 2
Slave 5
...

Slave 1	S	P	O
p_1	1‖1	1	1‖3
	1‖1	3	2‖4
	1‖1	5	1‖2
	1‖3	4	1‖4
p_6	6‖1	1	1‖1
	6‖3	1	3‖2
	6‖3	2	1‖5
	⋮		

Slave 2	S	P	O
p_2	2‖2	1	2‖3
	2‖2	3	2‖4
	2‖3	4	1‖4
p_7	7‖6	1	1‖2
	7‖6	1	3‖2
	7‖9	1	1‖2
	7‖9	2	5‖2
	⋮		

Slave 5	S	P	O
p_5	5‖2	2	5‖2
	5‖3	2	1‖5
	5‖7	4	1‖1
	5‖7	6	2‖9
p_{10}	10‖3	5	5‖2
	10‖6	2	9‖2
	10‖9	9	2‖5
	⋮		

Figure 3: Locality-based & horizontal partitioning of triples

$\mathrm{p}_2||o\rangle$ be an encoded RDF triple, and let n be the number of slaves. Then each RDF triple is sharded twice, once by sending it to slave $(\mathrm{p}_1 \mod n)$ and once by sending it to slave $(\mathrm{p}_2 \mod n)$.

EXAMPLE 4. *Consider the two triples* ⟨`Barack_Obama`, `won`, `Nobel_Peace_Prize`⟩ *and* ⟨`Barack_Obama`, `bornIn`, `Honolulu`⟩ *shown in Figure 1. Here,* `Barack_Obama` *and* `Honolulu` *belong to Supernode* 1 *and* `Nobel_Peace_Prize` *belongs to Supernode* 4*. Considering a cluster of 5 slaves, we distribute the first triple onto Slaves* 1 *and* 4*, whereas the second triple is hashed twice (but sent only once) to Slave* 1*.*

Locality-Based Sharding and Join-Ahead Pruning. As opposed to the random partitioning schemes used, e.g., in [17], our hashing scheme aims to preserve the locality information provided by the summary graph. Triples belonging to the same supernode are placed on the same horizontal partition which facilitates join-ahead pruning of partitions that do not contain any triples that are relevant with respect to an entire query. From Example 1, assume we know that only partitions p_1, p_2, p_4 are relevant for processing the SPO permutation of the query triple `?person <bornIn> ?city` because only p_1, p_2, p_4 are bound to the subject `?person` after processing the entire example query against the summary graph shown in Figure 1. As shown in Figure 3 (and from Example 4), only the first block each at Slaves 1, 2 and 4 thus is relevant for scanning the SPO permutation for this triple in this query.

5.4 Local Permutation Indexes

Upon receiving the sharded triples from the master, the slaves start creating their local permutation indexes in parallel. Each slave creates six large, in-memory vectors of triples, which will serve as our primary index structure for processing queries. Each of the six vectors corresponds to one SPO permutation of the three encoded $\langle \mathrm{p}_1||s, p, \mathrm{p}_2||o\rangle$ fields. For fast lookups of a given query triple with a set of supernode ids selected from the summary graph, we define methods for random access (via binary search) and sequential access (in the form of iterators) on top of these vector-based SPO lists. Figure 3 depicts an example of these SPO indexes at the slaves.

SPO Indexes. At each slave, the six SPO permutations are arranged into two groups: i) the *subject-key* indexes (SPO, SOP, PSO), and ii) the *object-key* indexes (OSP, OPS, POS). All triples hashed onto a slave node via their subject field $\mathrm{p}_1||s$ are added to the node's subject-key indexes. Likewise, triples hashed by their object field $\mathrm{p}_2||o$ are added to the node's object-key indexes. This way, each encoded RDF triple is replicated exactly six times across the compute cluster. At each slave, the three subject-key and the three object-key vectors have exactly the same size, respectively.

Sorting Triples. Each of the six triple vectors at a slave is sorted in lexicographic order with respect to its corresponding permutation of the $\langle \mathrm{p}_1||s, p, \mathrm{p}_2||o\rangle$ fields. The grid structure shown in Figure 3 thus preserves both locality information (i.e., the graph partitions) of the summary graph and guarantees coherence of triples with the same subjects, objects and predicates, respectively.

5.5 Local & Global Statistics

In order to create efficient join plans, we compute multiple statistics over both the data and the summary graph. These statistics include i) cardinalities of individual $p_1||s$ (subject), p (predicate), and $p_2||o$ (object) arguments in case of the data graph and ii) cardinalities of individual p_i (supernode), p (predicate) arguments in case of the summary graph. In addition, as in [14], we store cardinalities of iii) $(\mathrm{p}_1||s, \mathrm{p}_2||o)$ (subject, object), iv) $(p, \mathrm{p}_2||o)$ (predicate, object), v) $(p, \mathrm{p}_1||s)$ (predicate, subject), and vi) selectivities of (p_1, p_2) (predicate, predicate) pairs as part of the data graph statistics. We follow a similar approach for the summary graph and also store the cardinalities of individual vii) (p, p_i) (predicate, supernode) and viii) selectivities of (p_1, p_2) (predicate, predicate) pairs.

These statistics can only provide us with an exact cost for the first series of index scans, while cardinalities for joins need to be approximated. Estimating the cost of an entire query plan thus requires the recursive estimation of the cardinalities of intermediate relations obtained from joins, which can be formalized as

$$Card(R_{1,2}) := Card(R_1) \cdot Card(R_2) \cdot Sel(R_1, R_2) \qquad (2)$$

where $Sel(R_1, R_2)$ denotes the selectivity of the pair of predicates (p_1, p_2) associated with the triple patterns R_1 and R_2, respectively. The selectivities for the entire RDF data graph are first aggregated locally at the slaves (in the form of absolute cardinalities) and then merged globally at the master, while the ones for the summary graph are aggregated at the master node, only.

EXAMPLE 5. *For the triple patterns* R_1 : ⟨`?person`, `bornIn`, `?city`⟩ *and* R_2 : ⟨`?city`, `locIn`, `USA`⟩*, we store the cardinalities* $Card(R_1) = 4$ *and* $Card(R_2) = 5$ *at the master node. Similarly, we store the selectivity* $Sel(R_1, R_2) = 0.2$ *for the pair of predicates* (`bornIn`, `locIn`)*. From Equation* (2)*, we thus obtain* $Card(R_{1,2}) = 4$ *as the estimated number of joined triples.*

6. QUERY OPTIMIZATION & DISTRIBUTED PROCESSING

In this section, we present a detailed description of the two-staged optimization and processing strategy we follow in TriAD.

6.1 Two-Staged Query Processing Overview

A SPARQL query is parsed and translated into a query graph of the form $G_Q(V_Q, E_Q, L, \mathit{Vars}, \phi_Q)$ (see Definition 2) by assigning a unique id to each distinct variable in Vars, while constants in L are replaced by ids obtained from the forward dictionary. In the following, we refer to $E_Q = \{R_1, R_2, \dots, R_n\}$ as the set of query triple patterns that capture a conjunctive query pattern.

Stage 1. The first stage, called "pruning stage", is performed entirely at the master node. We first process the query against the summary graph to find bindings of supernode identifiers to query variables. For this, we employ an exploratory algorithm (similar to the one described in [2, 30]) for finding these supernode bindings. The reason behind choosing an exploratory-based algorithm over conventional joins is that, here, our objective is to only find supernode bindings for each query variable to facilitate join-ahead pruning at the actual SPO permutation indexes. For an efficient graph exploration, we determine the best exploration order, the *exploratory plan*, using a first DP-based optimizer over the summary graph statistics. The supernode bindings obtained from the pruning stage are relayed to the physical operators at the second stage.

Stage 2. In the second stage, we process the query against the data graph which is distributed across all slaves. Here, we follow a relational style of processing, aiming to generate the final join results of the SPARQL query. We determine the best join order by using a second DP-based optimizer (see, e.g., [14]) in combination with a

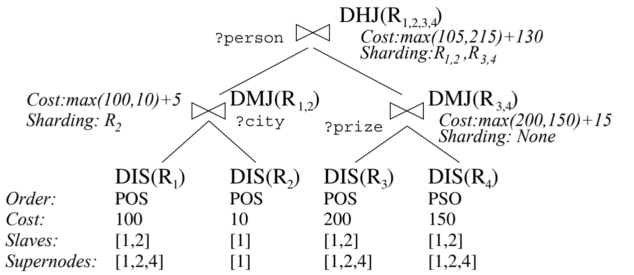


Figure 4: Global query plan for the query of Example 6

distribution-aware cost model as objective function. The supernode bindings obtained at the pruning stage are also used to (re-)estimate the cardinalities of input relations and are fed into the cost model for optimization. This *global query plan* generated at the master is then communicated to all slaves. Along with the global plan, the supernode bindings from Stage 1 are passed on to the slaves for pruning dangling triples (i.e., entire summary graph partitions) from the SPO permutation indexes. At each slave, the local query processor executes the plan by asynchronously sending and/or receiving intermediate join results to/from the other nodes. When query processing terminates, each slave holds its own partial query results which are then finally merged at the master.

6.2 Generating Supernode Bindings

The first stage of processing generates supernode bindings via a graph exploration approach. However, unlike the simpler 1-hop graph exploration described in [30], we perform a full graph exploration with back-propagation. That is, we add a supernode binding to a join variable only if this binding satisfies the entire query also with respect to the remaining join variables.

EXAMPLE 6. *Consider a SPARQL query consisting of the following four triple patterns R_1 to R_4.*

R_1 : `?person <bornIn> ?city.`
R_2 : `?city <locatedIn> USA.`
R_3 : `?person <won> ?prize`
R_4 : `?prize <hasName> ?name.`

For the fixed exploration order $\langle R_1, R_2, R_3, R_4\rangle$, we first find all possible bindings for variables `?person` *and* `?city` *for the query pattern R_1. Next, for the second query pattern R_2, we prune those bindings for variable* `?city` *that are not located in "*`USA`*". We propagate this information back to* `?person` *and thus prune supernode bindings for* `?person`*. Finally, with query patterns R_3, R_4, we filter out the bindings for variables* `?person`*,* `?city` *and accordingly add new bindings to* `?prize` *and* `?name`*.*

Exploratory Plan Optimization

A random exploration order of query patterns might make the summary graph processing inefficient and sometimes even slower than processing the data graph. To avoid this, we estimate the best exploration order by leveraging the summary graph statistics. To do so, we employ a first bottom-up DP algorithm to determine the order of triple patterns that yields the overall least cost estimate. At each DP step, we calculate the cost of the partial plan considered so far and prune if the current branch cannot contribute to the plan with the least cost anymore. Based on Equation (2), the cost of an entire exploration plan that is represented by a fixed order of triple patterns $R_1, \ldots, R_n$ can thus be estimated as follows.

$$Cost(\langle R_1, \ldots, R_n\rangle) \propto$$
$$Card(R_1) + \sum_{i=2}^{n} \Big(Card(R_i) \prod_{j=1}^{i} Sel(R_i, R_j) \Big) \quad (3)$$

Here, $Card(R_i)$ denotes the precomputed cardinality of query pattern R_i, and $Sel(R_i, R_j)$ represents the join selectivity of pairs of predicates (p_i, p_j) associated with triple patterns R_i, R_j, respectively (Section 5.5). This selectivity is set to 1 if R_i and R_j do not share any join variable. We remark that this estimation again assumes independence among join patterns.

6.3 Querying the Data Graph

With the supernode bindings at hand, Stage 2 of the query evaluation is performed over the indexed and sharded RDF data graph. Since there exist six SPO permutations of the entire RDF data graph, which are distributed across n slaves, each individual query pattern R_i could potentially be scanned in six different ways, and each such scan can be done in parallel across the slaves.

Physical Operators. Inspired by the reduced set of query operators in RDF-3X [14, 15], we employ only three distributed operators to construct a query plan in TriAD:

- **Distributed Index Scan (DIS)**: Invokes a parallel scan over a permutation list that is sharded across n slaves.
- **Distributed Merge Join (DMJ)**: Invokes a distributed merge-join across n slaves when both input relations are sorted according to the join key(s) in the query plan.
- **Distributed Hash Join (DHJ)**: Invokes a distributed hash-join across n slaves otherwise.

Each physical DIS operator is aware of the *locality* of the sharded list it scans, the *permutation order* chosen by the optimizer, and the *pruned summary graph partitions* determined by Stage 1. Moreover, both the DMJ and DHJ operators are aware of the *locality* of their input relations and their *join conditions* (see Figure 4).

Query-Time Sharding. Both the DMJ and DHJ operators may require sharding a relation at query time. Due to our index layout, the DMJ operator requires sharding of only at most one input relation R_i obtained from a DIS operator when R_i's triples were previously sharded (Section 5.3) on a non-join key. For instance, consider the left-hand DMJ shown in Figure 4. Here, using a DIS over the POS index yields all triples for R_2 whose objects are bound to "`USA`". Since R_2's object is not a join key for the left-hand DMJ, we need to shard R_2's triples according to the join key `?city` (the subject of R_2). On the other hand, the right-hand DMJ operator requires no query-time sharding at all when scanning the POS and PSO indexes, respectively, since both R_3's and R_4's triples were sharded on the join key `?prize`. Likewise, the upper DHJ operator requires sharding both of its intermediate input relations, since $R_{1,2}$ and $R_{3,4}$ are not sorted on their common join key `?person` and thus are misplaced among the slaves with respect to this key.

Global Query Plan Optimization

The choice of a physical join operator strongly influences the cost function determined by the DP optimizer. To initialize the DP table for each pattern R_i and SPO permutation k, which is distributed across n slaves, we set the DIS cost $Cost(R_i^k) \propto Card(R_i)/n$ if permutation k matches the binding pattern given by the constants in R_i; and we set it to be proportional to $|E_D|/n$ otherwise. For example, for the query pattern $\langle$`Barack_Obama`, `?p`, `?o`$\rangle$, the cost of scanning the matching triples over the SPO, SOP permutations is expected to be much lower compared to scanning them over the OPS, OSP, PSO and POS permutations. For calculating the actual costs $Cost(R_i^k)$ of an index scan, we multiply the basic cardinalities with a constant cost factor η^{DIS}.

After initializing the DP table with the first series of DIS costs, we continue to build a query plan that aims to reflect the optimal order of both *joining* and *shipping* intermediate results across the slaves. At each DP step, we join two subplans over two non-overlapping subqueries Q^{left} and Q^{right} into a new combined plan Q. The cost of Q is then recursively defined as follows.

$$Cost(Q) := \begin{cases} Cost(R_i^k) & \\ \quad \text{if } R_i \text{ denotes a DIS over permutation } k; & (4.1) \\ Cost(Q^{left}) + Cost(Q^{right}) & \\ \quad + Cost(Q^{left} \bowtie^{op} Q^{right}) & \\ \quad + Cost(Q^{left} \rightleftharpoons^{op} Q^{right}) \text{ otherwise.} & (4.2) \end{cases}$$

Here, $Cost(Q^{left} \bowtie^{op} Q^{right})$ denotes the cost of joining the two subqueries via operator *op*, which depends on the cardinalities of both Q^{left}, Q^{right} times a constant cost factor η^{op} for the respective join operator $op \in \{\text{DMJ}, \text{DHJ}\}$. Conversely, $Cost(Q^{left} \rightleftharpoons^{op} Q^{right})$ denotes the cost of shipping intermediate relations for Q^{left}, Q^{right} across the slaves before executing the actual join. This is again computed from the cardinalities of Q^{left}, Q^{right}, which are each multiplied with the width of their intermediate relations and a constant factor $\eta^{\rightleftharpoons}$ for the communication cost.

Cardinality (Re-)Estimation. Equation (4.1) captures the scan costs for a basic triple pattern R_i to be proportional to the cardinality that is available from our precomputed global statistics. Preprocessing the query against the summary graph however lets us refine these cardinalities by the amount of summary graph partitions that are actually selected for each R_i after the initial graph exploration step. Thus, let $Card(R_i)$ be the precomputed cardinality of a query pattern R_i over the RDF data graph, and let $|C_s|$, $|C_o|$ be the cardinalities of its subject s and object o, respectively, obtained from the precomputed summary graph statistics. Let $|C'_s|$, $|C'_o|$ be the number of supernode bindings obtained from Stage 1 of processing the query over the summary graph. We then (re-)estimate $Card'(R_i)$ via a simple linear interpolation as follows.

$$Card'(R_i) := \frac{|C'_s|}{|C_s|} \cdot \frac{|C'_o|}{|C_o|} \cdot Card(R_i) \tag{4}$$

These re-estimated cardinalities are plugged into Equation (4.1) and used by the optimizer when determining the global query plan.

Accounting for Parallel Operations. To accommodate for the parallel execution of two subplans Q^{left}, Q^{right} (Section 6.4), we further refine the cost function of Equation (4.2) as follows.

$$\begin{aligned} Cost(Q) \;\; := \;\; & \max\Big(Cost(Q^{left}), Cost(Q^{right})\Big) \\ & + \; Cost(Q^{left} \bowtie^{op} Q^{right}) \\ & + \; Cost(Q^{left} \rightleftharpoons^{op} Q^{right}) \end{aligned} \tag{5}$$

That is, at any DP step, the cost of the current (sub-)plan for Q is proportional to the cost of the concurrent execution of the subplans for Q^{left}, Q^{right}, rather than to the cost of their sequential execution. Another significant advantage of parallel executions—in addition to speeding up computations—is that it also better exploits the network bandwidth by sending more than one intermediate relation at a time via asynchronously exchanged messages.

EXAMPLE 7. *Figure 4 shows an example of a global plan returned by the optimizer for a two-node distribution. One can observe that the plan explicitly includes the locality and pruning information that each DIS operator has at the leaves. For instance, the POS list chosen for pattern R_2 entirely resides at Slave 1, whereas the ones for R_1, R_3, R_4 are distributed across both slaves. The plan also shows how the parallel execution of subplans affects the cost estimates for the DMJ and DHJ operators.*

6.4 Distributed Query Execution

The global query plan generated at the master is communicated to all slaves along with the supernode bindings. Each slave receives this plan, initializes its own instances of the physical operators (but over different chunks of the sharded SPO lists), and then starts processing the plan concurrently. The protocol that is executed at each of the slave nodes concurrently is shown in Algorithm 1.

Algorithm 1: Local query processor at Slave i

```
   Input: Global query plan with supernode bindings $Plan$;
   local SPO index $Idx$; number of slaves $n$;
   Output: Relation with partial query results $Relation$;
1  method Main($Plan$, $Idx$, $n$, $i$) {
2    $EP[1..l]$ ← CreateExecutionPaths($Plan$); //plan with $l$ leaf op's
3    for $j = 1..l$ do
4        START_THREAD(($EP[j]$, $Idx$) → Process);
5    $Alive[i]$ ← SendSlaveStatusToMaster($i$);
6    WAIT_ALL($EP[1..l]$); //synchronize on execution paths
7  return $EP[1].Relation$; } //return partial result relation for this slave
8  method Process($EP$, $Idx$) {
9    while $Op$ ← NextOperator($EP$) do
10       if $Op$ is DIS then
11           $SN[1..p]$ := GetSupernodeBindings($Op$); //for join-ahead pruning
12           $EP.Relation$ ← GetIterator($Op$, $Idx$, $SN[1..p]$); //binary search
13       else
14           $Alive[1..n]$ ← ReceiveSlaveStatusFromMaster();
15           if $Op.Sharding$ then
16               $Part[1..n]$ ← Shard($EP.Relation$); //repartition relation
17               $EP.Relation$ ← $Part[i]$; //keep partition $i$ locally
18               for $j \neq i$ && $Alive[j]$ do
19                   $Ack[j]$ ← MPI_Isend($Part[j]$, $j$, $EP.Id$);
20               for $j \neq i$ && $Alive[j]$ do
21                   $Ack[n+j]$ ← MPI_Ireceive($Part[j]$, $j$, $EP.Id$);
22                   $EP.Relation$ ← Merge($EP.Relation$, $Part[j]$);
23               WAIT_ALL($Ack[1..2n]$); //synchronize on incoming messages
24           $SibEP$ ← FindSiblingExecutionPath($Op$);
25           $R_1$ ← $EP.Relation$;
26           $R_2$ ← $SibEP.Relation$;
27           if $SibEP.Id < EP.Id$ then
28               STOP_THREAD($EP$);
29           $EP.Relation$ ← Join($R_1$, $R_2$,$Op$); } // $Op$ is DMJ or DHJ
```

Multi-Threaded, Asynchronous Plan Execution. The key to allow for a parallel, asynchronous execution of the global query plan lies in executing the plan in a multi-threaded fashion at each slave. The Main method of Algorithm 1 invokes a new thread (using the C++ Boost API) for each sequential *execution path* (EP) of operators in the query plan. An EP is a path from a leaf of the operator tree up to its root (the vertical dashed lines in Figure 5). At each slave, we start a separate thread for each such EP (Line 3), and we later join (i.e., synchronize) two threads into one at the lowest common join operator that two such EPs share (Line 27).

As shown in the Process method (and in Figure 5), the execution of an EP always starts with the DIS operators. Each DIS operator obtains the respective supernode bindings for join-ahead pruning as part of the global query plan (Line 11). Instead of building an intermediate relation, a DIS operator returns an iterator that directly points to the first qualifying tuple (obtained via binary search and the supernode bindings) in a sorted SPO permutation list (Line 12). These iterators are then passed to the parent DMJ operators to perform the joins directly on the raw indexes. Otherwise, if the operator is DMJ or DHJ and sharding is required, Slave i first shards the intermediate relation that it holds at its current EP (Line 16). Only in this case, Slave i needs to synchronize on incoming messages (Line 20) from the other $n-1$ slaves in order to merge the incoming (resharded) tuples into the current EP's intermediate relation (Line 22). Thus, each EP holds an intermediate relation which is iteratively passed on to a subsequent join operator (Line 29) in the execution path. Although the operators within an EP are executed sequentially, multiple such EPs (and thus operators) run in parallel and asynchronously at each slave, and across all slave nodes.

Asynchronous MPI Communication. Sharding an intermediate relation for a DMJ or DHJ operator at query time is a blocking op-

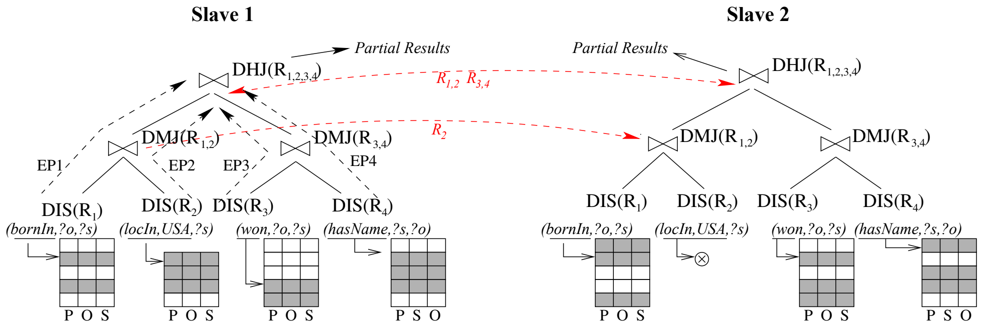


Figure 5: Distributed execution of the query shown in Example 6 with asynchronous communication (horiz. dashed lines)

eration that requires a synchronization step among all slaves. This may be a significant bottleneck, as it blocks the slaves from performing their partial join operations until all the slaves have received their corresponding chunks of tuples. We address this by using the asynchronous `MPI_Isend` and `MPI_Ireceive` methods of the MPICH2 API (Lines 19 & 21). Thus, without waiting for the entire sharding phase to finish among all slaves, a part of a DMJ or DHJ operation can be invoked locally on a slave as soon as this slave has received the $n - 1$ messages with the chunks of tuples it is responsible for (denoted by the horizontal dashed lines in Figure 5). Conversely, once a slave finishes all its execution paths, it broadcasts its completion to all other slaves via the master.

In summary, if query-time sharding is required prior to a join, then this step is comparable to a "Shuffle&Sort" phase of a Map-side join in MapReduce [10]. In our case, shuffling is not always required, and sorting is avoided entirely. Due to the layout of our distributed index structures (Sections 5.2 & 5.3), we can always rely on efficient DMJ operators for the first level of joins. At this first level, we need query-time sharding only if we join the subject of one query pattern on the object of another query pattern (i.e., we have an S-O or O-S join) and at least one of the non-joining subjects or objects is a constant. Conversely, a DHJ operator requires query-time sharding of either one (or both) of its input relations. During plan generation, this is taken into consideration by the optimizer together with the constant cost factors of the operators, such that we favor merge joins over hashing whenever possible.

EXAMPLE 8. *Figure 5 illustrates the distributed execution of the query plan depicted in Figure 4 (and shown in Example 6) for a two-node distributed setup. At the leafs, the DIS operators (e.g., for R_1) obtain the supernode bindings and each create an iterator over the pruned POS index (shaded partitions). Before invoking the left-hand DMJ on `?city`, and since R_2 at Slave 2 is empty, we repartition the triples of R_2 at Slave 1 into two partitions, one of which is sent to Slave 2 (denoted by a horizontal dashed line). The two right-hand DMJs on `?prize` at Slaves 1 and 2 require no communication, as their input triples are already in-place. Since the two DMJs order tuples on different join keys for `?city` and `?prize`, only the final DHJ requires sharding and shipping for both $R_{1,2}$ and $R_{3,4}$ for the join on `?person`. All of the DIS operators are performed in a fully distributed and multi-threaded fashion. Also the next level of DMJ runs in an asynchronous fashion. Only the final DHJ needs to wait until both of its incoming DMJ operators have finished generating their intermediate results.*

7. EVALUATION

We evaluated TriAD in comparison to two centralized RDF engines, RDF-3X [14] and BitMat [2], four distributed RDF engines, SHARD [17], H-RDF-3X [8], 4store [7] and the very recent Trinity.RDF [30] engine, one main-memory DBMS, MonetDB [22], as well as to Apache's Hadoop and Spark engines (the latter for comparing to their plain join performance). To study join-ahead pruning, we consider two variants of our system. The first, referred to as TriAD-SG, makes use of the summary graph, while the second, referred to as TriAD, performs a random partitioning of triples.

Benchmarks. We used the widely popular LUBM[5] synthetic benchmark, the real-world BTC 2012[6] dataset, and the recent WSDTS[7] SPARQL diversity test suite. For LUBM, we employed the data generator using UBA 1.7 in N3 format. Concerning the queries, we used the benchmark queries published in [2] and used by Trinity.RDF [30]. For BTC, we defined 8 queries of varying complexity similar to the ones published in [13], replacing only the operators that TriAD currently does not support (i.e., `DISTINCT` and `[]`).

TriAD Setup. We implemented TriAD in C++ using GCC-4.4 with -O3 optimization. We used MPICH2-1.4.1 and Boost 1.54.0 as external libraries. For TriAD-SG, we constructed our summary graph by partitioning the RDF data graph using the METIS 5.1 graph partitioner with a default configuration. To achieve a better performance during partitioning, we ignored edges connecting string literals, resulting in both time and space savings. We ran all experiments on a local compute cluster with 12 nodes (one of which was dedicated as the master node) which are connected via a 1GBit LAN connection. Each machine has 48GB of RAM, 16 quad-core CPUs of 2.4GHz, and runs Debian Linux 6.0.6.

Competitor Setup. To compare against Hadoop-based engines, we implemented H-RDF-3X [8] as our main competitor. For H-RDF-3X, we first partition the graph using METIS and assign each partition to a slave that runs RDF-3X 0.3.7 as its local RDF engine. For a fair comparison, and given that all LUBM queries have a diameter of less than 2, we employ a 1-hop replication. Moreover, since neither Trinity.RDF [30] nor its underlying Trinity graph engine [21] are openly available, Tables 1 and 4 thus depict the running times reported in [30] for the same benchmark setting but over a much stronger hardware configuration. Most notably, our available network bandwidth and main memory lie at 1GBit and 48GB as opposed to 40GBit and 96GB reported in [30], respectively. All other competitors are off-the-shelf installations within our cluster.

7.1 Results – LUBM-10240 Dataset

In our first series of experiments, we use the LUBM-10240 dataset which consists of about 1.84 billion triples (amounting to a size of 730 GB in raw N3 format). Queries $Q1$–$Q7$ [2, 30] can be classified as non-selective ($Q2$), selective in both the input relations and output size ($Q4$, $Q5$, $Q6$), and selective in output size ($Q1$, $Q3$, $Q7$). This setup is identical to the one used for evaluating Trinity.RDF [30], thus allowing us draw a careful comparison to their performance results. In Table 1, we depict the wall-clock processing times of both TriAD and TriAD-SG in comparison to all competitors. For TriAD-SG, we experimented with different summary graph sizes. Table 1 depicts our best setting, where 200,000 supernodes and 130,744,241 superedges reside at the master node.

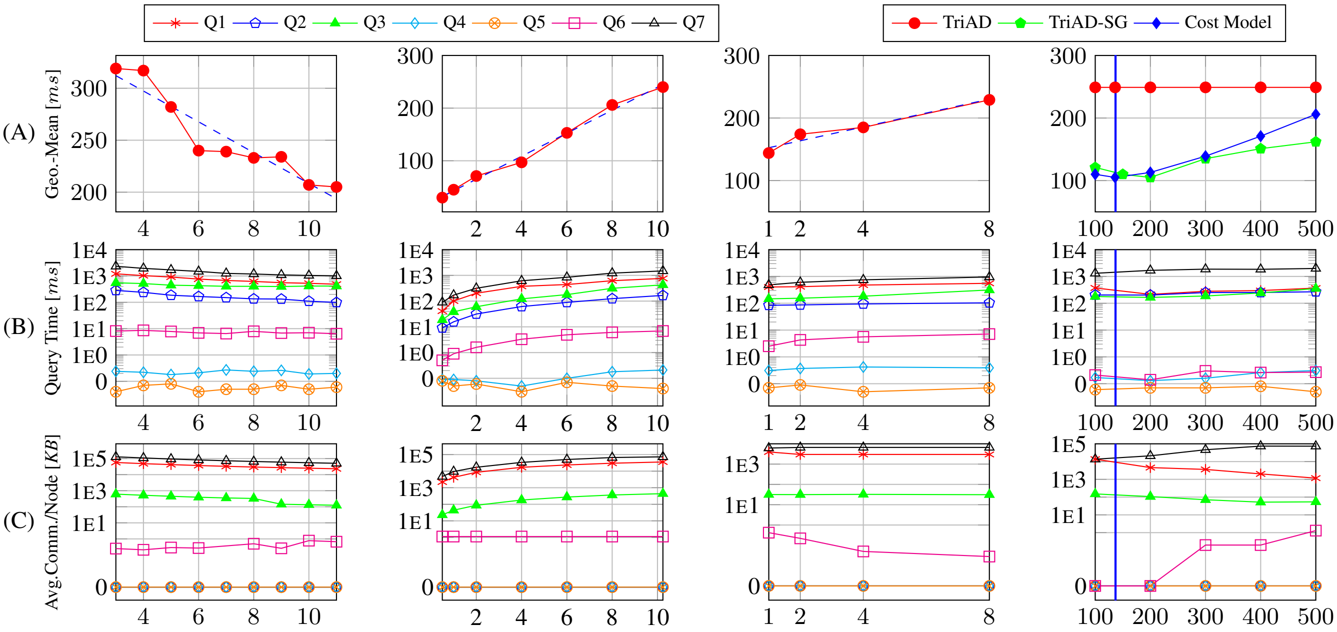
[5] `http://swat.cse.lehigh.edu/projects/lubm/`

[6] `http://challenge.semanticweb.org/`

[7] `https://cs.uwaterloo.ca/~galuc/wsdts/`

	TriAD	TriAD-SG (200K)	Trinity.RDF	SHARD	H-RDF-3X (cold)	H-RDF-3X (warm)	4store (cold)	4store (warm)	RDF-3X (cold)	RDF-3X (warm)	BitMat (cold)	BitMat (warm)
Q1	7,631	**2,146**	12,648	6.9E5	2.3E6	1.7E5	aborted	aborted	1.9E6	1.8E6	17,339	11,295
Q2	1,663	2,025	6,018	2.1E5	5.3E5	4,095	1.1E5	15,113	6.3E5	17,835	2.4E5	1.8E5
Q3	4,290	**1,647**	8,735	4.7E5	2.2E6	1.3E5	aborted	aborted	1.7E6	1.7E6	8,429	2,679
Q4	2.1	**1.3**	5	3.9E5	166	1	1,903	12	243	3	aborted	aborted
Q5	**0.5**	0.7	4	97,545	85	1	2,429	12	99	1	472	338
Q6	69	**1.4**	9	1.8E5	5.8E5	23,440	3,572	9	913	287	7,796	5,377
Q7	14,895	16,863	31,214	3.9E5	2.3E6	2.1E5	aborted	aborted	6.5E5	46,262	71,157	36,905
Geo-Mean	249	**106**	450	3.0E5	91,378	2,406	-	-	31,345	2,991	-	-

Table 1: LUBM-10240 – Query processing times (in *ms*)



(1) # Slaves (2) Data Size [k Univ.] (3) Weak Scalability [# Slaves,k Univ.] (4) Summary Size [k Part.]

Figure 6: TriAD (Cols. 1–3) & TriAD-SG (Col. 4) scalability experiments for various configurations of the LUBM benchmark

Starting from the non-selective query $Q2$, which contains a single join that returns a large number of results, TriAD outperforms all competitors, thus taking advantage of its distributed join execution. Trinity.RDF is about 3 times slower, since here graph-exploration provides no benefit for non-selective queries, thus retaining many bindings and performing a single, centralized join at the master node. H-RDF-3X, due its use of local RDF stores, can execute the join in parallel and runs faster than Trinity.RDF (warm cache) but due to the unbalanced partition sizes across the local RDF stores, H-RDF-3X remains slower than TriAD. In addition, TriAD, which uses main-memory backed indexes, can perform fast random-access jumps over its indexes. Here, the use of the summary graph (see TriAD-SG) even slightly hurts performance, since $Q2$ does not benefit from join-ahead pruning either.

For the selective queries $Q1$, $Q3$, $Q7$, TriAD manages again to outperform Trinity.RDF. The slower performance of Trinity.RDF apparently is due to its 1-hop distributed graph exploration method without back-propagation (which we conclude from the observation that these queries are only selective in their final output but non-selective for the lowest level of joins). For both $Q1$ and $Q3$, the summary graph with a full graph exploration (including back-propagation) improves the performance of TriAD, since pruning is very effective for these selective queries. Especially for $Q3$, which has an empty result, the summary graph prunes many SPO partitions which leads to performance gains over Trinity.RDF. This impact of full graph exploration is also shown by the centralized BitMat system which is faster than TriAD but slower than TriAD-SG. For query $Q7$, the pruning stage in TriAD-SG is not as effective, thus retaining many SPO partitions and resulting in an inferior performance compared to TriAD due to the overhead of shipping and comparing the supernode identifiers for our index scans. 4store repeatedly crashed on queries $Q1$, $Q3$, $Q7$ (marked as "aborted").

Queries $Q4$, $Q5$, which are processed against many low-cardinality input relations, can be considered as the best cases for effective join-ahead pruning. For these queries, the centralized RDF-3X engine with join-ahead pruning is very efficient. TriAD is slightly faster than RDF-3X (warm cache) and Trinity.RDF by using distributed joins with skip-ahead jumps over the index lists based on the supergraph partitions. In the case of TriAD-SG, where the first-stage processing is negligible, it performs similarly to TriAD.

Trinity.RDF performs better than TriAD for $Q6$, where large intermediate relations hamper the performance of TriAD. The use of the summary graph in TriAD-SG however is almost 50 times faster, thus reducing the size of the intermediate results significantly and outperforming Trinity.RDF. H-RDF-3X performs significantly worse in this case, since it breaks the query into smaller subqueries and fails to capitalize on the SIP benefits of RDF-3X.

Scalability. We studied both the strong and weak scalability of TriAD by increasing the number of available machines and the size of the data set. The results are shown in Figure 6. Figures 6.C.1 and 6.B.1 show the strong scalability in terms of query time when increasing the number of slaves from 2 to 11. Figure 6.C.1 shows the average communication costs per slave for this increasing number of slaves. For measuring strong scalability, we use the LUBM-

10240 dataset. (We omit the setting with a single slave as our indexes and statistics do not fit into 48GB of RAM.) We observe that our processing time decreases linearly as the number of machines increases and, as expected, we see the average communication cost per slave decreasing while the total communication cost is increasing. We also studied how TriAD performs as we increase the size of the data while keeping the number of machines fixed. Results are shown in Figures 6.A.3, 6.B3 and 6.C.3 and imply a very good scalability for TriAD with respect to the data size. Similarly, Figures 6.A.2, 6.B.2, 6.C.2 depict the case when we increase both the size of the data and the number of available machines. From the geometric means in Figure 6.A.2, we can observe that the variance is very low, thus confirming the afore behavior also in terms of weak scalability. Notice that the join multiplicities for $Q1$–$Q7$ are larger than 1, such that the result sizes also grow super-linearly.

Communication Costs. With regard to the communication costs among slaves, our measurements for LUBM-10240 are shown in Table 2 (in *KB*). The use of the summary graph generally achieves a better query performance by reducing the size of the intermediate results via join-ahead pruning, thus decreasing both the communication costs and the computational costs of the joins. The maximum gains appear for selective queries $Q1$, $Q3$, and $Q7$.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7
TriAD (*KB*)	35,720	0	439	<0.1	<0.1	1.13	73,141
TriAD-SG (*KB*)	4,587	0	107	0	0	0	21,051

Table 2: Communication costs for LUBM-10240

Impact of Summary Graph Size. The query times and the average communication costs for queries $Q1$–$Q7$ for different summary graph sizes are shown in Figures 6.A.4, 6.B.4 and 6.C.4. With increasing summary graph sizes, we generally observe increasing query times, which become dominated by processing the queries against the summary graph. We can also observe a decreasing trend for the communication costs (except for $Q7$) because of more pruning. Figures 6.A.4, 6.B.4 and 6.C.4 show the optimal number of partitions predicted by our cost model (blue vertical line). The TriAD baseline (red horizontal bar) is shown in Figure 6.A.4. The cost predicted by our cost model (green curve in Figure 6.A.4, see also Section 5.1) has been scaled linearly to fit this plot, which however does not affect the shape of the plot nor its minimum.

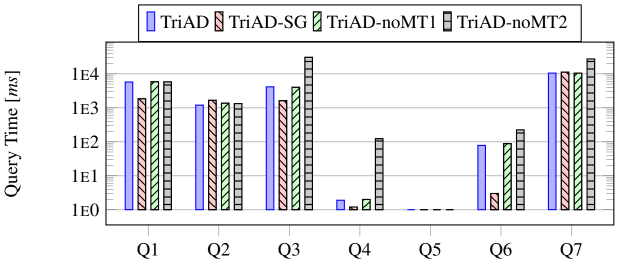


Figure 7: Impact of multi-threading in TriAD

Impact of Multi-Threading. We evaluated the gain of multi-threading and its effect on plan generation for the LUBM-10240 dataset on a 10-node setup. Figure 7 shows the query times of the different variants of TriAD on a logarithmic scale. To measure the effectiveness of multi-threading on both plan generation and query execution, we defined two variants: i) TriAD-noMT1 (using our multi-threading-aware cost model for optimization but single-threaded executions), and ii) TriAD-noMT2 (using a single-threaded mode for optimization and execution). For queries $Q3$ and $Q4$, allowing multi-threaded operations achieves an order of magnitude better performance results. A main reason for this large difference—besides a better CPU and network utilization—are improved query plans generated by the optimizer when multi-threading is enabled.

Relation Sizes		**Q5**	**Q2**
R_1 / R_2	LUBM-1000	10B / 3MB	9MB / 180MB
	LUBM-10240	70B / 29MB	103MB / 2GB
Query Time (in *sec*)		**Q5**	**Q2**
TriAD	LUBM-1000	<0.01	0.16
	LUBM-10240	<0.01	1.20
Apache Hadoop	LUBM-1000	21.17	29.69
	LUBM-10240	21.83	73.36
Apache Spark (cold / warm)	LUBM-1000	4.07 / 0.14	26.72 / 15.04
	LUBM-10240	9.36 / 0.48	116.25 / 96.12
MonetDB (cold / warm)	LUBM-1000	0.05 / 0.01	1.52 / 0.05
	LUBM-10240	0.11 / 0.02	26.83 / 0.23

Table 3: Single-join performance of various engines

	TriAD	**TriAD-SG**	**Trinity**	**RDF-3X**		**MonetDB**		**BitMat**	
		(17K)	**.RDF**	**(cold)**	**(warm)**	**(cold)**	**(warm)**	**(cold)**	**(warm)**
Q1	427	**97**	281	38,802	27,702	10,600	1,500	1,078	1,053
Q2	**117**	140	132	32,936	347	279	174	3,055	3,030
Q3	210	**31**	110	27,692	27,678	10,900	1,700	47	40
Q4	2	**1**	5	76	2	39	25	5,421	5,357
Q5	0.5	**0.2**	4	1	1	80	23	6	6
Q6	19	**1.8**	9	59	7	130	51	132	128
Q7	693	711	**630**	35,485	1,086	10,100	1,700	1,642	1,583
Geo.-Mean	39	**14**	46	1,280	170	748	216	277	362

Table 4: LUBM-160 – Query processing times (in *ms*)

Single-Join Performance. To evaluate the basic performance of joins in Apache Hadoop and Spark versus TriAD, we compared the built-in Map-side join function of Hadoop (over two sorted and key-partitioned input files) with the DMJ operator in TriAD. We ran the comparison over a 10-node cluster setup with two different LUBM scale factors. Table 3 shows the relation sizes and the query performance (this time in *seconds*) of Hadoop and Spark [29] for both a selective ($Q5$) and a non-selective ($Q2$) LUBM query, each consisting of just a single join operation. We can clearly observe that Hadoop-based joins should be avoided. MonetDB, in comparison, yields the by far best join performance when the input relations fit into the main memory of a single machine. It however degrades when optimizing complex SPARQL queries (see Table 4).

7.2 Results – LUBM-160 Dataset

We also evaluated the performance of TriAD and TriAD-SG over a smaller dataset. For a fair comparison, we used a single slave node setup for this, and the results are shown in Table 4. We can observe that TriAD continues to perform well for selective queries $Q4$, $Q5$, $Q6$ and the non-selective query $Q2$. For the remaining selective queries $Q1$, $Q3$, $Q7$, the large intermediate relations hamper performance, thus showing a negative impact on the centralized execution. Still, TriAD-SG benefits from join-ahead pruning and delivers a much better performance than the other systems except for query $Q7$. In this case, like in the LUBM-10240 dataset, TriAD-SG performs no pruning in the first stage and thus the overhead in the second stage marginally decreases its performance.

7.3 Results – BTC 2012 Dataset

Apart from the synthetic LUBM benchmark dataset, we evaluated TriAD over the real-world BTC benchmark. We considered queries $Q1$–$Q8$ published in [13]. Queries $Q1$, $Q2$, $Q8$ (4 joins), $Q3$ (5 joins) are star queries with result sizes of 1, 2, 1, 292, respectively. Queries $Q4$, $Q7$ (6 joins) and $Q5$, $Q6$ (4 joins) are combinations of star and path queries. Table 5 shows the performance of TriAD against the available competitors. (We omit SHARD and BitMat from the table as they failed to finish the indexing step.) We can observe that TriAD consistently outperforms the competitors. In the case of $Q6$, which has an empty result, our summary graph returns no bindings and thus entirely avoids query processing against the data graph. Also, one can observe the high running

	#Results	TriAD	TriAD-SG	H-RDF-3X		RDF-3X	
			(200K)	(cold)	(warm)	(cold)	(warm)
Q1	1	1.5	**0.3**	49	6	297	4
Q2	1	61	**3**	29	6	140	5
Q3	1	**1**	4	122	23	66	5
Q4	0	**0.6**	6	31,033	27,415	120	7
Q5	5	51	**5**	1.3E5	42,638	277	104
Q6	0	0.5	<**0.1**	5,476	153	53	24
Q7	0	50	**39**	89,922	34,906	2,900	2,386
Q8	292	128	**7**	1,338	**7**	4,590	31
Geo.-Mean	–	7.4	**1.5**	2,145	280	299	25

Table 5: BTC 2012 – Query processing times (in *ms*)

	#Slaves	L1-L5	S1-S7	F1-F5	C1-C3
		(Geo.-Mean)	(Geo.-Mean)	(Geo.-Mean)	(Geo.-Mean)
TriAD	1	**2**	**2**	94	494
TriAD-SG(75K)	1	8	4	35	767
TriAD	5	**2**	3	**29**	**270**
SHARD	5	3.2E5	5.8E5	7.1E5	7.7E5
RDF-3X (cold)	1	10,066	167	1,749	6,610
RDF-3X (warm)	1	18	**2**	41	354
MonetDB (cold)	1	3530	10,459	timeout	timeout
MonetDB (warm)	1	171	744	timeout	timeout

Table 6: WSDTS-1000 – Query processing times (in *ms*)

times for H-RDF-3X compared to RDF-3X. The reason again lies in breaking the queries into smaller queries, such that the SIP gains of RDF-3X remain under-utilized.

7.4 Results – WSDTS Dataset

We finally evaluated the performance of TriAD and TriAD-SG over the more diverse WSDTS dataset which consists of about 109 million triples. We generated 20 queries using the WSDTS query generator and categorize them into L (long path), S (star), F (snowflake) and C (complex). Table 6 shows the performance over TriAD and TriAD-SG against the available competitors. We can observe that TriAD continues to perform well for all query categories, especially for long path (L) and complex queries (C). On the other hand, TriAD-SG with summary-based pruning performs well for class F queries. For L, S, C class queries, TriAD-SG indeed shows some overhead due to its additional summary-graph processing. The performance dip of TriAD-SG here seems to be due to the dense nature of the WSDTS data graph and the lack of constants (besides predicates) in the SPARQL queries. MonetDB failed to finish $S1$, $F1$–$F5$, $C1$–$C3$ within a 10-minute limit (marked as “timeout”).

8. CONCLUSIONS

We presented TriAD, which combines intra-node multi-threading with asynchronous inter-node communication for the scalable processing of SPARQL queries. TriAD consistently outperforms both centralized and distributed RDF engines, which so far still largely rely on Hadoop-based joins, in which multiple join operators may indeed run in parallel but need to be synchronized at each level of the query plan before the next iteration of Hadoop-based joins is initiated. Especially our comparison to a single Map-side join in Apache’s Hadoop and Spark platforms reveals the overhead of the Map and Reduce paradigm for such a very basic query operation. For future work, we intend to further generalize our asynchronous communication principles to memory-resident DBMS architectures. We also intend to specifically investigate the behavior of our approach under different main-memory hierarchies, including recent NUMA architectures, which employ a similar form of distributed (but shared) memory model with a dedicated memory channel per CPU but slower memory access across channels.

9. REFERENCES

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. SW-Store: A vertically partitioned DBMS for Semantic Web data management. *The VLDB Journal*, 18(2), 2009.

[2] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "Bit" loaded: A scalable lightweight join query processor for RDF data. In *WWW*, 2010.

[3] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of ACM*, 51(1), 2008.

[4] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). *PVLDB*, 3(1), 2010.

[5] P. C. et al. NoSQL Databases for RDF: An Empirical Evaluation. In *ISWC*, 2013.

[6] T. M. Forum. MPI: A Message Passing Interface, 1993.

[7] S. Harris, N. Lamb, and N. Shadbolt. 4store: The design and implementation of a clustered RDF store. In *SSWS*, 2009.

[8] J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL Querying of Large RDF Graphs. *PVLDB*, 4(11), 2011.

[9] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.*, 20(1), 1998.

[10] J. Lin and C. Dyer. *Data-Intensive Text Processing with MapReduce*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers, 2010.

[11] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.

[12] T. Milo and D. Suciu. Index Structures for Path Expressions. In *ICDT’99*, volume 1540. Springer Berlin Heidelberg, 1999.

[13] T. Neumann and G. Weikum. Scalable join processing on very large RDF graphs. In *SIGMOD*, 2009.

[14] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1), 2010.

[15] T. Neumann and G. Weikum. x-RDF-3X: Fast Querying, High Update Rates, and Consistency for RDF Databases. *PVLDB*, 2010.

[16] F. Picalausa, Y. Luo, G. H. L. Fletcher, J. Hidders, and S. Vansummeren. A structural approach to indexing triples. In *ESWC*, 2012.

[17] K. Rohloff and R. E. Schantz. Clause-iteration with MapReduce to scalably query datagraphs in SHARD graph-store. In *DIDC*, 2011.

[18] S. Sakr and G. Al-Naymat. Relational Processing of RDF Queries: A Survey. *SIGMOD Rec.*, 38(4), 2010.

[19] S. Seufert, A. Anand, S. J. Bedathur, and G. Weikum. FERRARI: Flexible and efficient reachability range assignment for graph indexing. In *ICDE*, 2013.

[20] Z. Shang and J. X. Yu. Catch the Wind: Graph workload balancing on cloud. In *ICDE*, 2013.

[21] B. Shao, H. Wang, and Y. Li. Trinity: a distributed graph engine on a memory cloud. In *SIGMOD*, 2013.

[22] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold. Column-store support for RDF data management: not all swans are white. *PVLDB*, 1(2), 2008.

[23] C. A. Sugar, Gareth, and M. James. Finding the number of clusters in a data set: An information theoretic approach. *Journal of the American Statistical Association*, 98, 2003.

[24] P. Tsialiamanis, L. Sidirourgos, I. Fundulaki, V. Christophides, and P. A. Boncz. Heuristics-based query optimisation for SPARQL. In *EDBT*, 2012.

[25] J. Webber. A programmatic introduction to Neo4j. In *SPLASH*, 2012.

[26] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1), 2008.

[27] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. GraphX: a resilient distributed graph system on Spark. In *GRADES*, 2013.

[28] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu. TripleBit: a Fast and Compact System for Large Scale RDF Data. *PVLDB*, 6(7), 2013.

[29] M. Zaharia, N. M. M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. Tech Report UCB/EECS-2010-53, EECS Department, UC Berkeley, May 2010.

[30] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang. A distributed graph engine for web scale RDF data. *PVLDB*, 6(4), 2013.

[31] X. Zhang, L. Chen, Y. Tong, and M. Wang. EAGRE: Towards scalable I/O efficient SPARQL query evaluation on the cloud. In *ICDE*, 2013.

[32] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao. gStore: Answering SPARQL Queries via Subgraph Matching. *PVLDB*, 4(8), 2011.