`

Implement of Database Signature-Indexes

Aims

This project aims to give you an understanding ofhow

2021/8/24

- database files are structured and accessed
- how **s**uper**im**posed **c**odeword (SIMC) signatures are implemented
- how con**cat**enated **c**odeword (CATC) signatures are implemented
- how partial-match retrieval searching is implemented using signatures
- the performance differences between different types of signatures

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

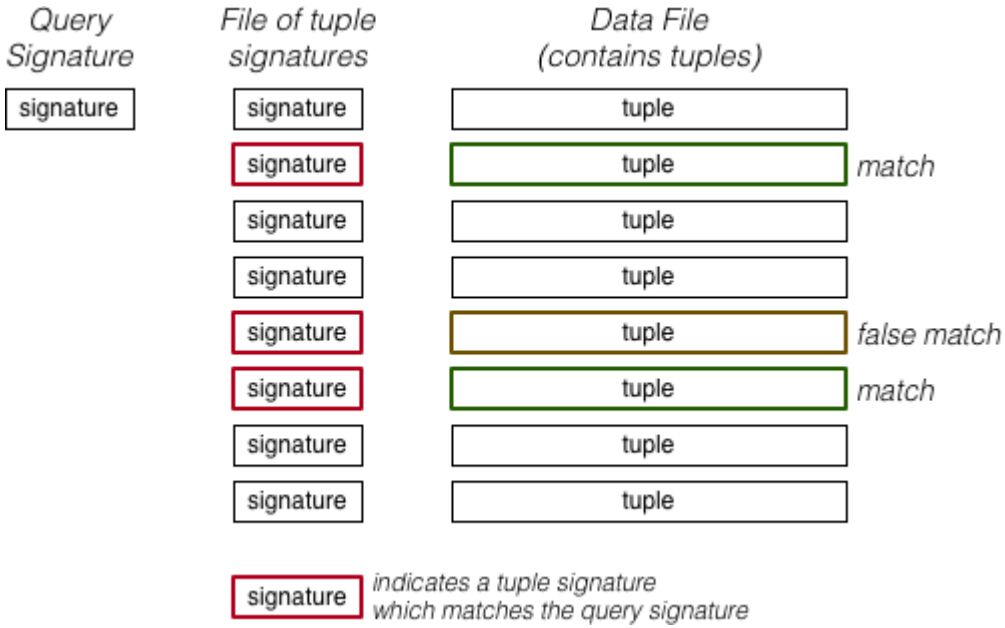
Note: this project *does not* require you to do anything with PostgreSQL.

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 can result in "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.

`



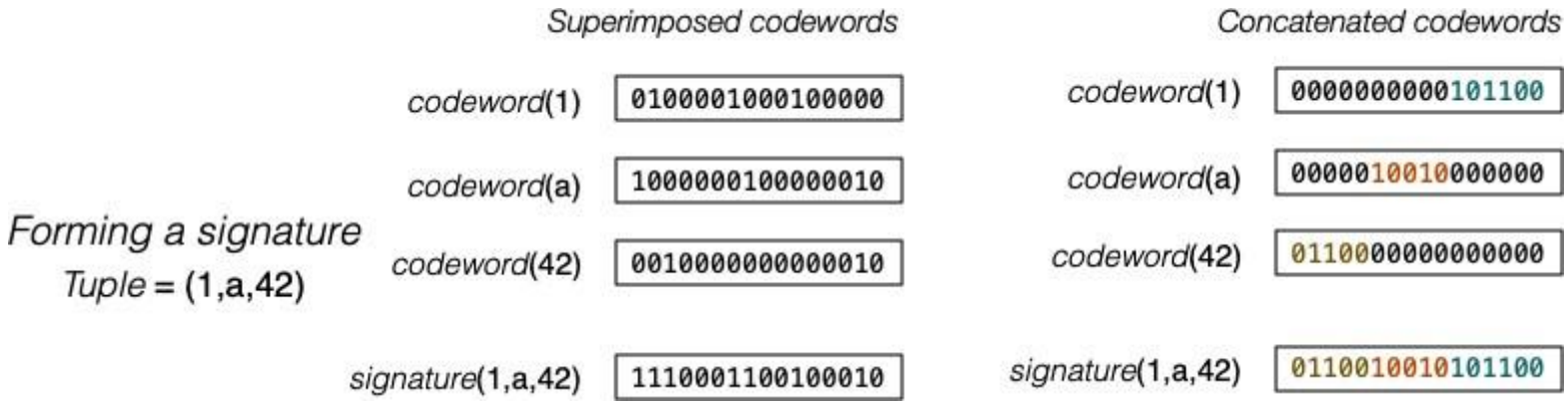
In files such as the above, queries are evaluated as follows:

```
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
    }
}
```

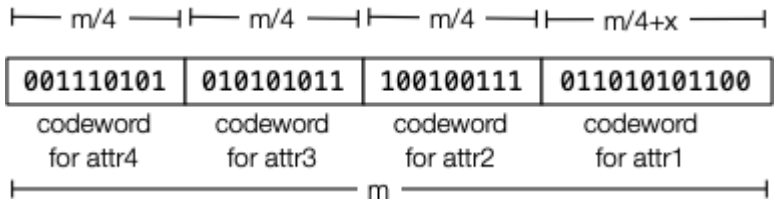
`

Signatures

We will consider two methods for building signatures: **s**uper**im**posed **c**odewords (SIMC), and con**cat**enated **c**odewords (CATC). Each codeword is formed using the value from one attribute.



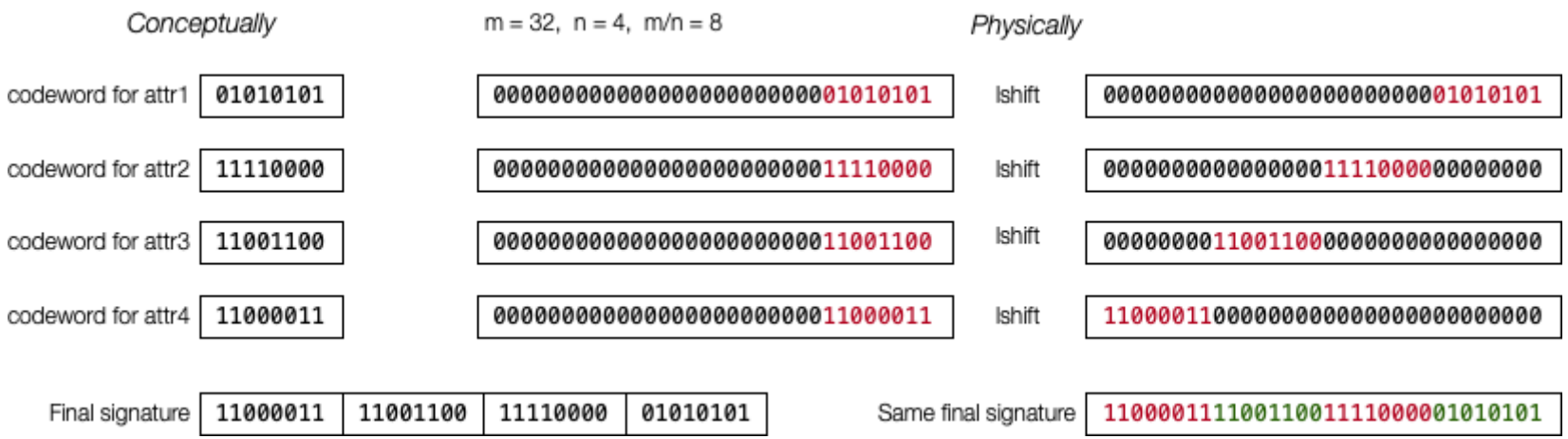
In SIMC signatures, all codewords and signatures are *m* bits wide, and each codeword has *k* bits set to 1. In CATC signatures, signatures are *m* bits wide, but codewords occupy approximately equal numbers of bits of the signature. Since there are *m* bits in the signature and *n* attributes, each codeword is *u* = *m/n* bits long, except for the lower-order codeword (the one for the first attribute). This codeword is *u* bits long + *m mod n* bits, so that the total number of codeword bits is equal to *m*. The following diagram shows the parts of a concenated codeword signature:



In this example, the signature is *m*=42 bits wide. Each codeword, except the lower-order one, is *u*=10 bits wide. The lower-order codeword has two extra bits to make up to 42. Each codeword has half of its bits set to 1; in CATC codewords, *k* = u/2. This is different to SIMC codewords, where we need to determine *k* to ensure that roughly half of the bits in the signature are set to 1.

The way we build CATC signatures is conceptually straightforward: form *n* codewords, each of which is *m/n* bits wide, and concatenate them. In practice, we build *n* codewords, each of which is *m* bits wide, with the lower-order *u* bits set as the codeword, and then, shifted into the position that it would occupy in a concatenated codeword signature. The diagram below illustrates this:

`



Note: the fact individual codewords are 8-bits long is not intended to suggest that codewords will always be whole bytes. Individual codewords would be 6-bits if *m* = 24, or 9-bits if *m* = 36. And, as noted above, if *m* = 42, the codeword for attribute 1 would be 12-bits and all other attributes would have 10-bit codewords.

In subsequent discussions, we denote the length of tuple signatures as *m*, the length of page signatures as m_p, and the length of CATC codewords as *u* (remembering that all SIMC codewords have the same length as the signatures they produce).

Relations

In our system, a 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 base type of signatures (`simc` or `catc`)
    - 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* = ⌈ *n/c*⌉ 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 incorporating the codewords from each attribute in the tuple. How this is done differs between SIMC and CATC, but the overall result is a single *m*-bit long signature. 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.

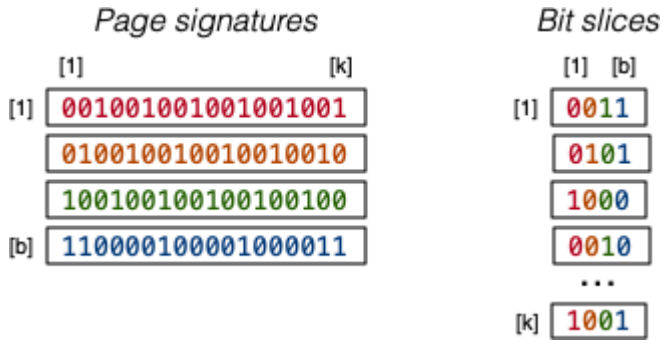
- `R.psig` containing page signatures, where each page contains
    - a count of the number of signatures in the page
    - the signatures themselves (as bit strings)

  Page signatures are much larger than tuple signatures, and are formed by incorporating the codewords of all attribute values in all tuples in the page. How this is done differs between SIMC and CATC, but the result is a single m_p-bit long signature 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:

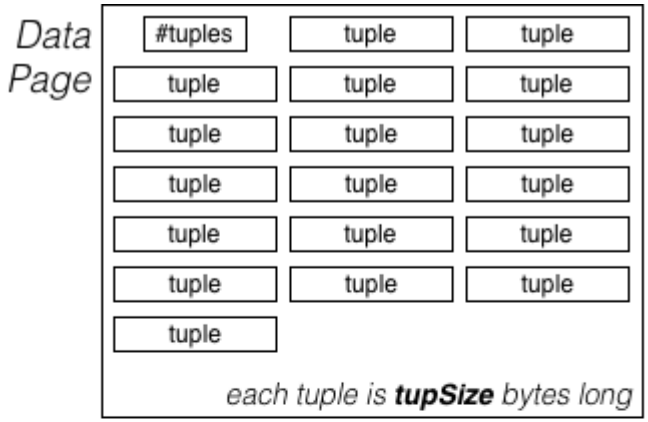


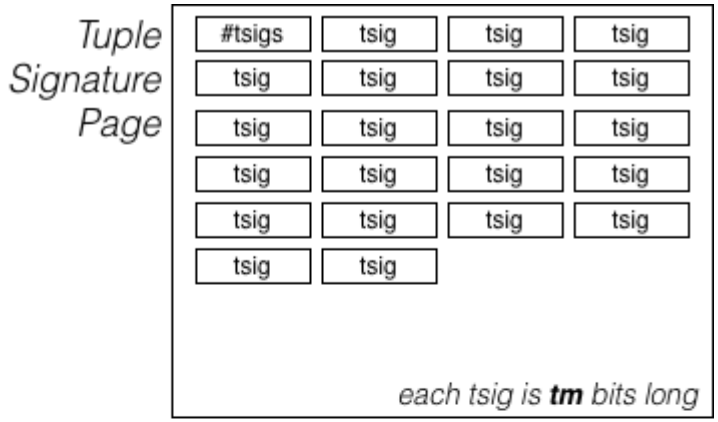
Pages

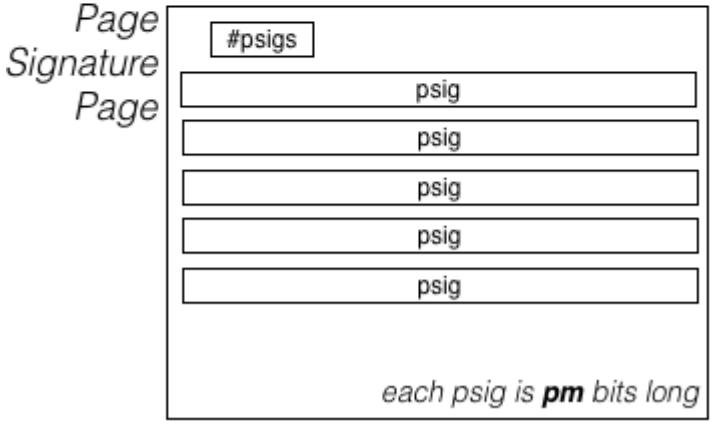
The different types of pages (tuple, signature, slice) were described above. Internally, all pages have a similar structure: a counter holding the number of items in the page, and the items themselves (tuples or signatures or slices). All of the items in a page are the same size. The following diagram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

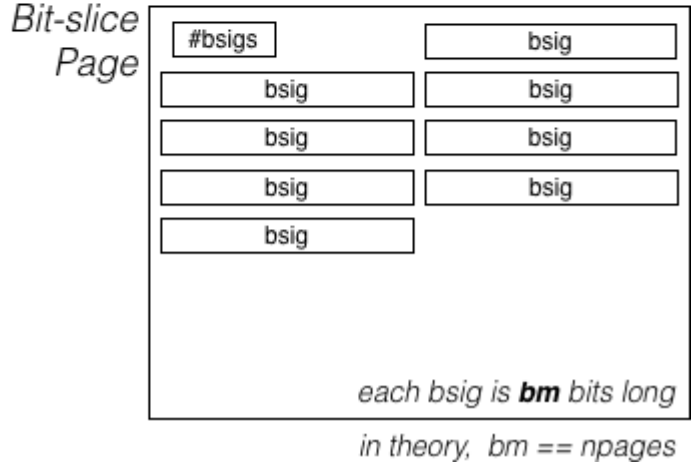
`

structure of pages in the files of a signature-indexed relation:









Commands

`

In our context, signature-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 SigType #tuples #attrs 1/pF`

Creates an empty relation called *RelName* with all tuples having *#attrs* attributes. *SigType* specifies how signatures should be formed, and can have one of two values: `simc` or `catc`. 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 corresponds to 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  simc  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 IndexType`

Takes a "query tuple" on the command line, and finds all tuples in the data pages of the relation *RelName* that match the query. *IndexType* has a value of either `t`, `p` or `b`, indicating whether it should used the **t**uple, **p**age, or **bi**t-sliced signatures. 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*,...,a_n-*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.

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.

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 see the following files in the directory: 2021/8/24

```
$ 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.

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
```


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. 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 simc 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   simc  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   simc  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:

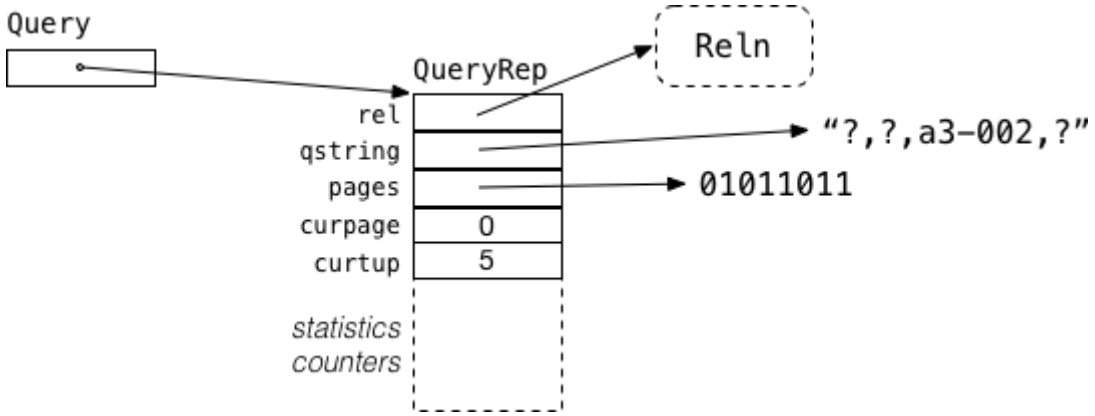
Relations (data type `Reln`)

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`)

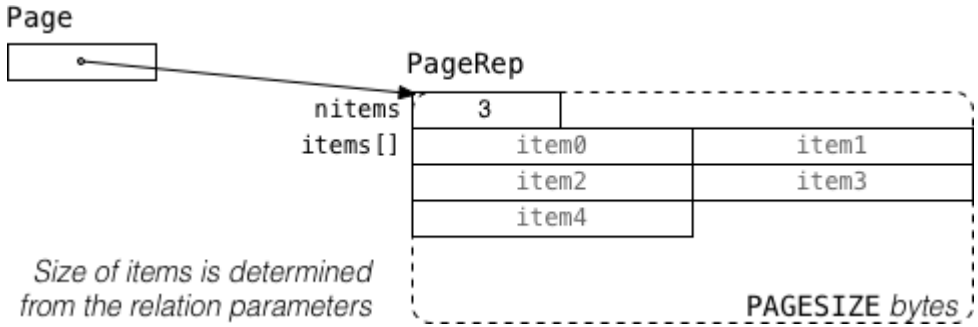
Queries are defined via a `QueryRep` structure which contains fields to represent the current state of the scan for the query, plus a collection of statistics counters. It is essentially like the query iteration structures described in lectures, and is used to control and monitor the query evaluation. 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`)

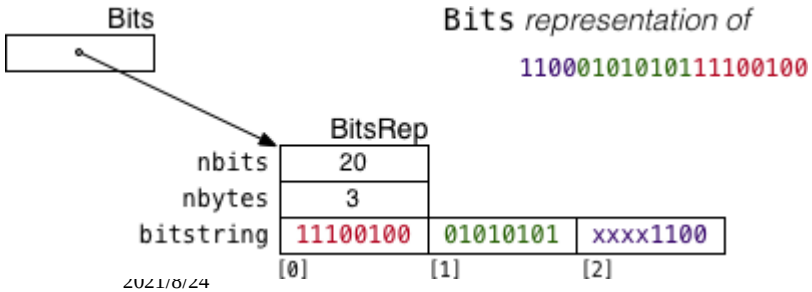
Pages are defined via a `PageRep` structure which contains a counter for the number of items, and then an array of bytes containing the actual items, whether they are tuples or signatures or slices. The size of each type of item is held in the `RelnParams` structure, and so `Page`s are typically considered 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:

`



2021/8/24

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

This project 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.

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.

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

`

Testing

The following simple tests provide a sanity-check for the code.


```
# make a file to hold 10000 6-attribute tuples
$ ./create R simc 10000 6 1000
# make some data, and save it in a file
$ ./gendata 10000 6 1234567 13 > R.in
# load tuples from R.in into files of relation R
$ ./insert R < R.in
# check the structure of R's files
$ ./stats R
Global Info:
Dynamic:
  #items:  tuples: 10000  tsigs: 10000  psigs: 137  bsigs: 6304
  #pages:  tuples: 137  tsigs: 27  psigs: 28  bsigs: 28
```

```
Static:
  tups   #attrs: 6  size: 56 bytes  max/page: 73
  sigs   bits/attr: 9
  tsigs  size: 88 bits (11 bytes)  max/page: 372
  psigs  size: 6304 bits (788 bytes)  max/page: 5
  bsigs  size: 144 bits (18 bytes)  max/page: 227
# linear scan of all data (i.e. run an open query; ignore signatures)
# if you want to see alll 10000 tuples, don't pipe through tail
$ ./select R '?,?,?,?,?,?' x | tail -6
# search for a tuple by the first attribute (not using signatures)
$ ./select R '1234999,?,?,?,?,?' x
1234999,UEkVEljYuGrloQCzLjmw,a3-183,a4-100,a5-017,a6-432
Query Stats:
# sig pages read:    0
# signatures read:   0
# data pages read:   137
# tuples examined:   10000
# false match pages: 136
# search for a tuple by the first attribute (using tuple signatures)
$ ./select R '1234999,?,?,?,?,?' t
1234999,UEkVEljYuGrloQCzLjmw,a3-183,a4-100,a5-017,a6-432
Query Stats:
# sig pages read:    27
# signatures read:   10000
# data pages read:   5
# tuples examined:   365
# false match pages: 4
# search for a tuple by first attribute (using page signatures)
$ ./select R '1234999,?,?,?,?,?' p
1234999,UEkVEljYuGrloQCzLjmw,a3-183,a4-100,a5-017,a6-432
Query Stats:
# sig pages read:    28
# signatures read:   137
# data pages read:   1
# tuples examined:   73
# false match pages: 0
# search for a tuple by first attribute (using bit-sliced signatures)
$ ./select R '1234999,?,?,?,?,?' b
1234999,UEkVEljYuGrloQCzLjmw,a3-183,a4-100,a5-017,a6-432
Query Stats:
# sig pages read:    7
# signatures read:   9
```

`

```
# data pages read:   1
# tuples examined:   73
# false match pages: 0
# check for expeected answers
$ grep 'a3-241,a4-158,a5-407' R.in
1237049,ovnsbtUWihCcCEoRWKcF,a3-241,a4-158,a5-407,a6-490
1242029,eptevNjxFwayfSGeFKrO,a3-241,a4-158,a5-407,a6-490
# search for tuples via several attributes (using no signatures)
$ ./select R '?,?,a3-241,a4-158,a5-407,?' x
1237049,ovnsbtUWihCcCEoRWKcF,a3-241,a4-158,a5-407,a6-490
1242029,eptevNjxFwayfSGeFKrO,a3-241,a4-158,a5-407,a6-490
Query Stats:
# sig pages read:    0
# signatures read:   0
# data pages read:   137
# tuples examined:   10000
# false match pages: 135
# search for tuples via several attributes (using tuple signatures)
$ ./select R '?,?,a3-241,a4-158,a5-407,?' t
1237049,ovnsbtUWihCcCEoRWKcF,a3-241,a4-158,a5-407,a6-490
1242029,eptevNjxFwayfSGeFKrO,a3-241,a4-158,a5-407,a6-490
Query Stats:
# sig pages read:    27
# signatures read:   10000
# data pages read:   2
# tuples examined:   146
# false match pages: 0
# search for tuples via several attributes (using page signatures)
$ ./select R '?,?,a3-241,a4-158,a5-407,?' p
1237049,ovnsbtUWihCcCEoRWKcF,a3-241,a4-158,a5-407,a6-490
1242029,eptevNjxFwayfSGeFKrO,a3-241,a4-158,a5-407,a6-490
Query Stats:
# sig pages read:    28
# signatures read:   137
# data pages read:   2
# tuples examined:   146
# false match pages: 0
# search for tuples via several attributes (using bit-sliced signatures)
$ ./select R '?,?,a3-241,a4-158,a5-407,?' b
1237049,ovnsbtUWihCcCEoRWKcF,a3-241,a4-158,a5-407,a6-490
1242029,eptevNjxFwayfSGeFKrO,a3-241,a4-158,a5-407,a6-490
Query Stats:
```

`

```
# sig pages read:    17
# signatures read:   27
# data pages read:   2
# tuples examined:   146
# false match pages: 0
# etc etc etc etc etc (think of more tests)
```

Based on the above, you should be able to devise other tests to check whether your `select` is producing the correct answers, and whether it's producing the same number of signatures reads and signature page reads. If it reads extra data pages using the same false match probability, that's not catastrophic.

The above methods, you should consider the sum of the page reads (both signature and data pages). The bestmethod is one that minimises this cost (e.g. read less signature pages, but read no more data pages). You can tune the search methods by changing the false match probability; a higher false match probability produces smaller signatures but results in more false-match pages being read.