Available online at www.sciencedirect.com
ScienceDirect
Information Systems 33 (2008) 203–220
www.elsevier.com/locate/infosys

An empirical evaluation of XQuery processors

Stefan Manegold

CWI, Kruislaan 413, 1098 SJ, Amsterdam, The Netherlands

Abstract

This paper presents an extensive and detailed experimental evaluation of XQuery processors. The study consists of running five publicly available XQuery benchmarks—the Michigan benchmark (MBench), XBench, XMach-1, XMark and X007—on six XQuery processors, three stand-alone (file-based) XQuery processors (Galax, Qizx/Open, Saxon-B) and three XML/XQuery database systems (BerkeleyDB/XML, MonetDB/XQuery, X-Hive/DB). Next to assessing and comparing the functionality, performance and scalability for the various systems, the major focus of this work is to report in detail about the experiences made while performing such an exhaustive study, to discuss all the problems that we encountered and how we solved them, and hence to hopefully provide some guidelines (or even a recipe) for performing reproducible large-scale experimental research and system evaluation.

Keywords: XML; XQuery; XQuery processors; Database systems; Benchmarks; Performance evaluation

1. Introduction

Experimental evaluation and comparison of (new) techniques, algorithms and/or complete systems is a vital means to assess the practical impact and benefit of research results, especially in applied domains such as data management systems. While many publications present experimental results, the extent of the presentation—or even the experiments themselves—are often very limited due to space, time and/or other resource constraints. The major focus of most research publications is on the (so-called) *scientific contributions*.

In this study, we shift the focus. Performing the actual experimental evaluation becomes the primary subject. Presenting the experimental setup in detail, we do not hesitate to reveal all the "nasty details" and "minor problems" that give us headaches and cause sleep-less nights. Although choosing XML data management using XQuery as the sample scenario for this study, we believe that the proposed techniques can easily be adapted to other data management scenarios. The major contribution is hence a detailed cookbook about how to conduct an experimental comparison and assessment of data management systems. However, though focusing on the actual experimentation techniques, we also present the "hard facts": the detailed performance results we gathered.

The remainder of this paper is organized as follows. We start with presenting our experimentation environment in Section 2. Section 3 lists the benchmarks that we are using and discusses adaptations of the benchmarks, their queries and documents that are necessary to perform our experiments. In Section 4, we introduce the six XQuery systems we use, and

E-mail address: stefan.manegold@cwi.nl

0306-4379/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.is.2007.05.004

explain in detail how we compile, install, configure and use them. Section 5 reveals how we run our experiments, validate results and timings, and collect the performance data that is presented in detail in Section 6. We draw final conclusions in Section 7.

2. The scenario

Given the popularity of XML in the data management world—both in research and in commercial environments—we chose XML data management using XQuery as the sample scenario for our experimental study. In particular, the idea is to run the five most popular publicly available XQuery benchmarks—the Michigan benchmark (MBench), XBench, XMach-1, XMark and X007—on the most popular publicly available XQuery processors. We picked three stand-alone (file-based) XQuery processors (Galax, Qizx/Open, Saxon-B) and three XML/XQuery database systems (BerkeleyDB/XML, MonetDB/XQuery, X-Hive/DB). For simplicity, we will use *engine* as a unified term to refer to both stand-alone (file-based) XQuery processors and XML/ XQuery database systems in the remainder of this text.

Despite using XML/XQuery as sample scenario, none of the general approaches and techniques presented and used in this work are limited to XML and/or XQuery. With obvious and straightforward adaptations, they can likewise be applied to other data management scenarios.

The chosen setup is very complex. First of all, there are 5×6 combinations of benchmarks with engines (actually 8×6, as XBench comes in four flavors, TC/SD, DC/SD, TC/MD, DC/MD). Secondly, each benchmark consists of 8 up to 46 individual queries (in total 163 queries for all five benchmarks). Thirdly, each benchmark provides (at least) 3–6 different document sizes. Hence, we face two major challenges: (1) running a huge set of experiments, and (2) collecting, analyzing, and presenting a huge amount of experimental results. To tackle these challenges, we decided to use the *XCheck benchmark platform* [1].

2.1. Benchmark tool: XCheck

XCheck [1] provides a convenient integrated platform to run various benchmarks (or experiments in general), each consisting of several individual queries and varying input documents, using different engines. Per benchmark, all experiments are performed by a single invocation of XCheck.

For each benchmark, XCheck iterates over all engines, and for each engine over all document sizes, executing each query of the respective benchmark on the given document size using the given engine. Each individual experiment is repeated $n + 1$ times. The execution times of the first run are neglected ("*warm-up*"). XCheck then calculates the average and standard deviation of the execution times for the remaining n runs.[1] Next to the overall execution time, XCheck allows to collect the breakdown times for document processing, query translation, query execution, and result serialization—as far as provided by the various engines (see Section 4.3.3 for details). Additionally, XCheck collects the sizes of the produced results and (predefined) error messages.

2.2. Customizing XCheck

XCheck can be extended to use new benchmarks and/or new engines by providing the respective information, e.g., the benchmark queries and documents (or the respective generators), in XML configuration files.

While being very convenient and working reliably, XCheck (Version 0.1.3) comes with two limitations that are quite relevant for our experiments.

The first limitation is that XCheck can only handle a single document per query. XCheck replaces the URI in the `fn:doc()` call in the queries to use the requested document (size), and only one URI is replaced. Hence, multi-document experiments/benchmarks cannot easily be run with XCheck. To circumvent this limitation while retaining the multi-document characteristics, we gather the URIs of all documents of each multi-document benchmark (XMach-1, XBench TC/MD & DC/MD) in a single XML file

```
<collection>
 <uri><!-- URI of document 1 --></uri>
 <!-- ... -->
 <uri><!-- URI of document n --></uri>
</collection>
```

and obtain the sequence of documents on-the-fly via the following preamble to each query:

```
let $docs :=
 for $d in doc("collection.xml")//uri/text()
 return doc($d)
```

[1] In our experiments, we use the default: $n = 3$.

The second limitation is the way how XCheck calls the engines to run a query. This is done via a single command line call. While this is sufficient for stand-alone (file-based) XQuery processors, it does not allow for a "natural" usage of XML/XQuery database systems. Firstly, most (XML/XQuery) database systems have a client–server architecture, where the server is running in the background, and each query is executed by calling of the client program that then connects to the server. Secondly, XML/XQuery database systems allow that the files that contain the XML documents need to be read only once, loading the documents into the database. All queries then only need to access the documents as stored in the database, not requiring the parsing of the original document with each individual query.

While having access to the source code of XCheck (it is basically a collection of perl scripts), we decided not to change the code, but rather exploit (mis-use?) some features of the *adapters* that specify for each individual engine how XCheck should execute a query. Next to the actual call to execute the query, XCheck allows for both a pre-processing- and a post-processing-call. All three interfaces are basically simple command line calls of arbitrary executables, parametrized with (the name/location of) the query-file and the document-file. Hence, a simple straightforward solution would be to start the database server in the pre-call, run the query via the client, and finally stop the server with the post-call. However, we think it is not "natural" to start and stop the database server for each individual query execution—let alone the extra overhead/delay that extends the overall execution time of the whole benchmark considerably.

Alternatively, we could start *all* required engines and load *all* used documents prior to starting the XCheck runs. However, this would mean that we: (1) require enough disk space to store all (possibly huge) documents of each benchmark in all databases concurrently, and (2) all database servers are running concurrently—though only one is active at a time, the "idle" ones consume vital memory, which might compromise the behavior of the active one.

Instead, we want to start each database server individually only once per benchmark, engine and document, and leave it running while all queries of the given benchmark are executed on the given document with the given engine. To implement this, we added two empty queries to each benchmark, containing only the comments "`(:StartServer:)`" and "`(:StopServer:)`". Used as first and last query for each benchmark, these queries trigger the pre- and post-calls to start and stop the respective database server. For all other queries, the pre- and post-calls do nothing.

Likewise, we added a query Q0 to all benchmarks, that loads the respective document into the database using the respective database's document loading functionality. Thus, XCheck automatically uses these queries to measure and collect the document loading times. All other benchmark queries then access only the pre-loaded document, just like a database scenario is supposed to work. To save disk space, our post-calls remove the document from the database once they see the respective "`(:StopServer:)`"-query.

3. The benchmarks

In this work, we consider the five most popular publicly available XML/XQuery related benchmarks: MBench [2], X007 [3], XBench [4], XMach-1 [5], XMark [6], as listed in Table 1. All benchmarks consist of a set of queries and provide document generators that allow to generate documents of various sizes.

3.1. Documents

Compiling, installing and running the document generators was no problem with most of the benchmarks. However, some generators required small fixes. For X007 and MBench, the generators required minor (quite obvious) changes to get the source code compiled with gcc/g++ 4.0.2 on Fedora Core 4 (details omitted here). The XBench document generator worked fine for the text-centric documents (TC/SD, TC/MD), but kept crashing with some "obscure" Java exception when trying to generate the document-centric documents (DC/SD, DC/MD) on our Fedora Core 4 (64-bit) systems using Java 1.5.0. Luckily, Loredana Afanasiev could provide us with the generated documents—at least the smaller sizes.

3.2. Queries

Except from XMark, not all queries of the benchmarks were publicly available in a form that complies with the latest XQuery syntax requirements. In fact, 62 out of the total 163 benchmark queries were not available as valid XQuery queries.

Table 1
Benchmarks, their document sizes and number of queries

XOO7 [3] (cfg)		XMark [6] (sf)		MBench [2] (sf)		XMach-1 [5] (# docs)		XBench [4] (size)		TC/SD	TC/MD (# docs)	DC/SD	DC/MD (# docs)	
Small3	2x 4.5 MB	0.001	110 KB											
Small6	2x 8.7 MB	0.01	1.1 MB											
Small9	2x 13 MB	0.1	11 MB	0.1	46 MB	100	2.3 MB	Small		11 MB	26	9.1 MB	2597	9.9 MB
Med3	2x 44 MB	1.0	110 MB	1.0	496 MB	1000	18 MB	Normal		104 MB	266	97 MB	25 925	100 MB
Med6	2x 86 MB	10.0	1.1 GB	10.0	4.8 GB	10000	174 MB	Large		1.1 GB	2666	1.1 GB	259 205 (?)	(?)
Med9	2x 129 MB	100.0	11 GB					Huge		11 GB	26666	16 GB	2 592 005 (?)	(?)
													(Document generator failed/crashed)	
	22		20		46		8	# Queries		17	19	16	15	

Our thanks go to Loredana Afanasiev [7] for making these queries compliant with the latest XQuery requirements,[2] so that they can be processed by most of the considered XQuery processors. Some queries still give syntactical or runtime errors with some of the engines, mainly due to limitations of the respective engines (see Section 6.1 for details).

4. The systems

To be included in our evaluation, XQuery processors need to fulfill three criteria:

(i) free public availability, either in open source, or at least as a binary (evaluation) version;
(ii) running under Linux on an x86_64 (AMD_64) or x86 (Intel i686) platform; and
(iii) supporting (a reasonable subset of) XQuery.

While there might be more systems fulfilling these criteria, we limit our evaluation to the following ones:

Engine		Version	Code	Bits
M	MonetDB/XQuery [8]	0.14.0	C	64
X	X-Hive/DB [9]	7.2.2	Java	64
B	BerkeleyDB/XML [10]	2.2.13	C/C++	64
S	Saxon-B [11]	8.7.1	Java	64
G	Galax [12]	0.6.10	OCaml	64
Q	Qizx/Open [13]	1.0	Java	64

The first three are XML/XQuery database systems, the last three are stand-alone (file-based) XQuery processors.

4.1. Compilation

While all chosen XQuery engines are publicly available, there is no unified way to install all of them. Neither are all engines available as ready-to-run packages suitable for our experimentation platform (cf., Section 6.2), nor are all of them available in open source. In the following, we briefly describe how we installed each engine.

[2] Cf., ⟨http://staff.science.uva.nl/~lafanasi/xcheck/queries.html⟩.

X-Hive/DB, Saxon-B, Qizx/Open: For the three Java-based engines, we use the pre-compiled `.jar` packages, even if the source code is available as well.

The remaining three engines—MonetDB/XQuery, BerkeleyDB/XML, Galax—are all available in open source. Hence, we compiled them optimized for our experimentation platform.

Galax: As recommended on the Galax web-site, we used the GODI installation and configuration tool to automatically compile and install a 64-bit version of Galax from the sources. We used the default optimization switches for the OCaml compiler, as GODI does not allow to change them (easily).

MonetDB/XQuery: We use the 64-bit, 32-bit OIDs binary RPMs as available from MonetDB's SourceForge site. These packages have been compiled with full optimization (`'configure --enable-optimize'`).[3]

BerkeleyDB/XML: We compiled a 64-bit version of the sources using gcc/g++ 4.0.2 and the same optimization switches[3] as with MonetDB/XQuery.

(In fact, we planned to analyze the impact of compiler optimization switches—mainly the difference between the default `-g -O2` and the excessive list used here—as well as the impact of using different compilers, e.g., the Intel compiler, on the various open source systems. However, time and resource limitations forced us to postpone this analysis. We plan to perform it and report the results as soon as time and resources allow it.)

4.2. Configuration

The basic idea is to use the default "*out-of-the-box*" configuration of the systems. We only applied some minor configurations related to memory. Given that we are using a 64-bit machine with 8 GB of main memory (see Section 6.2 for details), we allow the Java virtual machine for all Java-based systems to allocate up to 6 GB of main memory (`'java -mx6144m'`). Using the complete 8 GB is not possible, as some system processes use some memory (probably less than 2 GB, but we did not spend time on finding the maximum we could safely assign to Java; 6 GB seemed to work and be sufficient for moderate document sizes), and our machine is configured to not allow memory allocations that exceed the physical available memory. The latter is an attempt to avoid instable behavior of our machine under excessive virtual memory allocations.

In particular, we did *not* explicitly create any indices with the database systems, though especially X-Hive and BerkeleyDB/XML might benefit significantly from creating the right indices for the right queries (cf., [8]). The main reason for this decision was that we did not have enough in-depth knowledge of all systems to tune all of them equally well, and thus ensure a *fair* comparison. Hence, treating all systems equally with "no tuning at all" seems the fairest approach for now. In fact, investigating the impact/benefit of using indices and further tuning would provide enough material for a separate study of this kind.

4.3. Adapters

XCheck uses simple XML configuration files ("*adapters*") to specify the details how to call each individual engine and how to collect the detailed timing information (if available) from their output.

Saxon-B, *Galax*, *Qizx/Open*: For the stand-alone file-based processors, we use the default adapters that come with XCheck 0.1.3. In the following, we describe our new adapters for the three database engines used here.

4.3.1. Starting servers and loading documents

To model a "realistic" database usage scenario, we decided to start the database server and pre-load the XML documents from the benchmarks' document XML files into the database only once per benchmark and document, not repetitively for each query. To implement this, we exploit the concept of pre- and post-calls in the XCheck adapters. The principle mechanism is sketched in Section 2.2. We now discuss the details for each database engine.

MonetDB/XQuery: Our pre-call script starts the server in daemon mode via

```
'Mserver --set monet_daemon=yes
         --dbinit='module(pathfinder);''.
```

Loading the document simply requires execution of query `count(doc("<doc_uri>")[*])` (Q0), exploiting the document caching feature of MonetDB/XQuery: `fn:doc()` reads the document

[3] `gcc -O6 -fexpensive-optimizations -fomit-frame-pointer -finline-functions -falign-jumps=4 -frerun-cse-after-loop -falign-functions=4 -frerun-loop-opt -falign-loops=4 -funroll-loops`.

from the given URI and stores it in the database.[4] Subsequent `fn:doc()` calls with the same URI avoid reloading the document, unless its timestamp has changed. The respective post-call script stops the server by killing it.

X-Hive/DB: We start and stop the server via

`'XHStartServer'` and `'XHStopServer'`,

respectively. To load the documents, the execution script extracts the document URI from Q0 and calls a small Java program (an adaptation of the sample `StoreDocuments.java` that comes with X-Hive/DB) that stores the document in the database and assigns it the filename from the URI for later reference.

BerkeleyDB/XML: Though also a database system, BerkeleyDB/XML does not have a client/server architecture, but simply works as a stand-alone program that uses a persistent database storage. Hence, our pre-call script only needs to create a database, using the `createContainer` command of the `dbxml` console application. To load documents, we use the `putDocument` command of the `dbxml` console application. Like with X-Hive/DB, we extract the URI from Q0 and assign the filename from the URI for later reference.

4.3.2. Running queries

We use the following commands and respective commandline options to execute XQuery queries (given in a file) with the various engines:

Saxon-B:
'`java -cp saxon8.jar net.sf.saxon.Query`'.

Galax:
'`galax-run -output-xml`'.

Qizx/Open:
'`qizxopen_batch -serial`'.

MonetDB/XQuery:
'`MapiClient -lxquery -oxml`'.

X-Hive/DB:
We adapted the sample `XQuery.java` that comes with the distribution according to our needs.

BerkeleyDB/XML:
We use the `query` and `print` commands of the `dbxml` console application to execute queries.

[4] By default, when loaded implicitly via `doc` (`"<doc_uri>"`), MonetDB/XQuery keeps only documents up to 100 MB persistent in the database; for our experiments, we raise the limit to 20 GB (`--set xquery_cacheMB = 20000`).

4.3.3. Measuring time

While XCheck measures the overall evaluation time for each query execution itself, it can also collect more detailed timing information from the engines. The various engines provide different means to get detailed timing information.

Saxon-B: The default adapter calls `net.sf.saxon.Query` from `saxon8.jar` with the `-t` switch to get information about tree build time ("`Tree built in`"), "`Compilation time`", and "`Execution time`". The former two are taken as document processing time and query translation time, respectively. Query execution time is calculated as difference of `Execution time` and `Compilation time`. Result serialization time is not available for Saxon-B.

Galax: The default adapter calls `galax-run` with the `-monitor-time on` option to get the detailed timing for document processing, query translation, query execution and result serialization. (See also Section 5.2.)

Qizx/Open: The default adapter calls `qizxopen_batch` with the `-tex` switch to get information about "`evaluation time`" and "`display time`" which are used as query execution time and result serialization time, respectively. Document processing time and query translation time are not available for Qizx/Open.

MonetDB/XQuery: We call `MapiClient` with `-T` to get the detailed timings for document processing ("`Shred`"), query translation ("`Trans`"), query execution ("`Query`"), and results serialization ("`Print`"). We also get the total execution time ("`Timer`"), overruling XCheck's measurements.

X-Hive/DB: We add timing statements to the modified sample applications (`StoreDocuments.java` & `XQuery.java`) that measure the times taken by document loading, query translation (`rootLibrary.executeXQuery(theQuery);`), query execution (`result.next();`) and result serialization (`System.out.println(value.toString());`). Like with MonetDB/XQuery, we also measure the overall evaluation time, overruling the measurement done by XCheck.

BerkeleyDB/XML: We call `dbxml` with `-vv` to get the query translation time ("`Optimizer - Finished parse, time taken`") and query execution time ("`Query-Finished query execution, time taken`"). Additionally, we use the `time` prefix with the `putDocument` and `print` commands to get the document processing

Table 2
Which systems provides which timing

	M	X	B	S	G	Q
query *trans*lation	+	*	+	+	+	
query *exec*ution	+	*	+	+	+	+
result *seri*alization	+	*	+		+	+
*docu*ment processing	+	*	+	+	+	
*comm*unication	$T-(tran+exec+seri+docu)$					
*T*otal time	+	*	XCheck			

+: default *: self-made

time and result serialization time, respectively. Alternatively, we also could use the `time` prefix with the `query` command to get the sum of query translation time and query execution time. However, in many cases, it turned out that none of these timings can be trusted (see Section 5.2).

Table 2 summarizes, which engine provides which timing information. For the database engines (MonetDB/XQuery, X-Hive/DB, BerkeleyDB/XML), the document processing time represents only the time required to load, parse and process the original XML document in order to store it in the database. Any costs for accessing the document once it is stored in the database are included in the query processing times and/or communication times (cf., Section 5.2). Please note that the classification of detailed timings given here is based on the semantics we could easily derive from the respective systems' documentation (as far as available). There is no guarantee that all systems use the same semantics and/or definition for these detailed timings.

5. The experiments

Finally, we are ready to run our experiments. To ensure that the measured performance results do indeed make sense, we need to make sure that all engines work properly, with respect to both the actual XQuery processing and the collection of detailed timing information.

5.1. Checking/validating results

Given our extended (purely practical) experience in software testing and validation,[5] our first concerns are not the actual performance results, but rather the question whether the various engines indeed produce the correct results. Unfortunately, the benchmarks do not come with correct results. In fact, this is hardly feasible, given the randomness built into most document generators for good reasons. Moreover, even with correct results provided, validating the actual results requires more than a simple `diff`, as both XQuery semantics and XML specifications allow for some variation that cannot easily be recognized as equivalent. For the smallest document size of each benchmark, we did "by hand" verify "consistency" among the engines, i.e., all engines yield the same result for each query. For the remaining document sizes, we are left with what XCheck offers us. First, XCheck detects engine-specific error messages, using regular expressions given in the adapters. Second, XCheck collects the sizes of the produced query results. Though we do not have the resources to analyze this in detail, a quick comparison reveals that the all engines produce results "of similar size" for all document sizes and each query processed without errors.

5.2. Checking/validating timings

Already shortly after starting the experiments, we noticed that there were some inconsistencies with the detailed timings of some of the systems. Basically, the breakdown timings (document processing, query translation, query execution, result serialization) did not always add up to the total evaluation time. In most cases, the sum was less than the total; we assume that this is due to start-up and communication costs that are not included in the detailed timings. Once the execution time (per query) exceeds one minute (60 s), the detailed timings reported by Galax are so small, that their sum is only a minor fraction of the total execution time. We guess that the respective code is not correct, and hence, consider these timings as unreliable. In our graph in the next section, we hence depict these "missing times" as communication times.

More severe are the cases where the sum of the detailed timings exceed the total times (often significantly). This is mainly the case with BerkeleyDB/XML, regardless which of the alternatives we use to get the query execution times (cf., Section 4.3.3). Apparently, there is some bug in the code that measures the respective times in BerkeleyDB/XML. In the breakdown graphs in the next section, we mark these "excess times" as (void). In the

[5] Cf., <http://monetdb.cwi.nl/TestWeb/>.

Table 3
Errors and error codes used in Figs. 1–18

E01: preceding axis not supported (see Section 6.1)
E02: parsing / (static) typing
- *Qizx/Open*: MBench QA2; X007 Q23; XBench-DC/MD Q4,17; XBench-DC/SD Q17,20; XMach-1 Q3; XMark Q3,11,12,18.

E03: invalid variable reference (see Section 6.1)
E04: fatal XQuery compiler error (see Section 6.1)
E05: materialization out of bounds (see Section 6.1)
E06: out of memory (see Section 6.1)
E07: out of Java heap space
- *Saxon-B*: MBench QS6,J1-4 (496 MB), Q0-A6 (4.8 GB); XBench-TC/MD Q0-19 (≥2666 docs / 1.1 GB); XBench-TC/SD Q0-19 (≥1.1 GB); XMark Q1,4,6,7,18,19 (1.1 GB), Q0-20 (11 GB).
- *Qizx/Open*: MBench QA4 (496 MB), Q0-J2 (4.8 GB); XBench-TC/MD Q0-19 (26666 docs / 16 GB); XBench-TC/SD Q0-19 (11 GB); XMark Q0-20 (11 GB).

E08: segmentation fault
- *BerkeleyDB/XML*: MBench QS12 (≥496 MB).
- *Galax*: X007 Q19 (≥2x 44 MB); MBench QS35 (≥496 MB); XBench-DC/SD Q0-20 (1.1 GB); XMark Q0-20 (≥1.1 GB).

E09: abort
- *BerkeleyDB/XML*: XBench-TC/SD Q2,17,18 (≥1.1 GB); XMark Q7 (≥1.1 GB), Q20 (11 GB).

E10: unknown error/crash (cf., Sec. 6.1)
- *X-Hive/DB*: MBench QS12 (≥496 MB); X007 Q19 (2x 129 MB).
- *BerkeleyDB/XML*: MBench QS1,2,A2,6,QR1-A6 (4.8 GB); X007 Q14,15, Q19 (2x 129 MB); XBench-TC/MD Q19; XBench-TC/SD Q3,6,9,10 (≥1.1 GB); XMach-1 Q3,7; XMark Q6,14 (≥1.1 GB), Q2-4,13,15-19 (11 GB).

E11: cast not applied to a single atomic value
- *Galax*: XMark Q3 (≥1.1 MB). (new since [14])

doc: document loading failed
- *MonetDB/XQuery*: XBench-TC/MD Q1-19 (26666 docs / 16 GB); XBench-TC/SD Q1-19 (11 GB).

DNF: timeout (>1 h)
- *MonetDB/XQuery*: MBench QS12,A4,J1-4 (≥496 MB), QS33 (4.8 GB); XBench-DC/MD Q19 (25925 docs / 100 MB); XBench-TC/MD Q0 (26666 docs / 16 GB; **>24 h**); XBench-TC/SD Q0 (11 GB; **>24 h**); XMark Q10,11,12,19,20 (11 GB).
- *X-Hive/DB*: MBench QA4 (≥496 MB); XBench-TC/SD Q3 (≥1.1 GB); XMach-1 Q7 (10000 docs / 174 MB); XMark Q8,9,11,12 (≥110 MB), Q10,19 (11 GB).
- *BerkeleyDB/XML*: MBench QJ2, QS35,A4,J1,3,4 (≥496 MB); XBench-TC/MD Q18 (26666 docs / 16 GB); XBench-TC/SD Q2-19 (11 GB); XMark Q8,9,11,12 (≥110 MB), Q10 (≥1.1 GB).
- *Saxon-B*: MBench QA2,4 (≥496 MB); XMark Q8-12 (1.1 GB).
- *Galax*: MBench QA4,J1-4, QS12,25,33 (≥496 MB), Q0-A6 (4.8 GB); X007 Q0,6 (2x 129 MB); XBench-TC/MD Q0,2-19 (≥2666 docs / 1.1 GB), Q1 (26666 docs / 16 GB); XBench-TC/SD Q0-19 (≥1.1 GB); XMach-1 Q2,7 (10000 docs / 174 MB).
- *Qizx/Open*: MBench QJ3,4 (≥496 MB); XBench-TC/SD Q3 (≥1.1 GB); XMark Q10 (1.1 GB).

lic: 30-days evaluation license expired
- *X-Hive/DB*: MBench Q0-A6 (11 GB), QJ1-4; XBench-TC/MD Q0-19 (26666 docs / 16 GB); XBench-TC/SD Q0-19 (11 GB).

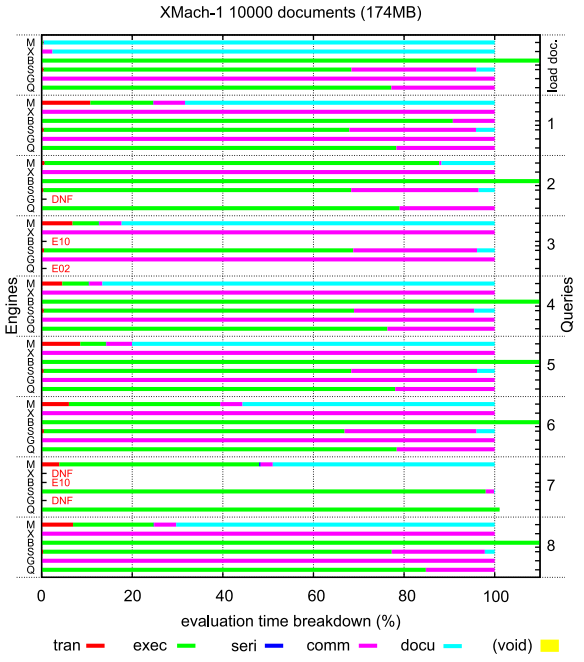

Fig. 1. XMach-1: execution time breakdown.

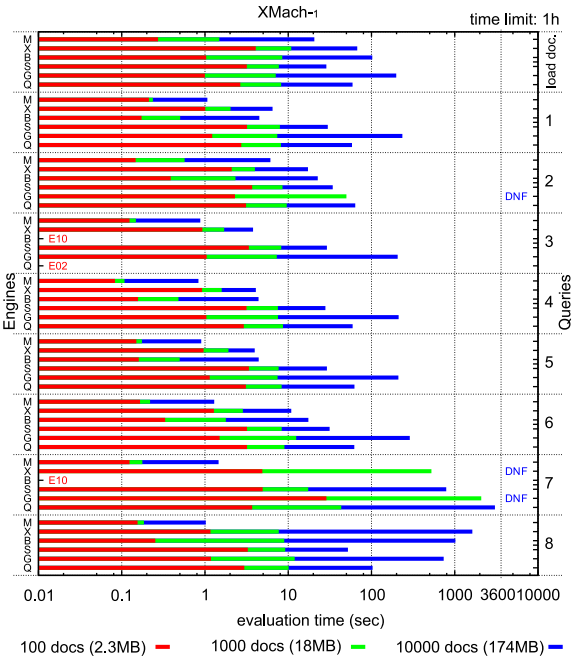

Fig. 2. XMach-1: scalability.

scalability graphs, we use the total times as measured by XCheck.

6. Experimental results

Figs. 1–18[6] present a subset of the performance results that we collected from our exhaustive experiments, running all benchmarks with various document sizes, as depicted in Table 1. To avoid "endless" runs, we limit the execution time for each single query to at most *one hour* (3600 s). Only for loading documents (query Q0), we allow up to *one day* (24 h).

The goal of this work is not to crown a single best system nor to advise users which system to use for their purpose. Rather, we want to provide detailed information and insights such that in particular developers can draw their own conclusions as to whether, where and how to improve their systems. Of course, also users are welcome to draw their own conclusions from the detailed results we provide.

On the *y*-axis of all graphs, we list all queries of the respective benchmark, identified by their number on the right-hand size of each graph. "load doc." identifies the document loading query Q0 as introduced in Sections 2.2 and 4.3. For each query, we list all six engines, identified by their first letter on the left-hand side of each graph. For each benchmark, we show two plots.

Execution time breakdown: For one sample document size per benchmark, the graphs on the left-hand side ("odd" Figs. $1, 3, \ldots, 17$)[6] depict the relative contribution of the detailed timings to the total evaluation time per engine and query. The different sections of the horizontal bars represent the various detailed timings, provided they are reported by the respective engine (cf., Section 4.3.3):

tran:	query translation,
exec:	query execution,
seri:	result serialization,
comm:	communication (cf., Section 5.2),
docu:	document processing,
(void):	wrongly reported times that exceed the actual total times (cf., Section 5.2).

As mentioned in Section 4.3.3, the detailed definition and semantics of these breakdown times are not standardized and can vary between the systems.

Scalability: The graphs on the right-hand side ("even" Figs. $2, 4, \ldots, 18$)[6] depict the total times

[6]A full-color version of this paper is available on-line at <http://www.cwi.nl/htbin/ins1/publications?request=abstract&key=Ma:IS:07>.

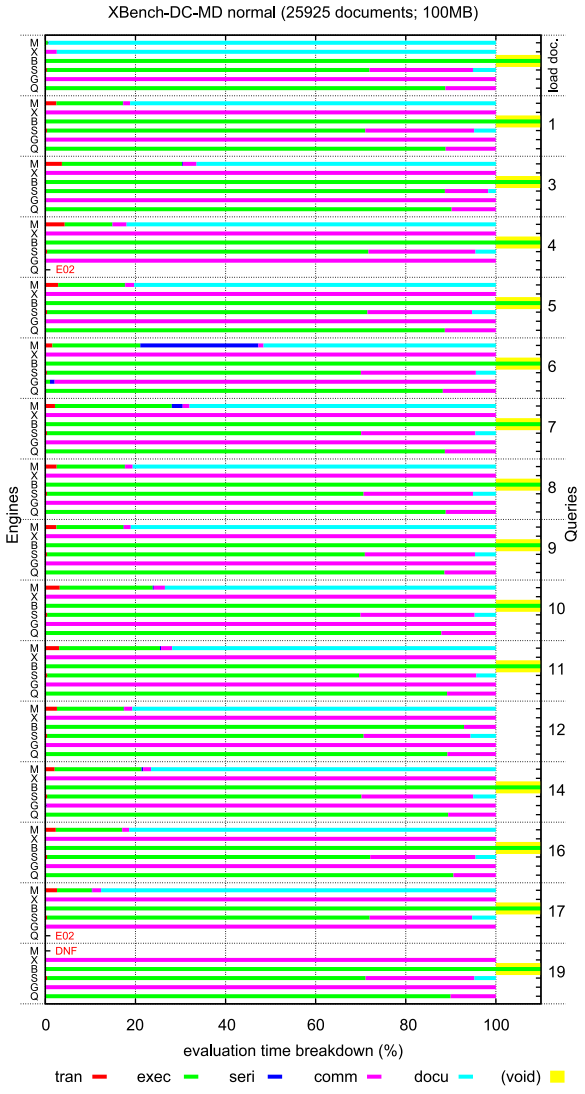

Fig. 3. XBench-DC/MD: execution time breakdown.

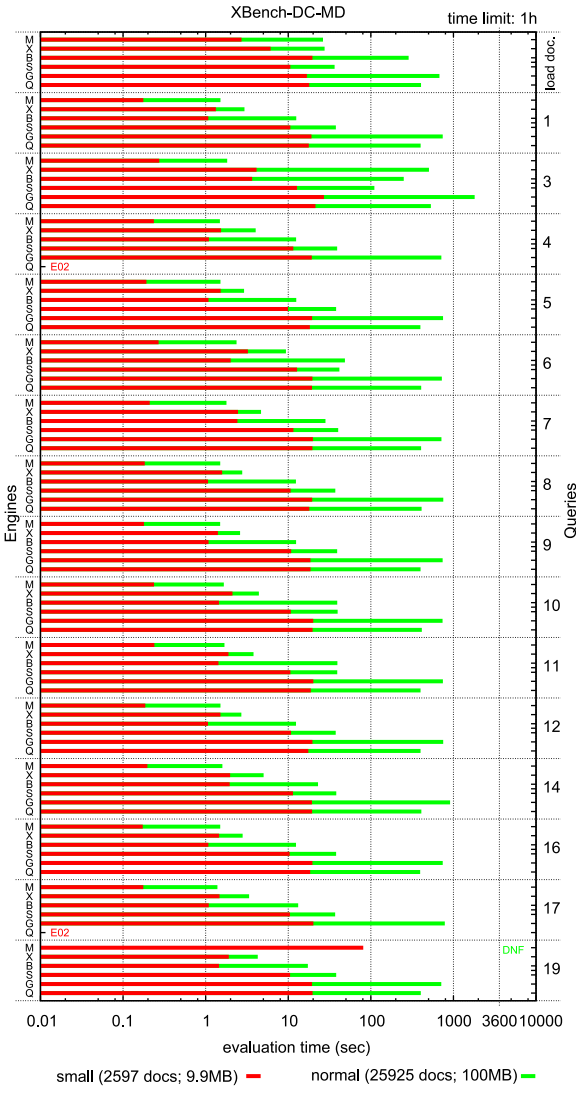

Fig. 4. XBench-DC/MD: scalability.

(wall-clock) for all document sizes of each benchmark. The execution time per engine, query and document is the length of the respective horizontal bar taken from the left margin of the graph, i.e., bars for smaller documents cover the left part of the bars of larger documents. To accommodate the results for all document sizes of one benchmark in one graph, we use a logarithmic scale (decimal base) for the x-axis of the scalability graphs.

All reported results are collected using XCheck as described in the previous sections. In particular, the measured times represent the average of the last three of four consecutive runs, i.e., "hot" results, neglecting the first "warm-up" run (cf., Section 2.1).

6.1. Errors

Some queries fail to execute successfully. Table 3 lists all errors that occur with all our experiments. We use the error codes (E01–E11, doc, DNF) from Table 3 to indicate the errors in Figs. 1–18. In the "scalability" figures, the error codes are depicted in the color of the smallest document size that the respective error occurs with. In various cases, the "unknown error/crash" (E10) could actually be caused by the fact that we kill the respective engine (or client) due to a timeout, in which case they should rather read DNF. However, we did not check this by hand in all cases.

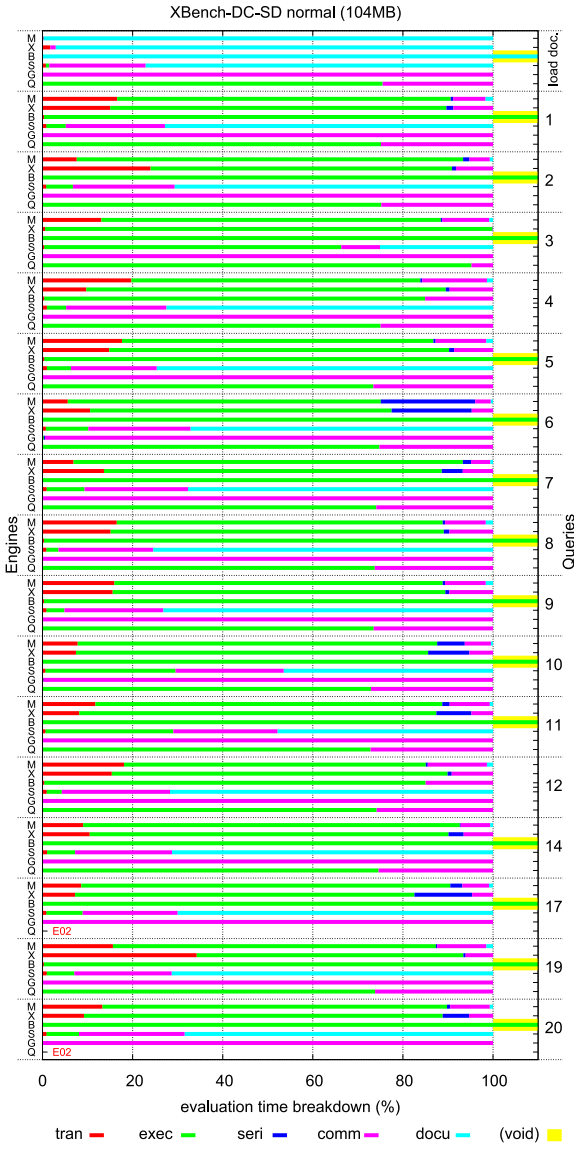

Fig. 5. XBench-DC/SD: execution time breakdown.

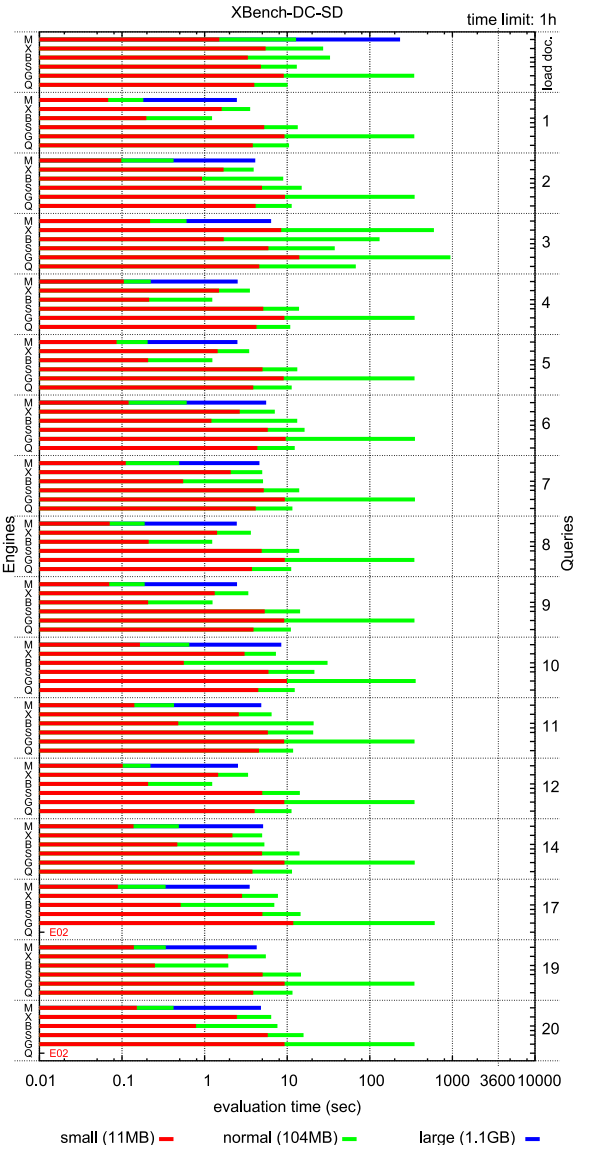

Fig. 6. XBench-DC/SD: scalability.

For ease of comparison, we use the same error codes as in [14], although errors E01, E03–E06 do not occur any more. The respective problems with MonetDB/XQuery 0.10.2 and Galax 0.5.0 (used in [14]) have been fixed in the newer version used here (MonetDB/XQuery 0.14.0 and Galax 0.6.10).

6.2. Hardware and operating system

Our experimentation platform is a dual 1.6 GHz AMD Opteron 242 (1 MB L2 cache) processor with 8 GB RAM and a RAID-5 disk subsystem (3ware 7810, configured with eight 250 GB IDE disks of 7200 RPM). The operating system is Fedora Core 4 (Linux 2.6.14 kernel), using a 64-bit address space. We use gcc/g++ 4.0.2 and Java 1.5.0 (64-bit).

7. Conclusions

First of all, we can conclude that our exercise demonstrates the feasibility of such an extensive and detailed experiment—though is requires quite some work, time, and resources. We describe our experimental setup in detail and explain, how we tackle

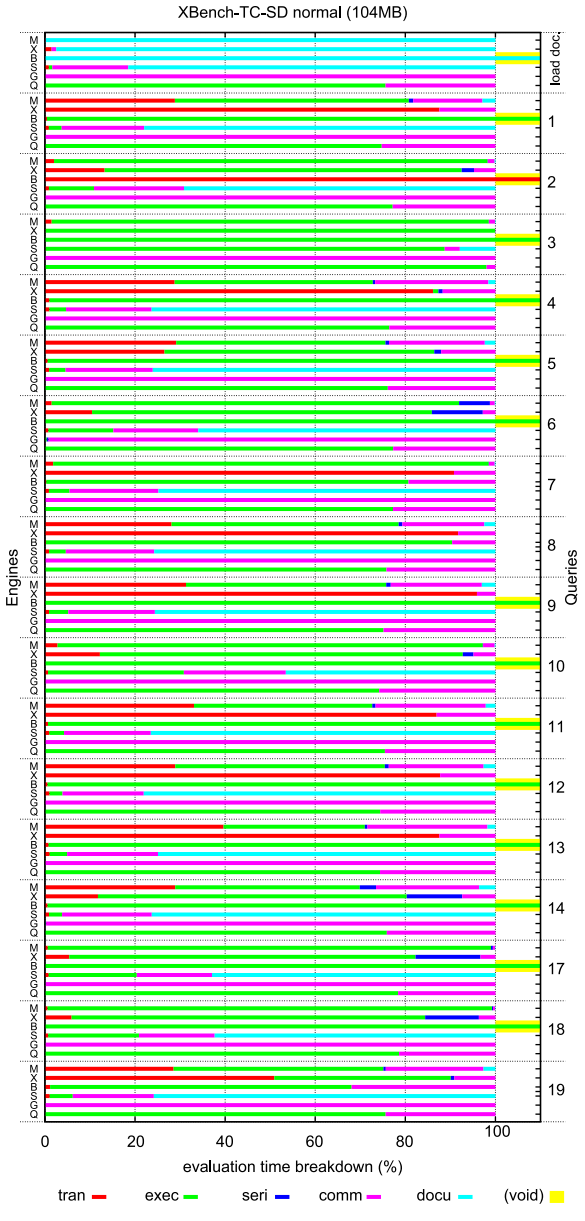

Fig. 7. XBench-TC/SD: execution time breakdown.

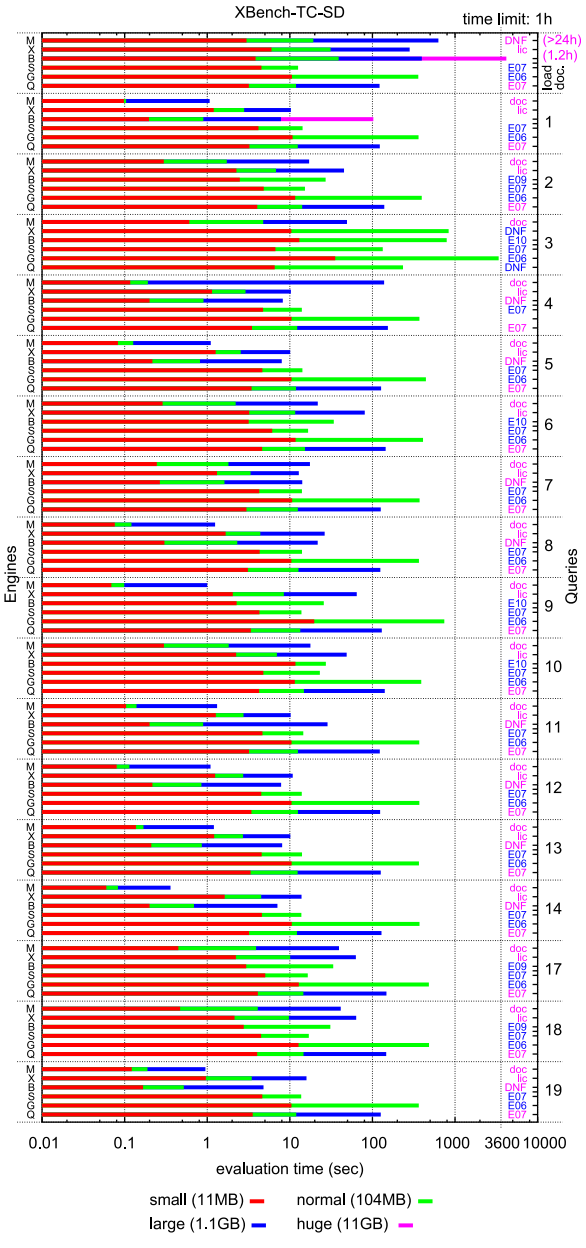

Fig. 8. XBench-TC/SD: scalability.

various problems to reach our ambitious goal. We hope, we provide all information that is necessary and sufficient to reproduce our results.

As expected, the actual performance results do not crown a single winner. However, some general trends can be observed. In a realistic database scenario, i.e., with the documents pre-loaded in the database, the database engines perform considerably better (up to two orders of magnitude) than the file-based stand-alone systems. Even if we add the initial document loading times, they are often still faster, but hardly ever slower. In pure document loading performance (query Q0), MonetDB/XQuery, Saxon-B and Qizx/Open lead the race neck-to-neck; X-Hive/DB and BerkeleyDB/XML follow within a factor 2–3; Galax runs about factor 10 behind the leaders. While all systems have their strengths and weaknesses, MonetDB/XQuery seems to be ahead of the pack in most cases, usually followed (in that order) by BerkeleyDB/XML,

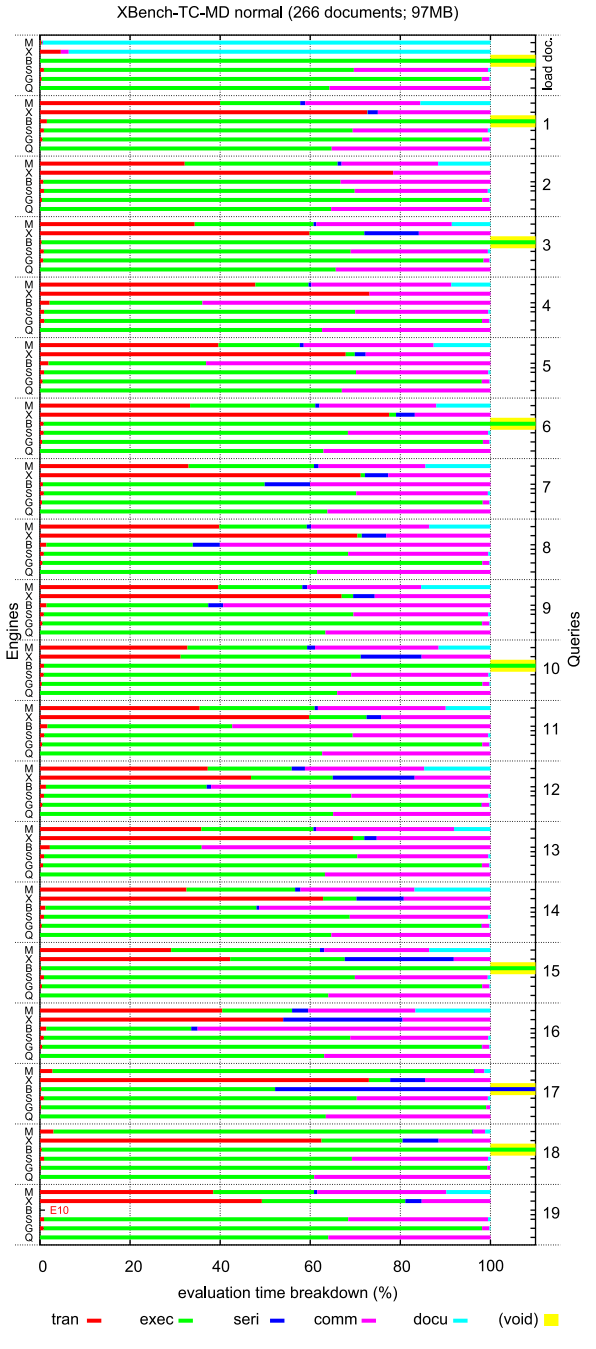

Fig. 9. XBench-TC/MD: execution time breakdown.

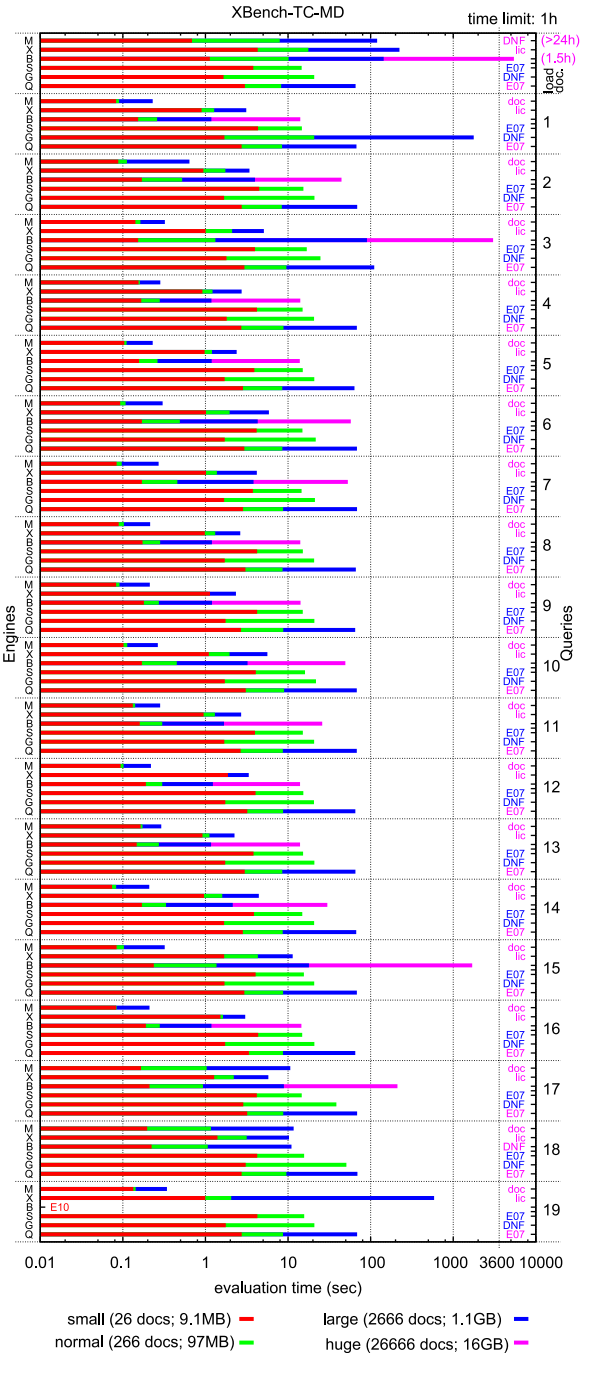

Fig. 10. XBench-TC/MD: scalability.

X-Hive/DB, Qizx/Open, Saxon-B, and finally Galax. Join recognition and processing (still) seems to be the biggest challenge to be solved.

Compared to [14], we upgraded two systems to newer versions. For MonetDB/XQuery, version 0.10.2 has been replaced by version 0.14.0, and for Galax, version 0.5.0 (pre-compiled 32-bit binary) has been replaced by version 0.6.10, which we compiled from the source into a 64-bit binary (cf., Section 4.1). For single-document benchmarks, the performance of MonetDB/XQuery 0.14.0 is within 20% of the performance of MonetDB/XQuery 0.10.2. With multi-document benchmarks, however, MonetDB/XQuery 0.14.0 shows an improvement of up to three

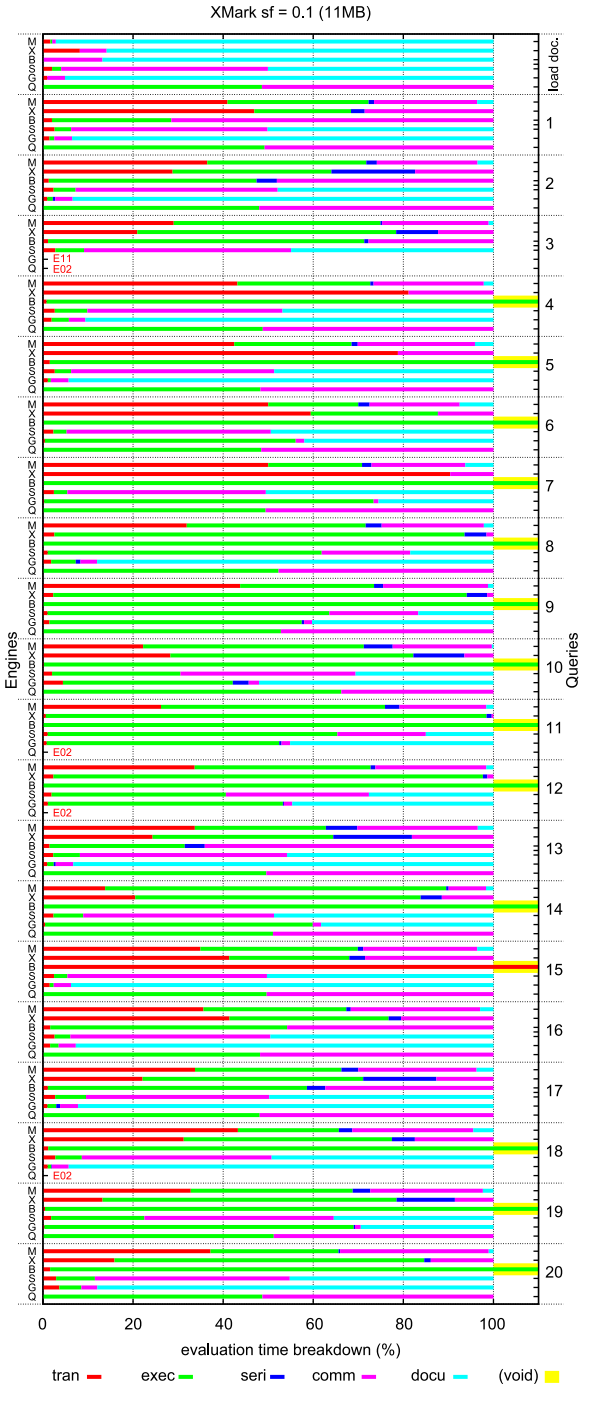

Fig. 11. XMark: execution time breakdown.

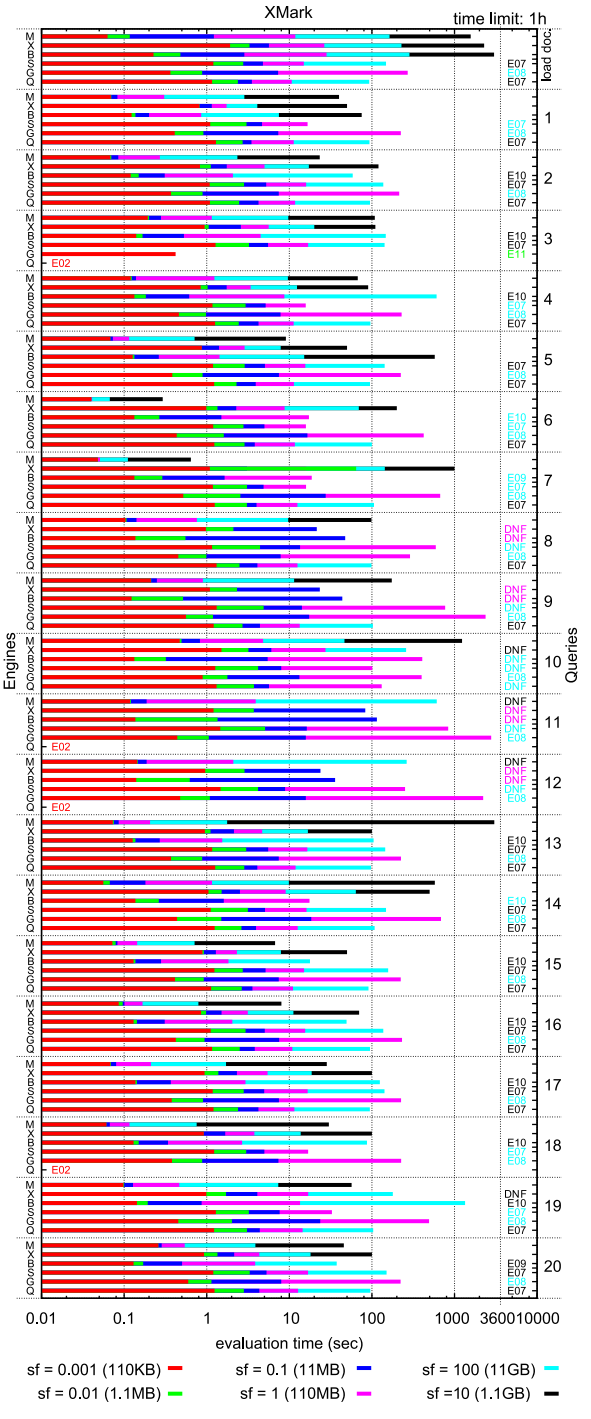

Fig. 12. XMark: scalability.

orders of magnitude. The new (64-bit) version of Galax runs much more stable and with much less memory problems than the old (32-bit) one. In particular, it is now able to successfully run the XMark join queries (Q8,9,11,12) on documents of up to 110 MB (sf = 1). However, overall scalability has hardly improved, and overall performance has dropped by a factor 2–5.

For the future, we plan to extend the scenario in various dimensions, e.g., including more systems

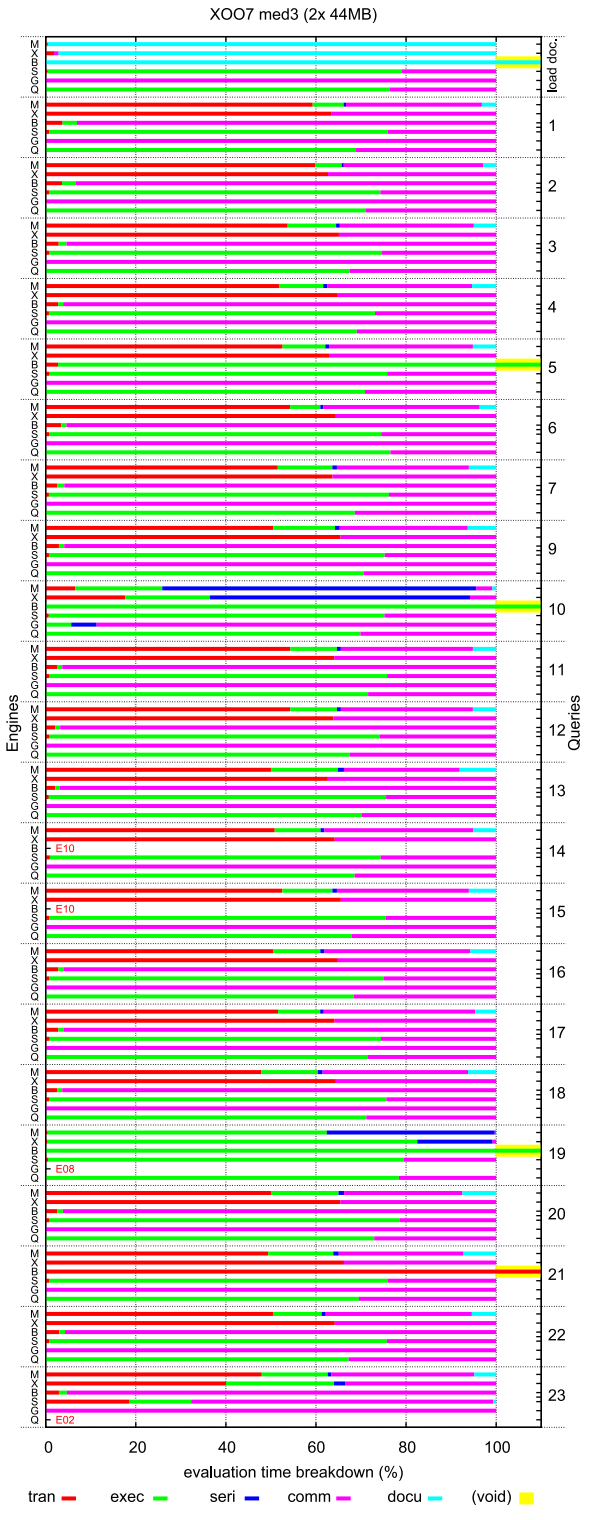

Fig. 13. X007: execution time breakdown.

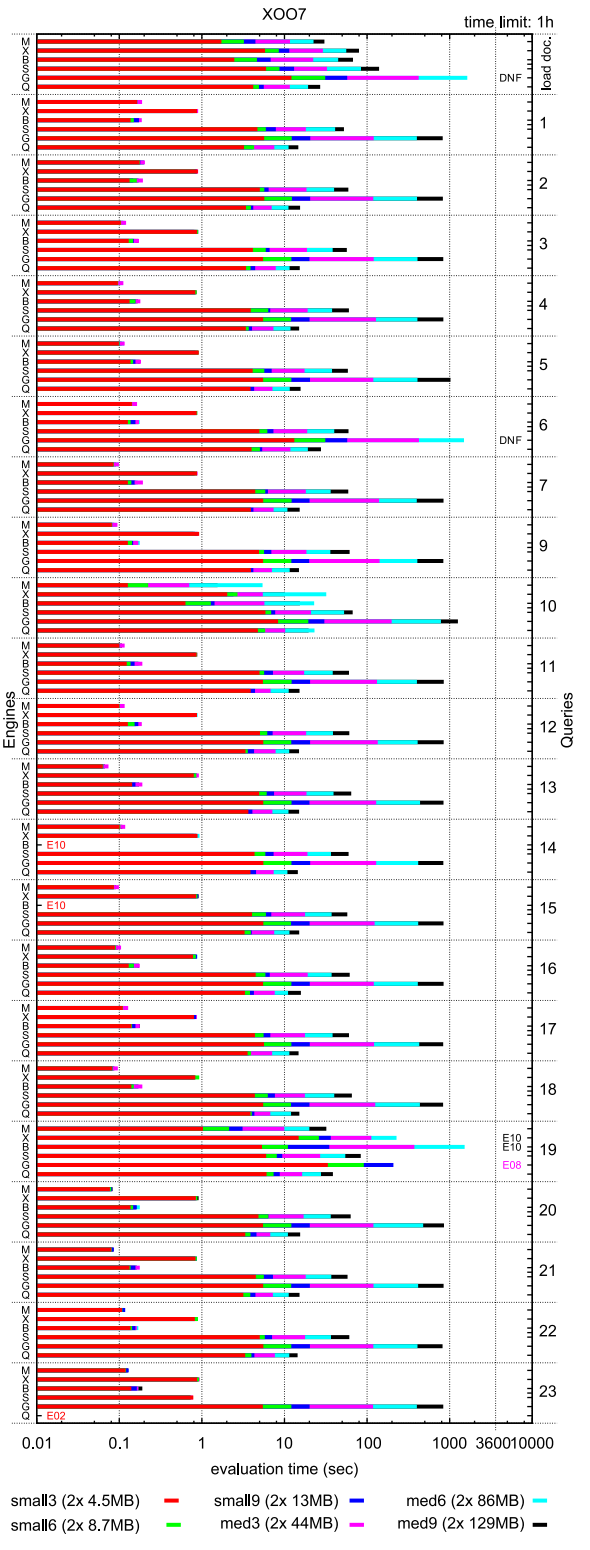

Fig. 14. XOO7: scalability.

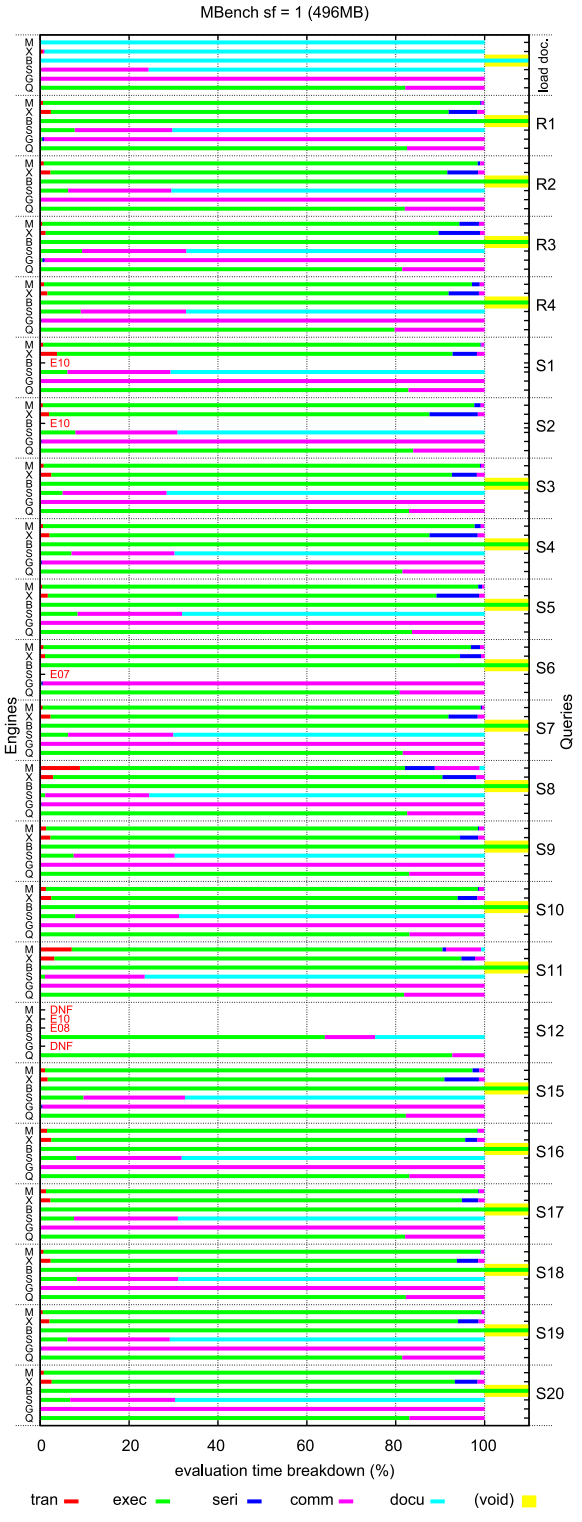

Fig. 15. MBench: execution time breakdown (1/2).

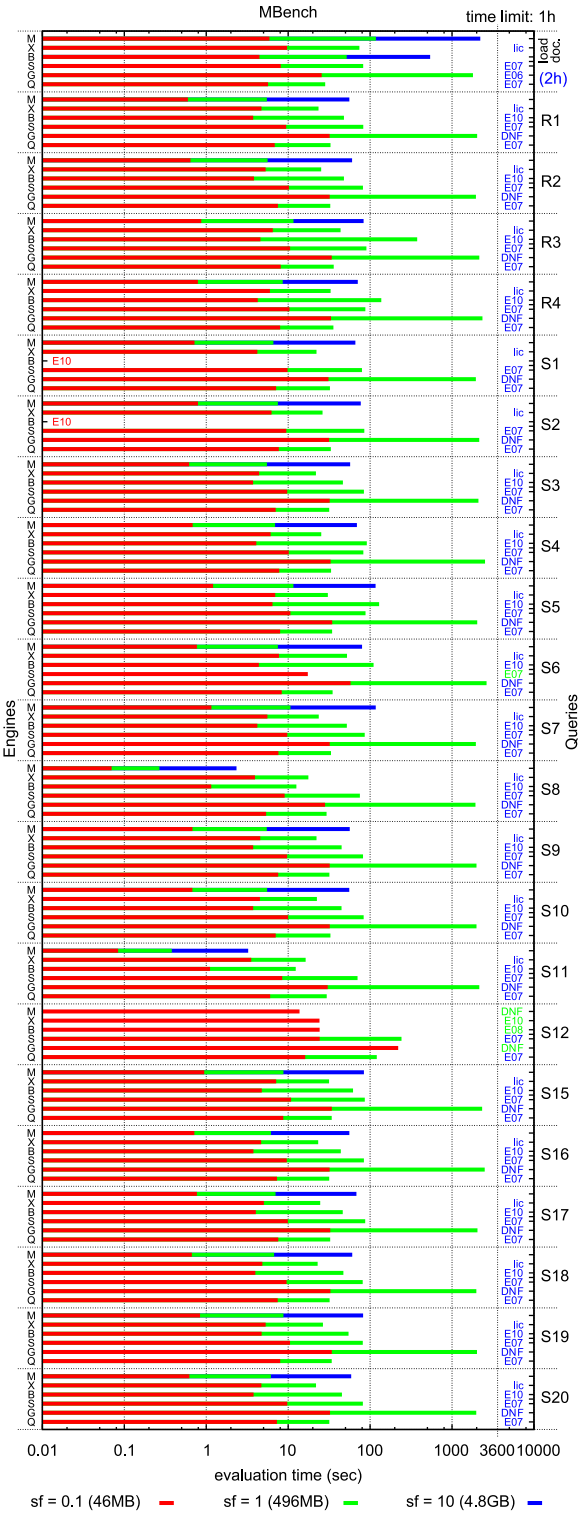

Fig. 16. MBench: scalability (1/2).

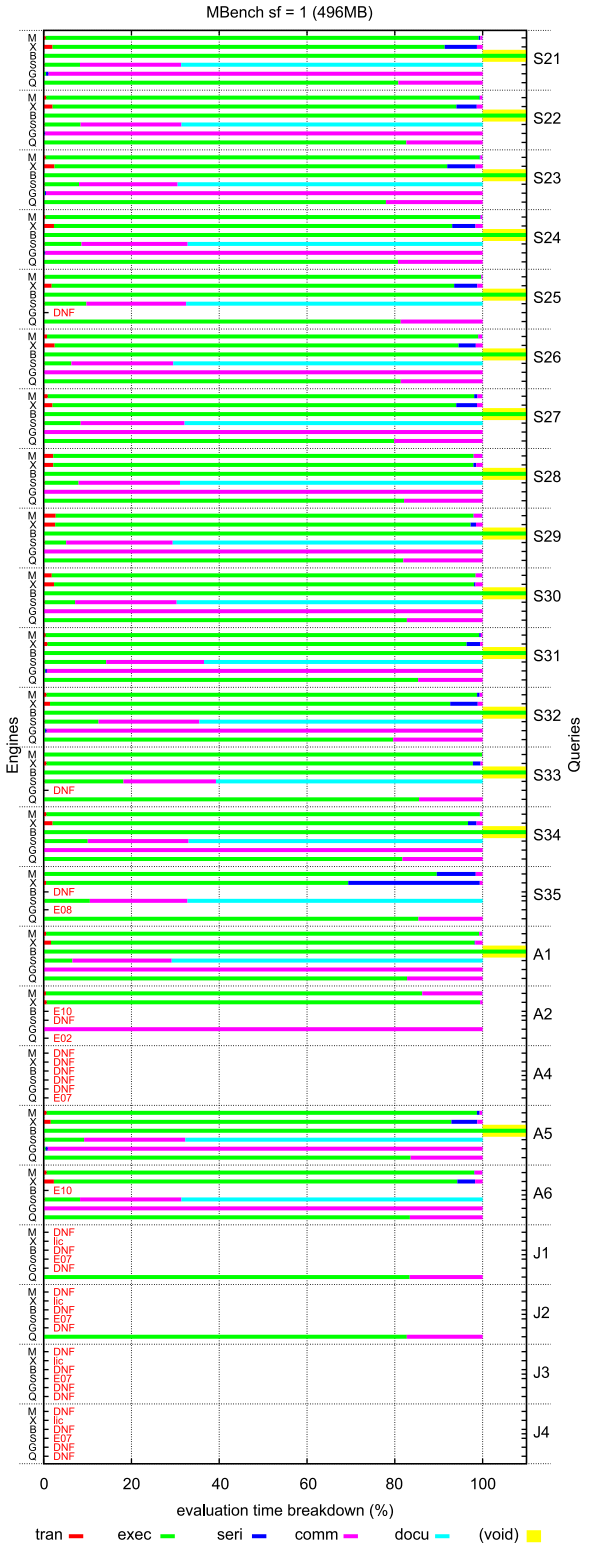

Fig. 17. MBench: execution time breakdown (2/2).

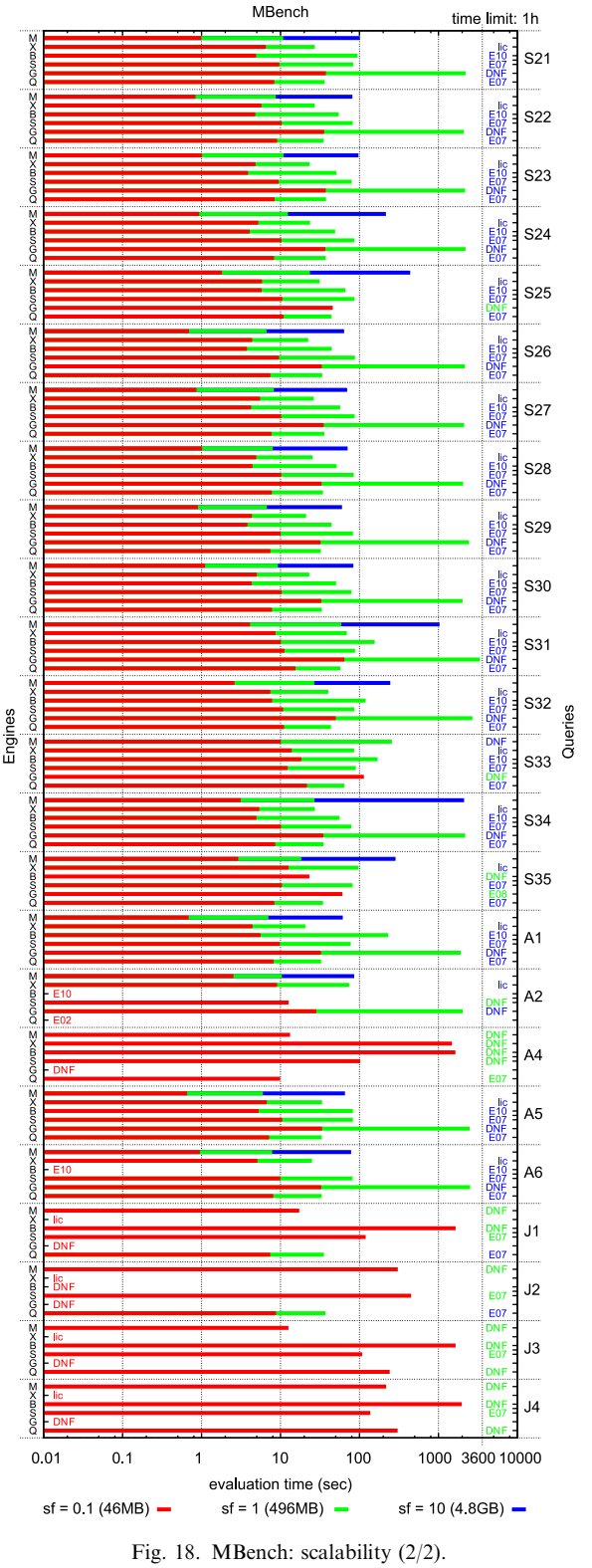

Fig. 18. MBench: scalability (2/2).

(e.g., eXist [15]) and benchmarks (e.g., XPathMark [16], MemBeR [17]), considering other compilers and optimization flags, using different hardware and operating systems, etc. The goal is not to find the single best setup, but rather to show that all these often neglected factors can influence experimental results considerably. Hence, all reports of experimental results should reveal all these information explicitly in order to: (1) provide all information to make the results reproducible, and (2) to put them in the right perspective. Finally, we hope that our experiences with XCheck will help to improve and extend this convenient experimentation tool.

References

[1] L. Afanasiev, M. Franceschet, M. Marx, E. Zimuel, XCheck: a platform for benchmarking XQuery engines, in: VLDB, 2006, demo. ⟨http://ilps.science.uva.nl/Resources/XCheck/⟩.
[2] K. Runapongsa, J. Patel, H. Jagadish, Y. Chen, S. Al-Khalifa, The Michigan benchmark: a microbenchmark for XML query processing systems, in: EEXTT, 2002, ⟨http://www.eecs.umich.edu/db/mbench/⟩.
[3] S. Bressan, G. Dobbie, Z. Lacroix, M. Lee, Y. Li, U. Nambiar, B. Wadhwa, X007: applying 007 benchmark to XML query processing tool, in: CIKM, 2001, ⟨http://www.comp.nus.edu.sg/~ebh/XOO7.html⟩.
[4] B. Yao, T. Özsu, N. Khandelwal, XBench benchmark and performance testing of XML DBMSs, in: ICDE, 2004, ⟨http://se.uwaterloo.ca/~ddbms/projects/xbench/⟩.
[5] T. Böhme, E. Rahm, XMach-1: a benchmark for XML data management, in: BTW, 2001, ⟨http://dbs.uni-leipzig.de/de/projekte/XML/XmlBenchmarking.html⟩.
[6] A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, R. Busse, XMark: a benchmark for XML data management, in: VLDB, 2002, ⟨http://xml-benchmark.org/⟩.
[7] L. Afanasiev, M. Marx, An analysis of the current XQuery benchmarks, in: ExpDB, 2006.
[8] P.A. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, J. Teubner, MonetDB/XQuery: a fast XQuery processor powered by a relational engine, in: SIGMOD, 2006, ⟨http://monetdb-xquery.org/⟩.
[9] X-Hive/DB, ⟨http://www.x-hive.com/products/db/⟩.
[10] Berkeley DB, XML, ⟨http://www.sleepycat.com/products/bdbxml.html⟩.
[11] Saxon-B, ⟨http://saxon.sourceforge.net/⟩.
[12] M. Fernández, J. Siméon, B. Choi, A. Marian, G. Sur, Implementing XQuery 1.0: the Galax experience, in: VLDB, 2003, ⟨http://www.galaxquery.org/⟩.
[13] Qizx/Open, ⟨http://www.axyana.com/qizxopen/⟩.
[14] S. Manegold, An empirical evaluation of XQuery processors, in: ExpDB, 2006, ⟨http://www.cwi.nl/htbin/ins1/publications?request=abstract&key=Ma:EXPDB:06⟩.
[15] eXist, ⟨http://exist.sourceforge.net/⟩.
[16] M. Franceschet, XPathMark: an XPath benchmark for XMark generated data, in: XSym, 2005.
[17] L. Afanasiev, I. Manolescu, P. Michiels, MemBeR: a micro-benchmark repository for XQuery, XSym, 2005.