Database Systems

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

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

Assignment 1: Storage Engine

Due date: 11:59pm Wednesday February 18th, 2015

Description | Logistics | Exercises | Relevant lectures

Description

A database storage engine provides an abstraction layer for the memory-disk storage boundary. With this abstraction, database queries can use iterators to traverse all of the records in a relation, and subsequently process these records. The storage engine provides efficient block-based access to the disk, as well as caching of blocks during iteration over a relation's tuples.

In this assignment, you will design and implement three components of a database storage engine, namely a contiguous and slotted page layout scheme, a heap file external data structure to store relations, and a buffer pool to cache pages as they are accessed from disk-resident heap files. Together, these components will let you load and store relational data in a binary format, access them in page-oriented fashion, and provide basic caching of "hot" pages in main-memory. The storage engine that you implement here will hold the datasets used throughout the course, and acts as an input to the query processing layer that we will see in the second exercise.

Specifically, the exercises making up this homework include:

- Implementing two page layout schemes to store multiple records.

- Implementing a heap file data structure to hold all of the pages (and records) comprising a relation.
- Implementing a buffer pool, as a cache for pages brought into memory from heap files.
- Conducting experiments on the storage performance of your components.

We will use the lab sessions on February 10th and 17th to discuss the design of the codebase, and the concepts behind our page and file implementations. **You are required to work in pairs on this exercise, and we recommend you get started early on this assignment (we estimate you will be writing approximately 1500-2000 lines of code).** We also 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 completed by the end of the first week.

Logistics

Support code The Python support code for this assignment is available **`~cs416/handouts/hw1/dbsys-hw1.tar.gz`** This includes utilities to define a database schema and records, and skeleton classes for your storage engine page, file and buffer pool implementations. Once you have copied and extracted the codebase, you should see the following directory and file structure:

```
dbsys-hw1/              ## Source code directory
|-- Catalog             ## Catalog utilities (i.e., schemas and types)
|---- Identifiers.py    ## Identifier types, to distinguish tuples, pages and files.
|---- Schema.py         ## Schema implementation
|
|-- Storage             ## DB storage layer implementation
|---- BufferPool.py     ## Buffer pool skeleton
|---- File.py           ## Heap file components
|---- FileManager.py    ## File manager, tracks which files implement which relations.
|---- Page.py           ## Contiguous page
|---- SlottedPage.py    ## Slotted page
|---- StorageEngine.py ## Storage engine API for other DBMS components
```

We use qualified names throughout this document to refer to Python classes, methods and variables. For example the name `Catalog.Schema.Types` refers to the `Types` class in the **Catalog/Schema.py** file, and similarly `Catalog.Schema.DBSchema` refers to the `DBSchema` class in the **Catalog/Schema.py** file.

You should not need to modify the Storage/FileManager.py or Storage/StorageEngine.py files in this assignment.

Getting started To get started with our codebase, we suggest getting familiar with our utilities for defining schemas, and creating tuples through the `Catalog.Schema.DBSchema` class. This class implements a database schema, storing the attributes and types that are typically defined in a SQL `CREATE TABLE` statement.

We have provided a few Python doctests for this class which highlights its usage. As their name suggests, doctests provide documentation-based testing for Python code, where we can include snippets of Python code in our documentation to describe how to use a class and its methods. Python can identify such code written in documentation and execute it to ensure it performs as expected. See the Python doctest documentation for more information: http://docs.python.org/3.3/library/doctest.html

Here is how we can run the doctests from the `dbsys-hw1` directory:

```
~/dbsys-hw1$ python3 Catalog/Schema.py -v
Trying:
    schema = DBSchema('employee', [('id', 'int'), ('dob', 'char(10)'), ('salary', 'int')])
Expecting nothing
ok
... [lots of testing output] ...
24 tests in 15 items.
24 passed and 0 failed.
Test passed.
```

The docstring for the `Catalog.Schema.DBSchema` class illustrates how we can create a schema, create tuples matching that schema, and pack and unpack these tuples as Python byte arrays. Here is the example in our docstring.

```
~/dbsys-hw1$ python3
Python 3.3.3
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>>
>>> from Catalog.Schema import DBSchema

# Create a schema object for an 'employee' relation, with three fields.
>>> schema = DBSchema('employee', [('id', 'int'), ('dob', 'char(10)'), ('salary', 'int')])

# Create an employee tuple, with the given values.
>>> e1 = schema.instantiate(1, '1990-01-01', 100000)
>>> e1
employee(id=1, dob='1990-01-01', salary=100000)

# We can serialize/deserialize the created instances with the 'pack' and 'unpack' methods.

# Note the examples below escape the backslash character to ensure doctests run correctly.
# These escapes should be removed when copy-pasting into the Python REPL.

>>> schema.pack(e1)
b'\\x01\\x00\\x00\\x001990-01-01\\x00\\x00\\xa0\\x86\\x01\\x00'

>>> schema.unpack(b'\\x01\\x00\\x00\\x001990-01-01\\x00\\x00\\xa0\\x86\\x01\\x00')
employee(id=1, dob='1990-01-01', salary=100000)

>>> e2 = schema.unpack(schema.pack(e1))
>>> e2 == e1
True
```

When we created the employee schema above, we provided a type for each field. The set of types available in our codebase can be found in `Catalog.Schema.Types`. This includes:

```
'byte'      : A byte, represented as a 'B' format string in Python's struct
'short'     : Short type, represented as a Python struct 'h'
'int'       : Integer type, represented as a Python struct 'i'
'float'     : Float type, represented as a Python struct f'
'double'    : Double type, represented as a Python struct 'd'
'char(n)'   : We only support fixed length strings, thus char types must include
              a length n, for example 'char(44)' for a string of length 44-bytes.
'text(n)'   : A synonym for 'char'
```

You should experiment with creating your own schemas and tuples, and packing and unpacking from their corresponding binary representation.

Next steps: After understanding the `DBSchema` class, we suggest you look through the doctests for the **Catalog.Identifers.py** file, particularly the `FileId, PageId, TupleId` classes. These classes are used to uniquely identify and reference the corresponding objects in the storage layer, and can act as keys in any data structure you create.

You will then be ready to start the exercises, starting with the `Page` and `SlottedPage` classes in Exercise 1. All of our skeleton classes include a few doctests to provide you a description of the anticipated usage of your implementation.

Also, please read the Support Code section at the end of this document, especially the part on deleting the data/ directory between runs.

Dataset We will continue to use the TPC-H dataset that you previously saw for this assignment. This time, we have included a larger (scale factor 1) dataset, which totals to 1GB of data. We have made this available on the CS grad/ugrad network, at the following filepath: `~cs416/datasets/tpch-sf1/`

Experiments CS 416 only This homework includes an experimental evaluation section in its exercises. You should aim to create reproducible experiments and visualizations as part of your handin via a combination of Python and shell scripts, and a README document.

To help with data visualization of experimental results, we recommend using the matplotlib package for plotting and analysis of experimental results. Matplotlib is already installed on the CS ugrad network. If you have not used matplotlib before, you should read through following tutorial as a starting point for its API: http://matplotlib.org/users/pyplot_tutorial.html. If you're connecting remotely to the ugrad network, you should run matplotlib in headless mode. The following Python code will do this (see here for more information: link). You should also save your plots with the `savefig()` function instead of show.

```
Python 2.7.5 (default, Nov  3 2014, 14:26:24)
>>> import matplotlib
>>> matplotlib.use('Agg')
>>> import matplotlib.pyplot as plt
>>> plt.plot([1,2,3])
>>> plt.savefig('myfig')
```

Exercises

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

1. Contiguous and Slotted Pages (40 points)

Pages are byte arrays that provide the storage for the binary representation of tuples created with a `DBSchema` object. We ask you to implement both a contiguous page, and a slotted page. This will require implementation of the following classes:

- `Storage.Page.Page`
- `Storage.Page.PageHeader`
- `Storage.SlottedPage.SlottedPage`
- `Storage.SlottedPage.SlottedPageHeader`

The contiguous page and its header should be implemented in the **Storage/Page.py** file, and the slotted page in the **Storage/SlottedPage.py** file. Both types of page are implemented as a fixed size, in-memory, mutable byte array through Python's BytesIO objects. By inheriting from `BytesIO`, we can modify the page at arbitrary offsets, and at arbitrary subranges with Python's slice operations, for example to insert or access packed tuples. In particular, we can use a BytesIO's `getbuffer()` method to obtain a memoryview object, through which we can perform simple byte-level operations. See the the example using the `getbuffer()` method at: http://docs.python.org/3/library/io.html#io.BytesIO

Design issue Both the contiguous page and the slotted page should have access to a BytesIO object. You should think about the code organization to achieve this, for example, you will have to decide whether:

- you should use an object-oriented approach, where your `Page` and `SlottedPage` classes inherit from `BytesIO`.
- your `Page` and `SlottedPage` classes could alternatively keep an instance of `BytesIO`.
- your `Page` could inherit from `BytesIO`, and your `SlottedPage` class could inherit from the `Page` class.

In this assignment, you only need to consider the case of tuples with a **fixed length** schema, and not a variable length schema.

Both types of pages keep metadata in the page header to track the readable and writable regions of a page. The contiguous page keeps a single offset (the free space offset) recording the first free writeable byte available in the page. The contiguous page inserts tuples at this offset, and any tuples that are removed must shift any subsequent tuples to preserve data contiguity. The slotted page keeps a set of slot data structures whose contents indicate whether a tuple exists at a given offset within the page. In this way, tuples can be inserted into a page at any offset whose slot indicates that the offset is unused. Insertions must however find an empty slot if one is available. Also, a tuple may be deleted by resetting its slot rather than shifting any data as performed by the contiguous page.

Page classes (i.e., both `Page` and `SlottedPage`) should support the following API. Below, `tupleId` corresponds to `Catalog.Identifiers.TupleId` objects. For contiguous pages, the `TupleId.tupleIndex` field describes the offset of a tuple in the page, while for a slotted page it defines the slot index. Also, `tupleData` corresponds to Python byte strings.

- `getTuple(self, tupleId)` - Given the index of a tuple in the page, return the tuple at that index. The return value should be a Python byte string which can later be unpacked with a DBSchema object.
- `putTuple(self, tupleId, tupleData)` - Given a packed tuple represented as a byte string, insert it into the page at the given tuple index.
- `insertTuple(self, tupleData)` - Insert the given tuple into the page. This should return a TupleId defining the index at which the tuple was inserted.
- `clearTuple(self, tupleId)` - Clear the contents of the given tuple.
- `removeTuple(self, tupleId)` - Delete the tuple at the specified index.

We suggest to keep the following fields in your page classes, but you are free to organize the class as you wish:

- `pageId` : a `Catalog.Identifiers.PageId` object that identifies the page.
- `header` : a `Storage.Page.PageHeader` object maintaining metadata for the page.

The page class methods above should use the page header to provide metadata operations. We suggest the following API for header classes (i.e., both `PageHeader` and `SlottedPageHeader`). You must determine what fields to keep in the header classes, but this must include at least a byte for a page's status flags (e.g., dirty bit), as well as the tuple size and page capacity.

- `hasFreeTuple(self)` : Returns whether a page has space for a tuple.
- `nextFreeTuple(self)` : Returns an index in the page for writing the next tuple.
- `nextTupleRange(self)` : A wrapper for `nextFreeTuple`, returning a pair of start and end offsets for writing a tuple.
- `isDirty(self)` : Returns whether the page is dirty. Every modification to the page (e.g., insertion/removal of a tuple) should mark the page as dirty.
- `setDirty(self, dirty)` : Sets the dirty bit on the page.
- `numTuples(self)` : Returns the number of tuples currently in the page.
- `freeSpace(self)` : Returns the free space available in the page.
- `usedSpace(self)` : Returns the page's used space.
- `pack(self)` : creates a binary representation of the page header.
- `unpack(cls, buffer)` : creates a header object by unpacking the binary representation in the given byte string.

We suggest to keep the following fields in your page classes, but you are free to organize the class as you wish:

- For the contiguous page header (i.e., `Storage.Page.PageHeader`)
  - `flags` : a single byte representing the page status (i.e., dirty bit)
  - `tupleSize` : the size of the tuples stored in the page.
  - `pageCapacity` : the size of the page associated with the header.
  - `freeSpaceOffset` : the position of the first writeable byte in the page.

- Our slotted page header (i.e., `Storage.SlottedPage.SlottedPageHeader`), inherits from the contiguous page header, so it has all of the above fields. You will additionally need to maintain a data structure for the slots themselves. We leave it to you to design this slot data structure.

Design issue: For both types of pages, you should consider how to pack the corresponding page header into the byte array backing the page. We suggest you store the page header at the beginning of the page's byte array, for example invoking a page header's `pack()` function from within the page's `pack()` function.

Design issue: We have provided a Python iterator for the `Page` class using the `getTuple` function that allows us to loop over all the tuples in a page. You will need to implement an iterator for the `SlottedPage` class depending on your choice of data structure to represent slots.

You may extend the API in any way you like, but you should provide at least the functions listed above.

2. Heap Files (35 points)

A heap file is used to store a database relation as an unsorted set of pages. Large relations can be broken into many heap files, for example, legacy file systems often had a 2GB maximum file size limit. Your storage engine should implement heap files, bearing in mind that tuples may be arbitrarily inserted or deleted from relations, and thus from both the pages and heap files in which they reside. These heap files will be used by a file manager, which keeps a mapping of relations and the files that store their content. For simplicity, you may assume that a relation is stored in a single file.
You should implement the following class in **Storage/File.py**:

- `Storage.File.StorageFile`

Much like a page header resides at the beginning of a page and maintains metadata for the page, a file header has an analagous relationship for storage files. That is, a file header also resides at the beginning of the file stored on disk, and below we briefly describe what metadata it contains.

Your heap file should implement the following functionality:

- methods to read and write pages held in memory, with the following signatures:
  - `readPage(self, pageId, page)` : Read the page at a specific index in the file into the given page object (which may be a Page or SlottedPage).
  - `writePage(self, page)` : Write the page object to the file. The page object will include a page id indicating the write location.

- methods to manage the allocation and creation of pages in the file.
  - `allocatePage(self)` : adds a new page to the end of the file. Python file objects enable this by seeking to the end of the file, and performing a write operation there.
  - `availablePage(self)` : returns the page id of the first page with available space. You should determine any data structure that the storage file must maintain to support this operation efficiently.

- methods to operate on page headers alone, without needing to access the full page. This will be useful for inserting data into a relation, where we can read page headers alone to find a page with sufficient free space in a file.
  - `readPageHeader(self, pageId)` : read and return the header of the page with the given page id.
  - `writePageHeader(self, page)` : writes a page header to disk. The page must already exist, that is we cannot extend the file with only a page header.

- tuple operations on the file.
  - `insertTuple(self, tupleData)` : inserts the given tuple to the first available page. This must determine the page to which we should insert the tuple.
  - `deleteTuple(self, tupleId)` : removes the tuple by its id, tracking if the page is now free.

    - `updateTuple(self, tupleId, tupleData)` : updates the tuple by id.

We suggest you keep the following fields in your `StorageFile` class:

- `bufferPool` : a buffer pool object.
- `fileId` : a `Catalog.Identifiers.FileId` object.
- `path` : a string denoting the path on disk.
- `header` : a `Storage.File.FileHeader` object keeping metadata for this file.
- `file` : a Python file object for the above path.
- `freePages` : a data structure, of your choosing, to track pages with free space.

We have implemented a `Storage.File.FileHeader` class for you, containing the following fields:

- `schema` : a `Catalog.Schema.DBSchema` object describing the records stored in the file associated with this file header.
- `pageSize` : the page size for the file.
- `pageClass` : the Python class implementing the pages stored in the file. By explicitly storing this in the header, we can use the same StorageFile implementation whether we want a file of contiguous pages, or a file of slotted pages.

Design note: our `Storage.File.FileHeader` class maintains `schema, pageSize, pageClass` instances for consistent reloading of files when in the `update` mode. This way a heap file is self-contained and all information need to operate on pages and tuples is present in the `FileHeader`.

Design issue: In a similar fashion to the page header and page type, you will need to store the file header at the beginning of the storage file. You should read over and determine how to use the `FileHeader.toFile()` and `FileHeader.fromFile` methods to read and write the file header as part of your solution.

We have provided Python iterators to access all of the pages and page headers in a file, as well as the tuples in a file, based on the above methods. You should not need to modify these.

3. Buffer Pool (25 points)

A buffer pool combines the functionality of a memory allocator and a cache for a database. In this exercise, we will ask you to implement a buffer pool that reserves a fixed amount of memory on database engine initialization. These pages should initially be considered as free pages, and are available for use in caching pages accessed from disk. As page access requests are submitted to the buffer pool, the buffer pool should check if the page already resides in memory and return it if available. Otherwise, the buffer pool should forward the access request to the appropriate heap file, supplying a free in-memory page that the heap file can fill in with data.
To act as a cache, the buffer pool will also need to evict pages as needed from the cache when pages are accessed. We leave the choice of eviction policy to you, but you should mention your choice when reporting numbers from your experiments.

For your buffer pool, you should implement the`Storage.BufferPool` class. To keep things simple in this exercise, we do not require you to handle concurrent access to the buffer pool. Your buffer pool should implement the following API:

- `hasPage(self, pageId)` : Returns whether the a page with the given page id is present in the buffer pool.
- `getPage(self, pageId)` : Read a page from disk, into the buffer pool.
- `discardPage(self, pageId)` : Removes a page from the page map, returning it to the free page list without flushing the page to the disk.
- `flushPage(self, pageId)` : Write a page back out to disk.
- `evictPage(self)` : Evict from the buffer pool using a LRU policy.

4. Experiments (25 points)

The last exercise is to evaluate the performance of your storage engine, and create an IPython Notebook that visualizes these experimental results. The focus of this exercise is to be able to produce repeatable, reproducible experimental evaluations of your performance-sensitive implementations, and be able to visualize the results of these evaluations in an easy-to-understand format. We will be studying two metrics: i) I/O performance in terms of the end-to-end throughput of your storage engine; ii) the space efficiency of the varying page types.

We have provided a workload generator in our handout. This workload generator takes as input four parameters:

- a file path specifying the location of the TPC-H dataset
- an input dataset scale factor, from 0.0 to 1.0
- a page size
- a workload mode identifier, from 1 to 4

For the experiments, we ask you to execute a varied mix of tuple insertions and deletions for the TPC-H dataset in your storage engine, and to plot two graphs, one each for these two metrics:

- measure the time taken in seconds to run a workload.
- measure the total size of all storage files in kilobytes after running each workload.

Each plot should contain a line for all combinations of workload mode, and for two page sizes (4Kb and 32Kb), yielding a total of 8 lines per graph. Each line should vary the input scale factor from 0.1 to 1.0 in increments of 0.1. While you are free to implement and test your experiments on your own machines, we ask that you ensure that the experiments run correctly on the ugrad cluster, and that your visualizations use the numbers obtained by running the final experiments on that cluster. This allows for a reasonable comparison between implementations, without having to account for varying hardware and operating system configurations.

Workload We supply a workload generator that will issue tuple read operations to your storage engine in a variety of access patterns and occurrence ratios. Our workload generator can be found in the `Utils.WorkloadGenerator.WorkloadGenerator` class. This provides the following methods:

- `runWorkload(self, datadir, scaleFactor, pageSize, workloadMode)`

This will create the TPC-H schema in your storage engine, prepopulate it with the dataset described at the beginning of the handout, and then issue further data modifications in one of four modes:

- Sequential reads
- Mostly sequential reads (80%)
- 50/50 sequential vs. random operations
- Highly random operations (80%)

We'll post an anonymized table for all students' results during the assignment so that you can see the kinds of performance that your classmates are achieving. This should help you to determine whether you are in the right ballpark for performance.

5. CS 600.416 Extensions (additional 25 points)

The upper-level exercises for this homework are to implement one additional type of page, namely a PAX page. PAX pages are a hybrid of the row-oriented slotted page schema as well as the columnar approach of the decomposition storage model (DSM). PAX partitions a single page into segments for each attribute present in the tuple, keeping a slot map for each segment. A PAX page header keeps track of the start of each

attribute segment. Since the advantage of PAX pages occurs when pulling in a subset of attributes, your PAX pages should also implement the following API:

- `getTupleField(self, tupleId, fieldPosition)` : retrieves only the field specified by the given field index for a tuple.
- an iterator to access a single column of data using the above `getTupleField` method.

You should implement your solution as a `Storage.PaxPage` class.
For experiments, run the 4 workloads as described above for the PAX pages. How do your results compare to the slotted page?

Summary of grading

Exercise	Points
1: contiguous and slotted page	40
2: heap file	35
3: buffer pool	25
4: experiments	25
5: PAX pages	25

Handin

As with HW0, you can send in your submission by email to `cs416@cs.jhu.edu`. Your submission must be a zip or tar.gz file of the same structure as the handout, including the handout code and your own. Name your submission `{group-members}.tar.gz`, and include a README file containing your group members' 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 3: page and file layout slides
Lecture 4: buffer pool slides

Support code

While you should not need to edit the **Storage/FileManager.py** and **Storage/StorageEngine.py** files, we briefly describe their contents and functionality here. The `Storage.FileManager.FileManager` class maintains two data structures:

- `relationFiles`: a mapping of logical relations created in the database to the file identifiers, corresponding to the storage file that holds the data for the relation.
- `fileMap`: a mapping of file identifiers to storage file instances.

It also provides the following methods to manipulate the relations present in the storage layer. In the following a `relId` is a string defining a relation name (e.g., "employee"):

- `relations(self)`: return the relation ids present in the file manager.
- `hasRelation(self, relId)`: checks if the relation is present in the file manager.
- `createRelation(self, relId, schema)`: creates the relation and its backing file with the given schema
- `addRelation(self, relId, fileId, storageFile)`: adds the relation with the given backing file to the file manager.
- `removeRelation(self, relId)`: removes the relation from the file manager, deleting the backing file.
- `detachRelation(self, relId)`: removes the relation from the file manager, without closing or deleting the backing file.

The `Storage.StorageEngine.StorageEngine` class contains instances of the `Storage.BufferPool.BufferPool` and `Storage.FileManager.FileManager` classes. This class is essentially a thin wrapper of the `FileManager`'s functionality, while also acting as the owner of the buffer pool for all other storage layer components. Thus all other database components can access a singleton buffer pool instance from the `StorageEngine`.

Using the storage engine doctests: When running the doctests for either the `Storage.FileManager.FileManager` or `Storage.StorageEngine.StorageEngine` classes, you will notice that we create a **`data/`** directory in your current working path. This contains the storage files created during the doctests. **You should delete this directory prior to any time you re-run the doctests**, to prevent the doctests from opening these files in the storage engine as existing relations.