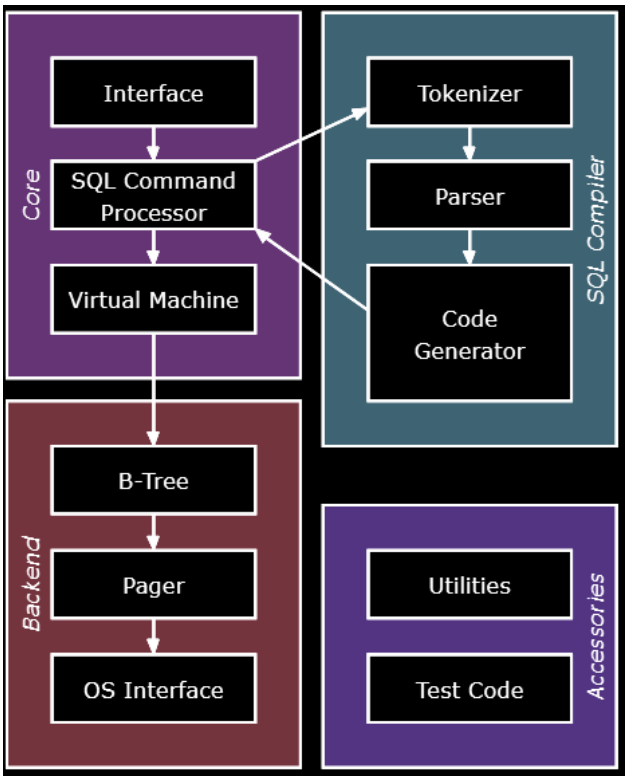
資料庫系統概論 HW3

10811020
陳光悅

2-A

1-a.



https://www.sqlite.org/arch.html

1-b

Interface：大部分的 C code Interface 可以在 main.c、legacy.c 和 vdbeapi.c 等檔案中找到。為了避免命名重複，SQLite library 中的所有的 external symbols 皆以 sqlite3 為前綴。而那些給公共 API 則以 sqlite3_開頭。Extension API 則還會再加上 extension name，如 sqlite3rbu_或 sqlite3session_。

Tokenizer：當要解讀 SQL 語句的字串時，會先 tokenizer 分解成數個 token。相關的 code 可以在 tokenize.c 找到。

Parser：之後 parser 會解讀 token。SQLite 是使用 Lemon parser generator，不容易出錯，並且 reentrant、thread-safe。

Code Generator：parser 器將 token 組合成 parse tree 後，便由 code generator 解析成 parse tree，並執行 SQL。這些 code generator 的檔案包括：attach.c、auth.c、build.c、delete.c、expr.c、insert.c、pragma.c、select.c、trigger.c、update.c、vacuum.c、where.c、whereexpr.c。其中，expr.c 負責處理 SQL 中表達式的部分，並生成相應可以被執行的指令。如：where* .c 負責生成 SELECT、

UPDATE 和 DELETE 中的 WHERE。而 attach.c、delete.c、insert.c、select.c、trigger.c、update.c 和 vacuum.c 則分別處理同名的 SQL。同一個 SQL 語句可能會有無數個不同的方法來查詢出結果，特別是 where*.c 和 select.c 中的邏輯，因此 code generator 還需要找出最佳的 algorithm。

Bytecode Engine：code generator 建立的 bytecode program 需要有一個 virtual machine 來執行。virtual machine 的 code 可以在 vdbe.c 中找到，vdbe.h 定義 virtual machine 與 SQLite library 之間的 interface，vdbeInt.h 定義 virtual machine 自己的 structure 和 interface。vdbe*.c 為 virtual machine 的輔助 code。vdbeaux.c 包含 virtual machine 使用的 interface modules。vdbeapi.c 包含 virtual machine 使用的 external interfaces，如 sqlite3_bind_int()和 sqlite3_step()。vdbemem.c 將變數（string、integer、floating point number、BLOB）儲存在 Mem 的 internal object 中。SQLite 使用 callbacks C code 來實現 SQL function。大部分的 built-in SQL functions（abs()、count()、substr()等）可以在 func.c 中找到。Date 和 time 轉換 function 可以在 date.c 中找到。其餘的 function（coalesce()、typeof()）則直接由 code generator 生成 bytecode。

B-Tree：SQLite 的 database 用 B-tree 儲存在 disk 裡面，可以在 btree.c 中找到。每個 table 和 database 的 index 都使用不同的 B-tree，所有的 B-tree 皆儲存在同一個 disk file 中。btree.h 定義了 B-tree subsystem 與 SQLite library 的 interface。

Page Cache：B-tree module 從 disk 中固定大小的 page 請求訊息，預設的 pages 大小是 4096 bytes。page cache 負責讀取、寫入、緩存 page，同時也提供 database file 的 rollback、atomic commit abstraction、locking。B-tree driver 從 page cache 請求特定的 page，並在通知 page cache 當 page 修改、commit、rollback 改變時。page cache 可以在 pager.c 中找到，WAL mode 則在 wal.c 中，in-memory caching 則在 pcache.c 和 pcache1.c。pager.h 定義了 page cache subsystem 與 SQLite library 的 interface。

OS Interface：SQLite 為了在不同的 operating system 都可以運作，而使用 VFS 的 abstract object，提供了在 disk 上打開、讀取、寫入和關閉 file 的方法。Unix 的 VFS 在 os_unix.c 中、Windows 的 VFS 在 os_win.c 中。

https://www.sqlite.org/arch.html

2-a.

Core：當我們下達一個 SQL 指令時，它會被先 tokenizer 分解成 token，並將這些 token 一一交給 parser。相關的 code 可以在 tokenize.c 找到。其中，sqlite3GetToken()、sqlite3RunParser()兩個 function 分別負責 tokenizer 與 parser SQL 指令。

SQL Complier：parser 器將 token 組合成 parse tree 後，便由 code generator 解析成 parse tree，並執行 SQL。這些 code generator 的檔案包括：attach.c、

auth.c、build.c、delete.c、expr.c、insert.c、pragma.c、select.c、trigger.c、update.c、vacuum.c、where.c、whereexpr.c。其中，expr.c 負責處理 SQL 中表達式的部分，並生成相應可以被執行的指令。如：where* .c 負責生成 SELECT、UPDATE 和 DELETE 中的 WHERE。而 attach.c、delete.c、insert.c、select.c、trigger.c、update.c 和 vacuum.c 則分別處理同名的 SQL。

Backend：SQLite 是透過 B-tree 來進行實作，相關的 code 可在 btree.c 找到。其中，資料庫中的每個 table 和 index，都是使用獨立的 B-tree。B-tree 的預設 pages 大小是 4096 位元組。相關的 code 可以在 pager.c 中找到。

https://www.sqlite.org/arch.html

2-b-i

WHERE：WHERE clause 可以拆成不同的 terms，以 AND 或 OR 連接。為了提高效率，這些 terms 會被分析，當他們滿足下列的條件時，則可利用 index 來找到滿足這些條件的數據。

```
column = expression
column IS expression
column > expression
column >= expression
column < expression
column <= expression
expression = column
expression > column
expression >= column
expression < column
expression <= column
column IN (expression-list)
column IN (subquery)
column IS NULL
column LIKE pattern
column GLOB pattern
```

這些 index 在被建立時，可以指定某些特定的 column 來作為 index 的關鍵 column（ `CREATE INDEX idx_ex1 ON ex1(a,b,c,d,e,...,y,z);` ）。

BETWEEN：如果 WHERE clause 的 term 包含 `expr1 BETWEEN expr2 AND expr3` 的形式，則可替換成 `expr1 >= expr2 AND expr1 <= expr3` 的 virtual terms，但不會生成額外的 byte-code。如果此時兩個 virtual terms 皆符合 index 的條件，那原始的 BETWEEN clause 則可被省略。

OR：如果 WHERE clause 的 term 是被用 OR 連接，則有兩種不同的方式處理。其一， `column = expr1 OR column = expr2 OR column = expr3 OR ...` 可以被改寫為 `column IN (expr1,expr2,expr3,...)` 。其二， `expr1 OR expr2 OR expr3` 可以被改寫為

```
rowid IN (SELECT rowid FROM table WHERE expr1
          UNION SELECT rowid FROM table WHERE expr2
          UNION SELECT rowid FROM table WHERE expr3)
```

。

LIKE：如果 WHERE clause 中有 LIKE 或 GLOB，當符合下述條件時，即可利用 index 來範圍搜尋的方式優化。

1. LIKE 或 GLOB 的右側必須是 string literal 或 parameter bound to a string literal，並且不是以 wildcard character 開頭。
2. LIKE 或 GLOB 的左側是 indexed column 的名稱或右側不能是「-」或數字開頭。

3. LIKE 和 GLOB 的 built-in functions 不能與 sqlite3_create_function()裡的一樣。
4. GLOB 必需使用 built-in BINARY collating sequence。
5. LIKE 如果是 case_sensitive_like mode，則 column 必需使用 BINARY collating sequence；反之，則使用 built-in NOCASE collating sequence。
6. 如果使用 ESCAPE，則 ESCAPE 的 character 必須是 ASCII 或 UTF-8。

https://www.sqlite.org/optoverview.html

2-b-ii

SQLite 使用 nested loop 連接操作，每個表都被一個 loop 連接。每個 loop 上可能使用一個或多個 index 來加快查詢，或者是 loop 可以是 full table scan，來掃過 table 中的每一的 row。因此 query planning 可以拆成兩個 subtask：

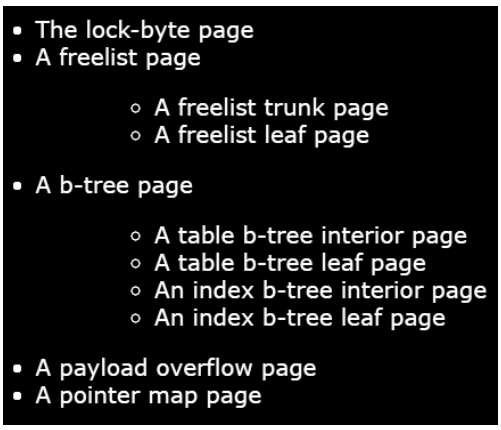
1. 選擇不同順序的 nested loop
2. 為每個 loop 選擇合適的 index

https://www.sqlite.org/queryplanner-ng.html

3-a

完整的 SQLite 資料庫通常是存放在 disk 中的單一文件夾中，文件夾的名字為 main database file。而在 transaction 期間，SQLite 並不會馬上將資料寫入 main database file 中，而是會先寫入 rollback journal 的文件夾中（write-ahead log (WAL) mode）。而當資料要修改時，未修改的原始內容先被寫入 rollback journal 中，如此可以降低資料庫在寫入時可能存在的損壞風險。如果在 transaction 完成前應用程序或主機電腦發生崩潰，此時就可以從 rollback journal 中恢復 main database file，此時的 rollback journal 可被稱為 hot journal 或 hot WAL file。

main database file 由一個或多個 page 組成，大小為 512～65536 byte，所有位於同一資料庫內的 page 大小皆相同。每一個 page 都有單一的用途，如下：



每個 page 的前 100 bytes 為 database file header，包含了所有關於此 page 存取資料的資訊，以 big-endian 的方式儲存。再之後才是 page 中真正存取的資料內容。database file header 的內容如下：

Database Header Format

Offset	Size	Description
0	16	The header string: "SQLite format 3\000"
16	2	The database page size in bytes. Must be a power of two between 512 and 32768 inclusive, or the value 1 representing a page size of 65536.
18	1	File format write version. 1 for legacy; 2 for WAL.
19	1	File format read version. 1 for legacy; 2 for WAL.
20	1	Bytes of unused "reserved" space at the end of each page. Usually 0.
21	1	Maximum embedded payload fraction. Must be 64.
22	1	Minimum embedded payload fraction. Must be 32.
23	1	Leaf payload fraction. Must be 32.
24	4	File change counter.
28	4	Size of the database file in pages. The "in-header database size".
32	4	Page number of the first freelist trunk page.
36	4	Total number of freelist pages.
40	4	The schema cookie.
44	4	The schema format number. Supported schema formats are 1, 2, 3, and 4.
48	4	Default page cache size.
52	4	The page number of the largest root b-tree page when in auto-vacuum or incremental-vacuum modes, or zero otherwise.
56	4	The database text encoding. A value of 1 means UTF-8. A value of 2 means UTF-16le. A value of 3 means UTF-16be.
60	4	The "user version" as read and set by the user_version pragma.
64	4	True (non-zero) for incremental-vacuum mode. False (zero) otherwise.
68	4	The "Application ID" set by PRAGMA application_id.
72	20	Reserved for expansion. Must be zero.
92	4	The version-valid-for number.
96	4	SQLITE_VERSION_NUMBER

https://www.sqlite.org/fileformat2.html

3-b

VFS 為 SQLite 提供 file 的打開、讀取、寫入、關閉等方法。

```
typedef struct sqlite3_file sqlite3_file;
struct sqlite3_file {
  const struct sqlite3_io_methods *pMethods;  /* Methods for an open file */
};
```

https://www.sqlite.org/c3ref/file.html

```
typedef struct sqlite3_vfs sqlite3_vfs;
typedef void (*sqlite3_syscall_ptr)(void);
struct sqlite3_vfs {
  int iVersion;            /* Structure version number (currently 3) */
  int szOsFile;            /* Size of subclassed sqlite3_file */
  int mxPathname;          /* Maximum file pathname length */
  sqlite3_vfs *pNext;      /* Next registered VFS */
  const char *zName;       /* Name of this virtual file system */
  void *pAppData;          /* Pointer to application-specific data */
  int (*xOpen)(sqlite3_vfs*, sqlite3_filename zName, sqlite3_file*,
               int flags, int *pOutFlags);
  int (*xDelete)(sqlite3_vfs*, const char *zName, int syncDir);
  int (*xAccess)(sqlite3_vfs*, const char *zName, int flags, int *pResOut);
  int (*xFullPathname)(sqlite3_vfs*, const char *zName, int nOut, char *zOut);
  void *(*xDlOpen)(sqlite3_vfs*, const char *zFilename);
  void (*xDlError)(sqlite3_vfs*, int nByte, char *zErrMsg);
  void (*(*xDlSym)(sqlite3_vfs*,void*, const char *zSymbol))(void);
  void (*xDlClose)(sqlite3_vfs*, void*);
  int (*xRandomness)(sqlite3_vfs*, int nByte, char *zOut);
  int (*xSleep)(sqlite3_vfs*, int microseconds);
  int (*xCurrentTime)(sqlite3_vfs*, double*);
  int (*xGetLastError)(sqlite3_vfs*, int, char *);
  /*
  ** The methods above are in version 1 of the sqlite_vfs object
  ** definition.  Those that follow are added in version 2 or later
  */
  int (*xCurrentTimeInt64)(sqlite3_vfs*, sqlite3_int64*);
  /*
  ** The methods above are in versions 1 and 2 of the sqlite_vfs object.
  ** Those below are for version 3 and greater.
  */
  int (*xSetSystemCall)(sqlite3_vfs*, const char *zName, sqlite3_syscall_ptr);
  sqlite3_syscall_ptr (*xGetSystemCall)(sqlite3_vfs*, const char *zName);
  const char *(*xNextSystemCall)(sqlite3_vfs*, const char *zName);
  /*
  ** The methods above are in versions 1 through 3 of the sqlite_vfs object.
  ** New fields may be appended in future versions.  The iVersion
  ** value will increment whenever this happens.
  */
};
```

https://www.sqlite.org/c3ref/vfs.html

```
typedef struct sqlite3_io_methods sqlite3_io_methods;
struct sqlite3_io_methods {
  int iVersion;
  int (*xClose)(sqlite3_file*);
  int (*xRead)(sqlite3_file*, void*, int iAmt, sqlite3_int64 iOfst);
  int (*xWrite)(sqlite3_file*, const void*, int iAmt, sqlite3_int64 iOfst);
  int (*xTruncate)(sqlite3_file*, sqlite3_int64 size);
  int (*xSync)(sqlite3_file*, int flags);
  int (*xFileSize)(sqlite3_file*, sqlite3_int64 *pSize);
  int (*xLock)(sqlite3_file*, int);
  int (*xUnlock)(sqlite3_file*, int);
  int (*xCheckReservedLock)(sqlite3_file*, int *pResOut);
  int (*xFileControl)(sqlite3_file*, int op, void *pArg);
  int (*xSectorSize)(sqlite3_file*);
  int (*xDeviceCharacteristics)(sqlite3_file*);
  /* Methods above are valid for version 1 */
  int (*xShmMap)(sqlite3_file*, int iPg, int pgsz, int, void volatile**);
  int (*xShmLock)(sqlite3_file*, int offset, int n, int flags);
  void (*xShmBarrier)(sqlite3_file*);
  int (*xShmUnmap)(sqlite3_file*, int deleteFlag);
  /* Methods above are valid for version 2 */
  int (*xFetch)(sqlite3_file*, sqlite3_int64 iOfst, int iAmt, void **pp);
  int (*xUnfetch)(sqlite3_file*, sqlite3_int64 iOfst, void *p);
  /* Methods above are valid for version 3 */
  /* Additional methods may be added in future releases */
};
```

https://www.sqlite.org/c3ref/io_methods.html

open：SQLite 在打開 file 時，會使用 xOpen。根據 database 的設定，xOpen 初始化 sqlite3_file 結構並設置 pMethods。

read：讀取數據時，會使用 xRead。根據 iOfst 決定開始讀取的位置，再根據 iAmt 決定讀取多少 data，並將其存儲在 buffer 中。

write：寫入數據時，會使用 xWrite。根據 iOfst 決定開始寫入的位置，再根據 iAmt 決定寫入多少 data。

close：關閉 file 時，會使用 xClose。釋放 file 所佔用的任何資源。

https://www.sqlite.org/vfs.html

4-a

SQLite 3.0.0 以 locking and journaling mechanism 來實現 concurrency control。pager 會使用一些特定的 modules（os_unix.c 或 os_win.c 等等），與 operating system 進行互動，為 operating system services 提供一致的 abstraction。

一個 database file 可以處於下述五種鎖定狀態的其中一種：

1. UNLOCKED：database 上沒有任何 lock，既不能 read，也不能 write，視為默認狀態。
2. SHARED：database 可以 read，但不能 write。可以多個 processes 同時持有 SHARED lock。
3. RESERVED：process 計劃之後的某個時刻 write database file，但目前只是在 read，而非 write。同一時間只能 active 一個 RESERVED lock。但可以多個 SHARED lock 與一個 RESERVED lock 同時存在。
4. PENDING：表示 process 希望盡快 write database file，只是在等所有剩下的 SHARED lock 被清除，以便獲得 EXCLUSIVE lock。當有 PENDING lock 時，則不允許 database 新增其他的 SHARED lock，但允許現有的 SHARED lock 繼續存在。
5. EXCLUSIVE：可以 write database file。文件上只允許有一個 EXCLUSIVE lock，而且此時不允許有任何其他類型的 lock 存在。

operating system 的 interface layer 可以識別和追蹤這五種 lock，而 page module 只會追蹤其中四種。因為在這四種 lock 中，PENDING lock 只是 EXCLUSIVE lock 的臨時過渡階段而已，因此 page module 並不會不追蹤 PENDING lock。

當 process 想要修改 database file（不是 WAL mode）時，他會先在 rollback journal 中記錄原始未修改過的內容。rollback journal 是一個普通的 disk file，與 database file 位於相同的目錄或文件夾中，名稱的前綴與 database file 相同，後綴為-journal。rollback journal 還記錄 database 的初始大小，因此如果數據庫文件增長，則可以在 rollback 時截斷回到其原始大小。

當 rollback journal 需要 rollback 恢復 database 的內容時，就被稱為 hot journal。當 process 正在更新 database 到一半時，如果發生 operating system 崩潰或斷電，就會導致更新無法完成，此時就需要建立 hot journal 以救回 database 裡面的內容。因此如果直到 process 完成時都沒有發生異常狀況，就永遠不會出現 hot journal。

4-b-i

當要修改 database 時，相關的 database file 會進行鎖定，防止誤改到其他的 database file。

如果是讀取操作時，會給以 SHARED lock，且可以同時多個 processes 進行；寫入操作時，會先給 RESERVED lock，告訴 processes 將有 database file 要進行寫入的動作，而當要真正開始寫入時，會先給 PENDING lock，再給 EXCLUSIVE lock。

4-b-ii

當我們對同一個 database 進行兩個 transaction，都是試圖去更新 database。理論上應該要等第一個 transaction 完成，才能進行下一個 transaction。但如果第二個 transaction 並沒有等第一個 transaction 完成，就直接開始進行第二個 transation，此時就會發生 nondeterministic。

因此在對 database 進行修改時，應該要有 locking mechanism，避免一個 transaction 尚未完成，就開始另一個 transation。此外還有 WAL module，也可以確保在系統在崩潰或斷電的情況下，所做到一半的更改可以被恢復，從而減少資料損壞的風險。

2-B Temporary Files Used By SQLite

SQLite 在處理 database 的過程中，會產生下述九種臨時文件。

1. Rollback journals
   rollback journal 實現了 atomic commit 和 rollback 的功能，與 database file 位於相同的目錄中，並與 database file 的名稱相同，只是在最後加上-journal。通常在 transaction 開始時創建，並在 transaction commit 或 rollback 後刪除。如果沒有 rollback journal，則當 transaction 在進行的過程中發生中斷，就無法 rollback，回朔到 transaction 之前的 database file。
2. Super-journals
   super-journal 是當一個 transaction 中對使用 ATTACH statement 添加到單個 database 連接的多個 database 進行修改時，作為 atomic commit 過程的一部分而使用的。super-journal 與 database file 位於相同的目錄中，具有隨機生成的後綴。super-journal 包含在 transaction 期間更改的所有附加 auxiliary database 的名稱。在 transaction commit 後，才會刪除 super-journal。在 transaction 中，super-journal 使得 database 中的所有更改，要麼全部 rollback，要麼全部 commit。super-journal 僅為涉及多個 database file 的 commit 操作所創建，其中至少要有兩個 database 滿足以下所有要求：
   1. transaction 有對該 database 做修改

2. PRAGMA synchronous 設置不是 OFF
3. PRAGMA journal_mode 不是 OFF、MEMORY 或 WAL

3. Write-ahead Log (WAL) files

   當在 WAL mode 運行時，WAL files 的功能與 rollback journal 相似，皆是實現了 atomic commit 和 rollback。且與 rollback journal 一樣，位於 database file 的相同目錄中，並與 database file 的名稱相同，只是在最後加上-wal。當 database 的第一個 connection 被打開時，WAL files 就會被創建，等到 database 的最後一個 connection 被關閉後，才會被刪除。但如果最後一個連接並不是正常的被關閉，則 WAL file 仍會保留在 filesystem 中，直到下一次打開 database 時，才會自動清除。

4. Shared-memory files

   在 WAL mode 運行時，與同一個 database file 有關聯的所有 SQLite database connection 需要共享一些內存，用來儲存 WAL file 的 index。因此只有在 WAL mode 時，才會有 shared-memory file。shared-memory file 與 database file 位於相同的目錄中，名稱與 database file 相同，是在最後加上-shm。shared-memory file 的惟一目的，是 WAL mode 訪問同一個 database 的多個 process 時，提供一個 shared-memory。因此如果 VFS 能夠提供一個 shared-memory 的訪問方法，則可以替代 shared-memory file。例如，當 PRAGMA locking_mode 設置為 EXCLUSIVE 時，此時只有一個 process 能夠訪問 database file，則 shared-memory 即可分配到 heap 中，而不是創建一個 shared-memory file。在創建 WAL file 時，shared-memory file 也會同時被創建，並且在刪除 WAL file 時會被刪除。在 WAL file 恢復期間，則根據正在恢復的 WAL file 的內容，shared-memory file 會被從頭開始重新創建。

5. Statement journals

   statement journal 是用在一個冗長的 transaction 中，rollback 部分結果。例如，假設有一個 UPDATE statement 試圖修改 database 中的 100 rows。但在修改前 50 rows 後，遇到 constraint violation，導致此 UPDATE statement 無法繼續進行。此時 statement journal 只需要 rollback 前 50 rows 的更改，使得數據庫恢復到 UPDATE statement 開始前的狀態。statement journal 只會在 UPDATE 或 INSERT statement 時被建立，因為只有兩個 statement 才有可能發生修改多 rows，並且會觸發 constraint violation 或 RAISE exception，需要恢復部分的結果。如果 UPDATE 或 INSERT 不包含在 BEGIN...COMMIT 中，並且在同一 database connection 上沒有其他的 active statements，則也不會 statement journal，因為此時使用 rollback journal 即可達到目的。statement journal 的名稱是隨機的，並且不一定與 main database 位於同一目錄。會在 transaction 結束時自動刪除。statement journal 的大小與 UPDATE 或 INSERT statement 所更改的大小成正相關。

6. TEMP databases

CREATE TEMP TABLE syntax 所創建的 table，與其相關的 index、trigger、view，都存儲在個別的 temporary database file 中。temporary database file 在首次遇到 CREATE TEMP TABLE syntax 時創建。這個單獨的 temporary database file 也有相應的 rollback journal。temporary database file 與其 rollback journal，只有在使用 CREATE TEMP TABLE syntax 時才會被創建。當使用 sqlite3_close()關閉 database connection 時，temporary database file 就會自動刪除。temporary database file 與使用 ATTACH statement 添加的 auxiliary database files 非常相似。但 temporary database 會在 database connection 關閉時自動刪除。temporary database 的設定永遠是 synchronous=OFF 和 journal_mode=PERSIST PRAGMA，且 temporary database 不能用於 DETACH，其他的 process 也不能 ATTACH temporary database。

7. Materializations of views and subqueries
   當 query 中有 subquery 時，就會需要將結果存儲在一個 temporary table 中。SQLite 中的 query optimizer 會嘗試避免 temporary table 的產生，但很多時候還是無法避免。此時，這些 temporary table 將會分別存儲在他們個別的 temporary file 中，並在 query 結束時自動刪除。
8. Transient indices
   SQLite 可以利用 transient index 來實現一些 SQL 的功能，例如：
   1. ORDER BY 或 GROUP BY clause
   2. aggregate query 中的 DISTINCT
   3. 使用 UNION、EXCEPT、INTERSECT 合併的 SELECT statement

   這些 transient index 都會存儲在各自的 temporary file 中，並且在這些的 statement 使用結束時自動刪除。
9. Transient databases used by VACUUM
   VACUUM 是一個在 SQLite database 中用於優化和壓縮存儲空間的指令。當 database 中的 table 被修改或刪除時，SQLite 可能會保留已使用的空間，而不會立即回收。這就會導致 database file 佔用不必要的磁盤空間。通過 VACUUM，SQLite 將重新組織 database file，並回收被刪除或修改過的空間。這樣可以壓縮 database file 的大小，釋放未使用的存儲空間，並提高 database 的性能。而 VACUUM 的工作原理是，創建一個 temporary file，然後將 database 的內容完整地重建到 temporary file 中，接著將 temporary file 的內容複製回原始的 database file 中，並刪除 temporary file。這個過程可以清理和壓縮 database file，並將其大小恢復到適合的大小。而在執行 VACUUM 時，會將 database 變為 read only，直到 VACUUM 完成。因此在執行 VACUUM 前，需要確保沒有其他 transaction 在使用該 database。

https://www.sqlite.org/tempfiles.html