Resource Planning for SPARQL Query Execution on Data Sharing Platforms

Stefan Hagedorn[1], Katja Hose[2], Kai-Uwe Sattler[1], and Jürgen Umbrich[3]

[1] Ilmenau University of Technology, Germany
[2] Aalborg University, Denmark
[3] Vienna University of Economy and Business, Austria

Abstract. To increase performance, data sharing platforms often make use of clusters of nodes where certain tasks can be executed in parallel. Resource planning and especially deciding how many processors should be chosen to exploit parallel processing is complex in such a setup as increasing the number of processors does not always improve runtime due to communication overhead. Instead, there is usually an optimum number of processors for which using more or fewer processors leads to less efficient runtimes. In this paper, we present a cost model based on widely used statistics (VoiD) and show how to compute the optimum number of processors that should be used to evaluate a particular SPARQL query over a particular configuration and RDF dataset. Our first experiments show the general applicability of our approach but also how shortcomings in the used statistics limit the potential of optimization.

1 Introduction

In recent years, we have witnessed the growing interest and necessity of enabling large-scale solutions for SPARQL query processing. In particular, the current trend of emerging platforms for sharing (Linked) Open Data aims at enabling convenient access to large numbers of datasets and huge amounts of data. One widely known system is the Datahub[4], a service provided by the Open Knowledge Foundation. Fujitsu published a single point of entry to the Linked Open Data Cloud[5] with the aim of hosting most datasets (as the license allows it) in the same format and in one place. Such a platform can be seen as a catalog component in a dataspace [4,18] with the main purpose to inventorize available datasets and their meta information. However, such platforms do not yet provide processing capabilities over the registered datasets and it is left to the users to provide additional services such as querying and analytics. Similar to [8], we believe that the next generation of such Linked Data catalogs will provide services to transform, integrate, query, and analyze data across all hosted datasets, considering access policies and restrictions. Those services will be able to exploit that such platforms are (i) typically deployed on a distributed (cloud) infrastructure with multiple processors and (ii) with multiple datasets available.

Advanced query functionalities will be a core service to enable users to explore and analyze the data. However, different privacy restrictions and licenses pose a big challenge on centralized and distributed query processing approaches. As such, it is very likely that those platforms keep datasets in separate partitions. While the research for

[4] http://datahub.io/
[5] http://lod4all.net/

Linked Data query processing mainly focused on single-node systems, cluster-based systems, and federated query approaches, not much attention was given to the multi-processor and multi-dataset setup. Although cluster-based systems use multiple processors, the data is considered as one big dataset and the original partitioning into multiple datasets does no longer exist – and in federated query approaches, the processors cannot be allocated that flexibly, e.g., it is not really possible to assign two processors to the same dataset to speed up processing a single query by using parallelization.

The challenges for the multi-node multi-dataset SPARQL query execution setup are to compute the optimal number of processors for a given number of datasets and queries as well as exploiting the partitioning of data. Queries that can be highly parallelized and do not require cross-dataset joins benefit from multiple allocated processors, whereas the query costs for other queries might increase with additional processors due to additional communication costs. Hence, given a data sharing platform with N available processors and M data partitions, the problem is to find the optimal processor allocation per dataset for a given query. The crucial component of this optimization problem is an appropriate cost model to estimate the query evaluation costs for a given configuration.

In this paper, we therefore investigate this problem and present a cost model that estimates query execution costs for different numbers of allocated processors and in doing so allows to find the optimum number of processors for a given query. Although we focus on the processor allocation problem, we want to emphasize that this cost model is also the foundation for workload-driven partition design. Accurate statistics are a crucial component of any cost model. Instead of proposing yet another standard for statistics, we focus on the widely used VoiD [2] statistics that many data sources already provide. Our experiments show the general applicability of our approach but also how shortcomings in the statistics limit the potential of optimization.

2 Background and Related Work

Hosting datasets and providing services on top of these datasets on a data sharing platform for many users results in a large volume of data as well as a high query load. In order to handle this load the data management layer of such a platform is typically implemented by a database cluster where the individual nodes store datasets or partitions of them. From a query processing perspective this represents a case of distributed query processing. Possible strategies range from federated to parallel query processing. In federated query processing usually a single instance of a query processor decomposes a given query into subqueries which are sent to the source nodes for evaluating. This allows to exploit multiple processing nodes and parallelism but only at the granularity of entire datasets. Examples of such processors for SPARQL are described e.g. in [16,7,1]. In contrast, parallel query processing for database clusters aims at exploiting not only the availability of multiple processing nodes but also the way the data is partitioned and placed on the nodes. This allows to implement different forms of parallelism (inter-query vs. intra-query and within a query inter-operator vs. intra-operator). However, this requires more control over the data placement. Examples of such cluster-based query systems for SPARQL are Virtuoso Cluster, Bigdata[6], and 4store [9]. Furthermore, MapReduce-based evaluation of SPARQL or more general graph-based queries can be seen as another example of cluster-based parallel processing [15,10].

[6] http://www.systap.com/bigdata.htm

Apart from appropriate partitioning techniques, query execution and optimization strategies exploiting these partitioning, a particular important task is resource planning and provisioning for queries for at least two reasons. First, even with high-speed networking interfaces data transfer between nodes is still more expensive than local memory/disk access. Thus, adding an additional processing node to a query pays off only if the amount of processable work is large enough. Second, keeping many nodes busy may block or delay other queries. Furthermore, if the data sharing platform runs on a hosted or on-demand computing infrastructure a waste of resources or over-provisioning should be avoided.

Earlier approaches on query resource allocation dealt only with the problem of allocating processors for multi-join queries [3], other works take also multi-dimensional resources (CPU, memory, etc.) into account [6]. More recent approaches like [5] go beyond predefined cost models which work only for operators known in advance by introducing a training phase using machine learning techniques.

3 Query strategies

The motivation of this work is to evaluate a multi-processor query execution strategy against alternatives that are using multiple processors and/or multiple partitions.

We assume a data sharing platform consists of several physical nodes with disks and processors to store the data(sets). Similar to any distributed platform, we assume that nodes can be used to access data and retrieve information based on basic index lookups. As such, given a query and a multi node data sharing platform with N processors and M dataset partitions, there are four scenarios how the query can be evaluated:

The *single node scenario* (1×1) classically assumes that we have one processor and one dataset. Note, this case is straightforward and we won't consider it in the following. The *multi-processor shared-disk scenario* $(N \times 1)$ uses several processors to evaluate a given query over one dataset. The *coordinated scenario* $(1 \times M)$ evaluates a given query over several datasets using one processor. Eventually, a query is evaluated with multiple processors over several datasets in the *multi-processor shared-nothing scenario* $(N \times M)$.

Query operators We assume that each node can retrieve the data for an atomic SPARQL triple query pattern and consider only simple query operators. Other operations like filters, grouping or aggregations are not in the scope for this work and are left for future efforts.

- The *scan operation* retrieves all relevant statements matching a certain triple pattern. We distinguish between a *full table scan* and an *index lookup* (e.g., based on an subject value index).
- The *union operation* combines the results of two or more scan operations for the same query pattern.
- The *join operation* combines the results of two different query patterns. We distinguish between *local* and *distributed* joins. The former joins two triple patterns on the same processor whereas the latter performs a join over two remote datasets.

Query shapes We target our analysis on the following three fundamental query shapes:

- The evaluation of a *single triple pattern*. This query also serves as a baseline since all other queries are a combination of triple patterns

– The evaluation of a two triple patterns with the join variable at the same position. Such queries are also called *star-shaped* queries if the join variable is at the subject or object position
– The evaluation of two or more triple patterns that form a chain/path, that is, two triple patterns do not share the same join variable position. Such queries are also called *path-shaped* queries

3.1 Evaluation strategies

Next, we need to decide on an evaluation strategy for a given a query and a processor-to-dataset configuration. For all these strategies, we assume the existence of a coordinator that will receive the query results from the processors. Further, we assume the availability of an sophisticated index to perform index scans and lookups over a subset of the actual dataset or over two or more datasets. The former is referred to as a dynamic split of a datasets, and the latter is referred to as a dynamic merge of datasets.

Lookup Strategy: Our evaluation strategy for a lookup query is as follows:

$1 \times M$**:** The processor evaluates the query sequentially on all M datasets by performing index scans and returns the union of all results.

$N \times 1$**:** We perform a dynamic partitioning and split the dataset into N parts. Afterwards, we have N processors that we can use to evaluate the query over N partitions in parallel. This can be considered a special case of the $N \times M$ setup where $N = M$, since the one partition is split into N dynamic ones.

$N \times M$**:** In case $N = M$, we can use one processor to execute the query over each partition in parallel and union the results afterwards. As the processor that was assigned to the largest partition will determine response time, we can decide to dynamically partition the datasets to create equal sized partitions per processor. In case $N > M$ (i.e., more processors than partitions), we will therefore split bigger datasets into smaller partitions until we can assign one processor to each partition and have it evaluate the query in parallel. In case $N < M$ (fewer processors than partitions), we merge the M datasets into bigger partitions so that eventually there is one processor per dataset – alternatively this can be assigning multiple partitions to one processor.

In all cases, after the query is executed at each processor, the results are transferred to the coordinator, i.e., a central instance that does query optimization, assigns tasks to processors, receives the (partial) results from the processors, combines (union) the received results, and outputs them to the user.

Join Strategy: For join queries the evaluation strategies are slightly more complex and the coordinator plays an even more important role. In addition, the costs of executing a join are determined based on what join algorithms the underlying system implements [14], e.g., merge joins, hash joins, etc.

$1 \times M$**:** The processor evaluates the query sequentially on all M datasets by evaluating both involved triple patterns on all the datasets and combining their results using an appropriate join implementation – as multiple datasets are involved and we cannot rely on any sorting order, this will be most likely a hash join.

$N \times 1$**:** Just as for lookup queries, we perform a dynamic partitioning and split the dataset into N parts. This will accelerate the processing of the two involved triple patterns because they can be evaluated in parallel now. However, computing the join efficiently becomes more challenging as in principle we could use pipelining in combination with a symmetric hash join [12] on one of the processors that receives partial results from the others. To abstract from available implementations, we can alternatively assume that we apply a coordinator join, i.e., a join variant that many existing systems apply and that lets the coordinator compute the join result based on the partial results received from and computed by the involved processors.

$N \times M$**:** In case $N = M$, we can use the procedure described for the $N \times 1$ case after splitting up the data into N partitions. However, there is a special case: if the partitions have been created in a way that evaluating the join over each partition in separate and union these join results produces all results, then we do not need to use a coordinator join (instead we can treat this as multiple independent 1×1 cases); we hence transfer less data between processors and can therefore be more efficient. Creating such a partitioning is usually too expensive on-the-fly, and we therefore do not consider it for the $N \times 1$ case. Hence, some queries can be evaluated very efficiently with *local joins* as an alternative to the *coordinator join* that we use for all other cases. In case $N > M$, similar to the $N \times 1$ case, the query optimizer needs to estimate if the gain of splitting up a partition outweighs the potential loss of not being able to apply a local join. This means that when splitting up a partition into equal sized parts and assigning them to different processors, it is no longer possible to apply a local join. But if the dataset is big, then the gain of parallel processing outweighs the costs of the more expensive join algorithm. In case $N < M$, the query optimizer can assign multiple datasets to one processor. Hence, some of the processors might be assigned only one partition, which corresponds to the 1×1 case while other processors are assigned multiple partitions which corresponds to the $1 \times M$ case – both can be handled as explained above.

4 Statistics and cost estimation

Applying the standard steps of query planning, the query is first parsed and normalized, and then transformed into an internal representation, e.g., a query operator tree. The optimizer then uses the query plan and available statistics to find an optimal query execution plan including join algorithms, processor allocation and how results are combined. The optimizer uses the statistics to estimate the costs, e.g., in terms of runtime, of multiple alternative query execution plans and chooses the cheapest one for execution. Finally, the chosen query execution plan is executed so that the result is available and can be output to the user or application that issued the query.

The correct estimation of the costs for a query evaluation strategy depends on the granularity of the available statistics for the data. Such statistics are typically cardinality and selectivity values/estimations for atomic query patterns. More complex statistics can contain information about correlation [13] or precomputed joins result sizes.

In this work, we assume that statistics are available in the form of VoiD descriptions [2], which are widely used to describe metadata about RDF datasets[7] and also frequently exploited as input for cardinality-based query optimizations [7]. These statistics can logically be divided into three parts: dataset statistics, property partition, and class partition. The dataset statistics describe standard statistics about the complete dataset similar to statistics in relational database systems, e.g., the total number of triples (void:triples, c_t), the total number of distinct subjects (void:distinctSubjects, c_s) and objects (void:distinctObjects, c_o). The property partition contains equivalents of these values for each property individually that occurs in the dataset ($cp_{p,t}, cp_{p,s}, cp_{p,o}$), e.g., $cp_{p,t}$ denotes the number of triples in the dataset with property p. And the class partition

[7] http://www.w3.org/TR/void/

states for each class how many entities of the class occur in the dataset (void:entities $cc_{c,e}$). The VoiD standard defines more pieces of information but we restrict our discussion to the ones that are relevant for our cost model.

Based on the information that we can obtain from the VoiD statistics, we can estimate cardinalities of triple patterns that we encounter as parts of SPARQL queries. Table 1 lists the formulas for all eight possible permutations of variables in triple patterns in general. Whenever a predicate is given in the triple pattern, we can make use of property partition statistics, which allows for a better estimation than relying on the dataset statistics only. The table also lists estimations for the special case that the predicate corresponds to `rdf:type`, for which we can also make use of the statistics in the class partition.

triple pattern	result cardinality ($card$)
?s ?p ?o	c_t
subjA ?p ?o	$\frac{c_t}{c_s}$
?s predA ?o	$cp_{predA,t}$
?s ?p objA	$\frac{c_t}{c_o}$
subjA predA ?o	$\frac{cp_{predA,t}}{cp_{predA,s}}$
subjA ?p objA	$\frac{c_t}{c_s \cdot c_o}$
?s predA objA	$\frac{cp_{predA,t}}{cp_{predA,o}}$

triple pattern	result cardinality ($card$)
subjA predA objA	$\frac{cp_{predA,t}}{cp_{predA,s} \cdot cp_{predA,o}}$
?s rdf:type ?o	$cp_{rdf:type,t}$
subjA rdf:type ?o	$\frac{cp_{rdf:type,t}}{cp_{rdf:type,s}}$
?s rdf:type objA	$cc_{objA,e}$
subjA rdf:type objA	$\frac{cp_{rdf:type,t}}{cp_{rdf:type,s} \cdot cp_{rdf:type,o}}$, or 0 if no *classPartition* for *objA* in statistics

Table 1. Cardinality estimates for triple patterns

A join is an operation that combines the results of multiple triple patterns. The join can be highly selective, in which case the cardinality of the join result will be smaller than the sum over the cardinalities of the input triple pattern results. On the other hand, the join can also have low selectivity, in which case the result cardinality can be greater than the sum of the input cardinalities. Cardinality estimation over joins without additional join statistics is therefore very difficult and its precision very much depends on the characteristics of the data it is applied on. Table 2 sketches the formulas we use to estimate the cardinality of joins based on the cardinalities of the triple patterns that provide the input to the join. These formulas are based on estimates that have been developed in the context of relational database systems [17].

4.1 Cost model

Based on the estimation of cardinalities that we have discussed above and the description of the evaluation strategies in Section 3.1, we can now define a cost model that estimates the costs of executing a query. Again, we will make use of standard techniques known from distributed database systems and apply them to the SPARQL use case [7]. To compute the costs of executing a query, we use the following parameters: t_{CPU} (time for a CPU cycle), t_{IO} (time for a IO operation), t_t (time for transferring one triple), and t_m (time for transferring a message, i.e., message overhead). Each of these parameters is configuration-specific and needs to be measured on the systems that we want to estimate costs for. For simplicity, our formulas assume that each of the involved processors and communication connections have the same characteristics – in cloud setups this is a valid assumption. The formulas are still applicable in the broader use case of distributed and federated architectures but should be extended to consider communication delays and processor characteristics for each involved processor/node

type	**pattern**	**cardinality** $card(join, partition)$
subject – subject	?s predA ?o . ?s predB ?o2	$\frac{card(pat_1) \cdot card(pat_2)}{\max(cp_{predA,s}, cp_{predB,s})}$
	?s predA objA . ?s predB objB	$\frac{card(pat_1) \cdot card(pat_2)}{\max(cp_{predA,s} \cdot cp_{predA,o}, cp_{predB,s} \cdot cp_{predB,o})}$
	?s predA ?o . ?s predB objB	$\frac{card(pat_1) \cdot card(pat_2)}{\max(cp_{predA,s}, cp_{predB,s} \cdot cp_{predB,o})}$
	?s predA objA . ?s predB ?o2	$\frac{card(pat_1) \cdot card(pat_2)}{\max(cp_{predA,s} \cdot cp_{predA,o}, cp_{predB,s})}$
	?s rdf:type ?o . ?s predB ?o2	$\frac{card(pat_1) \cdot card(pat_2)}{\max(cp_{rdf:type,s}, cp_{predB,s})}$
	?s predA ?o . ?s rdf:type ?o2	$\frac{card(pat_1) \cdot card(pat_2)}{\max(cp_{predA,s}, cp_{rdf:type,s})}$
	?s rdf:type ?o . ?s rdf:type ?o2	$\frac{card(pat_1) \cdot card(pat_2)}{cp_{rdf:type,s}}$
	?s rdf:type objA . ?s predB objB	$\frac{card(pat_1) \cdot card(pat_2)}{\max(cc_{objA,e}, cp_{predB,s} \cdot cp_{predB,o})}$
	?s predA objA . ?s rdf:type objB	$\frac{card(pat_1) \cdot card(pat_2)}{\max(cp_{predA,s} \cdot cp_{predA,o}, cc_{objB,e})}$
	?s rdf:type objA . ?s rdf:type objB	$\frac{card(pat_1) \cdot card(pat_2)}{\max(cc_{objA,e}, cc_{objB,e})}$
	?s rdf:type ?o . ?s predB objB	$\frac{card(pat_1) \cdot card(pat_2)}{\max(cp_{rdf:type,s}, cp_{predB,s} \cdot cp_{predB,o})}$
	?s predA ?o . ?s rdf:type objB	$\frac{card(pat_1) \cdot card(pat_2)}{\max(cp_{predA,s}, cc_{objB,e})}$
	?s rdf:type ?o . ?s rdf:type objB	$\frac{card(pat_1) \cdot card(pat_2)}{\max(cp_{rdf:type,s}, cc_{objB,e})}$
	?s rdf:type objA . ?s predB ?o2	$\frac{card(pat_1) \cdot card(pat_2)}{\max(cc_{objA,e}, cp_{predB,s})}$
	?s predA objA . ?s rdf:type ?o2	$\frac{card(pat_1) \cdot card(pat_2)}{\max(cp_{predA,s}, cp_{rdf:type,s})}$
	?s rdf:type objA . ?s rdf:type ?o2	$\frac{card(pat_1) \cdot card(pat_2)}{\max(cc_{objA,e}, cp_{rdf:type,s})}$
object – subject	?s predA ?o . ?o predB ?o2	$\frac{card(pat_1) \cdot card(pat_2)}{\max(cp_{predA,o}, cp_{predB,s})}$
	...	...
subject – object	?s predA ?o . ?s2 predB ?s	$\frac{card(pat_1) \cdot card(pat_2)}{\max(cp_{predA,s}, cp_{predB,o})}$
	...	...
object – object	?s predA ?o . ?s2 predB ?o	$\frac{card(pat_1) \cdot card(pat_2)}{\max(cp_{predA,o}, cp_{predB,o})}$
	...	...

Table 2. Cardinality estimates for specific join patterns, with pat_1 and pat_2 representing triple patterns whose results serve as input to the join

individually. Whereas t_{CPU} and t_{IO} determine the processing time (T_P), t_t and t_m describe the transfer time (T_T) which is the most important cost factor in multi-processor environments. In a multi-processor environment, we are mainly interested in response time, i.e., the time from issuing a query until the results are available, and not in processing time, i.e., the sum of all times that the processors altogether spend on evaluating the query. Hence, in the following we will focus on response time T_R, which considers processing time and transfer time and is determined by the slowest processor in a multi-processor environment, i.e., the maximum over the processing and transfer times of all involved processors determines response time.

In the following discussion, we focus on estimating the costs for join queries. The formulas can be simplified to the simpler case of a lookup for a single triple pattern and extended to cover the case of multiple joins. It is a well-known problem, however, that the accuracy of cardinality estimates degrades quickly with the number of joins in a query [11]. For ease of representation, we will refer to a dataset as partition ($partition$).

Single processor We assume that the processor has to read each tuple in the partition. For the $1 \times M$ strategy (1 processor and M partitions), we further assume that the processor evaluates the query sequentially on the M partitions and is instructed to do so by the coordinator. The special case of $M = 1$ corresponds to the 1×1 strategy. As efficient in-memory execution strategies are usually available and execution costs are typically dominated by I/O and communication costs, query optimizers in parallel/distributed database systems often neglect CPU costs. If we also assume that an index is available to access the triples matching triple pattern p_1 and triple pattern p_2 that serve as input to the join, we can use an index lookup and obtain for the processing cost:

$$T_P^{1\times M} = \sum_{i=1}^{M} (card(p_1, partition_i) + card(p_2, partition_i)) \cdot t_{IO}$$

As we have only one processor that accesses all the data, we can assume that the processor can also evaluate the join locally. Hence, we can estimate transfer time as:

$$T_T^{1\times M} = 2 \cdot t_m + \sum_{i=1}^{M} card(join, partition) \cdot t_t$$

The total response time corresponds to

$$T_R^{1\times M} = T_P^{1\times M} + T_T^{1\times M} \tag{1}$$

Multiple processors Estimations for the strategies with multiple processors need to consider the different (distributed) join implementations and distinguish how the optimizer makes use of the N available processors, i.e., whether all of them are used to read the partitions or if some of them are used to accelerate join processing by applying pipelining. As mentioned earlier, we abstract from available implementations (as each of them would entail a slightly different cost formula) and instead assume coordinator join or local joins.

$\boldsymbol{N \times 1}$ For the $N \times 1$ strategy (N processors and 1 partition), we additionally need to consider the costs of dynamically partitioning a partition. For simplicity, we assume that we can efficiently split up a big partition into equal-sized partitions ($partition_i$) in constant time t_{split}.

Based on these assumptions and the fact that the slowest processor determines the overall response time, we would need to compute response time and therefore also processing and transfer time for each processor independently.

However, in contrast to the simpler cases, we do not have statistics for the partitions created by the split. Hence, we use the available statistics for the original dataset ($partition$) and the uniformity assumption. We obtain:

$$T_P^{N_i, N\times 1} = \frac{card(p_1, partition) + card(p_2, partition)}{N} \cdot t_{IO} = T_P^{N\times 1}$$

Using multiple processors means that the coordinator needs to send queries to the processors and after their execution, the partial results matching both triple patterns needs to be sent to the coordinator, which will then compute the join (coordinator join). Note that we need to collect all partial results of all triple patterns to compute the join correctly as some answers might rely on the combination of data from different partitions. Hence, as transfer time for processor N_i, we obtain:

$$T_T^{N_i, N\times 1} = 2 \cdot t_m + \frac{card(p_1, partition) + card(p_2, partition)}{N} \cdot t_t = T_T^{N\times 1}$$

As response time is determined by the slowest processor, we need to compute:

$$T_R^{N_i, N\times 1} = t_{split} + \max_{i=1..N} (T_P^{N_i, N\times 1} + T_T^{N_i, N\times 1})$$

Because of the uniformity assumption and the assumption that the costs for the efficient split itself can be neglected, we obtain:

$$T_R^{N\times 1} = T_P^{N\times 1} + T_T^{N\times 1} \tag{2}$$

$\boldsymbol{N \times M}$ For the $N \times M$ case, we can basically reuse the formulas for the above mentioned cases. We will also make the very same simplifying assumptions. However, we need to distinguish between several special cases. Let us begin with the case where $N = M$ and each processor N_i is assigned exactly one of the partitions M_i. In that case the processors compute the answers to the triple patterns locally and send the partial results to the coordinator who computes the final join result (coordinator join) – again this guarantees that the join result is complete. This special case corresponds to running multiple 1×1 cases in parallel, which is similar to the $N \times 1$ case (Equation 2) but as we have statistics for each partition in separate, we can make better estimations for $T_{R_C}^{N=M}$ (C representing the coordinator join). To keep the notation simple, we assume that $partition_i$ is assigned to processor i and obtain:

$$\begin{aligned} T_{R_C}^{N_i,N=M} = \max_{i=1..N}(& \ (card(p_1, partition_i) + card(p_2, partition_i)) \cdot t_{IO} + \\ & 2 \cdot t_m + (card(p_1, partition_i) + card(p_2, partition_i)) \cdot t_t) \end{aligned} \quad (3)$$

As discussed in Section 3.1, it is sometimes possible to decide, based on information about the native partitioning of the data (input data set characteristics), that a *local join* over each partition in separate leads to the correct and complete result. For example, if we are looking for persons that were born and that died in the same city and the data was originally partitioned based on cities, then combinations with other partitions cannot lead to additional join results.

So, when a local join is possible, we can have each processor compute the join directly over the partition assigned to it – this corresponds to multiple parallel 1×1 cases with local joins. Hence, processing time $T_{P_L}^{N_i,N=M}$ (L stands for local join) is estimated as for the coordinator join above. Transfer time, however, is based on the join cardinality for the local join and not on the cardinality of the two triple patterns as it is the case for the coordinator join. Hence, for each processor we obtain:

$$\begin{aligned} T_{P_L}^{N_i,N=M} &= T_{P}^{N_i,N=M} = (card(p_1, partition_i) + card(p_2, partition_i)) \cdot t_{IO} \\ T_{T_L}^{N_i,N=M} &= 2 \cdot t_m + card(join, partition_i) \cdot t_t \end{aligned}$$

Processing time in consideration of parallel execution over multiple processors with each processor being assigned exactly one partition for the local join case is then estimated as:

$$\begin{aligned} T_{R_L}^{N=M} = \max\nolimits_{i=1..N}(& \ (card(p_1, partition_i) + card(p_2, partition_i)) \cdot t_{IO} + \\ & 2 \cdot t_m + card(join, partition_i) \cdot t_t) \end{aligned}$$

All other cases of $N \times M$, such as $N < M$ and $N > M$, can be derived based on these considerations as well. For each query execution plan that we want to estimate response time for, we first need to identify if for the query local joins are possible in general based on the native partitioning of the data. Then, if multiple processors have been assigned to the same partition, we can make use of the estimation formulas in Equation 2 – note that even if a local join would be possible on a partition in general, this is no longer possible if multiple processors are assigned to the same partition.

If one or multiple partitions have been assigned to the same processor and a local join was possible in the native partitioning of the assigned partitions, then we can use the estimation formulas of Equation 1. If the local join was not possible, then we have to compute a coordinator join instead.

In general, a particular query execution plan in the $N \times M$ case corresponds to a mixed case. For instance, assuming that a local join is possible in the native partitioning and the optimizer assigned 2 processors to the first partition and 1 processor to the second partition, then the 2 processors will have to rely on a coordinator join whereas the third processor can do the local join directly. As this is a simple combination of the above equations, we omit the respective formula in this paper.

5 Evaluation

For our experiments, we used a subset of DBPedia (version 3.8)[8]. We focused on information about countries, cities, and persons associated with these cities and grouped them by their continents to obtain a partitioning. The resulting dataset consists of 90344 persons, 6546 cities, 102 countries, and a total of 232598 statements.

We conducted experiments with 3, 5, and 7 partitions. We used SPARQL `CONSTRUCT` query to group information about countries, cities, and persons into ten groups representing parts of the world such as e.g., USA, Europe, east Asian countries, or Mexico. For the 3 partitions scenario, we grouped the 10 datasets into three parts of roughly equal size (e.g. , one partition with United States, one with Europe, east Asian countries, middle eastern countries, Mexico, and countries in South America and one partition with Africa, Oceania, south Asian countries, and northern American countries). Respectively, we performed similar distributions for the 5 and 7 partitions case. For our experiments we use 10 Fuseki[9] instances with Jena TDB, each running in their own virtual machine in a fully virtualized environment. In each partitioning scenario, we executed tests that used one up to ten Fuseki processors and distributed the partitions to the active processors. If split or merge operations were necessary, they were applied manually so that the resulting datasets are of approximately equal size.

Queries Our test set consists of 21 queries. The first 9 queries are simple triple pattern lookups with different selectivity. The rest of the queries contain one join, i.e., they have two triple patterns. There are 7 queries with a star join (queries 10 – 16) and 5 with a path join (queries 17 – 21). With our initial partitioning, for three of these queries the join has to be performed on the coordinator.

For each possible allocation of processors to partitions ($1 \times N$ ... $10 \times N$, with $N = 3,\ 5,\ or\ 7$) the queries were repeated several times to get reliable results.

The query coordinator is written in Java and uses HTTP GET requests to send the queries to the processors. The implementation follows the requirements introduced in section 3.1. The input values for our cost model were determined by practical experiments. The values are as follows (in *ms*): $t_m = 1.617$, $t_t = 0.004523$, and $t_{IO} = 0.374$

5.1 Results

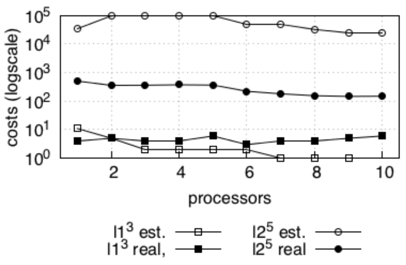
In the first experiments we evaluated the total execution time for a given query compared to the estimated cost calculated with our model.

Figure 1(a) shows the results for two lookup queries: lookup query 1 on three ($l1^3$) and lookup query 2 on five ($l2^5$) partitions. In this case we can observe that for query $l1^3$ our cost model works well. Whereas for query $l2^5$ the real execution times are far less than our model estimated.

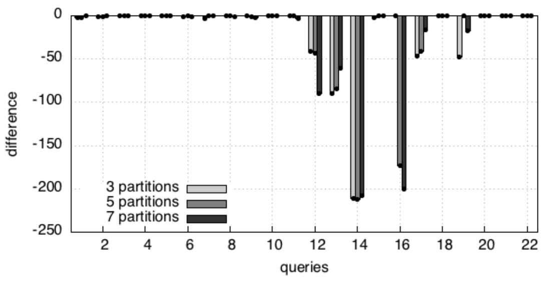
In Figure 1(c) we show the results for two star shape queries $s1^5$ and $s2^3$ and in Figure 1(d) the results for two path shape queries $p1^7$ and $p2^5$. In both cases our model does not work very well. However, for query $p2^5$ the values for both estimated and

[8] `http://wiki.dbpedia.org/Downloads`

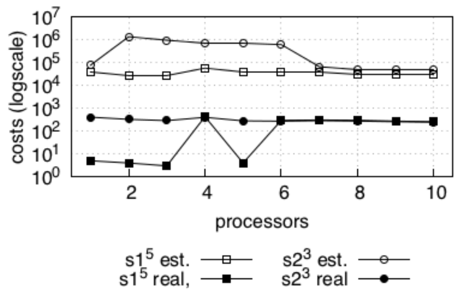
[9] `http://jena.apache.org/documentation/serving_data/`



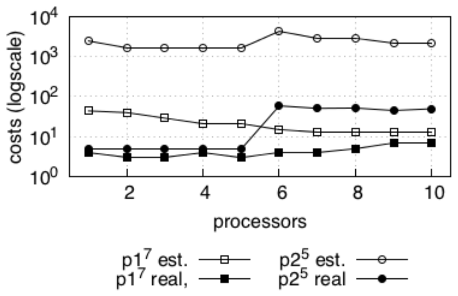
(a) Lookup queries



(b) Cost difference between real and estimated optimal processor allocation



(c) Star queries



(d) Path queries

Fig. 1. Illustration of estimated vs. real costs

real costs show the same trend. Also, one can see the switch from the local join to coordinator join. Up to five processors, a local join was used. When six processors are involved, at least one partition has to be split and we have to switch to a coordinator join for these partitions. This results in higher communication costs.

In a second set of experiments we evaluated our cost model and try to predict the optimal number of processors for a given query using the available VoiD statistics of the partitions. We computed the set of optimal processor allocation based on the estimated costs and real measured costs. Cost values were rounded up to milliseconds.

We compared the total costs for the real against the estimated optimal processor allocation for our three partitioning scenarios (cf. Figure 1(b)). We can see that especially for lookup queries (1 – 9) our cost estimation for the optimal processor allocation is almost identical to the real costs. However, for most join queries the estimated optimal costs differ from the real costs significantly. This happens because our estimation is based upon the VoiD statistics, which do not contain detailed information about certain predicates, but rather overall statistics for distinct objects, etc. This information alone is not enough to estimate join cardinalities for a given pattern.

6 Conclusion

Open Data hosting and sharing platforms tend to offer more and more services in addition to basic upload/download and catalog functionality. Typically, this requires to exploit cluster database technologies for using multiple processing and storage nodes. In this context, an important problem of query planning and execution is the allocation and provisioning of resources such as CPUs, memory, disk space etc.

In this paper, we have addressed this problem by presenting a cost model taking partitioning of datasets as well as the usage of intra-query parallelism into account. Using this cost model we have evaluated several fundamental SPARQL query execution strategies. Overall, our obtained results show that this cost model helps to determine the optimal number of processors while using only standard VoiD statistics. While our first experiments show the general applicability of our approach, we can clearly observe that for cross dataset joins the VoiD statistics are not fine grained enough to accurately predict the resulting join cardinalities. In addition, our experiments verified the need for dynamic aggregation or partitioning of statistics in the cases of dynamic splits or merges. As such, in future work we will primarily focus on using more detailed statistics as input for our cost model (e.g. characteristic sets, QTree). In addition, we will start to look into extending our cost model for other SPARQL operators such as FILTER and OPTIONAL.

References

1. M. Acosta, M.-E. Vidal, T. Lampo, J. Castillo, and E. Ruckhaus. ANAPSID: An Adaptive Query Processing Engine for SPARQL Endpoints. In *ISWC*, pages 18–34, 2011.
2. K. Alexander, R. Cyganiak, M. Hausenblas, and J. Zhao. Describing linked datasets. In *LDOW*, 2009.
3. M.-S. Chen, P. Yu, and K.-L. Wu. Scheduling and processor allocation for parallel execution of multijoin queries. In *ICDE*, pages 58–67, 1992.
4. M. J. Franklin, A. Y. Halevy, and D. Maier. From databases to dataspaces: a new abstraction for information management. *SIGMOD Record*, 34(4):27–33, 2005.
5. A. Ganapathi, H. A. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. I. Jordan, and D. A. Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *ICDE*, pages 592–603, 2009.
6. M. N. Garofalakis and Y. E. Ioannidis. Parallel query scheduling and optimization with time- and space-shared resources. In *VLDB*, pages 296–305, 1997.
7. O. Görlitz and S. Staab. SPLENDID: SPARQL Endpoint Federation Exploiting VOID Descriptions. In *COLD*, 2011.
8. S. Hagedorn and K.-U. Sattler. LODHub - A Platform for Sharing and Integrated Processing of Linked Open Data. In *DESWeb*, 2014.
9. S. Harris, N. Lamb, and N. Shadbolt. 4store: The design and implementation of a clustered RDF store. In *SSWS*, pages 94–109, 2009.
10. J. Huang, D. J. Abadi, and K. Ren. Scalable sparql querying of large rdf graphs. *PVLDB*, 4(11):1123–1134, 2011.
11. Y. E. Ioannidis and S. Christodoulakis. On the Propagation of Errors in the Size of Join Results. In *SIGMOD*, pages 268–277, 1991.
12. G. Ladwig and T. Tran. Linked data query processing strategies. In *ISWC*, pages 453–469, 2010.
13. T. Neumann and G. Moerkotte. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *ICDE*, pages 984–994, 2011.
14. T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *PVLDB*, 1(1):647–659, 2008.
15. A. Schätzle, M. Przyjaciel-Zablocki, and G. Lausen. PigSPARQL: mapping SPARQL to Pig Latin. In *SWIM*, page 4, 2011.
16. A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. FedX: Optimization Techniques for Federated Query Processing on Linked Data. In *ISWC*, pages 601–616, 2011.
17. A. N. Swami and K. B. Schiefer. On the Estimation of Join Result Sizes. In *EDBT*, pages 287–300, 1994.
18. J. Umbrich, M. Karnstedt, J. X. Parreira, A. Polleres, and M. Hauswirth. Linked Data and Live Querying for Enabling Support Platforms for Web Dataspaces. In *DESWeb*, pages 23–28, 2012.