

1. First Build

- First of all, I watched the instructor's demo on how to build sqlite3 from source code. Then, I followed his instruction and intended to do the project on my Mac environment without any virtual machine.
- I downloaded the source code from the website shown in the demo video and extracted it. Next, in terminal, I ran the command “sudo apt install gcc” but met a problem. The terminal said “Unable to locate an executable at /...”, which confused me. I was not sure whether my Mac has installed gcc before or not but I decided not to fix this problem but switch to my virtual machine to do this project instead.
- I have VMware Fusion running Linux on my Mac. I installed and have been using it since the beginning of this semester for my CS 149 (Operating Systems) so I am fairly familiar with it. Plus, I already installed gcc on the Linux so one less step to do. I continued to install tcl and started building the first build. Since my VM is quite slow, I used the native Mac environment to look and modify the source code and build the result code in my VM.
- Just like in the demo, I created the first build in the a folder on Desktop, named “bld0”. I ran “configure” from sqlite folder and “make” in the bld0 folder. I did not change any source code and the purpose of this first build is to just run the original version of sqlite3 on my VM.

2. Second Build

- After the original version of sqlite3 has run successfully on my VM, I started to actually do the project. Initially, I intended to research online to find out where to start, but I decided to take a look at the “src” folder of sqlite first for a warmup. In there, I found lots of C files and I just happened to see the insert.c file. I started working on it from that point to the end of the project without looking at any online resource.
- After opening insert.c by Geany, I was overwhelmed by the length of the file and how much stuff in it. Luckily, on the side bar, Geany showed a list of all functions, global variables, structs, and macros in the file. Luckily again, I saw the sqlite3Insert function in that list and checked on it. It is fair to say if I used a different app to open insert.c, I would have had much more difficult doing this project. Geany allowed me to finish the project in one day.
- After finding sqlite3Insert(), I read the first few lines of the function to check out its local variables. Although not understanding all these variables since it was the first time I had contact with sqlite3 source code, several ones caught my attention thanks to the author's comment:

```
SrcList *pTabList, /* Name of table into which we are inserting */  
Table *pTab;        /* The table to insert into. aka TABLE */  
int nColumn;       /* Number of columns in the data */
```

```
int nHidden = 0; /* Number of hidden columns if TABLE is virtual */
```

I didn't use all these variables at the end but they gave me a sense of what I need to do in this project.

- After the local variables, I did not focus on understanding every line of code but I skimmed through the whole function to see if my eyes could catch anything since it is a very long function! Luckily again, I saw in lines 939 - 941 the error message which is generated when we insert a tuple with different number of columns than the table:

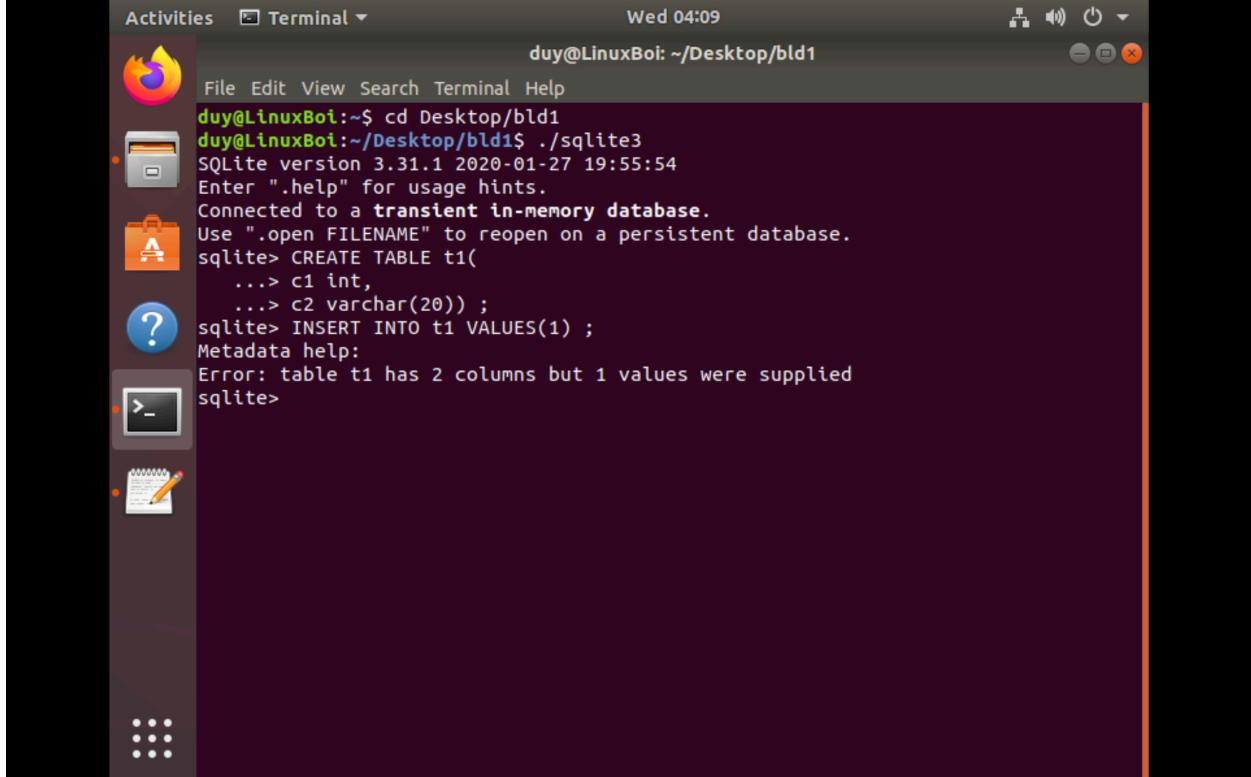
```
sqlite3ErrorMsg(pParse,
    "table %S has %d columns but %d values were supplied",
    pTabList, 0, pTab->nCol-nHidden, nColumn);
```

This is the error that should be generated along with our addition error message, so I knew that I have found the location where I should enter my code.

- I decided to test the first modification by adding a simple additional output: the string “Metadata help: ” before the sqlite error message. I added a simple output function as below:

```
puts("Metadata help: ");
```

Then I configured and built a new version of sqlite3. Below was the result of my second build, which showed that I was going in the right direction.



```
Activities Terminal Wed 04:09
duy@LinuxBoi: ~/Desktop/bld1
File Edit View Search Terminal Help
duy@LinuxBoi:~$ cd Desktop/bld1
duy@LinuxBoi:~/Desktop/bld1$ ./sqlite3
SQLite version 3.31.1 2020-01-27 19:55:54
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> CREATE TABLE t1(
    ...> c1 int,
    ...> c2 varchar(20)) ;
sqlite> INSERT INTO t1 VALUES(1) ;
Metadata help:
Error: table t1 has 2 columns but 1 values were supplied
sqlite>
```

3. Third (Final) Build

- With the success of the second build, I determined the remaining tasks is to find out how to print out the name of the table we are inserting into and the name of the columns of that table.
- I came back to the local variables that caught my attention during the process of the second build. In those, I decided to research about “Table pTab” first because it looks the most promising. In the insert situation in this project, we assume our insert command has the correct table name, so pTab should be initialized already before the point of our code modification.
- I tried to search for the type “Table” to understand more about what pTab has that I can use. However, I failed to find additional information about Table in insert.c. But I found out that insert.c includes sqliteInt.h (the internal header file of sqlite), so I guess Table can be in there. And it is.
- At line 2081 of sqliteInt.h, the content of struct Table is represented:

The screenshot shows a debugger interface with two panes. The left pane is a symbol browser showing various global variables and structures. The right pane is the content of the sqliteInt.h file, specifically focusing on the definition of the Table structure.

```

2073     );
2074
2075     /* Allowed values for VTable.eVtabRisk
2076     */
2077     #define SQLITE_VTABRISK_Low          0
2078     #define SQLITE_VTABRISK_Normal      1
2079     #define SQLITE_VTABRISK_High        2
2080
2081     /**
2082      ** The schema for each SQL table and view is represented in memory
2083      ** by an instance of the following structure.
2084      */
2085     struct Table {
2086         char *zName;           /* Name of the table or view */
2087         Column *aCol;          /* Information about each column */
2088         Index *pIndex;         /* List of SQL indexes on this table. */
2089         Select *pSelect;        /* NULL for tables. Points to definition if a view. */
2090         FKey *pFkey;           /* Linked list of all foreign keys in this table */
2091         char *zCollAff;        /* String defining the affinity of each column */
2092         ExprList *pCheck;       /* All CHECK constraints */
2093         /* ... also used as column name list in a VIEW */
2094         int tnNum;             /* Root BTREE page for this table */
2095         u32 nTabRef;           /* Number of pointers to this Table */
2096         u32 tabFlags;          /* Mask of TF_* values */
2097         i16 iPkKey;            /* If not negative, use aCol[iPkKey] as the rowid */
2098         i16 nCol;               /* Number of columns in this table */
2099         i16 nNvCol;             /* Number of columns that are not VIRTUAL */
2100         i32 nRowLogEst;        /* Estimated rows in table - from sqlite_stat1 table */
2101         LogEst nTabRowLogEst;   /* Estimated size of each table row in bytes */
2102         #ifdef SQLITE_ENABLE_COSTMULT
2103             LogEst costHult;    /* Cost multiplier for using this table */
2104         #endif
2105         u8 keyConf;             /* What to do in case of uniqueness conflict on iPkKey */
2106         #ifndef SQLITE_OMIT_ALTERTABLE
2107             int addColOffset;    /* Offset in CREATE TABLE stmt to add a new column */
2108         #endif
2109         #ifndef SQLITE OMIT_VIRTUALTABLE
2110             i32 nModuleArg;       /* Number of arguments to the module */
2111             char **azModuleArg;  /* 0: module 1: schema 2: vtab name 3...: args */
2112             VTable **pVTable;     /* List of VTable objects. */
2113         #endif
2114             Trigger *pTrigger;   /* List of triggers stored in pSchema */
2115             Schema *pSchema;     /* Schema that contains this table */
2116             Table *pNextZombie;   /* Next on the Parse.pZombieTab list */
2117         };
2118
2119     /**
2120      ** Allowed values for Table.tabFlags.
2121      */
2122     #define TF_000Hidden 0x00000000
2123     /* TF_000Hidden applies to tables or view that have hidden columns that are
2124     ** followed by non-hidden columns. Example: "CREATE VIRTUAL TABLE x USING
2125     ** vtab(a HIDDEN, b)". Since "b" is a non-hidden column but "a" is hidden,
2126     ** the TF_000Hidden attribute would apply in this case. Such tables require
2127     ** special handling during INSERT processing. The "000" means "Out Of Order".
2128     */

```

- There were several attributes of struct Table that seem to be useful:

```

char *zName;           /* Name of the table or view */
Column *aCol;          /* Information about each column */
i16 nCol;               /* Number of columns in this table */

```

- I continued to look for the Column type, which can be found at line 1914. It has:

```
char *zName; /* Name of this column, \000, then the type */
```

```

1900  ** Each SQLite module (virtual table definition) is defined by an
1901  ** instance of the following structure, stored in the sqlite3.aModule
1902  ** hash table.
1903  */
1904  struct Module {
1905      const sqlite3_module *pModule;      /* Callback pointers */
1906      const char *zName;                /* Name passed to create_module() */
1907      int nRefModule;                  /* Number of pointers to this object */
1908      void *pAux;                     /* pAux passed to create_module() */
1909      void (*xDestroy)(void *);        /* Module destructor function */
1910      Table *pEpoTab;                 /* Eponymous table for this module */
1911  };
1912
1913  /* Information about each column of an SQL table is held in an instance
1914  ** of the Column structure, in the Table.aCol[] array.
1915  */
1916  /* Definitions:
1917   *
1918   * "table column index" This is the index of the column in the
1919   * Table.aCol[] array, and also the index of
1920   * the column in the original CREATE TABLE stmt.
1921   *
1922   * "storage column index" This is the index of the column in the
1923   * record BLOB generated by the OP_MakeRecord
1924   * opcode. The storage column index is less than
1925   * or equal to the table column index. It is
1926   * equal if and only if there are no VIRTUAL
1927   * columns to the left.
1928   */
1929
1930  /* Struct Column {
1931      char *zName;                    /* Name of this column, \000, then the type */
1932      Expr *pBlt;                     /* Default value or GENERATED ALWAYS AS value */
1933      char *zColl;                   /* Collating sequence. If NULL, use the default */
1934      u8 notNull;                   /* An OE_code for handling a NOT NULL constraint */
1935      char affinity;                /* One of the SQLITE_AFF... values */
1936      u8 szEst;                     /* Estimated size of value in this column. sizeof(INT)=1 */
1937      u16 colFlags;                 /* Boolean properties. See COLFLAG_defines below */
1938  };
1939
1940  /* Allowed values for Column.colFlags:
1941
1942  #define COLFLAG_PRIMKEY 0x0001 /* Column is part of the primary key */
1943  #define COLFLAG_HIDDEN 0x0002 /* A hidden column in a virtual table */
1944  #define COLFLAG_HASTYPE 0x0004 /* Type name follows column name */
1945  #define COLFLAG_UNIQUE 0x0008 /* Column def contains "UNIQUE" or "PK" */
1946  #define COLFLAG_SORTERREF 0x0010 /* Use sorter-refs with this column */
1947  #define COLFLAG_VIRTUAL 0x0020 /* GENERATED ALWAYS AS ... VIRTUAL */
1948  #define COLFLAG_STORED 0x0040 /* GENERATED ALWAYS AS ... STORED */
1949  #define COLFLAG_NOTAVAIL 0x0080 /* STORED column not yet calculated */
1950  #define COLFLAG_JOURNAL 0x0100 /* Block reversion on GENERATED columns */
1951  #define COLFLAG_GENERATED 0x0050 /* Combo: _STORED, _VIRTUAL */
1952  #define COLFLAG_NOINSERT 0x0062 /* Combo: _HIDDEN, _STORED, _VIRTUAL */
1953
1954 */

```

line: 1930/4912 col: 0 sel: 0 INS TAB mode:LF encoding: UTF-8 filetype:C++ scope: unknown

- So, I figured out a way to access table name and column name:
pTab->zName and pTab->aCol[i].zName, with i as the index of the column
- My algorithm for printing out the schema of the table would be running the for loop n times, n is the number of columns of the table. In each iteration, I will use the index of the loop to access the index of the column. However, in the original error message, I saw that the number of columns the author used was “pTab->nCol-nHidden”. nHidden is “Number of hidden columns if TABLE is virtual” as documented by the author. I did not know what is virtual table, but I followed his notation for the number of columns.
- Finally, my complete code modification is highlighted as below:

```

Functions
    sqlite3OpenTable [26]
    sqlite3IndexAffinityStr [7]
    readsTable [170]
    exprColumnFlagUnion [1]
    sqlite3ComputeGenerate [341]
    autoIncBegin [341]
    sqlite3AutoincrementBegin [451]
    autoIncrStep [451]
    autoIncrEnd [464]
    sqlite3SetAutoincrementEnd [515]
    xferOptimization [515]
    sqlite3Insert [620]
    checkConstraintExprNo [620]
    sqlite3ExpReferencesL [620]
    sqlite3GenerateConstra [620]
    sqlite3CompleteInsertio [620]
    sqlite3OpenTableAndIn [620]
    xferCompatibleIndex [24]
    xferOptimization [251]
    sqlite3SetMakeRecordF [620]
    CKCNSTRNT_COLUMN [620]
    CKCNSTRNT_ROWID [620]
Macros
    autoIncBegin [509]
    autoIncrStep [510]
    isView [711]
    CKCNSTRNT_COLUMN [620]
    CKCNSTRNT_ROWID [620]
Variables
    sqlite3_xeropt_count [2]

Activities Text Editor - insert.c
Tue 23:44
insert.c
~/Desktop/sqlite/src
Save
Word Word
Next
line: 939 / 2867 col: 4 sel: 1
C Tab Width: 8 Ln 947, Col 10 INS

```

- And this is the result of the third (final) build with tested input:

```

Activities Terminal - insert.c
Wed 05:09
duy@LinuxBoi: ~/Desktop/bld2
File Edit View Search Terminal Help
duy@LinuxBoi:~$ cd Desktop/bld2
duy@LinuxBoi:~/Desktop/bld2$ ./sqlite3
SQLite version 3.31.1 2020-01-27 19:55:54
Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> CREATE TABLE t1(
    ....> c1 int,
    ....> c2 varchar(20),
    ....> c3 int,
    ....> c4 int) ;
sqlite> INSERT INTO t1 VALUES(1, 'hello', 2) ;
Metadata help: t1{c1, c2, c3, c4}
Error: table t1 has 4 columns but 3 values were supplied
sqlite> INSERT INTO t1 VALUES(1, 'cs157a', 80, 90, 100) ;
Metadata help: t1{c1, c2, c3, c4}
Error: table t1 has 4 columns but 5 values were supplied
sqlite>

```