es1024 HW5 skeleton code
16ecdae on Nov 8, 2018

1 contributor

463 lines (415 sloc) 16.4 KB Raw Blame History

```
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 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, BaseTransaction transaction) {
        this(metadata, metadata.getAllocator().allocPage(transaction), keys, children, transaction);
    }

    /**
     * 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, BaseTransaction transaction) {
        assert(keys.size() <= 2 * metadata.getOrder());
        assert(keys.size() + 1 == children.size());

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

    // Core API ////////////////////////////////////////////////////////////////
    // See BPlusNode.get.
    @Override
    public LeafNode get(BaseTransaction transaction, DataBox key) {
        int index = numLessThanEqual(key, keys);
        BPlusNode child = getChild(transaction, index);
        return child.get(transaction, key);
    }

    // See BPlusNode.getLeftmostLeaf.
    @Override
    public LeafNode getLeftmostLeaf(BaseTransaction transaction) {
        assert(children.size() > 0);
        BPlusNode child = getChild(transaction, 0);
        return child.getLeftmostLeaf(transaction);
    }

    // See BPlusNode.put.
    @Override
    public Optional<Pair<DataBox, Integer>> put(BaseTransaction transaction, DataBox key, RecordId rid)
    throws BPlusTreeException {
        int index = numLessThanEqual(key, keys);
        BPlusNode child = getChild(transaction, index);
        Optional<Pair<DataBox, Integer>> o = child.put(transaction, key, rid);

        // If our child didn't split, then we don't have to do anything.
        if (!o.isPresent()) {
            return Optional.empty();
        }

        // If our child did split, then we have to insert (a) the new key and (b)
        // the pointer to the newly created child. For example, we might go from an
        // index which looks like this:
        //
        //     +---+---+---+---+
        //     | a | b | c | e |
        //     +---+---+---+---+
        //    /    |   |   |    \
        //   0     1   2   3     5
        //
        // to an index which looks like this:
        //
        //     +---+---+---+---+---+
```

```
        //     | a | b | c | d | e |
        //     +---+---+---+---+---+
        //    /    |   |   |   |    \
        //   0     1   2   3   4     5
        //
        // Note that in this example, p = (d, 4).
        Pair<DataBox, Integer> p = o.get();
        keys.add(index, p.getFirst());
        children.add(index + 1, p.getSecond());

        // If we can accommodate the new key and child pointer (i.e. we don't have
        // more than 2d keys), then we're done (just don't forget to sync)!
        int d = metadata.getOrder();
        if (keys.size() <= 2 * d) {
            sync(transaction);
            return Optional.empty();
        }

        // On the other hand, if we overflow (i.e. we now have 2d + 1 keys), then
        // we have to split ourselves in two. Continuing the example from above
        // (with order 2), we would split ourselves into the following two inner
        // nodes:
        //
        //     left           right
        //     +---+---+      +---+---+
        //     | a | b |      | d | e |
        //     +---+---+      +---+---+
        //    /    |    \    /    |    \
        //   0     1     2  3     4     5
        //
        // We would then return the pair (c, left).
        assert(keys.size() == 2 * d + 1);
        List<DataBox> leftKeys = keys.subList(0, d);
        DataBox middleKey = keys.get(d);
        List<DataBox> rightKeys = keys.subList(d + 1, 2 * d + 1);
        List<Integer> leftChildren = children.subList(0, d + 1);
        List<Integer> rightChildren = children.subList(d + 1, 2 * d + 2);

        // Create right node.
        InnerNode n = new InnerNode(metadata, rightKeys, rightChildren, transaction);

        // Update left node.
        this.keys = leftKeys;
        this.children = leftChildren;
        sync(transaction);

        return Optional.of(new Pair<>(middleKey, n.getPage().getPageNum()));
    }

    // 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();
        while (data.hasNext() && keys.size() <= 2 * d) {
            BPlusNode rightChild = getChild(transaction, children.size() - 1);
            Optional<Pair<DataBox, Integer>> o = rightChild.bulkLoad(transaction, data, fillFactor);
            if (o.isPresent()) {
                Pair<DataBox, Integer> p = o.get();
                keys.add(keys.size(), p.getFirst());
                children.add(children.size(), p.getSecond());
            }
        }

        if (keys.size() <= 2 * d) {
```

```
            sync(transaction);
            return Optional.empty();
        }

        assert(keys.size() == 2 * d + 1);
        List<DataBox> leftKeys = keys.subList(0, d);
        DataBox middleKey = keys.get(d);
        List<DataBox> rightKeys = keys.subList(d + 1, 2 * d + 1);
        List<Integer> leftChildren = children.subList(0, d + 1);
        List<Integer> rightChildren = children.subList(d + 1, 2 * d + 2);

        // Create right node.
        InnerNode n = new InnerNode(metadata, rightKeys, rightChildren, transaction);

        // Update left node.
        this.keys = leftKeys;
        this.children = leftChildren;
        sync(transaction);

        return Optional.of(new Pair<>(middleKey, n.getPage().getPageNum()));
    }

    // See BPlusNode.remove.
    @Override
    public void remove(BaseTransaction transaction, DataBox key) {
        int index = numLessThanEqual(key, keys);
        BPlusNode child = getChild(transaction, index);
        child.remove(transaction, key);
    }

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

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

    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());
        }
    }

    // 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 {
            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(BaseTransaction transaction) {
        String s = "(";
        for (int i = 0; i < keys.size(); ++i) {
            s += getChild(transaction, i).toSexp(transaction);
            s += " " + keys.get(i) + " ";
        }
        s += getChild(transaction, children.size() - 1).toSexp(transaction) + ")";
        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(BaseTransaction transaction) {
        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(transaction, i);
            int childPageNum = child.getPage().getPageNum();
            lines.add(child.toDot(transaction));
            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(t, meta, p) loads a InnerNode from page p of
     * meta.getAllocator().
     */
    public static InnerNode fromBytes(BaseTransaction transaction, BPlusTreeMetadata metadata,
                                      int pageNum) {
        Page page = metadata.getAllocator().fetchPage(transaction, pageNum);
        Buffer buf = page.getBuffer(transaction);

        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, transaction);
    }

    // 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);
    }
}
```