

Join GitHub today

GitHub is home to over 31 million developers working together to host and review code, manage projects, and build software together.

[Sign up](#)

Branch: master ▾ DB2 / src / main / java / index / BPlusNode.java

[Find file](#) [Copy path](#) dreamlegends init add all files

2c520b8 7 days ago

1 contributor

244 lines (231 sloc) 10 KB

[Raw](#) [Blame](#) [History](#)   

```
1 package index;
2
3 import common.Pair;
4 import databox.DataBox;
5 import io.Page;
6 import table.RecordId;
7
8 import java.nio.ByteBuffer;
9 import java.util.Optional;
10
11 /**
12  * An inner node or a leaf node. See InnerNode and LeafNode for more
13  * information.
14  */
15 abstract class BPlusNode {
16     // Core API //////////////////////////////////////
17     /**
18      * n.get(k) returns the leaf node on which k may reside when queried from n.
19      * For example, consider the following B+ tree (for brevity, only keys are
20      * shown; record ids are omitted).
21      *
22      *
23      *           inner
24      *       +-----+
25      *       | 10 | 20 |   |
26      *       +-----+
27      *       /   |   \
28      *      /    |    \
29      *  +-----+ +-----+ +-----+
30      *  | 1 | 2 | 3 | |->| 11 | 12 | 13 | |->| 21 | 22 | 23 | |
31      *  +-----+ +-----+ +-----+
32      * leaf0      leaf1      leaf2
33      *
34      * inner.get(x) should return
35      *
36      * - leaf0 when x < 10,
37      * - leaf1 when 10 <= x < 20, and
38      * - leaf2 when x >= 20.
39      *
40      * Note that inner.get(4) would return leaf0 even though leaf0 doesn't
41      * actually contain 4.
42      */
43     public abstract LeafNode get(DataBox key);
```

```

47  /**
48  * n.getLeftmostLeaf() returns the leftmost leaf in the subtree rooted by n.
49  * In the example above, inner.getLeftmostLeaf() would return leaf0, and
50  * leaf1.getLeftmostLeaf() would return leaf1.
51  */
52  public abstract LeafNode getLeftmostLeaf();
53
54  /**
55  * n.put(k, r) inserts the pair (k, r) into the subtree rooted by n. There
56  * are two cases to consider:
57  *
58  * Case 1: If inserting the pair (k, r) does NOT cause n to overflow, then
59  * Optional.empty() is returned.
60  * Case 2: If inserting the pair (k, r) does cause the node n to overflow,
61  * then n is split into a left and right node (described more
62  * below) and a pair (split_key, right_node_page_num) is returned
63  * where right_node_page_num is the page number of the newly
64  * created right node, and the value of split_key depends on
65  * whether n is an inner node or a leaf node (described more below).
66  *
67  * Now we explain how to split nodes and which split keys to return. Let's
68  * take a look at an example. Consider inserting the key 4 into the example
69  * tree above. No nodes overflow (i.e. we always hit case 1). The tree then
70  * looks like this:
71  *
72  *
73  *
74  *
75  *
76  *
77  *
78  *
79  *
80  *
81  *
82  *
83  *
84  *
85  *
86  *
87  *
88  *
89  *
90  *
91  *
92  *
93  *
94  *
95  *
96  *
97  *
98  *
99  *
100  *
101  *
102  *
103  *
104  *
105  *
106  *
107  *
108  *
109  *
110  *
111  *
112  *
113  *
114  *
115  *
116  *
117  *
118  *
119  *
120  *
121  *
122  *
123  *
124  *
125  *
126  *
127  *
128  *
129  *
130  *
131  *
132  *
133  *
134  *
135  *
136  *
137  *
138  *
139  *
140  *
141  *
142  *
143  *
144  *
145  *
146  *
147  *
148  *
149  *
150  *
151  *
152  *
153  *
154  *
155  *
156  *
157  *
158  *
159  *
160  *
161  *
162  *
163  *
164  *
165  *
166  *
167  *
168  *
169  *
170  *
171  *
172  *
173  *
174  *
175  *
176  *
177  *
178  *
179  *
180  *
181  *
182  *
183  *
184  *
185  *
186  *
187  *
188  *
189  *
190  *
191  *
192  *
193  *
194  *
195  *
196  *
197  *
198  *
199  *
200  *
201  *
202  *
203  *
204  *
205  *
206  *
207  *
208  *
209  *
210  *
211  *
212  *
213  *
214  *
215  *
216  *
217  *
218  *
219  *
220  *
221  *
222  *
223  *
224  *
225  *
226  *
227  *
228  *
229  *
230  *
231  *
232  *
233  *
234  *
235  *
236  *
237  *
238  *
239  *
240  *
241  *
242  *
243  *
244  *
245  *
246  *
247  *
248  *
249  *
250  *
251  *
252  *
253  *
254  *
255  *
256  *
257  *
258  *
259  *
260  *
261  *
262  *
263  *
264  *
265  *
266  *
267  *
268  *
269  *
270  *
271  *
272  *
273  *
274  *
275  *
276  *
277  *
278  *
279  *
280  *
281  *
282  *
283  *
284  *
285  *
286  *
287  *
288  *
289  *
290  *
291  *
292  *
293  *
294  *
295  *
296  *
297  *
298  *
299  *
300  *
301  *
302  *
303  *
304  *
305  *
306  *
307  *
308  *
309  *
310  *
311  *
312  *
313  *
314  *
315  *
316  *
317  *
318  *
319  *
320  *
321  *
322  *
323  *
324  *
325  *
326  *
327  *
328  *
329  *
330  *
331  *
332  *
333  *
334  *
335  *
336  *
337  *
338  *
339  *
340  *
341  *
342  *
343  *
344  *
345  *
346  *
347  *
348  *
349  *
350  *
351  *
352  *
353  *
354  *
355  *
356  *
357  *
358  *
359  *
360  *
361  *
362  *
363  *
364  *
365  *
366  *
367  *
368  *
369  *
370  *
371  *
372  *
373  *
374  *
375  *
376  *
377  *
378  *
379  *
380  *
381  *
382  *
383  *
384  *
385  *
386  *
387  *
388  *
389  *
390  *
391  *
392  *
393  *
394  *
395  *
396  *
397  *
398  *
399  *
400  *
401  *
402  *
403  *
404  *
405  *
406  *
407  *
408  *
409  *
410  *
411  *
412  *
413  *
414  *
415  *
416  *
417  *
418  *
419  *
420  *
421  *
422  *
423  *
424  *
425  *
426  *
427  *
428  *
429  *
430  *
431  *
432  *
433  *
434  *
435  *
436  *
437  *
438  *
439  *
440  *
441  *
442  *
443  *
444  *
445  *
446  *
447  *
448  *
449  *
450  *
451  *
452  *
453  *
454  *
455  *
456  *
457  *
458  *
459  *
460  *
461  *
462  *
463  *
464  *
465  *
466  *
467  *
468  *
469  *
470  *
471  *
472  *
473  *
474  *
475  *
476  *
477  *
478  *
479  *
480  *
481  *
482  *
483  *
484  *
485  *
486  *
487  *
488  *
489  *
490  *
491  *
492  *
493  *
494  *
495  *
496  *
497  *
498  *
499  *
500  *
501  *
502  *
503  *
504  *
505  *
506  *
507  *
508  *
509  *
510  *
511  *
512  *
513  *
514  *
515  *
516  *
517  *
518  *
519  *
520  *
521  *
522  *
523  *
524  *
525  *
526  *
527  *
528  *
529  *
530  *
531  *
532  *
533  *
534  *
535  *
536  *
537  *
538  *
539  *
540  *
541  *
542  *
543  *
544  *
545  *
546  *
547  *
548  *
549  *
550  *
551  *
552  *
553  *
554  *
555  *
556  *
557  *
558  *
559  *
560  *
561  *
562  *
563  *
564  *
565  *
566  *
567  *
568  *
569  *
570  *
571  *
572  *
573  *
574  *
575  *
576  *
577  *
578  *
579  *
580  *
581  *
582  *
583  *
584  *
585  *
586  *
587  *
588  *
589  *
590  *
591  *
592  *
593  *
594  *
595  *
596  *
597  *
598  *
599  *
600  *
601  *
602  *
603  *
604  *
605  *
606  *
607  *
608  *
609  *
610  *
611  *
612  *
613  *
614  *
615  *
616  *
617  *
618  *
619  *
620  *
621  *
622  *
623  *
624  *
625  *
626  *
627  *
628  *
629  *
630  *
631  *
632  *
633  *
634  *
635  *
636  *
637  *
638  *
639  *
640  *
641  *
642  *
643  *
644  *
645  *
646  *
647  *
648  *
649  *
650  *
651  *
652  *
653  *
654  *
655  *
656  *
657  *
658  *
659  *
660  *
661  *
662  *
663  *
664  *
665  *
666  *
667  *
668  *
669  *
670  *
671  *
672  *
673  *
674  *
675  *
676  *
677  *
678  *
679  *
680  *
681  *
682  *
683  *
684  *
685  *
686  *
687  *
688  *
689  *
690  *
691  *
692  *
693  *
694  *
695  *
696  *
697  *
698  *
699  *
700  *
701  *
702  *
703  *
704  *
705  *
706  *
707  *
708  *
709  *
710  *
711  *
712  *
713  *
714  *
715  *
716  *
717  *
718  *
719  *
720  *
721  *
722  *
723  *
724  *
725  *
726  *
727  *
728  *
729  *
730  *
731  *
732  *
733  *
734  *
735  *
736  *
737  *
738  *
739  *
740  *
741  *
742  *
743  *
744  *
745  *
746  *
747  *
748  *
749  *
750  *
751  *
752  *
753  *
754  *
755  *
756  *
757  *
758  *
759  *
760  *
761  *
762  *
763  *
764  *
765  *
766  *
767  *
768  *
769  *
770  *
771  *
772  *
773  *
774  *
775  *
776  *
777  *
778  *
779  *
780  *
781  *
782  *
783  *
784  *
785  *
786  *
787  *
788  *
789  *
790  *
791  *
792  *
793  *
794  *
795  *
796  *
797  *
798  *
799  *
800  *
801  *
802  *
803  *
804  *
805  *
806  *
807  *
808  *
809  *
810  *
811  *
812  *
813  *
814  *
815  *
816  *
817  *
818  *
819  *
820  *
821  *
822  *
823  *
824  *
825  *
826  *
827  *
828  *
829  *
830  *
831  *
832  *
833  *
834  *
835  *
836  *
837  *
838  *
839  *
840  *
841  *
842  *
843  *
844  *
845  *
846  *
847  *
848  *
849  *
850  *
851  *
852  *
853  *
854  *
855  *
856  *
857  *
858  *
859  *
860  *
861  *
862  *
863  *
864  *
865  *
866  *
867  *
868  *
869  *
870  *
871  *
872  *
873  *
874  *
875  *
876  *
877  *
878  *
879  *
880  *
881  *
882  *
883  *
884  *
885  *
886  *
887  *
888  *
889  *
890  *
891  *
892  *
893  *
894  *
895  *
896  *
897  *
898  *
899  *
900  *
901  *
902  *
903  *
904  *
905  *
906  *
907  *
908  *
909  *
910  *
911  *
912  *
913  *
914  *
915  *
916  *
917  *
918  *
919  *
920  *
921  *
922  *
923  *
924  *
925  *
926  *
927  *
928  *
929  *
930  *
931  *
932  *
933  *
934  *
935  *
936  *
937  *
938  *
939  *
940  *
941  *
942  *
943  *
944  *
945  *
946  *
947  *
948  *
949  *
950  *
951  *
952  *
953  *
954  *
955  *
956  *
957  *
958  *
959  *
960  *
961  *
962  *
963  *
964  *
965  *
966  *
967  *
968  *
969  *
970  *
971  *
972  *
973  *
974  *
975  *
976  *
977  *
978  *
979  *
980  *
981  *
982  *
983  *
984  *
985  *
986  *
987  *
988  *
989  *
990  *
991  *
992  *
993  *
994  *
995  *
996  *
997  *
998  *
999  *

```

inner

```

+-----+
| 10 | 20 |   |   |
+-----+
/       |       \
+-----+ +-----+ +-----+
| 1 | 2 | 3 | 4 | -> | 11 | 12 | 13 |   | -> | 21 | 22 | 23 |   |
+-----+ +-----+ +-----+
leaf0      leaf1      leaf2

```

Now let's insert key 5 into the tree. Now, leaf0 overflows and creates a new right sibling leaf3. d entries remain in the left node; d + 1 entries are moved to the right node. DO NOT REDISTRIBUTE ENTRIES ANY OTHER WAY. In our example, leaf0 and leaf3 would look like this:

```

+-----+ +-----+
| 1 | 2 |   | -> | 3 | 4 | 5 |   |
+-----+ +-----+
leaf0      leaf3

```

When a leaf splits, it returns the first entry in the right node as the split key. In this example, 3 is the split key. After leaf0 splits, inner inserts the new key and child pointer into itself and hits case 0 (i.e. it does not overflow). The tree looks like this:

```

inner
+-----+
| 3 | 10 | 20 |   |
+-----+
/       |       \
+-----+ +-----+ +-----+
| 1 | 2 |   | -> | 3 | 4 | 5 | 6 | -> | 11 | 12 | 13 |   | -> | 21 | 22 | 23 |   |
+-----+ +-----+ +-----+
leaf0      leaf3      leaf1      leaf2

```

When an inner node splits, the first d entries are kept in the left node and the last d entries are moved to the right node. The middle entry is moved (not copied) up as the split key. For example, we would split the following order 2 inner node

```

114 *
115 * +---+---+---+
116 * | 1 | 2 | 3 | 4 | 5
117 * +---+---+---+
118 *
119 * into the following two inner nodes
120 *
121 * +---+---+---+ +---+---+---+
122 * | 1 | 2 |   |   | 4 | 5 |   |
123 * +---+---+---+ +---+---+---+
124 *
125 * with a split key of 3.
126 *
127 * DO NOT redistribute entries in any other way besides what we have
128 * described. For example, do not move entries between nodes to avoid
129 * splitting.
130 *
131 * Our B+ trees do not support duplicate entries with the same key. If a
132 * duplicate key is inserted, the tree is left unchanged and an exception is
133 * raised.
134 */
135 public abstract Optional<Pair<DataBox, Integer>> put(DataBox key, RecordId rid)
136     throws BPlusTreeException;
137
138 /**
139  * n.remove(k) removes the key k and its corresponding record id from the
140  * subtree rooted by n, or does nothing if the key k is not in the subtree.
141  * REMOVE SHOULD NOT REBALANCE THE TREE. Simply delete the key and
142  * corresponding record id. For example, running inner.remove(2) on the
143  * example tree above would produce the following tree.
144  *
145  *      inner
146  *      +---+---+---+
147  *      | 10 | 20 |   |
148  *      +---+---+---+
149  *      /   |   \
150  *  /       |       \
151  * /         |         \
152  * +---+---+---+ +---+---+---+ +---+---+---+
153  * | 1 | 3 |   | |->| 11 | 12 | 13 | |->| 21 | 22 | 23 | |
154  * +---+---+---+ +---+---+---+ +---+---+---+
155  * leaf0         leaf1         leaf2
156  *
157  * Running inner.remove(1) on this tree would produce the following tree:
158  *
159  *      inner
160  *      +---+---+---+
161  *      | 10 | 20 |   |
162  *      +---+---+---+
163  *      /   |   \
164  *  /       |       \
165  * /         |         \
166  * +---+---+---+ +---+---+---+ +---+---+---+
167  * | 3 |   |   | |->| 11 | 12 | 13 | |->| 21 | 22 | 23 | |
168  * +---+---+---+ +---+---+---+ +---+---+---+
169  * leaf0         leaf1         leaf2
170  *
171  * Running inner.remove(3) would then produce the following tree:
172  *
173  *      inner
174  *      +---+---+---+
175  *      | 10 | 20 |   |
176  *      +---+---+---+
177  *      /   |   \
178  *  /       |       \
179  * /         |         \
180  * +---+---+---+ +---+---+---+ +---+---+---+
181  * |   |   |   | |->| 11 | 12 | 13 | |->| 21 | 22 | 23 | |
182  * +---+---+---+ +---+---+---+ +---+---+---+

```

```

183     * leaf0          leaf1          leaf2
184     *
185     * Again, do NOT rebalance the tree.
186     */
187     public abstract void remove(DataBox key);
188
189     // Helpers //////////////////////////////////////
190     /** Get the page on which this node is persisted. */
191     abstract Page getPage();
192
193     // Pretty Printing //////////////////////////////////////
194     /**
195      * S-expressions (or sexps) are a compact way of encoding nested tree-like
196      * structures (sort of like how JSON is a way of encoding nested dictionaries
197      * and lists). n.toSexp() returns an sexp encoding of the subtree rooted by
198      * n. For example, the following tree:
199      *
200      *           +---+
201      *           | 3 |
202      *           +---+
203      *          /   \
204      * +-----+ +-----+
205      * | 1:(1 1) | 2:(2 2) | | 3:(3 3) | 4:(4 4) |
206      * +-----+ +-----+
207      *
208      * has the following sexp
209      *
210      * (((1 (1 1)) (2 (2 2))) 3 ((3 (3 3)) (4 (4 4))))
211      *
212      * Here, (1 (1 1)) represents the mapping from key 1 to record id (1, 1).
213      */
214     public abstract String toSexp();
215
216     /**
217      * n.toDot() returns a fragment of a DOT file that draws the subtree rooted
218      * at n.
219      */
220     public abstract String toDot();
221
222     // Serialization //////////////////////////////////////
223     /** n.toBytes() serializes n. */
224     public abstract byte[] toBytes();
225
226     /**
227      * BPlusNode.fromBytes(m, p) loads a BPlusNode from page p of
228      * meta.getAllocator().
229      */
230     public static BPlusNode fromBytes(BPlusTreeMetadata metadata, int pageNum) {
231         Page p = metadata.getAllocator().fetchPage(pageNum);
232         ByteBuffer buf = p.getByteBuffer();
233         byte b = buf.get();
234         if (b == 1) {
235             return LeafNode.fromBytes(metadata, pageNum);
236         } else if (b == 0) {
237             return InnerNode.fromBytes(metadata, pageNum);
238         } else {
239             String msg = String.format("Unexpected byte %b.", b);
240             throw new IllegalArgumentException(msg);
241         }
242     }
243 }

```