Kenny Yu
HUID: 30798260
CS165, Spring 2014
Final Project Writeup

## Overall Notes

- I worked alone on all the projects.
- My codebase: https://code.seas.harvard.edu/kennyyu-cs165/kennyyu-cs165
- My tests: https://code.seas.harvard.edu/~kennyyu/cs165-spring-2014/kennyyus-cs165-tests
- My blog post about error handling (I learned when working on the projects for this class): http://kennyyu.me/blog/2014/03/19/c-error-handling/

## Testing My System

To test my system, I have several kinds of tests:

- unit tests in the **tests/** directory - these are small unit tests to test my parser, searching, and list implementations

- **p2tests/, p3tests/, p4tests/** - these contain end-to-end unit tests to test the results of queries with expected outputs
    - You can run these with the ./test.sh script (see README)
    - ./test.sh p2tests
        - will find all the query plan files (*.txt), sort them by file name, execute them one by one, collect the standard out, and compare them against the expected output. If the output is in a different order, my script will sort the actual and expected output to check if lines are in different order
- **p3challenge/, p4challenge/** - challenge tests
- **perftests/** - these are performance tests to compare my database joins against mysql joins. See below for results.

## Join Performance

I tested the performance of loading the data from the CSV and then executing the following SQL queries (and its equivalent query plans):

SQL:

```
DROP TABLE IF EXISTS r;

CREATE TABLE r (
ra INT NOT NULL,
rc INT NOT NULL,
rd INT NOT NULL);

CREATE INDEX rc_index ON r (rc) USING BTREE;

DROP TABLE IF EXISTS s;

CREATE TABLE s (
sa INT NOT NULL,
sf INT NOT NULL,
sg INT NOT NULL);

CREATE INDEX sf_index ON s (sf) USING BTREE;
CREATE INDEX sa_index ON s (sa) USING BTREE;

LOAD DATA LOCAL INFILE 'r.csv' INTO TABLE r
FIELDS TERMINATED BY ',' ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 LINES
(ra, rc, rd);

LOAD DATA LOCAL INFILE 's.csv' INTO TABLE s
FIELDS TERMINATED BY ',' ENCLOSED BY '"'
```

```
LINES TERMINATED BY '\n'
IGNORE 1 LINES
(sa, sf, sg);

SELECT COUNT(r.rc)
FROM r
WHERE r.rc >= 1 and r.rc <= 9;

SELECT COUNT(s.sf)
FROM s
WHERE s.sf >= 31 and s.sf <= 99;

SELECT MAX(r.rd), MIN(s.sg), COUNT(r.rd), COUNT(s.sg)
FROM r,s
WHERE r.ra = s.sa
    AND r.rc >= 1
    AND r.rc <= 9
    AND s.sf >= 31
    AND s.sf <= 99;
```

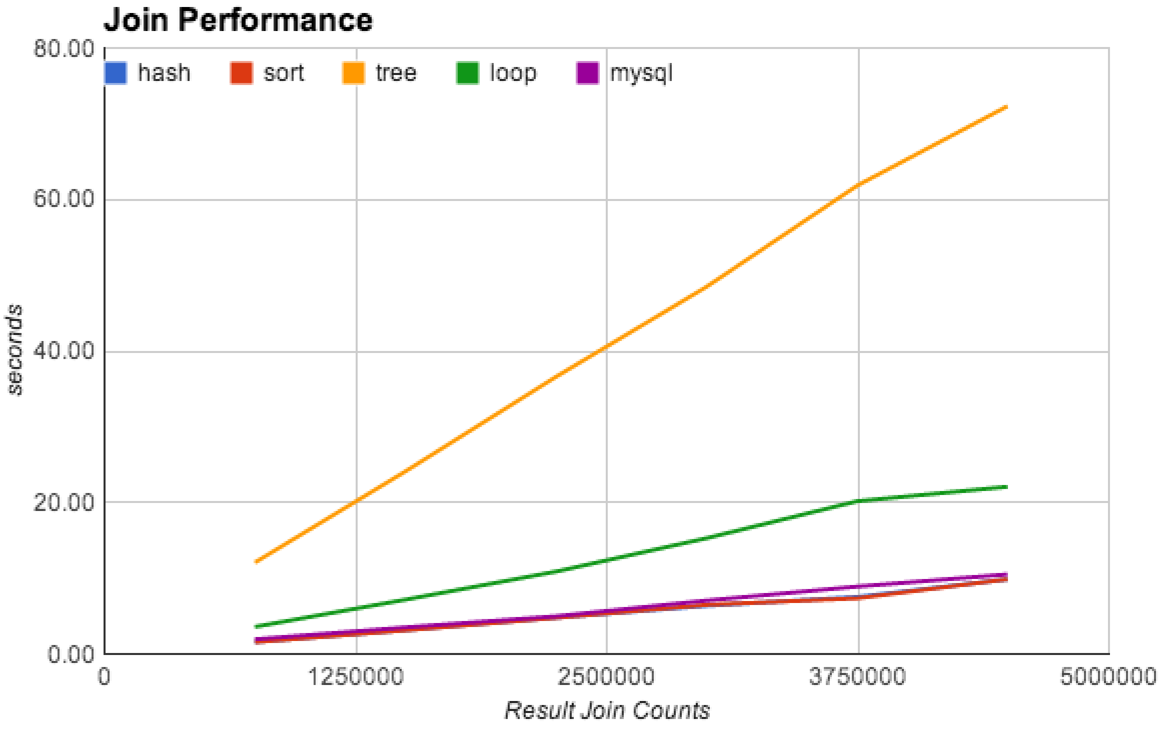
## QUERY PLAN:

```
create(rc,"b+tree")
create(ra,"unsorted")
create(rd,"unsorted")
load("r.csv")
create(sf,"b+tree")
create(sa,"b+tree")
create(sg,"unsorted")
load("s.csv")
rc_inter=select(rc,1,9)
sf_inter=select(sf,31,99)
join_input1=fetch(ra,rc_inter)
join_input2=fetch(sa,sf_inter)
count(join_input1)
count(join_input2)
r_results,s_results={hash,sort,tree,loop}join(join_input1,join_input2)
rd_values=fetch(rd,r_results)
sg_values=fetch(sg,s_results)
maxr=max(rd_values)
mins=min(sg_values)
cr=count(rd_values)
cs=count(sg_values)
tuple(maxr,mins,cr,cs)
```

See the perftests directory for the python script gen.py to generate the csv data and scripts to run the performance tests. Here are the parameters I used:

- "selrater": 0.75, [selection rate on column rc]
- "numr": multiples of 100,000 [number of rows in r]
- "nums": 10000, [number of rows in s]
- "selrates": 1.0, [selection rate on column sf, 100% to allow treejoins]
- "seed": 42,
- "amax": 1000, [number of different keys for ra = sa to join on]

| **result join count** | **hash** | **sort** | **tree** | **loop** | **mysql** |
|---:|---:|---:|---:|---:|---:|
| 748374 | 1.58 | 1.57 | 12.07 | 3.59 | 1.94 |
| 1496583 | 3.15 | 3.12 | 24.09 | 7.17 | 3.51 |
| 2248328 | 4.80 | 4.78 | 36.61 | 10.88 | 4.98 |
| 2997238 | 6.37 | 6.49 | 48.48 | 15.29 | 7.06 |
| 3747569 | 7.53 | 7.36 | 61.85 | 20.17 | 8.90 |
| 4495205 | 9.86 | 9.88 | 72.32 | 22.08 | 10.51 |



In my database:

- sort and hash joins perform very similar to one another (when all the data fit in memory)
- tree joins are slow, presumably from all the IO when probing the btree
- loop joins are about twice as slow as sort and hash joins
- sort, hash perform just as well as mysql

- If I do NOT build an index on s.sa in mysql, the time for the queries take on the other of MINUTES instead of seconds.

### Limitations

- I allow tree joins only on selections that select an entire column with a btree index.
- I can only support data that fit in memory

## Extending My System

If I had another month to work on my system, I would extend my project with the following:

- ***update/delete for sorted and btree columns***
- ***buffer pool*** - currently, whenever I read/write to a file, I allocate a buffer on the stack. This does not work well with recursion (e.g. btree insert).
    - I would extend my file_* interface to read/write to a buffer and pass a pointer to the buffer back, instead of forcing the caller to pass a pointer to an already existing buffer
- ***mmap files*** - this would be an optimization to avoid read/write system calls, and to quickly read/write to a file by mapping a file into my process's address space
- ***lazy select***

```
void cid_iter_init(struct cid_iterator *iter, struct column_ids *cids);
// advances the iterator position
bool cid_iter_has_next(struct cid_iterator *iter);
uint64_t cid_iter_get(struct cid_iterator *iter);
void cid_iter_cleanup(struct cid_iterator *iter);
```

- 
    - I currently already have an iterator interface for select intermediates (I represent ids using a bitmap after a select, and an explicit array of ids after a join). However, for columns with sorted and btree indices, it would be better to simply represent a select intermediate with two numbers (upper and lower bounds), and then only lazily materialize the IDs when needed.
- ***vectorized processing***
    - extend my iterator interface so that get() would return a chunk of IDs as opposed to a single ID

```
// Materialize and return a chunk of IDs as opposed to a single ID
void cid_iter_get(struct cid_iterator *iter,
                  uint64_t **retids,
                  unsigned *retn);
```

- ***use 2 phase locking to enable proper support for transactions***

    - currently, all of my columns act as monitors and all public API functions perform locking underneath the interface. However, I currently do not support atomicity of multiple column operations (e.g., the entirety of a select and the entirety of a fetch are each atomic, but there is no support for atomic select-then-fetch).
    - instead of hiding the synchronization underneath the column interface, I will expose the lock to enable 2 phase locking
    - how
        - the server will wait until it receives all the queries that belong to a single transaction
        - the server will determine all the resources that need to be acquired for the transaction, and then perform 2 phase locking and hold them all until a commit or abort.
- ***try to implement cracking***
    - support a new column type "crack" and "crack-follow"
        - crack columns will be cracked, and will act as heads of sideways cracking pairs
        - crack-follow columns will act as tails of sideways cracking pairs

# Lessons Learned

## What was difficult

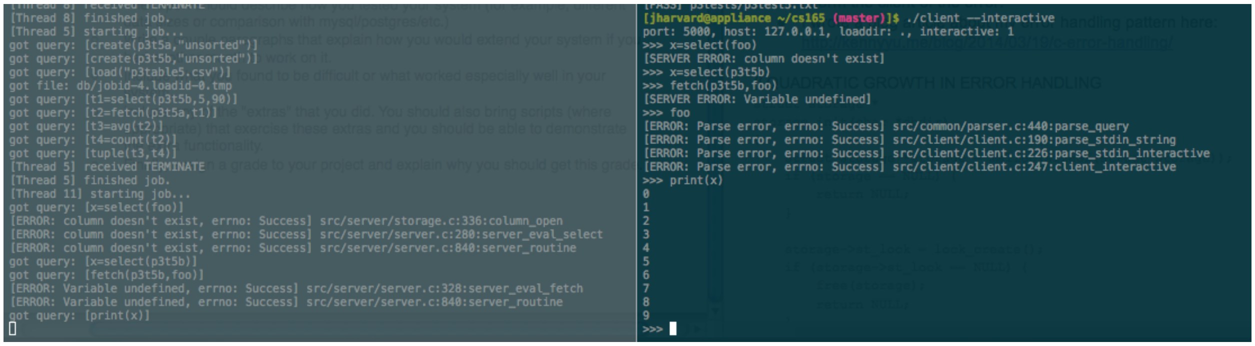
- Building all the glue for an end-to-end database was difficult and took away focus from the actual database work. For example, I spent much of my time building a robust database with:
    - proper error handling library
    - an rpc library
    - command line editing library
    - a threadpool implementation

  and this took time away from optimizing database.
- Debugging btree code was difficult, especially when I tested on tables with millions of rows.
- Designing all the on-disk data structures was challenging, especially to make data persistent on system restart, packing the appropriate data and metadata onto a page, and

## What worked well

### Robust Error Handling

- I wrote custom macros to handle errors properly and cleanup. Every time an error occurred (e.g., selecting on a column that doesn't exist, attempting to use a variable that was not defined, parser error, out of memory, etc.), I gracefully cleanup all resources and state changes, and I generate a stack trace on the server side, and then serialize the error back to the client to inform the client of the error. Screenshot:



```
[Thread 8] received TERMINATE
[Thread 8] finished job.
[Thread 5] starting job...
got query: [create(p3t5a,"unsorted")]
got query: [create(p3t5b,"unsorted")]
got query: [load("p3table5.csv")]
got file: db/jobid-4.loadid-0.tmp
got query: [t1=select(p3t5b,5,90)]
got query: [t2=fetch(p3t5a,t1)]
got query: [t3=avg(t2)]
got query: [t4=count(t2)]
got query: [tuple(t3,t4)]
[Thread 5] received TERMINATE
[Thread 5] finished job.
[Thread 11] starting job...
got query: [x=select(foo)]
[ERROR: column doesn't exist, errno: Success] src/server/storage.c:336:column_open
[ERROR: column doesn't exist, errno: Success] src/server/server.c:280:server_eval_select
[ERROR: column doesn't exist, errno: Success] src/server/server.c:840:server_routine
got query: [x=select(p3t5b)]
got query: [fetch(p3t5b,foo)]
[ERROR: Variable undefined, errno: Success] src/server/server.c:328:server_eval_fetch
[ERROR: Variable undefined, errno: Success] src/server/server.c:840:server_routine
got query: [print(x)]
```

```
[PASS] p3tests/p3test5.txt
[jharvard@appliance ~/cs165 (master)]$ ./client --interactive
port: 5000, host: 127.0.0.1, loaddir: ., interactive: 1
>>> x=select(foo)
[SERVER ERROR: column doesn't exist]
>>> x=select(p3t5b)
>>> fetch(p3t5b,foo)
[SERVER ERROR: Variable undefined]
>>> foo
[ERROR: Parse error, errno: Success] src/common/parser.c:440:parse_query
[ERROR: Parse error, errno: Success] src/client/client.c:190:parse_stdin_string
[ERROR: Parse error, errno: Success] src/client/client.c:226:parse_stdin_interactive
[ERROR: Parse error, errno: Success] src/client/client.c:247:client_interactive
>>> print(x)
0
1
2
3
4
5
6
7
8
9
>>>
```

- I wrote a blog article about my error handling pattern here: [http://kennyyu.me/blog/2014/03/19/c-error-handling/](http://kennyyu.me/blog/2014/03/19/c-error-handling/)

// QUADRATIC GROWTH IN ERROR HANDLING

```
struct storage *
storage_init(char *dbdir)
{
    struct storage *storage = malloc(sizeof(struct storage));
    if (storage == NULL) {
        return NULL;
    }

    storage->st_lock = lock_create();
    if (storage->st_lock == NULL) {
        free(storage);
        return NULL;
    }

    storage->st_open_cols = columnarray_create();
    if (storage->st_open_cols == NULL) {
        lock_destroy(storage->st_lock);
        free(storage);
        return NULL;
    }

    int result = mkdir(dbdir, S_IRWXU);
    if (result) {
        columnarray_destroy(storage->st_open_cols);
        lock_destroy(storage->st_lock);
        free(storage);
        return NULL;
    }
    strcpy(storage->st_dbdir, dbdir);
    char buf[128];
    sprintf(buf, "%s/%s", dbdir, METADATA_FILENAME);
    storage->st_file = file_open(buf);
    if (storage->st_file == NULL) {
        assert(rmdir(dbdir) == 0);
        columnarray_destroy(storage->st_open_cols);
        lock_destroy(storage->st_lock);
        free(storage);
        return NULL;
    }

    return storage;
}
```

// LINEAR GROWTH IN ERROR HANDLING

```
struct storage *
storage_init(char *dbdir)
{
    struct storage *storage = malloc(sizeof(struct storage));
    if (storage == NULL) {
        goto done;
    }

    storage->st_lock = lock_create();
    if (storage->st_lock == NULL) {
        goto cleanup_malloc;
    }

    storage->st_open_cols = columnarray_create();
    if (storage->st_open_cols == NULL) {
        goto cleanup_lock;
    }

    int result = mkdir(dbdir, S_IRWXU);
    if (result) {
        goto cleanup_colarray;
    }

    strcpy(storage->st_dbdir, dbdir);
    char buf[128];
    sprintf(buf, "%s/%s", dbdir, METADATA_FILENAME);
    storage->st_file = file_open(buf);
    if (storage->st_file == NULL) {
        goto cleanup_mkdir;
    }
    // success
    result = 0;
    goto done;

  cleanup_mkdir:
    assert(rmdir(dbdir) == 0);
  cleanup_colarray:
    columnarray_destroy(storage->st_open_cols);
  cleanup_lock:
    lock_destroy(storage->st_lock);
  cleanup_malloc:
    free(storage);
    storage = NULL;
  done:
    return storage;
}
```

// USING MY TRY PATTERN AND GENERATING A STACK TRACE

```
#ifndef _DBERROR_H_
#define _DBERROR_H_

enum dberror {
    DBSUCCESS = 0,
    DBENOMEM,
    ... // other errors
};

const char *dberror_string(enum dberror result);

void dberror_log(char *msg, const char *file,
                 int line, const char *func);

#define DBLOG(result) \
        dberror_log((char *) dberror_string((result)), \
                     __FILE__, __LINE__, __func__);

#endif

#ifndef _TRY_H_
#define _TRY_H_

#include <stddef.h>
#include "dberror.h"

#define TRY(result, expr, cleanup) \
    (result) = (expr); \
    if ((result)) { \
        DBLOG((result)); \
        goto cleanup; \
    }

#define TRYNULL(result, err, var, expr, cleanup) \
    (var) = (expr); \
    if ((var) == NULL) { \
        (result) = (err); \
        DBLOG((result)); \
        goto cleanup; \
    }

#endif

struct storage *
storage_init(char *dbdir)
{
    int result;
    struct storage *storage;

    TRYNULL(result, DBENOMEM, storage, malloc(sizeof(struct storage)), done);
    TRYNULL(result, DBENOMEM, storage->st_lock, lock_create(),
cleanup_malloc);
    TRYNULL(result, DBENOMEM, storage->st_open_cols, columnarray_create(),
cleanup_lock);
    TRY(result, mkdir(dbdir, S_IRWXU), cleanup_colarray);

    strcpy(storage->st_dbdir, dbdir);
    char buf[128];
    sprintf(buf, "%s/%s", dbdir, METADATA_FILENAME);
    TRYNULL(result, DBEIONOFILE, storage->st_file, file_open(buf),
cleanup_mkdir);

    // success
    result = 0;
    goto done;

  cleanup_mkdir:
    assert(rmdir(dbdir) == 0);
  cleanup_colarray:
    columnarray_destroy(storage->st_open_cols);
  cleanup_lock:
    lock_destroy(storage->st_lock);
  cleanup_malloc:
    free(storage);
    storage = NULL;
  done:
    return storage;
}
```

## Defensive programming

I was very liberal in my use of asserts to check all my invariants and preconditions and postconditions. As a result, it was very easy and fast to track down bugs in my system and debug them.

## File API

```
// these will alloc/free the in memory data structures
struct file *file_open(char *name);
void file_close(struct file *f);

// these will create/delete the on disk data structures
int file_alloc_page(struct file *f, page_t *retpage);
void file_free_page(struct file *f, page_t page);
bool file_page_isalloc(struct file *f, page_t page);

// returns the total number of pages in the file, alloc'ed or freed
page_t file_num_pages(struct file *f);
int file_read(struct file *f, page_t page, void *buf);
int file_write(struct file *f, page_t page, void *buf);
```

My file API allowed me to think of files at a much higher abstraction than worrying about file descriptors and byte offsets. I can only do IO at the page level and not the byte level. Also, I use the first few pages of a file for a bitmap to keep track of which pages have been allocated in the file. ***This allows me to reclaim and reuse pages within a file for deletions and inserts***.

Furthermore, since I already have a file abstraction, this makes it easier to extend my system later with a buffer pool abstraction by slightly changing the parameters to file_read/write:

```
struct buf;
// request a buffer and return a pointer to it
int file_read(struct file *f, page_t page, struct buf **retbuf);
int file_write(struct file *f, page_t page, struct buf *buf);
```

## Sorted Merge Join is as fast as a Static Hash Join

My sorted merge joins are as fast as my static hash joins. I have several theories on why this might be true:

- after sorting, the rest of the join is essentially sequential access of main memory (except when handling duplicates)
- for static hashing, the second phase (after computing the counts in the first phase) is essentially random access into memory, even though the hash table is one big sequential array. Furthermore, the probing phase is also random access of main memory. This leads to TLB misses and possibly page faults if the operating system has decided to swap out some of the pages containing the hash table.

### Range Queries on Sorted Columns and Btrees

I implemented my binary search to always return the **lower bound** into the list on where I would insert the target value if I were to insert the value into the list.

```
unsigned
binary_search(void *x, void *vals, unsigned nvals, size_t size,
              int (*compare)(const void *a, const void *b))
{
    unsigned l = 0;
    unsigned r = nvals;
    while (l < r) {
        unsigned m = l + (r - l) / 2;
        void *y = vals + size * m;
        if (compare(y, x) < 0) {
            l = m + 1;
        } else {
            r = m;
        }
    }
    return l;
}
```

This made selections on sorted columns and btrees extremely easy: I would make all select queries into half-open-half-closed range queries (select x ⇒ search for [x,x + 1); select all ⇒ search for [-inf,+inf); select lo,hi ⇒ search for [lo, hi + 1) ) and ***search for the lower bound of low and for the lower bound of high + 1.***

Once I found the lower bound positions of (low) and (high + 1) in my sorted projection or btree index, I can blindly take all values in inclusively from low up to but not including high + 1.

# Extras

## Concurrent queries

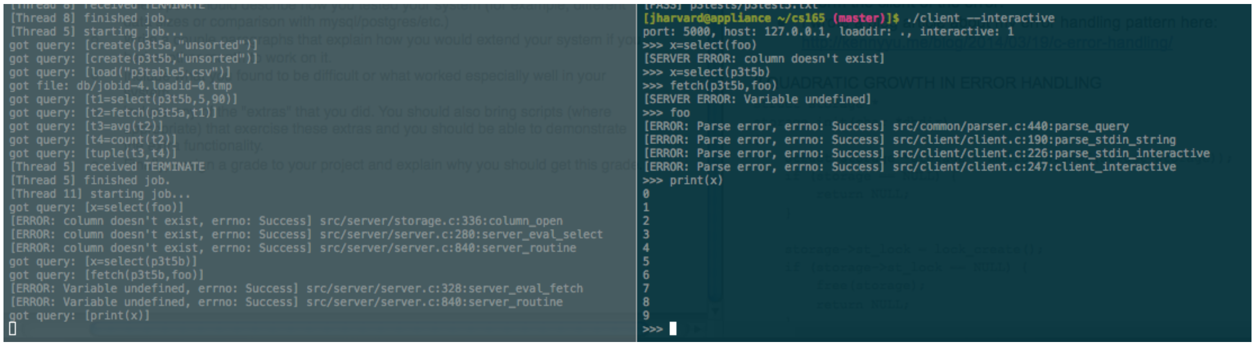
```
// create the directory if it doesn't exist, and init metadata file
struct storage *storage_init(char *dbdir);
void storage_close(struct storage *storage);
int storage_add_column(struct storage *storage, char *colname,
                       enum storage_type stype);

// if not in array, add it and inc ref count
int column_open(struct storage *storage, char *colname, struct column
**retcol);

// dec ref count, close it if 0
void column_close(struct column *col);

int column_insert(struct column *col, int val);
int column_update(struct column *col, struct column_ids *ids, int val);
int column_delete(struct column *col, struct column_ids *ids);
int column_load(struct column *col, int *vals, uint64_t num);
struct column_ids *column_select(struct column *col, struct op *op);
struct column_vals *column_fetch(struct column *col, struct column_ids *ids);
```

All of these column operations in my public interface are all atomic (the column implements synchronization like a monitor: the synchronization is hidden beneath the interface).

- ***As a result, I can support multiple concurrent queries on multiple columns, as long as there are no mutations in between a select and a fetch.***
  - I use ***reader-writer locks*** on columns.
- As a result, each insert,update,delete,load,select,fetch operation is singly atomic on a single column, but there is no way to make multiple operations on a single column or operations on multiple columns atomic.

## Robust Error Handling

See the “error handling” section above in what worked especially well for me.
Screenshot:



```
[Thread 8] received TERMINATE
[Thread 8] finished job.
[Thread 5] starting job...
got query: [create(p3t5a,"unsorted")]
got query: [create(p3t5b,"unsorted")]
got query: [load("p3table5.csv")]
got file: db/jobid-4.loadid-0.tmp
got query: [t1=select(p3t5b,5,90)]
got query: [t2=fetch(p3t5a,t1)]
got query: [t3=avg(t2)]
got query: [t4=count(t2)]
got query: [tuple(t3,t4)]
[Thread 5] received TERMINATE
[Thread 5] finished job.
[Thread 11] starting job...
got query: [x=select(foo)]
[ERROR: column doesn't exist, errno: Success] src/server/storage.c:336:column_open
[ERROR: column doesn't exist, errno: Success] src/server/server.c:280:server_eval_select
[ERROR: column doesn't exist, errno: Success] src/server/server.c:840:server_routine
got query: [x=select(p3t5b)]
got query: [fetch(p3t5b,foo)]
[ERROR: Variable undefined, errno: Success] src/server/server.c:328:server_eval_fetch
[ERROR: Variable undefined, errno: Success] src/server/server.c:840:server_routine
got query: [print(x)]
▯
```

```
[PASS] p3tests/p3test5.txt
[jharvard@appliance ~/cs165 (master)]$ ./client --interactive
port: 5000, host: 127.0.0.1, loaddir: ., interactive: 1
>>> x=select(foo)
[SERVER ERROR: column doesn't exist]
>>> x=select(p3t5b)
>>> fetch(p3t5b,foo)
[SERVER ERROR: Variable undefined]
>>> foo
[ERROR: Parse error, errno: Success] src/common/parser.c:440:parse_query
[ERROR: Parse error, errno: Success] src/client/client.c:190:parse_stdin_string
[ERROR: Parse error, errno: Success] src/client/client.c:226:parse_stdin_interactive
[ERROR: Parse error, errno: Success] src/client/client.c:247:client_interactive
>>> print(x)
0
1
2
3
4
5
6
7
8
9
>>> █
```

I wrote a blog article about my error handling pattern here:
http://kennyyu.me/blog/2014/03/19/c-error-handling/

### Performance Testing Against MySQL

See my join performance above. I beat mysql for 1000000+ rows.

### Cache-Conscious Code

I made sure that whenever I did operations that required reading the base data, I did so sequentially to minimize the number of buffer misses and disk IO:

- fetching after join: I sort the IDs, fetch sequentially, then resort the rows to maintain row integrity
- perform updates and deletions sequentially on the IDs

## Grade for Myself

Grade for myself: A

I think I deserve an A for my project because:

- I built a ***robust end-to-end system*** with:
  - ***robust error handling*** (stack trace, fail gracefully, and error propagation)
  - ***well-designed abstractions and reusable interfaces*** to modularize the different components of my system:
    - rpc library
    - threadpool library
    - column interface
    - storage interface
    - server session interface

        - error handling library
        - query parsing library
        - csv parsing library
- My ***code is extendible and clearly commented***
    - I would even advocate for using parts of my codebase as some distribution code for next year’s offering of the class so that students can focus on the actual database work and not on all the glue code for building a robust end-to-end system.
- ***extensive unit testing and performance testing***
    - See my testing section above
- ***designed for concurrency from the very start of the project***
    - From the very beginning of p2, I designed my system to support concurrent queries and designed all my interfaces to support this (rather than bolting it on at the end)