Database Systems

Course instructor: Yanif Ahmad.
Course staff: P.C. Shyamshankar (Shyam) (TA).
Schedule: MW 12-1.15pm, Hackerman 320.

Office hours:
Yanif: T 5:00-7pm, Malone 328.
Shyam: T 5:00-7pm, Malone 328.

Assignment 2: Query Processing

Due date: 11:59pm Wednesday March 11th, 2014

Description | Logistics | Exercises | Relevant lectures

Description

Building on our storage engine, our second assignment will cover the query operator algorithms used to efficiently process SQL queries over out-of-core datasets. These algorithms provide a physical layer implementation of the relational algebra operations that comprise query plans. Our query plans are represented as tree data structures, with each operator in the tree supporting a common abstraction in the form of an iterator for each input. In this way, a query plan is processed starting at its root operator and its request for the first output tuple from its output iterator. This recursively requests the next tuple from each of operator iterators in the plan tree, with query processing terminating when all of these iterators are complete. We will be using set-at-a-time processing techniques, where each operator will output its intermediate results in a temporary file in the storage engine.

Our support code for this assignment includes the implementation of three basic operators: table scans, selects and projects. This will help you get started with simple queries such as `select id, age from employees where salary < 50000`. In this assignment, you will extend the set of primitive operators, providing the implementation of unions, group-by aggregations, and a variety of join algorithms (block nested loops, indexed, and hash joins). We will be using Berkeley DB to provide unclustered index data

structures in this assignment, where these indexes will contain tuple identifiers to reference data in our storage engine.

By the end of this assignment, you will have a working single-threaded database engine that supports data loading and storage, and data processing for a subset of SQL-style queries. We will then use this engine to study further topics such as query optimization and distributed data management.

As with Assignment 1, we will use the lab sessions to discuss the design of the codebase, and the concepts behind our query engine implementation. **We encourage you to work in pairs on this exercise, and get started early on this assignment (we estimate you will be writing approximately 1000-1500 lines of code).** We strongly recommend you think through the design and sketch out pseudocode rather than diving straight into the exercises. You are welcome to ask design questions on Piazza. Try to have Exercise 1 and 2 completed by the end of the first week.

Logistics

Support code The Python support code for this assignment can be found at **`~cs416/handouts/hw2/dbsys-hw2.tar.gz`** on the ugrad network. In addition to a sample solution to the prior storage engine exercise, the new support code includes a **Query/** directory:

```
dbsys-hw2/                ## Source code directory
|-- Database.py           ## Top-level database API
|-- Catalog/              ## Catalog as in HW1
|-- Storage/              ## Storage engine as in HW1
|---- BufferPool.py       ## New in this codebase: BufferPool with pinning
|---- Index/
|------ IndexManager.py   ## New in this codebase: indexes for joins.
|-- Query/
|---- Plan.py             ## Query plan data structure
|---- Operator.py         ## Abstract base class for operator implementations
|---- Operators/          ## Operator implementations directory
|------ TableScan.py      ## Scan operator, using file-based iterators
|------ Select.py         ## Filters the input tuples based on a predicate
|------ Project.py        ## Projects attributes, without eliminating duplicates
|------ Union.py          ## Unions two multisets, which must have the same schema
|------ Join.py           ## Four joins: nested, block nested, index, hash
|------ GroupBy.py        ## Group by aggregation
|------ Sort.py           ## External sorting
```

Creating query plans: As a starting point, we suggest understanding how we create query plans in our database engine. Query plan construction involves two classes, the `Query.Plan.PlanBuilder`, which produces a `Query.Plan.Plan` object. Our `PlanBuilder` instances provide high-level methods to construct plans corresponding to SQL-like statements. This can be seen in the doctests for the `PlanBuilder` class, from example to create a simple select-from-where query:

```
>>> query1 = db.query().fromTable('employee').where("age < 30").finalize()
```

Above `db` is an instance of our `Database` class (from **Database.py**) which provides basic data structures to track the schemas and relations present in the application. Its `query()` method returns a `PlanBuilder` object which can then be used to directly build up the query plan. Above, the `fromTable('employee')` method adds a table scan operation, and then adds a select operation through the `where('age < 30')` method. Both the `fromTable` and `where` methods return a new `PlanBuilder` object, allowing compositional plan construction. The `finalize` method returns the query tree as a `Plan` object from the `PlanBuilder` Loosely speaking, in this way we are building up query expression trees in a similar fashion to LINQ (Microsoft's Language Integrated Query).

A `PlanBuilder`'s query construction methods include:

- **`fromTable(tableName)`**: adds a table scan (`Query.Operators.TableScan`) to the query plan, to access an input table for the query. The `tableName` argument is a string.

- **`where(predicate)`**: adds a filtering operator to the plan. The predicate is a string defining a Python expression that must evaluate to a boolean. In our operator definition (in `Query.Operators.Select`), we use Python's builtin eval function to interpret this expression. The expression has access to the fields in its tuple, for instance in the above example, the `age` field is an attribute in the Employee table.

- **`select(projectionDict)`**: adds a projection operator the the plan. This takes a projection dictionary as an argument, containing mappings from output attribute names to projection expressions and their types. Here is an example projection dictionary:

  ```
  { 'xplus2'   : ('x+2', 'double'),
    'distance' : ('math.sqrt(x*x+y*y)', 'double') }
  ```

  Assuming the operator's input schema has fields `x,y` (both doubles), this projection dictionary defines an output schema containing an `xplus2` attribute, and a `distance` attribute. These fields are computed using the expression defined as the first component of the dictionary's values, with the second component defining the expected type of the value. The supported types in our database are defined in `Catalog.Schema.Types`.

- **`union(planBuilder)`**: adds a union operator to combine the results of two queries. We assume the results are multisets, thus our union does not eliminate duplicates (i.e., UNION ALL in SQL). Union is a binary operator, and the plan builder on which we invoke the union method forms the left-hand side input to this binary operator. The argument is another `PlanBuilder` object, indicating the right-hand side of the binary operator. For example:

  ```
  >>> db.query().fromTable('employee').union(db.query().fromTable('employee2'))
  ```

  constructs a query plan that combines the Employee and Employee2 tables. **The schemas for both the left- and right-hand side subqueries for the union must have the same fields and types.**

- **`join(planBuilder, joinParameterDict)`**: adds a join operator to the plan. Much like the union operator, the plan builder to which the join method is applied acts as the left-hand input to the join, while the `PlanBuilder` provided as an argument acts as the right-hand input. **Note that we assume the two inputs do not have any overlapping attributes, that is, the left schema attributes must contain all different names than the right schema attributes.**

  The join operator takes several parameters depending on the particular join algorithm requested:
  - **method=string**: defines the join algorithm to use, which must be one of `nested-loops, block-nested-loops, indexed, hash`. Required for all
  - **expr=string**: defines the join predicate. Required for nested and block-nested join types. Optional for index and hash, where it specifies any additional join predicates beyond key equality.
  - **lhsSchema=DBSchema**: the schema of the left-hand input to the join. We include this as a parameter to allow you to rename schema fields.
  - **rhsSchema=DBSchema**: The schema of the right-hand input to the join. You can rename attributes by explicitly specifying this schema as needed.
  - **lhsKeySchema=DBSchema**: Required for indexed/hash joins. For an indexed join, you should perform an index lookup with the attributes specified in this key schema. For hash joins, the attributes in the left key schema are compared with the attributes in the right key schema.
  - **rhsKeySchema=DBSchema**: Required for hash joins. Used in the hash join's equality predicate.
  - **lhsHashFn=string**: Required for hash joins. A string specifying a hash expression. This should evaluate to an Python integer indicating the bucket for a tuple from the left input.
  - **rhsHashFn=string**: Required for hash joins. Similar to the lhsHashFn, selects a bucket for the right tuple. Hash joins use a block nested join to match left and right tuples in the same bucket.

  In our exercises, we ask you to complete a skeleton for a join operator, supporting the modes described above. When running experiments, you will have to pick hash functions appropriately for the data and

query workload.

- **`groupBy(planBuilder, groupParameterDict)`**: adds a group-by operator to the plan. The group-by is made up of two components: i) a list of grouping attributes which define partitions of the input set; ii) a list of aggregation expressions which are applied to the partitions. Takes the following parameters:
    - **groupSchema=DBSchema**: This schema specifies all attributes in the group-by clause.
    - **aggSchema=DBSchema**: This schema specifies the names and types of all aggregation expressions present in the operator.
    - **groupExpr=lambda**: defines the grouping function, that is, by what criteria (e.g. attribute tuple) different groups should be determined. For example, if we want to group employees by id, we would use `lambda e: e.id`. The result of the grouping function should be hashable. In Python, lists are not hashable, while tuples are. To support more complex groupings, we suggest you promote values returned by the group-by function to tuples as needed (i.e., by using the `isinstance` operator).
    - **aggExprs=[(init_value, aggregate-lambda, finalize-lambda)]**: defines the aggregation functionality for each partition. Aggregate-lambda is a is a function applied to each tuple as values are sent to their appropriate bin. This function takes two arguments: i) the accumulated value of the aggregate (e.g., a running sum or a running min), and ii) a tuple from the input. Finalize-lambda is applied to the aggregate value for each partition prior to producing the operator's results. This applied to every partition and its accumulated value. So for example, one could compute a median by accumulating each item in the aggregate-lambda, and then sorting and picking the median element in the finalize lambda.
      For example, the aggregate expressions for a sum and min aggregate on a field x are:

      ```
      sumExpr = [(0, (lambda acc,tup: acc+tup.x), (lambda acc: acc))]
      minExpr = [(sys.maxsize, (lambda acc,tup: min(acc, tup.x), (lambda acc: acc))]
      ```

    - **groupHashFn=lambda**: defines how to break up the grouping key domain into different groups, given the grouping function's output. For example, we might want to separate the id domain from above into 3 partitions, in which case we would use `lambda x: x % 3`. To support aggregations with a large number of groups, the group-by operator should use a partition file for each distinct value of the group hash function.

  Our exercises ask you to complete a group-by operator skeleton based on the above specification of its parameters. When using hash-based group-by aggregates, you will have to pick hash functions appropriately for the data and query workload.

The doctests for the `Query.Plan.PlanBuilder` class include examples of how to construct and run queries in our database.

Operator API: The `Query.Operator.Operator` class serves as the base class for all the operators in our query engine. A query plan consists of a tree of `Operator` objects. Each operator provides the following schema and metadata methods:

- **id**(): returns a unique string identifier for the operator
- **operatorType**(): returns a string describing the operator type
- **inputSchemas**(): returns a list of `DBSchema` objects defining the types and fields of the inputs provided to the operator.Operators such as selects, projects and group-by aggregates have single inputs, while joins and unions have multiple inputs.
- **schema**(): returns the output schema of the operator as a DBSchema object.
- **inputs**(): returns a list of child operator objects in the plan tree.

Query processing occurs through our implementation of an iterator interface for each operator. By supplying `__iter__` and `__next__` methods, we can use Python for loops to access the results of an operator's processing. Each operator serves as the input to the next operator in line. Iterating over an operator causes the operator to carry out its work, which will involve iterating over its child operator(s), which will in turn iterate over grandchild operators etc. Eventually a `TableScan` operator, whose only job is to read the data in the database, will be activated. We have provided two example operators in **Query/Operators/Select.py** and

Query/Operators/Project.py that are 1-to-1 (consume 1 input, produce at most 1 output).

Orthogonal to the iterator interface, operators can perform their processing in one of two modes: pipelined execution, or batch execution. Passing a boolean value as the `pipeline` keyword argument to the `Operator` constructor indicates whether the operator should work on the basis of page-at-a-time, or in set-at-a-time fashion. Pipelined execution requires the availability of a `processInputPage` method to process a single page of data. Batch execution requires a `processAllPages` method to be defined. You will primarily be developing operators that work in batch fashion in our exercises. These methods are invoked from the iterator interface.

We show the implementation of the `Query.Operators.Select.__iter__` method below:

```
1.  def __iter__(self):
2.    self.initializeOutput()
3.    self.inputIterator = iter(self.subPlan)
4.    self.inputFinished = False
5.
6.    if not self.pipelined:
7.      self.outputIterator = self.processAllPages()
8.
9.    return self
```

Here the select operator initializes its output file (more on this below), and then obtains a handle to its child operator's iterator. If the select operator's pipeline parameter is not set, we perform batch operation which computes all operator results when the iterator is constructed, and writes out results to a temporary output file. In this case, the operator's iterator simply scans the output tuples stored in a temporary file. We can see this in the definition of the `__next__` method, where we return `next(self.outputIterator)` (line 13):

```
1.  def __next__(self):
2.    if self.pipelined:
3.      while not(self.inputFinished or self.isOutputPageReady()):
4.        try:
5.          pageId, page = next(self.inputIterator)
6.          self.processInputPage(pageId, page)
7.        except StopIteration:
8.          self.inputFinished = True
9.
10.     return self.outputPage()
11.
12.   else:
13.     return next(self.outputIterator)
```

In the case where we are performing a pipelined selection, the control logic loops processing one page at a time (lines 3-8) until we have either consumed all input from the child operator, or we have fully packed an output page (with `self.isOutputPageReady`).

The final aspect to understand of our `Operator` class is its methods for managing and producing output tuples. The starting point for this is in `Query.Operators.Operator.initializeOutput` method. This method creates a temporary file for the operator's expected output tuples in the storage engine. This is a heap file on which we can invoke tuple methods such as `insertTuple` whenever we have a new output. Also, the method initializes the `Operator.outputPages` field as an empty list. This list tracks pages containing output tuples, that is they are the page objects backed by the temporary file. Above we have seen the `isOutputPageReady` and `outputPage` methods. These methods are defined in terms of the `Operator.outputPages` list, the former simply checking if the last page in the list (i.e., the page we last wrote to) is full, while the latter simply pops an element off the list.

In addition to the `initializeOutput` method, we have provided an `emitOutputTuple` method. You should use this method inside `processInputPage` and `processAllPages` whenever you want to write an output tuple. The `emitOutputTuple` method allocates a new output page on the temporary file as needed (that is if the previous output page is full), adds any new output page to the `outputPages` list, and then finally invokes the output page's `insertTuple` method with its tuple argument.

Exercises

CS 316 students should implement exercises 1-3 and 5. You can receive extra credit for exercises 4 and 6. CS 416 students should implement exercises 1-5. You can receive extra credit for completing exercise 6. Point totals for this homework are 100 points for CS 316, and 125 points for CS 416.

1. Union, and block nested loops join (25 points)

Implement the operators for union and block-nested-loops join. Union is a good place to start adding to the codebase, because it's relatively simple. Pay attention to the fact that union can work either in page-at-a-time or set-at-a-time mode, that is you will need to implement both of its `processInputPage` and `processAllPages` method.

Block-nested-loop join is a more efficient variant of the simple nested-loop join, whereby the outer relation is fully loaded into all available memory in the buffer pool. In order to keep block pages from being evicted from the buffer pool, you should use the page-pinning interface which has been introduced in this version of the codebase. We have provided a skeleton for the join operator in **Query/Operators/Join.py**, and for this exercise you will need to implement the `blockNestedLoops` method and any auxiliary methods as required.

2. Group-by aggregation (25 points)

Implement the group-by aggregation operator. This operator partitions the records into groups and computes an aggregation per group, for all tuples residing in the group. The grouping values are determined by groupExpr, while the number of partitions is determined by groupHashFn. Each tuple should be fed to the groupExpr and groupHashFn, and then added to the resulting partition using the aggregation function.

You should support the possibility of needing to operate on a large number of groups exceeding the size of main memory. Thus for each partition indiciated by the groupHashFn function, you should create a temporary partition file using the storage engine available in the operator. You should pick a suitable groupHashFn when implementing your queries in the experiments section, and you may assume that you only need to hash once to construct partition files (not recursively).

3. Hash join (25 points)

Implement the hash join operator through the `Query.Operators.Join.hashJoin` method. The hash join algorithm involves using a hashing function to partition both relations into small partitions that fit in memory. You should create a temporary partition file for each partition, for both relations. Pairs of in-memory partitions can then be processed with a block-nested loop join to find matching tuples. We have provided the parameters passed in to the hash join in the section on "Creating query plans" above. Please note that while the key schema paramaters define an equality predicate (where fields from `lhsKeySchema` must be equal to the fields from `rhsKeySchema`), the join expression may contain additional predicates (i.e., range predicates, etc) that must be evaluated as part of the matching.

4. External Sort (25 points - CS 600.416 only)

Implement an external sort operator. This operator should process the input in multiple passes, with the first pass create partitions that fit in main memory, along with performing an in-place sort in main-memory per partition. You should fill up available memory with as much as can fit from an input operator (using the **pinning interface which is new in this codebase**), and then save the pages back to disk. Feel free to use python's built-in in-place sort routine. This will create multiple sorted runs as described in our lecture. Later

phases must then merge the multiple runs by iterating simultaneously over 2 runs and producing a merged output in the desired sort order.

5. Experiments (25 points)

We want to test a variety of queries on our database framework and compare both the performance and results to those of Berkeley DB. For our dataset, we will supply the binary files used by our storage engine for the TPC-H dataset. This way you will not need to parse and load the dataset from a CSV format for every run.

Dataset We will make this available on the CS grad/ugrad network, at the following filepath: `~cs416/datasets/tpch-hw2/`.

Plots/figures For each query, graph the performance on Berkeley DB vs the performance using our codebase, for each kind of join operator that you implement. Also, for the third query below, try **2 different operator orders** for the query using our codebase and compare their performance.

1.
```
select p.name, s.name
from part p, supplier s, partsupp ps
where p.partkey = ps.partkey
  and ps.suppkey = s.suppkey
  and ps.availqty = 1
union all
select p.name, s.name
from part p, supplier s, partsupp ps
where p.partkey = ps.partkey
  and ps.suppkey = s.suppkey
  and ps.supplycost < 5;
```

2. **Query 1 from homework 0**

```
select part.name, count(*) as count
from part, lineitem
where part.partkey = lineitem.partkey and lineitem.returnflag = 'R'
group by part.name;
```

3. **Query 4 from homework 0**

```
create table temp as
  select n.name as nation, p.name as part, sum(l.quantity) as num
  from customer c, nation n, orders o, lineitem l, part p
  where c.nationkey = n.nationkey
    and c.custkey = o.custkey
    and o.orderkey = l.orderkey
    and l.partkey = p.partkey
  group by n.name, p.name;

select nation, max(num)
 from temp
 group by nation;
```

6. Indexed nested loop join (25 points Extra credit)

For extra credit, implement an indexed join operator with the `indexedNestedLoops` method in the join skeleton. Indexed-join involves looping over the left-hand side tuple and retrieving the corresponding right-hand side tuple using the index. This requires using the index interface of the storage layer, which is **new in this version of the codebase**.

Summary of grading

Exercise	Points
1: Union, and block nested loops join	25
2: Group-by aggregation	25
3: Hash join	25
4: External sort (416 only)	25
5: Experiments	25
6: Indexed nested loops join (extra credit)	25

Handin

As with HW0 and HW1, send your submission in by email to `cs416@cs.jhu.edu`. The submission must be a zip/tar.gz archive with the same structure as the handout, including both the handout code and your own. Name your submission with the names of your group members, and include a README with your names and email addresses.

IMPORTANT : do not include the supplied data files in your repository, these are large and should not be duplicated for handin.

Relevant lectures

Lecture 7: operator algorithms

Support code and extensions from previous handout

Pinning: Though most of the codebase realizes the same interfaces that were handed out in the previous exercise, there are some differences. One of these differences is pinning support, which is implemented in the BufferPool. Any page in the BufferPool can be pinned and unpinned. A pinned page will not be evicted from memory. This is very useful for algorithms that require a large chunk of memory, or possibly all of memory, to work with. If you're handling a page, you sometimes want to be sure that the page won't be evicted to disk, as that will severely impact your performance. This is what pinning is for.

Don't forget to unpin any pages you pinned once you're done with them, or the system will leak memory.

IndexManager: The second difference between this codebase and the previous one is the provision of indexes in the storage layer. This is done through the `Storage.Index.IndexManager` class. Accessed through the FileManager, this class gives you access to indexes over your data and the ability to use them for both queries and updates. The FileManager's index API consists mainly of

- `createIndex(relId, relSchema, keySchema, primary)`: Create an index to attach to a relation. 'primary' is a boolean flag specifying whether this index should be classified as the relation's primary index or a secondary index. This returns an index identifier, represented as an integer.
- `lookupByIndex(relId, indexId, keyData)`: Use an index to lookup a tuple in a relation.

- `deleteByIndex(relId, indexId, keyData)`: Use an index to delete a tuple.
- `updateByIndex(relId, indexId, keyData, tupleData)`: Use an index to update a tuple.

This interface is needed in order to implement the indexed nested-loop join. Our indexes also support scan operations (see `scanByIndex`) and wrapper methods to support the above index operators directly on the primary index of a relation (see the `{lookup|delete|update}ByKey` operations in the `FileManager`). These indexes are implemented as BerkeleyDB databases which store `TupleId` objects referencing the records kept in our heap files in the storage engine. This way, our indexes are unclustered indexes which perform random IO operations at the leaf layer.