QuickDB – Yet Another Database Management System?*

Radim Bača, Peter Chovanec, Michal Krátký, and Petr Lukáš

Department of Computer Science, FEECS, VŠB – Technical University of Ostrava
17. listopadu 15, Ostrava, 708 33, Czech Republic
{radim.baca,peter.chovanec,michal.kratky,petr.lukas}@vsb.cz

Abstract. Although DBMS (Database Management Systems) are often hidden for a user, they are a part of many applications utilized in day-by-day life. In general, we can suppose two main types of DBMS: OLTP (On-Line Transaction Processing) and OLAP (On-Line Analytical Processing). We can also distinguish another classification related to the connection of a client and DBMS: client-server and embedded DBMS. The embedded DBMS enable to achieve the maximum performance since an often slow network connection is not used. As a result, they are utilized by in-memory computations where the maximum throughput is required. Although it seems that a lot of high quality DBMS exist and another DBMS are not required, there is not a system to meet all demands of the real world. In this paper, we introduce our prototype of embedded database system called QuickDB. We show that it includes a wide variety of data structures and it provides more efficient performance in many cases compared to up-to-date (embedded) DBMS.

1 Introduction

Although DBMS (Database Management Systems) [9, 6] are often hidden for a user, they are a part of many applications used in day-by-day life. In general, we can suppose two main types of DBMS different in the workload for which they are designed: OLTP (On-Line Transaction Processing) and OLAP (On-Line Analytical Processing). Whereas the first type is mainly used by information systems where a user requires a support of transaction processing [9], the second type is used by, for example, business intelligence applications [15] or some computations and analysis where data structures of DBMS are utilized to manage data. We can also distinguish another classification related to the connection of a client and DBMS: client-server and embedded DBMS. The embedded DBMS enable to achieve the maximum performance since an often slow network connection is not used, therefore, they can be utilized in the case of in-memory computations where the maximum data throughput is required.

The major representatives of the client-server DBMS are the following systems: Oracle Database [18], Microsoft SQL Server [14], MySQL [16], PostgreSQL [21],

* This work is partially supported by SGS, VŠB – Technical University of Ostrava, No. SP2014/211, Czech Republic.

Firebird [8] and others. The major representatives of the embedded DBMS are the following systems: Microsoft Access[1], SQLite [20], Berkeley DB [17], eXtremeDB[2] and so on. When we consider only relational DBMS, many database systems have appeared during 35 years of their development (see the genealogy of relational DBMS [12]).

Although it seems that a lot of high quality DBMS exist and another DBMS are not required, there is not a system to meet all demands of the real world. For example, Berkeley DB provides the high performance of the B-tree and the paged queue, but it does not support any multidimensional data structure. On the other hand, in the case of libraries including an R-tree implementation (see Table 1), they support especially in-memory implementations, which penalizes them in the case of huge data. Moreover, they do not often support any other data structures. In this paper, we introduce a long-time project of the Database Research Group[3], a prototype DBMS called QuickDB [5]. It includes a wide variety of data structures and it provides more efficient performance in many cases compared to up-to-date DBMS. In Table 1, we show a comparison of individual features of selected DBMS and libraries. We can see that there are only two DBMS supporting all features: QuickDB and SQLite. However, SQLite do not use any cache buffer for data; only the operating system's cache is used during file operations. We must note that the table does not consider all features of DBMS (a support of transaction processing, availability for more platforms, e.g. UNIX and Windows, and so on). Since a performance comparison of embedded and client-server DBMS is rather problematic, QuickDB is compared only with embedded DBMS and libraries.

Table 1. Summary of supported features for all database systems and libraries compared in this article

DBS/Library	B-tree	Paged Queue (Heap table)	R-tree	32 bit	64 bit	In Memory/Disk Only/Cache Buffer
QuickDB [5]	✓	✓	✓	✓	✓	Cache Buffer
Berkley DB [17]	✓	✓	×	✓	✓	Cache Buffer
SQLite [20]	✓	✓	✓	✓	✓	Disk Only
Boost [4]	×	✓	✓	✓	✓	In Memory
libSpatialIndex [11]	×	×	✓	✓	✓	In Memory/ Disk Only
RTreeStar [19]	×	×	✓	✓	✓	In Memory
Superliminal Rtree [7]	×	×	✓	✓	✓	In Memory

[1] `http://office.microsoft.com/en-us/access/`
[2] `http://www.mcobject.com/extremedbfamily.shtml`
[3] `http://db.cs.vsb.cz/`

This paper is organized as follows. In Section 2, we describe an architecture of QuickDB. In Section 3, we put forward a comparison of QuickDB with other DBMS. In the last section, the paper content is resumed and the possibility of a future work is outlined.

2 QuickDB

In Figure 2, we see an architecture of QuickDB[4]. The cache buffer preserves pages of data structures to prevent disk accesses when a page is required. The overhead of the page size compared to the size on a disk is approximately 20%, i.e. the in-memory page size is 9,830B in the case of the 8kB page size. When a data structure operation returns a result, e.g. the range query, the result is stored in a ResultSet and returned to a user. After the user closes the ResultSet, it is returned to QuickDB.

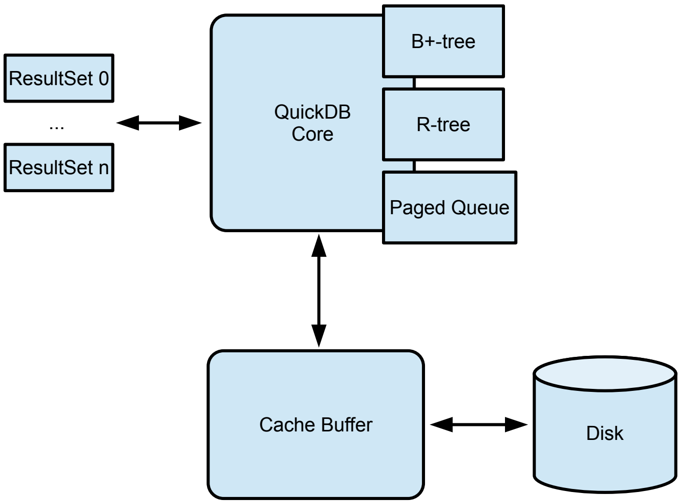

Fig. 1. An architecture of QuickDB

There exists an implementation of the B-tree, R-tree, and paged queue utilizing the core of QuickDB. The goal of QuickDB is to provide the maximum performance instead of an implementation of rich functionalities like a support of methods stored in a database and so on.

[4] QuickDB is implemented in C++.

3 Comparison of DBMS

3.1 Testing Environment

We perform a set of various tests where we compare our QuickDB with state-of-the-art implementations in each area. By the term area we mean a data structure type. Moreover, in Section 3.4, we show results of an application of QuickDB – a native XML database.

All experiments are executed in a single thread on an Intel Xeon 2.9 GHz CPU. We use a real data set of meteorological measurements of the Czech Hydrometeorological Institute[5], where the dimension of each record is 5. The collection contains 57,852,305 records and the total size of its textual representation is 1.12 GB. Queries used during the tests are randomly generated.

3.2 Comparison of Basic Data Structures

In this test, we compare the performance of two QuickDB basic data structures: the B-tree and the heap table (paged queue). The B-tree is a traditional data structure used by all relational DBMS [2]. The heap table is also supported by all relational DBMS and we test just a sequential scan in the heap table using a cursor.

B-tree We compare the QuickDB's B-tree performance of the insert and point query operations with the Berkeley DB and SQLite here. The performance of the insert operation can be influenced by several different aspects where we consider the following: (1) whether the data fit into the main memory cache or not, and (2) whether the data are sorted before the insert or not. All combinations for each B-tree are tested and the results are summarized in Table 2.

We can observe that SQLite performs worst in all cases. An overhead of SQLite is probably induced by the SQL interface and lack of its own buffer cache. Clearly, QuickDB and Berkeley DB have very similar performance on our data set, however, QuickDB slightly outperforms Berkeley DB in all tests. The only disadvantage of QuickDB is its a slightly bigger index.

Heap Table (Paged Queue) Now we compare the performance of the QuickDB's heap table with the Berkeley DB's queue. We use the Berkeley DB's queue since it supports the insert and sequential scan operations. The insert operation is preformed with limited memory, however, it is not very important here since only a sequential write is utilized. Then we perform the sequential scan with both cold and warm caches. The term warm cache means that all the data are already stored in the main memory.

Results are shown in Table 3. QuickDB is twice faster during the insert than Berkeley DB. Surprisingly, the performance of the Berkeley DB queue scan is

[5] `http://www.chmi.cz/`

Table 2. The B-tree comparison

	Operations	**QuickDB**	**Berkeley DB**	**SQLite**
80% of data fit in the main memory	Sorted insertion [thousands per second]	**531**	445	173
	Random insertion [thousands per second]	**289**	264	21
All data fit in the main memory	Sorted insertion [thousands per second]	**674**	526	173
	Random insertion [thousands per second]	**357**	312	21
	Queries [s]	**0.35**	0.38	0.64
	Index size [GB]	3.5	3.39	**2.57**

the same no matter whether the cache is cold or not. It leads us to a conclusion that Berkeley DB uses the OS file system cache which influences the results. However, QuickDB significantly outperforms Berkeley DB during in-memory sequential scan. On the other hand, the size of the Berkeley DB index file is quarter of the data set size, whereas, the QuickDB index file is equal to the data size.

Table 3. The heap table (page queue) comparison

	QuickDB	**Berkeley DB**
Inserting [thousands per second]	**2,244**	1,468
Sequential scan (warm) [s]	**4.64**	21.1
Sequential scan (cold) [s]	27.11	**21.15**
Index size [GB]	1.12	**0.25**

3.3 Comparison of Multidimensional Data Structures

In this section, we compare a performance a multidimensional data structure called the R-tree [10, 3] implemented in QuickDB with some other existing implementations. We compare the performance of both insert and range query operations. In the experiments, we use three collections of queries (a detail description is shown in Table 4).

The performance of inserting and query processing is compared with several libraries, namely Boost [4], libSpatialIndex [11], RTreeStar [19], Superliminal Rtree [7], and SQLite [20] as a representative of embedded database systems. Relational DBMS based on a client-server architecture (Oracle, PostgreSQL, MySQL, and so on) have not been taken into account since it is rather problematic to compare embedded and client-server DBMS.

Table 4. A basic characteristic of query groups

Query Group	**Result Size**	**Avg. Result Size**
1	1	1.0
2	$\langle 2, 999\rangle$	302.5
3	⟨10000, 99,999⟩	45,172

Since all tested libraries (except QuickDB and libSpatialIndex) support only an in-memory storage, no secondary storage has been used during the experiments and we do not measure the size of the indices. The page size is 8kB in all cases. In Table 5, we see that QuickDB and Boost provide the highest performance. While Boost is more efficient in the case of the insert operation, QuickDB slightly outperforms it in the case of query processing. We see that other implementations are outperformed by QuickDB and Boost.

Table 5. The R-tree comparison

DBMS/Library	**Insert Time** [thousands of inserts/s]	**Query Processing Time** [thousands of quries/s]		
		QG1	**QG2**	**QG3**
QuickDB	68.3	**38.9**	19.8	**1.7**
Boost	**97.2**	33.3	**20.0**	1.40
libSpatialIndex	12.2	18.5	13.2	0.74
RTreeStar	56.53	20.8	14.7	0.43
Superliminal Rtree	70.2	10.0	6.60	1.30
SQLite	66.4	17.30	12.10	0.76

3.4 Comparison of QuickDB Application

In this section, we present an application of the QuickDB framework. It is a native XML database called QuickXDB introduced in [13]. We briefly discuss how the QuickDB framework is exploited here and compare it with some other solutions.

Data structures There are two indexes representing an XML data collection, namely *document index* and *partition index* [1]. The document index serves as a primary access path since it fully describes both tree structure and actual text values of an XML collection[6]. We exploit two persistent data structures of the QuickDB framework: the B-tree and the paged queue.

[6] The reader is expected to understand the terms of the XML tree data model.

The B-tree of the document index has a *node label* as a key. There are two fundamental purposes of the node label: (1) it uniquely identifies each XML data node and (2) we are able to resolve a structural relationship of two nodes, e.g., one is a parent of another. The leaf nodes of the B-tree contain tags (node names) of XML nodes and pointers into the paged queue where text values are stored. The records for XML nodes in the leaf nodes of the B-tree are stored in the same order as in the original XML document or collection. Consequently, we can profit from using range queries to obtain the whole subtree of an XML node.

The partition index is the secondary access path. It is also a combination of the B-tree and the paged queue. However, here a tag name serves as a key and items of the B-tree leaf nodes include pointers to paged queues where corresponding node labels are stored in the document order.

Experimental evaluation We have performed a time comparison with 2 other native XML databases: MonetDB[7] and BaseX[8]. We have picked up 4 data collections: XMark[9] with factor $f = 10$ (1.1 GB), Swissprot (109 MB), TreeBank (82 MB), and DBLP (127 MB)[10].

For each collection, we have generated 3 sets of 50 distinct random XPath queries oriented purely on searching structural relationships between XML nodes. The sets differ in the upper range of their *selectivity*[11]: (1) selectivity up to 100%, (2) selectivity up to 10%, and (3) selectivity up to 1%. Different query selectivity can cause a different utilization of indexes. The complexity of queries vary for a different number of location steps (from 2 to 11) and also for a different number of branching predicates (from 0 to 4). Both ancestor-descendant and parent-child tree axes are randomly used. Branching predicates contain either single XPath subqueries or more subqueries connected by a logical conjunction.

Summarized results are presented in Figure 2. The values on the vertical axis stand for the total processing time of a set of queries. Each single query was evaluated 5 times, only arithmetic means of 3 times (without the best and the worst case) are considered.

We can see that our QuickXDB outperforms BaseX on all query sets except XMark with selectivity up to 1% and 10%. We also evaluate queries with the higher selectivity (up to 1% and 10%) on DBLP, Swissprot, and TreeBank collections faster than MonetDB. On the Swissprot query set with selectivity up to 1%, QuickXDB is more than 20× faster than BaseX and 8× faster than MonetDB. On the other hand, our QuickXDB is approximately 2.5× slower than MonetDB on DBLP and XMark query sets with selectivity up to 100% and 3.3× slower than BaseX on XMark with selectivity up to 10%.

[7] http://www.monetdb.org/XQuery/
[8] http://www.basex.org
[9] http://www.xml-benchmark.org/
[10] http://www.cs.washington.edu/research/xmldatasets/www/repository.html
[11] If we for example have a query `//a[./b]//c`, selectivity can be computed as `count(//a[./b]//c) div count(//c)`

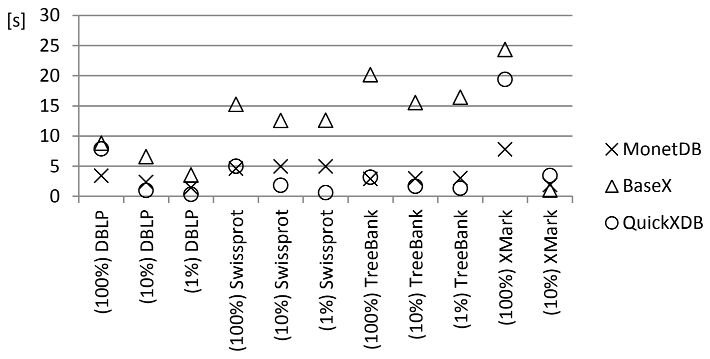

Fig. 2. A comparison of native XML databases

4 Conclusion

In this paper, we introduced a comparison of our prototype of embedded database system called QuickDB with other up-to-date embedded DBMS and libraries. As mentioned, the main goal of QuickDB is to provide the maximum performance. The results show that QuickDB outperforms these DBMS and libraries in a lot of cases. In our future work, we want to compare other features of DBMS, for example, scalability performance, transaction processing performance and so on.

References

1. R. Bača and M. Krátký. XML Query Processing Efficiency and Optimality. In *Proceeding IDEAS 12 Proceedings of the 16th International Database Engineering & Applications Symposium*, pages 8–13. ACM, 2012.
2. R. Bayer and E. M. McCreight. Organization and Maintenance of Large Ordered Indices. *Acta Inf.*, 1:173–189, 1972.
3. N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD 1990)*, 1990.
4. Boost.org. Boost C++ Libraries, `http://www.boost.org/`, 2014.
5. Database Reasearch Group. QuickDB, `http://db.cs.vsb.cz/`, 2014.
6. C. Date. *An Introduction to Database Systems.* Addison-Wesley, 8th edition, 2003.
7. G. Douglas. Superliminal Rtree, `http://superliminal.com/sources/sources.htm#C%20&%20C++%20Code`, 2014.
8. Firebird Foundation Incorporated. Firebird, `http://www.firebirdsql.org/`, 2014.
9. H. Garcia-Molina, J. Ullman, and J. Widom. *Database Systems: The Complete Book.* Prentice Hall, 2nd edition, 2008.
10. A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD 1984)*, pages 47–57. ACM Press, June 1984.

11. M. Hadjieleftheriou. libSpatialIndex, `http://libspatialindex.github.io/`, 2013.
12. Hasso-Plattner-Institut. The HPI Genealogy of Relational Database Management Systems, `http://www.hpi.uni-potsdam.de/naumann/projekte/rdbms_genealogy.html`, 2014.
13. P. Lukáš, R. Bača, and M. Krátký. QuickXDB: A Prototype of a Native XML DBMS. In *Proceedings of the Dateso 2013 Annual International Workshop*, pages 36–47, 2013.
14. Microsoft. Microsoft SQL Server 2012, `http://www.microsoft.com/en-us/sqlserver/default.aspx`, 2014.
15. S. Negash. Business Intelligence. *Communications of the Association for Information Systems*, 13, `http://aisel.aisnet.org/cais/vol13/iss1/15`, 2004.
16. Oracle. MySQL Community Edition, `http://www.mysql.com/products/community/`, 2014.
17. Oracle. Oracle Berkeley DB 12c, `http://www.oracle.com/technetwork/database/berkeleydb`, 2014.
18. Oracle. Oracle Database 12c, `http://www.oracle.com/us/products/database/overview/index.html`, 2014.
19. D. Spicuzza. R* Tree Implementation for C++, `http://www.virtualroadside.com/blog/index.php/2008/10/04/r-tree-implementation-for-cpp/`, 2014.
20. SQLite Consortium. Sqlite, `http://www.sqlite.org/`, 2014.
21. The PostgreSQL Global Development Group. PostgreSQL, `http://www.postgresql.org/`, 2014.