Introduction

This performance measurements were triggered by a competing company (AidAim) publishing the same benchmark comparing their SQLMemTable with kbmMemTable, ClientDataset and DBISAM3's memtable.

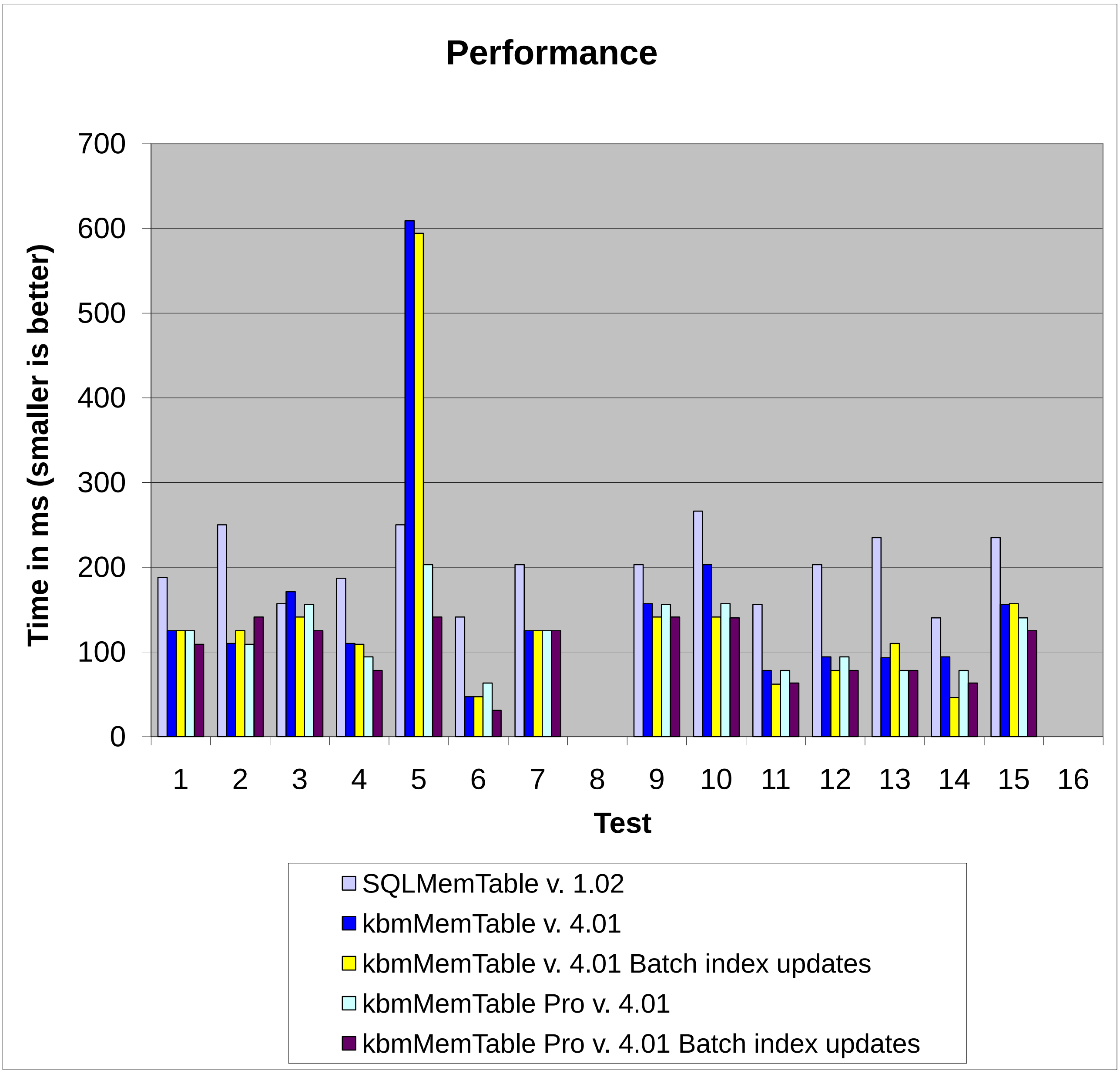
Since kbmMemTable is a vital part of many applications all over the world, C4D saw not other alternatives than to check what was the reasons for the performance differences, how does the performance look for other sized datasets and how could kbmMemTable be improved.

As will be shown in these measurements, kbmMemTable Std 4.01 and kbmMemTable Pro 4.01 performs very well in addition to being some of the most feature rich memorytables for the Borland community today.

Benchmarking is always a 'tricky' business in many ways. Its very difficult to select what to benchmark and how giving all parties a fair treatment. That was what happened in the original benchmark published by AidAim. The benchmark used for these tests are actually the same benchmark, although optionally (via $DEFINE's} slightly modified to be more fair to all parties (like not benchmarking screenupdates, allowing equal type of string comparisons etc.
But the benchmark is still not good enough. It is still a synthetic test, and time is still spend generating test data while benchmarking etc. Which all affect the end results. It may affect all tests equally, but if you then calculate how much faster one table is compared to another in that specific test, you will not get a correct result, since both results have been offset with a constant amount. (eg. 5 and 10 as percentage is not the same as 10 and 15 (offsetting the values with 5)

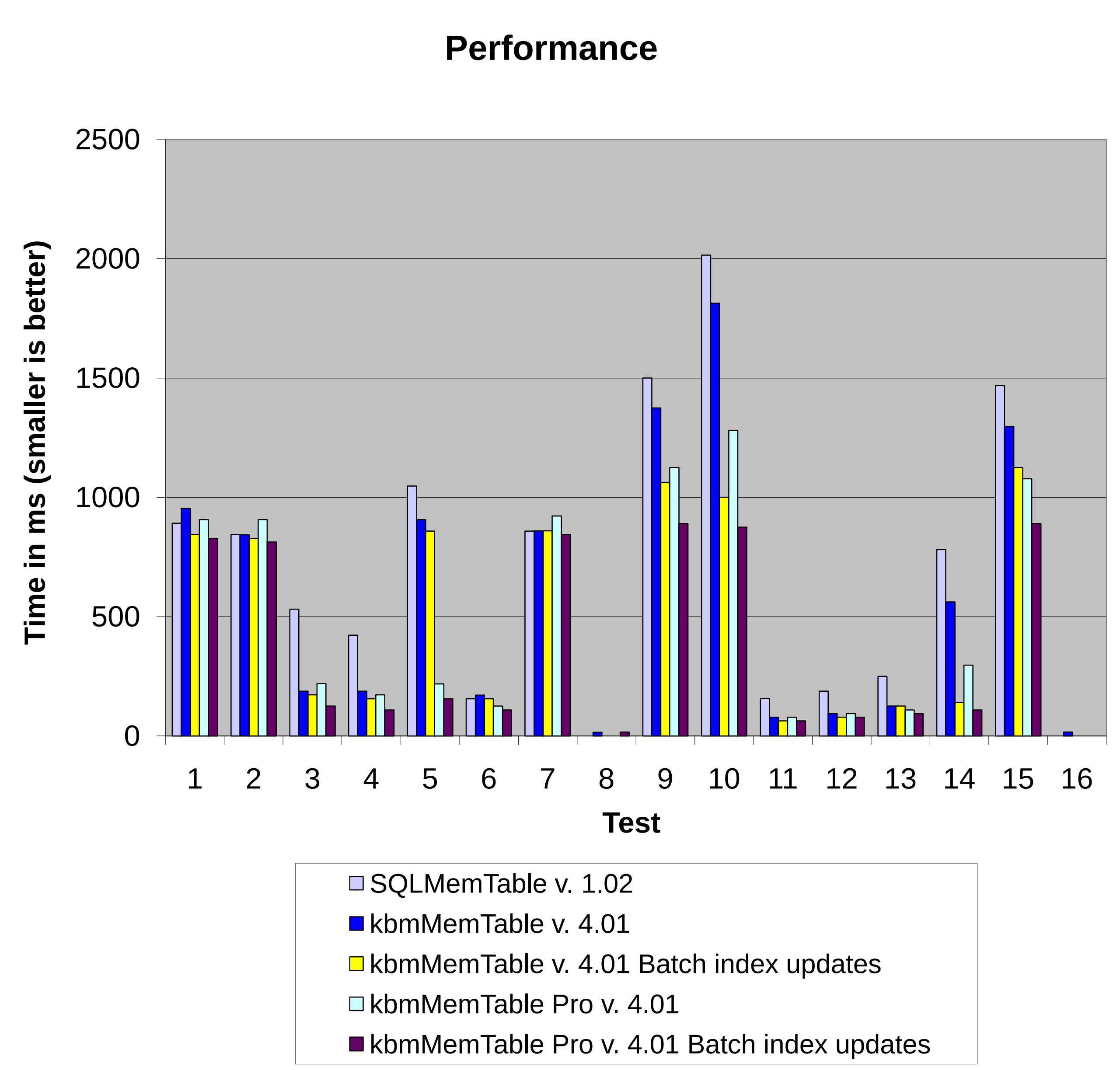
Benchmark 1.000 records

Test operation All measurements in 1/1000 secs (ms)	SQLMemTable v. 1.02	kbmMemTable v. 4.01	kbmMemTable v. 4.01 Batch index updates	kbmMemTable Pro v. 4.01	kbmMemTable Pro v. 4.01 Batch index updates
1) Insert wo indexes	188	125	125	125	109
2) Edit wo indexes	250	110	125	109	141
3) Locate by ID wo indexes	157	171	141	156	125
4) Locate by FInteger wo indexes	187	110	109	94	78
5) Locate by FString wo indexes	250	609	594	203	141
6) Delete wo indexes	141	47	47	63	31
7) Append wo indexes	203	125	125	125	125
8) Closetable wo indexes	0	0	0	0	0
9) Insert with indexes	203	157	141	156	141
10) Edit with indexes	266	203	141	157	140
11) Locate by ID with indexes	156	78	62	78	63
12) Locate by FInteger with indexes	203	94	78	94	78
13) Locate by FString with indexes	235	93	110	78	78
14) Delete with indexes	140	94	46	78	63
15) Append with indexes	235	156	157	140	125
16) Closetable with indexes	0	0	0	0	0
	2814	2172	2001	1656	1438



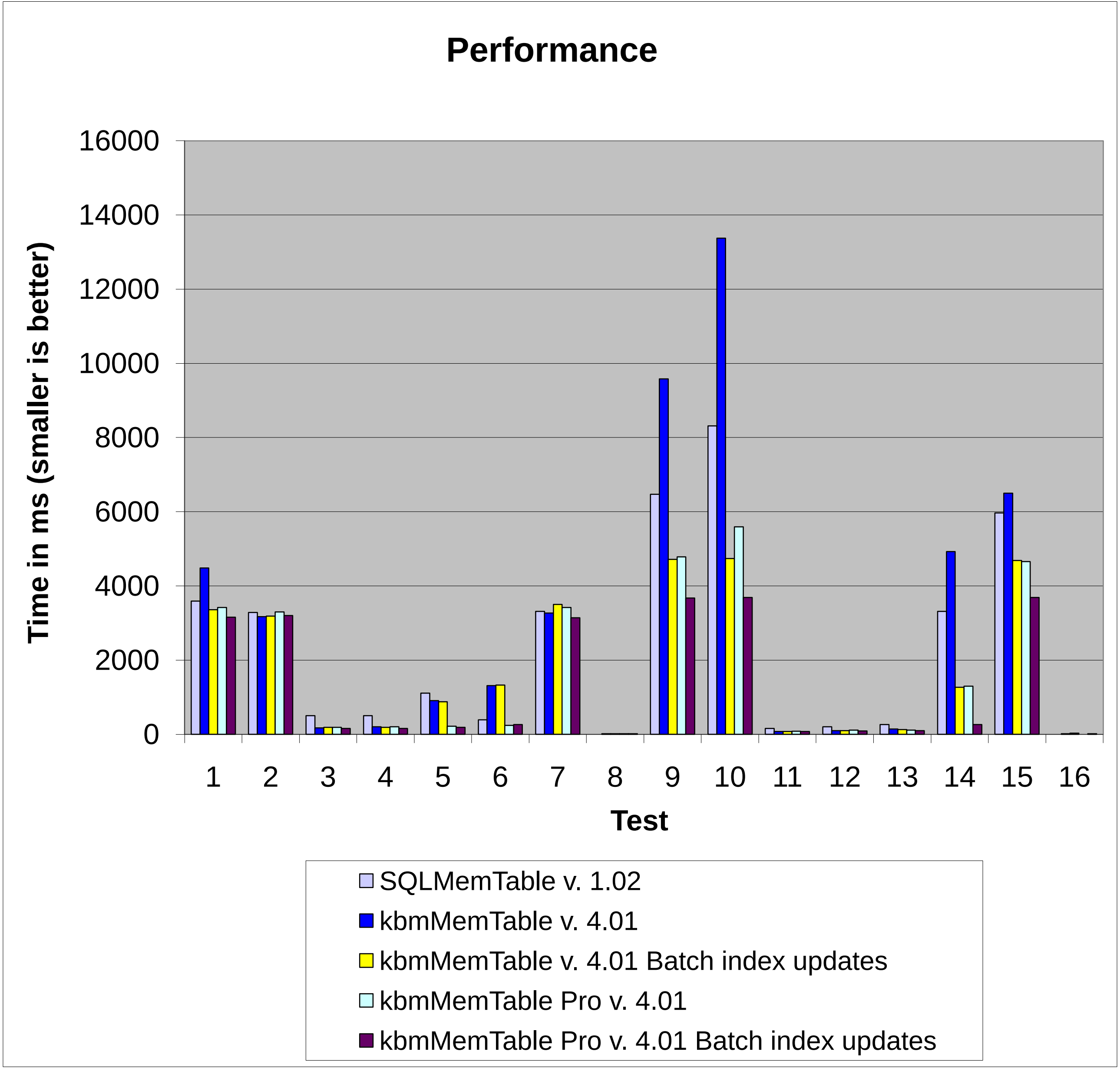
Benchmark 10.000 records

Test operation All measurements in 1/1000 secs (ms)	SQLMemTable v. 1.02	kbmMemTable v. 4.01	kbmMemTable v. 4.01 Batch index updates	kbmMemTable Pro v. 4.01	kbmMemTable Pro v. 4.01 Batch index updates
1) Insert wo indexes	891	954	844	906	828
2) Edit wo indexes	844	843	828	906	813
3) Locate by ID wo indexes	531	188	172	219	125
4) Locate by FInteger wo indexes	422	187	156	172	109
5) Locate by FString wo indexes	1047	907	859	218	156
6) Delete wo indexes	156	171	156	125	109
7) Append wo indexes	859	860	860	922	844
8) Closetable wo indexes	0	15	0	0	16
9) Insert with indexes	1500	1375	1063	1125	890
10) Edit with indexes	2015	1813	1000	1281	875
11) Locate by ID with indexes	157	78	63	78	63
12) Locate by FInteger with indexes	187	94	78	94	78
13) Locate by FString with indexes	250	125	125	109	94
14) Delete with indexes	781	562	141	297	109
15) Append with indexes	1469	1297	1125	1078	890
16) Closetable with indexes	0	16	0	0	0
	11109	9485	7470	7530	5999



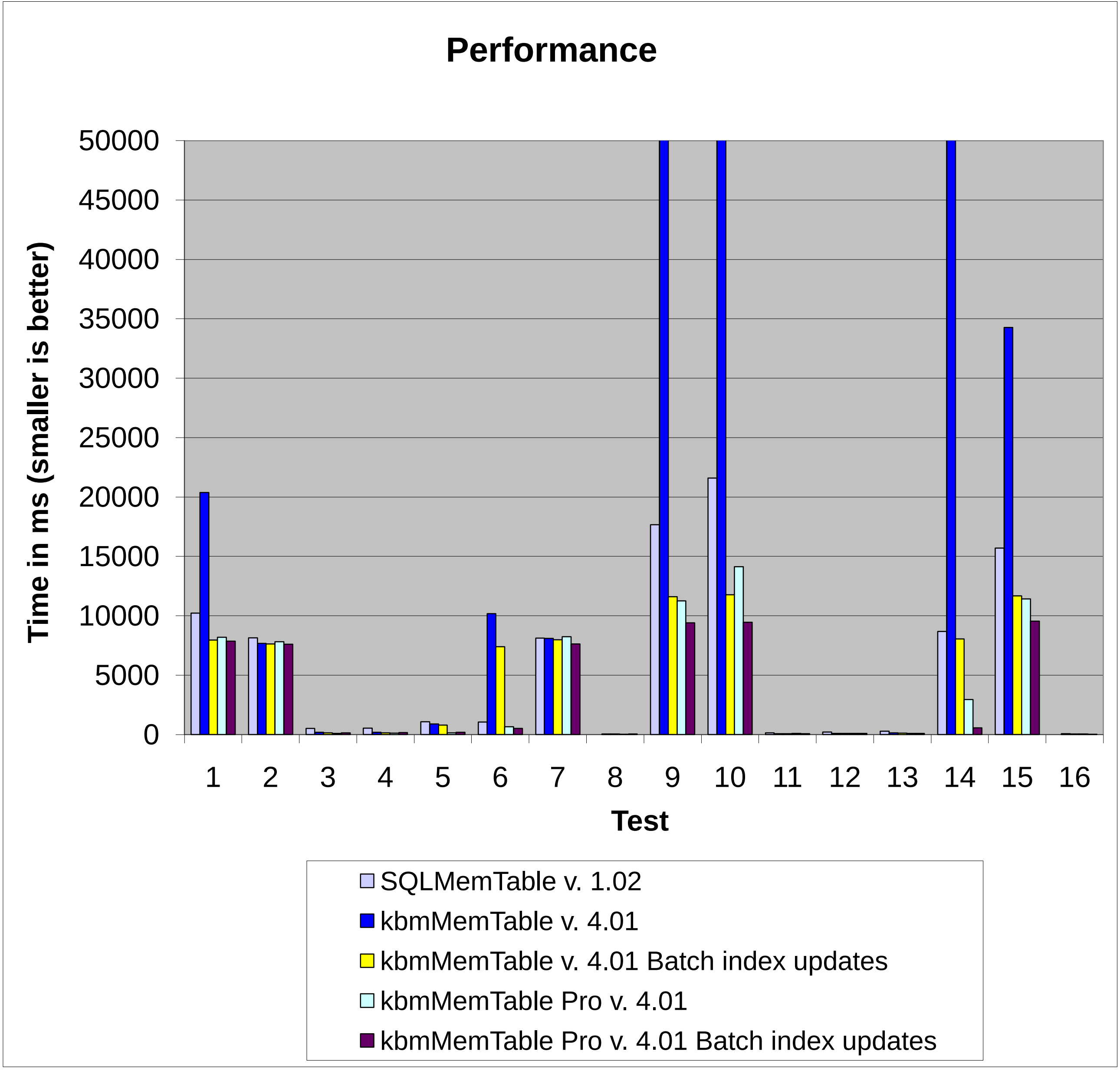
Benchmark 40.000 records

Test operation All measurements in 1/1000 secs (ms)	SQLMemTable v. 1.02	kbmMemTable v. 4.01	kbmMemTable v. 4.01 Batch index updates	kbmMemTable Pro v. 4.01	kbmMemTable Pro v. 4.01 Batch index updates
1) Insert wo indexes	3594	4484	3359	3422	3156
2) Edit wo indexes	3281	3172	3188	3296	3203
3) Locate by ID wo indexes	500	172	187	188	156
4) Locate by FInteger wo indexes	500	203	188	203	156
5) Locate by FString wo indexes	1110	907	875	219	188
6) Delete wo indexes	390	1312	1328	238	265
7) Append wo indexes	3313	3265	3500	3422	3141
8) Closetable wo indexes	0	16	16	15	16
9) Insert with indexes	6469	9578	4718	4782	3672
10) Edit with indexes	8312	13375	4735	5593	3688
11) Locate by ID with indexes	156	78	78	79	78
12) Locate by FInteger with indexes	203	94	94	109	93
13) Locate by FString with indexes	266	140	125	109	94
14) Delete with indexes	3313	4922	1265	1297	266
15) Append with indexes	5968	6500	4688	4656	3687
16) Closetable with indexes	0	16	31	0	16
	37375	48234	28375	27628	21875



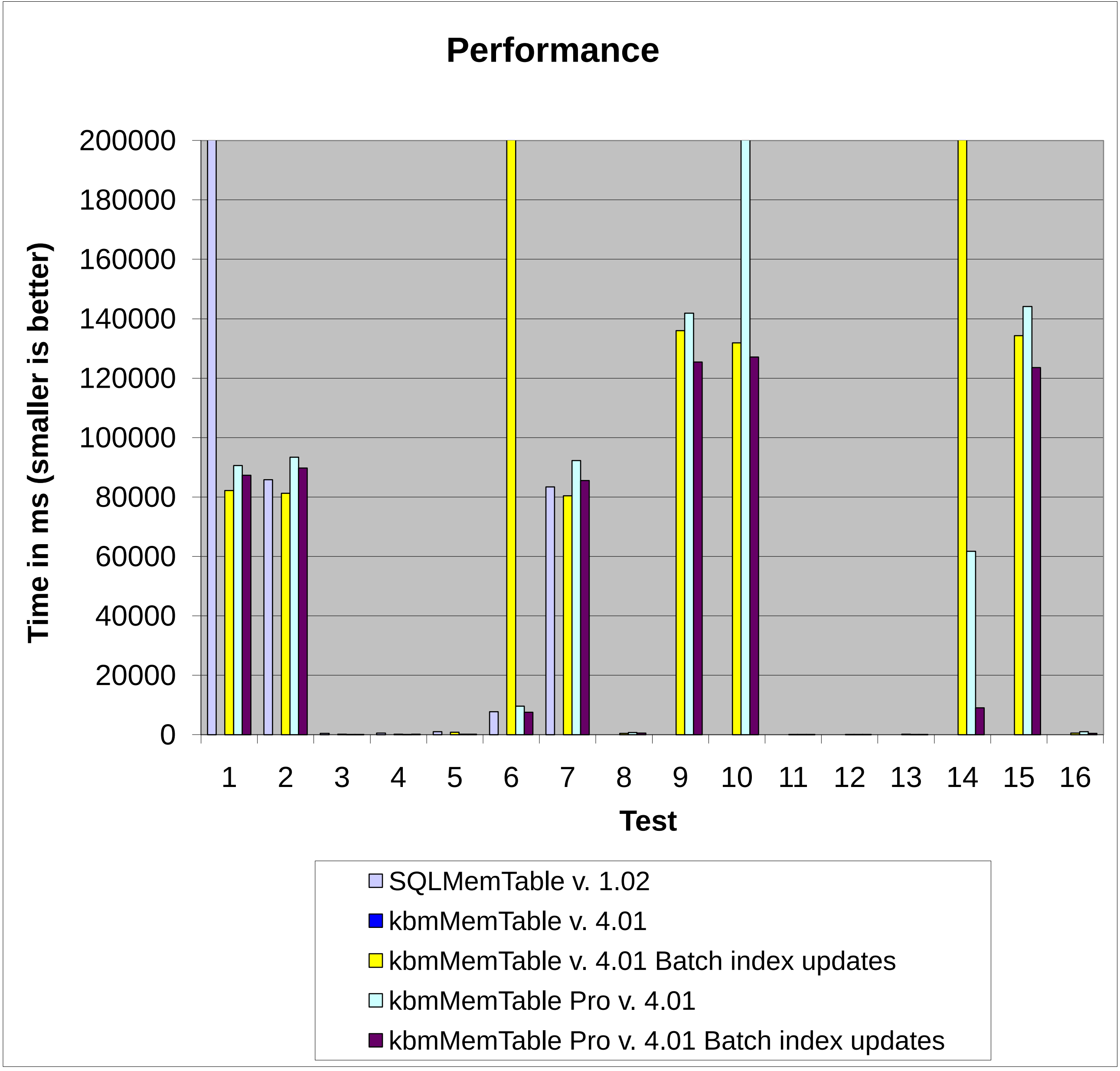
Benchmark 100.000 records

Test operation	SQLMemTable v. 1.02	kbmMemTable v. 4.01	kbmMemTable v. 4.01	kbmMemTable Pro v. 4.01	kbmMemTable Pro v. 4.01
All measurements in 1/1000 secs (ms)			Batch index updates		Batch index updates
1) Insert wo indexes	10218	20375	7968	8188	7875
2) Edit wo indexes	8141	7688	7641	7828	7609
3) Locate by ID wo indexes	516	187	156	110	141
4) Locate by FInteger wo indexes	546	203	157	125	171
5) Locate by FString wo indexes	1079	906	812	157	204
6) Delete wo indexes	1062	10172	7406	656	531
7) Append wo indexes	8125	8093	7984	8250	7625
8) Closetable wo indexes	0	47	47	31	47
9) Insert with indexes	17656	68906	11610	11250	9407
10) Edit with indexes	21594	104438	11781	14141	9453
11) Locate by ID with indexes	157	78	78	94	78
12) Locate by FInteger with indexes	218	93	94	94	94
13) Locate by FString with indexes	297	141	125	94	93
14) Delete with indexes	8688	54094	8062	2953	563
15) Append with indexes	15703	34281	11672	11422	9547
16) Closetable with indexes	0	78	63	47	32
	94000	309780	75656	65440	53470



Benchmark 1.000.000 records

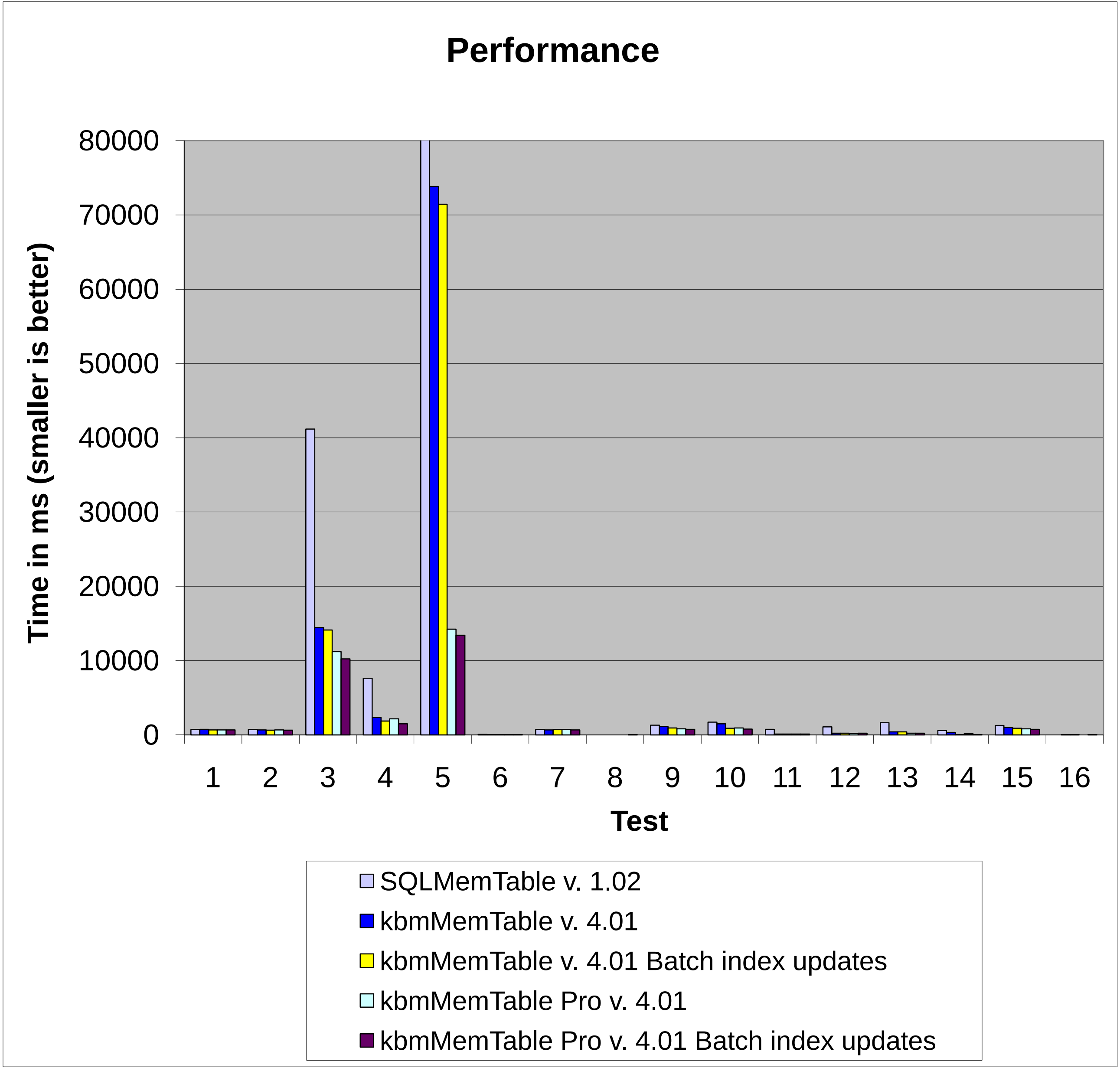
Test operation All measurements in 1/1000 secs (ms)	SQLMemTable v. 1.02	kbmMemTable v. 4.01	kbmMemTable v. 4.01 Batch index updates	kbmMemTable Pro v. 4.01	kbmMemTable Pro v. 4.01 Batch index updates
1) Insert wo indexes	272797	Operates, but is taking a very long time	82204	90625	87281
2) Edit wo indexes	85875		81250	93422	89765
3) Locate by ID wo indexes	469		203	125	109
4) Locate by FInteger wo indexes	516		219	125	141
5) Locate by FString wo indexes	1063		875	157	172
6) Delete wo indexes	7734		3254781	9578	7578
7) Append wo indexes	83359		80422	92234	85531
8) Closetable wo indexes	0		500	734	516
9) Insert with indexes	Crashed after inserting 20% with 'out of memory'		135953	141906	125453
10) Edit with indexes			131834	208172	127125
11) Locate by ID with indexes			79	94	94
12) Locate by FInteger with indexes			109	109	109
13) Locate by FString with indexes			141	110	110
14) Delete with indexes			3218812	61703	9031
15) Append with indexes			134328	144141	123547
16) Closetable with indexes			563	1016	500
			7122273	844251	657062



Alternative 100.000 record benchmark

This benchmark uses the DBMemBenchmark with the defines PROGRESS undefined, and LOCATE_ALL defined.
What that means is that no progress bars will be updated, and the locate tests is performed the same number of times as the number of records in the benchmark.
That means instead of the locates are performed only 1000 times, they are in this alternative benchmark performed 10.000 times.

Test operation All measurements in 1/1000 secs (ms)	SQLMemTable v. 1.02	kbmMemTable v. 4.01	kbmMemTable v. 4.01 Batch index updates	kbmMemTable Pro v. 4.01	kbmMemTable Pro v. 4.01 Batch index updates
1) Insert wo indexes	704	750	672	672	672
2) Edit wo indexes	703	672	640	657	640
3) Locate by ID wo indexes	41172	14453	14125	11187	10250
4) Locate by FInteger wo indexes	7609	2343	1860	2156	1485
5) Locate by FString wo indexes	93312	73829	71437	14250	13422
6) Delete wo indexes	63	15	16	32	15
7) Append wo indexes	703	688	703	718	656
8) Closetable wo indexes	0	0	0	0	16
9) Insert with indexes	1297	1109	922	828	734
10) Edit with indexes	1703	1500	907	938	766
11) Locate by ID with indexes	750	125	125	125	125
12) Locate by FInteger with indexes	1063	218	203	187	203
13) Locate by FString with indexes	1625	407	406	235	235
14) Delete with indexes	609	343	31	140	15
15) Append with indexes	1266	1000	906	829	735
16) Closetable with indexes	0	16	16	0	15
	152579	97468	92969	32954	29984



Notes

1) SQLMemTable do not handle locale specific indexes why the kbmMemTable tests equally was performed on non localized strings.

2) The testsuite was changed slightly from the original one published by AidAim to include Append and table close
(should have been EmptyTable, but SQLMemTable failed that)

3) Further the test suite contans a BATCH definition which can be enabled or disable depending on if the kbmMemTable test should
be performed with or without batch index handling.

4) All tests have been performed on a P4 2.6Ghz with 512MB RAM, running XP Home Edition. The suite was compiled with
Delphi 7 with optimizations enabled.

5) The testsuite is in reality not extremely fair to any of the products with respect to real time measurements since the loops contain lots of
Application.HandleMessage and other such stuff which influence the time measurements.
Since both the SQLMemTable and kbmMemTable tests are subject to that, a comparison between the two is still possible.

6) kbmMemTable was run in Performance mode pfFast.

7) Benchmarking with 1 million records ended up in an out of memory exception for SQLMemTable while inserting with indexes.

8) If you run a non SQLMemTable test first and then the SQLMemTable test afterwards, SQLMemTable do not correctly show numbers for non indexed
operation. This is due to a bug in SQLMemTable v. 1.02.

Conclusions

For most practical uses of a memtable, kbmMemTable Std. v. 4.01 performs significantly better than SQLMemTable even without batch indexing. In lots of real life uses, one would use batch indexing in which case kbmMemTable often performs the operation in half the time taken by SQLMemTable.

On extremely large amounts of records (>50.000), kbmMemTable without batched indexes is hurt by the fact that each update/insert/delete results in rearranging a Tlist, while SQLMemTable internally uses another type of linked storage not hurt as severely in these situations.
With batched indexes, kbmMemTable however do not suffer as severely from that problem except when deleting records the way this benchmark does, deleting all records using the delete method, one by one. One would usually use EmptyTable or Close in this situation.

kbmMemTable Pro 4.01 outperforms SQLMemTable in several cases with more than 800%, and on average around 50% on the locate limited tests. On the alternative test where the full range of records are being located, kbmMemTable Pro 4.01 performs SQLMemTable v. 1.02 with 400%

kbmMemTable Pro 4.01 is freely available for all holders of kbmMW commercial licenses
kbmMemTable Std 4.01 is freely available and open source.

Get the latest versions from **www.components4developers.com**