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

es1024 Fixed testQueryWithIndex/testSortedScanNoIndexLocking failing with hw... c35db97 on Nov 11, 2018

1 contributor

240 lines (216 sloc) 9.35 KB Raw Blame History

```
package edu.berkeley.cs186.database.query;

import edu.berkeley.cs186.database.Database;
import edu.berkeley.cs186.database.DatabaseException;
import edu.berkeley.cs186.database.concurrency.LockType;
import edu.berkeley.cs186.database.concurrency.LockUtil;
import edu.berkeley.cs186.database.databox.DataBox;
import edu.berkeley.cs186.database.table.Record;
import edu.berkeley.cs186.database.table.Schema;
import edu.berkeley.cs186.database.table.stats.TableStats;
import edu.berkeley.cs186.database.table.stats.Histogram;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.NoSuchElementException;

public class IndexScanOperator extends QueryOperator {
    private Database.Transaction transaction;
    private String tableName;
    private String columnName;
    private QueryPlan.PredicateOperator predicate;
    private DataBox value;

    private int columnIndex;

    /**
     * An index scan operator.
     *
     * @param transaction the transaction containing this operator
     * @param tableName the table to iterate over
     * @param columnName the name of the column the index is on
     * @throws QueryPlanException
     * @throws DatabaseException
     */
    public IndexScanOperator(Database.Transaction transaction,
                             String tableName,
                             String columnName,
                             QueryPlan.PredicateOperator predicate,
                             DataBox value) throws QueryPlanException, DatabaseException {
        super(OperatorType.INDEXSCAN);
        this.tableName = tableName;
        this.transaction = transaction;
```

```
        this.columnName = columnName;
        this.predicate = predicate;
        this.value = value;
        this.setOutputSchema(this.computeSchema());
        columnName = this.checkSchemaForColumn(this.getOutputSchema(), columnName);
        this.columnIndex = this.getOutputSchema().getFieldNames().indexOf(columnName);

        /*
        this.stats = this.estimateStats();
        this.cost = this.estimateIOCost();
        */
    }

    public String str() {
        return "type: " + this.getType() +
               "\ntable: " + this.tableName +
               "\ncolumn: " + this.columnName +
               "\noperator: " + this.predicate +
               "\nvalue: " + this.value;
    }

    /**
     * Returns the column name that the index scan is on
     *
     * @return columnName
     */
    public String getColumnName() {
        return this.columnName;
    }

    /**
     * Estimates the table statistics for the result of executing this query operator.
     *
     * @return estimated TableStats
     */
    public TableStats estimateStats() throws QueryPlanException {
        TableStats stats;

        try {
            stats = this.transaction.getStats(this.tableName);
        } catch (DatabaseException de) {
            throw new QueryPlanException(de);
        }

        return stats.copyWithPredicate(this.columnIndex,
                                       this.predicate,
                                       this.value);
    }

    /**
     * Estimates the IO cost of executing this query operator.
     * You should calculate this estimate cost with the formula
     * taught to you in class. Note that the index you've implemented
     * in this project is an unclustered index.
     *
     * You will find the following instance variables helpful:
     * this.transaction, this.tableName, this.columnName,
     * this.columnIndex, this.predicate, and this.value.
     *
     * You will find the following methods helpful: this.transaction.getStats,
     * this.transaction.getNumRecords, this.transaction.getNumIndexPages,
     * and tableStats.getReductionFactor.
     *
     * @return estimate IO cost
     * @throws QueryPlanException
     */
    public int estimateIOCost() throws QueryPlanException {
```

```
        long numRecords;
        long numIndexPages;
        TableStats tableStats;
        try {
            numRecords = this.transaction.getNumRecords(this.tableName);
            numIndexPages = this.transaction.getNumIndexPages(this.tableName, this.columnName);
            tableStats = this.transaction.getStats(this.tableName);

        } catch (DatabaseException err) {
            throw new QueryPlanException("Can't find the number of records in IndexScanOperator#estimateIOCost().");

        }

        return (int)(tableStats.getHistograms().get(columnIndex).getCount() +
                     numIndexPages); //round up and cast to an int
    }

    public Iterator<Record> iterator() throws QueryPlanException, DatabaseException {
        return new IndexScanIterator();
    }

    public Schema computeSchema() throws QueryPlanException {
        try {
            return this.transaction.getFullyQualifiedSchema(this.tableName);
        } catch (DatabaseException de) {
            throw new QueryPlanException(de);
        }
    }

    /**
     * An implementation of Iterator that provides an iterator interface for this operator.
     */
    private class IndexScanIterator implements Iterator<Record> {
        private Iterator<Record> sourceIterator;
        private Record nextRecord;

        public IndexScanIterator() throws QueryPlanException, DatabaseException {
            this.nextRecord = null;
            if (IndexScanOperator.this.predicate == QueryPlan.PredicateOperator.EQUALS) {
                this.sourceIterator = IndexScanOperator.this.transaction.lookupKey(
                                          IndexScanOperator.this.tableName,
                                          IndexScanOperator.this.columnName,
                                          IndexScanOperator.this.value);
            } else if (IndexScanOperator.this.predicate == QueryPlan.PredicateOperator.LESS_THAN ||
                       IndexScanOperator.this.predicate == QueryPlan.PredicateOperator.LESS_THAN_EQUALS) {
                this.sourceIterator = IndexScanOperator.this.transaction.sortedScan(
                                          IndexScanOperator.this.tableName,
                                          IndexScanOperator.this.columnName);
            } else if (IndexScanOperator.this.predicate == QueryPlan.PredicateOperator.GREATER_THAN) {
                this.sourceIterator = IndexScanOperator.this.transaction.sortedScanFrom(
                                          IndexScanOperator.this.tableName,
                                          IndexScanOperator.this.columnName,
                                          IndexScanOperator.this.value);
                while (this.sourceIterator.hasNext()) {
                    Record r = this.sourceIterator.next();

                    if (r.getValues().get(IndexScanOperator.this.columnIndex)
                            .compareTo(IndexScanOperator.this.value) > 0) {
                        this.nextRecord = r;
                        break;
                    }
                }
            } else if (IndexScanOperator.this.predicate == QueryPlan.PredicateOperator.GREATER_THAN_EQUALS) {
                this.sourceIterator = IndexScanOperator.this.transaction.sortedScanFrom(
                                          IndexScanOperator.this.tableName,
                                          IndexScanOperator.this.columnName,
                                          IndexScanOperator.this.value);
```

```
            }
        }

        /**
         * Checks if there are more record(s) to yield
         *
         * @return true if this iterator has another record to yield, otherwise false
         */
        public boolean hasNext() {
            if (this.nextRecord != null) {
                return true;
            }
            if (IndexScanOperator.this.predicate == QueryPlan.PredicateOperator.LESS_THAN) {
                if (this.sourceIterator.hasNext()) {
                    Record r = this.sourceIterator.next();
                    if (r.getValues().get(IndexScanOperator.this.columnIndex)
                            .compareTo(IndexScanOperator.this.value) >= 0) {
                        return false;
                    }
                    this.nextRecord = r;
                    return true;
                }
                return false;
            } else if (IndexScanOperator.this.predicate == QueryPlan.PredicateOperator.LESS_THAN_EQUALS) {
                if (this.sourceIterator.hasNext()) {
                    Record r = this.sourceIterator.next();
                    if (r.getValues().get(IndexScanOperator.this.columnIndex)
                            .compareTo(IndexScanOperator.this.value) > 0) {
                        return false;
                    }
                    this.nextRecord = r;
                    return true;
                }
                return false;
            }
            if (this.sourceIterator.hasNext()) {
                this.nextRecord = this.sourceIterator.next();
                return true;
            }
            return false;
        }

        /**
         * Yields the next record of this iterator.
         *
         * @return the next Record
         * @throws NoSuchElementException if there are no more Records to yield
         */
        public Record next() {
            if (this.hasNext()) {
                Record r = this.nextRecord;
                this.nextRecord = null;
                return r;
            }
            throw new NoSuchElementException();
        }

        public void remove() {
            throw new UnsupportedOperationException();
        }
    }
}
```