dreamlegends / DB2

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

Find file Copy path

dreamlegends init add all files

2c520b8 7 days ago

1 contributor

370 lines (333 sloc) 12.3 KB

Raw Blame History

```java
package index;

import java.nio.ByteBuffer;
import java.util.ArrayList;
import java.util.Iterator;
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 leaf of a B+ tree. Every leaf in a B+ tree of order d stores between d and
 * 2d (key, record id) pairs and a pointer to its right sibling (i.e. the page
 * number of its right sibling). Moreover, every leaf node is serialized and
 * persisted on a single page; see toBytes and fromBytes for details on how a
 * leaf is serialized. For example, here is an illustration of two order 2
 * leafs connected together:
 *
 *   leaf 1 (stored on some page)          leaf 2 (stored on some other page)
 *   +-------+-------+-------+-------+     +-------+-------+-------+-------+
 *   | k0:r0 | k1:r1 | k2:r2 |       | --> | k3:r3 | k4:r4 |       |       |
 *   +-------+-------+-------+-------+     +-------+-------+-------+-------+
 */
class LeafNode 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 record ids of this leaf. `keys` is always sorted in ascending
  // order. The record id at index i corresponds to the key at index i. For
  // example, the keys [a, b, c] and the rids [1, 2, 3] represent the pairing
  // [a:1, b:2, c:3].
  //
  // Note the following subtlety. keys and rids are in-memory caches of the
  // keys and record ids stored on disk. Thus, consider what happens when you
  // create two LeafNode objects that point to the same page:
  //
```

```
//   BPlusTreeMetadata meta = ...;
//   int pageNum = ...;
//   Page page = allocator.fetchPage(pageNum);
//   ByteBuffer buf = page.getByteBuffer();
//
//   LeafNode leaf0 = LeafNode.fromBytes(buf, meta, pageNum);
//   LeafNode leaf1 = LeafNode.fromBytes(buf, meta, pageNum);
//
// This scenario looks like this:
//
//   HEAP                        | DISK
//   ===============================================================
//   leaf0                       | page 42
//   +-------------------------+ | +-------+-------+-------+-------+
//   | keys = [k0, k1, k2]     | | | k0:r0 | k1:r1 | k2:r2 |       |
//   | rids = [r0, r1, r2]     | | +-------+-------+-------+-------+
//   | pageNum = 42            | |
//   +-------------------------+ |
//                               |
//   leaf1                       |
//   +-------------------------+ |
//   | keys = [k0, k1, k2]     | |
//   | rids = [r0, r1, r2]     | |
//   | pageNum = 42            | |
//   +-------------------------+ |
//                               |
//
// Now imagine we perform on operation on leaf0 like leaf0.put(k3, r3). The
// in-memory values of leaf0 will be updated and they will be synced to disk.
// But, the in-memory values of leaf1 will not be updated. That will look
// like this:
//
//   HEAP                        | DISK
//   ===============================================================
//   leaf0                       | page 42
//   +-------------------------+ | +-------+-------+-------+-------+
//   | keys = [k0, k1, k2, k3] | | | k0:r0 | k1:r1 | k2:r2 | k3:r3 |
//   | rids = [r0, r1, r2, r3] | | +-------+-------+-------+-------+
//   | pageNum = 42            | |
//   +-------------------------+ |
//                               |
//   leaf1                       |
//   +-------------------------+ |
//   | keys = [k0, k1, k2]     | |
//   | rids = [r0, r1, r2]     | |
//   | pageNum = 42            | |
//   +-------------------------+ |
//                               |
//
// Make sure your code (or your tests) doesn't use stale in-memory cached
// values of keys and rids.
private List<DataBox> keys;
private List<RecordId> rids;

// If this leaf is the rightmost leaf, then rightSibling is Optional.empty().
// Otherwise, rightSibling is Optional.of(n) where n is the page number of
// this leaf's right sibling.
private Optional<Integer> rightSibling;

// Constructors //////////////////////////////////////////////////////////////
/**
 * Construct a brand new leaf node. The leaf will be persisted on a brand new
 * page allocated by metadata.getAllocator().
 */
public LeafNode(BPlusTreeMetadata metadata, List<DataBox> keys,
                List<RecordId> rids, Optional<Integer> rightSibling) {
  this(metadata, metadata.getAllocator().allocPage(), keys, rids,
       rightSibling);
}
```

```java
/**
 * Construct a leaf node that is persisted to page `pageNum` allocated by
 * metadata.getAllocator().
 */
private LeafNode(BPlusTreeMetadata metadata, int pageNum, List<DataBox> keys,
                 List<RecordId> rids, Optional<Integer> rightSibling) {
  assert(keys.size() <= 2 * metadata.getOrder());
  assert(keys.size() == rids.size());

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

// Core API //////////////////////////////////////////////////////////////////
// See BPlusNode.get.
@Override
public LeafNode get(DataBox key) {
  return this;
}

// See BPlusNode.getLeftmostLeaf.
@Override
public LeafNode getLeftmostLeaf() {
  return this;
}

// 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");
}

// Iterators /////////////////////////////////////////////////////////////////
/** Return the record id associated with `key`. */
public Optional<RecordId> getKey(DataBox key) {
  throw new UnsupportedOperationException("TODO: implement");
}

/**
 * Returns an iterator over the record ids of this leaf in ascending order of
 * their corresponding keys.
 */
public Iterator<RecordId> scanAll() {
  return rids.iterator();
}

/**
 * Returns an iterator over the record ids of this leaf that have a
 * corresponding key greater than or equal to `key`. The record ids are
 * returned in ascending order of their corresponding keys.
 */
public Iterator<RecordId> scanGreaterEqual(DataBox key) {
  int index = InnerNode.numLessThan(key, keys);
  return rids.subList(index, rids.size()).iterator();
}

// Helpers ///////////////////////////////////////////////////////////////////
```

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

  /** Returns the right sibling of this leaf, if it has one. */
  public Optional<LeafNode> getRightSibling() {
    if (!rightSibling.isPresent()) {
      return Optional.empty();
    }

    int pageNum = rightSibling.get();
    return Optional.of(LeafNode.fromBytes(metadata, pageNum));
  }

  /** Serializes this leaf to its page. */
  private void sync() {
    page.getByteBuffer().put(toBytes());
  }

  /**
   * Returns the largest number d such that the serialization of a LeafNode
   * with 2d entries 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 + 4 + n * (keySize + ridSize)
    //
    // where
    //
    //   - 1 is the number of bytes used to store isLeaf,
    //   - 4 is the number of bytes used to store a sibling pointer,
    //   - 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
    //   - ridSize is the number of bytes of a RecordId.
    //
    // Solving the following equation
    //
    //   n * (keySize + ridSize) + 9 <= pageSizeInBytes
    //
    // we get
    //
    //   n = (pageSizeInBytes - 9) / (keySize + ridSize)
    //
    // The order d is half of n.
    int keySize = keySchema.getSizeInBytes();
    int ridSize = RecordId.getSizeInBytes();
    int n = (pageSizeInBytes - 9) / (keySize + ridSize);
    return n / 2;
  }

  // For testing only.
  List<DataBox> getKeys() {
    return keys;
  }

  // For testing only.
  List<RecordId> getRids() {
    return rids;
  }

  // Pretty Printing ///////////////////////////////////////////////////////////
  @Override
  public String toString() {
    return String.format("LeafNode(pageNum=%s, keys=%s, rids=%s)",
                         page.getPageNum(), keys, rids);
  }
  @Override
  public String toSexp() {
    List<String> ss = new ArrayList<>();
    for (int i = 0; i < keys.size(); ++i) {
      String key = keys.get(i).toString();
      String rid = rids.get(i).toSexp();
      ss.add(String.format("(%s %s)", key, rid));
    }
    return String.format("(%s)", String.join(" ", ss));
  }

  /**
   * Given a leaf with page number 1 and three (key, rid) pairs (0, (0, 0)),
   * (1, (1, 1)), and (2, (2, 2)), the corresponding dot fragment is:
   *
   *   node1[label = "{0: (0 0)|1: (1 1)|2: (2 2)}"];
   */
  @Override
  public String toDot() {
    List<String> ss = new ArrayList<>();
    for (int i = 0; i < keys.size(); ++i) {
      ss.add(String.format("%s: %s", keys.get(i), rids.get(i).toSexp()));
    }
    int pageNum = getPage().getPageNum();
    String s = String.join("|", ss);
    return String.format("  node%d[label = \"{%s}\"];", pageNum, s);
  }

  // Serialization ///////////////////////////////////////////////////////////
  @Override
  public byte[] toBytes() {
    // When we serialize a leaf node, we write:
    //
    //   a. the literal value 1 (1 byte) which indicates that this node is a
    //      leaf node,
    //   b. the page id (4 bytes) of our right sibling (or -1 if we don't have
    //      a right sibling),
    //   c. the number (4 bytes) of (key, rid) pairs this leaf node contains,
    //      and
    //   d. the (key, rid) pairs themselves.
    //
    // For example, the following bytes:
    //
    //   +----+-------------+-------------+----+-------------------+
    //   | 01 | 00 00 00 04 | 00 00 00 01 | 03 | 00 00 00 03 00 01 |
    //   +----+-------------+-------------+----+-------------------+
    //    \__/ \___________/ \___________/ \______________________/
    //     a    b             c             d
    //
    // represent a leaf node with sibling on page 4 and a single (key, rid)
    // pair with key 3 and page id (3, 1).

    // All sizes are in bytes.
    int isLeafSize = 1;
    int siblingSize = Integer.BYTES;
    int lenSize = Integer.BYTES;
    int keySize = metadata.getKeySchema().getSizeInBytes();
    int ridSize = RecordId.getSizeInBytes();
    int entriesSize = (keySize + ridSize) * keys.size();
    int size = isLeafSize + siblingSize + lenSize + entriesSize;

    ByteBuffer buf = ByteBuffer.allocate(size);
    buf.put((byte) 1);
    buf.putInt(rightSibling.orElse(-1));
    buf.putInt(keys.size());
    for (int i = 0; i < keys.size(); ++i) {
      buf.put(keys.get(i).toBytes());
      buf.put(rids.get(i).toBytes());
    }
    return buf.array();
  }

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

    assert(buf.get() == (byte) 1);

    int s = buf.getInt();
    Optional<Integer> rightSibling = s == -1 ? Optional.empty() : Optional.of(s);

    List<DataBox> keys = new ArrayList<>();
    List<RecordId> rids = new ArrayList<>();
    int n = buf.getInt();
    for (int i = 0; i < n; ++i) {
      keys.add(DataBox.fromBytes(buf, metadata.getKeySchema()));
      rids.add(RecordId.fromBytes(buf));
    }

    return new LeafNode(metadata, pageNum, keys, rids, rightSibling);
  }

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

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