Benchmark

7

Abstract

All authors of persistent systems claim that their systems are super fast. This chapter compares the performance of ten major persistent systems on a benchmark which involves up to one million books and a many-to-many relation between books and their authors. The books can be with or without abstracts. The results are most interesting and intriguing.

Keywords

Performance • Testing • Persistence • Benchmark • Persistent system • Persistent data structures • DOL • Memory blasting • PPF • Boost • PSE Pro • SQLite • Java serialization • C# serialization • QSP • Post++

7.1 History of this Benchmark

We, the authors, met for the first time at the Department of Biochemistry, South Bohemia University, Nové Hrady, Czech Republic, when discussing the architecture of software[1] for processing the output of liquid mass spectrometers.[2] In this project, 2 GB of data must by restructured, stored and analyzed by complex algorithms on standard PC hardware within 3 minutes. The calculation involves removal of random noise and conversion of the raw data into a spectrum of peaks. The work is supported by E.U. and is still in progress at the time of writing.[3]

Each of us comes from an opposite corner of the programming profession. Petr is a young application programmer who is always on lookout for new, better ways of

WARNING: Comparing times shown in this chapter without considering features of each system as discussed in the previous chapters may lead to a wrong conclusion about what is the "best" persistent system—if there is such a thing.

[1] Urban et al. (2009, 2012).

[2] http://en.wikipedia.org/wiki/Liquid_chromatography-mass_spectrometry

[3] E.U. grant CENAQUA CZ1.05/2.1.00/01.0024.

J. Soukup and P. Macháček, *Serialization and Persistent Objects*,
DOI 10.1007/978-3-642-39323-5_7,

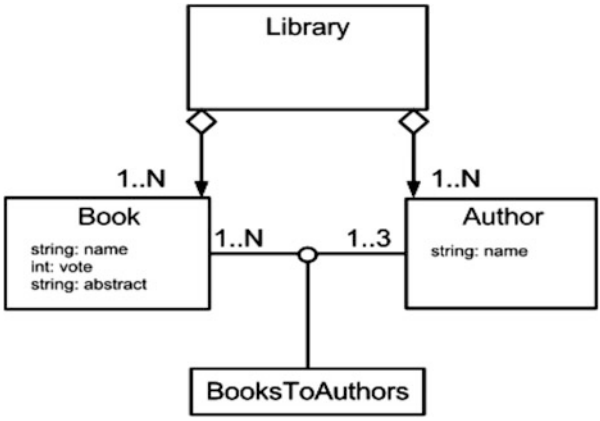

Fig. 7.1 ER diagram of the benchmark. Except for small aesthetic differences, also the UML class diagram

programming and Jiri spent the past 24 years designing tools for a new, more efficient software design, with emphasis on automatic persistence and advanced class libraries.

Later on, when we began to work on this book, we discussed how to compare the performance of the various persistent systems, and we decided that Jiri should not be involved in their evaluation; it would be difficult for him to remain unbiased. All the code, testing, and most of the text in this chapter, was produced by Petr.

Our first benchmark was based on the mass spectrometer project, which used numerous bi-directional associations (one-to-many and many-to-many) that are not supported by any standard library. Except for DOL and PPF, running this benchmark with the existing persistent system required extensive custom modifications—essentially replacing all the bi-directional associations by several containers or pointers. We planned to test about ten typical persistent system, and doing everything ten times was not only beyond our capacity—the testing would depend very much on how we would modify each system.

We eventually decided to strip the benchmark to the bare bones as described in Sect. 7.2 and Fig. 7.1, using class names that would be more familiar to the average reader. We believe that the ManyToMany association should be in the benchmark in order to reflect the complexity of real-life projects, and that blocks of unstructured text (here book abstracts) often occur in practical problems.

We also added a step in which one quarter of the objects is removed. We believe that removal of objects is critical in many applications, and should be part of any benchmark.

The steps which we observed are the same as in our original benchmark. We test the time to build the data organization, traverse and sort them, search, remove data, store them to disk, and then—in a separate run, retrieve them from the disk. We also observe the size of the disk file and check the integrity of the data.

When displaying the results graphically, we ran into the problem of numbers falling into a wide range where linear graphs show clearly which programs are the worst rather than which are the best. In such cases, we use logarithmic scale. When analyzing the results, always watch for what type of scale is used! It may appear that one system is twice as fast as another, but when you interpret the scale correctly you see it is ten times faster.

7.2 Persistent Systems Tested

In alphabetic order:

- Boost C++ library with STL containers or Boost data structures
- C# serialization, as implemented in .NET
- DOL (Data Object Library) library of persistent data structures (Code Farms)
- Java serialization, as implemented in java.io
- Java serialization combined with InCode data structures
- ObjectStore © PSE Pro for C++ combined with InCode data structures
- POST++ (Persistent Object Storage for C++) with STL containers
- PPF (Persistent Pointer Factory) with InCode data structures (Code Farms)
- QSP (Quasi Single Page) persistence for Objective-C[4] combined with InCode
- SQLite database for both persistence and data relations (no data structures used)

In graphs and tables, we abbreviate the names of some technologies, for example P++ stands for POST++, J for Java, B for BOOST or PSE for "PSE Pro"

Since persistent objects are an alternative to using a database, we included SQLite in our testing. It is a relational database which requires one either to match the benchmark data structures to the database format or to forget about object-oriented programming and remain in the realm of relational thinking—essentially to replace all the data structures by the database.[5]

The InCode library includes bi-directional associations, and we used it in environments where the persistent system does not support such associations. Using the same library helps to reduce dependency of the results on the implementation of the data structures.

We find the tables and graphs presented below most interesting. However, when comparing individual technologies, it would be a grave mistake to look just at the speed of the processing or the size of the output files. The performance is important, but sometimes the flexibility to support evolving software or the ability to transfer data between different operating systems may outweigh the performance. For the features of individual products see Tables 7.1 and 7.2.

The value of measuring performance is twofold:

1. It tells us how the different approaches to persistence, rather than the individual products compare to each other: serialization vs. working with memory pages, XML serialization vs. binary serialization or primary data storage in memory vs. on disk.
2. If you choose any particular product because of the features it offers, the performance results tell you what penalty you are going to pay for those features.

Notes

– We also wondered whether there should be a column for integrity checking, but then we decided not to include it. Integrity checking is usually performed by the

[4] The prototype described in Sects. 2.5 and 6.5.

[5] This is what we did when testing with SQLite.

Table 7.1 Persistent systems and the features they offer—look for more explanation in the text

	Change or add to class/ alloc (a)	Persistence needs code generator (b)	Reuse of free objects (c)	Schema evolution/ portable data (d)	Pointers soft/hard (e)	Loads only data actually needed (f)	Trans-actions (g)
DOL (3 modes)	+class	Y	free list OS	Y/Y	hard		
PPF +InCode	+class		free list		soft	Y	
Java +InCode	+class		OS		hard		
POST++	+class		memory manager	limited/N	hard	Y	Y
Java serialization	+class		OS	Y/Y	hard		
C# serialization	+class		OS	Y/Y	hard		
PSE Pro	+alloc	Y	memory manager	Y/N	hard	Y	ACID
Boost	+class		OS	Y/Y	hard		
SQLite	DB		OS	Y/Y	soft	Y	ACID
Objective-C QSP	+class		memory manager		hard		

Table 7.2 Persistent systems and class libraries

	Data structure libraries
DOL (3 modes)	Integrated with an extensive library of intrusive and bi-directional associations
	Other libraries: would have to be re-coded
PPF	Has no special library, works with InCode (only some classes so far)
	Other libraries: replace pointers by the PPF smart pointer
Java serialization	Designed to work with Java Collections
	Other libraries: may be used without conversion
POST++	Integrated with JudyLibrary, STL with serious limitations
	Other libraries: pointers must be registered by a special statement
C# serialization	Designed to work with Java Collections
	Other libraries: may be used without conversion
PSE Pro	Has its own collection classes and a version of STL
	Other libraries: all allocation calls must be converted
Boost	Integrated with the extensive Boost libraries and STL
	Other libraries: user must code serialization methods
SQLite	Does not have data structures, only a limited choice of relations
	Other libraries: cannot be used
Objective-C QSP	Has no special library, works with InCode (only some classes so far)
	Other libraries: register pointers as in POST

data structure library, not by the persistence. Libraries that come with DOL and InCode do provide integrity checking.
– We did not test Objective-C with Archiving because it would take too much work to prepare the benchmark in that style.

Columns in Table 7.1

Column (a) When making a program persistent, most systems require the application to add something to its classes (to make classes persistent). That does not apply to SQLite which is a database. PSE Pro does not require any additions to the classes; instead, all allocation calls within the application code must be modified.

Column (b) Usually when code generator is used it implies a simpler user interface. InCode library always uses a code generator, but that is for data structures, not for the persistence.

Column (c) Serializations usually leave the management of free objects to the operating system. All the systems based on memory paging manage free objects, but we suspect that sophistication and performance differs significantly from one system to another. The benchmark only checks whether the removal of some objects will reduce the disk file which is an indication that some memory management is in place.

Column (d) This column combines two related features: support for schema evolution[6] and the ability to transfer data between different environments, e.g. between Windows and Unix. In DOL, it is the ASCII mode that supports schema evolution. In Java serialization and C# serialization the XML format supports it.

Column (e) Hard pointers are traversed at the same speed as if you don't use persistent objects. Soft pointers perform some arithmetic on each dereference. Note, however, that the benchmark results for PPF later in this chapter are surprisingly good in spite of using soft pointers.

Column (f) When processing certain types of data, for example in reservation systems, we need the ability to work with only a small subset of the data. In other situations such as VLSI CAD systems, all the data is needed in memory.

Column (g) As defined in Chap. 1, the scope of this book does not include multi-user systems. However, in some application, support of transactions is a bonus. ACID stands for *Atomicity*, *Consistency*, *Isolation*, *Durability*.

7.3 Description of the Benchmark

The benchmark includes classes Library, Book, Author and BooksToAuthors; see Fig. 7.1.

[6] Sometimes also called *schema migration*.

Class *Book* has three non-structural members[7]:

- **name** is randomly generated title of the book in format "XX book N", where XX are two random ASCII characters and N is a random number.
- **vote** is the number of votes by the readers, random integer.
- **abstract** is a text of random length, max. 512 characters.

Class *Author* has one non-structural member:

- **name** is randomly generated similar name of the author, in format "XX author N".

Class *BooksToAuthors* represents the link in the ManyToMany relation, and it has only structural members.

There are 5-times fewer Authors than Books. A Book can have up to three Authors, an Author can have any number of Books. The Books in the Library are sorted by vote. In real application, Books would likely be stored in a dictionary indexed by the Book's name. Considering the tests we performed, this was not necessary.

As we expected, participants questioned the usefulness of the benchmark and whether it reflects the characteristics of real life projects. The fact that the performance of several products were significantly improved (more than an order of magnitude) as the result of this competition—see Sect. 7.9, we believe, is the ultimate proof of its value.

7.4 Monitored Data

We monitored time needed for individual tasks performed in the benchmark. For each number of books, we repeated the run five-times and recorded minimum, maximum and average values. As can be seen from the graphs, the difference between the runs was insignificant (Sects. 7.4, 7.5, 7.7 and Figs. 7.3 and 7.9).

We tested three sizes of problem:

N = 50,000, 250,000 and 1,000,000 books

all three with/without book abstracts.

We did not perform any monitoring or processing involving abstracts. We only wanted to see their influence on the size of the disk file, and possibly on the times of individual tasks due to increased system paging.

Each tests consisted of two runs:

- The first run executed tasks 1–4.
- The second run executed tasks 5–10.

We tested separately the performance for the data with/without abstracts.

Checksums were used to make sure that the data was stored and retrieved without corruption and in full size.

Hardware used for the testing:

- CPU—AMD Phenom II, X4 965 3.4 GHz (4 cores), 64 bit;
- RAM—KINGSTON DDR3 2000 MHz CL9, 6 GB (3 × 2 GB);

[7] Numbers or text, anything but references (or pointers).

Table 7.3 Monitored tasks

No.	Task	Description
1	Create	Time to create the library of N books
2	Sort	Time to sort books by vote[a]
3	Save	Time to save the data to disk
4	FileSize 1	Size of the disk file
5	Open	Time to read the data from disk, swizzle pointers
6	TopVoted	Time needed to find five books with the top votes
7	Traverse	Time to search all books for a substring in their name
8	Delete	Time to remove every fourth book
9	Save 2	Time to save the reduced data
10	FileSize 2	Size of the disk file after the data reduction[b]

[a]Collections in the InCode library are mostly based on intrusive linked lists. Sorting a linked list is a massive and random pointer exercise
[b]Some persistent systems do not reduce the data space

– HDD—MAXTOR DiamondMax 23, SATA II NCQ 7200 rpm 32 MB, 1000 GB;
– OS—Windows 7 Home Premium, 64 bit.

The programs were compiled for ×86 architecture (32 bit). The C++ and C# programs were compiled with VS2010, the Java programs with JDK 1.7.0_02. Objective-C programs were compiled with GNUstep under Windows 7. Additional details will be discussed later.

During testing the benchmark was always the only program running, with all the RAM and CPU at its disposal. Each output file had a unique name combining the technology with the number of books and the repetition index; for example `library_ppf_50000_1.dat`.

7.5 Specifics of Individual Technologies

The performance of a persistent system depends on the implementation of the data structures. We had two choices:

1. To enforce identical data structures for all the persistent systems we tested. [That could produce misleading results for the systems where special data structures are a part of the solution.]
2. To measure each persistent system with the data structures it normally uses. [This is a more realistic overall evaluation.]

We favour approach 2, and for those persistent systems that do not have any specific library of data structures we used the InCode library which, in our opinion, provides the best performance for this type of the problem.

The main issue was the implementation of the ManyToMany association. DOL and InCode libraries already have a generic ManyToMany class. In environments to which InCode has not yet been ported, we implemented ManyToMany as two containers, and tested various combinations as shown in Listing 7.1. Only the best implementation for each technology is shown in the final results.

Note the difference between implementations A and B. Under A there are no references between classes Book and Author in either direction. Under B, such references are used. As we began to test, we quickly found that the B-style is unusable with Java and with C# exporting to XML. We were getting stack overflow caused by the recursive implementation of Java serialization. This was discussed in Sects. 1.5.2, 1.5.3 and 1.5.4 including examples demonstrating the problem.

Listing 7.1 Tested implementations of ManyToMany. Names of containers are generic. For example `HashMap<>` used here is `Dictionary<>` in C#, `HashMap` in Java and `std::map<>` in C++

```
(A1)    class Library {
              HashMap<Book, Vector<Author>> booksToAuthors;
                HashMap<Author, Vector<Book>> authorsToBooks;
        }
(A2)    class Library {
              HashMap<Book, LinkedList<Author>> booksToAuthors;
                HashMap<Author, LinkedList<Book>> authorsToBooks;
        }
(A3)    class Library {
              HashMap<Book, HashSet<Author>> booksToAuthors;
                HashMap<Author, HashSet<Book>> authorsToBooks;
        }
(B1)    class Book {
             Vector<Author> authors;
        }
        class Author {
              Vector<Book> books;
        }
(B2)    class Book {
             LinkedList<Author> authors;
        }
        class Author  {
        LinkedList<Book> books;
        }
(B3)    class Book {
             HashSet<Author> authors;
        }
        class Author {
              HashSet<Book> books;
         }
```

7.6 Benchmark Rules

1. When coding the benchmark, we asked each author or person responsible for the product (participant) to review our design.
2. Participants could submit their own implementations in source so that we could check it and run it on our testing hardware.
3. When several implementations using the same product[8] were available, the best results would be used in the final tables and graphs.
4. When a participant improved his/her system beyond the official version, the improved results would be accepted only if the participant revealed technical details of the improvement.
5. Participants would be continuously informed about the results of others, and about the improvements others decided to share.
6. The competition ran for several months, and there was no time limit. In order to prevent incorrect or inefficient use of the tested systems we encouraged their authors or their support groups to cooperate with us on coding the benchmark.

7.7 Testing Details

7.7.1 Java Serialization

In Java, the best performing implementation used pattern A3. The tables also show the results for the combination of Java serialization with the Java version of InCode library.

The serialization used objects ObjectInputStream and ObjectOutputStream from package java.io.

Many containers in the InCode library are intrusive and create long chains of references. As discussed in Sect. 1.4.2, Java serialization cannot handle this type of data and crashes with stack overflow. As a workaround we expanded InCode containers with a method that explicitly writes to disk all its objects, which is essentially the method whereby Java handles its own containers.

When running on Java virtual machine (JRE) we used parameter Xmx1000m, which allows Java to use 1 GB of RAM.

7.7.2 C# Serialization

The implementation of ManyToMany used pattern A3.

Tests show that C# binary serialization is very slow, especially the de-serialization where the time increases rapidly with the number of books. We have already mentioned in Sect. 1.4.3 that its prime use is in .NET Remoting. It is

[8] For example, using different class libraries or different data structures.

unsuitable for the type of data we have in the benchmark. The results are not included in the book but they are on the website.

Besides the binary serialization, we also tested the XML serialization which, to our surprise, proved to be much faster. We used two styles of formatting the disk data:

- Binary format, invoked by object `BinaryFormatter` in namespace `System.Runtime.Serialization.Formatters`.
- XML format, invoked by object `DataContractSerializer` in namespace `System.Runtime.Serialization`.

Serialization based on `DataContractSerializer` is not fully automatic. The user must add attribute `DataContract()` to every class, and attribute `DataMember()` to every member to be serialized. In order to minimize the disk space we exported all members with one-character names. For example, class Library was exported as XML element L.[9]

7.7.3 DOL (C++)

DOL has its own library of persistent data structures which includes ManyToMany. For the relation between Library and Book we used DoubleCollect, which is a doubly-linked intrusive linked list which protects data integrity.

The benchmark tested all three persistent modes supported by DOL:

- Binary serialization, each object storing its binary image—fast and space efficient.
- ASCII serialization member by member, in a portable format which also supports class changes. The disk file is larger than for the binary serialization.
- Memory blasting[10] which allocates objects from memory pages, which are then stored without looking at individual objects, and is super fast.

7.7.4 PPF (C++)

Originally PPF did not have its own library, but now there is a version of InCode which works with PPF. Note that the prime data storage in PPF is not memory but disk; the data is paged to memory on demand. Pointers are swizzled any time they are dereferenced. This naturally leads to a longer traversal time, but very short time for open and save. Each class has its own file, so there are as many output files as there are classes in the application.

---

[9] This is done by using attribute `DataContract(Name="L")`.

[10] See Sect. 2.2.2; for more details (Soukup 1994, p. 379); how to use it http://www.codefarms.com/docs/dol/index.htm, Sect. 13.2, *Memory management*.

7.7.5 POST++ (C++)

POST++ library allocates data from one large block of memory, and it does not come with data structures required for this benchmark. It can store STL containers but under a rather restricting condition: when opening the disk file, the block must be stored at the same base address where it was before saving to disk. In other words, values of all pointers must remain the same. No swizzling required.

We did not find this approach very practical. For data using 100 MB of memory or more, the mapping to the same address often did not work, and we had to restart the program several times.

The author[11] of POST++ recommends use of an address which is not occupied by other DLLs. It sounds simple, but starting from Windows Vista, operating systems randomize locations of DLLs. This also would not work when transferring data between two different computers that use different DLLs.

ManyToMany was implemented using pattern B3.

7.7.6 SQLite (C++)

Code of this benchmark is quite different from all the other technologies. Instead of data structures such as List or Array, it uses the relational database, with the schema from Fig. 7.2. SQLite supports many features including transactions.[12]

SQLite was set to work as fast as possible by using:
PRAGMA journal_mode = MEMORY[13]
PRAGMA synchronous = OFF[14]

7.7.7 PSE Pro for C++ from ObjectStore (c)

Our benchmark would not be complete without this well established commercial product from the company which on their website claims "performance beyond reach", "world's highest performance" and "fast, instant access". PSE Pro is a single-process, small footprint object database management solution based on memory paging. PSE stands for Personal Storage Edition, and it is a light complement of the main ObjectStore product, which is a full-fledged OODBS, ObjectStore © Enterprise.

[11] Konstantin Knizhnik, Russia.
[12] http://www.sqlite.org/features.html
[13] http://www.sqlite.org/pragma.html#pragma_journal_mode
[14] http://www.sqlite.org/pragma.html#pragma_synchronous

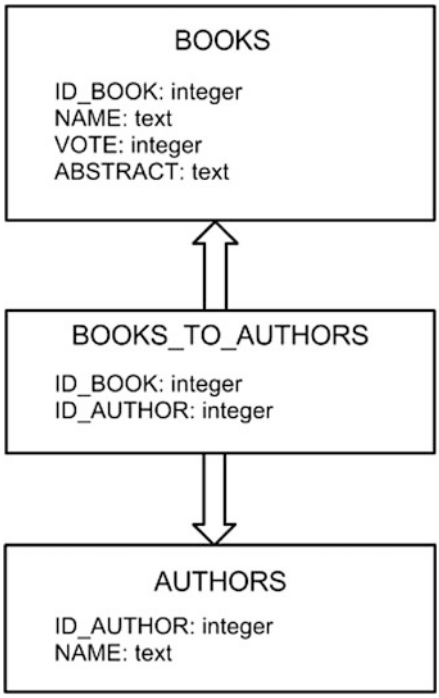

Fig. 7.2 Benchmark schema when using SQLite

As with the other products, we aimed for the best results PSE Pro can produce, and we coded the benchmark jointly with the PSE Pro support group, which guaranteed that PSE Pro was used properly. The PSE Pro license does not allow users to publish results of any benchmarks, but we received a special permit from the company to include PSE Pro in this chapter. Do not confuse PSE Pro with the main product of ObjectStore company, the ObjectStore Enterprise (TM).

7.7.8 BOOST (C++)

We tested both the *binary* serialization and the *text* serialization. The *text* serialization is somewhat similar to DOL ASCII mode, but requires more manual input. The *binary* serialization isn't similar to binary DOL though. BOOST *binary* stores member by member using binary format, while DOL *binary* stores binary images of entire objects without breaking them into members.

As one could expect, binary BOOST was faster than text BOOST. Under "BOOST", graphs and tables in this book show the results of binary serialization. The book website shows results for both serialization styles.

We used BOOST persistence version 1.52.2, and ManyToMany was implemented using pattern A3.

7.7.9 QSP (Objective-C)

Objective-C and its NS library provide a built-in serialization called *Archiving*, which requires so much manual input that it does not fit our definition of automatic persistence—see Sect. 1.4.4.

In Sect. 2.5 we explained new, not previously published persistence based on memory pages, called Quasi-Single-Page persistence, and in Chap. 6 we explained how this new approach can be used for truly automatic persistence in Objective-C.

The QSP benchmark used a prototype[15] of this persistence combined with InCode data structures.

7.8 Results

From the book website www.codefarms.com/book, you can download complete results of the benchmark, with more details than it was possible to show in this chapter. You can view these results either with MS Excel 322 (files with type xls) or with OpenOffice, LibreOffice 325 (files with type ods). You can also download the benchmark implementation with various products and languages, including batch files which compile and run them.

Most of the results that follow are for one million books without abstracts. We do not show results for C# binary serialization because it was so slow that the results were completely out of the range for the other technologies.

Some tables show total times for several tasks, for example for 2(sort) + 6 (topVoted) + 7(traverse). The total is more meaningful when individual technologies use different data structures.

Observation: PPF, and POST++ page disk to memory on demand. PPF is using soft pointers, POST++ uses hard pointers. One would expect that, in traversal, PPF would be slower. Why is it significantly faster in Table 7.4? The only explanation we can think of is that the softness of the PPF smart pointer is only a few arithmetic operations, which may be less overhead than the paging and transaction management performed in P++.

[15] The source is available on the website, but be aware that, except for this benchmark, it has not been used on any serious project yet.

Table 7.4 Overall results for one million books without abstract

Technology/ test	Create (s)	Open (s)	Sort+ TopVoted+ traverse (s)	Save (s)	Delete (s)	FileSize 1 (MB)	FileSize 2 (MB)	Total mean time (s)
DOL (bin)	1.34	18.7	2.17	4.72	0.32	112.71	84.99	31.11
DOL (mb)	2.31	1.20	2.17	0.48	0.20	78.51	78.51	6.85
DOL (ASCII)	1.32	25.09	2.19	11.40	0.33	196.85	152.94	49.29
PPF[a]	3.07	1.30	2.73	0.53	0.54	88.07	88.07	8.67
P++(set)	3.26	3.67	0.65	3.37	0.58	320.26	320.26	14.89
J (set)	14.49	50.12	1.34	33.67	3.79	81.47	63.63	127.63
J (InCode)	11.24	29.51	3.48	22.01	0.32	43.84	34.41	82.05
C# XML (set)	11.92	30.73	5.80	15.19	0.58	644.15	496.23	76.13
SQLite	6.88	0.03	45.69	<0.01	159.48	131.72	131.72	212.08
PSE Pro	2.99	1.51	1.52	1.46	1.39	106.48	106.48	10.65
Boost (bin)[b]	5.63	6.45	0.39	8.86	0.79	84.46	65.48	28.35
ObjC (QSP)	2.66	2.08	1.99	6.98	1.10	133.72	103.39	21.16

Follow footnotes for the stories of products that were significantly improved during the benchmark competition—this table already shows the improved results

[a]The overall time for PPF was reduced 12.5 times by taking advantage of the cache on the modern hard drives. The information was shared with other participants. For full story and technical details, see Sect. 7.9.1

[b]The overall time for Boost was reduced 7.5 times by correcting a performance bug in Boost serialization. For full story and technical details, see Sect. 7.9.2

We also tested the benchmark on Mac[16] and on iPhone[17]. The results are for general interest only; we cannot compare with Tables 7.3 and 7.4 because of the differences in the hardware:
MacBookPro10.2 with Intel Core i5, 2.5 GHz.
APPLE SSD SM128E with 121.33 GB

Observation: In Tables 7.3 and 7.4, as expected, traversal time for technologies based on memory paging is longer. The result is that for intensive algorithms the overall time will become more favourable for technologies that, between `open` and `save`, keep the data in the same memory location.

Konstantin Knizhnik ran the benchmark with POST++ under Linux, using a computer with an SSD which is faster than normal HDD. His result for "Top voted

[16] For the full source see directory bk/chap7/benchApple.

[17] For more information, see Sect. 6.3.

Table 7.5 Overall results for one million books with abstracts

Technology/ test no	Create (s)	Open (s)	Sort+ TopVoted+ traverse (s)	Save (s)	Delete (s)	FileSize 1 (MB)	FileSize 2 (MB)	Total mean time (s)
DOL (bin)	2.17	29.83	2.21	6.38	0.39	380.44	285.78	46.00
DOL (mb)	2.71	3.74	2.49	2.72	0.23	343.01	343.01	14.38
DOL (ASCII)	2.14	38.50	2.21	16.29	0.39	489.48	368.22	72.73
PPF	5.88	4.12	2.86	1.90	0.63	359.29	359.29	16.22
P++(set)	4.40	7.59	6.10	6.90	1.08	679.18	679.18	32.98
J (set)	17.62	59.17	1.47	42.95	6.09	212.04	161.53	157.66
J (InCode)	17.74	35.66	3.62	29.89	0.25	301.59	227.76	105.57
C# XML (set)	13.39	32.57	5.99	15.48	0.60	762.83	585.25	80.30
SQLite	23.64	0.02	50.08	<0.01	182.85	452.42	452.42	256.59
PSE Pro	4.96	4.22	1.52	6.87	2.18	371.40	371.40	28.72
Boost (bin)	6.52	8.15	0.41	9.35	0.82	337.41	255.20	31.97
ObjC (QSP)	3.08	4.71	2.14	10.91	1.14	396.97	300.83	30.75

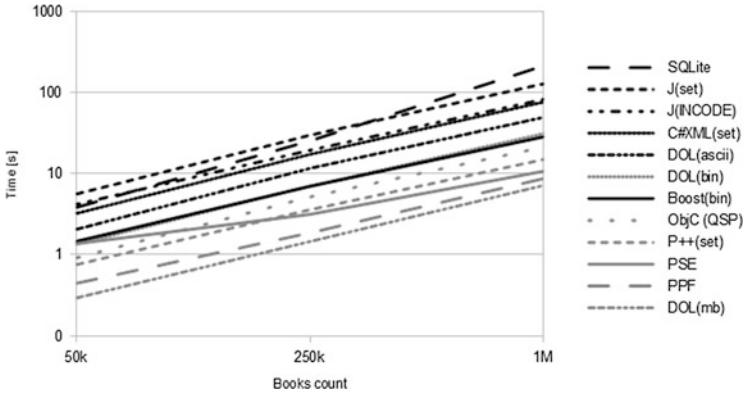

Fig. 7.3 Average total times, without abstracts. When there is no dark top, there was no file reduction

books" and 1 M books with abstracts was 5.6 sec compared to 27 sec in Table 7.5. This can be a rough indication of how much the performance can be improved by using different hardware and operating system (Figs. 7.3, 7.4, 7.5, 7.6, 7.7, 7.8 and 7.9; Tables 7.6 and 7.7).

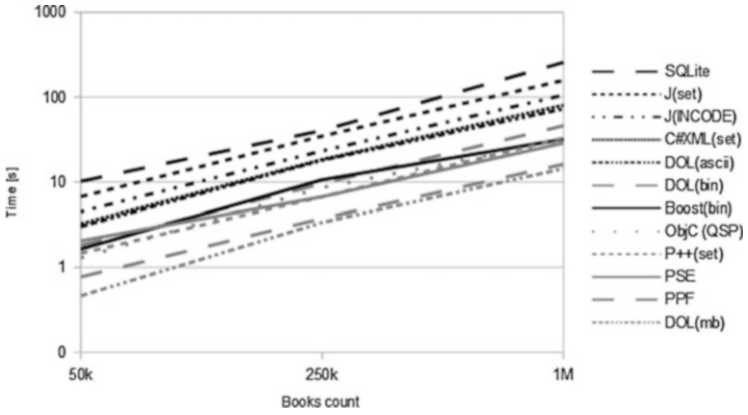

Fig. 7.4 Average total times, with abstracts

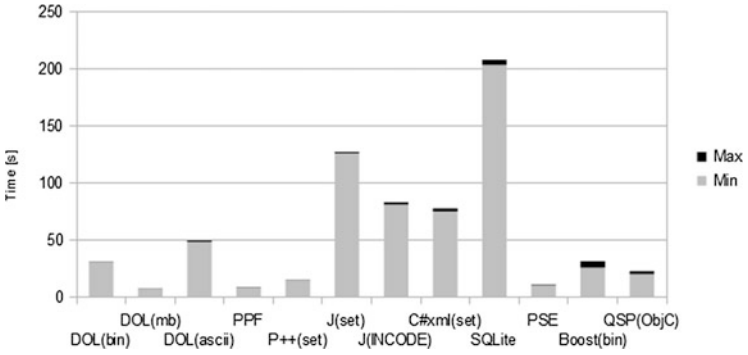

Fig. 7.5 Minimum and maximum total times, one million books, without abstracts

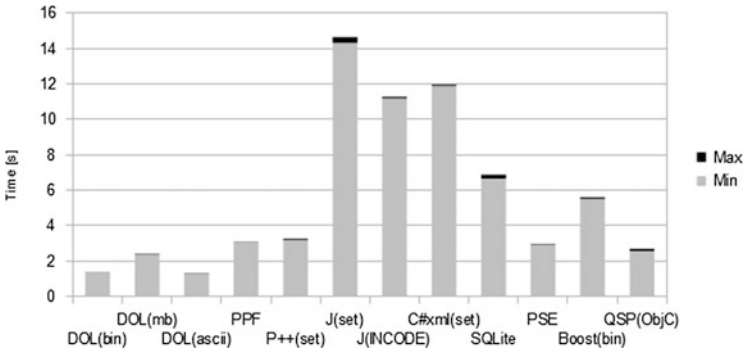

Fig. 7.6 Minimum and maximum times to create data, one million books without abstracts

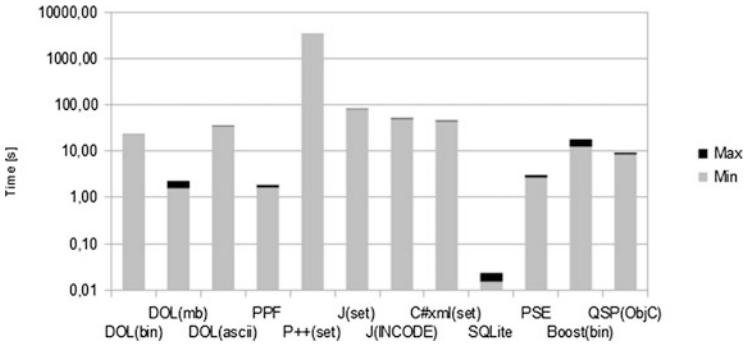

Fig. 7.7 Minimum and maximum times for combined save+open, one million books, without abstracts. Serializations move all the data between the memory and disk on open or save. In technologies based on memory paging, data moving blends with traversing the data

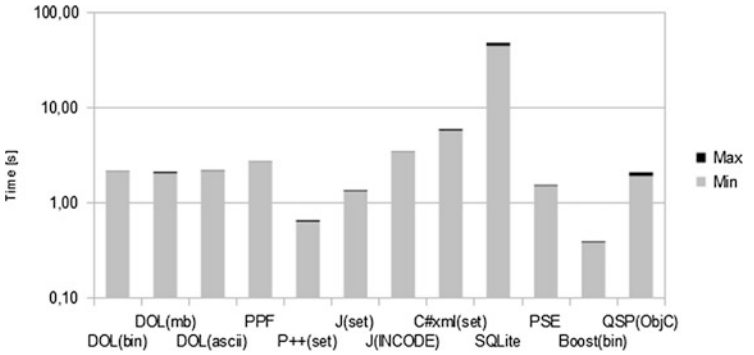

Fig. 7.8 Minimum and maximum of the total (sort+topVoted+traversal), for one million books without abstracts

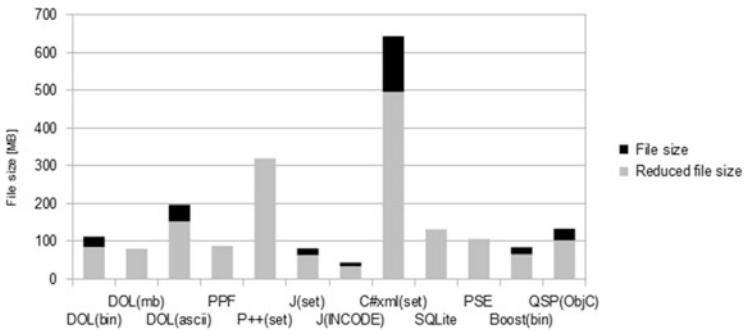

Fig. 7.9 Size of the disk file before and after one quarter of the data has been removed, for one million books, without abstracts. When there is no dark top, there was no file reduction

Table 7.6 Times on MacBookPro and iPhone 5 are very close (times in sec, file sizes in MB)

QSP with SSD 1M books	Create+ Sort+ TopVoted	Open	Traverse	Save	Delete	FileSize 1	FileSize 2	Total mean time
MacBookPro (no abstracts)	2.75	0.43	0.18	3.14	0.25	248	191	9.07
MacBookPro (with abstracts)	3.32	0.63	0.20	4.57	0.26	526	400	12.4
iPhone 5 (no abstracts)	2.75	0.43	0.19	3.14	0.25	248	191	9.34

Table 7.7 Average total times for different number of books (times in sec)

	No abstract			With abstract		
Technology/books count	50 k	250 k	1 M	50 k	250 k	1 M
DOL (bin)	1.36	7.00	31.11	1.83	9.57	46.00
DOL (mb)	0.26	1.38	6.85	0.46	3.37	14.38
DOL (ASCII)	2.06	11.59	49.29	3.02	18.06	72.73
PPF	0.44	1.86	8.67	0.78	3.77	16.21
P++(set)	0.76	3.59	14.89	1.47	6.72	32.98
J (set)	5.61	29.70	127.63	6.74	34.73	157.66
J (InCode)	4.18	19.40	82.05	4.51	23.23	105.57
C# XML(set)	3.23	17.24	76.13	3.25	18.40	80.30
SQLite	3.84	25.33	212.08	10.20	40.41	256.59
PSE Pro	1.37	3.14	10.65	2.06	6.73	28.72
Boost (bin)	1.47	6.97	28.35	1.65	10.64	31.97
ObjC (QSP)	0.86	4.77	19.74	1.30	8.62	30.75

7.9 Improvements

Benchmark rule No.4 was:

> *When a participant improves his/her system beyond the publically available version, the improved results will be accepted only if the participant reveals the technical details of the improvement.*

This chapter provides full stories and the technical details that may improve the performance of your persistence system by the order of magnitude.

7.9.1 Warmup of the Hard Drive Cache (PPF)

This improvement is specific for problems where all the data must be in virtual memory for fast traversal such as, for example, in VLSI CAD systems. Our benchmark also falls into this category.

The IO cache on today's hard disks is 64 MB and it keeps increasing. With RAM well in the GB range, the internal buffers used by the disk drivers may be even larger.

If a persistent system is based on memory paging, and all the data can fit into this combined fast storage, any re-read of the data is lightning fast.[18] But does the order in which the pages warm up (or move to the cache) matter?

This question led Soukup to the following experiment. When running the PPF benchmark,[19] he did the opposite to running the CacheKiller: he traversed the entire disk data in a serial manner, without doing anything with it.

That did not take long, left the disk image in the cache, and made PPF amazingly fast. The total time including the initial warm-up was 12.5× shorter comparing to the run after the CacheKiller, where PPF would slowly warm up by randomly accessing the pages.

Useful Trick No. 5.
Before the run which loads the data from disk to memory, read the entire disk file into a temporary buffer. This can be a short buffer which you keep overwriting until the end-of-file. This moves the entire disk file into the cache, and the subsequent load will be just as fast as if you ran it right after saving the data.

[18] This idea is based more on experimental evidence than on the exact knowledge of the HD construction or of the internal design of the disk drive.

[19] This idea is applicable only to persistent systems based on memory paging, because all the other systems read the disk sequentially anyway.

Table 7.8 Initial warm-up or preloading pages serially significantly improves the overall time

Technology	Create (s)	Open (s)	Sort+ TopVoted + Traverse (s)	Save (s)	Delete (s)	FileSize 1 (MB)	FileSize 2 (MB)	Total mean time (s)
PPF orig	7.27	0.01	4.79	1.05	296.4	351.90	351.90	310.30
PPF warm	5.91	3.67	9.46	0.80	3.85	358.18	359.18	24.18
PPF preload	5.88	4.12	3.66	1.90	0.63	359.29	359.29	16.22

Times for one million books with abstracts

This performance improvement is easy to explain. For mechanical hard drives, the repositioning of the head is the main source of the delay, and traversing the disk sequentially minimizes this delay. The performance improvement may not be as significant for solid state drives (SSD).

Soukup added this warm-up to PPF as one of the options, and we invited all the other participants to use the idea if they thought it would improve the performance of their system. Similar additions improved the performance of POST++ and PSE Pro by about the same rate.

The advantage of Useful Trick No. 5 is that it we can apply it to any persistent system without having any information about its internal implementation, size of pages, etc. PSE Pro support group proposed the following improvement, for situations where we have the information about the paging of the particular system.

Useful Trick No. 6.
Before the run which loads the data from disk to memory, access one pointer for each page, in the order in which the pages are stored on disk. This reads the disk sequentially in the cache and, at the same time, it loads all the pages directly into their memory locations.

This loads the pages to the cache sequentially just as the warmup does it, but it also loads all the pages to memory. Useful Trick No. 5 throws away what it reads, and only later do these pages move from the cache to memory. Table 7.8 shows the effect of using warmup or preloading for PPF.

7.9.2 Problem with Collecting Objects (Boost)

For testing Boost serialization[20] we chose the Boost implementation of the STL library. The first results (see Table 7.9) were a disaster with the worst total time of all tested systems, 30-times slower than DOL mb.

[20] We used Ver.1.53.

Table 7.9 Original poor results obtained with BOOST Ver.1.5.3, and improved results by Ramey and Machacek (times in sec, file sizes in MB)

1M books without abstracts	Create	Open	Sort+ TopVoted+ Traverse	Save	Delete	FileSize 1	FileSize 2	Total mean time
(1) Original Machacek	4.28	193.86	0.96	10.34	0.51	84.42	79.99	218.46
(2) New Ramey	9.21	13.7	14.0	8.24	1.30	70.4	53.9	51.3
(3) New Machacek	5.62	7.06	0.40	8.75	0.81	84.4	65.5	28.9

The author of Boost serialization, Robert Ramey, analyzed the problem and found the problem in the algorithm which collects all objects before saving to disk. In the case where STL containers were given pointers, it performed excessive searches.

Situations like that can happen easily, for example, when invoking:

```
class Book { ... };
std::set<Book*> books;
```

and we wondered why, with the world-wide use of Boost, nobody has complained about it.

Our discussion about the internal workings of the algorithm convinced us that even after this fix, the algorithm is still quite inefficient. It moves around objects instead of pointers, thus changing the object address; yet this address is used as a key in the std::set.

If this part of the Boost serialization is re-implemented with the algorithms from Sect. 2.1.6 (Useful Trick No. 4), it would be at least ten-times faster.

It took several weeks of hard negotiations and testing to find the best data structures for the benchmark. We clearly had different views of what would be a "modern and efficient" implementation. The benchmark requires two OneToMany associations and one ManyToMany associations that should be implemented to allow fast removal of objects, and also, in one case (books), sorting. STL does not provide such classes.[21]

Ramey's design (row No. 2 in Table 7.9) involved:

```
std::set<Book> books;
std::string title;
std::string abstract;
std::set<Author> authors;
std::string name;
std::multiset<BookToAuthor, order_dereferenced_pair<
   std::set<Book>::iterator,  std::set<Author>::iterator> authorToBooks;
std::multiset<AuthorToBook, order_dereferenced_pair<
    std::set<Author>::iterator,  std::set<Book>::iterator> bookToAuthors;
```

[21] InCode library has them but we did not want to go through the conversion of InCode to Boost. Also, we believe that persistent systems should be tested with their native libraries.

Machacek's design (row No. 3 in Table 7.9) involved:

```
std::set<Book*> books;
std::string title;
std::string abstract;
std::set<Author*> authors;
std::string name;
std::map<Book*, std::set<Author*>> booksToAuthors;
std::map<Author*, std::set<Book*>> authorsToBook;
```

Note that both designs use std::set, which is internally implemented as a binary tree. It is not as fast to remove an object from such a tree as it is from an intrusive linked list,[22] but it is reasonably fast. However, the set of books cannot be sorted by the vote, because the vote count is not unique. Both versions use a temporary array of Book* pointers, and sort this array with *qsort*.

Since our benchmark rules called for the best results to be published, Table 7.4 shows the results of Machacek's implementation. The improved version of Boost, Ver.1.5.4, will be released in June 2013.

Observation.
If you recently noticed a significant performance improvement of Boost serialization (version 1.5.4 or higher), Code Farms PPF library (version 3.5 or higher), or ObjectSTORE (c) PSE Pro, possibly running over ten-times faster on certain types of data, it is likely the result of the intense but friendly competition associated with this benchmark.

[22] Such as available from InCode or DOL.