Rick Sarkar

SQLite Implementation Notes

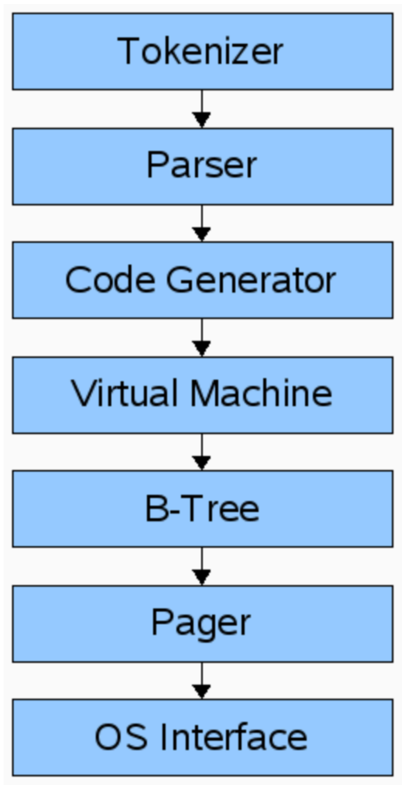
Basic overview of Database Models:

- Relations are sets of tuples (each row is a tuple)
- Tuples are lists of values, where each tuple has the same number of and corresponding components
- Attributes are column headers that are the meaning of each component in a tuple
- Relation name, attribute names, and data types form the schema for the relation
- Database schema is a collection of relation schema

ACID Properties of Transactions:

- A: Atomicity. A transaction is either fully executed or not at all.
- C: Consistency. transactions follow the constraints and expectations on relationships within the database
- I: Isolation. Transaction execution is handled such that they *appear* to be executed in isolation (e.g. no other transaction being executed at the same time)
- D: Durability. The change in the database the transaction caused is never lost once the transaction is executed. This is done using logs and secondary storage that is safe in the event of a crash

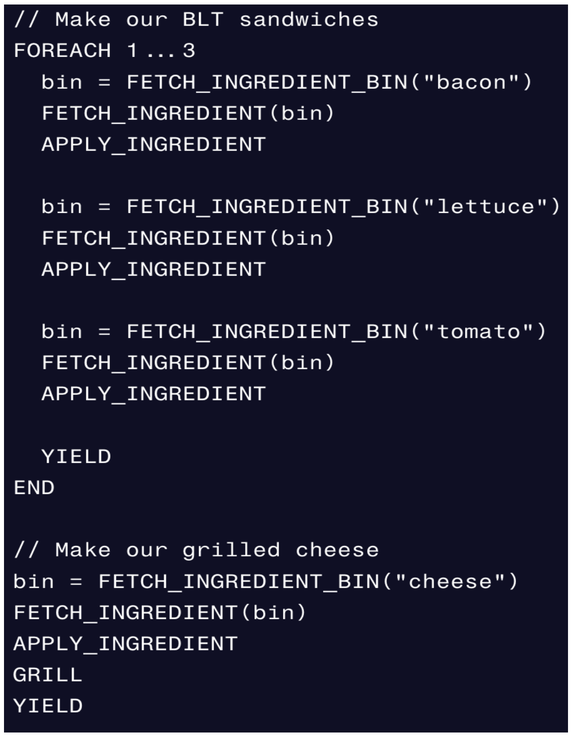
SQLite Internals:

- First, we look at the path a query takes in order to retrieve or change data
- Architecture:
- 

- **Front-End:** Tokenizer, Parser, and Code Generator. The output of the front-end is SQLite virtual machine bytecode (essentially a compiled program that can operate on the database)
- **Back-End:** The rest (VM, B-Tree, Pager, and OS Interface) comprise the back-end.

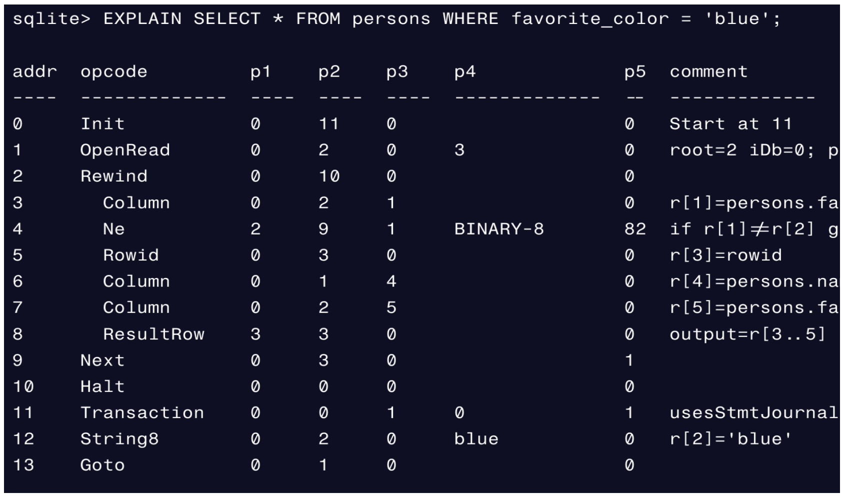
Tokenizer + Parser:

- The tokenizer will take the actual query as input, and break it into words/tokens, like so:
-
  ```
  Make 3 BLT sandwiches hold the mayo, 1 grilled cheese
  ```
- Becomes
-
  ```
  "MAKE", "3", "BLT", "SANDWICHES", "HOLD", "THE", "MAYO", ",", "1",
  "GRILLED", "CHEESE"
  ```
- From there, the parser will try to structure the stream of tokens into an Abstract Syntax Tree (AST)
-
  ```
  {
    "command": "MAKE",
    "sandwiches": [
      {
        "type":"BLT",
        "count": 3,
        "remove": ["MAYO"]
      },
      {
        "type": "GRILLED CHEESE",
        "count": 1
      }
    ]
  }
  ```
- Then, we can create a plan on executing the query -- one that is optimized and can be re-used in the future



```
// Make our BLT sandwiches
FOREACH 1...3
  bin = FETCH_INGREDIENT_BIN("bacon")
  FETCH_INGREDIENT(bin)
  APPLY_INGREDIENT

  bin = FETCH_INGREDIENT_BIN("lettuce")
  FETCH_INGREDIENT(bin)
  APPLY_INGREDIENT

  bin = FETCH_INGREDIENT_BIN("tomato")
  FETCH_INGREDIENT(bin)
  APPLY_INGREDIENT

  YIELD
END

// Make our grilled cheese
bin = FETCH_INGREDIENT_BIN("cheese")
FETCH_INGREDIENT(bin)
APPLY_INGREDIENT
GRILL
YIELD
```

-
- This is bytecode produced by a code generator, which is ran by the Virtual Machine
- Use “EXPLAIN (QUERY) (NOT QUERY PLAN) to display an English representation of bytecode



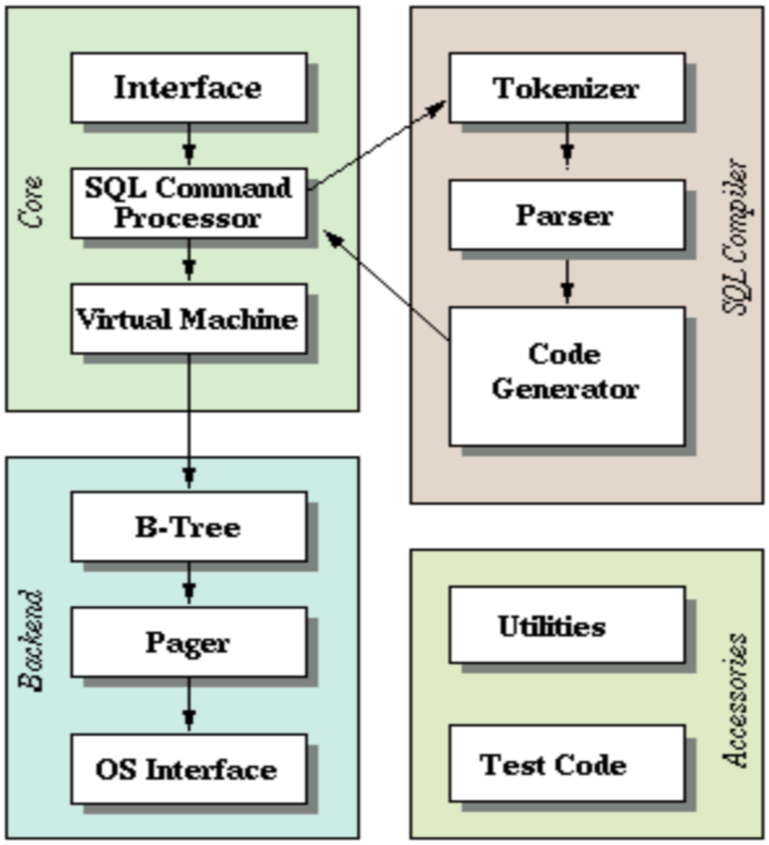
```
sqlite> EXPLAIN SELECT * FROM persons WHERE favorite_color = 'blue';

addr  opcode         p1    p2    p3    p4             p5  comment
----  -------------  ----  ----  ----  -------------  --  -------------
0     Init           0     11    0                    0   Start at 11
1     OpenRead       0     2     0     3              0   root=2 iDb=0; p
2     Rewind         0     10    0                    0
3       Column       0     2     1                    0   r[1]=persons.fa
4       Ne           2     9     1     BINARY-8       82  if r[1]≠r[2] g
5       Rowid        0     3     0                    0   r[3]=rowid
6       Column       0     1     4                    0   r[4]=persons.na
7       Column       0     2     5                    0   r[5]=persons.fa
8       ResultRow    3     3     0                    0   output=r[3..5]
9     Next           0     3     0                    1
10    Halt           0     0     0                    0
11    Transaction    0     0     1     0              1   usesStmtJournal
12    String8        0     2     0     blue           0   r[2]='blue'
13    Goto           0     1     0                    0
```

-
- Some key instructions from the SQLite VM instruction set:
    - OpenRead: creates a cursor that is used to iterate over or move around a table
    - Rewind: moves cursor to first entry of database
    - ResultRow: gives result to caller by copying out three registers
    - Resumes once sqlite3_step() is called.

Back-end overview:

- Tables and indexes are stored as B-tree (balanced)
- Each node in tree is one page, which can be retrieved from or saved to disk by prompting the pager
- Pager has a cache for recently accessed memory pages
- OS interface depends on what SQLite system was compiled for

Another Visual:



Non-SQL statements are known as <u>meta-commands</u> (ex: .exit, .tables, etc)

Memory overview:

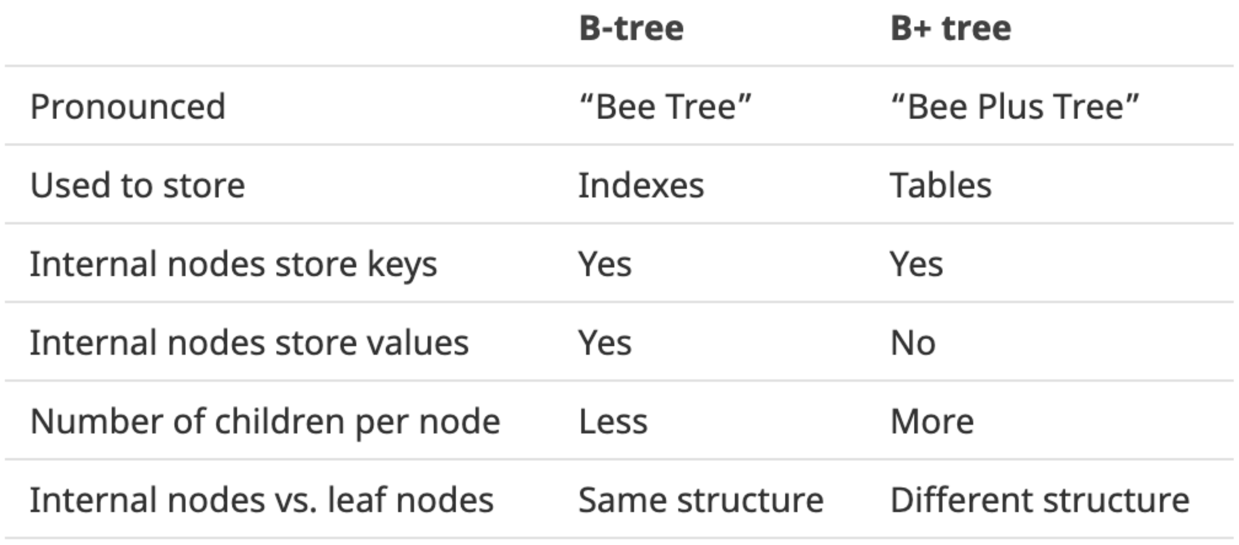
- Store rows in blocks of memory called pages
- Each page stores as many rows as it can fit
- Rows are serialized into a compact representation with each page

- Pages are allocated on demand
- Keep a fixed-size array of pointers to pages
- To ensure persistence of database, the compact rows are saved to a file that can be loaded upon start

We can use a Cursor to move throughout the table. That is, we want to be able to:

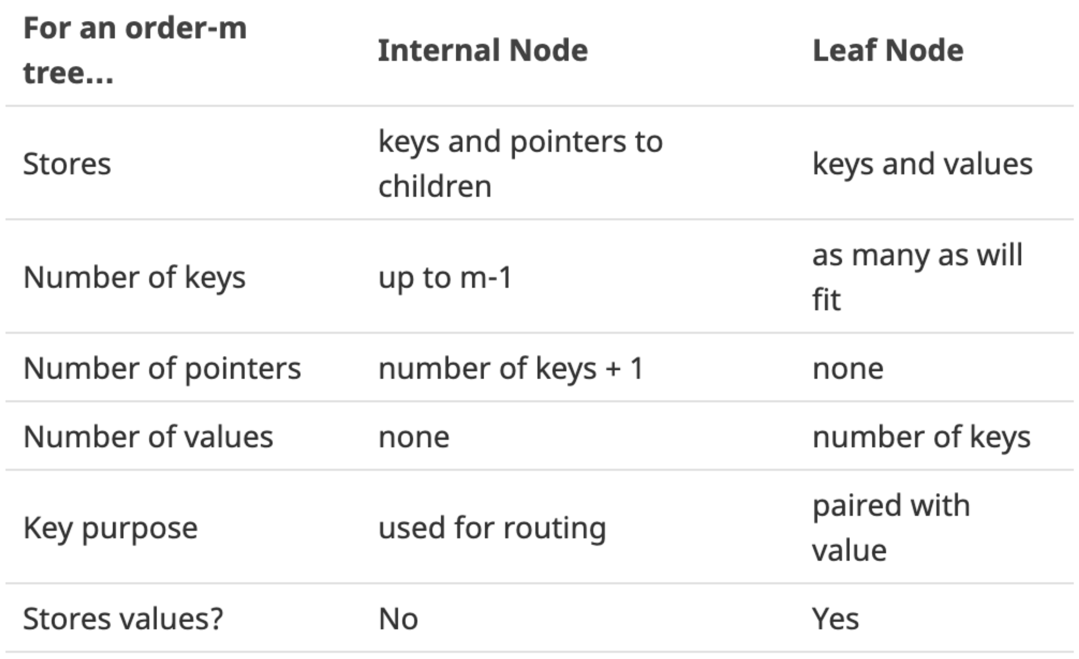
- Create a cursor at the beginning of the table
- Create a cursor at the end of the table
- Access the row the cursor is pointing to
- Advance the cursor to the next row
- Delete the row pointed to by a cursor
- Modify the row pointed to by a cursor
- Search a table for a given ID, and create a cursor pointing to the row with that ID

A Closer look at the B-Tree

- Not the same as a binary tree -- moreso a self-*balancing* tree
- Some goals of a data structures for tables/indexes:
    - Searching for a particular value is fast (at most logarithmic time)
    - Inserting / deleting a value already found is fast (about constant time to rebalance)
    - Traversing a range of values is fast (unlike a hash map) -- this requires some means of grouping
- Max number of children **m** is the **order** of the tree
- Each node must have at least ⌈m/2⌉ children for a mostly balanced tree
- Note that:
    - Leaf nodes have 0 children, including if the root is also a leaf
    - The root node can have fewer than m children but must have at least 2
    - Structure of B-tree for indexes, and B-tree for tables (known as a B+ tree) is different
- 

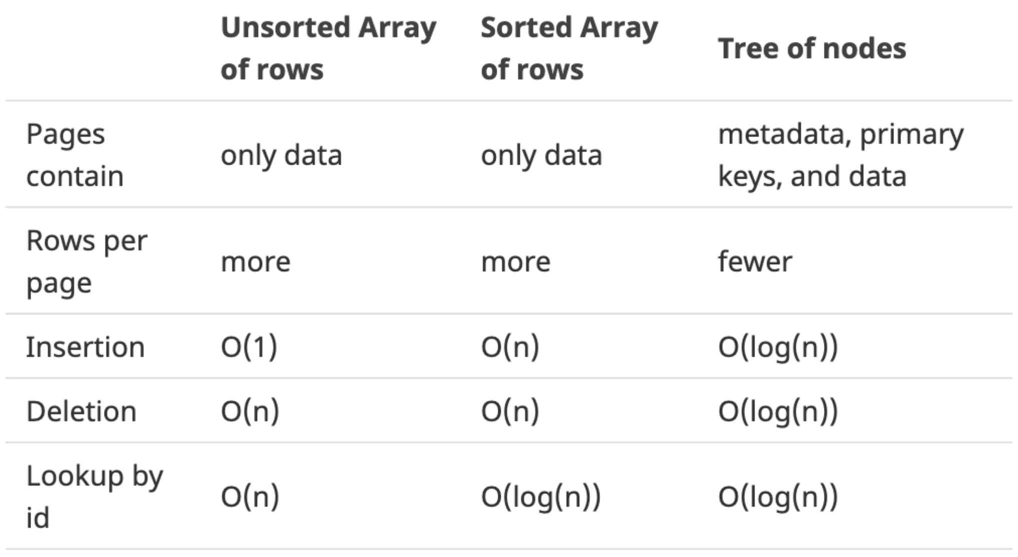
	B-tree	**B+ tree**
Pronounced	“Bee Tree”	“Bee Plus Tree”
Used to store	Indexes	Tables
Internal nodes store keys	Yes	Yes
Internal nodes store values	Yes	No
Number of children per node	Less	More
Internal nodes vs. leaf nodes	Same structure	Different structure

- Nodes with children are internal nodes, as opposed to leaf nodes with none



For an order-m tree...	Internal Node	Leaf Node
Stores	keys and pointers to children	keys and values
Number of keys	up to m-1	as many as will fit
Number of pointers	number of keys + 1	none
Number of values	none	number of keys
Key purpose	used for routing	paired with value
Stores values?	No	Yes

-
- Comparison between unsorted array of rows, sorted array of rows, and tree of nodes:

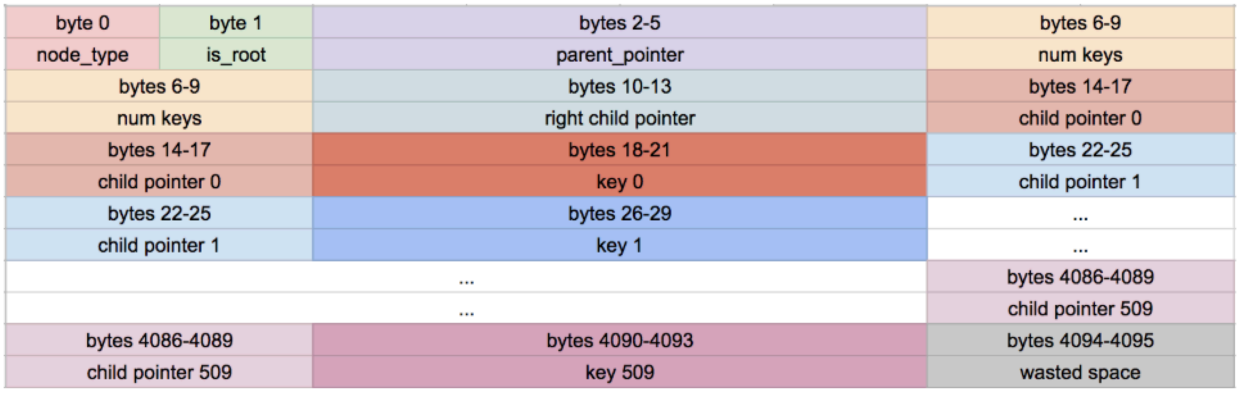


	Unsorted Array of rows	Sorted Array of rows	Tree of nodes
Pages contain	only data	only data	metadata, primary keys, and data
Rows per page	more	more	fewer
Insertion	O(1)	O(n)	O(log(n))
Deletion	O(n)	O(n)	O(log(n))
Lookup by id	O(n)	O(log(n))	O(log(n))

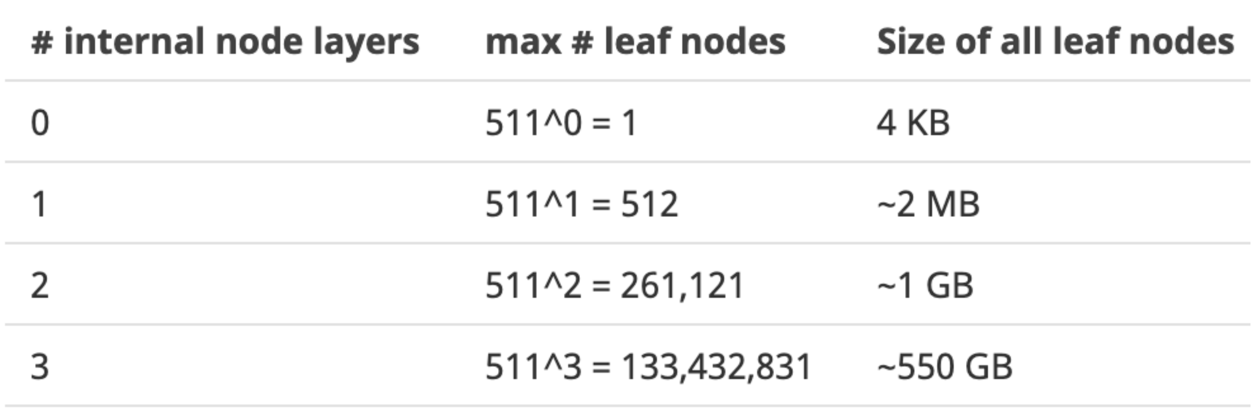
  -
- Handling of splitting/rebalancing:
  - If there is no space on the leaf node, we would split the existing entries residing there and the new one (being inserted) into two equal halves: lower and upper halves.
  - Keys on the upper half are strictly greater than those on the lower half.
  - We allocate a new leaf node, and move the upper half into the new node.

- Creating a new root:
    - Let N be the root node.
    - First allocate two nodes, say L and R.
    - Move lower half of N into L and the upper half into R.
    - Now N is empty. Add 〈L, K,R〉 in N, where K is the max key in L.
    - Page N remains the root.
    - Note that the depth of the tree has increased by one, but the new tree remains height balanced without violating any B+-tree property.

- Internal Node Layout:

- 

byte 0	byte 1	bytes 2-5	bytes 6-9
node_type	is_root	parent_pointer	num keys
bytes 6-9		bytes 10-13	bytes 14-17
num keys		right child pointer	child pointer 0
bytes 14-17		bytes 18-21	bytes 22-25
child pointer 0		key 0	child pointer 1
bytes 22-25		bytes 26-29	...
child pointer 1		key 1	...
...			bytes 4086-4089
...			child pointer 509
bytes 4086-4089		bytes 4090-4093	bytes 4094-4095
child pointer 509		key 509	wasted space

- 

# internal node layers	max # leaf nodes	Size of all leaf nodes
0	511^0 = 1	4 KB
1	511^1 = 512	~2 MB
2	511^2 = 261,121	~1 GB
3	511^3 = 133,432,831	~550 GB

References:

https://cstack.github.io/db_tutorial/parts/part1.html
https://fly.io/blog/sqlite-virtual-machine/
https://sqlite.org/opcode.html#Abortable
https://www.sqlite.org/arch.html
https://play.google.com/store/books/details/Sibsankar_Haldar_SQLite_Database_System_Design_and?id=9Z6IQQnX1JEC&hl=en
https://books.google.com/books/about/Database_System_Implementation.html?id=jOVQAAAAMAAJ&source=kp_book_description