Chapter 6
Physical Optimization

Abstract Different algorithms exist to compute the result of a logical operator like AND, OPT, or SORT. A *physical operator* implements one of the algorithms to compute the result of a logical operator. The different physical operators sometimes have different constraints on the input data like that the input data must be sorted, or are faster than others for special types of input data, for example, when the input data fit into main memory. The context of an operator can be described by the estimations of properties of its input data. For each (logical) operator in the operatorgraph, *physical optimization* aims to choose the physical operator with the best estimated execution times in the operator's context.

As well as describing the physical operators, we in this chapter present our new approaches to efficient RDF data management and join optimization for small datasets and for large-scale datasets with over one billion triples.

For small datasets, where the data can be indexed in main memory, in-memory indices can significantly speed up query processing because (after loading the data) no disk accesses need to be done for query processing. B$^+$-trees are optimized for disk indices of large-scale datasets, as they are optimized for blockwise sequential accesses of disks. For main-memory indices, hash indices are preferable as an index access can be done in constant time, as only a hash function must be applied to the key to retrieve the (main memory) address of the indexed element. Therefore, we use hash indices to manage small RDF datasets. Based on the triple nature of RDF data, we create seven hash indices in order to retrieve in-memory RDF data quickly. On the basis of the SPARQL-specific properties and the seven indices, we develop a new, efficient approach to computing join by dynamically restricting triple patterns. A performance evaluation demonstrates that the new approach outperforms other state-of-the-art in-memory databases.

Since the Semantic Web datasets are becoming increasingly large, developing efficient techniques to speeding up querying large-scale Semantic Web data is a key issue for Semantic Web applications. When data are already sorted, from relational database research, merge joins are known to be the fastest join algorithms on large-scale data. Therefore, recent approaches focus on the presorting of Semantic Web data during index construction, and thus the fast merge join can be used without a sorting phase at runtime for some joins. When data for succeeding joins become unsorted, the hash join is typically used. In this chapter, we propose a sorting

numbering scheme for large RDF datasets, based on which we can fast sort any intermediate and final querying results. Applying our sorting numbering scheme, all joins can be computed using the merge join with a fast sorting phase. Besides being a significant benefit to merge joins, our fast sorting technique can also remarkably speed up the elimination of duplicates. Our experiments show that a merge join using our fast sorting technique outperforms greatly the hash join and that our sorting numbering scheme integrated into any index approaches significantly speeds up querying large-scale Semantic Web data.

6.1 Motivation

Semantic Web ontologies and RDF knowledge bases are becoming increasingly large. The examples of large RDF data with millions and even billions of facts include the UniProt comprehensive catalogue of protein sequence, function, and annotation data (Swiss Institute of Bioinformatics 2009), the RDF data extracted from Wikipedia (Auer et al. 2007), the Princeton University's WordNet (Assem et al. 2006), and the Billion Triples Challenge (Semantic web challenge 2009). Examples of other RDF data include RSS 1.0 (Beged-Dove et al. 2001) and FOAF (Brickley and Miller 2007). Therefore, an important research task is developing efficient approaches to processing SPARQL queries over very large RDF data.

An amount of work (e.g., Chong et al. 2005; Guha 2010; Harris and Gibbins 2003; Pan and Heflin 2003; Volz et al. 2003; Wilkinson 2006) maps the RDF data format to the format of relational databases and SPARQL queries to SQL queries, thus leveraging the proved database technologies. However, the relational optimizations do not specialize on the data model of RDF triples and the usage of SPARQL triple patterns. Furthermore, this kind of approach also fails to handle very large RDF databases (see Abadi et al. 2007; Neumann and Weikum 2008, 2009; Weiss et al. 2008).

As well as the common and similar properties between the RDF data and the relational tables, and between SPARQL and SQL, RDF and SPARQL also have their own properties. Relational optimization techniques do not specialize on the data model of triples and the usage of triple patterns in query languages. These RDF- and SPARQL-specific features have attracted the attention and interests of researchers to develop new optimization techniques, for example, Abadi et al. (2007) and its generalization in Weiss et al. (2008) and Neumann et al. (2008). Weiss et al. (2008) and Neumann and Weikum (2008) use six indices according to the six possibilities SPO, SOP, PSO, POS, OSP, and OPS to order RDF triples. For example, the SPO collation order regards the subject (S) of an RDF triple as primary order criterion, the predicate (P) as secondary, and the object (O) as tertiary order criterion. However, (Weiss et al. 2008; Neumann and Weikum 2008) still apply conventional relational merge join algorithms to compute the joins of triple patterns. These contributions do not fully exploit RDF- and SPARQL-specific properties for optimizations of, for example, in-memory join computation, which are also studied in this work. The approaches (Abadi et al. 2007; Neumann and Weikum 2008, 2009; Weiss et al. 2008) avoid costly self-joins in one large triple table and can handle quite large-scale Semantic Web data. For the first several joins, the fast merge joins can be directly applied on already sorted data. However, in general, not all joins can be computed with the merge join algorithm without requiring extra sorting phases at runtime. This happens already for nonbushy queries with only three triple patterns, for example, (?a < origin > <DLC> .), (?a < records > ?c .) and (?c < type > ?b .). When data become unsorted for succeeding joins, hash joins are used in these approaches. From the relational database research, hash joins are known to be the fastest approach to computing joins, which do not require sorted input data. Hash joins are very efficient and cheap if at least one of two operands of the join can fit into main memory. However, when querying the very large Semantic Web databases, one cannot assume that the data for hash joins can always or even often fit into main memory.

In this chapter, we suggest a sorting numbering scheme for efficiently querying large-scale Semantic Web databases. On the basis of our sorting numbering scheme, we develop a fast sorting approach. When the data for succeeding joins become unsorted and are too large to fit into memory, using the merge joins with our fast sorting technique is more efficient than using a hash join on the unsorted data. Furthermore, our fast sorting technique is of great benefit to the queries, which require duplicate elimination. The sorting numbering scheme can be integrated into any index approaches like the ones of (Neumann and Weikum 2008, 2009; Weiss et al. 2008) to speed up querying the very large Semantic Web databases.

Overall, the contributions of our work to large-scale datasets include as follows:

- A sorting numbering scheme based on the RDF- and SPARQL-specific properties for managing and efficiently querying large-scale Semantic Web databases
- A fast sorting technique based on the sorting numbering scheme for computing any joins using the fast merge join approach, and for eliminating duplicates efficiently
- Integration of our sorting numbering scheme into index approaches for speeding up querying very large Semantic Web data
- The idea of using integer identifiers of RDF terms as presorting numbers for fast sorting intermediate and final results of queries, such that our approach of sorting numbering can efficiently support updates and does not need any additional storage space
- A concept-proof prototype, including the implementations of our sorting numbering schemes and fast sorting algorithms, reimplementations of several existing approaches, and integration of our approach into these existing approaches, in order to compare different approaches
- A performance analysis, which demonstrates that the application of our fast sorting technique significantly speeds up the join computation and duplicate elimination when querying large-scale Semantic Web databases

Besides increasingly larger datasets, the main memory sizes of typical computer configurations increase continually. Therefore, more and more datasets used in real-world applications can be managed completely in main memory, and thus also in-memory databases become increasingly important. Just applying optimizations for large-scale datasets in in-memory databases lead to suboptimal query processing, that is, special optimizations using the elements in main memory that can be directly addressed away from constraints of sequential disk accesses can boost query evaluation. Our approach for in-memory databases joins triple patterns by dynamically restricting triple patterns, that is, joining one triple pattern to the solution of the previous triple pattern or of the join of previous triple patterns. In order to compute the join of two triple patterns, we first compute one triple pattern and then use the resultant data to replace the corresponding variables in another triple pattern. In this way, the join computation only involves retrieving from the given RDF data. Therefore, we only need to create seven hash indices (S, SP, SPO, SO, P, PO, and O using keys on the subject (S), predicate (P), and/or object (O)) on the original RDF data in order to fast retrieve data specified by any triple pattern. In comparison with the indices described in (Weiss et al. 2008; Neumann and Weikum 2008), where accessing the index requires time-consuming searches in B^+-trees and sorted lists, using our indices, we can access the result of any triple pattern with *one* index access in main memory.

The contributions of this chapter for in-memory databases include as follows:

- An approach to manage and access RDF data efficiently using seven indices
- An approach to efficiently compute joins for in-memory database engines
- An experimental evaluation, which shows that our approach is faster than in-memory adaptations of disk-based approaches (e.g., Weiss et al. 2008) and other existing state-of-the-art in-memory engines for SPARQL processing

6.2 Related Work

Several index approaches are developed to manage RDF data for efficient query processing. Abadi et al. (2007) suggest a vertical partitioning approach to storing RDF data. In this scheme, the RDF triples are stored in two-column tables, and each table manages one property and contains a subject and an object column. Each table is sorted by subject, and thus particular subjects can be accessed quickly, such that fast merge joins can be applied while joins are processed on the subject. Abadi et al. (2007) employ a column-oriented DBMS (e.g., Stonebraker et al. 2005) to manage these property tables in order to leverage its benefits of compressibility and performance. The property tables in (Abadi et al. 2007) have considerable advantages for the SPARQL triple patterns, where the predicate is a RDF term but not a variable. However, this approach does not sufficiently support the efficient processing of the

(continued)

triple patterns, where the predicate is a variable (see Neumann and Weikum 2008, 2009; Weiss et al. 2008).

For efficient processing of more general queries, *Hexastore* (Weiss et al. 2008) and *RDF3X* (Neumann and Weikum 2008, 2009) use six indices corresponding to the six collation orders SPO, SOP, PSO, POS, OSP, and OPS to manage RDF triples. Depending on which positions in a triple pattern contain RDF terms (e.g., the subject and the object), one of the indices (e.g., SOP) is used to efficiently retrieve the data by using a prefix search. Using these collation orders, some joins can be computed using the fast merge join approach over sorted data. In comparison, our approach optimizes the evaluation of remaining joins, when data become unsorted.

For every collation order, for example, SPO, (Weiss et al. 2008) proposes to associate a subject key s_i to a sorted vector of n_i property keys, $\{p^i{}_1, p^i{}_2, \ldots, p^i{}_{ni}\}$. Each property key $p^i{}_j$ is, in its turn, linked to an associated sorted list of $k_{i,j}$ object keys. These object lists are shared in indices for corresponding collation orders; for example, the object lists for SPO are also shared by the index for PSO. RDF3X (Neumann and Weikum 2008, 2009) uses a more elegant solution to store data sorted according to the six collation orders: employing just B$^+$-trees and prefix searches and thus gaining a simpler and faster index structure than (Weiss et al. 2008).

RDF3X (Neumann and Weikum 2008, 2009) and Hexastore (Weiss et al. 2008) use sophisticated data structures to compress their index structures. RDF3X (Neumann and Weikum 2008, 2009) also supports additional special aggregated indices for fast processing of a special kind of queries. However, important features of SPARQL like data types are currently not supported by RDF3X. Neumann and Weikum (2008, 2009) also describe some cardinality estimation techniques of SPARQL join results and a sophisticated plan generator. Furthermore, (Neumann and Weikum 2009) introduce the Sideways Information Passing (SIP) strategy for optimizing query processing.

If meta nodes are ignored, the approaches described in (Harth and Decker 2005) and in (Groppe et al. 2009a) use the seven indices S, SP, SPO, SO, P, PO, and O for retrieving the result of a triple pattern within one index access. When B$^+$-trees and prefix searches are used, then the number of indices can be reduced to four (accesses to the S and SP indices can be answered by a prefix search in the SPO index; accesses to the P index by the PO index). While (Harth and Decker 2005) show that four B$^+$-tree indices work well for disk-based Semantic Web applications, (Groppe et al. 2009a) demonstrate that seven hash indices perform well for in-memory SPARQL engines. (Groppe et al. 2009a) first compute one triple pattern and then use the resultant values to replace the corresponding variables in the following triple patterns. In this way, the join computation only involves the retrieving from the given RDF data, and thus the 7 indices on the original data are enough, and neither sorting nor hashing of the solutions are needed. However, this join

(continued)

approach becomes inefficient when the data do not fit into memory any more because it causes extra disk accesses.

The SPARQL-engine Kowari (Wood et al. 2005) envisions statement-based queries. A statement-based query lacks one or two parts of a triple, and its answer is a set of resources that complement the missing parts. If meta nodes are ignored, the number of required indices of the Kowari solution is 3, defined by the three cyclic orderings SPO, POS, and OSP. Since the other three indices SOP, PSO, and OPS are missing, Kowari cannot efficiently process general queries.

Other SPARQL engines such as Jena (see Wilkinson 2006), 3store (Harris and Gibbins 2003), DLDB (Pan and Heflin 2003), KAON (Volz et al. 2003), Oracle (Chong et al. 2005), and rdfDB (Guha 2010) utilize a traditional relational database or Berkeley DB as their underlying persistent data store (Matono et al. 2005). Most of these SPARQL engines store RDF triples directly in relational or hash tables, and thus simple statement-based queries can be satisfactorily processed by such systems. However, the conventional approaches are not efficient for more complex queries (Matono et al. 2005) involving, for example, multiple filtering steps.

Some systems such as Jena (Wilkinson 2006) attempt to create relational-like property tables out of RDF data, and these tables gather together information about multiple properties over a list of subjects. Still, these schemes do not perform well for queries that need to combine data from several tables (Abadi et al. 2007). A relational-like structure on RDF data results in a sparse representation with many NULL values in the formed property tables. Handling such sparse tables, as opposed to denser ones, requires a significant computational overhead (Abadi et al. 2007).

Angles and Gutiérrez (2005) and Hayes and Gutiérrez (2004) deal with the possibility of storing RDF data as a graph, but do not sufficiently address the scalability questions either. Matono et al. (2003) and Kim et al. (2005) propose a path-based approach for managing RDF data and store subgraphs into distinct relational tables. These systems do not provide the scalability necessary for querying large-scale data. As well as (Neumann and Weikum 2008, 2009; Bernstein et al. 2007) also use selectivity estimation techniques for query optimization. However, this approach focuses on small RDF graphs, which fit into main memory, and thus also faces scaling problems. Liarou et al. (2007) investigate SPARQL extensions for handling continuous queries.

6.3 Indexing

An index stores key-value pairs in such a datastructure that a value can be efficiently retrieved for a given key. Query processing is often speeded up by storing the input data in indices and by accessing the results of query subexpressions using these indices. In the following subsections, we describe in-memory and disk-based indices for the processing of SPARQL queries.

6.3.1 Building In-Memory Indices

In this section, we first focus on in-memory indices, where the input data fits into main memory. In main memory, spread elements can be addressed directly via, for example, hash tables and we do not need to consider *sequential* disk-based accesses. We describe the usage of seven indices in order to retrieve the result of any triple pattern with one index access. RDF is a set of triples $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$. Triple patterns of SPARQL queries contain either constant values or variables, for example, *(?article rdf:type bench:Article)*. In order to access the result of any triple pattern (*s p o*) efficiently, we use the sequence of constant values in triple patterns as keys to create indices. Therefore, for any RDF data and SPARQL queries, we only need to construct seven indices using *s*, *p*, *o*, *sp*, *so*, *po*, and *spo* as keys, respectively. The seven indices are enough for any triple pattern in SPARQL queries to retrieve RDF data quickly. As our experiments show, constructing and managing seven indices is also practical even for large RDF data, which still fit into main memory.

Example 1 (Index with sp as key). The following RDF graph is the input RDF graph:

D={ *(:article1, rdf:type, bench:Article),*
(:article1, rdf:type, bench:JournalArticle),
(:article1, dc:creator, :person1),
(:inproc1, rdf:type, bench:Inproceedings) }

The following table presents the index with *sp* as key and the triples for the corresponding key:

Key *sp*	Triples
:article1 rdf:type	{*(:article1, rdf:type, bench:Article),* *(:article1,rdf:type,bench:JournalArticle)*}
:article1 dc:creator	{*(:article1, dc:creator, :person1)*}
:inproc1 rdf:type	{*(:inproc1,rdf:type,bench:Inproceedings)*}

If we use indices, which allow a prefix search like supported by B$^+$-trees but not by hash tables, then we can reduce the number of indices to four, as a *s*-index access and a *sp*-index access can be answered by a prefix search on the *spo* index, and a *p*-index access by a prefix search on the *po* index. While B$^+$-trees or variants of B$^+$-trees seem to be the best choice for indices of disk-based approaches (see Weiss et al. 2008; Neumann and Weikum 2008), our experimental evaluation shows that hash tables are the best choice for in-memory databases, as one index access can be done in constant time.

The seven indices are generated only once when reading the input data and can be afterward used for the evaluation of any SPARQL query during the lifetime of the in-memory database engine process. As we will see later on, for joining the results of any triple patterns, we need only these seven indices.

6.3.2 Building Disk-Based Indices

Most Semantic Web query evaluators use B$^+$-trees to store large-scale RDF datasets on disks (e.g., Weiss et al. 2008; Neumann and Weikum 2008, 2009). B$^+$-trees can be built very efficiently from a sorted list of data by avoiding expensive node splitting. Among a number of sorting algorithms, *merge sort* scales well for very large data and performs especially well for external sorting. Therefore, our SPARQL engine uses the merge sort technique to sort data for constructing indices efficiently. In the merge sort algorithm, our SPARQL engine uses a sort&merge-heap (Groppe and Groppe 2010) with replacement selection (Friend 1956) in order to increase the size of the initial runs. We now describe the internals of our SPARQL engine for large-scale datasets. Note that the internals of other SPARQL engines may slightly differ, but the main principles are the same. Three different types of indices are created and maintained in our SPARQL engine: dictionary indices, evaluation indices, and histogram indices.

6.3.2.1 Dictionary Indices

Semantic Web query evaluators such as RDF3X (Neumann and Weikum 2008, 2009) and Hexastore (Weiss et al. 2008) as well as our SPARQL engine use dictionary indices to map RDF terms into integer ids. One advantage of ids is lower space requirements in the evaluation indices storing the input RDF triples as an integer is stored instead of a possibly large string. Furthermore, more space can be saved by avoiding storing leading zero bytes and using difference encoding, which we will explain in detail in the section about the evaluation indices. Solutions using ids consume less space such that the memory footprint is smaller and/or more solutions can be processed without swapping to hard disks and thus improving the performance. Using ids have disadvantages in seldom cases when operations like sorting or relational comparisons like $<$, $<=$, $>=$, and $>$ require the RDF terms instead of the ids causing high costs for large intermediate results because of the materializations of the RDF terms. Furthermore, displaying the final query result has also high costs whenever the query result is large. However, the advantages typically outweigh the disadvantages of using ids for large-scale datasets.

One dictionary index maps RDF terms into integer ids; one translates integer ids back into RDF terms. The dictionary indices do not fit into main memory for large-scale datasets such that our SPARQL engine uses B$^+$-trees for the dictionary indices. When storing RDF terms in the dictionaries, we use difference encoding in order to save storage space: we determine common left substrings of the current and previously stored strings and store only the length of the common left substring together with the remaining right substring of the current string. Furthermore, after transforming id values of query results back to RDF terms, we cache the RDF terms with their ids together in order to avoid multiple materializations. We use the strategy of least recently used (LRU) caches for the accesses to the B$^+$-tree nodes in order to further improve the performance of these materializations.

The dictionary indices are used to transform RDF triples into id triples, which are consisting of ids instead of RDF terms and are then stored in the evaluation indices: Id triples are obtained from RDF triples by using the dictionary index from strings to ids and mapping the RDF terms of the subject, predicate, and object from the triples to their ids. If many RDF triples must be transformed into id triples like when importing a large dataset into the database, then it is not efficient to query the dictionary index for every single RDF term accessing a path from the root to a leaf of the B$^+$-tree for mapping RDF terms into ids. It is more efficient to use three passes through the RDF triples, where first the subjects, then the predicates, and finally the objects are transformed into ids. In each pass, the triples are first sorted according to the component (subject, predicate, or object) to be transformed into ids. Afterward, we iterate through the sorted RDF triples and simultaneously through the sorted RDF terms of the dictionary index. We read RDF terms from the dictionary index until we have found the entry for the current component of the RDF triple and replace this component with the corresponding id. Afterward, we read the next RDF triple and proceed as before. In this way, we only need one pass through the sorted RDF triples to be imported and the dictionary index to transform one component of the RDF triples into their ids. Furthermore, we can also use SIP strategies as we have discussed when introducing B$^+$-trees to increase the performance during iterating through the dictionary index. Afterward, we sort our (partly) transformed triples to be imported according to the next component and replace their RDF terms with the corresponding ids like before until all components are transformed into ids.

6.3.2.2 Evaluation Indices

These indices are used for the evaluation of SPARQL queries. They are constructed from sorted id triples according to the six collation orders SPO, SOP, PSO, POS, OSP, and OPS of RDF. Large-scale datasets are hard to manage without compression. Using (integer) ids instead of RDF terms already compresses the indices much. We can compress such indices further by storing only the different components of a triple in comparison to the last stored triple. For example, assuming the collation order SPO and a last stored triple (`<a>`, `<b>`, `<c>`), we only need to store the object `<d>` for a triple (`<a>`, `<b>`, `<d>`). Of course, we have to additionally store two bits for distinguishing if all three components of a triple must be stored, two components, or only one component. An integer id of an RDF term is typically represented by four bytes. For small integer ids, some leading bytes of these four bytes are zero. We can now store two bits for distinguishing if no, one, two, or three leading bytes are zero, and only store the bytes without leading zeros. We further know that the id *i1* of the triple's object must be larger than the id *i2* of the last triple's object in our example because of the SPO collation order. Therefore, we only need to store the difference *i1–i2*, which may have more leading zero bytes. However, for the object of triples, where numerical values and language-tagged literals can occur, we must consider special RDF properties, which we explain in the following paragraph:

In RDF data, typed literals like integer values of XML Schema can have several representations, for example, `2` and `+2`. During query processing, they are treated as value-equal integer values. Therefore, they should be assigned with the same id. Otherwise, some processing, like the computation of join, might create wrong results. However, the W3C test cases (Feigenbaum 2008) show that the query results must contain the original representation. Similar remarks also apply to language-tagged literals. For example, "`Text`"`@DE` and "`Text`"`@de` are treated as equal values, but the original representation must be maintained for the final result. Therefore, we additionally store an id, which refers to the original representation in the index, additionally to the id for value-equal literals, if necessary. Different representations of equal values are only possible for typed literals and language-tagged literals, which can only occur in objects of triples. Therefore, we only need to store an additional id for the objects if the original representation differs from the indexed one. In comparison, the original RDF3X prototype (Neumann and Weikum 2008, 2009) does not support data types, and considers, for example, the same integer values `2` and `+2` to be two different literals.

Furthermore, every node in our B$^+$-trees has an integer id, which allows us to highly compress references to B$^+$-tree nodes as well. Figures 6.1 and 6.2 present the stored bits and bytes for a leaf in a B$^+$-tree, when an id triple is stored (see Fig. 6.1) or a reference to the next B$^+$-tree node (see Fig. 6.2). Figure 6.3 presents the stored bits and bytes for a B$^+$-tree interior node, when a key in form of an id triple with a reference to its B$^+$-tree child node is stored. The last entry of a B$^+$-tree interior node contains only a reference to a B$^+$-tree child node. For this last entry, we use the same stored bits and bytes as presented in Fig. 6.2 for a B$^+$-tree leaf: We just store the reference to the B$^+$-tree child node instead of the reference to the next B$^+$-tree leaf.

The original RDF3X prototype (Neumann and Weikum 2008, 2009) supports additional special aggregated indices for fast processing a special kind of queries. For example, the two triples (`<a>`, `<b>`, `<C>`) and (`<a>`, `<b>`, `<D>`) are aggregated as (`<a>`, `<b>`, `2`) in the aggregated SP index in order to represent two triples with a common subject `<a>` and a common predicate `<b>` (but with different objects). The SP index can be used to fast retrieve the result of a triple pattern `<a> ?p ?o`, if the variable `?o` is neither further used nor occurs in the final result. Considering that such cases occur seldom in real world and the additional costs for maintaining the aggregated indices, aggregated indices are not supported by our SPARQL engines.

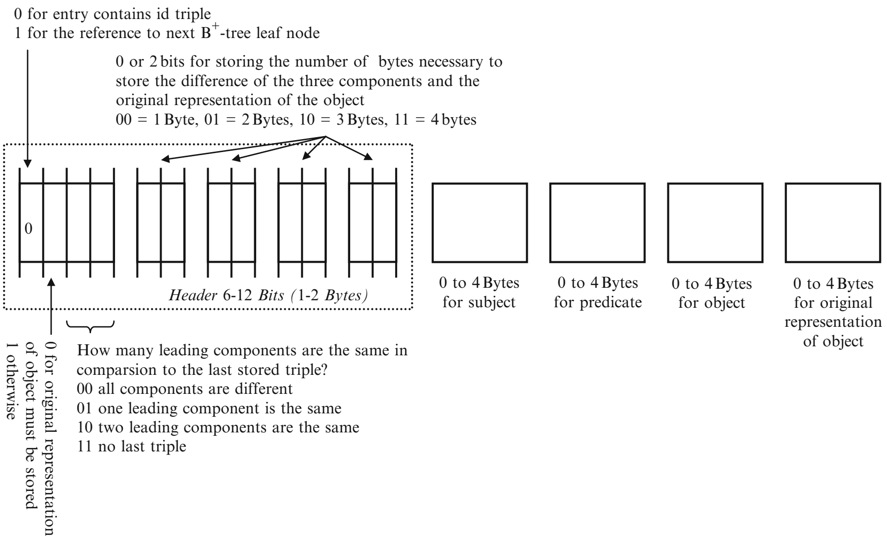
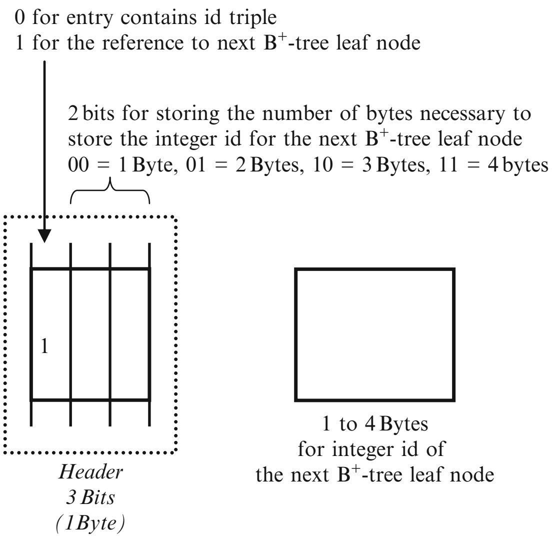

0 for entry contains id triple
1 for the reference to next B+-tree leaf node
0 or 2 bits for storing the number of bytes necessary to store the difference of the three components and the original representation of the object
00 = 1 Byte, 01 = 2 Bytes, 10 = 3 Bytes, 11 = 4 bytes
0
Header 6-12 Bits (1-2 Bytes)
0 to 4 Bytes for subject
0 to 4 Bytes for predicate
0 to 4 Bytes for object
0 to 4 Bytes for original representation of object
0 for original representation of object must be stored
1 otherwise
How many leading components are the same in comparsion to the last stored triple?
00 all components are different
01 one leading component is the same
10 two leading components are the same
11 no last triple

Fig. 6.1 Entry for id triple in B$^+$-tree leaf

Fig. 6.2 Entry for integer id for next B$^+$-tree leaf node


0 for entry contains id triple
1 for the reference to next B+-tree leaf node
2 bits for storing the number of bytes necessary to store the integer id for the next B+-tree leaf node
00 = 1 Byte, 01 = 2 Bytes, 10 = 3 Bytes, 11 = 4 bytes
1
Header 3 Bits (1Byte)
1 to 4 Bytes for integer id of the next B+-tree leaf node

6.3.2.3 Histogram Indices

Our plan generator uses equi-depth histograms (Piatetsky-Shapiro and Connell 1984) for estimating the cardinality of results of triple patterns and for calculating the overall cost of a plan. A histogram is created for a triple pattern and a specific

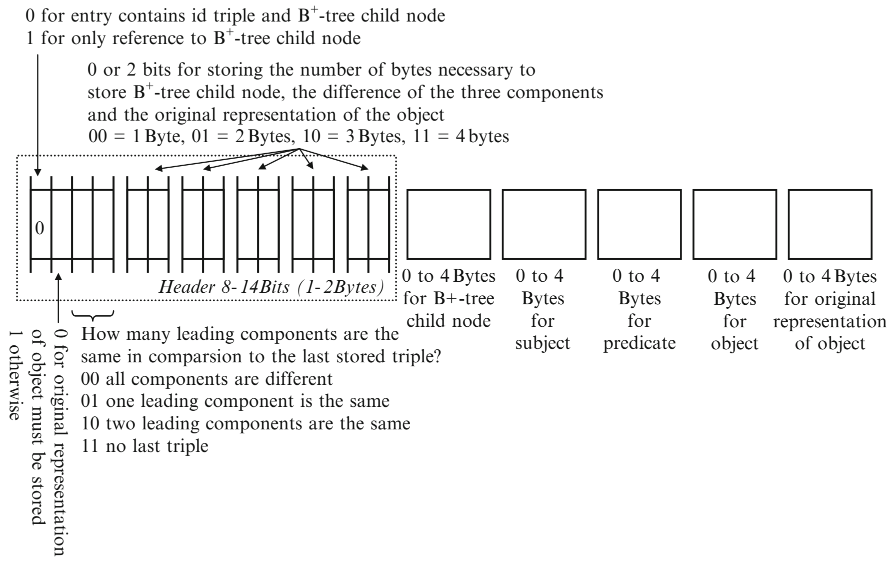

0 for entry contains id triple and B+-tree child node
1 for only reference to B+-tree child node
0 or 2 bits for storing the number of bytes necessary to store B+-tree child node, the difference of the three components and the original representation of the object
00 = 1 Byte, 01 = 2 Bytes, 10 = 3 Bytes, 11 = 4 bytes
0
Header 8-14Bits (1-2Bytes)
0 to 4 Bytes for B+-tree child node
0 to 4 Bytes for subject
0 to 4 Bytes for predicate
0 to 4 Bytes for object
0 to 4 Bytes for original representation of object
How many leading components are the same in comparsion to the last stored triple?
00 all components are different
01 one leading component is the same
10 two leading components are the same
11 no last triple
0 for original representation of object must be stored
1 otherwise

Fig. 6.3 Entry for id triple and B^+-tree child node in B^+-tree inner node

variable of it. Each interval in the histogram contains the number of the triples allocated in this interval and the numbers of distinct values. In order to speed up the generation of equi-depth histograms, we use a special B^+-tree for each collation order. In each inner node of this special B^+-tree, we store the number of triples, the number of distinct subjects, predicates, and objects, and three bits. The three bits indicate whether the subject, predicate, or object of the first triple F in the subtree is different from the triple before F.

Using this special B^+-tree and especially its additional information, we can very quickly find the corresponding information related to a triple pattern by not only passing leaf nodes, but also jumping over whole subtrees. Therefore, histograms of triple patterns can be constructed very efficiently from these histogram indices. This is also shown by our experimental results. Figure 6.4 illustrates how to construct the histogram for the variable `?v` and the triple pattern `(3, ?v, ?o)` from a histogram index. Since these additional B^+-trees are only needed for efficiently computing histograms, their updates can be delayed to the times with low workload.

Once a histogram has been calculated, it is stored in a separate index and can be reused for the triple patterns with the same RDF terms and variables at the same positions, but independent from the names of variables. While histogram computations using histogram indices need some seconds for large-scale datasets like the one from the Billion Triples Challenge (Semantic web challenge 2009), reusing a histogram by retrieving it from the separate index only consumes milliseconds.

Other contributions to Semantic Web databases do not use histogram indices: (Weiss et al. 2008) do not use any histograms for query optimization, which leads to inefficient query plans. Neumann and Weikum (2008) compute histograms using

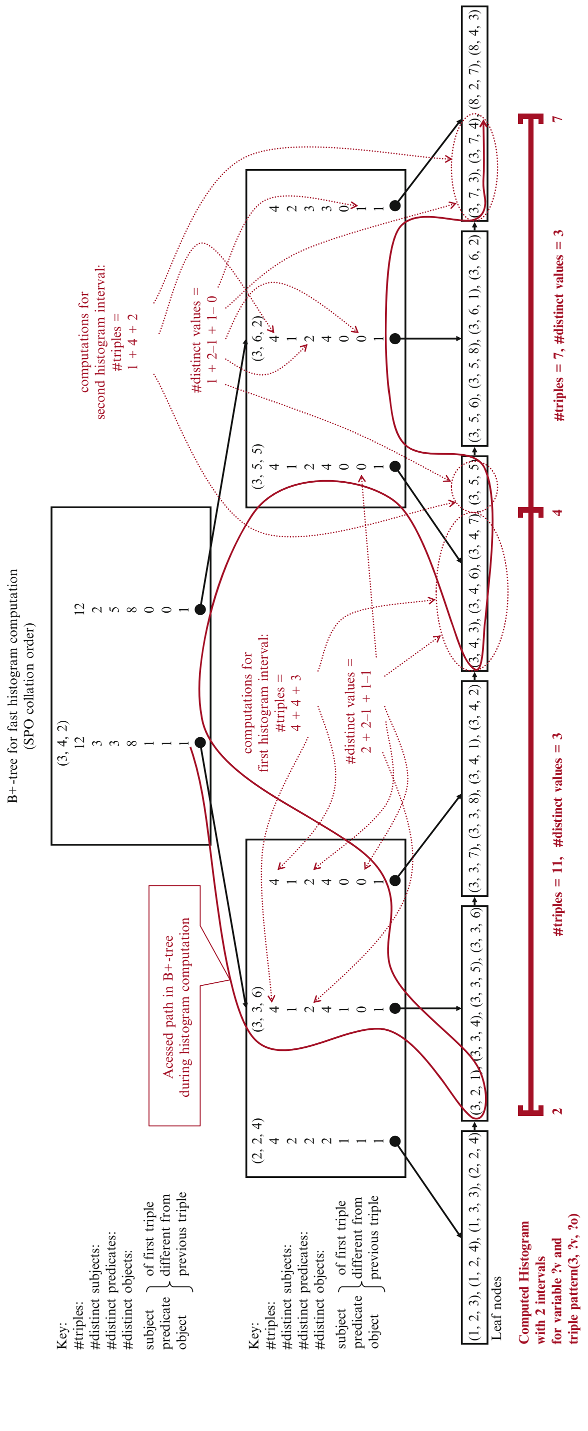


Fig. 6.4 Fast construction of the histogram for the variable ?v and the triple pattern (3, ?v, ?o) using a histogram index

the evaluation indices, which is too slow for large datasets. Neumann and Weikum (2009) precompute all possible joins facing problems with efficiency and cost estimations behind joining only two triple patterns.

6.4 Pipelining Versus Materialization

Many physical operators can determine first solutions after some and not all input data come in and can deliver these solutions early to its succeeding operators, which is called *pipelining*. For the support of pipelining, an operator has to support the iterator concept. The iterator concept requires the support of the methods *open()* for starting the computation, *next()* to retrieve the first (just after an *open()* call) as well as the next result, and *close()* to end computing and to free resources. The *open()* method of an operator calls the *open()* methods of its operands. The *next()* method gets solutions of the operator's operands by calling their *next()* methods until a solution can be determined, which is then returned. Thus, *next()* returns the solutions of an operator one by one. The *close()* method calls the *close()* methods of its operands before freeing its own resources.

The opposite *materialization strategy* first finishes the computation of an operator and determines the whole result and then proceeds to the next one. Few solutions may fit into main memory, but a large number of solutions must be stored on disk. Consequently, the costs for disk accesses slow down the performance. Contrary to the materialization strategy, pipelining does not require any materialization of intermediate results, as the solutions of operator results are computed one by one on demand of the parent's operator when calling the *next()* method avoiding huge costs for disk accesses.

6.4.1 Pipeline-Breaker

Some operators like the Sort operator cannot start the computation until all the data come in. These operators are called *pipeline-breaker*. Nevertheless, these operators can still support the iterator concept by just reading all the input data after an *open()* call, computing its result, and returning them one by one after *next()* calls. Thus, an operator does not need to know if its parent's operator is a pipeline-breaker or not and can just use the iterators for accessing its input.

6.4.2 Sideways Information Passing

SIP passes information from one operand to the other sideways in the operator-graph. For example, a merge join algorithm requires the input of its operands to be sorted. Furthermore, if a solution *A* is read from one of the operands, then solutions of the other operand are read by the merge join algorithm until a solution equal to or larger than *A* is read. In a more intelligent way, we can pass *A* to the other operand and the other operand may optimize retrieving a solution equal to or larger than *A*. For this purpose, we extend the iterator concept for SIP, such that a *next*(*lower-Limit*) method must be supported. *next*(*lowerLimit*) methods return a solution equal to or larger than *lowerLimit* for sorted results. Note that this method has been already introduced for prefix searches in B^+-trees, such that index scans retrieving the result of triple patterns are already optimized for processing *next*(*lowerLimit*) method calls. Furthermore, the information *lowerLimit* may be passed not only between the operands of an operator, but also to their operands, such that the optimization potential is enormous.

6.5 Join Algorithms

We first review the traditional join algorithms Nested-Loop, Merge, Index, and Hash Join. Afterward, we present our new approaches to join computation for the in-memory and large-scale RDF databases.

6.5.1 Nested-Loop Join

The nested-loop join is one of the simplest join algorithms and performs well for very small unsorted data. Therefore, this join algorithm is often used as part of other more complex join algorithms like the hash join.

In its simplest form, the nested-loop join contains a nested loop iterating through the solutions of its left and right operand. In each iteration, if the two solutions of the left and right operands can be joined, the joined result is returned. Otherwise, the loops proceed. Let *R* and *S* contain the solutions of its left and right operands, *joinable(r, s)* be a function to check if *r* and *s* can be joined, and *join(r, s)* computes the join between *r* and *s*. The following pseudo code presents the nested-loop join algorithm:

```
FOR EACH s IN S DO
    FOR EACH r IN R DO
        IF(joinable(r, s)) {
            OUTPUT join(r, s);
        }
```

The runtime complexity is O(|*S*|*|*R*|). If the results of both operands of a join do not have any variables in common, then every solution of one operand must be combined with every solution of the other operand; that is, the join degenerates to a Cartesian product. Therefore, the *worst case* runtime complexity is O(|*S*|*|*R*|) for any join algorithm. However, the *average* runtime complexity for typical input data is often better for other join algorithms up to linear complexity O(|*S*| + |*R*|) for merge joins, which may be O(min(|*S*|, |*R*|)) or even better when using SIP strategies.

6.5.1.1 Iterator Version

In this subsection, we describe the iterator version of the nested-loop join algorithm. We therefore present the pseudo code of the three methods *open()*, *next()*, and *close()* here as follows:

```
open()  {
      R.open();
      S.open();
      s = S.next();
}

next() {
      DO {
            r = R.next();
            IF(r == null){ // R is exhausted for the current r
                  R.close();
                  s = S.next();
                  IF(s==null){ // both R and S are exhausted!
                        return null;
                  {
                  R.open();
                  r = R.next();
            }
      } WHILE (!joinable(r, s));
      RETURN join(r, s);
}

close(){
      R.close();
      S.close();
}
```

The new main ideas and principles of the other physical operators can be understood from the materialization versions of these physical operators. Therefore and due to simplicity of presentation, we will not present the iterator version for the other physical operators and discuss only the version for the materialization strategy.

6.5.1.2 Block-Based Nested-Loop Join

Database systems usually work on pages, which are arrays of bytes with fixed length (typically 8 kb). A page contains input data, metainformation, and solutions. Pages are stored on disk and loaded into main memory (using a buffer manager) for reading and manipulating. Solutions (and data) are typically organized in pages to

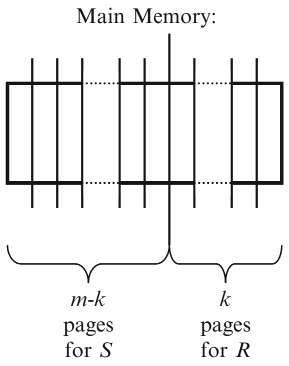


Fig. 6.5 Reserving pages for S and R in main memory

be controlled by the buffer manager. The buffer manager holds frequently accessed pages in main memory, such that the page(s) containing solutions (and data) may be never (temporarily) stored on disk if they fit completely into main memory.

The design of some physical operators pays attention to the page-oriented organization of data and optimizes the number of page accesses. These operators assume that a part of the main memory with the size for m pages is available for them.

The block-based nested-loop join reserves k pages for the result R of its first join operand and thus $m - k$ pages for the result S of its second join operand (see Fig. 6.5). According to the iterator concept, the solutions of the join are computed one by one after each call of *next()*. Thus, the join operator does not reserve any page for the result of the join. However, the operator calling *next()* of the join may reserve a page for the result of the join and may temporarily store the result on disk if necessary.

The block-based nested-loop join works as follows (see Fig. 6.6): First of all, it loads $m - k$ pages of S and k pages of R into main memory and then joins the $m - k$ pages of S with the k pages of R using, for example, the "normal" nested-loop join algorithm discussed in the previous subsections. Afterward, it loads the next k pages of R into main memory and joins these k pages of R with the already loaded $m - k$ pages of S. The process is repeated until all pages of R are joined with the first $m - k$ pages of S. Note that in the last round $\leq k$ pages are joined with the first $m - k$ pages of S. The next $m - k$ pages of S are then loaded into main memory and joined with R in the same way as the first $m - k$ pages of S. The process is repeated until all pages of R are joined one time with all pages of S, and therefore whole R is joined with whole S.

We have an exercise at the book webpage http://www.ifis.uni-luebeck.de/~groppe/SemWebDBBook/ for analyzing the number of pages accessed by this variant of the nested-loop join.

After each round joining $m - k$ pages of S, k pages of R are still in main memory, which are then replaced with the first k pages of R to be joined with the next $m - k$ pages of S. An optimization of the presented block-based nested-loop

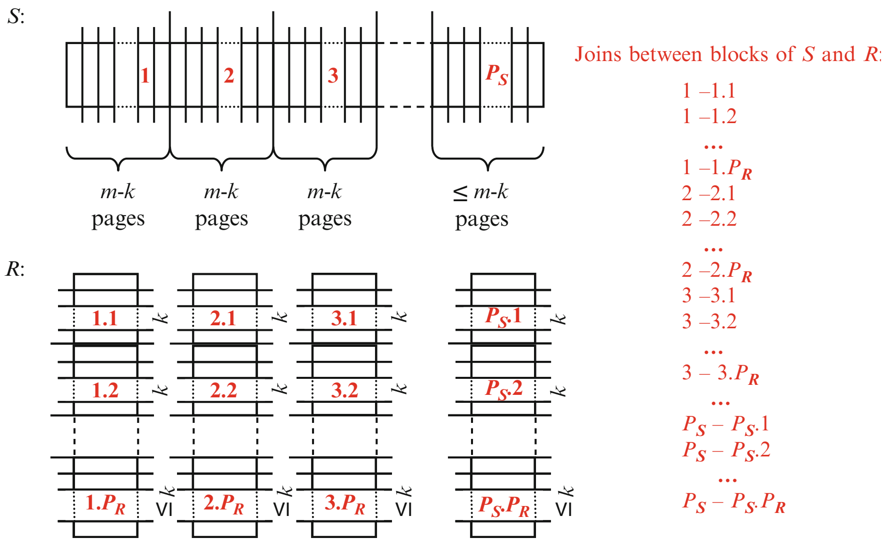


Fig. 6.6 Block-based nested-loop join

join uses these k pages of R from the last round and joins them with the next $m - k$ pages of S before this optimization proceeds joining with the first k pages of R. In this way, k pages are less loaded in each round (except of the initial round).

6.5.2 *Merge Join*

Let V_L be the set of bound variables of the left operand of a join, and V_R be the set of bound variables of the right operand. We call the variables in $V_L \cap V_R$ the *join variables*. The merge join requires the solutions of its operands sorted in the same way according to its join variables. We assume that the input is sorted in descendant order, such that the solutions of an operand are equal to or larger than the previous solutions. Note that the merge join algorithm can be easily adapted to consume sorted data in ascendant order by exchanging larger than comparisons in the pseudo code given below with smaller than comparisons. For an example of the merge join algorithm applied to the solutions of its operands, see Fig. 6.7. The merge join algorithm first reads the solutions from both operands and checks if they are joinable (i.e., the bound values of the join variables are the same). If they are *not* joinable, the next solution of the operand with the smaller values is read, because we are sure that the remaining solutions of the other operand are equal or larger values according to the sort criterion and thus cannot be joined with the smaller value. In the case that they are joinable, the merge join algorithm reads the next solutions from both operands until the bound values of the join variables differ from

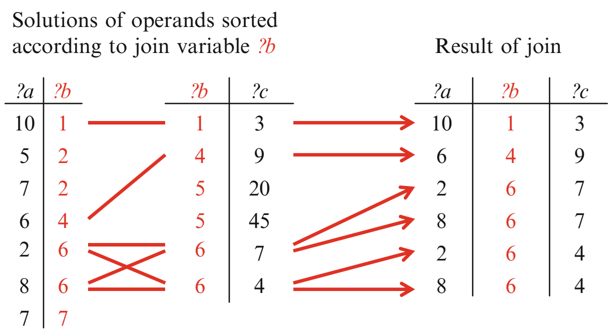


Fig. 6.7 Merge join example

the first read solution. All the read solutions with the same bound values of the join variables of both operands must now be joined and returned. The pseudo code for the materialization strategy variant of the merge join algorithm is therefore as follows:

```
S.open();
R.open();
s=S.next();
r=R.next();
WHILE(s!=null && r!=null){
    IF(s < r){
        s=S.next();
    } ELSE IF(r < s){
        r=R.next();
    } ELSE{
        s1=s;
        r1=r;
        operand1 = {};
        operand2 = {};
        WHILE(joinable(r1, s)){
            operand1 = operand1 ∪ {s};
            s=S.next();
        }
        WHILE(joinable(r, s1)){
            operand2 = operand2 ∪ {r};
            r=R.next();
        }
        FOREACHs2 IN operand1 DO
            FOR EACH r2 IN operand2 DO
                OUTPUT join(s2, r2);
    }
}
```

We will analyze the average runtime complexity in an exercise at the book webpage http://www.ifis.uni-luebeck.de/~groppe/SemWebDBBook/.

6.5.2.1 Merge Join and Sideways Information Passing

When using SIP strategies, we can use the *next*(*lowerLimit*) method to retrieve directly an element equal to or larger than *lowerLimit*. The merge join can obviously benefit from the SIP strategy, because for unequal solutions of both operands, we already know that the larger solution *lowerLimit* is the lower limit for the next solution of the other operand retrieved by calling *next*(*lowerLimit*). The following pseudo code contains the SIP version of the merge join algorithm, where SIP related code is marked with boldface:

```
S.open();
R.open();
s=S.next();
r=R.next();
WHILE(s!=null && r!=null){
    IF(s < r){
        s=S.next(r);
    } ELSE IF(r < s){
        r=R.next(s);
    } ELSE {
        s1=s;
        r1=r;
        operand1 = {};
        operand2 = {};
        WHILE(joinable(r1, s)){
            operand1 = operand1 ∪ {s};
            s=S.next();
        }
        WHILE(joinable(r, s1)){
            operand2 = operand2∪ {r};
            r=R.next();
        }
        FOR EACH s2 IN operand1 DO
            FOR EACH r2 IN operand2 DO
                OUTPUT join(s2, r2);
    }
}
```

When using the iterator versions of the operators, a SIP information may be passed through many operators until an index scan for retrieving the solutions of a triple pattern is reached. The index scan then can use the *next(lowerLimit)* method of a B^+-tree to jump over possibly huge data. This will lead to enormous reduction in the runtime especially for large-scale datasets.

6.5.3 Index Join

The index join utilizes a given index of one of its join operands to optimize join processing: The index join iterates through the solutions of one join operand and searches for relevant join partners, that is, solutions with the same bound values for the join variables, from the other join operand by using its index. An operand typically provides an index like a B$^+$-tree if it is an index scan operator for retrieving the result of a triple pattern. We express the index join in pseudo code as follows:

```
FOR EACH s IN S DO
    FOR EACH r IN index(R, s) DO
        OUTPUT join(s, r);
```

The function *index*(*R*, *s*) is a function using a given index of the operator *R* to determine relevant join partners for the solution *s*. This join algorithm is typically used whenever one operand *R* provides an index and the solutions of the other operand *S* are not sorted according to the join variables, such that a merge join cannot be used without a preceding sorting phase. Let $g(|R|)$ be the costs of an index access on data with size $|R|$, then the index join has the runtime complexity $O(|S|*g(|R|))$ under the assumption that the data *R* has the property that *index*(*R*, *s*) never returns large sets of data, but only one or few solutions. Theoretically, $g(|R|)$ is in $O(\log(|R|))$ for B$^+$-trees, but practically B$^+$-trees are typically quite flat with a height below 5 even for large data; that is, an index access is done in (nearly) constant time. Therefore, we can achieve a (nearly) linear complexity $O(|S|)$ for the index join for such kind of data, which is often the case for real-world data.

If both operands of a join are not index scans, but their solutions fit into main memory, then it is also efficient to first index the results of one operand *R* using an in-memory hash table and then apply the index join using the just created index. For indexing the solutions of *R*, a hash function over the join variables must be used, which maps bound values of the join variables in a solution to an integer, and such that the join partners for the solutions of *S* can be found within one index access. This type of index join is often also called *in-memory hash join*. The in-memory hash join is very efficient, because creating the index can be done in linear time to the results of *R*, that is, $O(|R|)$, and retrieving the join partners of *R* can be done in constant time. Under the assumption that we only have one or a few join partners for each solution of *S* in *R*, the average runtime complexity for joining is $O(|S|)$ and thus the overall average runtime complexity is $O(|R| + |S|)$.

6.5.4 Hash Join

We have presented the in-memory hash join, but we now describe the hash join for large data. Like the in-memory hash join, we use again hash functions over the join variables. The large data hash join first distributes the input data into smaller partitions using hash functions over the join variables. After the partitions become small enough, a general-purpose join algorithm, like the block-based nested-loop join, is used to join corresponding partitions of both operands.

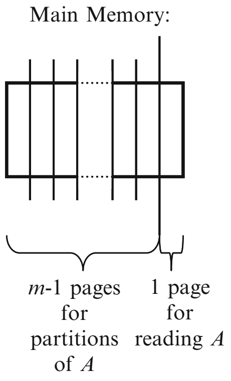


Fig. 6.8 Pages in main memory for distributing the solutions of the operand *A* during the partitioning phase

In detail, we first determine the operand *A* with fewer solutions: If the sizes of the operands' solutions are not known, then the query optimizer can estimate which operand has fewer solutions. The results of the operand *A* is then partitioned into several *partitions*, that is, the solutions with the same result using the hash function are stored in the same partition. If the hash join operator can obtain *m* pages in main memory for its computations, then the hash join uses $m - 1$ partitions stored in $m - 1$ pages in main memory, and the remaining one page for reading in the data to be distributed (see Fig. 6.8). If one page of these $m - 1$ pages for the partitions is filled up, then the page is swapped to disk. When this partition round is finished, then the (not necessarily full) pages in main memory are stored on disk.

If one partition is larger than $m - 1$ pages, then another partition round for this large partition is performed using another hash function over the join variables. The process is repeated, until the size of each partition of *A* is less than *m* pages. For the reason of finishing the partitioning phase more early, we first distribute the solutions of the operand *A*, which has fewer solutions. We now distribute the solutions of the other operand *B* into partitions using the same hash functions as for *A* and in the same way as for *A*: If a partition was distributed for *A*, then we redistribute the partition for *B* using the same hash function, too. Unlike the partitions of *A*, the partitions of *B* do not need to fit into $m - 1$ pages. In the joining phase, we first load $m - 1$ pages of *A* into main memory and join them with the pages of *B*'s corresponding partition one by one by a block-based nested-loop join. Figure 6.9 presents an overview of the different phases of the hash join approach. Two partitioning rounds are performed in Fig. 6.9.

This hash join is known to be the fastest general-purpose join algorithm whenever the input data are neither sorted nor an already existing index can be used for joining.

We will analyze the number of pages accessed by the hash join algorithm in an exercise at the book webpage http://www.ifis.uni-luebeck.de/~groppe/SemWebDBBook/.

However, the hash join algorithm as described before can run into an infinity loop whenever a partition of *A* does not become smaller during redistributions,

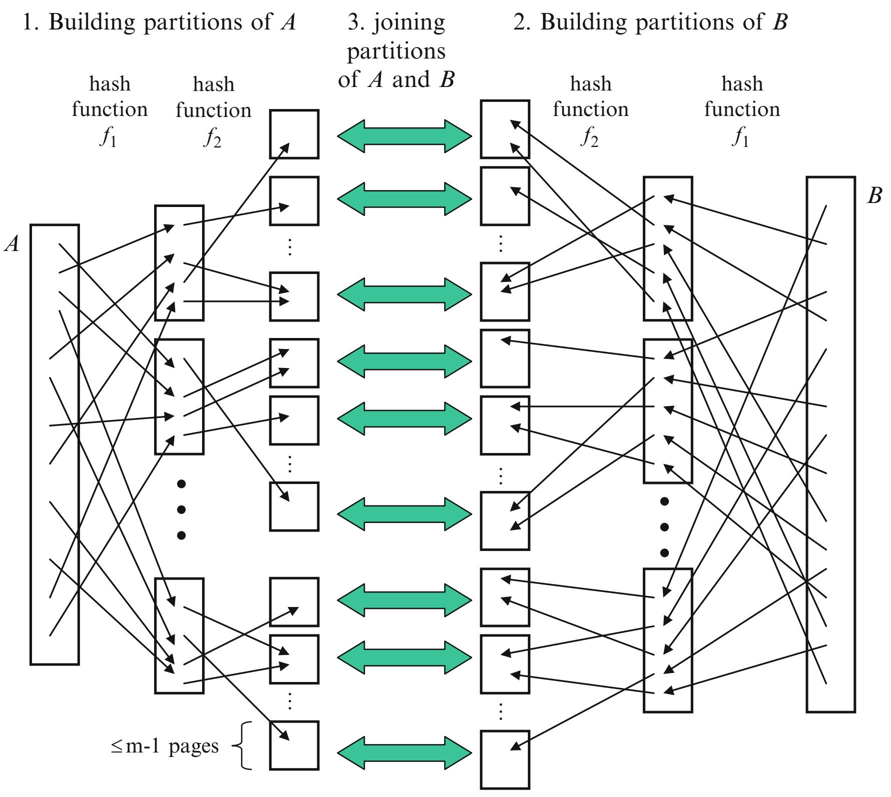

1. Building partitions of A
3. joining partitions of A and B
2. Building partitions of B
hash function f_1
hash function f_2
hash function f_2
hash function f_1
A
B
≤m-1 pages

Fig. 6.9 Phases of hash join

as the solutions of this partition contain only the same bound values for the join variables. Fortunately, other general-purpose join algorithms than the block-based nested-loop join having the solutions of one operand completely in main memory can be used for joining this partition after several redistributions do not result in smaller partitions.

6.5.4.1 Hash Join and Sideways Information Passing

The hash join first reads in all solutions of one operand A for partitioning before any solutions of the other operand B is read. Thus, we already know the join partners, that is, solutions with the same bound values for the join variables, of the solutions of B in A and could filter out irrelevant solutions of B early, which do not have any join partners in A. Using all the join partners of A for this purpose is neither practical nor scalable because of a possibly large set of join partners. However, we can use a *bloom filter* for discarding irrelevant solutions early. The bloom filter uses a bit vector and sets those bits, which are returned by a given hash function for the solutions of A. The bloom filter can be calculated during reading in the results of A. In our LUPOSDATE SPARQL engines, bloom filters are constructed in the SIP-FilterOperator (Iterator) operators. Afterward, the bloom filter can be attached to those operators in B, which bind values to join variables. These operators – typically index scans for determining the solutions of triple patterns – can now check by using the bloom filter if a corresponding join partner in B can exist. For determining the corresponding bit in the bit vector, the same hash function as when creating the bloom filter on the bound value of the join variables is applied to the solution of the triple pattern. If the bit is cleared, then we can surely discard this solution as it does not have any join partner in A. Due to the fact that hash functions may map several different values to the same integer, the bit for a solution without any join partners in A may be nevertheless set, such that we may have some *false drops*.

We have learned that a B^+-tree offers the *next(lowerLimit)* method, which returns the next value equal to or larger than *lowerLimit* for a prefix search. B^+-trees can optimize the application of the *next(lowerLimit)* method using interior nodes of the B^+-tree. However, a bloom filter does not contain the information for such *lowerLimit* parameter, but we can determine a lower limit of the distance from the current value. We assume that the hash function h was used to construct a bloom filter. For example, we discard a solution with an id 10 bound to the join variable, because in our example, $h(10)$ returns a bit position, which is cleared in the bloom filter. Furthermore, the bits $h(11)$, $h(12)$, until $h(50)$ are also cleared in the bloom filter, but the bit $h(51)$ is set in the bloom filter. Therefore, we know that a lower limit for a relevant solution not to be discarded has a value with id 51 or higher, and we can call *next(*51*)* to retrieve it. Thus, in more general, if a join variable contains a value with id *id1* and the bit $h(id1)$ is cleared in the bloom filter, then we can determine *id2*, such that the bit $h(id2)$ is set in the bloom filter and no *id3* exists, such that $id1 < id3 < id2$ holds and the bit $h(id3)$ is set in the bloom filter. Then, *id2* can be used to jump over the solutions, which do not have any join partner in A, and to retrieve directly a solution, which possibly has a join partner in A, by calling *next(id2)*.

6.6 Dynamically Restricting Triple Patterns

This join approach has been especially developed for in-memory join computation and is a variant of the index join. We assume that the seven in-memory hash indices S, SP, SPO, SO, P, PO, and O are given. A naive way to compute the join of two triple patterns is first evaluating individual triple patterns separately and then to perform a join on their results using standard join algorithms. Figure 6.10 visualizes this naive approach. However, we can leverage the properties of RDF and SPARQL in order to compute joins more efficiently.

In this section, we present a new and efficient approach to computing the join of triple patterns: we compute joins by dynamically generating *more restrictive triple patterns* (see Definition 11 in the previous chapter).

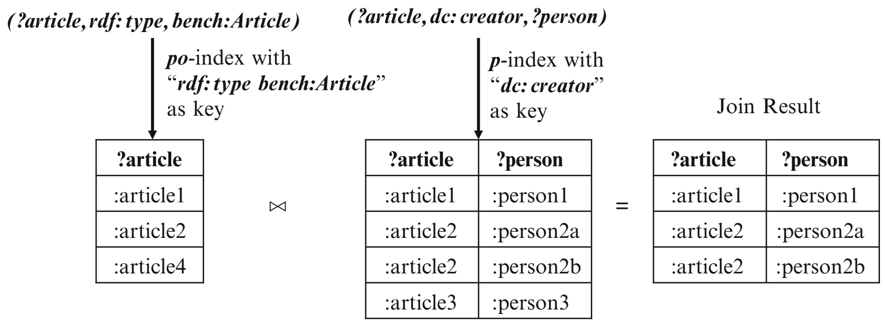


Fig. 6.10 A naive way to compute join

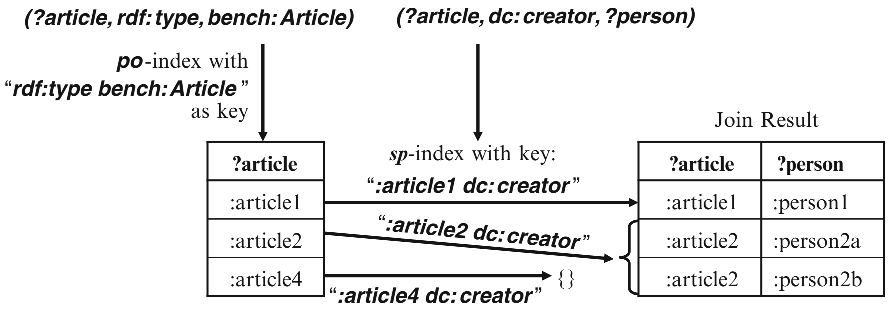


Fig. 6.11 Our approach to compute join

For example, *(?article rdf:type bench:Article)* is more restrictive than *(?article dc:creator ?person)*. It is reasonable to assume that a more restrictive pattern typically retrieves less data than the less restrictive triple patterns.

In order to compute the join of two triple patterns, we compute the first triple pattern by means of the seven indices and then use the solutions of this triple pattern to replace the corresponding variables in the next triple pattern. In this way, we get a more restrictive triple pattern. By one index access, we can get the joined result of the current solution of the first triple pattern with the solutions of the second triple pattern directly and do not need to check the join condition. Figure 6.11 visualizes our approach to compute the join. If there are more triple patterns, we use the result of the already joined triple patterns to dynamically restrict the next triple pattern.

In this way, the join computation only involves the seven in-memory indices. Note that the seven indices only have to be generated once when reading the input data.

Example 2 (Dynamically generating more restrictive triple patterns). In order to evaluate the graph pattern

```
(((  (?article rdf:type bench:Article)            AND
     (?article dc:creator ?person)                AND
     (?inproc rdf:type bench:Inproceedings)       AND
     (?inproc dc:creator ?person2)                AND
     (?person foaf:name ?name)                    AND
     (?person2 foaf:name ?name2))
     FILTER (?name=?name2))
     PROJ{?person, ?name}) DISTINCT
```

over the following input RDF graph:

```
D={  (:article1, rdf:type, bench:Article),              (t1)
     (:article2, rdf:type, bench:Article),              (t2)
     (:article4, rdf:type, bench:Article),              (t3)
     (:article1, dc:creator, :person1),                 (t4)
     (:article2, dc:creator, :person2a),                (t5)
     (:article2, dc:creator, :person2b),                (t6)
     (:article3, dc:creator, :person3),                 (t7)
     (:inproc1, rdf:type, bench:Inproceedings),         (t8)
     (:inproc1, dc:creator, :person1),                  (t9)
     (:person1, foaf:name, “Hans Fortune”)    }         (t10)
```

we first determine the solutions of the first triple pattern *(?article rdf:type bench: Article)* by using the *po* index with key “*rdf:type bench:Article*”. The triples *(t1)*, *(t2)*, and *(t3)* are returned for this index access and thus the result of the first triple pattern is $<$ {(*?article*, *:article1*)}, {(*?article*, *:article2*)}, {(*?article*, *:article4*)}$>$. We then bind each value of *?article* in the resultant set to the same variable in the second triple pattern *(?article dc:creator ?person)* and construct three more restricted triple patterns: *(article1 dc:creator ?person)*, *(article2 dc:creator ?person)*, and *(article4 dc: creator ?person)*. For each new triple pattern, we use the *sp* index with the sequence of the subject and predicate literals as key. Therefore, the join of the first and second triple pattern is $<$ {(*?article*, *:article1*), (*?person*, *:person1*)}, {(*?article*, *:article2*), (*?person*, *:person2a*)}, {(*?article*, *:article2*), (*?person*, *:person2b*)}$>$.

The third triple pattern *(?inproc rdf:type bench:Inproceedings)* does not have any common variables with the join result of the first and the second triple patterns. We evaluate this triple pattern by using the *po* index with “*rdf:type bench:Inproceedings*” as key and get the solution $<${(*?inproc*, *:inproc1*)}$>$. Then, we compute the Cartesian product of the join of the first and second triple patterns and the result of the third triple pattern and get the join result of the first three triple patterns: $<${(*?article*, *:article1*), (*?person*, *:person1*), (*?inproc*, *:inproc1*)}, {(*?article*, *:article2*), (*?person*, *:person2a*), (*?inproc*, *:inproc1*)}, {(*?article*, *:article2*), (*?person*, *:person2b*), (*?inproc*, *:inproc1*)}$>$. We analogously proceed with the remaining triple patterns. The overall result of the whole SPARQL query is $<$ {(*?person*, *:person1*), (*?name*, *“Hans Fortune”*)}$>$.

It is obvious that the given order for join computation is not optimal as the costly Cartesian product should be applied later or even avoided. Recall that the previous chapter introduced several approaches to optimizing the join order.

6.7 Sorting Numbering Scheme

We describe our specialized join approach for large-scale RDF datasets in this section. Since RDF data are modeled as a set of triples, six collation orders SPO, SOP, PSO, POS, OSP, and OPS are sufficient for sorting RDF data in any order. Therefore, recent approaches (e.g., Neumann and Weikum 2008, 2009; Weiss et al. 2008) use six indices, each for one collation order, to manage RDF data. This storage schema allows for quick and scalable general-purpose query processing.

Furthermore, these approaches also adopt the technique of dictionary encoding (Abadi et al. 2007) to map RDF terms into integer identifiers (ids). Therefore, id triples are stored in six indices rather than the original literal triples. Actually, the sort criterion in evaluation indices of the other RDF stores such as *Hexastore* (Weiss et al. 2008) and *RDF3X* (Neumann and Weikum 2008, 2009) as well as our SPARQL engines are according to these ids rather than the RDF terms. Thus, operators like merge joins also use sort criteria according to the ids rather than according to the RDF terms themselves. Only a sort operator following the SPARQL specification (Prud'hommeaux and Seaborne 2008) requires a sort criterion according to the RDF terms instead of the ids. The main purpose of using integer ids to replace RDF terms is for compressing the RDF store in these approaches.

However, we also find another important application for these integer ids: they can be used to fast sort solutions. Having the capability of fast sorting, query processing, like computation of joins and elimination of duplications, can be performed more efficiently over large datasets. When we use these ids for sorting solutions, we call them *presorting numbers*.

6.7.1 Joins Without Presorting Numbers

For example, we have a SPARQL query with the three triple patterns TP1 (`?a` $<$ `origin` $>$ `<DLC>`), TP2 (`?a` $<$ `records` $>$ `?c`), and TP3 (`?c` $<$ `type` $>$ `?b`). When evaluating the query, we can first join the triple patterns TP1 and TP2 over the variable `?a`, or TP2 and TP3 over `?c`, or TP1 and TP3. The third alternative, the join between TP1 and TP3, is actually a Cartesian product having high costs, and thus we do not consider this join ordering further. State-of-the-art database management systems use selectivity estimations in order to determine the best join ordering, which we discuss in previous sections.

In order to perform a join on `?a` between TP1 and TP2, the existing approaches, Hexastore (Weiss et al. 2008) and RDF3X (Neumann and Weikum 2008, 2009), compute TP1 (`?a` $<$ `origin` $>$ `<DLC>`) according to the POS (or OPS) collation order and get the result sorted according to `?a`; compute TP2 (`?a` $<$

`records` > `?c`) according to PSO and get the result sorted also according to `?a`. Consequently, a merge join can be directly used to compute the join of TP1 and TP2 over `?a`. The second join is computed between the results of the first join and of the remaining triple pattern TP3 (`?c` < `type` > `?b`), and thus the join variable is `?c`. Since the result of the first join is sorted according to `?a`, the second join is computed using the hash join (see Fig. 6.12a).

Likewise, a merge join can be directly used for the join of the triple patterns TP2 (`?a` < `records` > `?c`) and TP3 (`?c` < `type` > `?b`), but cannot be directly used for the succeeding join between the results of the first join and that of the remaining triple pattern TP1 (`?a` < `origin` > <DLC>).

Sorting is time-consuming. It is a well-known fact that any sorting algorithm based on comparing and exchanging values needs at least O(*N**log(*N*)) steps, where *N* is the number of elements to be sorted. Therefore, instead of sorting data for performing a merge join, these approaches typically choose a hash join algorithm. A hash join does not require sorted input data and is usually faster than a normal merge sort join requiring an additional sorting phase. The hash join is simple and efficient when at least one of two operands of it fits into main memory. If neither of the two operands of a hash join fits into memory, the disk-based hash join algorithms become expensive (see Elmasri and Navathe 2000; Garcia-Molina et al. 2002). Our experiments show that a merge join with our fast sorting phase based on the presorting numbers outperforms significantly hash joins for large data.

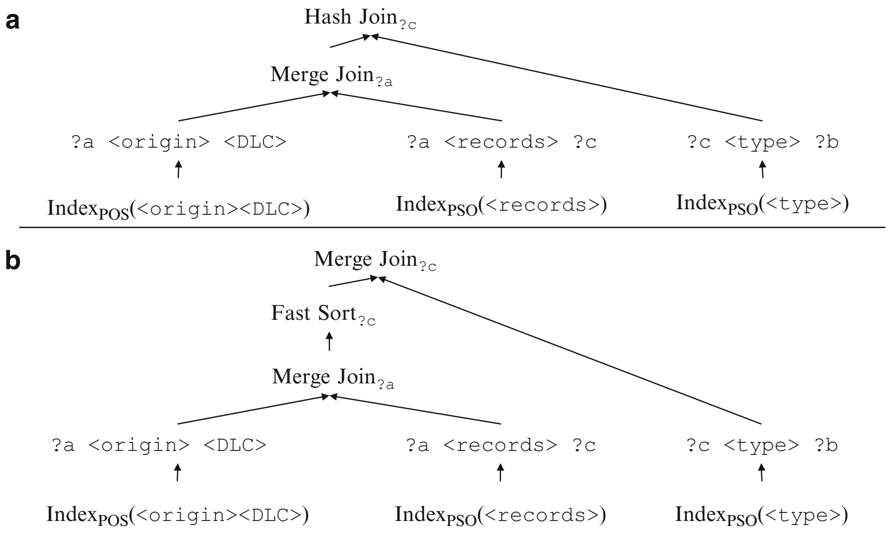


Fig. 6.12 Join computation of a nonbushy query with three triple patterns (**a**) without and (**b**) with using fast sorting

6.7.2 Joins with Presorting Numbers

We take as example the same query as in the previous section. Like RDF3X and Hexastore, we use a merge join to compute the first join between, for example, TP1 (`?a < origin > <DLC>`) and TP2 (`?a < records > ?c`) over `?a`. If we use the ids of RDF terms as presorting numbers, we can fast sort the result of this join according to `?c`. We use a variant of bucket sort (Knuth 1998) specialized to external sorting and also able to sort according to order criteria with several variables, as we will explain in later subsections. Consequently, the second join can be computed using a merge join instead of a hash join.

Figure 6.13 illustrates our fast sorting technique using the ids as presorting numbers. If the RDF data contain n different literals, then each literal can be mapped into an integer id in $[1, n]$. In order to sort solutions, we use n buckets, numbered from 1 to n. We describe in later subsections how to reduce the number of buckets.

During computing the first join between TP1 (`?a < origin > <DLC>`) and TP2 (`?a < records > ?c`), once a binding has been determined as a solution of this join, it is put into the corresponding bucket according to the id value of `?c`. Once the join computation is finished, the result in all these buckets has been sorted according to `?c` and can be retrieved by accessing their contents in the order of the buckets.

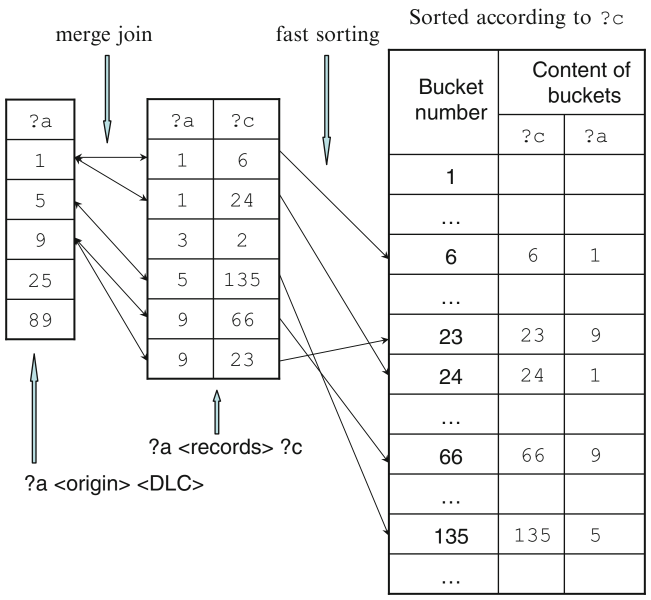


Fig. 6.13 Fast sorting according to `?c` using the ids of RDF terms as presorting numbers when computing join

The last triple pattern `?c` $<$ `type` $>$ `?b` is computed using the index PSO, and thus its result is also sorted according to `?c`. Therefore, the second join between the results of the first join and of the last triple pattern can be computed using a merge join (see Fig. 6.12b).

6.7.3 *Optimization of Fast Sorting*

Usually, the presorting numbers (ids) of solutions are located at a certain range $[m, k]$, where $1 \leq m \leq k \leq n$, rather than dispersed over the whole id space. In this case, we need only $k + 1 - m$ buckets for sorting. We can determine the range $[m, k]$ during cardinality estimation of the results of triple patterns and of joins in the logical optimization phase. In the logical optimization phase, we can therefore store the minimum m and the maximum k of the range at the operator for the corresponding triple pattern in the execution plan.

If the range of the presorting numbers is relatively small, then we can store the buckets in main memory by simply using, for example, an array to store the solutions. If the range of the presorting numbers is large, then we have to store the buckets in an external storage like a hard disk. One way is to store each bucket in a single file. However, the number of solutions stored in a bucket is typically small and often even 1. Managing a large number of buckets with little content is inefficient.

An alternative without this disadvantage for sorting solutions is to divide the whole range of the presorting numbers into m smaller ranges. We then use m merge sorts for the m smaller ranges, each of which employs a heap for replacement selection, to sort the results with large range. We describe this approach in Fig. 6.14. In this way, we use much less buckets to store all sorted results. Our experiments show that using this way to sort data is very fast. In our experiments, we have used 1,000 ranges and heaps of size 256.

However, the ids of solutions are usually not continual integers, that is, there are gaps among these ids. This might lead to a bad distribution among m smaller ranges when using equal range sizes. This can be solved by using histograms of triple patterns.

For each kind of triple patterns, an equi-depth histogram is constructed. Among other information, each interval in this histogram contains the range and the number of triples allocated in this interval. Equi-depth histograms have the property to divide the data in such a way that each interval has the same or at least similar numbers of triples. Therefore, the intervals in the equi-depth histograms can be directly used as the smaller ranges into which the whole range of the presorting numbers is divided. In this way, we can get a perfect distribution among smaller ranges.

6.7.4 *Sorting for Complex Joins*

So far, the joins we consider have only one join partner, that is, only one common variable between two triple patterns. In most cases, the joins have only one join

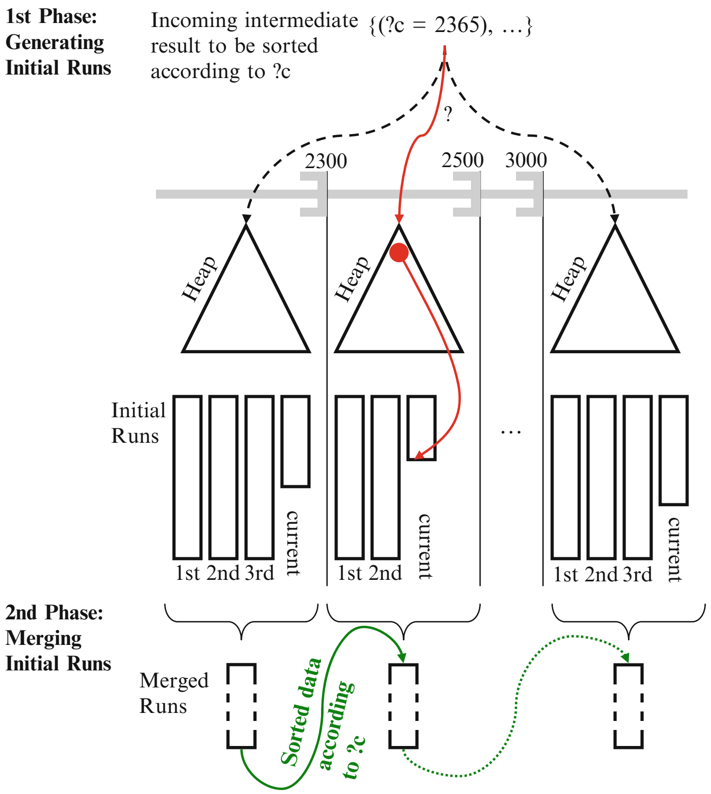
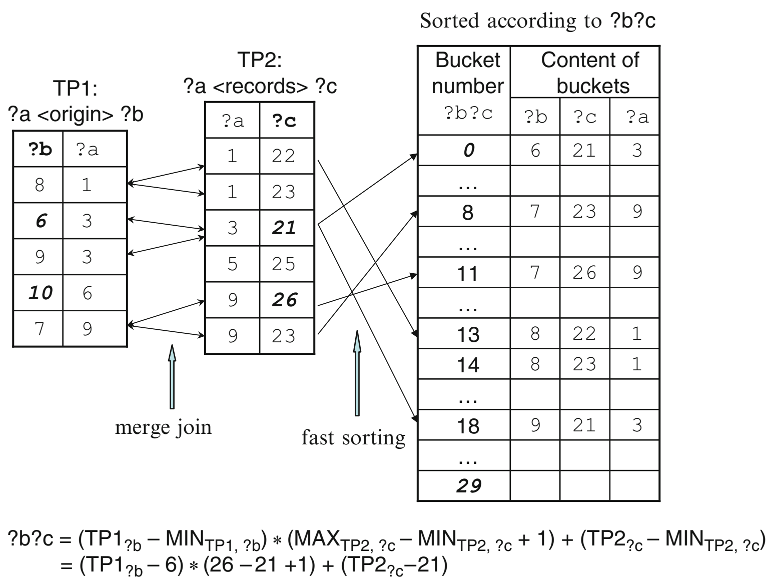

1st Phase: Generating Initial Runs
Incoming intermediate result to be sorted according to ?c
{(?c = 2365), ...}
?
2300
2500
3000
Heap
Heap
Heap
Initial Runs
current
current
current
1st 2nd 3rd
1st 2nd
1st 2nd 3rd
...
2nd Phase: Merging Initial Runs
Merged Runs
Sorted data according to ?c

Fig. 6.14 Using multiple merge sorts with replacement selection to fast sort data with large range

partner, even if there are many joins in a query. Nevertheless, our fast sorting approach described so far can be extended to handle the joins with arbitrary numbers of join partners. The idea is to combine several presorting numbers to a unique presorting number.

For example, the execution plan for the three triple patterns TP1 (`?a` < `origin` > `?b`), TP2 (`?a` < `records` > `?c`), and TP3 (`?c ?b` < `text`>) is first to compute the join with the join partner ?`a` between TP1 and TP2 and then to compute the join with the two join partners `?b` and `?c` between the results of the first join and of TP3. In order to use the merge join to compute the second join, the result of the first join needs to be sorted according to both variables `?c` and `?b`.

Figure 6.15 demonstrates how to sort the join result of TP1 and TP2 according to `?b` as the primary order and `?c` as the secondary order, denoted by `?b?c`, while computing the join. In order to sort the join result according to `?b?c`, we use (1) the id values of `?b` returned by TP1, denoted as $\texttt{TP1}_{\texttt{?b}}$; and (2) the id values of `?c` returned by TP2, denoted as $\texttt{TP2}_{\texttt{?c}}$. Let $\texttt{MAX}_{\texttt{TP1,?b}}$ and $\texttt{MIN}_{\texttt{TP1,?b}}$ be



?b	?a
8	1
6	3
9	3
10	6
7	9

?a	?c
1	22
1	23
3	***21***
5	25
9	***26***
9	23

Bucket number	Content of buckets		
?b?c	?b	?c	?a
0	6	21	3
...			
8	7	23	9
...			
11	7	26	9
...			
13	8	22	1
14	8	23	1
...			
18	9	21	3
...			
29			

$$?b?c = (TP1_{?b} - MIN_{TP1,\,?b}) * (MAX_{TP2,\,?c} - MIN_{TP2,\,?c} + 1) + (TP2_{?c} - MIN_{TP2,\,?c})$$
$$= (TP1_{?b} - 6) * (26 - 21 + 1) + (TP2_{?c} - 21)$$

Fig. 6.15 Sorting the result of join of TP1 and TP2 according to ?b (primary order) and ?c (secondary order)

the maximal and minimal values of $\mathtt{TP1}_{\mathtt{?b}}$, and $\mathtt{MAX}_{\mathtt{TP2,?c}}$ and $\mathtt{MIN}_{\mathtt{TP2,?c}}$ be the maximal and minimal values of $\mathtt{TP2}_{\mathtt{?c}}$. We use the formula,

$$(\mathrm{TP1}_{?b} - \mathrm{MIN}_{\mathrm{TP1},?b}) * (\mathrm{MAX}_{\mathrm{TP2},?c} - \mathrm{MIN}_{\mathrm{TP2},?c} + 1) + (\mathrm{TP2}_{?c} - \mathrm{MIN}_{\mathrm{TP2},?c})$$

$$= (\mathrm{TP1}_{?b} - 6) * (26 - 21 + 1) + (\mathrm{TP2}_{?c} - 21),$$

to sort and compute unique presorting numbers of the join result between TP1 and TP2. Note that this formula is very similar to the function used in compilers for mapping an entry of a multidimensional array to a memory address.

Proposition 1. *The formula,*

$$(\mathrm{TP1}_{?b} - \mathrm{MIN}_{\mathrm{TP1},?b}) * (\mathrm{MAX}_{\mathrm{TP2},?c} - \mathrm{MIN}_{\mathrm{TP2},?c}+1) + (\mathrm{TP2}_{?c} - \mathrm{MIN}_{\mathrm{TP2},?c}),$$

generates the unique combined presorting numbers and defines `?b` *as the primary sort criterion and* `?c` *as the secondary sort criterion. The minimal combined presorting number is 0 and the maximal combined presorting number is*

$$(\mathrm{MAX}_{\mathrm{TP1},?b} - \mathrm{MIN}_{\mathrm{TP1},?b}) * (\mathrm{MAX}_{\mathrm{TP2},?c} - \mathrm{MIN}_{\mathrm{TP2},?c}+1) + (\mathrm{MAX}_{\mathrm{TP2},?c} - \mathrm{MIN}_{\mathrm{TP2},?c}).$$

Proof Sketch. The first part of the formula, $(\mathrm{TP1}_{?b} - \mathrm{MIN}_{\mathrm{TP1},?b})$, is used to compute the number of values of `?b`. Each value of `?b`, that is, $\mathrm{TP1}_{?b}$, can be combined with any $\mathtt{TP2}_{?\mathtt{c}}$, and thus the number of possible combinations is the second part * ($\mathrm{MAX}_{\mathrm{TP2},?\mathrm{c}} - \mathrm{MIN}_{\mathrm{TP2},?\mathrm{c}} + 1$). The first and second parts together ($\mathrm{TP1}_{?\mathrm{b}} - \mathrm{MIN}_{\mathrm{TP1},?\mathrm{b}}$)*($\mathrm{MAX}_{\mathrm{TP2},?\mathrm{c}} - \mathrm{MIN}_{\mathrm{TP2},?\mathrm{c}} + 1$) allocate $\mathrm{TP1}_{?\mathrm{b}} - \mathrm{MIN}_{\mathrm{TP1},?\mathrm{b}}$ spaces. Each space has the size $\mathrm{MAX}_{\mathrm{TP2},?\mathrm{c}} - \mathrm{MIN}_{\mathrm{TP2},?\mathrm{c}} + 1$ and is for one $\mathtt{TP1}_{\mathtt{POS}}$ to combine with any $\mathtt{TP2}_{?\mathtt{c}}$. This part is also used to sort `?b`. The third part + ($\mathrm{TP2}_{?\mathrm{c}} - \mathrm{MIN}_{\mathrm{TP2},?\mathrm{c}}$) sorts `?c` in each space. In the space where one $\mathrm{TP1}_{?\mathrm{b}}$ is located, the possible maximal combined presorting number is $\max_{\mathrm{p}} = (\mathrm{TP1}_{?\mathrm{b}} - \mathrm{MIN}_{\mathrm{TP1},?\mathrm{b}}) * (\mathrm{MAX}_{\mathrm{TP2},?\mathrm{c}} - \mathrm{MIN}_{\mathrm{TP2},?\mathrm{c}} + 1) + (\mathrm{MAX}_{\mathrm{TP2},?\mathrm{c}} - \mathrm{MIN}_{\mathrm{TP2},?\mathrm{c}})$. In the space where $1 + \mathrm{TP1}_{?\mathrm{b}}$ is located, the possible minimal combined presorting number is $\min_{\mathrm{p+1}} = (1 + \mathrm{TP1}_{?\mathrm{b}} - \mathrm{MIN}_{\mathrm{TP1},?\mathrm{b}}) * (\mathrm{MAX}_{\mathrm{TP2},?\mathrm{c}} - \mathrm{MIN}_{\mathrm{TP2},?\mathrm{c}} + 1)$. Since $\min_{\mathrm{p+1}} - \max_{\mathrm{p}} = 1$, $\mathrm{TP1}_{?\mathrm{b}} - \mathrm{MIN}_{\mathrm{TP1}-?\mathrm{b}}$ spaces are disjoint and the combined presorting numbers are unique.

Proposition 2. *In general, in order to sort data according to* `n` *variables, we need* `n` *types of presorting numbers* $\mathtt{P}_1, \ldots, \mathtt{P}_n$, *where the maximal presorting numbers are* $\mathtt{MAX}_{\mathtt{P1}}, \ldots, \mathtt{MAX}_{\mathtt{P}n}$, *and all the minimal presorting numbers are 0. If the range of the original presorting numbers* `p` *is in* `[MIN, MAX]`, *then we compute a new* $\mathtt{p}' = \mathtt{p-MIN}$, *and thus the minimal value of* $\mathtt{p}'$ *is* 0. *We can combine these presorting numbers and retrieve the unique combined presorting numbers in* `n!` *different ways. For example, the formula,* $(\ldots((\mathrm{P}_1{*}(\mathrm{MAX}_{\mathrm{P2}} + 1) + \mathrm{P}_2){*}(\mathrm{MAX}_{\mathrm{P3}} + 1) + \mathrm{P}_3){*}(\ldots(\mathrm{MAX}_{\mathrm{P}n} + 1)) + \mathrm{P}_n$, *computes the unique presorting numbers and defines* `P1` *as the primary sort criterion,* `P2` *as the secondary sort criterion, ...,* `Pn` *as the nth sort criterion. The maximum of the combined presorting numbers is* $(\ldots((\mathrm{MAX}_{\mathrm{P1}}{*}(\mathrm{MAX}_{\mathrm{P2}} + 1) + \mathrm{MAX}_{\mathrm{P2}}){*}(\mathrm{MAX}_{\mathrm{P3}} + 1) + \mathrm{MAX}_{\mathrm{P3}}){*}\ldots){*}(\mathrm{MAX}_{\mathrm{Pn}} + 1) + \mathrm{MAX}_{\mathrm{P}n}$.

Proof Sketch. According to Proposition 1, the most inner part, $\mathtt{P}_1{*}(\mathtt{MAX}_{\mathtt{P2}} + 1) + \mathtt{P}_2$, generates the unique presorting numbers of the combined $\mathtt{P}_1$ and $\mathtt{P}_2$, and defines $\mathtt{P}_1$ as the primary sort criterion and $\mathtt{P}_2$ as the secondary sort criterion, denoted by $\mathtt{P}_1\mathtt{P}_2$; $(\mathtt{P}_1\mathtt{P}_2{*}(\mathtt{MAX}_{\mathtt{P3}} + 1) + \mathtt{P}_3)$ generates the unique presorting numbers of the combined $\mathtt{P}_1$, $\mathtt{P}_2$, and $\mathtt{P}_3$, and defines $\mathtt{P}_1\mathtt{P}_2$ as the primary sort criterion and `P3` as the secondary sort criterion, denoted by $\mathtt{P}_1\mathtt{P}_2\mathtt{P}_3$. We proceed analogously until we get the unique presorting number of the combined $\mathtt{P}_1, \ldots, \mathtt{P}_n$. In this way, we can prove the correctness of the formula.

The `MAX–MIN` range can grow quickly when processing a join with multiple join partners. However, a large range of presorting numbers is not a problem for our fast sorting approach, because we divide the whole range of the presorting numbers into *m* smaller ranges and (merge) sort the ranges independently from each other. Therefore and based on the experience in our experiments, we neither need more space nor more time for a larger range, if the number of data to be sorted is the same.

6.7.5 Additional Benefits from SIP Strategies

A merge join looks for the equivalent values from two sorted operands by comparing their results pairwise. If two values are unequal, the merge join continues reading the following data from the operand with the smaller value until an equal or larger value is seen. When processing very large datasets, reading each data in turn for finding a certain value is quite time-consuming.

An improvement is using the SIP strategy as described in (Neumann and Weikum 2009). SIP applies the information of the larger value L to the side of the operand with the smaller value for directly going to the data, which is equal to or larger than L. In particular, when data are stored using B^+-trees, the larger value L can be used as key for directly finding the wanted leaf. Using SIP allows jumping over big gaps by accessing interior nodes of the B^+-trees and thus avoiding searching along a possibly long chain of leaves.

For hash joins, the SIP strategy can use bloom filters. A bloom filter is a bit vector, which is created by applying a hash function to one operand. This bit vector is afterward used to filter out irrelevant data of the other operand. The bloom filter can also be used to compute the number of distinct values, which can be jumped over. The upper bound for this number corresponds to the number of unset bits in the bloom filter. However, since a hash function might map many distinct values to a same integer, the real number of distinct values to be jumped over can be (much) larger. Furthermore, the number of distinct values is at most the length of the bit vector of the bloom filter. Therefore, using bloom filters is not scalable.

Having our fast sorting capability, the efficient merge join can be applied instead of the hash join. Furthermore, using SIP merge joins can jump over bigger gaps than hash joins. Therefore, the application of our sorting numbering scheme in combination with SIP can significantly speeds up query processing.

6.8 Optional

The Optional operator returns the result of a join between its operands and additionally all not joined solutions of its left operand. Therefore, we can modify any join algorithm to return joined solutions and additionally not joined solutions of the left operand. We do not provide adapted algorithms computing the result of an Optional operation for all discussed join algorithms here and leave this to the reader, but present the adapted algorithm of the merge join algorithm using SIP in the next subsection.

6.8.1 MergeOptional

Assuming S to be the left and R to be the right operand, we modify the merge join algorithm using SIP to calculate the result of an OPTIONAL construct in the following way:

```
S.open();
R.open();
s=S.next();
r=R.next();
WHILE(s!=null && r!=null){
    IF(s < r){
            OUTPUT s;
            s=S.next();
    } ELSE IF(r < s){
            r=R.next(s);
    } ELSE {
            s1=s;
            r1=r;
            operand1 = {};
            operand2 = {};
            WHILE(joinable(r1, s)){
                    operand1 = operand1 $\cup$ {s};
                    s=S.next();
            }
            WHILE(joinable(r, s1)){
                    operand2 = operand2 $\cup$ {r};
                    r=R.next();
            }
            FOR EACH s2 IN operand1 DO
                    FOR EACH r2 IN operand2 DO
                            OUTPUT join(s2, r2);
    }
}
WHILE(s!=null){
    OUTPUT s;
    s=S.next();
}
```

In this version of the algorithm, we return *s* as result whenever no join partners in *R* can be found. The modified code is marked with boldface.

6.9 Duplicate Elimination

SPARQL uses the modifier *DISTINCT* to require a result without duplicates. In this subsection, we describe different algorithms for the elimination of duplicates.

6.9.1 Duplicate Elimination Using Hashing

This version of duplicate elimination first partitions its input data by using hash functions until all partitions fit into main memory. Afterward, the algorithm can use any in-memory algorithm for duplicate elimination by, for example, determining the set of solutions in each partition. Sets can be determined from a sequence of solutions containing duplicates by, for example, using a hash table or balanced trees like AVL trees.

6.9.2 Duplicate Elimination Using Sorting

Duplicate elimination using sorting just first sorts its solutions and afterward scans the sorted solutions, and returns only those solutions, which are different to their previous ones. In some query plans, the solutions are already sorted before the *DISTINCT* operator, such that the *DISTINCT* operator does not need to sort the data initially.

6.9.3 Duplicate Elimination Using Presorting Numbers

Fast sorting data using presorting numbers is of great benefit not only to merge joins, but also to many other operations in query processing. Using the presorting numbers, *DISTINCT* operations can be efficiently performed: when processing the operation just before the *DISTINCT* operation, once a solution is computed, it is sorted using the corresponding presorting numbers in the same way for sorting the join result as illustrated in Fig. 6.13. However, we only store one of the solutions with the same presorting number. When the operation before the *DISTINCT* operation is finished, its result will not contain the duplicates. Consequently, a separate *DISTINCT* operation is not needed anymore.

6.10 Cost Model

In the previous chapter, we have already described logical plan generation based on estimations of the result cardinality. The main goal there was to reduce the number of solutions as early as possible in the operatorgraph.

Different physical operators have different costs concerning CPU processing time, I/O costs for reading and storing solutions, as well as costs for used space in main memory or on disk. The physical plan generator takes all these costs into consideration, computes the total costs of possible physical operators, and chooses the physical operator with the best estimated total costs. As I/O costs are typically the lion's share in the total costs, that is, I/O operations are the slowest ones in a nondistributed computer system, most physical query optimizers focus on the I/O costs. Some query optimizers determine the I/O costs by estimating the number of solutions to be read and written by a specific operator. Other query optimizers compute the I/O costs by estimating the number of page accesses for a specific operator, which considers the block nature of I/O devices such as hard disks. Both variants produce similar good results, but cannot be mixed.

6.11 Performance Evaluation

For the performance evaluation, we consider two main scenarios.

In the first scenario, we use small to middle-sized datasets, which completely fit into main memory. In this scenario, our approaches for main-memory indexing and join processing promise optimal performance.

In the second scenario, we use large-scale datasets with over one billion triples. These datasets do not fit into main memory anymore and we comprehensively analyze the disk-based indexing (and joining) approaches.

6.11.1 Performance Evaluation for In-memory Databases

The SP^{2}B benchmark (Schmidt et al. 2009) includes a set of 18 queries, which contain more features of SPARQL and address more optimization techniques than many other Semantic Web benchmarks such as the LUBM benchmark (Guo et al. 2005). The SP^{2}B benchmark uses a data generator, which can generate data of different size. The SP^{2}B data set imitates an RDF version of the real-world DBLP data set (Ley 2010); that is, the data structure of the SP^{2}B data set is very similar to real-world data. Furthermore, the SP^{2}B benchmark does not consider inference based on an ontology, which we do not consider here due to our focus on basic join algorithms. Therefore, we choose the SP^{2}B benchmark in our experiments. In our figures, we present the average of ten execution times of reading the input and generating the indices, and of applying the SPARQL queries of the SP^{2}B benchmark.

The test system uses an Intel Core 2 Duo CPU T7500 with 2.2 GHz, 2 GB main memory, Windows XP Professional 2002, and Java 1.6. We use Jena ARQ (Wilkinson et al. 2003) and Sesame (Broekstra et al. 2002) as SPARQL database engines since they support the current SPARQL version (Prud'hommeaux and Seaborne 2008), which is not fully supported by many other SPARQL processing engines. Furthermore, we have implemented an in-memory version of the approach presented in (Weiss et al. 2008), which we call *In-memory Hexastore* in the following paragraphs. In-memory Hexastore uses merge joins to join two triple patterns at the first level and standard relational join algorithms like index joins for succeeding joins. Our implementation of In-memory Hexastore reorders the join operands according to the result sizes of triple patterns.

In the figures, we call our approach *RestrictingTP*. We have measured different variants of our approach. *RestrictingTP-OrderSize* represents the execution times of our approach when reordering the triple patterns according to the result sizes of each triple pattern. *RestrictingTP-OrderVar* represents the execution times of our approach when reordering the triple patterns according to the restrictiveness of triple patterns. *RestrictingTP-OrderVarSize* represents the hybrid approach, which orders the triple patterns primarily according to the restrictiveness of triple patterns and secondarily according to the result sizes of the triple patterns. We use hash maps for our seven indices, but use B$^+$-trees whenever we mark the experiment with B^+, where we also order the triple patterns primarily according to the restrictiveness and secondarily according to the result sizes of the triple patterns.

6.11.1.1 Index Construction Time

Reading the input data and constructing the indices only need to be done *once*. Afterward, the indices can be used to evaluate many queries. Thus, less time for query evaluation is more critical than less time for index construction. The index construction time for Sesame and Jena is the smallest (see Fig. 6.16). The index construction time for *RestrictingTP*, *RestrictingTP-orderSize*, *RestrictingTP-orderVar*, and *RestrictingTP-orderVarSize* is obviously the same, and thus we use the time for *RestrictingTP* for all of them, which are approximately two times slower than Jena and 3.6 times slower than Sesame. In-memory Hexastore and B + perform the worst.

6.11.1.2 Query Evaluation

For all queries used in this experiment, the time to compile queries and perform the logical and physical optimization step is below one millisecond.

For each query of the SP^{2}B benchmark (Schmidt et al. 2009), In-memory Hexastore performs worst, as In-memory Hexastore processes time-consuming searches in sorted lists for index accesses, which is avoided in our approach.

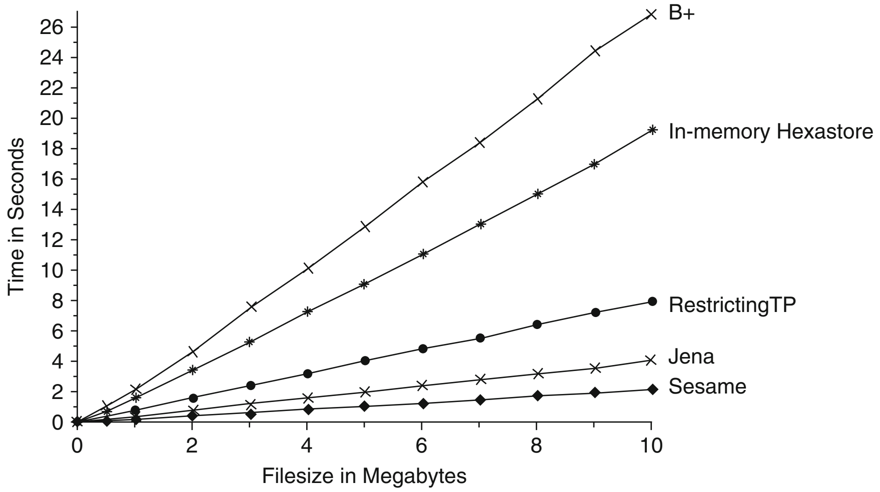


Fig. 6.16 Index construction time

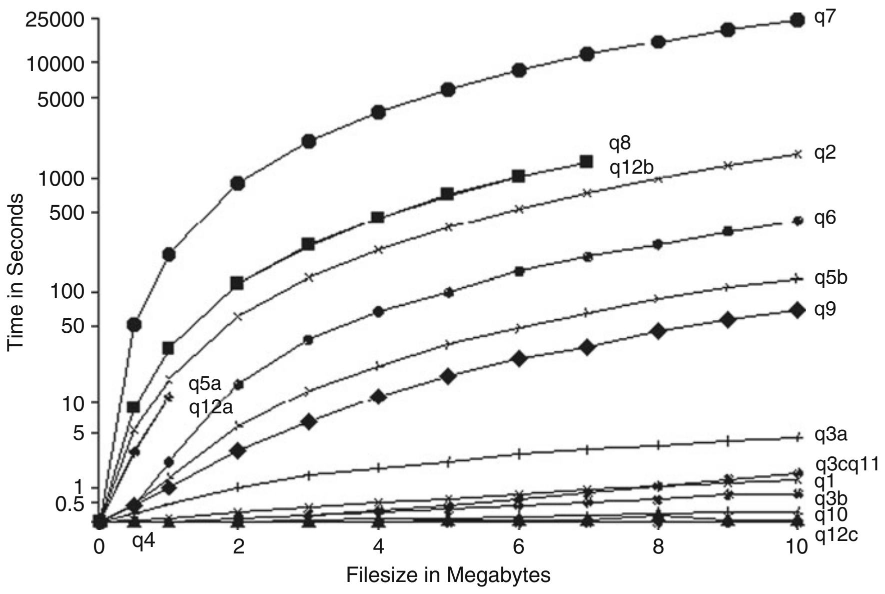


Fig. 6.17 Execution times of the queries q1–q12c of In-memory Hexastore

Furthermore, out-of-memory errors occur for the queries q4, q5a, and q12a. For simplicity of presentation, we present the execution times of In-memory Hexastore in an extra figure (see Fig. 6.17) and do not present them in the figures especially for the different queries.

Query q1. This query consists of three triple patterns, the overall result of which contains only one solution. All approaches except *In-memory Hexastore* need only less than 4 ms.

Query q2. This query consists of ten triple patterns, one of which is in an OPTIONAL construct. The result is additionally sorted. Sesame and Jena perform best (see Fig. 6.18). The implementation of sorting can be still much optimized in the benchmarked version of our SPARQL engine, such that we believe that our approach performs much better after these optimizations have been done.

Query q3a–c. These queries consist of two triple patterns and a comparison of a variable with a constant value. Due to space limitations, we present the execution times of only q3a here (see Fig. 6.19). For q3a, our approach performs best and then Sesame and Jena. For q3b and q3c, all approaches and query engines except of In-memory Hexastore need less than 100 ms, where *RestrictingTP-orderVarSize* performs best, the execution times of which are below 4 ms.

Query q4. This query consists of eight triple patterns with a less than value comparison ("<") between two variables. Figure 6.20 shows that *RestrictingTP-orderVarSize* performs best. Jena is the slowest. *In-memory Hexastore*, *RestrictingTP*, and *RestrictingTP-orderSize* compute the triple patterns in an inefficient way, such that out-of-memory errors occur.

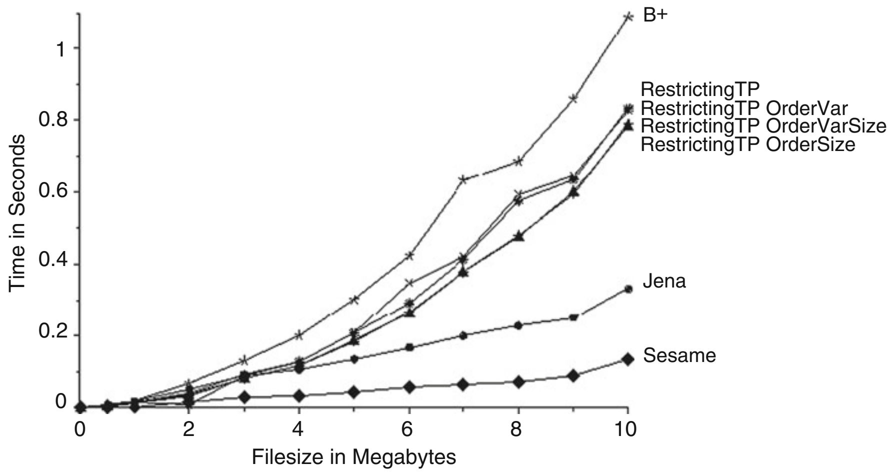


Fig. 6.18 Execution times of q2

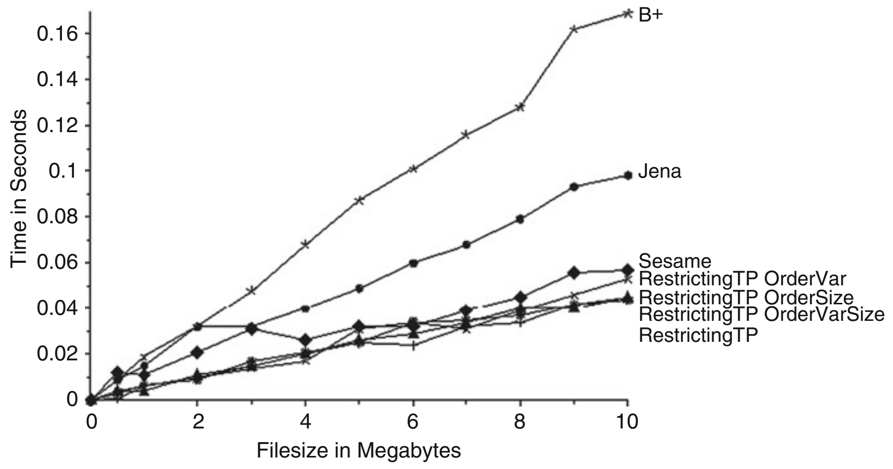


Fig. 6.19 Execution times of q3a

Queries q5a and q5b. q5b is a manually optimized query of q5a. *Restricting-orderVar* and *Restricting-orderVarSize* perform best for q5a (see Fig. 6.21). For q5b, *Restricting-orderVarSize* is slightly slower than Sesame (0.3 s), but 156 times faster than Jena.

Query q6. Our approaches are slightly slower than Sesame and Jena (see Fig. 6.22) since the complex OPTIONAL constructs with many filter expressions in q6. In the implementation of our prototype, there is still much space for optimizing this kind of queries.

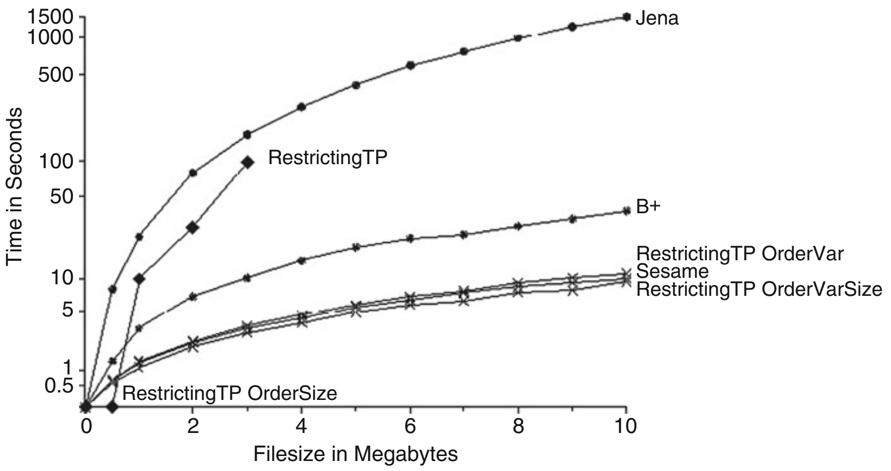


Fig. 6.20 Execution times of q4

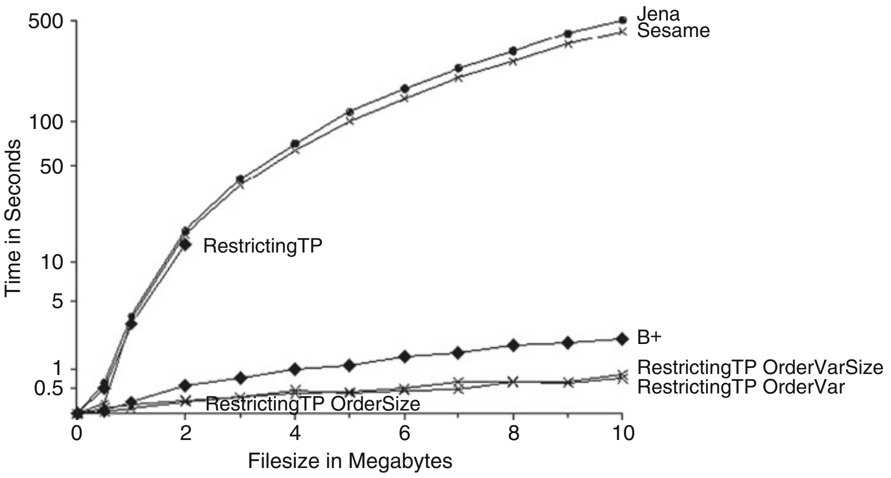


Fig. 6.21 Execution times of q5a

Query q7. q7 consists of several nested OPTIONAL constructs. Here, our approaches perform much better than the others (see Fig. 6.23).

Query q8 and q9. q8 and q9 contain common subexpressions in the operands of UNION, which is currently not optimized by our prototype. Therefore, the execution times of our prototype are higher than those of Sesame and Jena (see Fig. 6.24).

Queries q10 and q11: q10 and q11 consist of one triple pattern, the result of which is retrieved below 300 ms for all different approaches and query engines except of In-memory Hexastore.

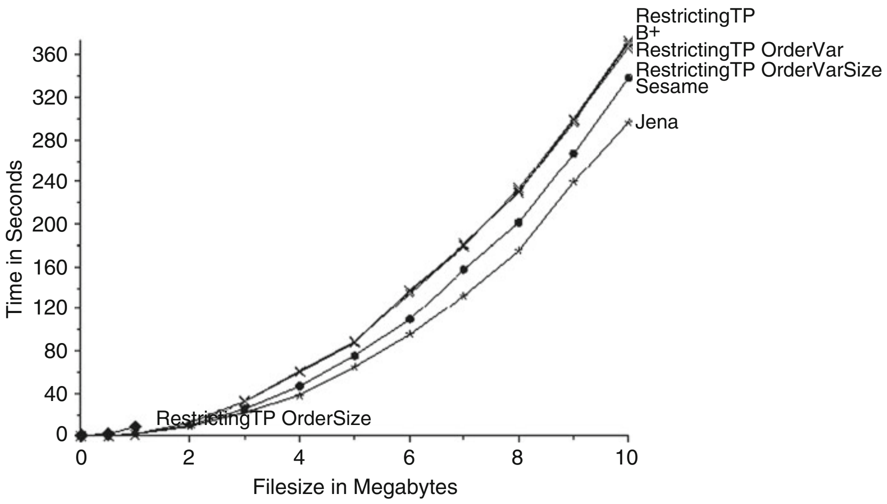


Fig. 6.22 Execution times of q6

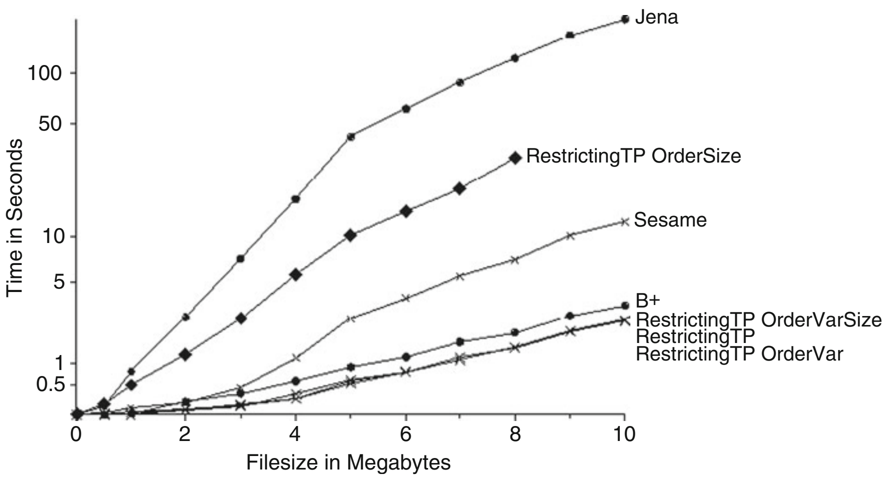


Fig. 6.23 Execution times of q7

Queries q12a–c. These queries contain the ASK construct; that is, they return true if there is some result and otherwise false. One optimization is to abort the evaluation of queries if there is at least one result. Our approaches perform similarly to the other approaches for q12c and are slightly slower for q12a and q12b.

Average execution times of all queries q1–q12c. Figure 6.25 shows the average execution times of all queries q1–q12c. Overall, our approaches and especially *RestrictingTP-OrderVarSize* are the fastest. Sesame is more than two times slower, and Jena is more than six times slower than *RestrictingTP-OrderVarSize*. In-memory Hexastore is the slowest approach.

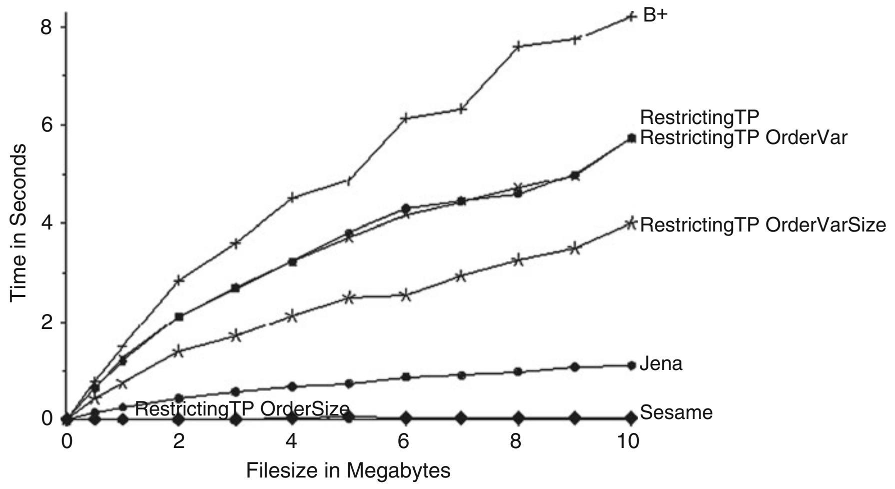


Fig. 6.24 Execution times of q8

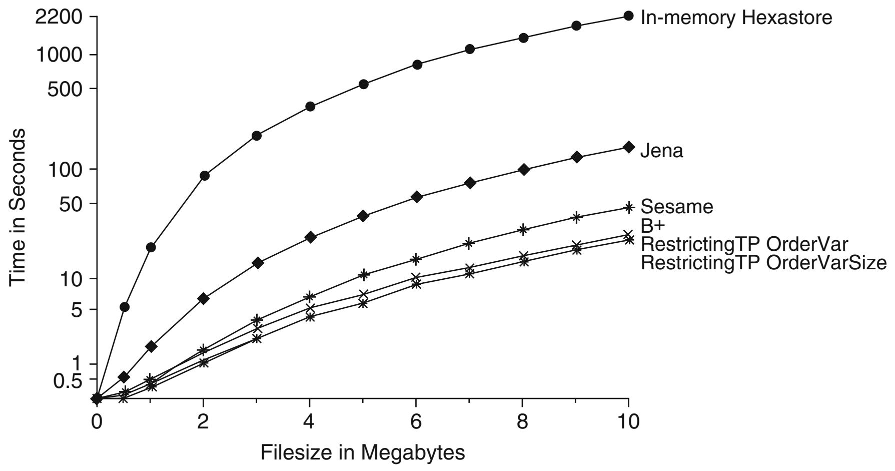


Fig. 6.25 Average execution times of all queries q1–q12c

6.11.2 Performance Evaluation for Large-Scale Datasets

We study the performance benefits for our sorting numbering scheme integrated into other index approaches. For these experiments, we focus on the index approach *RDF3X* in (Neumann and Weikum 2008, 2009), since it is similar to Hexastore in (Weiss et al. 2008), but uses a simpler and faster index structure than (Weiss et al. 2008). We compare the pure *RDF3X* approach, that is, using hash joins when data become unsorted, with the *RDF3X-Sort* approach, that is, our sorting numbering scheme integrated into the *RDF3X* approach. *RDF3X-Sort* uses fast sorting for applying merge joins to replace hash joins and for elimination of duplicates.

The original RDF3X prototype (Neumann and Weikum 2008, 2009) has several limitations:

- It does not support full SPARQL 1.0.
- It only supports very simple filter expressions.
- It neglects prefixes in predicates. This improves the performance, but leads to information loss. For example, the two different triples (<a>, <www.film/title>, "Ratatouille") and (<a>, <www.game/title>, "Ratatouille") are stored as one triple (<a>, <title>, "Ratatouille") in the original RDF3X prototype.
- It does not support data types, such that, for example, the two identical integer values $+2$ and 2 are treated as two different strings.
- It supports only in-memory hash joins; that is, if the operands of hash joins cannot fit into memory, RDF3X cannot process the hash joins.

In order to lift these limitations and avoid problems resulting from not supported features of the original RDF3X prototype, we have reimplemented the RDF3X approach. Our reimplementation successfully runs all the W3C test cases (Feigenbaum 2008), which contain over 200 queries. As we will show in the following subsections, the execution times of our reimplementation are similar to, and often outperform those of the original RDF3X prototype, compared with results of Neumann and Weikum (2009), although we have used Java as programming language and the RDF3X system is implemented in C++. An online demonstration of our implementations is publicly available (see Groppe and Groppe 2009; Groppe et al. 2009b).

The test system for the evaluation of queries uses a Dual Quad Core Intel CPU X5550 computer with 2.67 GHz, 6 GB main memory, Windows XP Professional (x64 Edition), and Java 1.6 64 bit. We have run the experiments ten times and present the average execution times as well as the standard deviation of the sample.

In order to build indices faster over the two very large datasets, index constructions are performed in a cluster with additional 6 Intel Core 2 Quad CPU Q9400 computers, each with 2.66 GHz, 4 GB main memory, Windows XP Professional (32 bit), and Java 1.6.

We use two large-scale datasets: UniProt (Swiss Institute of Bioinformatics 2009) and Billion Triples Challenge (BTC) (Semantic web challenge 2009), and corresponding queries. Three kinds of indices are constructed over the two datasets: two dictionary indices for mapping between RDF terms and integer ids; six evaluation indices according to the six collation orders of RDF for evaluating SPARQL queries; six histogram indices for fast generating histograms for triple patterns.

6.11.2.1 UniProt

UniProt (Swiss Institute of Bioinformatics 2009) is a comprehensive repository of protein sequence and annotation data. We have used the version 15.14 of the 9th February 2010 of it, which contains over 1.5 billion triples.

We have used the queries of Neumann and Weikum (2009), which we name UP 1 to UP 8. However, the number of results of these queries and especially the number of solutions for the hash joins are quite small. Therefore, we added some additional queries (EUP 1 to EUP 8) with bigger cardinalities, which are presented below.

Table 6.2 presents the types of operations of the UniProt queries performed by the pure *RDF3X* approach without using our fast sorting technique. In comparison, the *RDF3X-Sort* approach, that is, our sorting numbering scheme integrated into the *RDF3X* approach, uses fast sorting and merge joins instead of hash joins and optimizes duplicate elimination by also using our fast sorting technique. Table 6.1 presents the execution costs by the two approaches when they process these UniProt queries over 1.5 billion triples.

The queries UP 1 to UP 8 from RDF3X (Neumann and Weikum 2009) are favorable to RDF3X, since the number of solutions is quite small, and thus the simple and fast in-memory hash join can be used. Therefore, the performance gain is quite small by RDF3X-Sort for these queries. However, for queries with larger intermediate results, RDF3X-Sort is often about 10 times up to 32 times faster than the RDF3X approach (see EUPs 1, 2, 3, and 5).

The time for index construction for the over 1.5 billion triples was 57 h. The space consumption is 19.5 GB for the two dictionary indices, 47.1 GB for the six evaluation indices, and 47.1 GB for the six histogram indices.

Table 6.2 presents the types of operations of the UniProt queries performed by the pure *RDF3X* approach without using our fast sorting technique, when processing the UniProt queries. In the following paragraphs, we provide the additional queries for the UniProt dataset.

We assume that all UniProt queries define the namespaces `rdf`, `rdfs`, and `up` by

Table 6.1 Evaluation times (in seconds) for UniProt Data

Query	RDF3X	RDF3X-sort	RDF3X/ RDF3X-sort	Histogram computation
UP 1	0.0391 ± 0.0106	0.0375 ± 0.0174	1.043	0.14
UP 2	0.35 ± 0.021	0.35 ± 0.02	1	1.014
UP 3	0.925 ± 0.0702	0.8969 ± 0.0474	1.031	0.395
UP 4	0.321 ± 0.025	0.309 ± 0.013	1.039	0.233
UP 5	0.79 ± 0.04	0.79 ± 0.05	1	0.005
UP 6	0.55 ± 0.006	0.55 ± 0.01	1	0.02
UP 7	0.5453 ± 0.0047	0.5438 ± 0.0061	1.003	0.042
UP 8	0.8469 ± 0.0117	0.8515 ± 0.0203	0.995	0.286
EUP 1	27.320 ± 4.427	2.7468 ± 0.1044	9.946	2.061
EUP 2	3.1172 ± 0.1078	0.3391 ± 0.1081	9.193	0.572
EUP 3	1620.1 ± 5.98	49.6735 ± 0.236	32.615	1.614
EUP 4	1.55 ± 0.018	0.8657 ± 0.0102	1.79	1.05
EUP 5	0.2096 ± 0.0102	0.0172 ± 0.0046	12.186	0.595
EUP 6	2.5141 ± 0.0203	1.9937 ± 0.0077	1.261	0.7624
EUP 7	42.6124 ± 0.783	4.9063 ± 0.1245	8.685	0.5
EUP 8	0.2843 ± 0.0645	0.0376 ± 0.0187	7.561	0.55

```
PREFIX rdf:
       <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX up:   <http://purl.uniprot.org/core/>.
```

Query EUP 1:

```
select * where{
?x rdf:type up:Sequence_Conflict_Annotation;
 up:conflictingSequence ?y.
?y rdf:type up:External_Sequence.}
```

Query EUP 2:

```
select * where {
?c rdf:type up:Concept; rdfs:label ?l;
    up:obsolete "true"; rdfs:subClassOf ?c2.
?c2 rdfs:label ?l2.}
```

Query EUP 3:

```
select * where {
?x up:sequenceFor ?y; rdf:type up:Sequence.
?y rdf:type up :Protein;up:reviewed "false";
   up:created "2009-07-28".}
```

Query EUP 4:

```
select * where {
?x up:date "1996"; rdf:type>?t.
?t rdfs:subClassOf ?c.}
```

Query EUP 5:

```
select * where {
?x up:cofactor "Iron"; rdfs:subClassOf ?c.
?c up:name ?n.}
```

Query EUP 6:

```
select * where {
?x rdf:type up:Tissue; rdfs:label ?l1;
   rdfs:seeAlso ?y.
?y rdfs:label ?l2; up:database "eVOC".}
```

Query EUP 7:

```
select distinct * where {
?x rdf:type up:Sequence_Conflict_Annotation;
   up:conflictingSequence ?y.
?y rdf:type up:External_Sequence.}
```

Query EUP 8:

```
select distinct * where {
?x up:cofactor "Iron"; rdfs:subClassOf ?c.
?c up:name ?n.}
```

Table 6.2 Operations by RDF3X for the UniProt queries

Query	Number of merge joins	Number of hash joins	DISTINCT
UP 1	2	1	
UP 2	10	2	
UP 3	10	1	
UP 4	8	2	
UP 5	5	2	
UP 6	11	1	
UP 7	11	1	
UP 8	11	1	
EUP 1	1	1	
EUP 2	3	1	
EUP 3	3	1	
EUP 4	1	1	
EUP 5	1	1	
EUP 6	3	1	
EUP 7	1	1	√
EUP 8	1	1	√

6.11.2.2 Billion Triples Challenge

The major part of the dataset of the Billion Triples Challenge (BTC) (Semantic web challenge 2009) was crawled during February/March 2009 based on datasets provided by, for example, Falcon-S, Sindice, Swoogle, SWSE, and Watson. We have imported *all* over 830 million distinct triples of the Billion Triples Challenge. In comparison, the performance analysis in Neumann and Weikum (2009) used only a subset of it.

Like the UniProt queries used by RDF3X (Neumann and Weikum 2009), the queries for BTC used by (Neumann and Weikum 2009) return very small intermediate and final results. Therefore, we also use some additional queries (EBTC 1 to EBTC 8) with bigger cardinalities (see below), as well as the ones of (Neumann and Weikum 2009) (BTC 1 to BTC 8). Table 6.4 presents the query operations performed by the original RDF3X approach, and Table 6.3 presents the processing times of these queries by the two approaches.

Although the queries BTC 1 to BTC 8 are designed to retrieve very small results and thus are favorable to RDF3X, the RDF3X-Sort still has a similar (or a slightly better) evaluation performance (up to 24%). When processing those queries with large intermediate and final results (EBTC 1 to EBTC 8), the RDF3X-Sort approach shows significant performance improvements and is up to several orders of magnitude better than the pure RDF3X.

The time for index construction for the BTC dataset with over 830 million distinct triples was 30 h. The space consumption is 31.1 GB for the dictionary indices, 30.8 GB for the evaluation indices, and 30.8 GB for the histogram indices.

Table 6.3 Evaluation times (in seconds) for BTC Data

Query	RDF3X	RDF3X-sort	RDF3X/ RDF3X-sort	Histogram computation
BTC 1	0.0469 ± 0.0155	0.045 ± 0.0049	1.042	23.5
BTC 2	0.0359 ± 0.0076	0.0359 ± 0.0105	1	7.419
BTC 3	0.3032 ± 0.0547	0.3032 ± 0.0443	1	2.642
BTC 4	39.3234 ± 0.123	38.0327 ± 0.0491	1.034	73.23
BTC 5	0.37 ± 0.046	0.3344 ± 0.0341	1.106	0.631
BTC 6	1.3172 ± 0.0477	1.061 ± 0.063	1.241	4.801
BTC 7	0.3265 ± 0.0227	0.3264 ± 0.0127	1.0003	27.829
BTC 8	0.2266 ± 0.0144	0.2124 ± 0.0103	1.067	38.018
EBTC1	46.8687 ± 0.851	1.8563 ± 0.0918	25.248	1.116
EBTC2	30.4593 ± 1.0404	3.0844 ± 0.1191	9.875	15.461
EBTC3	0.8626 ± 0.0649	0.1188 ± 0.01443	7.26	0.759
EBTC4	531.9172 ± 2.340	0.636 ± 0.094	836.35	1.017
EBTC5	46.303 ± 2.827	4.014 ± 0.2672	11.535	2.514
EBTC6	1602.692 ± 24.59	36.8641 ± 0.317	43.476	12.341
EBTC7	3.4765 ± 0.2576	0.3142 ± 0.1046	11.065	0.9767
EBTC8	0.8797 ± 0.0627	0.1203 ± 0.035	7.313	0.774

Table 6.4 Operations by RDF3X for the BTC queries

Query	Number of merge joins	Number of hash joins	DISTINCT
BTC 1	3	0	
BTC 2	3	0	
BTC 3	4	0	
BTC 4	5	1	
BTC 5	2	1	√
BTC 6	2	2	√
BTC 7	4	3	√
BTC 8	3	1	
EBTC 1	1	1	
EBTC 2	2	1	
EBTC 3	1	1	
EBTC 4	1	1	
EBTC 5	2	1	
EBTC 6	4	1	
EBTC 7	2	1	√
EBTC 8	1	1	√

Table 6.4 presents the query operations performed by the original RDF3X approach. The additional BTC queries, which we have used in the experiments, are provided below as follows:

We assume that all BTC queries define the namespaces `rdf`, `rdfs`, `foaf`, `dbpedia`, `purl`, `sioc`, `skos`, `atom`, `geoont`, `geocountry`, and `geopos` by

```
PREFIX rdf:
     <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:
     <http://www.w3.org/2000/01/rdf-schema#>
PREFIX foaf:    <http://xmlns.com/foaf/0.1/>
PREFIX dbpedia: <http://dbpedia.org/property/>
PREFIX purl:    <http://purl.org/dc/elements/1.1/>
PREFIX sioc:    <http://rdfs.org/sioc/ns#>
PREFIX skos:   <http://www.w3.org/2004/02/skos/core#>
PREFIX atom:    <http://www.w3.org/2005/>
PREFIX geoont:  <http://www.geonames.org/ontology#>
PREFIX gecountry:
                <http://www.geonames.org/countries/#>
PREFIX geopos:
          <http://www.w3.org/2003/01/geo/wgs84_pos#>.
```

Query EBTC 1:

```
select * where {
?x foaf:depiction ?d;
    dbpedia:hasPhotoCollection ?y.
?y foaf:maker ?m.}
```

Query EBTC 2:

```
select * where {
?x purl:title "Wimbledon_College_of_Art";
    sioc:has_creator ?c; sioc:links_to ?l.
?l purl:title ?t.}
```

Query EBTC 3:

```
select * where {
?x skos:subject <http://dbpedia.org/resource/
                       Category:1960_in_Formula_One>;
    dbpedia:wikilink ?l.
?l foaf:name ?n.}
```

Query EBTC 4:

```
select * where {
?m purl:title ?t.
?x foaf:made ?m; foaf:nick ?i.}
```

Query EBTC 5:

```
select * where {
?x atom:Atomuri ?u; atom:Atomname ?n.
?y atom:Atomname ?n; atom:Atomemail ?m.}
```

Query EBTC 6:

```
select * where {
?x geoont:name> ?n1; geopos:lat ?l;
   geoont:inCountry geocountry:DE.
?y geoont:name ?n2; geopos:lat ?l;
   geoont:inCountry geocountry:DE.}
```

Query EBTC 7:

```
select distinct ?t where {
?x purl:title "Wimbledon_College_of_Art";
   sioc:has_creator ?c; sioc:links_to ?l.
?l purl:title ?t.}
```

Query EBTC 8:

```
select distinct ?nwhere {
?x skos:subject <http://dbpedia.org/resource/
                        Category:1960_in_Formula_One>;
    dbpedia:wikilink ?l.
?l foaf:name ?n.}
```

6.11.2.3 Performance Gains

Several main factors contribute to the significant performance gains from our presorting numbering scheme, when processing queries with larger intermediate results:

1. A merge join with a fast sorting phase is more efficient than a hash join.
2. It often occurs that one operand of joins is already sorted accordingly. Under such cases, RDF3X-Sort only needs to sort another operand. In comparison, when using a hash join, two operands must be processed anyway.
3. Using the SIP strategy merge joins can benefit more than hash joins, by jumping over much larger gaps.

6.12 Summary and Conclusions

We develop a new and efficient approach to computing joins in memory by dynamically restricting triple patterns and using seven indices.

Our experimental evaluation shows that our proposed approach for joining the result of triple patterns in memory is faster than in-memory variants of disk-based join algorithms and common in-memory SPARQL database engines. Concretely, the average execution of all SP^{2}B benchmark queries (Schmidt et al. 2009) of our approach is at least two times faster (see Fig. 6.25) than the compared approaches and common in-memory SPARQL database engines.

For efficiently querying the large-scale Semantic Web, we propose a sorting numbering scheme in order to fast sort solutions of SPARQL queries. Having the fast sorting capability, a merge sort join can be efficiently applied to compute the joins, data of which are unsorted. For large data sets, in combination with SIP strategies, the application of merge joins instead of hash joins leads to remarkable performance improvements. Elimination of duplicates also benefits significantly from the fast sorting capability. Our approach neither requires more space in the indices nor has extra update costs, since we use the ids of RDF terms as presorting numbers. By using histograms for the determination of the subranges for the buckets to be sorted in external mass storage, we ensure a good distribution between the buckets even if there are gaps in the values to be sorted, that is, intervals in the ids, which do not occur in the values, are quite common in large datasets, and could occur after updates.

Our experimental results show that merge joins using our fast sorting algorithm are more efficient than hash joins, and our fast sorting capability is a big benefit for elimination of duplicates. Our sorting numbering scheme significantly speeds up querying very large Semantic Web databases.