Project report

The project represents a simple implementation of a in-memory column-store database with persistence capabilities.

Correctness: all tests except for 32 and 33 (deletes and updates)

Lacking functionality:
- B-trees are not persisted
- Shared scans are not implemented
- Deletes are not supported
- Crashes on Linux machines (has been tested on mac: mostly likely due to sockets issues)

Each column is represented by metadata and vector struct:

```
struct column {
    char *name;
    struct table *table;
    bool clustered;
    enum col_status status;
    column_index *index;
    struct vec data;
};
```

Vector itself is a dynamic array of its that shrinks and expands by the number of items

```
struct vec {
    size_t sz;
    size_t capacity;
    int *vals;
};
```

Database supports the following operations:
- point select in a column
- range select in a column
- range select in a result
- project a column or a result
- relational insert
- finding min, max and average values in columns or results
- simple hash joins
- bulk loading from file
- adding and subtracting two vector results
- creating database, tables and columns

Database and some indices are saved on disk upon shutdown command. Lazy loading has been added so that only columns and indices that are required by queries are loaded into memory.

Indexing: clustered index; secondary indexes on unsorted columns and B+ indexes on any columns

Clustered index is created upon bulk loading and the order is maintained after insertions

Sorted secondary index is represented by a dynamic array of pairs sorted by val; supports insertions. Allows fast binary search ranges, but is expensive to maintain.

```
struct sindex {
   int val;
   unsigned pos;
};
```

B+ tree index of fanout ~33 allows fast insertions and range searches only somewhat slower than binary search (experiments showed a slowdown of 1.56 compared to binary search).

```
struct btnode {
   enum btnode_type ntype;
   short ksz;
   int keys[MAXKEYS];
   bset idref[FANOUTSET];  // only used in leafs to distinguish between id and ids
   union value {
         struct btnode *child;
         int id;
         struct vec *ids;
   } values[FANOUT];
};
```

Shared scans (unimplemented): one possible way to implement a shared scan would be to collect up to N select queries within a certain timeframe, detect the M queries that need to scan the same column, divide the column into L2 cache sizes and perform multiple scans over the same chunk of the column by a single core (~ thread). Even though each chunk will be scanned M times, it will remain in L2 cache after the first scan such that subsequent scans will not have cache misses. In the end, the results need to be merged

Joins:

- Block-nested loop joins are supported, but are slow.
- Simple hash join with two passes (one to determine the size of the hash table and the second to perform a join) runs much faster. Array of pointers to buckets that contain pairs of (val,pos).
- Grace hash join support can be added as an cache-conscious optimization of the current hash join implementation.

Inserts: relational inserts into unclustered and clustered tables and all indices.
Deletes are not supported at the moment
Updates can be modeled as inserts after deletes with support of additional data structures that track recent positions that have been deleted.