dreamlegends init add all files 2c520b8 7 days ago

1 contributor

343 lines (308 sloc) 10.3 KB

```java
package index;

import java.nio.ByteBuffer;
import java.util.ArrayList;
import java.util.List;
import java.util.Objects;
import java.util.Optional;

import common.Pair;
import databox.DataBox;
import databox.Type;
import io.Page;
import table.RecordId;

/**
 * A inner node of a B+ tree. Every inner node in a B+ tree of order d stores
 * between d and 2d keys. An inner node with n keys stores n + 1 "pointers" to
 * children nodes (where a pointer is just a page number). Moreover, every
 * inner node is serialized and persisted on a single page; see toBytes and
 * fromBytes for details on how an inner node is serialized. For example, here
 * is an illustration of an order 2 inner node:
 *
 *     +----+----+----+----+
 *     | 10 | 20 | 30 |    |
 *     +----+----+----+----+
 *    /     |    |     \
 */
class InnerNode extends BPlusNode {
  // Metadata about the B+ tree that this node belongs to.
  private BPlusTreeMetadata metadata;

  // The page on which this leaf is serialized.
  private Page page;

  // The keys and child pointers of this inner node. See the comment above
  // LeafNode.keys and LeafNode.rids in LeafNode.java for a warning on the
  // difference between the keys and children here versus the keys and children
  // stored on disk.
  private List<DataBox> keys;
  private List<Integer> children;

  // Constructors //////////////////////////////////////////////////////////////
  /**
   * Construct a brand new inner node. The inner node will be persisted on a
   * brand new page allocated by metadata.getAllocator().
   */
  public InnerNode(BPlusTreeMetadata metadata, List<DataBox> keys,
                   List<Integer> children) {
    this(metadata, metadata.getAllocator().allocPage(), keys, children);
  }

  /**
   * Construct an inner node that is persisted to page `pageNum` allocated by
   * metadata.getAllocator().
   */
  private InnerNode(BPlusTreeMetadata metadata, int pageNum, List<DataBox> keys,
                    List<Integer> children) {
    assert(keys.size() <= 2 * metadata.getOrder());
    assert(keys.size() + 1 == children.size());

    this.metadata = metadata;
    this.page = metadata.getAllocator().fetchPage(pageNum);
    this.keys = keys;
    this.children = children;
    sync();
  }

  // Core API //////////////////////////////////////////////////////////////////
  // See BPlusNode.get.
  @Override
  public LeafNode get(DataBox key) {
    throw new UnsupportedOperationException("TODO: implement");
  }

  // See BPlusNode.getLeftmostLeaf.
  @Override
  public LeafNode getLeftmostLeaf() {
    throw new UnsupportedOperationException("TODO: implement");
  }

  // See BPlusNode.put.
  @Override
  public Optional<Pair<DataBox, Integer>> put(DataBox key, RecordId rid)
      throws BPlusTreeException {
    throw new UnsupportedOperationException("TODO: implement");
  }

  // See BPlusNode.remove.
  @Override
  public void remove(DataBox key) {
    throw new UnsupportedOperationException("TODO: implement");
  }

  // Helpers ///////////////////////////////////////////////////////////////////
  @Override
  public Page getPage() {
    return page;
  }

  private BPlusNode getChild(int i) {
    int pageNum = children.get(i);
    return BPlusNode.fromBytes(metadata, pageNum);
  }

  private void sync() {
    page.getByteBuffer().put(toBytes());
  }

  // Just for testing.
  List<DataBox> getKeys() {
    return keys;
  }
  // Just for testing.
  List<Integer> getChildren() {
    return children;
  }

  /**
   * Returns the largest number d such that the serialization of an InnerNode
   * with 2d keys will fit on a single page of size `pageSizeInBytes`.
   */
  public static int maxOrder(int pageSizeInBytes, Type keySchema) {
    // A leaf node with n entries takes up the following number of bytes:
    //
    //   1 + 4 + (n * keySize) + ((n + 1) * 4)
    //
    // where
    //
    //   - 1 is the number of bytes used to store isLeaf,
    //   - 4 is the number of bytes used to store n,
    //   - keySize is the number of bytes used to store a DataBox of type
    //     keySchema, and
    //   - 4 is the number of bytes used to store a child pointer.
    //
    // Solving the following equation
    //
    //   5 + (n * keySize) + ((n + 1) * 4) <= pageSizeInBytes
    //
    // we get
    //
    //   n = (pageSizeInBytes - 9) / (keySize + 4)
    //
    // The order d is half of n.
    int keySize = keySchema.getSizeInBytes();
    int n = (pageSizeInBytes - 9) / (keySize + 4);
    return n / 2;
  }

  /**
   * Given a list ys sorted in ascending order, numLessThanEqual(x, ys) returns
   * the number of elements in ys that are less than or equal to x. For
   * example,
   *
   *   numLessThanEqual(0, Arrays.asList(1, 2, 3, 4, 5)) == 0
   *   numLessThanEqual(1, Arrays.asList(1, 2, 3, 4, 5)) == 1
   *   numLessThanEqual(2, Arrays.asList(1, 2, 3, 4, 5)) == 2
   *   numLessThanEqual(3, Arrays.asList(1, 2, 3, 4, 5)) == 3
   *   numLessThanEqual(4, Arrays.asList(1, 2, 3, 4, 5)) == 4
   *   numLessThanEqual(5, Arrays.asList(1, 2, 3, 4, 5)) == 5
   *   numLessThanEqual(6, Arrays.asList(1, 2, 3, 4, 5)) == 5
   *
   * This helper function is useful when we're navigating down a B+ tree and
   * need to decide which child to visit. For example, imagine an index node
   * with the following 4 keys and 5 children pointers:
   *
   *     +---+---+---+---+
   *     | a | b | c | d |
   *     +---+---+---+---+
   *    /    |   |   |    \
   *   0     1   2   3     4
   *
   * If we're searching the tree for value c, then we need to visit child 3.
   * Not coincidentally, there are also 3 values less than or equal to c (i.e.
   * a, b, c).
   */
  public static <T extends Comparable<T>> int numLessThanEqual(T x, List<T> ys) {
    int n = 0;
    for (T y : ys) {
      if (y.compareTo(x) <= 0) {
        ++n;
      } else {
```

```java
        break;
      }
    }
    return n;
  }

  /** Same as numLessThanEqual but for < instead of <= */
  public static <T extends Comparable<T>> int numLessThan(T x, List<T> ys) {
    int n = 0;
    for (T y : ys) {
      if (y.compareTo(x) < 0) {
        ++n;
      } else {
        break;
      }
    }
    return n;
  }

  // Pretty Printing ////////////////////////////////////////////////////////////
  @Override
  public String toString() {
    String s = "(";
    for (int i = 0; i < keys.size(); ++i) {
      s += children.get(i) + " " + keys.get(i) + " ";
    }
    s += children.get(children.size() - 1) + ")";
    return s;
  }

  @Override
  public String toSexp() {
    String s = "(";
    for (int i = 0; i < keys.size(); ++i) {
      s += getChild(i).toSexp();
      s += " " + keys.get(i) + " ";
    }
    s += getChild(children.size() - 1).toSexp() + ")";
    return s;
  }

  /**
   * An inner node on page 0 with a single key k and two children on page 1 and
   * 2 is turned into the following DOT fragment:
   *
   *   node0[label = "<f0>|k|<f1>"];
   *   ... // children
   *   "node0":f0 -> "node1";
   *   "node0":f1 -> "node2";
   */
  @Override
  public String toDot() {
    List<String> ss = new ArrayList<>();
    for (int i = 0; i < keys.size(); ++i) {
      ss.add(String.format("<f%d>", i));
      ss.add(keys.get(i).toString());
    }
    ss.add(String.format("<f%d>", keys.size()));

    int pageNum = getPage().getPageNum();
    String s = String.join("|", ss);
    String node = String.format("  node%d[label = \"%s\"];", pageNum, s);

    List<String> lines = new ArrayList<>();
    lines.add(node);
    for (int i = 0; i < children.size(); ++i) {
      BPlusNode child = getChild(i);
      int childPageNum = child.getPage().getPageNum();
      lines.add(child.toDot());
      lines.add(String.format("  \"node%d\":f%d -> \"node%d\";",
                              pageNum, i, childPageNum));
    }

    return String.join("\n", lines);
  }

  // Serialization ///////////////////////////////////////////////////////////
  @Override
  public byte[] toBytes() {
    // When we serialize an inner node, we write:
    //
    //   a. the literal value 0 (1 byte) which indicates that this node is not
    //      a leaf node,
    //   b. the number n (4 bytes) of keys this inner node contains (which is
    //      one fewer than the number of children pointers),
    //   c. the n keys, and
    //   d. the n+1 children pointers.
    //
    // For example, the following bytes:
    //
    //   +----+-------------+----+-------------+-------------+
    //   | 00 | 00 00 00 01 | 01 | 00 00 00 03 | 00 00 00 07 |
    //   +----+-------------+----+-------------+-------------+
    //    \__/ \___________/ \__/ \_________________________/
    //     a    b             c    d
    //
    // represent an inner node with one key (i.e. 1) and two children pointers
    // (i.e. page 3 and page 7).

    // All sizes are in bytes.
    int isLeafSize = 1;
    int numKeysSize = Integer.BYTES;
    int keysSize = metadata.getKeySchema().getSizeInBytes() * keys.size();
    int childrenSize = Integer.BYTES * children.size();
    int size = isLeafSize + numKeysSize + keysSize + childrenSize;

    ByteBuffer buf = ByteBuffer.allocate(size);
    buf.put((byte) 0);
    buf.putInt(keys.size());
    for (DataBox key : keys) {
      buf.put(key.toBytes());
    }
    for (Integer child : children) {
      buf.putInt(child);
    }
    return buf.array();
  }

  /**
   * InnerNode.fromBytes(m, p) loads a InnerNode from page p of
   * meta.getAllocator().
   */
  public static InnerNode fromBytes(BPlusTreeMetadata metadata, int pageNum) {
    Page page = metadata.getAllocator().fetchPage(pageNum);
    ByteBuffer buf = page.getByteBuffer();

    assert(buf.get() == (byte) 0);

    List<DataBox> keys = new ArrayList<>();
    List<Integer> children = new ArrayList<>();
    int n = buf.getInt();
    for (int i = 0; i < n; ++i) {
      keys.add(DataBox.fromBytes(buf, metadata.getKeySchema()));
    }
    for (int i = 0; i < n + 1; ++i) {
      children.add(buf.getInt());
    }
    return new InnerNode(metadata, pageNum, keys, children);
  }

  // Builtins ///////////////////////////////////////////////////////////////
  @Override
  public boolean equals(Object o) {
    if (o == this) {
      return true;
    }
    if (!(o instanceof InnerNode)) {
      return false;
    }
    InnerNode n = (InnerNode) o;
    return page.getPageNum() == n.page.getPageNum() &&
           keys.equals(n.keys) &&
           children.equals(n.children);
  }

  @Override
  public int hashCode() {
    return Objects.hash(page.getPageNum(), keys, children);
  }
}
```