Assignment 2

Multi-attribute Linear Hashed Files

Last updated: **Thursday 31st March 8:40am**
Most recent changes are shown in red ... older changes are shown in brown.

Aims

This assignment aims to give you an understanding of

- how database files are structured and accessed
- how multi-attribute hashing is implemented
- how linear hashing is implemented

The goal is to build a simple implementation of a linear-hashed file structure that uses multi-attribute hashing.

Summary

Deadline:	9pm on **Friday 15 April**
Late Penalty:	0.05 *marks* off the ceiling mark for each hour late
Marks:	Contributes **20 marks** toward your total mark for this course.
Submission:	From Webcms3: Assignments > Assignment 2 > Submission > upload `ass2.zip` From Linux: `give cs9315 ass2 ass2.zip`

The `ass2.zip` file must contain your `Makefile` plus all of your `*.c` and `*.h` files.
Details on how to build the `ass2.zip` file are given below.

Make sure that you read this assignment specification *carefully and completely* before starting work on the assignment.
Questions which indicate that you haven't done this will simply get the response "Please read the spec".

Note: this assignment does not require you to do anything with PostgreSQL.

Introduction

Linear hashed files and multi-attribute hashing are two techniques that can be used together to produce hashed files that grow as needed and which allow all attributes to contribute to the hash value of each tuple. See the course notes and lecture slides for further details on linear hashed files and multi-attribute hashing.

In our context, multi-attribute linear-hashed (MALH) files are file structures that represent one relational table, and can be manipulated by three commands:

A `create` command
Creates MALH files by accepting four command line arguments:

- the name of the relation
- the number of attributes
- the initial number of data pages (rounded up to nearest 2^n)
- the multi-attribute hashing choice vector

This gives you storage for one relation/table, and is analogous to making an SQL data definition like:

```
create table R ( a1 text, a2 text, ... an text );
```

Note that, internally, attributes are indexed 0..*n*-1 rather than 1..*n*.

The following example of using `create` makes a table called `abc` with 4 attributes and 8 initial data pages:

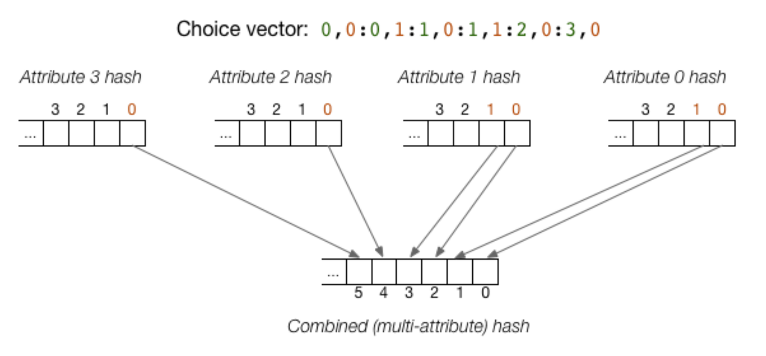
```
$ ./create  abc  4  6  "0,0:0,1:1,0:1,1:2,0:3,0"
```

Note that 6 will be rounded up to the nearest 2^n (i.e. to 8). If we'd written 8, we would have gotten the same result.

The choice vector (fourth argument above) indicates that

- bit 0 from attribute 0 produces bit 0 of the MA hash value
- bit 1 from attribute 0 produces bit 1 of the MA hash value
- bit 0 from attribute 1 produces bit 2 of the MA hash value
- bit 1 from attribute 1 produces bit 3 of the MA hash value
- bit 0 from attribute 2 produces bit 4 of the MA hash value
- bit 0 from attribute 3 produces bit 5 of the MA hash value

The following diagram illustrates this scenario:



The above choice vector only specifies 6 bits of the combined hash, but combined hashes contain 32 bits. The remaining 26 entries in the choice vector are automatically generated by cycling through the attributes and taking bits from the *high-order* hash bits from each of those attributes.

An `insert` command
Reads tuples, one per line, from standard input and inserts them into the relation specified on the command line. Tuples all take the form $val_1, val_2, ..., val_n$. The values can be any sequence of characters except `','` and `'?'`.

The bucket where the tuple is placed is determined by the appropriate number of bits of the combined hash value. If the relation has 2^d data pages, then *d* bits are used. If the specified data page is full, then the tuple is inserted into an overflow page of that data page.

A `select` command

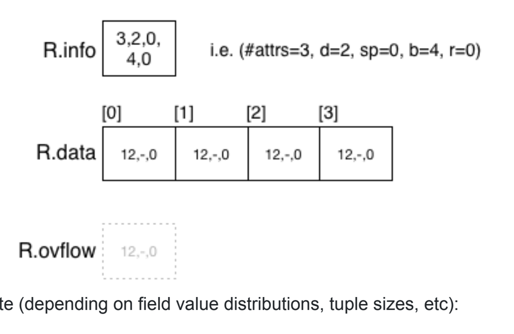
Takes a "query tuple" on the command line, and finds all tuples in either the data pages or overflow pages that match the query. Queries take the form $val_1, val_2, ..., val_n$, where some of the val_i can be `'?'` (without the quotes). Such "attributes" represent wild-cards and can match any value in the corresponding attribute position. Some example query tuples, and their interpretation are given below. You can find more examples in the lecture slides and course notes.

```
?,?,?     # matches any tuple in the relation
10,?,?    # matches any tuple with 10 as the value of attribute 0
?,abc,?   # matches any tuple with abc as the value of attribute 1
10,abc,? # matches any tuple with 10 and abc as the values of attributes 0 and 1
```

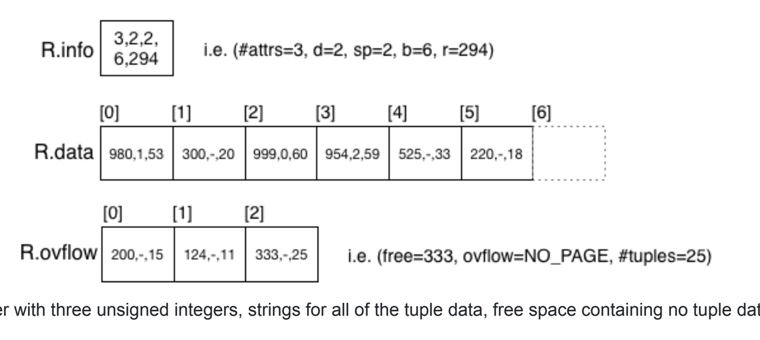
A MALH relation `R` is represented by three physical files:

- `R.info` containing global information such as
  - a count of the number of attributes
  - the depth of main data file (*d* for linear hashing)
  - the page index of the split pointer (*sp* for linear hashing)
  - a count of the number of main data pages
  - the total number of tuples (in both data and overflow pages)
  - the choice vector (*cv* for multi-attribute hashing)
- `R.data` containing data pages, where each data page contains
  - offset of start of free space
  - overflow page index (or `NO_PAGE` if none)
  - a count of the number of tuples in that page
  - the tuples (as comma-separated C strings)
- `R.ovflow` containing overflow pages, which have the same structure as data pages

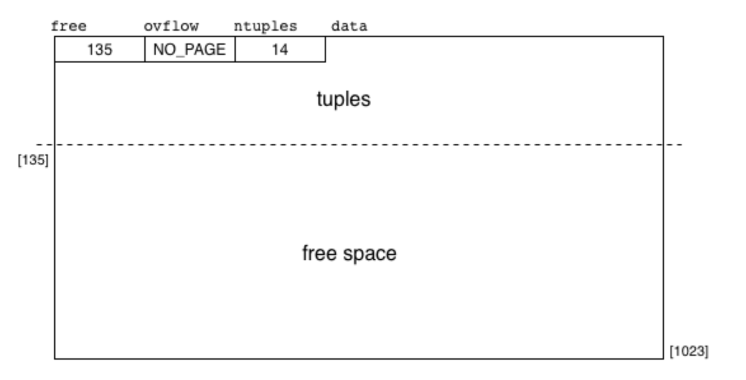
When a MALH relation is first created, it is set to contain a 2^n pages, with depth *d=n* and split pointer *sp=0*. The overflow file is initially empty. The following diagram shows an MALH file `R` with initial state with *n=2*.



After 294 tuples have been inserted, the file might have the following state (depending on field value distributions, tuple sizes, etc):



Pages in MALH files have the following structure: a header with three unsigned integers, strings for all of the tuple data, free space containing no tuple data. The following diagram gives an exmple of this:



We have developed some infrastructure for you to use in implementing multi-attribute linear-hashed (MALH) files. You may use this infrastructure or replace parts of it (or all of it) with your own, but your MALH files implementation must conform to the conventions used in our code. In particular, you should preserve the interfaces to the supplied modules (e.g. `Reln`, `Page`, `Query`, `Tuple`) and ensure that your submitted ADTs work with the supplied code in the `create`, `insert` and `select` commands.

Setting Up

You should make a working directory for this assignment and put the supplied code there. Read the supplied code to make sure that you understand all of the data types and operations used in the system.

```
$ mkdir Your/ass2/Directory
$ cd Your/ass2/Directory
$ unzip /web/cs9315/22T1/assignments/ass2/ass2.zip
```

You should see the following files in the directory:

- `create.c` ... a main program that creates a new MALH relation
- `dump.c` ... a main program that lists all tuples in an MALH relation
- `insert.c` ... a main program that reads tuples and insert them
- `select.c` ... a main program that finds tuples matching a PMR query
- `stats.c` ... a main program that prints info about an MAH relation
- `gendata.c` ... a main program to generate random tuples
- `bits.h`, `bits.c` ... an ADT for bit-strings
- `chvec.h`, `chvec.c` ... an ADT for choice vectors
- `hash.h`, `hash.c` ... the PostgreSQL hash function
- `page.h`, `page.c` ... an ADT for data/overflow pages
- `query.h`, `query.c` ... an ADT for query scanners (incomplete)
- `reln.h`, `reln.c` ... an ADT for relations (partly complete)
- `tuple.h`, `tuple.c` ... an ADT for tuples (partly complete)
- `util.h`, `util.c` ... utility functions

This gives you a partial implementation of MALH files; you need to complete the code so that it provides the functionality described below.

The supplied code actually produces executables that work somewhat, but are missing a working query scanner implementation (from `query.c`), a proper MA hash function (from `tuple.c`), and splitting and data file increase (from `reln.c`). Effectively, they give a static hash file structure with overflows.

To build the executables from the supplied code, do the following:

```
$ make
gcc -Wall -Werror -g -std=c99   -c -o create.o create.c
gcc -Wall -Werror -g -std=c99   -c -o query.o query.c
gcc -Wall -Werror -g -std=c99   -c -o page.o page.c
gcc -Wall -Werror -g -std=c99   -c -o reln.o reln.c
gcc -Wall -Werror -g -std=c99   -c -o tuple.o tuple.c
gcc -Wall -Werror -g -std=c99   -c -o util.o util.c
gcc -Wall -Werror -g -std=c99   -c -o chvec.o chvec.c
gcc -Wall -Werror -g -std=c99   -c -o hash.o hash.c
gcc -Wall -Werror -g -std=c99   -c -o bits.o bits.c
gcc   create.o query.o page.o reln.o tuple.o util.o chvec.o hash.o bits.o   -o create
gcc -Wall -Werror -g -std=c99   -c -o dump.o dump.c
gcc   dump.o query.o page.o reln.o tuple.o util.o chvec.o hash.o bits.o   -o dump
gcc -Wall -Werror -g -std=c99   -c -o insert.o insert.c
gcc   insert.o query.o page.o reln.o tuple.o util.o chvec.o hash.o bits.o   -o insert
gcc -Wall -Werror -g -std=c99   -c -o select.o select.c
gcc   select.o query.o page.o reln.o tuple.o util.o chvec.o hash.o bits.o   -o select
gcc -Wall -Werror -g -std=c99   -c -o stats.o stats.c
gcc   stats.o query.o page.o reln.o tuple.o util.o chvec.o hash.o bits.o   -o stats
gcc -Wall -Werror -g -std=c99   -c -o gendata.o gendata.c
gcc   gendata.o query.o page.o reln.o tuple.o util.o chvec.o hash.o bits.o   -o gendata
```

This should not produce any errors on the CSE servers; let me know ASAP if this is not the case.

Once you have the executables, you could build a sample database as follows:

```
$ ./create  R  3  4  "0,0:0,1:0,2:1,0:1,1:2,0"
cv[0] is (0,0)
cv[1] is (0,1)
cv[2] is (0,2)
cv[3] is (1,0)
cv[4] is (1,1)
cv[5] is (2,0)
cv[6] is (0,31)
cv[7] is (1,31)
cv[8] is (2,31)
...
cv[30] is (0,23)
cv[31] is (1,23)
```

This command creates a new table called `R` with 3 attributes. It will be stored in files called `R.info`, `R.data` and `R.ovflow`. The data file initially has 4 pages (so depth *d*=2). The overflow file is initially empty. The lower-order 6 bits of the choice vector are given on the command line; the remaining bits are auto-generated. Given the file size (4 pages), only two of the hash bits are actually needed.

You could check the status of the files for table `R` via the `stats` command:

```
$ ./stats  R
Global Info:
#attrs:3  #pages:4  #tuples:0  d:2  sp:0
Choice vector
0,0:0,1:0,2:1,0:1,1:2,0:0,31:1,31:2,31:0,30:1,30:2,30:0,29:1,29:2,29:0,28:1,28:2,28:
0,27:1,27:2,27:0,26:1,26:2,26:0,25:1,25:2,25:0,24:1,24:2,24:0,23:1,23
Bucket Info:
#   Info on pages in bucket
    (pageID,#tuples,freebytes,ovflow)
0   (d0,0,1012,-1)
1   (d1,0,1012,-1)
2   (d2,0,1012,-1)
3   (d3,0,1012,-1)
```

Since the file is size 2^d, the split pointer sp = 0. The rest of the global information should be self explanatory, as should the choice vector. The bucket info shows a quadruple for each page; since there are no overflow pages (yet), only data pages appear. The pageID value in each quad consists of the character `'d'` (indicating a data file), plus the page index. Each page is 1024 bytes long, which includes a small header, plus 1012 bytes of free space for tuples. There are currently zero tuples in any of the pages. The overflow page IDs are all -1 (for `NO_PAGE`) to indicate that no data page has an overflow page.

You can insert data into the table using the `insert` command This command reads tuple from its standard input and inserts them into the named table. For example, the command below inserts a single tuple into the `R` MALH files:

```
$ ./insert R
100,abc,xyz
hash(100) = 00011100 00101000 10100111 11101100
Ctl-D
```

The `insert` command prints the hash value for the tuple (based on just the first attribute), and then inserts it into the file. Since the table is currently empty, this tuple will be inserted into page 0. Why page 0? You should be able to answer this by knowing the depth and the hash value. If you then check with the `stats` command you will see that there is a single tuple in the files, and it's in page 0.

Typing many individual tuples is tedious, so we have provided a command, `gendata`, which can generate tuples appropriate for a given table. It takes four comand line arguments, only two of which are compulsory: the number of tuples to generate, and the number of attributes in each tuple. a sample usage:

```
$ $ ./gendata  5  3
1,sandwich,pocket
2,circus,spectrum
3,snail,adult
4,crystal,fungus
5,bowl,surveyor
```

This generates five tuples, each with three attributes. The first attribute is a unique ID value; the other attributes are random words. You can modify the starting ID value and the seed for the random number generator from the command line.

You could use `gendata` to generate large numbers of tuples, and insert them as follows:

```
$ ./gendata 250 3 101 | ./insert R
hash(101) = 11110100 01100100 11010000 00110000
hash(102) = 00100101 10100110 10100001 11100100
hash(103) = 10110011 11001111 10100111 00001000
hash(104) = 00001100 11100000 10000011 11000000
...
hash(348) = 11110000 01011110 01000010 00101001
hash(349) = 01101101 01100101 00011111 10100111
hash(350) = 10011011 01100101 01111001 11001000
```

This will insert 250 tuples into the table, with ID values starting at 101. You can check the final state of the database using the `stats` command. It should look something like:

```
$ ./stats R
Global Info:
#attrs:3  #pages:4  #tuples:251  d:2  sp:0
Choice vector
0,0:0,1:0,2:1,0:1,1:2,0:0,31:1,31:2,31:0,30:1,30:2,30:0,29:1,29:2,29:0,28:1,28:2,28:
0,27:1,27:2,27:0,26:1,26:2,26:0,25:1,25:2,25:0,24:1,24:2,24:0,23:1,23
Bucket Info:
#    Info on pages in bucket
     (pageID,#tuples,freebytes,ovflow)
[ 0]  (d0,56,4,0) -> (ov0,15,737,-1)
[ 1]  (d1,57,2,3) -> (ov3,2,981,-1)
[ 2]  (d2,59,1,2) -> (ov2,2,976,-1)
[ 3]  (d3,54,7,1) -> (ov1,6,905,-1)
```

This shows that each data page has one overflow page, and that each data page has roughly the same number of tuples. The bucket starting at data page 0 has a few more tuples than th other buckets, because it has more tuples (15) in the overflow page. Note that page IDs in the overflow pages are distinguished by starting with `"ov"`. Note also that e.g. the data page at position 3 in the data file has an overflow page at position 1 in the overflow file; this is because page 3 filled up before pages 1 and 2.

One other thing to notice here is that the file has not expanded. It still has the 4 original data pages. Even if you added thousands of tuples, it would still have only 4 data pages. This is because linear hashing is not yet implemented. Implementing it is one of your tasks.

You could then use the `select` command to search for tuples using a command like:

```
$ ./select R '101,?,?'
```

This aims to find any tuple with 101 as the ID value; there will be one such tuple, since ID values are unique. This returns no solutions because query scanning is not yet implemented. Implementing it is another of your tasks.

Task 1: Multi-attribute Hashing

The current hash function does not use the choice vector to produce a combined hash value. It simply uses the hash value of the first attribute (the ID value) to generate a hash for the tuple. Your first task is to modify the `tupleHash()` function to use the relevant bits from each attribute hash value to form a composite hash. The choice vector determines the "relevant" bits. You can find more details on how a multi-attribute hash value is produced in the lecture slides and notes.

Task 2: Selection (Querying)

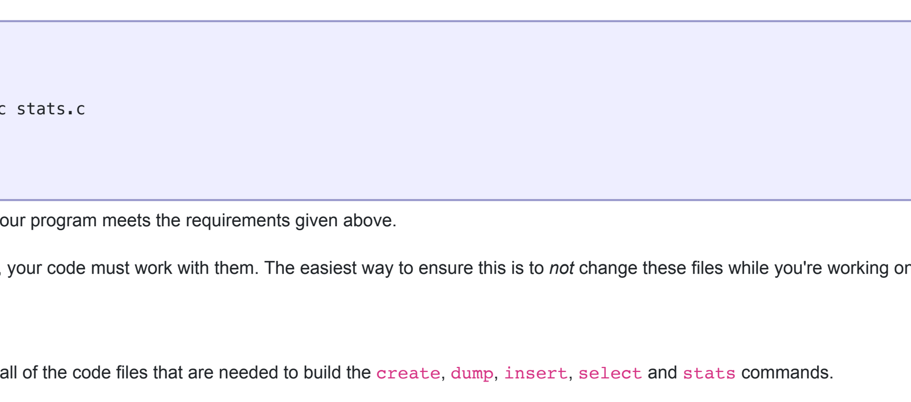
The query scan data type is found in `query.c` and `query.h` and is used only in `select.c`. At present, the data type is incomplete. You need to design a suitable query scanning data structure and implement the operations on it. The functions contain rough approximations to the algorithms you will need to build; you can find more details in the lecture slides and course notes. Most (all?) of the helper functions you'll need are in other data types, but you can add any others that you find necessary.

Task 3: Linear Hashing

As noted above, the current implementation is essentially a static version of single-attribute hashing. You need to add functionality to ensure that the file expands after every *c* insertions, where *c* is the page capcity *c = floor(B/R) ≈ 1024/(10*n)* where *n* is the number of attributes. Add one page at the end of the file and distribute the tuples in the "buddy" page (at index 2^d less) between the old and new pages. Determine where each tuple goes by considering *d+1* bits of the hash value. This will involve modifying the `addToRelation()` function, and will most likely require you to add new functions into the `reln.c` file (and maybe other files).

You can simplify the standard version of linear hashing by *not* removing overflow pages from the overflow chain of the data page they are attached to. This may result in some data pages having multiple empty overflow pages; this is ok if they are eventually used to hold more tuples.

The following diagram shows an example of what might occur during a page split:



How we Test your Submission

We will compile your submission for testing as follows:

```
$ unzip YourAss2.zip
$ tar xf OurMainPrograms.tar
# extracts our copies of ...
# create.c dump.c insert.c select.c stats.c
$ make
# should produce executables ...
# create dump insert select stats
```

We will then run a range of tests to check that your program meets the requirements given above.

Since we are using the original `create.c`, etc., your code must work with them. The easiest way to ensure this is to *not* change these files while you're working on the assignment.

Submission

You need to submit a single `tar` file containing all of the code files that are needed to build the `create`, `dump`, `insert`, `select` and `stats` commands.

Note that we will use the original versions of `create.c`, `dump.c`, `insert.c`, `select.c`, `stats.c`, and `gendata.c` for testing your code. This means that any functions you write must use the same interface as defined in the ADT `*.h` files.

When you want to submit your work, do the following:

```
$ cd Your/ass2/Directory
$ zip ass2.zip Makefile *.c *.h
```

Once you have generated the `ass2.zip` file, you can submit it via Webcms3 or the `give` command.

If you creathe tar file as above it will most likely contain `create.c`; this harmless as we will over-write them with our versions before testing.

Have fun, *jas*