Chapter 8
Parallel Databases

Abstract While a number of optimizing techniques have been developed to efficiently process increasing large Semantic Web databases, these optimization approaches have not fully leveraged the powerful computation capability of modern computers. Today's multicore computers promise an enormous performance boost by providing a parallel computing platform. Although the parallel relational database systems have been well built, parallel query computing in Semantic Web databases have not extensively been studied. In this work, we develop the parallel algorithms for join computations of SPARQL queries. Our performance study shows that the parallel computation of SPARQL queries significantly speeds up querying large Semantic Web databases.

8.1 Motivation

The Semantic Web databases are becoming increasingly large, and a number of approaches and techniques have been suggested for efficiently managing and querying large-scale Semantic Web databases. However, current algorithms are implemented for sequential computation and do not fully leverage the computing capabilities of current standard multicore computers. Therefore, the querying performance of the Semantic Web databases can be further improved by maximally employing the capabilities of multicore computers, that is, parallel processing of SPARQL queries.

A parallel database system improves performance through parallelization of various operations, such as building indexes and evaluating queries. While centralized database systems allow parallelism between transactions (multiuser synchronization), parallel database systems additionally use parallelism between queries inside a transaction, between the operators and within individual operators of a query.

Ideally, the speedup from parallelization would be linear – doubling the number of processing units should halve the runtime. However, very few parallel approaches can achieve such ideal speedup. Since the existence of nonparallelizable parts, most of them have a near-linear speedup for small numbers of processing

S. Groppe, *Data Management and Query Processing in Semantic Web Databases*,
DOI 10.1007/978-3-642-19357-6_8,

units, and the speedup does not become larger than a certain constant value for large numbers of processing units. The potential speedup of an algorithm on a parallel computing platform is governed by Amdahl's law (Amdahl 1967). The Amdahl's law discloses that a small portion of the problem, which cannot be parallelized, will limit the overall speedup available from parallelization by a constant value. For this reason, parallel computing is only useful for either small numbers of processors or highly parallelizable problems. We will see that SPARQL query evaluation is highly parallelizable.

Two major techniques of parallelism used in parallel database systems are *pipelined* parallelism and *partitioned* parallelism (DeWitt and Gray 1992). By streaming the output of one operator into the input of another operator, the two operators can work in series giving pipelined parallelism. N operators executed using pipelined parallelism can achieve a potential speedup of N. By partitioning the input data into multiple parts, an operator can be run in parallel using the multiple processing units, with each working on a part of the data. The partitioned data and execution provide partitioned parallelism.

Parallelism can be obtained from conventional sequential algorithms by using *split* and *merge* operators. The proven and efficient techniques (e.g. Mishra and Eich 1992) developed for centralized database systems can be leveraged in a parallel system enhanced with the split and merge operators. New issues that need to be addressed in parallel query processing include data partitioning and parallel join computation. The strategies of data partitioning contain (multidimensional) range partitioning, round-robin partitioning, and hash partitioning (Graefe 1993). A large number of optimizing techniques for parallel join processing of relational queries have been developed (e.g., Boral et al. 1990; DeWitt et al. 1986; Kitsuregawa et al. 1983; Kitsuregawa and Ogawa 1990; Schneider and DeWitt 1989, 1990; Wolf et al. 1990; Zeller and Gray 1990), most of which focus on parallel hash-join algorithms.

Parallel computing and parallel relational databases have been employed for many years, and a number of efficient techniques have been developed, which can be leveraged for parallel processing of SPARQL. However, optimizing techniques for parallel relational databases do not specialize on the triple model of RDF and triple patterns of SPARQL queries. In this chapter, we develop a parallel Semantic Web database engine based on the RDF- and SPARQL-specific properties. We focus on parallelization approaches for standard hardware with multiple processing cores and common memory and shared disks. This chapter contains contributions from (Groppe et al. 2011a).

The contributions of this chapter are as follows:

- Parallel join computations especially for
  - Hash joins using a distribution thread, and
  - Merge joins using partitioned input,
- An approach to computing operands in parallel, and
- An experimental evaluation, which shows the performance benefits of parallel data structures and algorithms in Semantic Web databases.

8.2 Types of Parallelisms

Different types of parallelisms can be used during query processing (see Fig. 8.1). We describe some of these types in detail in the following paragraphs:

1. A transaction contains several queries and updates of a database application. Transactions typically conform to the ACID properties of databases:
   - (A) Atomicity: A transaction should be processed atomic; that is, the effects of all its queries and updates are visible to other transactions or none of them.
   - (C) Consistency: The database should be left in a consistent state before and after the transaction.
   - (I) Isolation: The transaction should be processed isolated; that is, the effect of the transaction should be the same as when all transactions are sequentially processed.
   - (D) Durability: The effect of a successful transaction must be durable even after system crashes, damages of storage systems, or other erroneous software or hardware.

   The multiuser synchronization of databases ensures the ACID properties for transactions, but allows processing different transactions in parallel (as much as possible considering conflicts between different transactions).
2. As a transaction contains several queries and updates, these queries and updates can be processed in parallel if they do not depend on each other: If an update

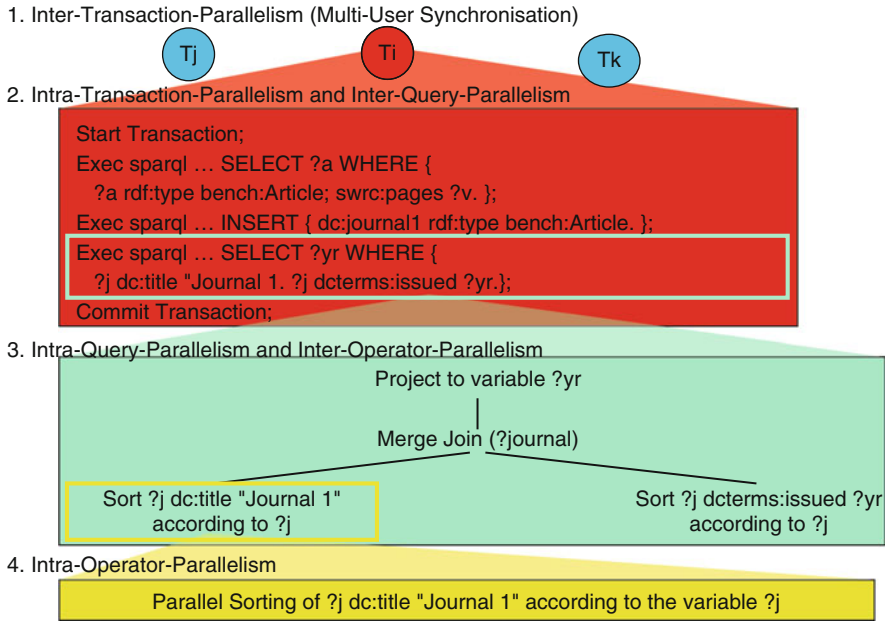

Fig. 8.1 Types of parallelisms in query processing

inserts, deletes, or modifies a triple, which influences the result of another query in the transaction, then the update and query must be processed in the order they occur in the transaction. The transaction in Fig. 8.1 contains an insertion of the triple *dc:journal1 rdf:type bench:Article*, which is matched by the triple pattern *?a rdf:type bench:Article* of the first query, such that the first query must be processed before the insertion. However, the last query does not contain any triple pattern matching the triple *dc:journal1 rdf:type bench:Article*, and therefore the last query can be processed in parallel to the insertion and also in parallel to the first query of the transaction.

3. A query (and also an update) of a transaction is transformed into an operator-graph consisting of several operators to be executed by the database system. Two forms of parallelisms can be applied here: The first form applies operators independent from each other in parallel. The second uses pipelining, where already computed intermediate results are transferred to subsequent operators as early as possible, which already processes this intermediate result further leading to a smaller memory footprint and to saving i/o accesses. We have already described pipelining in the chapter about physical optimization.
4. Many operators themselves can use parallel algorithms to determine their results. Prominent examples of such operators are sorting and join operators. For data parallelism, one tries to distribute the data into different disjoint fragments, such that operations like sorting or joins can be done in parallel in the different fragments. The extent of parallelism can be chosen dependent on the size of data; that is, larger data can be distributed into more fragments, such that more computers can be used to process the data in parallel leading to scalable solutions. Furthermore, data parallelism can be combined with pipelining.
   There are two forms of I/O parallelism (see Fig. 8.2):

   - The access parallelism uses different I/O devices like hard disks to process one job. Access parallelism is important for data parallelism and the distributed fragments should be therefore stored on different I/O devices.
   - During job parallelism independent jobs are processed on different I/O devices in parallel. Applying job parallelism is important for intertransaction parallelism and interquery parallelism.

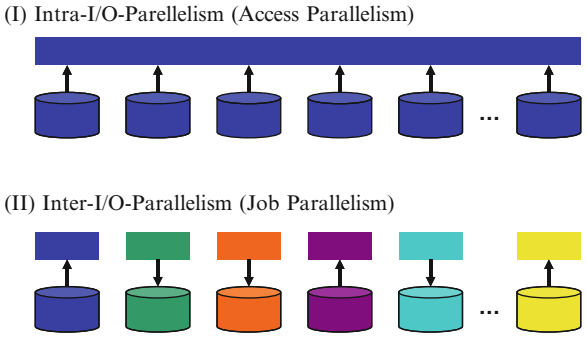

Fig. 8.2 Types of I/O parallelisms

8.3 Amdahl's Law

The batch speedup determines the response time improvement of parallel processing. The batch speedup for N computers is defined to be the response time when using one computer (and sequential algorithms) divided by the response time when using N computers, parallel algorithms, and the same data. Ideal would be a doubled batch speedup when using two times more computers (see Fig. 8.3). A linear improvement would be still fine, because one could determine the number of computers needed to obtain any response time one wants. However, experiments show that in typical cases the batch speedup does not increase or only slightly increases after a certain upper limit has been reached. In many cases, the batch speedup even decreases for more computers.

The reasons for the limits of scalability are startup and termination overhead of the parallel algorithms, inferences when accessing the logical and physical resources, overloads of individual computers, lock conflicts, limited partitioning possibilities of data, transactions and queries, and skew in the execution times of subtasks.

Amdahl's law now can determine an upper limit for the batch speedup, if the fraction of the execution time of the non-optimized, sequential part of the algorithm is known in relation to the overall execution time (see Fig. 8.4). Using this formula, one can determine a maximal batch speedup of 20 if the sequential fraction is only 5%.

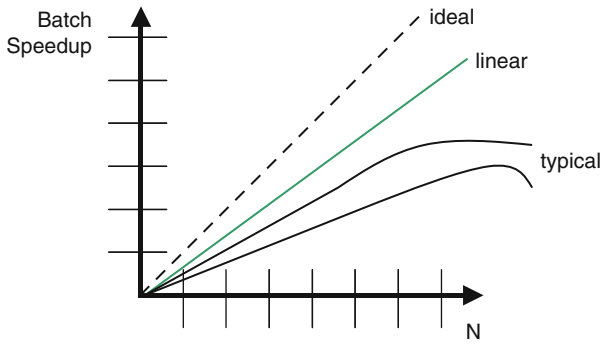

Fig. 8.3 Batch speedup dependent on the number N of computers

$$\text{Batch Speedup} = \frac{1}{(1-F_{opt}) + \frac{F_{opt}}{S_{opt}}}$$

F_{opt} = Fraction of the optimized (parallelized) component ($0 \le F_{opt} \le 1$)

S_{opt} = Speedup for the optimized (parallelized) component

Fig. 8.4 Formula for batch speedup

8.4 Parallel Monitors and Bounded Buffers

An important concept in parallel computing is parallel monitors. A parallel monitor is a high-level synchronization concept and is introduced in (Hoare 1974). It is a program module for concurrent programming with common storage, and has entry procedures, which are called by threads. The monitor guarantees mutual exclusion of calls of entry procedures: at most one thread executes an entry procedure of the monitor at any time. Condition variables may be defined in the monitor and used within entry procedures for condition synchronization.

An example of a parallel monitor is a bounded buffer. Mainly, a bounded buffer has two operations: *put* to store an element in the bounded buffer and *get* to retrieve an element from the bounded buffer. The bounded buffer has a specific constant limit for the number of stored elements. If a thread tries to put an element into a full bounded buffer, then the bounded buffer forces the calling thread to sleep until another thread calls *get* to retrieve one element. A *get* on an empty bounded buffer causes the calling thread to sleep until another thread puts one element inside.

The bounded buffers are typically used in producer/consumer patterns, where several threads produce elements and other threads consume these elements.

The advantage of bounded buffers in comparison to unbounded buffers is that the usage of the main memory is bounded, that is, consumer threads cannot store more elements than allowed by the main memory, thus avoiding the problem of out-of-memory errors. Therefore, we use bounded buffers for the communication between threads for the parallel join computation.

8.5 Parallel Join Using a Distribution Thread

When we use several threads for join computation (see Fig. 8.5) and the input is not partitioned yet, we have to partition the input data among these join threads using, for example, (multidimensional) range partitioning or hash partitioning.

The data must be partitioned according to the join variables in the following way: the solutions (of two join operands) with the same values bound to the join variables are distributed to the same thread. This ensures (a) that each join thread only involves the data in its own bounded buffers, and (b) that the overall join result is correct. Hash partitioning uses a hash function over the values of the join variables to determine to which join thread an input element is distributed. Hash partitioning promises good distributions and efficient partitioning computation. We hence use the hash partitioning for most parallel join processing.

For each operand of joins, we use a *data partitioning thread* to distribute the data into the bounded buffers of join threads (see Fig. 8.5). The join thread reads data from its two bounded buffers and performs a sequential join algorithm. If one join thread has finished its computation, then its result can be processed by succeeding operators (without waiting for the other join threads). This approach is the fastest one of parallelizing join algorithms such as the hash join and the index join.

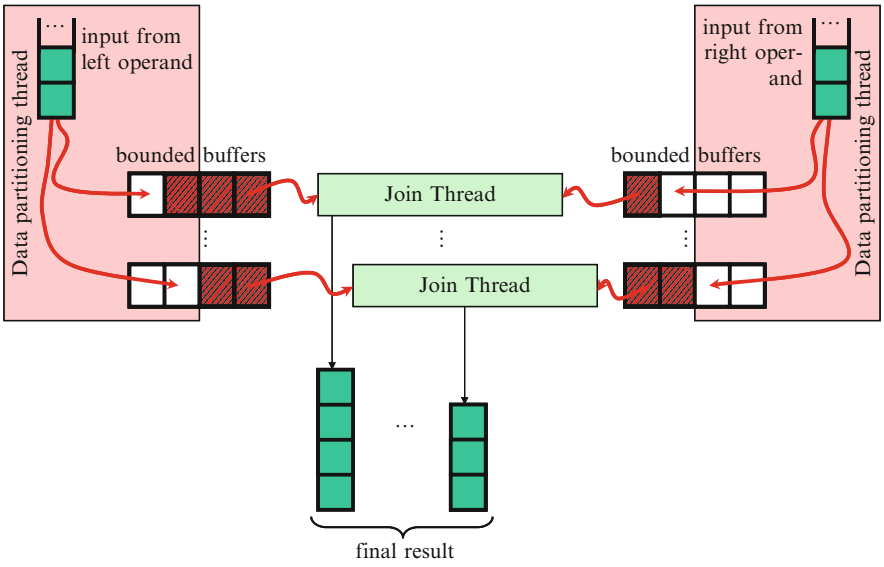

Fig. 8.5 Parallel join computation

When processing large data, the joins such as the disk-based hash join and index join involve complex operations during joining. Parallelizing these join algorithms, even plus the overhead of data partitioning, still can speed up join processing.

Merge joins on already sorted data are very fast and cheap in terms of I/O and CPU costs. The benefit from parallelizing merge joins cannot compensate the overhead of data partitioning even for large input data and large join results. We discuss how to parallelize merge joins in another way in the next section.

8.6 Parallel Merge Join Using Partitioned Input

As we mentioned before, a parallel merge join does not speed up the processing of joins if an additional partitioning process is needed. However, the processing performance benefits from the parallel computation of merge joins, if the input is already partitioned.

The merge join operator typically follows either the triple pattern operator in an operator graph, or the operators such as filters, which do not destroy the partitions of the input. Furthermore, the output of a merge join with such partitioned input is again partitioned, such that succeeding merge joins can use partitioned input as well. In order to generate a partitioned input for the merge join, we can retrieve the result of triple patterns using range partitioning (see Fig. 8.6). Range partitioning in comparison to hash partitioning has the advantage that the data in all the ranges can be read and processed in parallel.

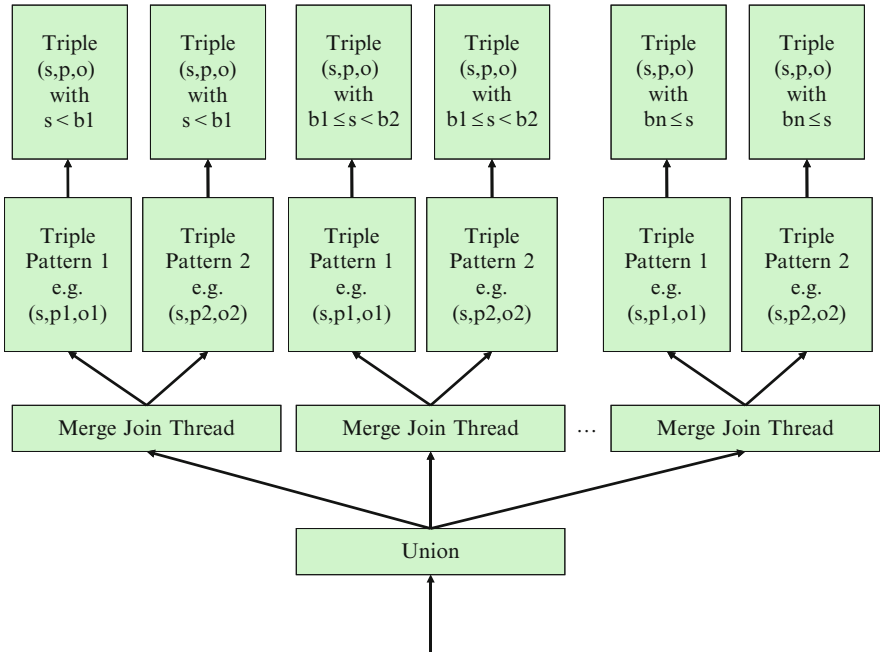

Fig. 8.6 Parallel merge join using range partitioning

SPARQL engines for large-scale data, such as *Hexastore* (Weiss et al. 2008) and *RDF3X* (Neumann and Weikum 2008, 2009), use six indices corresponding to 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, many joins can be computed using the fast merge join approach over sorted data. RDF3X employs just B^+-trees for each collation order and prefix searches, thus gaining a simpler and faster index structure than Hexastore.

Employing B+-trees as the indices for each collation order has another advantage: The results of retrieving B^+-trees can be very easily partitioned according to the range information in the histograms of triple patterns. For each kind of triple patterns, an equi-depth histogram (Piatetsky-Shapiro and Connell 1984) is constructed (see Chapter Logical Optimization) during the query optimization phase of our SPARQL engine. Among other information, each interval in this histogram contains the range and the number of triples allocated in this interval. We use special histogram indices to fast compute histograms over large datasets (see Chapter Physical Optimization).

Figure 8.7 describes how to get the partitioned results of a triple pattern using range partitioning: We assume that the data in the ranges [(3, 2, 1), (3, 4, 7)] will be distributed to the first partition and the data in [(3, 5, 5), (3, 7, 4)] to the second

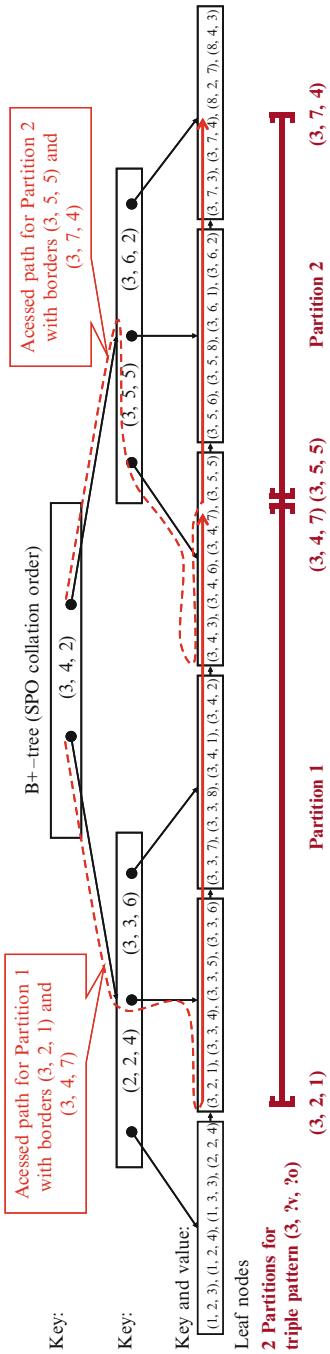

Fig. 8.7 B^+-tree accesses for partitioned input. Integer ids are used instead of RDF terms as components of the indexed triples

partition. Two partitioning threads can perform the range partitioning in parallel: One first searches for the border (3, 2, 1) in the B^+-tree and then follows the chain of leafs until the border (3, 4, 7), and the retrieved data belong to the first partition; another starts searching from the border (3, 5, 5) until the border (3, 7, 3) and retrieves the data for the second partition.

All approaches described so far for parallel joins apply also for the computation of OPTIONAL constructs. They are left outer-joins and can hence be analogously parallelized.

8.7 Parallel Computation of Operands

Another way to parallelize joins is to process their operands in parallel. In order to parallelize the processing of join operands, we use two *operand threads* (see Fig. 8.8). An operand thread computes its operand and puts the result into its bounded buffer.

Join approaches such as hash joins first read in the whole data of one operand and afterward start reading from the other operand. For such joins, the parallelism is not high, depending on the size of the bounded buffer. For the joins such as merge joins, which synchronously process the operands' data, two operand threads can work in parallel.

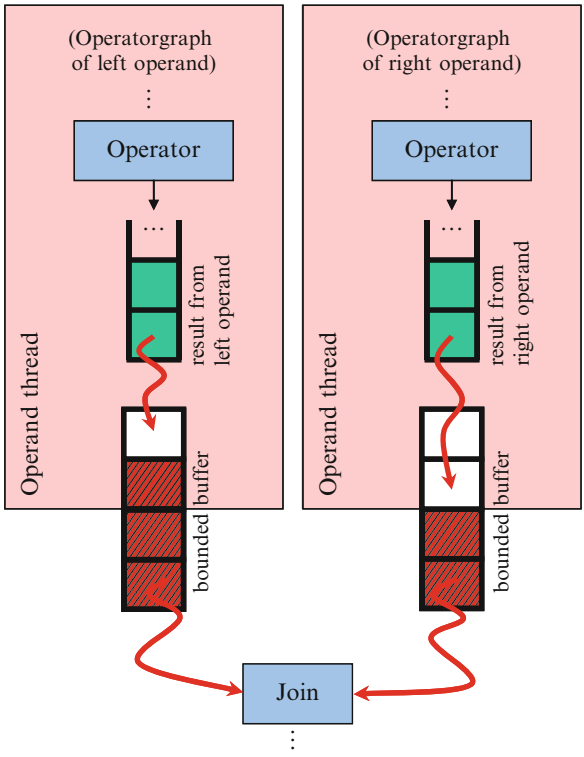

Fig. 8.8 Parallel computation of operands

However, advanced techniques such as sideways information passing (SIP) (Neumann and Weikum 2009) cannot be applied to parallel computation of operands, as using SIP to compute the next solution of one operand relies on the current solution of the other operand.

The experiments show that the parallel computation of operands speeds up the evaluation of some queries, especially if the processing of operands involves complex computations, but the previously discussed approaches for parallel joins are superior.

8.8 Performance Evaluation

We study the performance benefits for parallel join computations. 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 sequential join algorithms, with several parallel versions of the *RDF3X* approach: *PHJ RDF3X* is the *RDF3X* approach using a parallel hash-join algorithm with eight join threads and distribution threads; *PMJ RDF3X* uses parallel merge join algorithms with eight merge join threads with partitioned input; *PO RDF3X* computes the operands of the last hash join in parallel. Combinations such as *PMJ PO RDF3X* combine together several parallel approaches such as *PMJ RDF3X* and *PO RDF3X*.

The original RDF3X prototype (Neumann and Weikum 2008, 2009) has several limitations, as already explained in the chapter about physical optimization. In order to lift these limitations and avoid problems resulting from not supported features of the original RDF3X prototype, we use again our reimplementation of the RDF3X approach.

The test system for the performance analysis uses an 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 have run the experiments ten times and present the average execution times and the standard deviation of the sample.

We use the large-scale datasets of the Billion Triples Challenge (BTC) (Semantic web challenge 2009) and corresponding queries.

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.

The queries (BTC 1 to BTC 8) used by (Neumann and Weikum 2009) return very small intermediate and final results, which can be processed directly in memory. For these queries, the parallel approaches often do not show benefits and are dominated by their overhead. Therefore, we also use several additional queries (EBTC 1 to EBTC 8) with bigger cardinalities (see the Chapter Physical Optimization). Table 8.1 presents the processing times by the different approaches.

Although the queries BTC 1 to BTC 8 are designed to retrieve very small results and thus are favorable to the RDF3X using sequential algorithms, except for the query BTC 7, our parallel approaches have similar performance as the sequential

Table 8.1 Evaluation times (in seconds) for BTC Data. Entries in bold face part mark significantly fastest runtimes

Query	RDF3X	PHJ RDF3X	PMJ RDF3X	PH PMJ RDF3X	PO RDF3X	PO PHJ RDF3X	Size of results
BTC 1	0.047 ± 0.014	0.047 ± 0.026	0.047 ± 0.015	0.047 ± 0.033	0.045 ± 0.013	0.046 ± 0.028	2
BTC 2	0.042 ± 0.016	0.046 ± 0.01	0.119 ± 0.134	0.11 ± 0.13	0.12 ± 0.012	0.12 ± 0.01	48
BTC 3	0.452 ± 0.082	0.436 ± 0. 085	0.656 ± 0.047	0.74 ± 0.05	0.454 ± 0.105	0.48 ± 0.04	34
BTC 4	44.9 ± 0.1	45.99 ± 0.13	**31.98 ± 2.99**	33.8 ± 0.9	46.1 ± 0.2	44.86 ± 0.12	3
BTC 5	3.58 ± 0.12	3.52 ± 0.04	3 ± 0.06	**2.99 ± 0.1**	5.2 ± 0.2	5.15 ± 0.13	0
BTC 6	1.61 ± 0.03	1.65 ± 0.08	0.959 ± 0.564	0.76 ± 0.04	**0.63 ± 0.05**	0.76 ± 0.1	1
BTC 7	0.629 ± 0.077	6.33 ± 0.07	6.96 ± 2.74	13.7 ± 0.7	170 ± 1	173 ± 1	0
BTC 8	0.381 ± 0.070	0.36 ± 0.05	0.88 ± 0.09	0.87 ± 0.07	0.52 ± 0.07	0.52 ± 0.08	0
EBTC1	153.2 ± 3.7	**105.9 ± 1.7**	153.16 ± 3.38	113.44 ± 3.08	149 ± 2	139 ± 2	798,553
EBTC2	6.05 ± 0.41	**4.41 ± 0.71**	6.58 ± 0.34	4.69 ± 0.14	6.58 ± 0.7	5.6 ± 0.9	1,206
EBTC3	2.08 ± 0.05	3.12 ± 0.072	2.13 ± 0.13	2.13 ± 0.17	1.97 ± 0.13	3.3 ± 0.12	57
EBTC4	1,883 ± 32	**1,241 ± 26**	1,844 ± 18	1,297 ± 21	1,834 ± 21	1,850 ± 37	134,588
EBTC5	136.9 ± 4.1	145 ± 20	134 ± 3	160 ± 29	**129 ± 3**	137 ± 18	3,856,586
EBTC6	2,810 ± 24	1,736 ± 85	2,649 ± 39	1,799 ± 60	2,677 ± 9	**1,555 ± 38**	58,849,326
EBTC7	6.4 ± 0.34	**4.87 ± 0.85**	6.8 ± 0.62	5.37 ± 0.61	7.4 ± 0.9	5.33 ± 0.84	18
EBTC8	2.15 ± 0.07	3.01 ± 0.07	2.10 ± 0.06	3.07 ± 0.012	2.06 ± 0.13	3.35 ± 0.16	57

one. Our parallel approaches significantly outperform the sequential approach for the queries BTC 4 (PMJ RDF3X being the fastest), BTC 5 (PH PMJ RDF3X being the fastest) and BTC 6 (PO RDF3X being the fastest). For the EBTC queries, PHJ RDF3X is the fastest approach when evaluating EBTC 1, EBTC2, EBTC4, and EBTC 7, PO RDF3X is the fastest approach for EBTC 5, and PO PHJ RDF3X is the fastest one for EBTC 6. Parallel join algorithms work well whenever join results are large: If a merge join has a large result, PMJ RDF3X or PH PMJ RDF3X belongs to the fastest ones. If a hash join has a large result, PHJ RDF3X is the fastest.

8.9 Performance Gains and Loss

Several main factors contribute to the gain and loss of performance from parallelizing Semantic Web database engines:

(a) PHJ RDF3X performs best whenever hash joins have large results.
(b) Whenever merge joins have to process large input and have large results, then PMJ RDF3X or PH PMJ RDF3X outperforms the other sequential and parallel approaches.
(c) The overhead of parallel processing such as data partitioning will dominate if involved data are small in size.
(d) Advanced techniques such as sideways information passing (SIP) (Neumann and Weikum 2009) cannot be applied to some parallel computations.

Therefore, it is the task of the query optimizer to estimate the sizes of the join results and choose the proper join algorithms.

8.10 Summary and Conclusions

Since the disappearing of single-core computers, parallel computing has become the dominant paradigm in computer architectures. Therefore, developing parallel programs to fully employ the computing capabilities of multicore computers is under the necessity. This is especially important for time-consuming processing like querying growingly large Semantic Web databases. In this work, we develop a parallel SPARQL engine, especially focusing on the parallelism of join computation. For different join algorithms, we propose different parallel processing in order to maximally gain from parallel computing.

Our experimental results show that parallel join computation outperforms sequential join processing for large join results; otherwise, the parallel overhead compensates the performance improvements through parallelization. Therefore, the query optimizer must decide when to apply parallel join approaches. The proper application of parallel computation can significantly speed up querying very large Semantic Web databases.