Scalable Join Processing on Very Large RDF Graphs

Thomas Neumann
Max-Planck-Institut fü Informatik
Saarbrücken, Germany
neumann@mpi-inf.mpg.de

Gerhard Weikum
Max-Planck-Institut fü Informatik
Saarbrücken, Germany
weikum@mpi-inf.mpg.de

ABSTRACT

With the proliferation of the RDF data format, engines for RDF query processing are faced with very large graphs that contain hundreds of millions of RDF triples. This paper addresses the resulting scalability problems. Recent prior work along these lines has focused on indexing and other physical-design issues. The current paper focuses on join processing, as the fine-grained and schema-relaxed use of RDF often entails star- and chain-shaped join queries with many input streams from index scans.

We present two contributions for scalable join processing. First, we develop very light-weight methods for sideways information passing between separate joins at query run-time, to provide highly effective filters on the input streams of joins. Second, we improve previously proposed algorithms for join-order optimization by more accurate selectivity estimations for very large RDF graphs. Experimental studies with several RDF datasets, including the UniProt collection, demonstrate the performance gains of our approach, outperforming the previously fastest systems by more than an order of magnitude.

1. INTRODUCTION

1.1 Background

The RDF data format, originally proposed for the Semantic Web, has gained popularity for building large-scale data collections. In particular, biologists and other life scientists like RDF because of its ease of use and flexibility [5, 37]. They can easily collect data without a schema-first database design phase – a paradigm known as "pay-as-you-go" dataspaces [16]. Moreover, it is easy to add metadata, annotations, and lineage information, and represent all of this uniformly together with the primary data. Querying provides certain degrees of schema agnosticity that relational systems do not offer: property names, the RDF counterpart of attribute names, can be left unspecified in queries. A second application area where RDF develops major momentum is knowledge-management communities such as DBpedia [9] or freebase [13] that aim to build large collections of facts about entities and their relations with RDF-based representations. Information extraction from Wikipedia and other Web sources (e.g., [2, 35, 40]) and social-tagging communities (e.g., [6, 12]) have a similar flavor.

RDF data collections can be seen as entity-relationship graphs with edges corresponding to *(subject, property, object) (SPO) triples* (with property often also referred to as predicate). For example, the fact that the French novelist (and Nobel prize winner) Jean-Marie Le Clezio has written the book "Desert" would be represented by the following four triples:

```
(Id1, hasName, Jean-Marie Le Clezio),
(Id1, hasNationality, French),
(Id1, hasWritten, Id2), (Id2, hasTitle Desert).
```

Querying such RDF data amounts to evaluating graph patterns, expressed in the SPARQL query language. For example, the query with the following three *triple patterns*

```
?x hasName ?y .  ?x hasNationality French .
?x hasProfession novelist
```

finds the names of all French novelists (where the dot denotes a conjunction). This example shows that the fine-grained modeling by triples and the pattern-oriented querying often entails manyway joins, at least conceptually. The example can be seen as a star (self-) join over a universal triples table. Queries that connect different entities via relations (e.g., authors and their books and the characters in these books) involve potentially long chain joins. Although it is conceivable to cluster RDF triples and map clusters onto relational-style tables with multiple properties (e.g., all persons together with their names, nationalities, professions), such approaches face major complexity regarding an appropriate physical design. Recent work [1, 25, 31, 38] has shown that column-store representations or using a single triples table for storage drastically simplify the physical-design problem and often provide superior performance.

1.2 Problem

Large join queries are an inherent characteristic of searching RDF data, posing a performance challenge already for medium-sized datasets with tens of millions of triples [1]. We now see RDF datasets that contain close to a *billion triples*. For example, the Uniprot collection [37] is a huge biological dataset about proteins and combines rich annotations from various other collections, with a total of 845 million triples. At this year's ISWC conference, the Semantic-Web community has addressed a Billion Triples Challenge [29], using a heterogeneous collection that includes DBpedia and

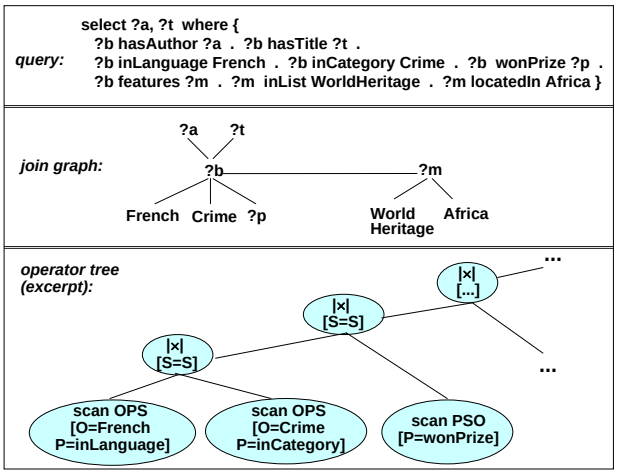

Figure 1: Query, Join Graph and Operator Tree

Freebase, with a total of more than 1.1 billion triples. The proposed solutions (including the official winners) limited themselves to simple navigational access and avoided complex queries. But the need for more advanced declarative querying is emerging and poses a tremendous performance challenge for this kind of huge RDF graphs.

Recent research on RDF engines, most notably [1, 25, 31, 38], goes a long way towards the above scalability goals, but cannot handle hundreds of millions of triples with interactive response times for complex queries. The probably fastest system, according to the published performance numbers, is RDF-3X [25]. It is based on a single triples table, uses aggressive indexing of all SPO permutations with high compression, employs fast merge joins based on these indexes, and has an advanced optimizer for determining the best join ordering. Other systems like the ones based on C-Store [1] and MonetDB [31] have similar capabilities regarding their query processors (also making extensive use of merge joins with very good L2 cache hit rate). Nevertheless, queries that have non-selective subqueries still require *scanning large fractions of indexes*, even if the final result of the entire query is very small. As the data volume and index sizes grow towards the billion-triples mark, these index scans become a bottleneck, inevitably degrading response times to tens of seconds or more and also hurting throughput on a multi-user server. In addition to the query execution costs, a second problem is that *selectivity estimation on RDF graphs* becomes less accurate with increasing graph size (and heterogeneity), which in turn leads to suboptimal decisions by the join-order optimization.

As a concrete example, consider the query shown in Figure 1, as a SPARQL query, a join graph, and an operator tree for a reasonable execution plan. Intuitively, the query aims to find authors and titles of French crime novels that have won prizes and feature African landmarks that are in the Unesco World Heritage list. Suppose the optimizer decides to perform the star join on books first, before joining this intermediate result with the landmark part of the join graph. Further suppose that the star join first joins the results on the filters for the inLanguage and inCategory predicates, by scanning the OPS index that provides triples (of short identifiers) in object-property-subject order. Subsequently, the next join is performed with a PSO index scan for the wonPrize predicate. Both joins are merge joins, and they are pipelinedtogether with the index scans. If neither inLanguage=French nor inCategory=Crime are selective predicates, then the first join inevitably has to scan very large parts of the PSO index and would produce a large intermediate output that is streamed into the second join.

1.3 Solution

We address the described problem of large index scans in a twofold manner: a run-time technique for accelerating query executions, and a technique for improving the join-order optimization by better selectivity estimates. First, we introduce a novel, very light-weight, RDF-specific technique for *sideways information passing* across different joins and index scans within pipelined execution plans. We enhance operators to build filters on subject, property, or object identifiers, and pass these to other joins and scans that may reside in a very different part of the operator tree but need to process comparable identifiers. In the example operator tree of Figure 1, assuming that the wonPrize predicate is very selective, the subject identifiers that are produced by the PSO scan could be passed to the two OPS scans, enabling them to skip large fractions of the OPS index that would not find a partner for the second join anyway (although they would yield intermediate output for the first join). This approach is related to semi-join programs [4, 33] and magic-set rewritings [24, 30], but in contrast to these classical methods, our solution is a run-time technique that works for all possible execution plans (without any complications for the query optimizer), is suitable for operator pipelines, has extremely low overhead, and yields major benefits for operators within the same pipeline. Second, we enhance the *selectivity estimator* of the query optimizer by using very fast index lookups on specifically designed aggregation indexes, rather than relying on the usual kinds of coarse-grained histograms. This provides much more accurate estimates at compile-time, at a fairly small cost that is easily amortized by providing better directives for the join-order optimization.

We implemented all our methods in the open-source system RDF-3X [28]. Our experiments show that we can speed up RDF-3X by more than an order of magnitude on datasets that have close to a billion triples. We compare our approach to the original RDF-3X and also to the alternative architectures proposed by [26] and [1] based on MonetDB [23] and PostgreSQL [27] open-source systems, as leading representatives of column-store and row-store engines. Overall, the paper makes the following novel contributions:

- a run-time technique, with extremely low overhead, for sideways information passing across joins and scans in large operator trees with pipelined operators;
- a highly effective technique for improving selectivity estimation based on specifically designed aggregation indexes;
- an integrated system-level solution that puts all these techniques together for scalable RDF querying;
- an extensive experimental comparison of our approach against three existing systems, using the very large RDF graphs of UniProt [37] and the Billion Triples Challenge [29].

The rest of the paper has the following structure. Section 2 discusses related work. Section 3 presents the new method for sideways information passing. Section 4 presents the improved selectivity estimator. Finally, Section 5 describes our extensive experimental evaluation.

2. RELATED WORK

RDF Engines. Work on RDF engines goes back to the early days of Semantic-Web research. Sesame [7, 26] and Jena [39, 19] are among the most mature and popular systems, but were not designed for scalability. Experimental studies (e.g., [1, 25]) show severe shortcomings for large datasets and advanced queries. The more recent Yars2 engine [17] appears to be more scalable, but also seems to be limited to navigational access and simple queries. Commercial mainstream systems support RDF as well (e.g. [8]), but various studies indicate that good performance is achievable only with sophisticated tuning [1, 25]. Finally, several academic database researchers have embarked on high-performance RDF engines, which led to a number of interesting architectures in the last two years. [1] has presented a column-store-oriented architecture; its implementation based on C-Store showed tremendous performance gains over other kinds of RDF engines. [31] has extended this architectural rethinking by mapping RDF data and queries onto both row-store (PostgreSQL) and column-store (MonetDB) architectures with systematic studies of their respective pros and cons. Most recently, [25] developed the RDF-3X open-source system, and experimentally showed that its triples-table storage, together with aggressive indexing and smart query optimization, can outperform all other approaches by a large margin.

Storage. Traditionally, the physical design of RDF systems has been based on clustered property tables (e.g., [8, 7, 39]). Such approaches group RDF triples by their properties – the counterpart of relational attribute names – and combine triples that share many property names into the same storage-level table. This physical organization resembles conventional row-store of relational database systems, but it entails difficult tuning issues. To avoid these complications, a "giant triples table" approach with a generic subject(S)-property(P)-object(O) schema has been used in several systems. In contrast to the clustered property tables, this triples-table design leads to many self-joins, posing difficulties to query processing and query optimization. On the other hand, its simplicity leads to streamlined execution plans that benefit from fast sequential access, high data compression, compact code, and better memory locality, resulting in superior overall performance. Similar arguments hold for approaches that map triples onto a column-store with binary relations, effectively declustering the S, P, and O components across different tables.

Indexing. All these physical design alternatives are complemented by additional indexes on combinations of S, P, and O. Some systems index judiciously chosen combinations and permutations of S, P, and O; others leave the choice to a human tuning expert. [38] and [25] have proposed very aggressive indexing, with indexes on all possible permutations of SPO triples. By their high compression rates, these exhaustive indexes have surprisingly small space consumption. Even more aggressively, [36] has proposed indexing entire paths of the RDF graph and their SPO labels. However, as indexing all paths is infeasible, this approach entails difficult optimization problems of identifying the most beneficial paths. All well-engineered systems store and index short, fixed-length identifiers rather than string literals of S, P, and O values; the mapping between literals and identifiers is kept in separate dictionary tables.

Selectivity Estimation. Regardless of the physical organization, advanced queries over RDF data require selectivity estimators for selections (filters) and joins to predict the costs of different execution plans. The simplest approach, which is effective for selections, is to build histograms on the combined values of SPO triples [34]. For joins, histograms or other forms of statistical synopses could be built over the binary-join triple pairs, but the total size of such a synopsis is usually prohibitive. Another interesting approach is to precompute frequent paths (i.e., frequently occurring sequences of S, P, O labels) in the RDF data graph, and keep statistics for a small number of frequent/beneficial paths [21, 25]. This can be done using graph-mining techniques, and has been shown to be useful for selectivity estimation of star-joins and join chains.

Join Ordering. As RDF querying typically entails many-way joins (either on the triples table or the binary tables of a column-store or across different clustered property tables), determining the best (or a near-optimal) join ordering is the most crucial issue in query optimization. [25] has shown that the join order in execution plans for SPARQL queries can have a dramatic performance impact. Solutions can leverage the rich work on relational query optimization. Most state-of-the-art methods employ dynamic programming for enumerating plans in an appropriate order. These methods come in both bottom-up [22] and top-down [10] manner, starting with the simplest subqueries or the entire query, respectively. Physical properties, like the collation order of intermediate results (so-called "interesting orders" [15]), can be considered in dynamic programming, too. For certain types of join graphs - cliques and also stars, the join-ordering problem is known or conjectured to be NP-hard. Thus, exact optimization - minimizing the total cost of the execution plan according to the underlying cost model - is feasible only for up to 10-20 joins, within a few seconds. In many settings, optimization time is part of the user-perceived overall response time, because queries are generated by data-exploration tools at run-time on behalf of interactive users. A major alternative to dynamic programming is randomized search, for example, using simulated annealing or genetic programming [14]. Often, this cheaper alternative produces very good execution plans, but it bears the risk of occasionally returning very poor plans.

Sideways Information Passing. No matter how good the optimizer-generated plan is, joins remain a costly operation. When join inputs are ordered by join-attribute values (e.g., because a previous join or index scan preserves this interesting order), merge joins are usually the fastest algorithm; otherwise hash joins may be preferable. There are various ways of reducing the cost of a single join within a complex operator tree by considering non-local information (i.e., information that does not come from the two operands of the join). These methods are broadly referred to as sideways information passing (SIP) [32, 18]: they pass information across operators in a way that cuts through the tree structure of the execution plan; hence the term "sideways".

SIP strategies come in two flavors: *compile-time* and *run-time* methods. Compile-time methods include semi-join reducers [4, 20] and magic-set transformations [24, 30]. Semi-joins send join-attribute values, i.e., projections or Bloom-filter-based approximations, to other operands in the tree, where these value lists serve as run-time filters. By cascading this principles, entire semi-join programs can be built.

Magic-set rewritings adapt and extend this paradigm across entire query blocks, including nested SQL blocks and queries over views. All of these are optimizations that need to be pre-planned at compile-time in a cost-conscious manner [33, 30]; once a semi-join is built in the execution plan, it will be executed at run-time even if turns out to be non-selective. In contrast, adaptive run-time methods aim to postpone such decisions until the affected operators are actually executing. Adaptive query processing [3, 11] has mostly focused on run-time strategies for join re-ordering on a per-tuple basis and other adjustments of the execution plan's structure.

The recent work by [18] has addressed SIP strategies for joins where compile-time cost models are complemented by run-time benefit estimators so that SIP-based filters are adaptively enabled or disabled while the query is executing. This framework is closest to the SIP techniques developed in the current paper. However, the SIP filters of [18] are built only for results of *completed* operators and exploited only by joins (and group-by). In contrast, our SIP method works for in-progress operators and provides on-the-fly benefits for concurrently running joins as well as index scans. This is crucial for a *pipelined* query processor, and especially attractive for the many joins in our setting of SPO indexes. Our method is extremely light-weight and tailored to RDF processing, whereas [18] focuses on relational engines and needs to carefully optimize the SIP because of its overhead.

3. SIDEWAYS INFORMATION PASSING

3.1 Principles

We assume a system architecture that provides us with indexes on all six permutations of identifier triples: SPO, SOP, PSO, POS, OSP, OPS. These should be clustered B^+-tree indexes, containing directly the triples rather than pointers to triples. Mapping dictionaries between identifiers and the full literals are kept in separate indexes. Merge joins operate directly on the triples indexes; merge operators can be pipelined when no intermediate sort is needed (which would often be the case with the rich choices of indexes to drive the merge steps). Additionally, hash joins will be considered, too. Finally, we assume that a join-order optimizer chooses a cost-model-based optimal or near-optimal ordering for binary joins (for queries with up to 10-20 joins). Optimality is meant relative to the cost model; so misestimation of subquery selectivities and execution costs may degrade the plan quality (see Section 4). All this is available as a package in the RDF-3X engine. Other RDF engines provide subsets of these performance features; the missing features could be added with reasonable programming effort to these systems.

Although such an architecture is already a very powerful machinery for executing advanced SPARQL queries, it exhibits performance deficiencies when the underlying RDF graphs and the resulting indexes become very large. As an example, consider again the operator tree from Section 1, shown in Figure 2 with its inputs from the underlying index scans. Recall that all index scans and merge joins are pipelined, so conceptually all operators run in parallel. The scan for the inCategory=Crime predicate produces many results, and it seems inevitable that the - potentially very long - index range with Crime books needs to be completely read. The scan for the inLanguage=French predicate is much more selective, but the join of these two scans may still produce a fairly large output. On the other hand, the scan for the won-

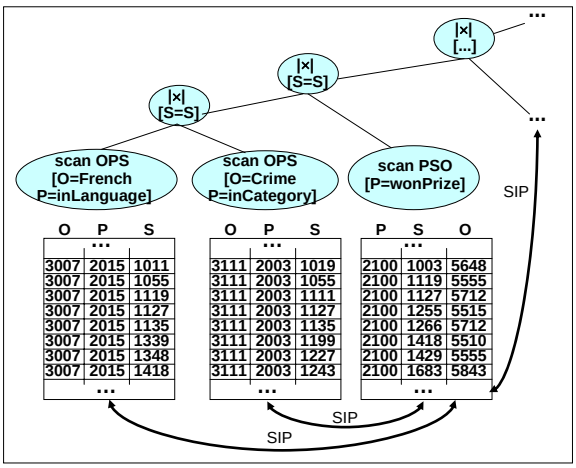

Figure 2: Operator Tree and its Index Inputs

Prize predicate is very selective, as only few books have won awards. So the subsequent join with the left-hand subtree will eventually produce a small result.

This example illustrates the key problem that queries over large RDF graphs need to scan large fractions of indexes and merge long lists of intermediate results. The pattern matching nature of SPARQL makes it surprisingly difficult to avoid this phenomenon, as individual triple patterns can be very unselective, matching a large part of the RDF graph. Only the subsequent joins eliminate the irrelevant triples.

Figure 2 also illustrates the main idea towards reducing the data volume that needs to be processed during scans and merge joins (and also hash joins, as we will see later). We observe that for the entire tree to produce a result, the S values in the left-hand subtree need to find a matching S value in the scan on the PSO index for the wonPrize predicate. By the pipelined nature of the overall query execution, the scan on PSO proceeds in parallel with the merge on the two OPS scans, and can, therefore, send information about its S values to the two OPS scans. In particular, whenever the PSO scan encounters large gaps in the S values, e.g., the ones between 1127 and 1255 and between 1266 and 1418, this is valuable information that would allow the other scans to skip large parts of their indexes. This is exactly what our SIP strategy does. In the figure, the PSO scan passes information about gaps to the otherwise unselective OPS scan for the inCategory predicate. In addition, the two fairly selective scans for inLanguage and wonPrize *mutually* exchange information about gaps. This is exactly the effect that one would obtain from semi-join programs or magic-set rewritings. However, in contrast to these compile-time methods, our SIP strategy is a light-weight run-time method that fits nicely with pipelined executions and scales very well with increasingly complex join trees.

We refer to our method as *ubiquitous SIP*, or *U-SIP* for short, because of is dynamic and light-weight nature: it consists of on-the-fly information throughout the execution of operators. This is in contrast to previously proposed SIP strategies such as [18] where filter information is passed on only when the producing operator has completed its execution. Moreover, U-SIP is conceptually enabled between all (meaningful) pairs of operators and does not need any cost-conscious compile-time optimization and, thus, neither any complicated run-time adjustments of cost estimates.

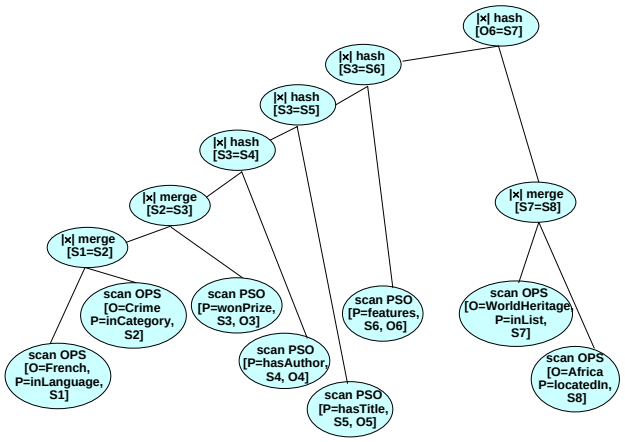

Figure 3: Operator Tree with Explicit Variables

In general, our solution for reducing the data volume to be read or merged is to enhance each operator in an operator tree with the following capabilities:

- Knowledge about which other operators can potentially benefit from value-gap information. This is derived at query compile-time, based on reasoning about possible variable bindings for the given query.
- Light-weight bookkeeping about significant gaps in sorted value lists seen during scans or merges, and light-weight domain filters built by hash joins. The key point here is to keep the overhead extremely low.

All run-time mechanisms work perfectly asynchronously on a per-operator basis. This is key to adding dynamic SIP without interfering with the pipelining architecture and potential multi-threading of the execution engine. This aspect is very similar to the eddies approach [3] for adaptive query processing, but eddies focused on join re-ordering whereas we address the very different issue of light-weight SIP.

3.2 Compile-Time Preparation

Although RDF is based on triples and index scans always process triples, cascading joins in composite SPARQL queries actually build tuples with potentially higher arity and the output projection may also contain bindings of more than three variables. To make this situation clearer and allow precise reasoning about variable bindings, we represent execution plans for SPARQL queries as operator trees with separate variables for each operator. So variables correspond to the notion of variables in a domain calculus rather than a relational tuples calculus.

This is illustrated in Figure 3 for the example query of Section 1. The figure shows the full query with different choices of join implementation by merge or hash join; the subquery of Figure 2 corresponds to a subtree in the lower left part. Note that the query includes a join predicate of the form object=subject, in addition to the typical star-join pattern with subject=subject predicates. In SPARQL, queries may even contain property=object conditions. Although these may look odd from a schema-first relational purist viewpoint, real-life, heterogeneously modeled RDF data like the UniProt collection may naturally lead to such queries.

Now we would like to augment the query operator tree with *U-SIP edges* that denote the potential sideways flow of information between operators. The basis for deriving these edges is the concept of *equivalent variables*. Two variables are considered equivalent ($v_1 \equiv v_2$) if they must have exactly the same bindings to values. Consequently we derive the equivalence class $E(v)$ of a variable v by computing the set of equivalent variables:

$$\begin{aligned} v_1 \equiv v_2 &:= \text{an operator contains the condition } v_1 = v_2 \\ E_0(v) &:= \{v\} \\ E_n(v) &:= \{v_2 | v_1 \in E_{n-1}(v) \wedge v_1 \equiv v_2\} \\ E(v) &:= \cup_{n=0}^{\infty} E_n(v) \end{aligned}$$

Note that the equivalence classes derived from SPARQL queries tend to be much larger than the equivalence classes that one would see in corresponding SQL queries. Triple patterns frequently form stars around certain subjects to specify desired properties. In addition to triple patterns, explicit filter conditions, written as `FILTER(?a=?b)` in SPARQL, also contribute to the equivalence classes. In the operator tree of Figure 3, the variables $\{S1, S2, S3, S4, S5, S6\}$ form one equivalence class and $\{O6, S7, S8\}$ form a second class.

U-SIP Edges. The equivalence classes describe which variables must end up with the same binding to produce an output tuple. During query execution, we therefore keep track of the *potential domain* of each equivalence class, and discard variable bindings that can never make it into the result. The key point of our U-SIP method is to augment the operator tree by run-time means for passing domain information about variable bindings between *all* pairs of index scans that have variables in the same equivalence class. We will discuss in the following subsections how to implement this with very low overhead. For index scans that drive merge joins, only a single "next" (or gap) value needs to be communicated between operators. In addition, we augment hash joins to build compact domain filters that are available to all subsequent operators.

Next Information. The very compact "next" (gap) information is beneficial only for operators within the same pipeline. Therefore, we limit the U-SIP edges to those pairs of index scans that reside in the same operator pipeline. Pipeline-breaking operators, such as hash joins (i.e., the hash-table build part) or explicit sorting (e.g., on behalf of grouping or output ordering – a feature not in the current SPARQL standard, but well conceivable), incur a partitioning of the variable equivalence classes. More formally, a bidirectional U-SIP edge is created between two index-scan operators o_1 and o_2 if and only if there is no pipeline-breaking operator in the subtree rooted at the lowest common ancestor of o_1 and o_2. In the example of Figure 3, we create U-SIP edges for all pairs in $\{S1, S2, S3\}$ and for $\{S7, S8\}$.

Domain Filters. In addition to these U-SIP edges, global *domain filters* are planned for all equivalence classes with a variable that is used for the build part of a hash join. We use compact Bloom filters as a representation. These are held in shared memory and are accessible by all operators (details given below). In the example, this procedure leads to filters for the equivalence classes $\{S3, S4, S5, S6\}$ and $\{O6, S7, S8\}$. S1 and S2 do not directly contribute to the filter for the first equivalence class, because they refer to scans and merge joins that will run before the build phase of the hash join for the S3=S4 predicate. In general it is sufficient to add one equivalence class representative per subtree to the filter, as all equivalent variables will have the same value bindings.

The overall result of the compile-time preparation is an

augmented execution plan that tells the run-time system where SIP information should be exchanged between operators and where domain filters should be accumulated in global domain filters shared by all (subsequent) operators.

3.3 Run-Time Handling of Merge Joins

Merge joins read their inputs in ascending order of join-variable values. Therefore, all equivalent variables involved in a merge join can be safely assigned the largest current variable binding among these variables. We refer to this value as the *next binding* for the equivalence class.

The bookkeeping for computing next bindings is very simple and light-weight. Operators like index scans and merge joins maintain cursor positions and a one-step look-ahead anyway. To pass SIP information to other operators, they merely need to record their next bindings in a shared-memory data structure. The receiving operator can fetch these values, one for each "sending" operator, and compute the maximum at its own discretion. This includes the option of dynamically enabling or disabling the SIP strategy, based on the "density" of the observed next-binding values, possibly for a subset of variables in an equivalence class. It is not clear, though, to what extent such sophistication pays off. Subsection 3.5 will present a simple heuristic strategy that appears to be most effective with almost zero overhead.

3.4 Run-Time Handling of Hash Joins

In contrast to merge joins, the out-of-order nature of hashing does not allow us to implement skipping by merely identifying gaps and communicating a single "next" value. However, we can still use SIP information for skipping by capturing an approximation of the join variable's actual domain. To this end, we would need to capture an entire set of values. This may become very large and lead to a fairly expensive full-fledged semi-join. In this situation, the cost of the semi-join may be prohibitive relative to its expected benefit. Instead, we use Bloom filters, i.e., hash-based bit vectors, to represent the observed values, which require only a fixed, small amount of space and can be probed very efficiently. If we observe only few values, the Bloom filters perform very well (with few false positives).

For each equivalence class of variables, we construct a *domain filter* describing the potential variable bindings. It consists of a Bloom filter and range bounds:

min	max	Bloom filter (1024 bytes)

We distinguish between the *potential domain*, which is the same for all variables in an equivalence class, and *observed domains* which are maintained by the build phases of individual hash join operators. The potential domain is initially $(0, \infty, 1*)$ as every value is valid, while observed domains are initialized to $(\infty, 0, 0*)$ as no values have been observed yet. As we want to use Bloom filters for testing the *absence* of certain value *ranges*, we cannot use arbitrary hash functions but require that the function $h(x)$ for mapping values x onto bit-vector positions between 0 and $m-1$ (for bit-vector length m) is *distance-preserving* in the following sense (related to order-preserving hashing, but a weaker condition). For a bit vector $D[0..m-1]$ and values x, y we require:

$$\begin{array}{lll} ((h(x) < h(y) & \wedge & D[h(x)] = D[h(y)] = 1 \\ & \wedge & \forall i : h(x) < i < h(y) : D[i] = 0) \quad \vee \\ (h(x) > h(y) & \wedge & D[h(x)] = D[h(y)] = 1 \\ & \wedge & \forall i : i > h(x) \vee i < h(y) : D[i] = 0)) \\ \Rightarrow & & (y < x \quad \vee \quad |y - x| > c(h(y) \ominus h(x))) \end{array}$$

with a positive constant c and a modulo-m difference $\ominus$ defined as:

$$j \ominus i = \begin{cases} j - i & \text{if } j > i \\ m - i + j & \text{otherwise} \end{cases}$$

Although this looks complicated, it is actually a fairly simple condition stating that a run of zero bits in the bit vector between $h(x)$ and $h(y)$, where runs can be extended modulo m, denotes that the next set bit corresponds to a value y with a specific distance to x – provided that $y > x$. For the case $y < x$ no statement is possible (and not needed either by our method). A family of hash functions with this property is the transformation $h(x) = ax \mod m$ with a positive multiplicative constant a. In the following, we commit ourselves to this family of functions for ease of presentation.

All domain filters are kept in shared memory as globally accessible data structures. Every hash-join build operator first creates its private Bloom filter. When the build is completed, the private filter is intersected with the shared-memory global filter. This is done - at different points - by all hash builds that have variables in the same equivalence class. Note that, unlike with merge joins, the variables whose domains are updated by a hash build are not necessarily join variables. Rather, the hash build can update the filtes for all variables that it sees in its input tuples. For example, a join whose build-part variable refers to a subject can provide a filter on the property or object variables of its input, and this may prove beneficial for subsequent joins, filters, or scans on these variables or even other variables in their equivalence classes.

After building a hash table from input containing a variable v, we know that we have seen all possible bindings of all variables in $E(v)$. Most likely we have seen a superset, as some potential bindings will be discarded by subsequent joins, but no other binding can make it into the final query result. If we later scan an index with a variable from $E(v)$, we can skip all entries that have values not seen in the build phase. But we need to "decode" the "next" value from the corresponding Bloom filter D. This is where we need the distance-preservation of the filter's hash function. Given the scan's current binding $cur(v)$, according to its cursor position, we can compute a lower bound for the gap that we can skip. The next possible binding for variable v is:

$$next(v) := \begin{cases} next(D.min) & \text{if } cur(v) < D.min \\ \infty & \text{if } cur(v) > D.max \\ cur(v) & \text{if } D.filter \text{ contains } cur(v) \\ cur(v) + \frac{1}{a}(x1 \ominus h(cur(v))) & otherwise \end{cases}$$

where $x1$ is the position of the next one-bit entry in the bit-vector modulo m, and $\ominus$ is the modulo-m difference.

The pruning power of the domain filters increases over time, as the domain becomes more and more restrictive. Note that one could update the domain more frequently than just during the build phase of the hash join. However, updating the domain too early (e.g., during the scans or sooner after the scans) has little impact on pruning and causes non-negligible CPU overhead. Hash joins not only build domain filters but also probe them during their build phases: they discard all tuples with values outside the corresponding potential domains, eliminating intermediate tuples.

3.5 Run-Time Acceleration of Index Scans

Index scans that receive SIP information may skip entries in the B$^+$-tree. When reading a triple containing the variable v, the scan can immediately skip to the $max\{next(x)|x \in E(v)\}$ value. If the target is outside the current page, the skip is made using the B$^+$-tree, skipping all irrelevant entries – potentially several leaf pages.

The main question is how often a scan should check for skips. In principle, one could check after every triple, but we found that this causes significant CPU overhead and would disrupt the high L2 cache locality of sequential scans. We therefore check for skipping only when reading a new page into memory. Reading a new page requires more complex processing anyway for uncompressing the page contents. We use the first entry on the page's list as a candidate binding for the variables, and then compute the next possible binding. If this is beyond the current page, we immediately skip the whole page and use the B$^+$-tree to retrieve the next relevant page. On the other hand, if the next possible binding is still on the current page, we skip all triples in between, but read all subsequent triples on the current page without skip checks.

The described technique reduces the amortized CPU costs to nearly zero (there are many thousands of triples on a page), while still allowing us to skip large chunks of irrelevant index parts.

4. SELECTIVITY ESTIMATION

4.1 Problem and Approach

The good performance of RDF-3X relies on join-order optimization, which in turn requires reasonably accurate selectivity estimates. The original paper [25] built equi-depth histograms by aggregating triples into buckets, and also introduced frequent-paths synopses. These statistics were constrained by a fixed-size space budget, so that all data structures for selectivity estimation would fit in a small amount of memory. This works well for RDF datasets with tens of millions of triples and a moderate number of different property names. But for the billion-scale datasets that we consider in this paper, we found that both the size of the data and the increasing diversity of property names lead to inaccurate estimators that would misguide the query optimizer. We provide quantitative evidence for this claim in Subsection 4.4.

One way to avoid this degradation would be to enlarge the space for statistical synopses in proportion to the data size, but heterogeneity and other factors may actually require a superlinear increase to keep the estimators accurate. Moreover, whatever memory we allocate for statistics is no longer available for data caching and working memory which are not free at all in a multi-user server with high throughput demands. In general, selecting the right kinds of synopses and their sizes is a fairly difficult task; it may be viewed as a smaller-scale version of the physical-design problem.

To escape these problems, we pursue a radically different approach. Instead of using aggregated statistics, we compute *exact* result cardinalities for single triple patterns with at least one variable. This is done at query compile-time, as part of finding the best execution plan, but we do consider it as part of the user-perceived response time as RDF queries may be generated by interactive tools and during running applications. As we will see below, the computation for selectivity estimation typically requires one or two lookups in small-sized B$^+$-trees, which is insignificant compared to the actual query execution on huge RDF graphs. In the following subsections, we discuss the estimators for individual triple patterns, which can use aggregated indexes that already exist in RDF-3X, and for joins between two triple patterns, for which we construct and materialize additional specifically designed index structures with affordable space consumption.

4.2 Individual Triple Patterns

The existing index structures on all SPO permutations and their pre-aggregated binary projections SP, PS, SO, OS, PO, and OP and unary projections S, P, and O (see [25]) allow us to efficiently count the number of triples that match a single pattern without materializing the full triples themselves. The binary-projection indexes contain all (identifier-encoded) values for an SP pair (or a PS pair etc.) and precomputed counts for their frequencies. The unary-projection indexes contain all S values (or P or O values) and precomputed counts for their frequencies. Note that these aggregated indexes are orders of magnitude smaller than the full-triple indexes for SPO, PSO, etc.

If a triple pattern contains no variable, its result cardinality is 1 (assuming the result is not empty). If it contains only variables, the result cardinality is the number of triples in the database, which is known from the available database statistics. Otherwise the triple pattern consists of one or two constants and two or one variable. We can use the aggregated indexes on binary or unary projections to look up the cardinality, using the constants as search keys and performing a point query in the appropriate aggregated index.

4.3 Joins Between Two Triple Patterns

When estimating join selectivity estimates, there is a trade-off between accuracy and computational costs. Computing the exact join cardinality is nearly as expensive as actually executing the join. However, we can bring a join between two triple patterns into a form that is much easier to estimate, assuming independence of the filter predicates in the two triple patterns. This is illustrated by Figure 4, where $c1, c2, c3, c4$ denote constants and $s1, p1, o1, s2, p2, o2$ are variables. A join of two triple patterns (Figure 4 a) can be interpreted as a self-join of all triples with selections for the patterns (Figure 4 b), which can be transformed into a join of one triple patterns with all triples in the database and a final selection (Figure 4 c). This last form would be very inefficient for query execution, but as it is equivalent to the original form, we can use it for selectivity estimation. We can compute the selectivity of the top-most selection easily by computing the result cardinality of the second triple pattern. Then the join selectivity estimation boils down to estimating the *selectivity of one triple pattern joined with all other triples*.

There are three different cases for joining a triple pattern with all triples in the database, depending on the number of variables in the triple pattern. To participate in a join, a triple pattern must contain at least one variable (otherwise one cannot formulate a join condition). In the one-variable case, all triple patterns have the form (c_1, c_2, v) with join variable v (which can be in each of the S, P, O positions, but we simplify the presentation by assuming that v is in the last position). There are three ways to join this triple pattern with all other triples, namely by comparing v to the

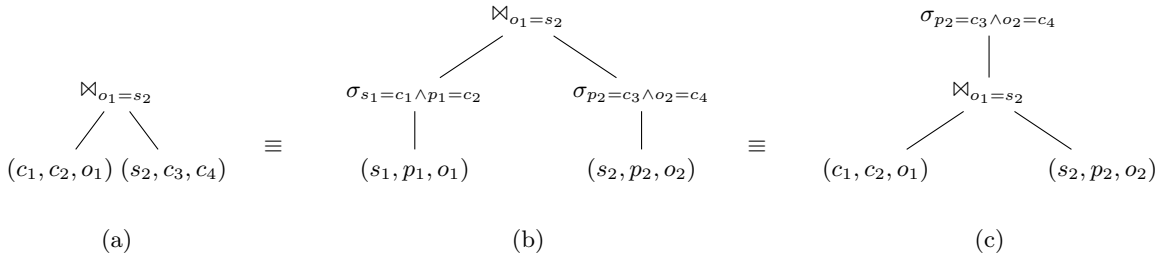

Figure 4: Reformulating a pattern join to simplify selectivity estimation

subject, property, or object of the triples in the database. Assuming we join with the subject, we want to compute

$$\begin{aligned} & sel((c_1, c_2, v) \bowtie_{v=s_2} (s_2, p_2, o_2)) \\ = & \frac{|(c_1, c_2, v) \bowtie_{v=s_2} (s_2, p_2, o_2)|}{|(c_1, c_2, v)||s_2, p_2, o_2|} \\ = & \frac{\sum_{x \in \Pi_v(c_1, c_2, v)} |(x, p_2, o_2)|}{|(c_1, c_2, v)||s_2, p_2, o_2|} \end{aligned}$$

The last form, using a summation instead of counting the join result size, is a bit unusual, but the computation is directly supported by the available index structures. The unary-projection indexes give $|(x, p_2, o_2)|$ (for each value x of the join variable v) with one B$^+$-tree point query; thus the whole summation can be computed efficiently by using an index-based nested-loop join. This yields the desired join selectivity. The v = *property* and v = *object* cases are analogous.

The above nested-loop join would be affordable at compile-time, but when a query is issued, optimized, and executed within the same user-visible time window, this selectivity computation may be too expensive. Therefore, we precompute the join selectivities by the above method, for all possible choices of the one or two constants in a triple pattern, and we materialize the result in additional indexes (previously not existing in RDF-3X). There are three possible way to have a triple pattern with one variable (the variable is S, P, or O); so we perform three separate computations and store the results in B$^+$-trees indexed by the two constants included in the pattern. Building these indexes is still relatively inexpensive. In our experiments with billion-scale datasets, the computation costs were dominated by the full table scan required for each index, as the unary projection indexes are very small and cheap to join.

For organizing the leaf pages of these additional indexes, we dediced to deviate from the layout that RDF-3X uses for its regular indexes. For selectivity estimation, we will only read two leaf pages for each join. Moreoever, we pay attention to good compression for best possible in-memory caching. For each entry in a leaf page we store the tuple $(c_1, c_2, s_s, s_p, s_o)$, i.e., the constants and the size of the join result when comparing v with subject, property, and object of all triples in the database. We store these tuples organized by position, that is, first all values of c_1, then all of c_2, etc. We use byte-wise Gamma encoding for all values, and gap compression to reduce the values of c_1 and c_2. This results in long sequences of very similar values, whose space consumption is further reduced by LZ77 compression.

The scenario with two variables in the triple pattern is similar to the one-variable case, except that we now have six join possibilities. We compute all six of them, and store the counts as described above. Overall the index is much smaller, as there is only one constant in the triple pattern and thus much fewer combinations. As we have three possible positions for the constant in the triple, we again build three index structures.

The last scenario is a triple pattern with three variables, i.e., all positions are unknown. This boils down to a self join of all triples with themselves. There are nine possible ways to join the triples, we thus compute all nine possible result cardinalities via merge joins over the unary projection indexes and store them.

Using these materialized join statistics, we can compute the join selectivity between arbitrary triple patterns very accurately. In fact the prediction is exact if the independence assumption between selection predicates in the participating triple patterns holds. Note that the computation is asymmetric. We keep the constants of one triple pattern and treat the other triple pattern as a full table scan followed by a selection after the join (cf. Figure 4 (c)). When the predicates are not independent, this creates an estimation error. We are free to choose for which of the two patterns we keep the constants, we pick the one with the smaller result cardinality as this reduces the uncertainty and thus improves accuracy. Overall these statistics give extremely good estimation results, with only a slight increase in database size.

4.4 Measurements

We measured the accuracy of these techniques by using the Billion Triple dataset from Section 5 and comparing the predicted join selectivites for all joins occuring in the 8 benchmark queries with the real join selectivities. The benchmark workload contains a total of 60 joins between two triple patterns; we aggregated the relative errors over these 60 joins:

$$relative\,error = \frac{|actual\ selectivity - estimated\ selectivity|}{actual\ selectivity}$$

where *selectivity* (between 0 and 1) denotes the fraction of the Cartesian-product size between the two participating triple patterns.

The results including lookup times are shown in Figure 5. Our approach improves the estimation accuracy by several orders of magnitude, which is essential for high-quality query optimization. On average (mean error), we misestimate the selectivity by a factor of 6, whereas the coarse-grained histograms of the original RDF-3X system lead to huge estimation errors that can easily misguide the join-order optimizer.

It takes 30 minutes to precompute the new indexes and they occupy 3 GB on disk. But this extra cost needs to be seen in relation to a total of 11 hours for bulk-loading and indexing the dataset (using the original RDF-3X) and a total database size of 41 GB. So we pay less than 10 percent overhead at database build-time. At query compile-

method	lookup time [μs]	min error	5% quantile	median error	95% quantile	max error	mean error
RDF-3X	7	0.28	24.93	27,939.00	$3.3*10^7$	$8.1*10^7$	$5.5*10^6$
our approach	246	<0.01	<0.01	0.93	82.48	89.06	6.38

Figure 5: Join selectivity estimation errors for the Billion Triple dataset

and-run-time, we pay a much smaller cost in terms of memory consumption because of high locality. For the measured workloads, the relevant pages of the new index structures were almost always in memory in the warm-cache steady-state case, and did not take away any significant amount of memory from the data cache or workspaces of the query processor. This is also underlined by the lookup times for selectivity estimation, shown in Figure 5: compared to RDF-3X, our lookup times are substantially higher, but they are still well below a millisecond and thus negligible relative to the CPU consumption of the query optimizer and the actual query run-times.

5. EVALUATION

We implemented our methods by modifying the open-source system RDF-3X [28]. In the experimental evaluation, we measured the unmodified base system (*RDF-3X*) against the enhanced system that contains our methods (*our approach*). Note that we only compare query-execution performance, as the rest of the system (storage, query translation, etc.) is unchanged.

As additional competitors we included the vertical partitioning from [1] implemented using *MonetDB*, and a triples-store implementation using *PostgreSQL*. For all experiments we first imported the data into RDF-3X, and then used the conversion scripts included in the RDF-3X distribution for building the alternative systems. This guarantees that all systems use exactly the same triples and the same dictionaries for mapping between literals and internal identifiers. For running queries on MonetDB and PostgreSQL, we modified the SPARQL compiler of RDF-3X to generate suitable SQL queries. For the queries in our experiments, the resulting SQL code consisted of simple select-project-join queries (i.e., no nested SQL blocks etc.).

We used two large datasets with over half a billion triples each: a large subset of the *Billion Triples* Challenge [29] and the UniProt [37] collection. In addition, we ran measurements with one dataset from the original RDF-3X paper [25], as a calibration yardstick. The results are discussed in the following subsections.

All experiments were performed on a Dell D620 PC with a 2 Ghz Core 2 Duo processor, 2 GBytes of memory, and running a 64-bit Linux 2.6.24 kernel. As proposed in [25], we performed *cold-cache* experiments by dropping all file-system caches before restarting the various systems and running the queries. We repeated this procedure five times and took the minimum execution time for each query, to avoid artifacts due to background system activity. For *warm-cache* experiments we ran the queries five times without dropping the caches.

5.1 Billion Triples

For the first experiment, we used the dataset of Billion Triples Challenge [29]. It contains data from twelve different sources, including DBpedia, Freebase, Swoogle, and others. We omitted the Webscope part for licensing reasons, and imported all other sources (88GB in N-Triples form). Some data was noisy and violated the N-Triples syntax (usually due to lack of proper escaping of Web-data strings). We ignored all triples that we could not parse with a strict parser. This resulted in 562,469,278 unique triples (about 10 times bigger than the largest RDF dataset used in previous papers on RDF query processing). The space consumption of RDF-3X, MonetDB, and PostgreSQL for this dataset was 41GB, 20GB, and 90GB, respectively. One peculiarity of this dataset is that is has nearly 80,000 distinct property names. This is not an issue for RDF-3X or PostgreSQL, but the vertical partitioning approach using MonetDB handles this poorly when queries have property variables.

We ran queries with increasing complexity, starting from simple ones such as *country, longitude, and latitude of the Eiffel Tower* (Q1) to sophisticated ones such as *actors who are married to each other and born in the same place* (Q7) or *politicians mentioned in news articles about "Blackwater"* (Q8). The SPARQL formulation of all queries is given in the appendix. These queries resemble quiz questions, but they are reasonable representatives of advanced join queries that connect different pieces of information from the various data sources that constitute the Billion Triples collection.

The query run-times are shown in Figure 6. In general MonetDB performs poorly here, PostgreSQL better, and RDF-3X is the best among the previous approaches. Our methods outperform all competitors including the original RDF-3X by a large margin, improving cold cache times by nearly a factor of 4 in the geometric mean, and the warm-cache times by more than a factor of 15. Our approach was consistently best for all queries; for one query we gained more than a factor of 50. The gains are lower in the cold-cache case, as run-times with a cold cache are dominated by disk I/O, and seeking (i.e., disk-arm movement) on disk is slow which reduces the benefit of SIP.

5.2 UniProt

For the second experiment, we used the UniProt dataset [37], which consists of 57GB of protein information. This dataset contains 845,074,885 unique triples – much larger than our first dataset, but there are only 95 distinct property names in the UniProt collection.

We ran different queries, starting from a simple one for the name of a protein (Q1) to finding annotations of proteins in E. coli bacteria (Q8). The SPARQL formulation of all queries is given in the appendix.

The query run-times are shown in Figure 7. Here, the performance gains by our approach are even greater, as the queries are more complex than the queries on the first dataset. PostgreSQL performed surpisingly well in this experiment, but the original RDF-3X still outperformed PostgreSQL. Our methods improved RDF-3X by more than a factor of 8 in the cold-cache case and more than a factor of 50 for warm caches. For several queries, we gained more than a 100x improvement over the original RDF-3X; for the most complex query we are more than three orders of magnitude faster than all competitors. Note that these improvements are relevant not only for accelerating individual queries, but the savings in resource consumption directly translate into

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	geom. mean
cold caches									
our approach	2.95	0.33	4.01	26.84	3.59	0.56	3.99	27.28	3.50
RDF-3X	4.05	0.37	4.43	120.45	6.75	1.23	>30min	45.62	>12.36
MonetDB	>30min	>30min	654.06	>30min	>30min	702.40	19.60	>30min	>801.40
PostgreSQL	17.12	>30min	641.64	373.85	>30min	7.65	>30min	51.11	>235.19
warm caches									
our approach	0.15	0.01	0.16	1.87	0.18	0.05	0.29	1.99	0.19
RDF-3X	0.56	0.02	0.33	107.45	3.71	0.78	>30min	15.28	>3.65
MonetDB	>30min	>30min	646.92	>30min	>30min	680.20	0.92	>30min	>543.81
PostgreSQL	1.56	>30min	639.64	212.96	>30min	1.38	>30min	10.36	>107.39

Figure 6: Query run-times in seconds for the Billion Triples dataset

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	geom. mean
cold caches									
our approach	1.33	10.03	0.62	10.62	13.74	0.25	0.31	0.41	1.57
RDF-3X	27.93	179.29	0.84	>30min	15.72	3.81	0.83	2.11	>12.95
MonetDB	54.92	2024.51	924.21	>30min	737.90	1300.26	811.06	1013.53	>786.14
PostgreSQL	8.46	46.06	2.95	>30min	>30min	1.29	1.51	2.44	>19.09
warm caches									
our approach	0.08	0.78	0.02	1.47	0.70	0.01	0.01	0.01	0.07
RDF-3X	5.18	160.88	0.12	>30min	2.22	2.90	0.12	0.26	>3.71
MonetDB	2.80	1992.22	917.02	>30min	726.72	1285.46	809.08	949.75	>533.97
PostgreSQL	4.84	2.52	1.60	>30min	>30min	0.24	0.22	0.36	>5.75

Figure 7: Query run-times in seconds for the UniProt dataset

a much higher throughput for multi-user servers or much lower cost/performance for cloud-computing platforms.

5.3 Effect of Individual Techniques

To measure the effects of our individual techniques in isolation, we ran all experiments with a variant where only our methods for sideways information passing were enabled (*U-SIP only*) and selectivity estimation used the original RDF-3X techniques, and with a variant without SIP but with our new selectivity estimation methods (*estimator only*).

The results are shown in Figure 8. They show that both techniques are important, depending on the characteristics of the underlying data. For the Billion Triples dataset, the improved selectivity estimator is more important: there are fewer possibilities for skipping data due to the heterogenity of the dataset, but this diversity makes selectivity estimation much more critical. As shown by the warm-cache results, U-SIP helps skipping data, but this yields only moderate gains. The estimator-only variant achieves a 10x improvement over RDF-3X and is relatively close to the performance of our full-fledged approach (within a factor of 2).

The UniProt data is more homogeneous; here the effect of U-SIP is much more prominent, saving a factor of 2-3 for the entire workload and more than one order of magnitude for some of the queries. Selectivity estimation is still crucial, though; a bad join order severely hurts performance.

The combination of both methods (*our approach*) achieves major benefits compared to both of the limited variants, especially in the warm-cache case. Clearly, there are synergetic effects between better selectivity estimation and the SIP strategies. Better join orders also enable better opportunities for run-time skipping.

5.4 Experiments on Smaller Graphs

For completeness we also performed experiments on the smaller RDF graphs used in [25]. The results for the Library-Thing dataset (36,203,751 triples) from the original RDF-3X paper are shown in Figure 9; see [25] for the queries. Here, SIP is not that beneficial anymore, as there is less data to be skipped. But it is interesting to see that the overhead of U-SIP is negligible; in fact, our approach still achieves small but consistent improvements over RDF-3X.

6. CONCLUSION

We believe that our work is another important step towards providing scalable database capabilities for very large Semantic-Web data in RDF format. We started with the already well-engineered RDF-3X system, identified its bottlenecks for billion-scale datasets and advanced declarative queries, and improved its response times by more than an order of magnitude without incurring any non-negligible overhead. Applications that can benefit from scalable RDF processing are not only Semantic-Web endeavors, but include biological repositories and social communities.

Although we used RDF-3X as a testbed, our contributions are general and could be adopted by other engines as well (possibly with higher coding efforts). The very light-weight "ubiquitous" sideways information passing in pipelined operator trees is a novel method that can greatly accelerate index scans, the basic "workhorse" in query processing with a huge performance impact. The new indexes for selectivity estimation present an interesting deviation from the general paradigm of compact statistical synopses, by investing considerable but affordable space to improve estimation accuracy while keeping the estimation itself very fast.

One of the most intriguing open issues that we plan to address in future work is adapting our methods to multi-core processors. Methods from parallel DBMS's, developed a decade ago, are obviously a good starting point, but may require rethinking in the specific context of RDF data and memory- and CPU-intensive processing of complex queries.

Billion									
	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	geom. mean
cold caches									
our approach	2.95	0.33	4.01	26.84	3.59	0.56	3.99	27.28	3.50
estimates only	3.25	0.35	4.39	30.34	3.87	0.58	4.07	31.45	3.80
SIP only	3.13	0.36	4.04	104.85	6.55	1.24	>30min	34.34	>11.15
RDF-3X	4.05	0.37	4.43	120.45	6.75	1.23	>30min	45.62	>12.36
warm caches									
our approach	0.15	0.01	0.16	1.87	0.18	0.05	0.29	1.99	0.19
estimates only	0.28	0.02	0.34	3.71	0.36	0.08	0.38	4.05	0.36
SIP only	0.15	0.01	0.16	94.22	3.82	0.78	>30min	2.28	>2.01
RDF-3X	0.56	0.02	0.33	107.45	3.71	0.78	>30min	15.28	>3.65

UniProt									
	Q1	Q2	Q3	Q4	Q5	Q7	Q8	Q9	geom. mean
cold caches									
our approach	1.33	10.03	0.62	10.62	13.74	0.25	0.31	0.41	1.57
estimates only	1.72	63.27	0.69	181.80	16.49	0.27	0.38	0.45	3.17
SIP only	11.16	107.40	0.71	>30min	15.39	0.25	0.65	1.63	>7.07
RDF-3X	27.93	179.29	0.84	>30min	15.72	3.81	0.83	2.11	>12.95
warm caches									
our approach	0.08	0.78	0.02	1.47	0.70	0.01	0.01	0.01	0.07
estimates only	0.22	9.48	0.08	111.75	1.64	0.01	0.01	0.02	0.29
SIP only	5.38	98.86	0.02	>30min	2.16	0.01	0.04	0.07	>1.01
RDF-3X	5.18	160.88	0.12	>30min	2.22	2.90	0.12	0.26	>3.71

Figure 8: Run-times in seconds with individual techniques

	A1	A2	A3	B1	B2	B3	C1	C2	geom. mean
cold caches									
our approach	0.23	0.91	20.23	0.17	0.25	2.99	0.25	1.09	0.76
RDF-3X	0.28	1.01	21.85	0.14	0.34	4.17	0.28	1.21	0.89
MonetDB	2.14	1.41	1220.09	1.63	2.20	>30min	1.66	>30min	>22.72
PostgreSQL	20.78	1.43	715.64	0.88	2.13	>30min	5108.01	1031.63	>66.40
warm caches									
our approach	0.01	0.04	1.17	0.02	0.09	0.78	0.02	0.12	0.05
RDF-3X	0.05	0.15	0.95	0.01	0.12	1.61	0.03	0.26	0.13
MonetDB	0.82	0.77	1171.82	0.56	0.63	>30min	0.59	>30min	>12.22
PostgreSQL	12.31	0.05	611.41	0.02	0.66	>30min	5082.34	1013.01	>21.52

Figure 9: Query run-times in seconds for the LibraryThing dataset

7. REFERENCES

[1] D. J. Abadi et al. Scalable semantic web data management using vertical partitioning. In *VLDB*, 2007.

[2] S. Auer et al. Dbpedia: A nucleus for a web of open data. In *ISWC/ASWC*, 2007.

[3] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD*, 2000.

[4] P. A. Bernstein and D.-M. W. Chiu. Using semi-joins to solve relational queries. *J. ACM*, 28(1):25–40, 1981.

[5] Bio2RDF, semantic web atlas of postgenomic knowledge about human and mouse. http://www.bio2rdf.org/.

[6] U. Bojars et al. Interlinking the social web with semantics. *IEEE Intelligent Systems*, 23(3), 2008.

[7] J. Broekstra et al. Sesame: An architecture for storing and querying rdf data and schema information. In *Spinning the Semantic Web*, 2003.

[8] E. I. Chong et al. An efficient sql-based rdf querying scheme. In *VLDB*, 2005.

[9] Dbpedia 3.2 downloads. http://wiki.dbpedia.org/Downloads32.

[10] D. DeHaan and F. W. Tompa. Optimal top-down join enumeration. In *SIGMOD*, 2007.

[11] A. Deshpande et al. Adaptive query processing. *Foundations and Trends in Databases*, 1(1), 2007.

[12] Open directory RDF dump. http://rdf.dmoz.org/.

[13] Metaweb technologies: Freebase data dumps. http://download.freebase.com/datadumps/.

[14] C. A. Galindo-Legaria, A. Pellenkoft, and M. L. Kersten. Fast, randomized join-order selection - why use transformations? In *VLDB*, 1994.

[15] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[16] A. Y. Halevy, M. J. Franklin, and D. Maier. Principles of dataspace systems. In *PODS*, pages 1–9, 2006.

[17] A. Harth et al. Yars2: A federated repository for querying graph structured data from the web. In *ISWC/ASWC*, 2007.

[18] Z. G. Ives and N. E. Taylor. Sideways information passing for push-style query processing. In *ICDE*, 2008.

[19] Jena: a Semantic Web Framework for Java. http://jena.sourceforge.net/.
[20] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.
[21] A. Maduko et al. Estimating the cardinality of rdf graph patterns. In *WWW*, 2007.
[22] G. Moerkotte and T. Neumann. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *VLDB*, 2006.
[23] MonetDB. http://monetdb.cwi.nl/.
[24] I. S. Mumick and H. Pirahesh. Implementation of magic-sets in a relational database system. In *SIGMOD*, 1994.
[25] T. Neumann and G. Weikum. Rdf-3x: a risc-style engine for rdf. *PVLDB*, 1(1):647–659, 2008.
[26] OpenRDF. http://www.openrdf.org/index.jsp.
[27] PostgreSQL. http://www.postgresql.org/.
[28] RDF-3X. http://www.mpi-inf.mpg.de/~neumann/rdf3x.
[29] Semantic web challenge 2008. billion triples track. http://challenge.semanticweb.org/.
[30] P. Seshadri et al. Cost-based optimization for magic: Algebra and implementation. In *SIGMOD*, 1996.
[31] L. Sidirourgos et al. Column-store support for rdf data management: not all swans are white. *PVLDB*, 1(2):1553–1563, 2008.
[32] M. Steinbrunn et al. Bypassing joins in disjunctive queries. In *VLDB*, 1995.
[33] K. Stocker et al. Integrating semi-join-reducers into state of the art query processors. In *ICDE*, 2001.
[34] M. Stocker et al. Sparql basic graph pattern optimization using selectivity estimation. In *WWW*, 2008.
[35] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: a core of semantic knowledge. In *WWW*, 2007.
[36] O. Udrea, A. Pugliese, and V. S. Subrahmanian. Grin: A graph based rdf index. In *AAAI*, 2007.
[37] Uniprot RDF. http://dev.isb-sib.ch/projects/uniprot-rdf/.
[38] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.
[39] K. Wilkinson et al. Efficient rdf storage and retrieval in jena2. In *SWDB*, 2003.
[40] F. Wu and D. S. Weld. Automatically refining the wikipedia infobox ontology. In *WWW*, 2008.

APPENDIX

A. QUERIES

For completeness we include the SPARQL queries used in our evaluation.

Billion Triples Dataset.

prefix geo: <http://www.geonames.org/>,
pos: <http://www.w3.org/2003/01/geo/wgs84_pos#>,
dbpedia: <http://dbpedia.org/property/>,
dbpediares: <http://dbpedia.org/resource/>,
owl: <http://www.w3.org/2002/07/owl#>
Q1: select ?lat ?long where { ?a [] "Eiffel Tower". ?a geo:ontology#inCountry geo:countries/#FR. ?a pos:lat ?lat. ?a pos:long ?long. }
Q2: prefix select ?b ?p ?bn where { ?a [] "Tim Berners-Lee". ?a dbpedia:dateOfBirth ?b. ?a dbpedia:placeOfBirth ?p. ?a dbpedia:name ?bn. }
Q3: select ?t ?lat ?long where { ?a dbpedia:wikilink dbpediares:List_of_World_Heritage_Sites_in_Europe. ?a dbpedia:title ?t. ?a pos:lat ?lat. ?a pos:long ?long. ?a dbpedia:wikilink dbpediares:Middle_Ages. }
Q4: select ?l ?long ?lat where { ?p dbpedia:name "Krebs, Emil". ?p dbpedia:deathPlace ?l. ?c [] ?l. ?c geo:ontology#featureClass geo:ontology#P. ?c geo:ontology#inCountry geo:countries/#DE. ?c pos:long ?long. ?c pos:lat ?lat. }
Q5: select distinct ?l ?long ?lat where { ?a [] "Barack Obama". ?a dbpedia:placeOfBirth ?l. ?l pos:lat ?lat. ?l pos:long ?long. }
Q6: select distinct ?d where { ?a dbpedia:senators ?c. ?a dbpedia:name ?d. ?c dbpedia:profession dbpediares:Veterinarian. ?a owl:sameAs ?b. ?b geo:ontology#inCountry geo:countries/#US. }
Q7: select distinct ?a ?b ?lat ?long where { ?a dbpedia:spouse ?b. ?a dbpedia:wikilink dbpediares:actor. ?b dbpedia:wikilink dbpediares:actor. ?a dbpedia:placeOfBirth ?c. ?b dbpedia:placeOfBirth ?c. ?c owl:sameAs ?c2. ?c2 pos:lat ?lat. ?c2 pos:long ?long. }
Q8: select ?a ?y where { ?a a <http://dbpedia.org/class/yago/Politician110451263>. ?a dbpedia:years ?y. ?a <http://xmlns.com/foaf/0.1/name> ?n. ?b [] ?n. ?b <http://purl.org/dc/elements/1.1/subject> "Blackwater". }

UniProt Dataset

prefix uni: <http://purl.uniprot.org/core/>,
uniprot: <http://purl.uniprot.org/>,
schema: <http://www.w3.org/2000/01/rdf-schema#>,
file: <file:///uniprot.rdf#_>
Q1: select ?protein ?name where { ?protein a uni:Protein; uni:encodedBy [uni:name "CRB"]; uni:name ?name }
Q2: select ?a ?vo where { ?a uni:mnemonic ?vo. ?a uni:replacedBy uniprot:uniprot/P62965. ?a a uni:Protein. ?a uni:modified "1990-11-01". ?a uni:replacedBy uniprot:uniprot/P62966. ?b uni:modified "2005-08-30". ?b uni:replacedBy uniprot:uniprot/P62964. ?b uni:reviewed "false". ?b uni:obsolete "true". ?b a uni:Protein. ?a uni:replacedBy ?ab. ?ab [] ?b. ?ab uni:classifiedWith uniprot:keywords/845. }
Q3: select ?a ?vo where { ?a schema:seeAlso ?vo. ?a uni:annotation file:7A64A6. ?a uni:classifiedWith uniprot:keywords/67. ?a uni:annotation file:7A649B. ?a uni:annotation file:7A64AF. ?a schema:seeAlso uniprot:embl-cds/AAN81952.1. ?b uni:reviewed "true". ?b schema:seeAlso uniprot:geneid/1025922. ?b schema:seeAlso uniprot:smr/P0A7A1. ?b schema:seeAlso uniprot:embl-cds/AAP18215.1. ?a uni:replaces ?ab. ?ab uni:replacedBy ?b. }
Q4: select ?a ?vo ?ab where { ?a uni:modified ?vo. ?a uni:obsolete "true". ?a a uni:Protein. ?a uni:mnemonic "Q5RL09_MOUSE". ?a uni:replacedBy ?c. ?b uni:mnemonic "Q8C625_MOUSE". ?b uni:replacedBy ?c. ?b uni:obsolete "true". ?a [] ?ab. ?b [] ?ab. ?ab uni:classifiedWith uniprot:go/0006468. }
Q5: select ?a ?b ?vo where { ?a uni:mnemonic ?vo. ?a uni:reviewed "false". ?a uni:replacedBy ?b. ?b uni:mnemonic []. ?b uni:reviewed "false". ?b uni:replacedBy ?c. ?c uni:mnemonic []. ?c uni:reviewed "true". }
Q6: select ?a ?vo where { ?a uni:encodedBy ?vo. ?a schema:seeAlso uniprot:refseq/NP_346136.1. ?a schema:seeAlso uniprot:tigr/SP_1698. ?a uni:memberOf uniprot:uniref/UniRef100_Q04J74. ?a schema:seeAlso uniprot:pfam/PF00842. ?a schema:seeAlso uniprot:prints/PR00992. ?b uni:annotation file:58FFF. ?b uni:sequence uniprot:isoforms/P0A2W9-1. ?b uni:replaces uniprot:uniprot/Q54899. ?b uni:modified "2008-07-22". ?b a uni:Protein. ?a uni:replaces ?ab. ?ab uni:replacedBy ?b. }
Q7: select ?a ?vo where { ?a uni:modified ?vo. ?a uni:reviewed "false". ?a uni:mnemonic "REST_CHICK". ?a a uni:Protein. ?a uni:obsolete "true". ?a uni:replacedBy uniprot:uniprot/O42184. ?b uni:mnemonic "REST_CHICK". ?b uni:modified "1999-07-15". ?b a uni:Protein. ?b uni:replacedBy uniprot:uniprot/O42184. ?b uni:mnemonic "RSN_CHICK". ?a uni:replacedBy ?ab. ?ab uni:replaces ?b. }
Q8: select ?a ?vo where { ?a uni:annotation ?vo. ?a schema:seeAlso uniprot:interpro/IPR000842. ?a uni:annotation file:540A71. ?a schema:seeAlso uniprot:geneid/945772. ?a uni:annotation file:540A7D. ?a uni:citation uniprot:citations/9298646. ?b uni:obsolete "true". ?b uni:replacedBy uniprot:uniprot/P0A718. ?b uni:reviewed "true". ?b uni:mnemonic "KPRS_ECOLI". ?b a uni:Protein. ?a uni:replaces ?ab. ?ab uni:replacedBy ?b. }