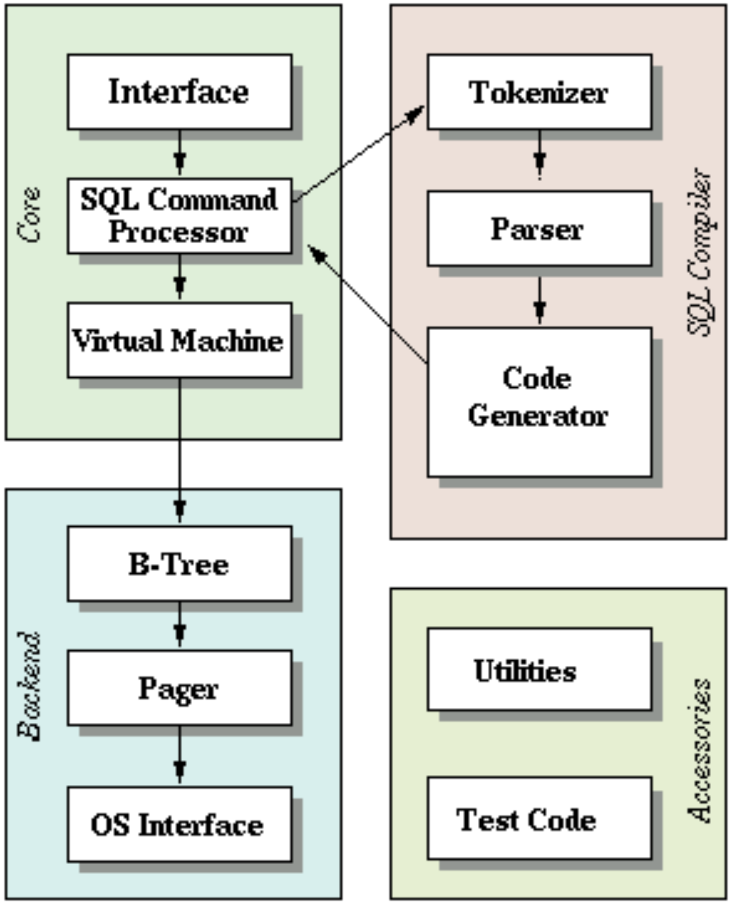
Anatomy of a Database System

Mangesh Bendre
<mangeshbendre@gmail.com>

Block Diagram of SQLite

- Execution of SQL

1. Interface
2. Tokenizer
3. Parser
4. Code Generator for VM
5. Virtual DB Engine
6. B-tree
7. Pager
8. OS interface



1. Interface

- shell.c
- Provides a command line interface to accept the SQL, execute it and display the results.
- When SQLite is used as a library, the application performs duties of an interface by calling the SQLite API
  - sqlite3_open
  - sqlite3_exec
  - sqlite3_close

1. Interface (Example)

- Execute a SQL via the API

```
char *sql = “SELECT * FROM EMPLOYEE;”;
rc = sqlite3_open(“my.db”, &db);
rc = sqlite3_exec(db, sql, callback, 0, &zErrMsg);
sqlite3_close(db);
static int callback(void *NotUsed, int argc, char **argv, char **azColName){
    int i;
    for(i=0; i<argc; i++)
        printf("%s = %s\n", azColName[i], argv[i] ? argv[i] : "NULL");
    return 0;
}
```

2. Tokenizer

- tokenize.c
- Parse the input SQL to tokenize it.
- Tokens defined as TK_ (Tokens defined in Parser used in tokenizer)
- Tokens defined here are available in the parser.

2. Tokenizer (Example)

- The SQL is tokenized as below:

1.	TK_SELECT	SELECT
2.	TK_SPACE	
3.	TK_STAR	*
4.	TK_SPACE	
5.	TK_FROM	FROM
6.	TK_SPACE	
7.	TK_ID	EMPLOYEE
8.	TK_SEMI	;

3. Parser

- parse.y
- LEMON parser – LALR (Look Ahead Left to Right) Parser
- Grammar rules to parse SQL
- Tokens defined in Parser will be available in the tokenizer.

3. Parser (Example)

- The SELECT statement matches the following grammar construct:

```
cmd ::= select(X).  {
 SelectDest dest = {SRT_Output, 0, 0, 0, 0};
 sqlite3Select(pParse, X, &dest);
 sqlite3SelectDelete(pParse->db, X);
}
oneselect(A) ::=
    SELECT distinct(D) selcollist(W) from(X) where_opt(Y)
     groupby_opt(P) having_opt(Q) orderby_opt(Z) limit_opt(L). {
A = sqlite3SelectNew(pParse,W,X,Y,P,Q,Z,D,L.pLimit,L.pOffset);
}
selcollist(A) ::= sclp(P) STAR.
from(A) ::= FROM seltablist(X).
```

4. Code Generator for VM

- select.c/where.c/insert.c/expr.c
- Handles VDBE code generation for the SQL
- Parser based on the grammar calls the respective function to generate the VBDE code.
- The VDBE code is generated based on the logic of the SQL.
- SQL Optimization is also done as part of code generation

4. Code Generator for VM (Example)

- sqlite3Select (select.c)
- Start Code generation with

```
v = sqlite3GetVdbe(pParse);
```

- Generate Op-codes with

```
sqlite3VdbeAddOp4
```

4. Code Generator for VM (Example)

```
sqlite> explain select * from employee;
addr  opcode         p1    p2    p3    p4         p5  comment
----  -------------  ----  ----  ----  ---------  --  -------------
0     Trace          0     0     0                00
1     Goto           0     10    0                00
2     OpenRead       0     2     0     2          00  employee
3     Rewind         0     8     0                00
4     Column         0     0     1                00  employee.name
5     Column         0     1     2                00  employee.age
6     ResultRow      1     2     0                00
7     Next           0     4     0                01
8     Close          0     0     0                00
9     Halt           0     0     0                00
10    Transaction    0     0     0                00
11    VerifyCookie   0     1     0                00
12    TableLock      0     2     0     employee   00
13    Goto           0     2     0                00
```

5. Virtual DB Engine

- vdbe.c
- Similar to assembly programs
- Linear sequence of operations with Op-codes and corresponding Operands
- Operands
  - P1, P2 and P3 are Integers
  - P4 is a null terminated String
  - P5 is a unsigned character

5. Virtual DB Engine (cont.)

- Big SWITCH statement with different Op-codes as different case statements.
- Column P1 P2 P3 P4 P5
  - Interpret the data that cursor P1 points to as a structure built using the MakeRecord instruction.
  - Extract the P2th column from this record. If there are less that (P2+1) values in the record, extract a NULL.

Record Format: Variable Length Integer (VARINT)

- Static Huffman encoding of 64-bit twos-complement integers

Bytes	Value	Bit Pattern
1	7 bit	0xxxxxxx
2	14 bit	1xxxxxxx 0xxxxxxx
3	21 bit	1xxxxxxx 1xxxxxxx 0xxxxxxx
4	28 bit	1xxxxxxx 1xxxxxxx 1xxxxxxx 0xxxxxxx
5	35 bit	1xxxxxxx 1xxxxxxx 1xxxxxxx 1xxxxxxx 0xxxxxxx
.		
.		

Record Format: Record Structure

- The Record structure is stored for each of the record.
- The header size and the types are stored as varints.

```
------------------------------------------------------------------------
| hdr-size | type 0 | type 1 | ... | type N-1 | data0 | ... | data N-1 |
------------------------------------------------------------------------
```

Record Format: Serial Type

- The Serial type identifies the data type. Column data type is irrelevant.

serial type	bytes of data	type
0	0	NULL
1	1	signed integer
2	2	signed integer
3	3	signed integer
4	4	signed integer
5	6	signed integer
6	8	signed integer
7	8	IEEE float
8	0	Integer constant 0
9	0	Integer constant 1
10,11		reserved for expansion
N>=12 and even	(N-12)/2	BLOB
N>=13 and odd	(N-13)/2	text

6. B-tree

- btree.c
- A single B-Tree structure is stored using one or more database pages.
- A page contains a single B-tree node.
  - Table B-tree
    - 64-bit integer as keys
  - Index B-tree
    - Database records as keys

7. Pager

- pager.c
- Used to access a DB file.
- Caching of pages.
- It implements atomic commits/rollbacks by use of a journal file.
- Implements file locking.
  - Exclusive lock while writing.
  - Shared lock while reading.

8. OS Interface

- os.c – os_unix.c/os_win.c
- Abstraction layer to interface with the Operation System

FreeDB – A schema free DB

What is FreeDB

- Schema Free tables.
  - Values as name value pairs - Pair

    (“id”,”1001”)
  - Values as Lists - List

    [10, 20, 30, 40]
- By combination of the above two constructs flexible structures can be constructed.
  - [(“id”, “1001”),(“name”,”MSB”)]

Implementation of Free DB using SQLite

High level Implementation

- Tokenizer
  - To tokenize new tokes (‘[‘ and ‘]’)
- Parser
  - To support new SQL syntax (List & Pair) for free DB
- Code Generator
  - Generate new Opcodes to handle (List & Pair)
- VDBE
  - Execute new Opcodes

Tokenizer (tokenize.c)

- Support for List by use of [and]

```
case ']': {
*tokenType = TK_RB;
return 1; }
case '[': {
if (comma_found) {
*tokenType = TK_LB;
return 1; }
else
*tokenType = c==']' ? TK_ID :TK_ILLEGAL;
return i;
}
```

Parser

- Syntax for List and Pair

```
expr(A) ::= pair(X). {A = X;}
expr(A) ::= list(X). {A = X;}
/* (“price”, 99.99) */
pair(A) ::= LP expr(X) COMMA expr(Y) RP.
{ spanBinaryExpr
(&A,pParse,TK_PAIR,&X,&Y); }
/* [61820, 61821, 61822] */
list(A) ::= LB itemlist(X) RB. { A.pExpr =
sqlite3PExpr(pParse, TK_LIST, 0, 0, 0);
A.pExpr->x.pList = X; }
```

Expression Processing (expr.c) - List

- Inserts a new op-code **OP_List** and then adds the list of the expressions that form the list

```
case TK_LIST:
{ int n;
sqlite3VdbeAddOp2(v, OP_List, pExpr-
>x.pList->nExpr, inReg);
for (n=0;n<pExpr->x.pList->nExpr;n++)
{ inReg = sqlite3ExprCodeTarget(pParse,
pExpr->x.pList->a[n].pExpr, inReg+1) ;
}
```

Expression Processing (expr.c) - Pair

- Inserts a new op-code **OP_StringPair** and then adds the two expressions that form the pair.

```
case TK_PAIR: {
sqlite3VdbeAddOp2(v, OP_StringPair, 2, inReg);

inReg = sqlite3ExprCodeTarget(pParse, pExpr->pLeft,
inReg+1);
inReg = sqlite3ExprCodeTarget(pParse, pExpr->pRight,
inReg+1) ;
```

VDBE (vdbe.c)

- Define a case statement to define a new op-code.

```
case OP_StringPair: {
assert( pOp->p1 == 2 );
pOut->u.i = pOp->p1;
pOut->aux_flags = MEM_Pair; break;
}
```

Record Format Serial Type

- Two new serial types to identify the new data types

serial type	bytes of data	type
N>=12 and 00	(N-12)/4	BLOB
N>=13 and 01	(N-13)/4	text
N>=12 and 10	(N-12)/4	Pair
N>=13 and 11	(N-13)/4	List

Thank You