best99317 / **Fa18HW5**

Branch: master ▾ Fa18HW5 / src / main / java / edu / berkeley / cs186 / database / index / **LeafNode.java** Find file Copy path

es1024 HW5 skeleton code 16ecdae on Nov 8, 2018

1 contributor

480 lines (430 sloc) 17.9 KB Raw Blame History

```java
package edu.berkeley.cs186.database.index;

import java.nio.ByteBuffer;
import java.util.*;

import edu.berkeley.cs186.database.BaseTransaction;
import edu.berkeley.cs186.database.common.Buffer;
import edu.berkeley.cs186.database.common.Pair;
import edu.berkeley.cs186.database.databox.DataBox;
import edu.berkeley.cs186.database.databox.Type;
import edu.berkeley.cs186.database.io.Page;
import edu.berkeley.cs186.database.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, BaseTransaction transaction) {
        this(metadata, metadata.getAllocator().allocPage(transaction), keys, rids,
             rightSibling, transaction);
```

```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, BaseTransaction transaction) {
        assert(keys.size() == rids.size());

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

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

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

    // See BPlusNode.put.
    @Override
    public Optional<Pair<DataBox, Integer>> put(BaseTransaction transaction, DataBox key, RecordId rid)
    throws BPlusTreeException {
        // Our implementation of B+ trees does not support duplicates!
        if (keys.contains(key)) {
            String message = String.format("Duplicate key %s inserted.", key);
            throw new BPlusTreeException(message);
        }

        // Insert the new key and record id into the leaf node. For example, we
        // might go from a leaf node which looks like this:
        //
        //   +-------+-------+-------+-------+
        //   | k1:r1 | k2:r2 | k3:r3 | k5:r5 |
        //   +-------+-------+-------+-------+
        //
        // to one which looks like this:
        //
        //   +-------+-------+-------+-------+-------+
        //   | k1:r1 | k2:r2 | k3:r3 | k4:r4 | k5:r5 |
        //   +-------+-------+-------+-------+-------+
        //
        // In this example, put was called with key k4 and record id r4.
        int index = InnerNode.numLessThanEqual(key, keys);
        keys.add(index, key);
        rids.add(index, rid);

        // If we can accommodate the new key and record id (i.e. the number of
        // entries does not exceed 2d), then we're done (just don't forget to
        // sync)!
        int d = metadata.getOrder();
        if (keys.size() <= 2 * d) {
            sync(transaction);
            return Optional.empty();
        }
        // If our leaf node overflows (i.e. we have 2d + 1 entries), then we have
        // to split the leaf node. We put d entries on the left and d + 1 entries
        // on the right. Continuing our example from above, we would split into the
        // following two leaf nodes:
        //
        //   left                  right
        //   +-------+-------+  +-------+-------+-------+
        //   | k1:r1 | k2:r2 |  | k3:r3 | k4:r4 | k5:r5 |
        //   +-------+-------+  +-------+-------+-------+
        //
        // and we would return the pair (k3, right).
        assert(keys.size() == 2 * d + 1);
        List<DataBox> leftKeys = keys.subList(0, d);
        List<DataBox> rightKeys = keys.subList(d, 2 * d + 1);
        List<RecordId> leftRids  = rids.subList(0, d);
        List<RecordId> rightRids  = rids.subList(d, 2 * d + 1);

        // Create right node.
        LeafNode n = new LeafNode(metadata, rightKeys, rightRids, rightSibling, transaction);
        int pageNum = n.getPage().getPageNum();

        // Update left node.
        this.keys = leftKeys;
        this.rids = leftRids;
        this.rightSibling = Optional.of(pageNum);
        sync(transaction);

        return Optional.of(new Pair<>(rightKeys.get(0), pageNum));
    }

    // See BPlusNode.bulkLoad.
    @Override
    public Optional<Pair<DataBox, Integer>> bulkLoad(BaseTransaction transaction,
            Iterator<Pair<DataBox, RecordId>> data,
            float fillFactor)
    throws BPlusTreeException {
        int d = metadata.getOrder();
        if (fillFactor * 2 * d <= 0) {
            throw new BPlusTreeException("Cannot bulk-load to empty leaves.");
        }

        int numKeys = (int) Math.ceil(2 * d * fillFactor);
        for (int i = keys.size(); i < numKeys && data.hasNext(); ++i) {
            Pair<DataBox, RecordId> pair = data.next();
            keys.add(pair.getFirst());
            rids.add(pair.getSecond());
        }

        if (!data.hasNext()) {
            sync(transaction);
            return Optional.empty();
        }

        List<DataBox> rightKeys = new ArrayList<>();
        List<RecordId> rightRids = new ArrayList<>();
        Pair<DataBox, RecordId> pair = data.next();
        rightKeys.add(0, pair.getFirst());
        rightRids.add(0, pair.getSecond());

        // Create right node.
        LeafNode n = new LeafNode(metadata, rightKeys, rightRids, Optional.empty(), transaction);
        int pageNum = n.getPage().getPageNum();

        // Update left node.
        this.rightSibling = Optional.of(pageNum);
        sync(transaction);
        return Optional.of(new Pair<>(rightKeys.get(0), pageNum));
    }

    // See BPlusNode.remove.
    @Override
    public void remove(BaseTransaction transaction, DataBox key) {
        int index = keys.indexOf(key);
        if (index != -1) {
            keys.remove(index);
            rids.remove(index);
        }
        sync(transaction);
    }

    // Iterators ////////////////////////////////////////////////////////////////
    /** Return the record id associated with `key`. */
    public Optional<RecordId> getKey(DataBox key) {
        int index = keys.indexOf(key);
        return index == -1 ? Optional.empty() : Optional.of(rids.get(index));
    }

    /**
     * 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(BaseTransaction transaction) {
        if (!rightSibling.isPresent()) {
            return Optional.empty();
        }

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

    /** Serializes this leaf to its page. */
    private void sync(BaseTransaction transaction) {
        Buffer b = page.getBuffer(transaction);
        byte[] newBytes = toBytes();
        byte[] bytes = new byte[newBytes.length];
        b.get(bytes);
        if (!Arrays.equals(bytes, newBytes)) {
            page.getBuffer(transaction).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(BaseTransaction transaction) {
        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(BaseTransaction transaction) {
        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(BaseTransaction transaction, BPlusTreeMetadata metadata,
                                     int pageNum) {
        Page page = metadata.getAllocator().fetchPage(transaction, pageNum);
        Buffer buf = page.getBuffer(transaction);

        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, transaction);
    }

    // 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);
    }
}
```