Assignment 2

SIMC Signature Index Files

Last updated: **Tuesday 14th April 10:19pm**
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 **s**uper**im**posed **c**odeword (SIMC) signatures are implemented
- how partial-match retrieval searching is implemented using SIMC signatures

The goal is to build a simple implementation of a SIMC signature file, including applications to create SIMC files, insert tuples into them, and search for tuples based on partial-match retrieval queries.

Summary

Deadline:	23:59pm on **Sunday 26 April**
Late Penalty:	0.125 *marks* off the ceiling mark for each hour late (i.e. 3 marks/day)
Marks:	contributes **15 marks** toward your total mark for this course. (note that the item marks sum to 18; this will be converted into a mark out of 15 by multiplying by 15/18)
Groups:	do this assignment in pairs or individually (you can use the same groups as for Assignment 1)
Submission:	login to Course Web Site > Assignments > Assignment 2 > Submission > upload `ass2.tar` or login to any CSE server > `give cs9315 ass2 ass2.tar`
Workspace:	any machine wth a C compiler (preferably `gcc`); you do not need to use Grieg

The `ass2.tar` file must contain the `Makefile` plus all of the `*.c` and `*.h` files that are needed to compile the `create`, `insert` and `select` executables.

You are *not* allowed to change the following files: `create.c`, `insert.c`, `select.c`, `stats.c`, `dump.c`, `hash.h`, `hash.c`, `x1.c`, `x2.c`, `x3.c`. We supply them when we/you test your files, so any changes you make will be overwritten. Do not include them in the `ass2.tar` file. Details on how to build the `ass2.tar` file are given below.

Note that the `create.c`, `insert.c`, `select.c`, `stats.c`, `dump.c` code assumes that you honour the interfaces to the ADTs defined in the `*.[ch]` file pairs. If you change the interfaces to data types like `bits.h` and `page.h`, then your program will be treated as incorrect.

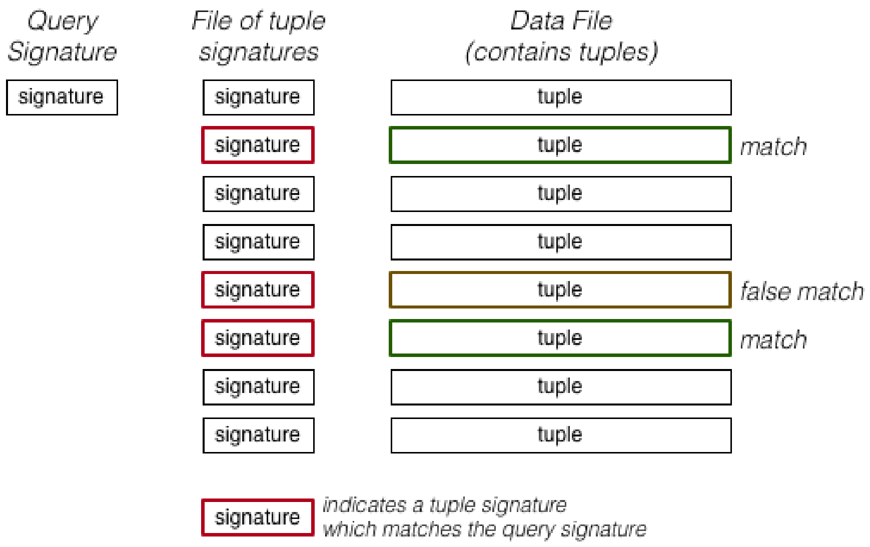
Make sure that you read this assignment specification *carefully and completely* before starting work on the assignment. Questions which indicate that you haven't done this will simply get the response "Please read the spec".

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

Introduction

Signatures are a style of indexing where (in its simplest form) each tuple is associated with a compact representation of its values (i.e. its *signature*). Signatures are used in the context of partial-match retrieval queries, and are particularly effective for large tuples. Selection is performed by first forming a *query signature*, based on the values of the known attributes, and then scanning the stored signatures, matching them against the query signature, to identify potentially matching tuples. Only these tuples are read from the data pages and compared against the query to check whether they are true matching tuples. Signature matching allows for "false matches", where the query and tuple signatures match, but the tuple is not a valid result for the query. Note that signature matching can be quite efficient if the signatures are small, and efficient bit-wise operations are used to check for signature matches.

The kind of signature matching described above uses one signature for each tuple (as in the diagram below). Other kinds of signatures exist, and one goal is to implement them and compare their performance to that of tuple signatures.

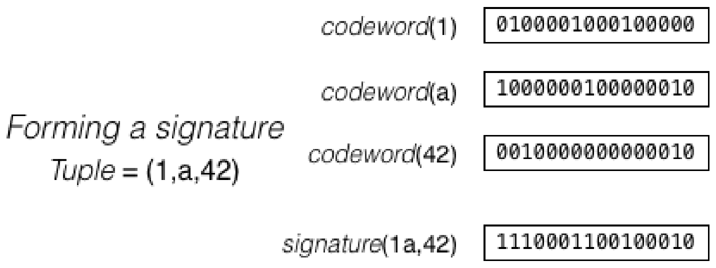

Query Signature
File of tuple signatures
Data File (contains tuples)
signature
tuple
match
false match
match
indicates a tuple signature which matches the query signature

In files such as the above, queries are evaluated as follows:

```
Input: pmr query, Output: set of tuples satisfying the query
qrySig = makeSignature(query)
Pages = {}  // set of pages containing possibly matching tuples
foreach tupSig in SignatureFile {
    if (tupSig matches qrySig) {
        // potential match
       PID = page of tuple associated with tupSig
       add PID to Pages
    }
}
Results = {}  // set of tuples satisfying query
foreach PID in Pages {
    buf = fetch data page PID
    foreach tuple in buf {
        // check for real match
        if (tuple satisfies query) add tuple to Results
    }
}
```

Note that above algorithm is an abstract view of what you must implement in your code. The function `makeSignature()` does not literally exist, but you need to build analogues to it in your code.

Signatures can be formed in several ways, but we will consider only signatures that are formed by **s**uper**im**posing **c**odewords (SIMC). Each codeword is formed using the value from one attribute.


Forming a signature
Tuple = (1,a,42)
codeword(1) 0100001000100000
codeword(a) 1000000100000010
codeword(42) 0010000000000010
signature(1a,42) 1110001100100010

In our system, a SIMC relation `R` is represented by five physical files:

- `R.info` containing global information such as
  - the number of attributes and size of each tuple
  - the number of data pages and number of tuples
  - the sizes of the various kinds of signatures
  - the number of signatures and signature pages
  - etc. etc. etc.

  The `R.info` file contains a copy of the `RelnParams` structure given in the `reln.h` file (see below).

- `R.data` containing data pages, where each data page contains
  - a count of the number of tuples in the page
  - the tuples (as comma-separated character sequences)

Each data page has a capacity of *c* tuples. If there are *n* tuples then there will be $b = \lceil n/c \rceil$ pages in the data file. All pages except the last are full. Tuples are never deleted.

- `R.tsig` containing tuple signatures, where each page contains
    - a count of the number of signatures in the page
    - the signatures themselves (as bit strings)

  Each tuple signature is formed by superimposing the codewords from each attribute in the tuple. If there are *n* tuples in the relation, there will be *n* tuple signatures, in b_t pages. All tuple signature pages except the last are full.

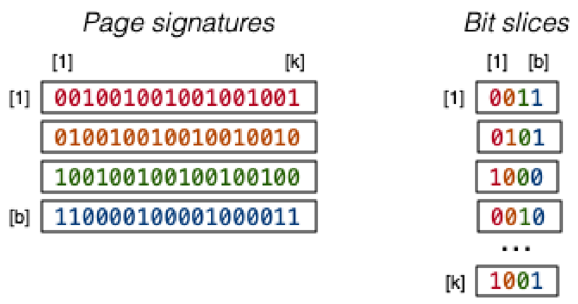
- `R.psig` containing page signatures, where each page contains
    - a count of the number of signatures in the page
    - the signatures themselves (as bit strings)

  Page signatures are much larger than tuple signatures, and are formed by superimposing the codewords of all attribute values in all tuples in the page. There is one page signature for each page in the data file.

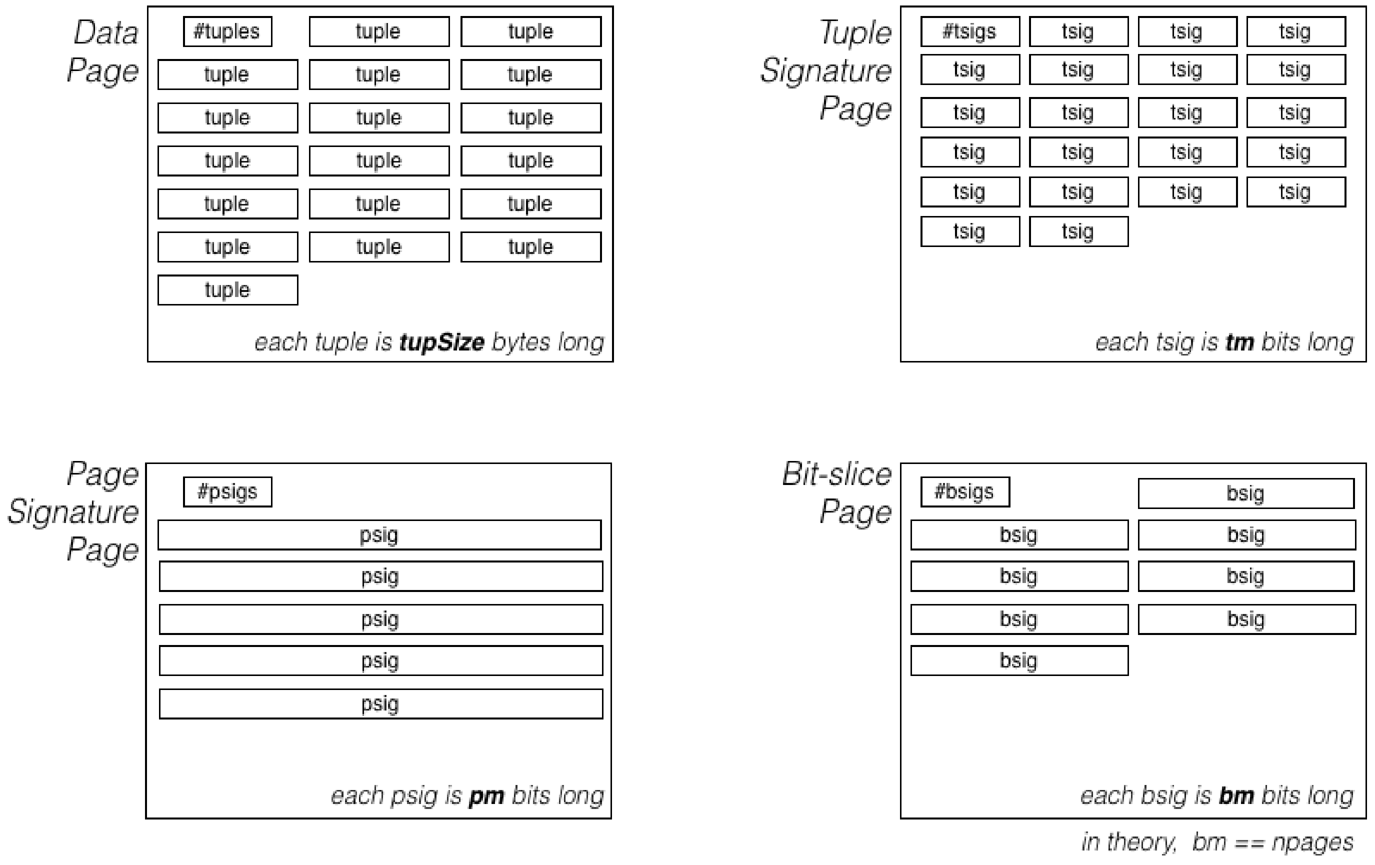
- `R.bsig` containing bit-sliced signatures, where each page contains
    - a count of the number of signatures in the page
    - the bit-slices themselves (as bit strings)

  Bit-slices give an alternate 90^o-rotated view of page signatures. If there are *b* data pages, then each bit-slice is *b*-bits long. If page signatures are *pm* bits long, then there are *pm* bit-slices.

The following diagram gives a very simple example of the correspondence between page signatures and bit-slices:



The different types of pages (tuple, signature, slice) were described above. Internally, all pages have a similar structure: a counter holding the number of items in the page, and the items themselves (tuples or signatures or slices). All of the items in a page are the same size. The following diagram shows the structure of pages in the files of a SIMC relation:



We have developed some infrastructure for you to use in implementing SIMC files. The code we give you is not complete; you can find the bits that need to be completed by searching for `TODO` in the code.

How you implement the missing parts of the code is up to you, but your implementation must conform to the conventions used in our code. In particular, you should preserve the interfaces to the supplied modules (e.g. `Bits`, `Reln`, `Query`, `Tuple`) and ensure that your submitted modules work with the supplied code in the `create`, `insert` and `select` commands.

Commands

In our context, SIMC-indexed relations are a collection of files that represent one relational table. These relations can be manipulated by a number of supplied commands:

`create` `RelName #tuples #attrs 1/pF`

Creates an empty relation called *RelName* with all tuples having *#attrs* attributes. The *#tuples* parameter gives the expected number of tuples that are likely to be inserted into a relation; this, in turn, determines parameters like the number of data pages and length of bit-sliced superimposed codewords. The *1/pF* parameter gives the inverse of the false match probability; for example, a value of 1000 gives a false match probability of 1/1000 (0.001).

These parameters are combined using the formulas given in lectures to determine how large tuple- and page-signatures are. Each bit-slice has a number of bits equal to the number of data pages, which is determined from *#attrs*, *#tuples* and the page size.

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

```
create table R ( a₁ integer, 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 relation called `abc` where each tuple has 4 attributes and the indexing has a false match probability of 1/100. The relation can hold up to 10000 tuples (it can actually hold more, but only the first 10000 will be indexed via the bit-sliced signatures).

```
$ ./create  abc  10000  4  100
```

`insert` `RelName`

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 alpha-numeric characters and `'-'`. The characters `','` (field separator) and `'?'` (query wildcard) are treated specially.

Since all tuples need to be the same length, it is simplest to use `gendata` to generate them, and pipe the generated tuples into the `insert` command

`select` `RelName QueryString SigType`

Takes a "query tuple" on the command line, and finds all tuples in the data pages of the relation *RelName* 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). Some examples, 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 1
?,abc,?    # matches any tuple with abc as the value of attribute 2
10,abc,?   # matches any tuple with 10 and abc as the values of attributes 1 and 2
```

There are also a number of auxiliary commands to assist with building and examining relations:

`gendata` `#tuples #attributes [startID] [seed]`

Generates a specified number of *n*-attribute tuples in the appropriate format to insert into a created relation. All tuples are the same format and look like

```
UniqID, RandomString, a3-Num, a4-Num, . . . , aₙ-Num
```

For example, the following 4-attribute tuples could be generated by a call like `gendata 1000 4`

```
7654321,aTwentyCharLongStrng,a3-013,a4-001
3456789,aTwentyChrLongString,a3-042,a4-128
```

Of course, the above call to `gendata` will generate 1000 tuples like these.

A tuple is represented by a sequence of comma-separated fields. The first field is a unique 7-digit number; the second field is a random 20-char string (most likely unique in a given database); the remaining fields have a field identifier followed by a non-unique 3-digit number. The size of each tuple is

```
7+1 + 20+1 + (n-2)*(6+1)-1  = 28 + 7*(n-2) bytes
```

The `-1` is because the last attribute doesn't have a trailing comma, and `(n-2)*(6+1)` assumes that it does.

Note that tuples are limited to at most 9 attributes, which means that the maximum tuple size is a modest 77 bytes. (If you wish, you can work with larger tuples by tweaking the `gendata` and `create` commands and the `newRelation()` function, but this not required for the assignment).

stats `RelName`

Prints information about the sizes of various aspects of the relation. Note that some aspects are static (e.g. the size of tuples) and some aspects are dynamic (e.g. the number of tuples). An example of using the `stats` command is given below.

You can use it to help with debugging, by making sure that the files have been correctly built after the `create` command, and that the files have been correctly updated after some tuples have been `insert`ed.

dump `RelName`

Writes all tuples from the relation *RelName*, one per line, to standard output. This is like an inverse of the `insert` command. Tuples are dumped in a form that could be used by `insert` to rebuild a database.

You can use it to help with debugging, by making sure that the tuples are inserted correctly into the data file.

Setting Up

You should make a working directory for this assignment and put the supplied code there, and start reading 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/cs9315/20T1/assignments/ass2/ass2.zip
```

You should see the following files in the directory:

```
$ ls
Makefile        dump.c          psig.c          stats.c         x1.c
bits.c          gendata.c       psig.h          tsig.c          x2.c
bits.h          hash.c          query.c         tsig.h          x3.c
bsig.c          hash.h          query.h         tuple.c
bsig.h          insert.c        reln.c          tuple.h
create.c        page.c          reln.h          util.c
defs.h          page.h          select.c        util.h
```

The `.h` files define data types and function interfaces for the various types used in the system. The corresponding `.c` files contain the implementation of the functions on the data type. The remaining `.c` files either provide the commands described above, or are test harnesses for individual types (`x1.c`, `x2.c`, `x3.c`). You can add additional testing files, bu there is no need to submit them.

The above files give you a partial implementation of SIMC indexing. You need to complete the code so that it provides the functionality described above.

You should be able to build the supplied partial implementation via the following:

```
$ make
gcc -std=gnu99 -Wall -Werror -g   -c -o query.o query.c
gcc -std=gnu99 -Wall -Werror -g   -c -o page.o page.c
gcc -std=gnu99 -Wall -Werror -g   -c -o reln.o reln.c
gcc -std=gnu99 -Wall -Werror -g   -c -o tuple.o tuple.c
gcc -std=gnu99 -Wall -Werror -g   -c -o util.o util.c
gcc -std=gnu99 -Wall -Werror -g   -c -o tsig.o tsig.c
gcc -std=gnu99 -Wall -Werror -g   -c -o psig.o psig.c
gcc -std=gnu99 -Wall -Werror -g   -c -o bsig.o bsig.c
gcc -std=gnu99 -Wall -Werror -g   -c -o hash.o hash.c
gcc -std=gnu99 -Wall -Werror -g   -c -o bits.o bits.c
gcc -std=gnu99 -Wall -Werror -g   -c -o create.o create.c
gcc -o create create.o query.o page.o reln.o tuple.o util.o tsig.o psig.o bsig.o hash.o bits.o -lm
gcc -std=gnu99 -Wall -Werror -g   -c -o insert.o insert.c
gcc   insert.o query.o page.o reln.o tuple.o util.o tsig.o psig.o bsig.o hash.o bits.o   -o insert
gcc -std=gnu99 -Wall -Werror -g   -c -o select.o select.c
gcc   select.o query.o page.o reln.o tuple.o util.o tsig.o psig.o bsig.o hash.o bits.o   -o select
gcc -std=gnu99 -Wall -Werror -g   -c -o stats.o stats.c
gcc   stats.o query.o page.o reln.o tuple.o util.o tsig.o psig.o bsig.o hash.o bits.o   -o stats
gcc -std=gnu99 -Wall -Werror -g   -c -o gendata.o gendata.c
gcc -o gendata gendata.o util.o -lm
gcc -std=gnu99 -Wall -Werror -g   -c -o dump.o dump.c
gcc   dump.o query.o page.o reln.o tuple.o util.o tsig.o psig.o bsig.o hash.o bits.o   -o dump
gcc -std=gnu99 -Wall -Werror -g   -c -o x1.o x1.c
gcc -o x1 x1.o query.o page.o reln.o tuple.o util.o tsig.o psig.o bsig.o hash.o bits.o
gcc -std=gnu99 -Wall -Werror -g   -c -o x2.o x2.c
gcc -o x2 x2.o query.o page.o reln.o tuple.o util.o tsig.o psig.o bsig.o hash.o bits.o
gcc -std=gnu99 -Wall -Werror -g   -c -o x3.o x3.c
gcc -o x3 x3.o query.o page.o reln.o tuple.o util.o tsig.o psig.o bsig.o hash.o bits.o
```

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

The `gendata` command should work completely without change. For example, the following command generates 5 tuples, each of which has 4 attributes. Values in the first attribute are unique; values in the second attribute are highly likely to be unique. Note that the third and fourth attributes cycle through values at different rates, so they won't always have the same number.

```
$ ./gendata 5 4
1000000,lrfkQyuQFjKXyQVNRTyS,a3-000,a4-000 -> 0
1000001,FrzrmzlYGFvEulQfpDBH,a3-001,a4-001 -> 0
1000002,lqDqrrCRwDnXeuOQqekl,a3-002,a4-002 -> 0
1000003,AITGDPHCSPIjtHbsFyfv,a3-003,a4-003 -> 0
1000004,lADzPBfudkKlrwqAOzMi,a3-004,a4-004 -> 0
```

The `create` command itself is complete, but some of the functions it calls are not complete. It will allow you to make an empty relation, although without a complete bit-slice file (you add this as one of the assignment tasks). The `stats` command is complete and can display information about a relation. Using these commands, you could do the following: use the `create` command to create an empty relation which can hold 4-attribute tuples and able to index up to 5000 tuples (using bit-slices), with a false match probability of 1/1000. The `stats` command then displays the generated parameter values.

```
$ ./create R 5000 4 1000
$ ./stats R
Global Info:
Dynamic:
  #items:  tuples: 0  tsigs: 0  psigs: 0  bsigs: 0
  #pages:  tuples: 1  tsigs: 1  psigs: 1  bsigs: 1
Static:
  tups   #attrs: 4  size: 42 bytes  max/page: 97
  sigs   bits/attr: 9
  tsigs  size: 64 bits (8 bytes)  max/page: 511
  psigs  size: 5584 bits (698 bytes)  max/page: 5
  bsigs  size: 56 bits (7 bytes)  max/page: 584
$
```

You can apply the formulae for calculating the various quantities to check that the above values make sense. Note that the bits for signatures are rounded up to the next multiple of 8 (why waste a few bits?). Note also that all pages are defined to be 4096 bytes. Finally, note that `create` makes a file with one empty page for each of the files holding tuples and signatures.

As supplied, the `insert` command inserts tuples into the data pages, but does not generate any signatures. Using `gendata` is the easiest (and safest) way to add valid tuples. You can then check that the tuples have been inserted via the `dump` command, and see how the parameters have changed using `stats` again.

```
$ ./gendata 5 4 | ./insert -v R
Inserting: 1000000,lrfkQyuQFjKXyQVNRTyS,a3-000,a4-000
Inserting: 1000001,FrzrmzlYGFvEulQfpDBH,a3-001,a4-001
Inserting: 1000002,lqDqrrCRwDnXeuOQqekl,a3-002,a4-002
Inserting: 1000003,AITGDPHCSPIjtHbsFyfv,a3-003,a4-003
```

```
Inserting: 1000004,lADzPBfudkKlrwqAOzMi,a3-004,a4-004
$ ./stats R
Global Info:
Dynamic:
  #items:  tuples: 5  tsigs: 0  psigs: 0  bsigs: 0
  #pages:  tuples: 1  tsigs: 1  psigs: 1  bsigs: 1
Static:
  tups   #attrs: 4  size: 42 bytes  max/page: 97
  sigs   bits/attr: 9
  tsigs  size: 64 bits (8 bytes)  max/page: 511
  psigs  size: 5584 bits (698 bytes)  max/page: 5
  bsigs  size: 56 bits (7 bytes)  max/page: 584
$
```

Note that the only difference between the above stats and the stats for the newly-created file is the 5 tuples. There are no signatures, no new pages, etc.

The `dump` command is complete; it simply scans the data file and displays any tuples it finds, e.g.

```
$ ./dump R
1000000,lrfkQyuQFjKXyQVNRTyS,a3-000,a4-000
1000001,FrzrmzlYGFvEulQfpDBH,a3-001,a4-001
1000002,lqDqrrCRwDnXeuOQqekl,a3-002,a4-002
1000003,AITGDPHCSPIjtHbsFyfv,a3-003,a4-003
1000004,lADzPBfudkKlrwqAOzMi,a3-004,a4-004
$
```

The `select` command, as supplied, is not complete. However, once it is working (at least with tuple signatures), you should be able to ask queries like:

```
$ ./select  R  '1000001,?,?'  t       # not enough attrs
Invalid query: 101,?,?
$ ./select  R  '1000001,?,?,?'  t
1000001,FrzrmzlYGFvEulQfpDBH,a3-001,a4-001
Query Stats:
# signatures read:   5
# sig pages read:    1
# tuples examined:   5
# data pages read:   1
# false match pages: 0
$ ./select  R  '1000001,?,a3-002,?'  t
Query Stats:
# signatures read:   5
# sig pages read:    1
# tuples examined:   0
# data pages read:   0
# false match pages: 0
$ ./select  R  '1000001,?,a3-002,?'  x
Query Stats:
# signatures read:   0
# sig pages read:    0
# tuples examined:   5
# data pages read:   1
# false match pages: 1
```

Some explanation:

- The second query finds a match because there is a tuple with the value `1000001` for its first attribute. The `?` represent "don't care" or wild-card values.
- The third query fails because the tuple with `1000001` for its first attribute, does not have the value `a3-002` for its third attribute.
- The fourth query performs a linear scan of the data file. Since the query itself is the same as the third query, there are no matching tuples. This query reads every data page (there is only one). Any data page read, which does not contain matching tuples, is counted as a "`false page match`".
- The `t` at the end of the query tells the query evaluator to use tuple signatures as a first-pass filter. Other possibilities are `p` for page signatures or `b` for bit-sliced signatures. If you use a character other than `t`, `p`, or `b`, or don't specify a signature type, the evaluator uses a linear scan and checks all tuples.
- With all types of signatures, queries run in two phases:

- use the signatures to determine which pages may contain matching tuples
- read each of these pages and check all tuples to find real matches

- The query statistics are maintained in a `Query` data structure while the query is executing.

Data Types

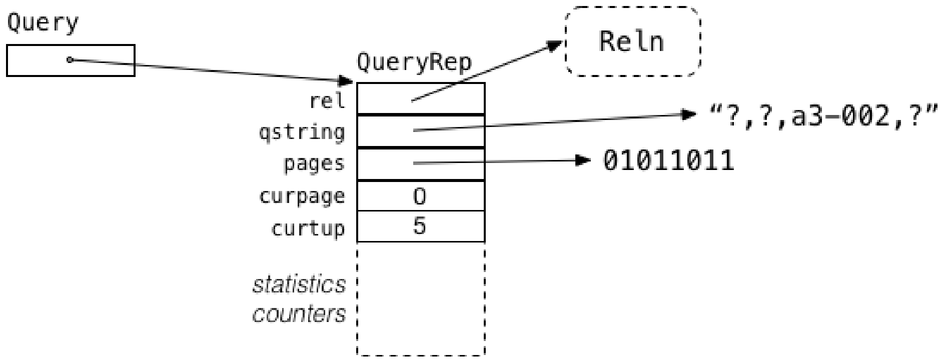
There are four important data types defined in the system:

Relations (data type `Reln`)

Relations are defined by three data types: `Reln, RelnRep, RelnParams`. `Reln` is just a pointer to a `RelnRep` object; this is useful for passing to functions that need to modify some aspect of the relation structure. `RelnRep` is a representation of an open relation and contains the parameters, plus file descriptors for all of the open files. `RelnParams` is a list of various properties of the database. See `reln.h` for details.

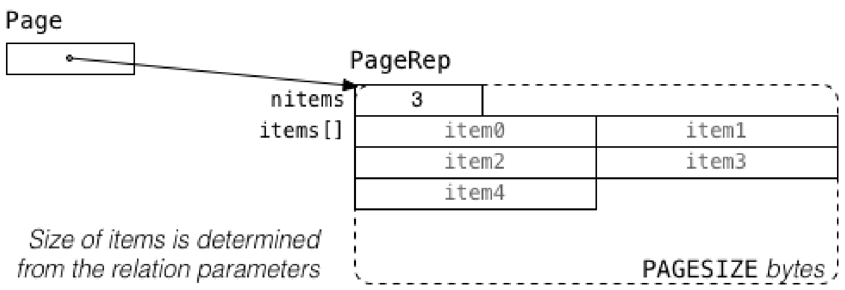
Queries (data type `Query`)

Queries are defined via a `QueryRep` structure which contains fields to represent the current state of the scan for the query, plus a collection of statistics counters. It is essentially like the query iteration structures described in lectures, and is used to control and monitor the query evaluation. The `QueryRep` structure also contains a reference to the relation being queried, and a copy of the query string. The `Query` data type is simply a pointer to a `QueryRep` structure. See `query.h` for details. The following diagram might also help:



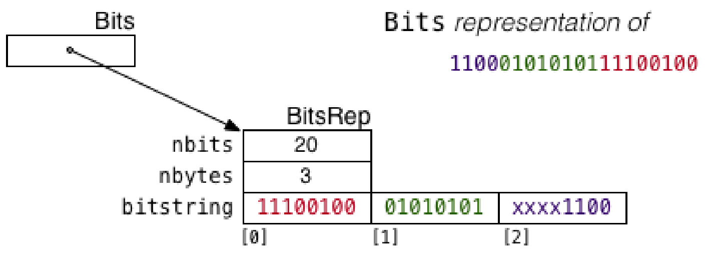
Pages (data type `Page`)

Pages are defined via a `PageRep` structure which contains a counter for the number of items, and then an array of bytes containing the actual items, whether they are tuples or signatures or slices. The size of each type of item is held in the `RelnParams` structure, and so `Page`s are typically considered in conjunction with `Reln`s. The `Page` data type is simply a pointer to a `PageRep` structure. See `page.h` for details. The following diagram might also help:



Bit-strings (data type `Bits`)

Bit-strings are defined via a `BitsRep` structure which contains two counters (one for the number of bits, and the other for the number of bytes used to represent the bit-string). The `BitsRep` structure also contains an array of bytes which hold the bits in the string; the array is created when and instance of a `Bits` data type is created. Note that `Bits` is an ADT, so the concrete data structure is hidden from its clients; the `Bits` data type is simply a pointer to a `BitsRep` structure. See `bits.c` for details of the data structure, and `bits.h` for the function interface. The following diagram might help:



Tuple (data type `Tuple`)

> Tuples are just character sequences (like C strings). See `tuple.h` for details.

There are also a range of (hopefully) self-explanatory data types defined in `defs.h`. The various signature types are represented as bit-strings (`Bits`).

Goal

Your goal for this assignment is to complete the implementation of the various components of the system, so that it can handle all three kinds of signatures. This includes adding/modifying signatures when new tuples are added, and using these signatures in answering queries. Note that ew don't consider operations like `DELETE` or `UPDATE` in this assignment.

The header (`.h`) files contain definitions of the data types used in the system.

Each of the source code (`.c`)files contains comments on each function, describing briefly what it should do. Some functions contain `TODO` comments to indicate where you need to complete them. You can put all of the code in the indicated function, or you can write new functions that these functions use.

You are free to change any file except `create.c`, `insert.c`, `select.c` and `hash.c`. Since you can't change these files, you also cannot change the interfaces to the data types that they use (`Reln`, `Query`, `Page`, `Bits`). Basically, all of the functions mentioned in the `.h` files for these types must exist, with the same interface, but you can implement their internals however you like. You can also add extra functions to each data type (i.e. extend its interface) if that helps.

The files `x1.c`, `x2.c`, `x3.c` can be changed, but aren't relevant to the assignment, except to help with debugging some of the data types.

Task 1: A Bit-string Type (2 marks)

Implement all of the incomplete functions in the `bits.c` file, to produce a working bit-string data type. The functions to complete are flagged with `TODO`, and the purpose of each should be clear from the comment at the start of the function and its name. The `x1.c` file contains some simple test cases for the `Bits` type.

Task 2: Scanning for Results (3 marks)

After you have `Bits` working, you can start to implement query evaluation, although without indexing. The `startQuery()` function parses the query string and then uses the appropriate type of signature to generate a list of pages which potentially contain matching tuples. This list is implemented as a bit-string where a 1 indicates a page which needs to be checked for matches. At this stage, all of the signature types mark all pages as potential matches, so all pages need to be checked.

Note that the `startQuery()` function can return `NULL`. It should do so only if the query string contains the wrong number of attributes for the relation.

Before this will work, you need to implement the `scanAndDisplayMatchingTuples()` function, which performs the check for matching tuples in each of the marked pages. This function, as well as finding and displaying result tuples, maintains the query statistics for number of data pages read, and number of pages that were read but contained no matching tuples.

For this task, you need to complete the `scanAndDisplayMatchingTuples()` function from the `query.c` file. This function behaves roughly as follows:

```
foreach PID in 0 .. npages-1 {
    if (PID is not set in MatchingPages)
        ignore this page
    for each tuple T in page PID {
        if (T matches the query string)
            display it as a query result
    }
    if (no tuples in page PID are results)
        count it as a false match page
}
```

Task 3: Tuple Signatures (4 marks)

Implement indexing by using tuple-based signatures (i.e. each tuple has its own signature, stored in the *Rel*.`tsig` file). You will need to complete the `makeTupleSig()` and `findPagesUsingTupSigs()` functions in the `tsig.c` file, and add some code to the `addToRelation()` function in `reln.c`.

The `addToRelation()` function inserts a tuple into the next available slot in the data file, but currently does nothing with signatures. You should add code here which generates a tuple signature for the new tuple and inserts it in the next available slot in the *Rel*.tsig file.

The `makeTupleSig()` function takes a tuple and returns a bit-string which contains a superimposed codeword signature for that tuple. It behaves roughly as follows:

```
Tsig = AllZeroBits
for each attribute A in tuple T {
    CW = codeword for A
    Tsig = Tsig OR CW
}
```

A method for computing codewords is given in the lecture notes.

The `findPagesUsingTupSigs()` take a tuple signature and scans the *Rel*.tsig file, comparing that signature to the stored tuple signatures. It builds a bit-string showing which pages contain at least one "matching" tuple. It behaves roughly as follows:

```
QuerySig = makeTupleSig(Query)
Pages = AllZeroBits
foreach Tsig in tsigFile {
    if (Tsig matches QuerySig) {
        PID = data page for tuple corresponding to Tsig
        include PID in Pages
    }
}
```

Note that the ith tuple in the data file has its correpsonding signature as the ith signature in the *Rel*. `tsig` file. However, since tuples and tuple signatures are different sizes, the page that the signature appears on will not necessarily have the same page ID as the page in which the corresponding tuple is located.

Task 4: Page Signatures (4 marks)

Implement indexing using page-level signatures (`psigs`).

This is similar to how tuple-level signature indexing is done, except that the sigantures are larger. The functions that you need to complete are `makePageSig()` and `findPagesUsingPageSigs()` in the `psig.c` file. You will also need to add more code to the `addToRelation()` function to maintain page signatures when new tuples are inserted.

One major difference between tuple signatures and page signatures is that page signatures are not a one-off insertion. When a new tuple is added, its page-level signature needs to be included page signature for the page where it it is inserted. The process can be described roguhly as follows:

```
new Tuple is inserted into page PID
Psig = makePageSig(Tuple)
PPsig = fetch page signature for data page PID from psigFile
merge Psig and PPsig giving a new PPsig
update page signature for data page PID in psigFile
```

The `makePageSig()` function be used to generate a page-level signature for the query, and then used to generate a bit-string of matching pages roughly as follows:

```
QuerySig = makePageSig(Query)
Pages = AllZeroBits
foreach Psig in psigFile {
    if (Psig matches QuerySig) {
        PID = data page corresponding to Psig
        include PID in Pages
    }
}
```

Task 5: Bit-sliced Signatures (5 marks)

Implement indexing using bit-sliced page signatures.

Each bit-slice is effectively a list of pages that have a specific bit from the page-signature set to 1 (e.g. if a page-level signature has bit 5 set to one, the bit-slice 5 has a 1 bit for every page with a page signature where bit 5 is set). This drives both the updating of bit-slices and their use in indexing.

You will need to modify the functions: `newRelation()` in `reln.c`, `addToRelation()` in `reln.c`, and `findPagesUsingBitSlices()` in `bsig.c`. The modifications to `newRelation()` are relatively straightforward, but remember to update the relation parameters appropriately.

The `addToRelation()` should take a tuple, produce a page signature for it, then update all of the bit-slices corresponding to 1-bits in the page signature. This can be described roughly as follows:

```
PID = data page where new Tuple inserted
Psig = makePageSig(Tuple)
for each i in  0..pm-1 {
    if (Psig bit[i] is 1) {
        Slice = get i'th bit slice from bsigFile
        set the PID'th bit in Slice
        write up
```