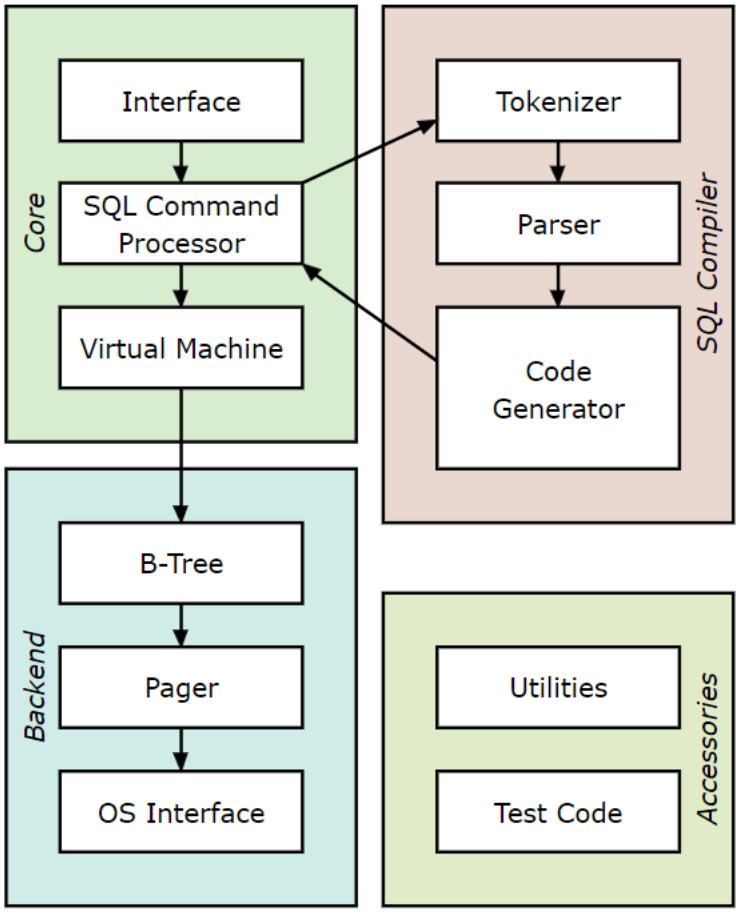
DBMS HW3

110652021 龔大承

1. Describe the general (high-level) structure of SQLite code base:

a. show a diagram of SQLite structure

(link: https://www.sqlite.org/arch.html)



根據連結中的圖表可知，SQLite 主要可以分成四大部分，分別是 core, SQL Compiler, Backend 以及 Accessories.其中每部分又可再細分為更小的 component.

b. list the components of SQLite and describe their functionality

(link: https://www.sqlite.org/arch.html)

根據連結中的圖表(問題 1 中的圖表)，我定義 component 是圖表中的每個小方框，也就是分別為:

Interface(link: https://www.sqlite.org/c3ref/intro.html)

SQLite 的 interface 主要包含三大類，分別是物件列表(List of object), 常數列表(List of constant)以及函數列表(List of function)，這三個列表分別定義了 SQLite 中所使用到的物件及抽象資料型別、使用到的 constant、及所有會在運作中使用到的函式。關於這些功能的 code 主要包含在 main.c, legacy.c, and vdbeapi.c 三個檔案中。

Tokenizer:

當每個 SQL 被下給 SQLite 時，Tokenizer 會將整句 SQL 分解成大小不等的 Token，再將這些 Token 送給 Parser 去做解析最後執行，關於這部分的 code 主要在 tokenize.c 中。

Parser:

在拿到 Tokenizer 送出的不同的 Token 後，Parser 將會根據這些 SQL 的文法來賦予這些 Token 他們的意義，在這個階段中 SQL 將從單純的文字敘述轉變成有意義的查詢語句。Parser 將會根據 SQL 建立執行時的 Parse tree 以便 Query 的查詢。這些 code 主要在 parse.y 中。

Code generator:

在這部分，Code generator 會根據 parse tree 來執行相對應的查詢，一個 query 主要由 select, where, insert, update 等等語句組成，故簡單的 expression 會經由 expr.c 中的程式進行整理，並根據 attach.c, delete.c, insert.c, select.c, trigger.c update.c, and vacuum.c 這些檔案中的程式將整個 query 轉換成相對應的程式，其中需要特別注意的是 where*.c 以及 select.c 這兩個檔案，因為大部分跟 select 及 where 相關的 query 都有很多種不同的表達方式，每種方式執行起來的時間也會大相逕庭，所以這兩個檔案不僅僅會單純地將 SQL 轉換成可執行的 bytecode，還會利用內建的 AI 來選擇最有效率的執行方式。

Bytecode engine:

前面的 code 轉換成 bytecode 後將會由 Virtual Machine 來負責執行，然而執行中也會遇到需要其他 SQL 資源的時候，這時 vdbe.h 這個標頭檔就會建立起 Virtual Machine 和其他的 source 之間的 interface，來完成 query 的執行，並且還有 vdbeaux.c, vdbeapi.c, vdbemem.c 這些檔案也都有類似的功能。

B-Tree:

因為在每個 Table 中都需要自己的 B-Tree 來作為儲存資料的結構，所以 SQLite 也需要定義 B-Tree 的實作，這些相對應的檔案存在於 btree.c 中。

Page Cache:

B-Tree 以固定的大小來讀取硬碟中的內容，而 page cache 主要負責存取、寫入以及快取這些 page，並且在 atomic, commit 後需要復原時可以由 page cache 中的內容做為 recover 依據。這部分的 code 主要存在於 pager.c 中。

OS-Interface:

為了提高 SQLite 的在不同作業系統之間的相容性，SQLite 建立了名為 VFS 的特殊物件，這個 object 有開檔、讀檔、寫入檔案、關閉檔案的功能，並且根據所在作業系統的不同(僅限於 windows, unix)提供了相對應的函式(EX:time(), date())。

c. recall the lecture of Database System Concepts, compare components of SQLite to the components mentioned in the lecture slides, what are the special features of SQLite:
(link: https://www.sqlite.org/serverless.html)

印象中老師上課所提到的 database 觀念中和 SQLite 相同的有:Tokenizer, Parser, B-tree, pager(以上都只是我認為有關連性但不確定是否是相同的東西)，沒有聽過的觀念有 OS-interface:沒印想老師有提到說 DB 在不同的作業系統下會有甚麼不同的行為、Bytecode 及 VM 相關的部分:老師上課時好像沒有特別提到說 query 最後會在 virtual machine 上執行其查詢的動作，還有 SQLite 使用了 VFS 來做為檔案管理的途徑，使得在不同作業系統間的相容性變得較高。

最重要的一點是，SQLite 是 serverless 的 database，是目前唯一使用這樣模式的 database，也就是說，整個SQLite 的運行都在 user 的電腦上，這使得 SQLite 和其他我們聽過的如 mysql 等等的資料庫有很大的不同，他可以直接在磁碟上進行資料的讀寫，不用利用 internet 和 server 做資料 的交換，因此速度上有不同。然而，使用 serverless 的 database 也比較容 易有 bug 的產生，因為在個人電腦上執行較在完整的 server 上執行更難維 護，而且 SQLite 因為直接存取磁碟的資料內容，這些資料更容易被其他應用程式給使用，有可能產生問題。

2. Describe how SQLite process a SQL statement (25 points)(link: https://www.sqlite.org/howitworks.html#compiling_sqlite_programs)

a. list the components included in the procedure, describe their roles in the procedure (a brief explanation is enough here)

我認為會使用到的 component 如下:

Tokenizer:

首先 SQL statement 會被送給 Tokenizer，並經由 Tokenizer 產生相對應的 Tokens，這些 Tokens 可以是:

1. 關鍵字(keyword):”SELECT”, “UPDATE”.
2. 特定 table 或 column 的識別字(identifier)
3. 標點符號:’,’, ‘==’, ‘;’.
4. 數值或是字串:1, 2.09, “name”.
5. 空白或是註解

其中空白及註解會在產生 Token 時捨棄，其餘的 Tokens 則會被送往 parser

Parser:

Parser 會根據收到的 Tokens 來決定執行的樹(AST, Abstract syntax tree) https://en.wikipedia.org/wiki/Abstract_syntax_tree)，每個 AST 都代表一個 program 執行時的流程，並且決定了程式的運作效率，在這個階段，因為對於同一個 SQL 也許有很多種會產生一樣結果的寫法，SQLite 內建了 AI 的機制會傾向於產生效率最高(執行時間最短)的 AST 來加速 query 的進行。Parser 會講產生出來的 AST 送給 code generator，以將程式的運作流程轉換為相對應的程式。

Code generator:

在接收到 Parser 產生的 AST 後，code generator 也會利用 AI 來找出效率最好的 query 方式，並且挑選最適合用於 implement 這樣的 AST 的演算法，並產生相對應的 bytecode，產生出來可執行的 bytecode 稱為”prepared statement”.

根據上述的三個 componet 就可以將 SQL 的語句轉換成可以執行的 bytecode

b. describe how SQLite optimize the query execution in detail (link: https://www.sqlite.org/optoverview.html)

i. explain how each term (where, like, between etc.) are optimized, how indexes are used in the optimization:

1.WHERE:

首先，我們需要分出兩種情形，若我們的 where 子句中是包含 and 連接的條件，則我們將視這幾個由 and 連接起來的條件為 terms，若是由 or 連接的條件則將整個 where 子句視為一個大的 term，並且對這個”大”的 term 執行 or 子句的最佳化(待會才會回答的問題，這裡先不討論)

在 where.c 中的函數 whereLoopAddBtreeIndex()裡用挑選最好的 indices 來達成優化，利用這個函數可以挑選出執行效率最佳的 indices 集合，以達到最好的執行效率，其宣告及註解如下圖所示:

```
/*
** We have so far matched pBuilder->pNew->u.btree.nEq terms of the
** index pIndex. Try to match one more.
**
** When this function is called, pBuilder->pNew->nOut contains the
** number of rows expected to be visited by filtering using the nEq
** terms only. If it is modified, this value is restored before this
** function returns.
**
** If pProbe->idxType==SQLITE_IDXTYPE_IPK, that means pIndex is
** a fake index used for the INTEGER PRIMARY KEY.
*/
static int whereLoopAddBtreeIndex(
  WhereLoopBuilder *pBuilder,     /* The WhereLoop factory */
  SrcItem *pSrc,                  /* FROM clause term being analyzed */
  Index *pProbe,                  /* An index on pSrc */
  LogEst nInMul                   /* log(Number of iterations due to IN) */
```

2.BETWEEN:whererangeskipscanest

在 query 的處理時，會將 between 子句視為兩個條件(大於及小於)的交集以幫助進行分析，但產生 bytecode 則不一定是依照將 between 拆解過後的結果來產生，其拆解方式如下:

If a term of the WHERE clause is of the following form:

expr1 **BETWEEN** *expr2* **AND** *expr3*

Then two "virtual" terms are added as follows:

expr1 **>=** *expr2* **AND** *expr1* **<=** *expr3*

若使用在 between 條件的 column 剛好是 index 的話，則 SQLite 可以利用這欄 index 來快速選出 between 所篩選出來的資料。

3.OR:

在 or 的最佳化中若 or 子句是由同個 column=某個值的條件連結而成，則可以用 IN 來改寫:

column **=** *expr1* **OR** *column* **=** *expr2* **OR** *column* **=** *expr3* **OR ...**

Then that term is rewritten as follows:

column **IN** (*expr1*,*expr2*,*expr3*,...)

改寫完成後，只需要根據正常的 index 規則來進行 IN 語句的查詢即可。

若 or 子句無法使用上面的方法進行優化(即 column 不同或並非=)則將這些條件分別視為獨立的 where 子句並用各自的 indexing 進行評估，其效果如下:

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

也就是說，將符合不同條件的結果分別取出再取聯集。

4.LIKE:

我們可以先定義 LIKE 子句的優化必須有甚麼限制:

(1) Like 的右邊不可以放以奇怪的字開頭的字串

(2) LIKE 的左邊不可以放數字 EX:123 LIKE ….

更精確地說，LIKE 的左邊必須是 indexed 的欄位且 LIKE 右側的敘述不可以”-“開頭，這些限制條件是為了避免在字典排序中”9”>”10”但實際上 10>9 造成的錯誤

(3) 必須注意是否是 case sensitive(定義在 pragma 中)

(4) 必須注意 LIKE 語句以及欄位的編碼格式(UTF-8, ASCII…)

SQLite 會將 LIKE 改寫為兩個虛擬的 term 的聯集，並利用者兩個 term 來進行 indexing，最後再對取出的結果找出符合 LIKE 條件的結果。

Source code 中的一些函式可以幫助 like 的優化:

尋找可用的 index(where.c):

```
/*
** Initialize a WHERE clause scanner object.  Return a pointer to the
** first match.  Return NULL if there are no matches.
**
** The scanner will be searching the WHERE clause pWC.  It will look
** for terms of the form "X <op> <expr>" where X is column iColumn of table
** iCur.   Or if pIdx!=0 then X is column iColumn of index pIdx.  pIdx
** must be one of the indexes of table iCur.
**
** The <op> must be one of the operators described by opMask.
**
** If the search is for X and the WHERE clause contains terms of the
** form X=Y then this routine might also return terms of the form
** "Y <op> <expr>".  The number of levels of transitivity is limited,
** but is enough to handle most commonly occurring SQL statements.
**
** If X is not the INTEGER PRIMARY KEY then X must be compatible with
** index pIdx.
*/
static WhereTerm *whereScanInit(
```

將 like 轉換成 range query(wherecode.c):

```
/*
** If the most recently coded instruction is a constant range constraint
** (a string literal) that originated from the LIKE optimization, then
** set P3 and P5 on the OP_String opcode so that the string will be cast
** to a BLOB at appropriate times.
**
** The LIKE optimization trys to evaluate "x LIKE 'abc%'" as a range
** expression: "x>='ABC' AND x<'abd'".  But this requires that the range
** scan loop run twice, once for strings and a second time for BLOBs.
** The OP_String opcodes on the second pass convert the upper and lower
** bound string constants to blobs.  This routine makes the necessary changes
** to the OP_String opcodes for that to happen.
**
** Except, of course, if SQLITE_LIKE_DOESNT_MATCH_BLOBS is defined, then
** only the one pass through the string space is required, so this routine
** becomes a no-op.
*/
static void whereLikeOptimizationStringFixup(
  Vdbe *v,                /* prepared statement under construction */
  WhereLevel *pLevel,     /* The loop that contains the LIKE operator */
  WhereTerm *pTerm        /* The upper or lower bound just coded */
){
```

5.JOIN:

在 SQLite 中，join 主要由巢狀的 for loop 來完成，最簡單的 join 方式是以最左邊的 table 作為最外圈的 loop，最右邊的 table 作最內圈的 loop 來進行 join operation，然而這並不是一個不可改變的順序，也就是說，我們可以改變 loop 的順序來達到 join 的優化。Optimizer 在遇到 inner join 時會挑選最佳的執行順序來進行優化，但對於 outer join 因為不具有交換律的關係，所以不能任意更動執行順序，對於 cross join 則必定會照原本的順序進行 join，這是為了讓使用者可以強制 SQLite 進行特定順序的 join。

以下給出一個例子:

```
CREATE TABLE node(
   id INTEGER PRIMARY KEY,
   name TEXT
);
CREATE INDEX node_idx ON node(name);
CREATE TABLE edge(
   orig INTEGER REFERENCES node,
   dest INTEGER REFERENCES node,
   PRIMARY KEY(orig, dest)
);
CREATE INDEX edge_idx ON edge(dest,orig);
```

```
SELECT *
  FROM edge AS e,
       node AS n1,
       node AS n2
 WHERE n1.name = 'alice'
   AND n2.name = 'bob'
   AND e.orig = n1.id
   AND e.dest = n2.id;
```

Option 1:

```
foreach n1 where n1.name='alice' do:
  foreach n2 where n2.name='bob' do:
    foreach e where e.orig=n1.id and e.dest=n2.id
      return n1.*, n2.*, e.*
    end
  end
end
```

Option 2:

```
foreach n1 where n1.name='alice' do:
  foreach e where e.orig=n1.id do:
    foreach n2 where n2.id=e.dest and n2.name='bob' do:
      return n1.*, n2.*, e.*
    end
  end
end
```

這三張圖片分別是 table 的內容，query 的語句及 join 的執行方式，SQLite 可以輕鬆地利用內部的運算建立出所有可行的 join 順序並估計各種順序所需要的時間，因為每種執行方式所需的步驟數差異很大，所以 SQLite 會挑選出時間最快的作為執行的選擇。

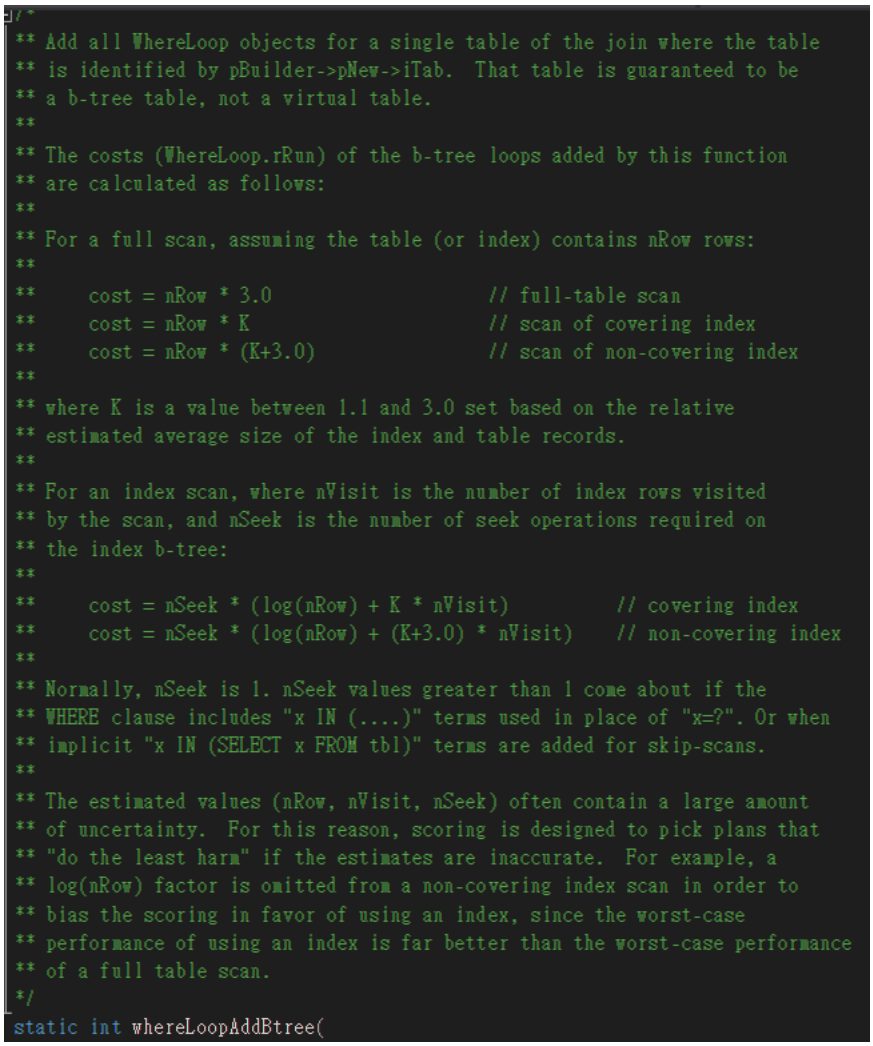
ii. explain the query planner as detail as possible:

(link: https://www.sqlite.org/queryplanner-ng.html)

Query planner 可以挑選出執行 SQL 效率最佳的演算法，對於數個小 table 的 join 可能只有幾種演算法，且效率差別不大，但對於更多的 table 的 join 則可能有數千甚至上萬種演算法可以執行 join，故我們需要 query 來幫助我們挑選出最快最適合的演算法來執行 query。

藉由 query planner 的幫助，資料庫的使用者可以更專注於下 SQL 而不必每次都想出最優的 SQL 語句，所以 query planner 可以使資料庫管理者的負擔降低許多，因此可以提高資料庫管理者的工作效率(因為時間可以全部用來想出 SQL 的寫法而不用親自進行優化)

在 where.c 中的 whereLoopAddBtreeIndex()可以挑選出最好的 index 來進行優化，whereLoopAddBtree()則是從所有的執行可能中挑出效率最好的行方式，利用計算每個執行方式的 cost，選出其中最小的 cost 的 query 方式，其定義及註解如下圖:



```
/*
** Add all WhereLoop objects for a single table of the join where the table
** is identified by pBuilder->pNew->iTab.  That table is guaranteed to be
** a b-tree table, not a virtual table.
**
** The costs (WhereLoop.rRun) of the b-tree loops added by this function
** are calculated as follows:
**
** For a full scan, assuming the table (or index) contains nRow rows:
**
**     cost = nRow * 3.0                    // full-table scan
**     cost = nRow * K                      // scan of covering index
**     cost = nRow * (K+3.0)                // scan of non-covering index
**
** where K is a value between 1.1 and 3.0 set based on the relative
** estimated average size of the index and table records.
**
** For an index scan, where nVisit is the number of index rows visited
** by the scan, and nSeek is the number of seek operations required on
** the index b-tree:
**
**     cost = nSeek * (log(nRow) + K * nVisit)          // covering index
**     cost = nSeek * (log(nRow) + (K+3.0) * nVisit)    // non-covering index
**
** Normally, nSeek is 1. nSeek values greater than 1 come about if the
** WHERE clause includes "x IN (....)" terms used in place of "x=?". Or when
** implicit "x IN (SELECT x FROM tbl)" terms are added for skip-scans.
**
** The estimated values (nRow, nVisit, nSeek) often contain a large amount
** of uncertainty.  For this reason, scoring is designed to pick plans that
** "do the least harm" if the estimates are inaccurate.  For example, a
** log(nRow) factor is omitted from a non-covering index scan in order to
** bias the scoring in favor of using an index, since the worst-case
** performance of using an index is far better than the worst-case performance
** of a full table scan.
*/
static int whereLoopAddBtree(
```

3. Describe the interaction between SQLite and OS (20 points)

a. describe how SQLite stores files in file system (format, pages, headers etc.) in detail

1. Database File

   (link: https://www.sqlite.org/fileformat2.html)

   完整的 database 通常包含磁碟上的一個 file，稱為”main database file”，在 transaction 進行時，SQLite 會將 transaction 所會使用到或改變到的資訊儲存在另一個名為”rollback journal”的 file 中。

   - Hot journals

     有時 transaction 結束前電腦就當機或者停止運作，這時我們會需要”rollback journals”來幫助我們恢復資料庫的資料，在這時我們稱”rollback journals”為”Hot journals”

   - Pages

     Main database file 由一個或多個 pages 組成，這些 page 的大小可以是 512-65536 中任意 2 的次方，但前提是每個 page 的大小必須是相同的

Page 的編號可以從 1 到 2^32-2，因此最小的 SQLite file 的大小是 512byte，最大理論上則可以到達 281TB)左右，但實際上在使用到那麼大的空間之前 SQLite 就會到達自己的極限。
在 pager.c 中，有提到如何對 pages 進行寫入的函數 sqlite3PagerWrite()會檢查一個 page 是否可以被寫入，其定義如下:

```
/*
** Mark a data page as writeable. This routine must be called before
** making changes to a page. The caller must check the return value
** of this function and be careful not to change any page data unless
** this routine returns SQLITE_OK.
**
** The difference between this function and pager_write() is that this
** function also deals with the special case where 2 or more pages
** fit on a single disk sector. In this case all co-resident pages
** must have been written to the journal file before returning.
**
** If an error occurs, SQLITE_NOMEM or an IO error code is returned
** as appropriate. Otherwise, SQLITE_OK.
*/
int sqlite3PagerWrite(PgHdr *pPg){
  Pager *pPager = pPg->pPager;
  assert( (pPg->flags & PGHDR_MMAP)==0 );
  assert( pPager->eState>=PAGER_WRITER_LOCKED );
  assert( assert_pager_state(pPager) );
  if( (pPg->flags & PGHDR_WRITEABLE)!=0 && pPager->dbSize>=pPg->pgno ){
    if( pPager->nSavepoint ) return subjournalPageIfRequired(pPg);
    return SQLITE_OK;
  }else if( pPager->errCode ){
    return pPager->errCode;
  }else if( pPager->sectorSize > (u32)pPager->pageSize ){
    assert( pPager->tempFile==0 );
    return pagerWriteLargeSector(pPg);
  }else{
    return pager_write(pPg);
  }
```

sqlite3PagerCommitPhaseOne()會將 file load 到 pager 中:

```
/*
** Sync the database file for the pager pPager. zSuper points to the name
** of a super-journal file that should be written into the individual
** journal file. zSuper may be NULL, which is interpreted as no
** super-journal (a single database transaction).
**
** This routine ensures that:
**
**   * The database file change-counter is updated,
**   * the journal is synced (unless the atomic-write optimization is used),
**   * all dirty pages are written to the database file,
**   * the database file is truncated (if required), and
**   * the database file synced.
**
** The only thing that remains to commit the transaction is to finalize
** (delete, truncate or zero the first part of) the journal file (or
** delete the super-journal file if specified).
**
** Note that if zSuper==NULL, this does not overwrite a previous value
** passed to an sqlite3PagerCommitPhaseOne() call.
**
** If the final parameter - noSync - is true, then the database file itself
** is not synced. The caller must call sqlite3PagerSync() directly to
** sync the database file before calling CommitPhaseTwo() to delete the
** journal file in this case.
*/
int sqlite3PagerCommitPhaseOne(
```

在 transaction 發生，database 已經被更新到最新的 state，會呼叫 Sqlite3PagerCommitPhaseTwo()，其定義如下:

```
/*
** When this function is called, the database file has been completely
** updated to reflect the changes made by the current transaction and
** synced to disk. The journal file still exists in the file-system
** though, and if a failure occurs at this point it will eventually
** be used as a hot-journal and the current transaction rolled back.
**
** This function finalizes the journal file, either by deleting,
** truncating or partially zeroing it, so that it cannot be used
** for hot-journal rollback. Once this is done the transaction is
** irrevocably committed.
**
** If an error occurs, an IO error code is returned and the pager
** moves into the error state. Otherwise, SQLITE_OK is returned.
*/
int sqlite3PagerCommitPhaseTwo(Pager *pPager){
```

- Database Header

  Database file 的前 100 個 bytes 是由 Database 的 Header 所構成的，這些欄位提供了關於整個 database 的資訊，包含資料庫 file 的大小、資料庫版本、資料庫格式、資料庫快取等等的資

訊，這些欄位資料提供了有關於整個 database 的大多資訊，可以幫助 SQLite 理解資料庫的內容，因為資料太多我將表格列於下圖:

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

2. Lock-byte page:
   為了 win95 作業系統所生，因為現在的作業系統不需要 file locking 的功能，所以沒有實際效果。
3. Free list:
   Freelist 包含了沒有被使用的 page 的列表，若有個 Page 的內容都被刪除，則他會被放到 Free list 中，若接下來有需要使用到新的 page 則可以從 Free list 中尋找空的 page。
4. B-tree pages(Format):
   包含 B-tree 相關的 pages，有 key 以及 data，在 SQLite 中有相對應的尋找 B-tree 中資料的演算法(存放於前面說過的 btree.c 中)，藉由 B-tree 的幫助，SQLite 可以在更短的時間內有效地找出資料，包含兩種 B-tree: Table B-trees and Index B-trees
   B-tree pages 包含兩種重要的資訊 key 及 data，order 的 B-tree 需要有 d 個 key(n <= d <= 2n)，在每個 page 中，若有 d 個 key 則有 k+1 個 pointer 指向更接近 leaf 的 pages(在上課中有教過)，在新增資料時可能會產生 key 的數量超過 2n 的情況，這時必須要產生新的 pages(也在上課教過)，同樣的也有刪除某些資料後會有合併後不需要某些 page 的狀況，這時也須要有相對應的 b-treeU 演算法來支援這樣的操作。

以下附上 B-tree page 的 header:

B-tree Page Header Format

Offset	Size	Description
0	1	The one-byte flag at offset 0 indicating the b-tree page type. • A value of 2 (0x02) means the page is an interior index b-tree page. • A value of 5 (0x05) means the page is an interior table b-tree page. • A value of 10 (0x0a) means the page is a leaf index b-tree page. • A value of 13 (0x0d) means the page is a leaf table b-tree page. Any other value for the b-tree page type is an error.
1	2	The two-byte integer at offset 1 gives the start of the first freeblock on the page, or is zero if there are no freeblocks.
3	2	The two-byte integer at offset 3 gives the number of cells on the page.
5	2	The two-byte integer at offset 5 designates the start of the cell content area. A zero value for this integer is interpreted as 65536.
7	1	The one-byte integer at offset 7 gives the number of fragmented free bytes within the cell content area.
8	4	The four-byte page number at offset 8 is the right-most pointer. This value appears in the header of interior b-tree pages only and is omitted from all other pages.

在 btree.c 有以下函數:

sqlite3BtreeOpen()這個 function 可以打開一個 btree 的 file:

```
/*
** Open a database file.
**
** zFilename is the name of the database file.  If zFilename is NULL
** then an ephemeral database is created.  The ephemeral database might
** be exclusively in memory, or it might use a disk-based memory cache.
** Either way, the ephemeral database will be automatically deleted
** when sqlite3BtreeClose() is called.
**
** If zFilename is ":memory:" then an in-memory database is created
** that is automatically destroyed when it is closed.
**
** The "flags" parameter is a bitmask that might contain bits like
** BTREE_OMIT_JOURNAL and/or BTREE_MEMORY.
**
** If the database is already opened in the same database connection
** and we are in shared cache mode, then the open will fail with an
** SQLITE_CONSTRAINT error.  We cannot allow two or more BtShared
** objects in the same database connection since doing so will lead
** to problems with locking.
*/
int sqlite3BtreeOpen(
  sqlite3_vfs *pVfs,      /* VFS to use for this b-tree */
  const char *zFilename,  /* Name of the file containing the BTree database *
  sqlite3 *db,            /* Associated database handle */
  Btree **ppBtree,        /* Pointer to new Btree object written here */
  int flags,              /* Options */
  int vfsFlags            /* Flags passed through to sqlite3_vfs.xOpen() */
```

sqlite3BtreeInsert()則和他的名字一樣可以將 record insert 到 btree 中:

```
/*
** Insert a new record into the BTree.  The content of the new record
** is described by the pX object.  The pCur cursor is used only to
** define what table the record should be inserted into, and is left
** pointing at a random location.
**
** For a table btree (used for rowid tables), only the pX.nKey value of
** the key is used. The pX.pKey value must be NULL.  The pX.nKey is the
** rowid or INTEGER PRIMARY KEY of the row.  The pX.nData,pData,nZero fields
** hold the content of the row.
**
** For an index btree (used for indexes and WITHOUT ROWID tables), the
** key is an arbitrary byte sequence stored in pX.pKey,nKey.  The
** pX.pData,nData,nZero fields must be zero.
**
** If the seekResult parameter is non-zero, then a successful call to
** sqlite3BtreeIndexMoveto() to seek cursor pCur to (pKey,nKey) has already
** been performed.  In other words, if seekResult!=0 then the cursor
** is currently pointing to a cell that will be adjacent to the cell
** to be inserted.  If seekResult<0 then pCur points to a cell that is
** smaller then (pKey,nKey).  If seekResult>0 then pCur points to a cell
** that is larger than (pKey,nKey).
**
** If seekResult==0, that means pCur is pointing at some unknown location.
** In that case, this routine must seek the cursor to the correct insertion
** point for (pKey,nKey) before doing the insertion.  For index btrees,
** if pX->nMem is non-zero, then pX->aMem contains pointers to the unpacked
** key values and pX->aMem can be used instead of pX->pKey to avoid having
** to decode the key.
*/
int sqlite3BtreeInsert(
```

sqlite3BtreeDelete()則是從 Btree 中刪除某個 record 的函數:

```
/*
** Delete the entry that the cursor is pointing to.
**
** If the BTREE_SAVEPOSITION bit of the flags parameter is zero, then
** the cursor is left pointing at an arbitrary location after the delete.
** But if that bit is set, then the cursor is left in a state such that
** the next call to BtreeNext() or BtreePrev() moves it to the same row
** as it would have been on if the call to BtreeDelete() had been omitted.
**
** The BTREE_AUXDELETE bit of flags indicates that is one of several deletes
** associated with a single table entry and its indexes.  Only one of those
** deletes is considered the "primary" delete.  The primary delete occurs
** on a cursor that is not a BTREE_FORDELETE cursor.  All but one delete
** operation on non-FORDELETE cursors is tagged with the AUXDELETE flag.
** The BTREE_AUXDELETE bit is a hint that is not used by this implementation,
** but which might be used by alternative storage engines.
*/
int sqlite3BtreeDelete(BtCursor *pCur, u8 flags){
```

sqlite3BtreeCursur()則是以遞迴的方式去查找 record 的所在:

```
int sqlite3BtreeCursor(
  Btree *p,                                   /* The btree */
  Pgno iTable,                                /* Root page of table to open */
  int wrFlag,                                 /* 1 to write. 0 read-only */
  struct KeyInfo *pKeyInfo,                   /* First arg to xCompare() */
  BtCursor *pCur                              /* Write new cursor here */
){
  if( p->sharable ){
    return btreeCursorWithLock(p, iTable, wrFlag, pKeyInfo, pCur);
  }else{
    return btreeCursor(p, iTable, wrFlag, pKeyInfo, pCur);
  }
}
```

b. describe how SQLite control file read, write, close etc

(link: https://www.sqlite.org/vfs.html)

在 SQLite 中，為了能在不同的作業系統上輕易地進行檔案的讀寫，並且增加 file 的可攜性，使用了名為 VFS(Virtual File System)的系統，使用 VFS 的好處是當我們要進行檔案的 IO 操作時，都可以使用 VFS 中固定的 method，在 SQLite 中，內建就支援的 operating system 有 UNIX 以及 Windows，因此使用這兩個作業系統的電腦可以簡單的利用 VFS 來運作 SQLite。

(link: https://www.sqlite.org/c3ref/io_methods.html)

藉由 VFS 的幫助，我們可以開啟 file 並且讀取或寫入 file，每個 VFS 都可以藉由 xopen()這個 method 開啟特定的 file，這時這個 VFS 就是像一個 database，我們可以藉由 VFS 來對 file 進行讀寫的操作。

```
int (*xOpen)(sqlite3_vfs*, sqlite3_filename zName, sqlite3_file*,
             int flags, int *pOutFlags);
```

上面的函數可以打開某個檔案，但對於檔案的讀寫，我們需要別的函數來完成，在資料夾中有很明顯的 os.c 檔案中就包含了用於讀寫的函數:

```
/*
** The following routines are convenience wrappers around methods
** of the sqlite3_file object.  This is mostly just syntactic sugar. All
** of this would be completely automatic if SQLite were coded using
** C++ instead of plain old C.
*/
void sqlite3OsClose(sqlite3_file *pId){
  if( pId->pMethods ){
    pId->pMethods->xClose(pId);
    pId->pMethods = 0;
  }
}
int sqlite3OsRead(sqlite3_file *id, void *pBuf, int amt, i64 offset){
  DO_OS_MALLOC_TEST(id);
  return id->pMethods->xRead(id, pBuf, amt, offset);
}
int sqlite3OsWrite(sqlite3_file *id, const void *pBuf, int amt, i64 offset){
  DO_OS_MALLOC_TEST(id);
  return id->pMethods->xWrite(id, pBuf, amt, offset);
}
```

這些函數是不分作業系統皆可以執行的，執行這些函數時會呼叫 sqlite3_file -> pMethods->xClose/xRead/xWrite 等等函數，在不同的作業系統中 VFS 會有不同的呼叫(根據 os_win.c 及 os_unix.c)，以達到在不同作業系統中都可以運作的功能。

4. Describe the concurrency control of SQLite (20 points)

a. describe how SQLite handle concurrency control (file locking, journal files etc.) in detail

(link: https://www.sqlite.org/lockingv3.html)

在 SQLite 中，locking 及 concurrency control 是由 pager 來進行的，藉由 pager 來達到上課中所提到 ACID(atomic, consistent, isolated, durable)的目的。

在 SQLite 中，每個 file 都有五種 state，分別是:

1.unlocked:file 目前未被讀或寫，任何程序都可以讀取或寫入 unlocked 的檔案。

2.shared:檔案正被一個或多個程序讀取(read)中，在 shared 狀態下別的程序可以讀取但不能寫入這個 file。

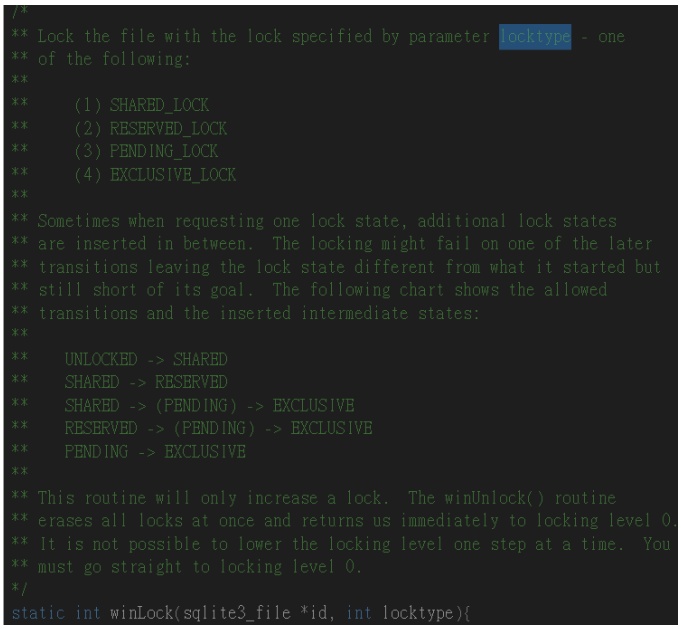
3.reserved:檔案可能會被寫入但目前只被讀取，在 reserved 狀況下 file 仍可以被其他程序以 shared 的方式 lock 住(但一個 file 同時間只能被一個程序 reserved)。

4.pending:檔案已經被某個程序 reserved，但因為將此檔案 reserved 起來的程序即將對檔案進行寫入，所以不能接受其他程序的 locking，在這個階段下檔案會等待在 reserved 時有 shared 此檔案的程序都讀取完成後，就會進入 exclusive 的狀態。

5.exclusive:此時只有 pending 的程序對此檔案進行 locking，所以這個程序可以進行所需要的寫入，等待寫入全部執行完畢後檔案會恢復成 unlocked 的狀態。

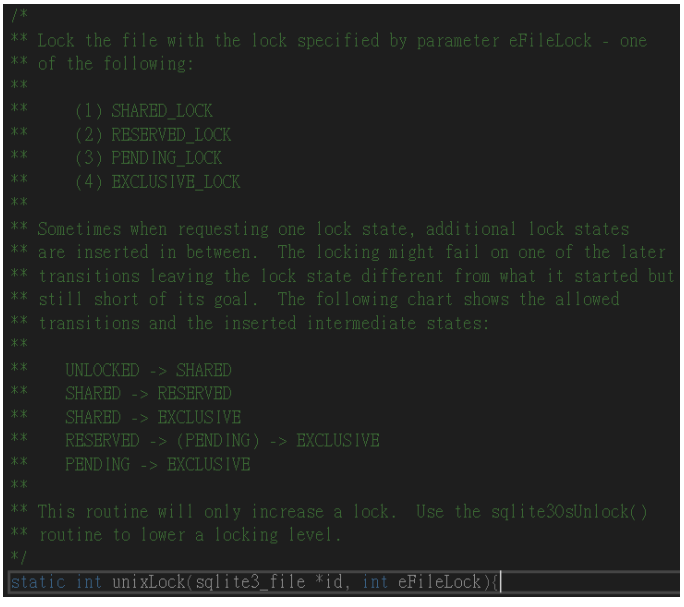
在 os interface 中有定義對這五種狀態的轉換所需要的函數，雖然在不同作業系統，他們都有各自用來指定 locking 狀態的函數，藉由這些函數 Database 可以給定 file 當前的 locking 狀態:

os_win.c:



```
/*
** Lock the file with the lock specified by parameter locktype - one
** of the following:
**
**     (1) SHARED_LOCK
**     (2) RESERVED_LOCK
**     (3) PENDING_LOCK
**     (4) EXCLUSIVE_LOCK
**
** Sometimes when requesting one lock state, additional lock states
** are inserted in between.  The locking might fail on one of the later
** transitions leaving the lock state different from what it started but
** still short of its goal.  The following chart shows the allowed
** transitions and the inserted intermediate states:
**
**    UNLOCKED -> SHARED
**    SHARED -> RESERVED
**    SHARED -> (PENDING) -> EXCLUSIVE
**    RESERVED -> (PENDING) -> EXCLUSIVE
**    PENDING -> EXCLUSIVE
**
** This routine will only increase a lock.  The winUnlock() routine
** erases all locks at once and returns us immediately to locking level 0.
** It is not possible to lower the locking level one step at a time.  You
** must go straight to locking level 0.
*/
static int winLock(sqlite3_file *id, int locktype){
```

os_unix.c:



```
/*
** Lock the file with the lock specified by parameter eFileLock - one
** of the following:
**
**     (1) SHARED_LOCK
**     (2) RESERVED_LOCK
**     (3) PENDING_LOCK
**     (4) EXCLUSIVE_LOCK
**
** Sometimes when requesting one lock state, additional lock states
** are inserted in between.  The locking might fail on one of the later
** transitions leaving the lock state different from what it started but
** still short of its goal.  The following chart shows the allowed
** transitions and the inserted intermediate states:
**
**    UNLOCKED -> SHARED
**    SHARED -> RESERVED
**    SHARED -> EXCLUSIVE
**    RESERVED -> (PENDING) -> EXCLUSIVE
**    PENDING -> EXCLUSIVE
**
** This routine will only increase a lock.  Use the sqlite3OsUnlock()
** routine to lower a locking level.
*/
static int unixLock(sqlite3_file *id, int eFileLock){
```

至於 journal file 的概念老師上課中就有提到，當我們對 database 進行 transcation 時，concurrency control 會確保每個 file 只被一個connection 寫入，以免發生要讀取檔案卻讀取到錯誤的 record 的結果，但這不能使 SQLite 具有恢復(rollback)的功能，也就是說，如果今天有connection

對某個 file 進行 update 到一半，server 突然斷電或是 connection 出了一些問題導致與 server 失去連結，update 的動作可能只完成了一半，這樣會造成相當嚴重的問題，舉例來說若 A 要轉給 B 十塊錢，但只執行完 A 扣十塊而沒有執行到 B 加十元的動作，總體的金額就會出問題，這就破壞了 ACID 中的 consistency，所以 SQLite 中也需要使用 rollback journal 來幫助進行資料的復原，今天任何寫入要發生時，原本在 file 中的資料會先被保留，要寫入的資料則會被寫入 journal file 中，若最後 transaction 被成功提交(commit)接下來才對 file 進行 update，若接下來出現需要進行復原或是取消 transaction 的動作，就可以利用 rollback journal 中的資料來進行還原，從 file 的 update 可以分出數種狀況，在 pager.c 中有提到會執行甚麼動作:

```
** List of state transitions and the C [function] that performs each:
**
**   OPEN              -> READER              [sqlite3PagerSharedLock]
**   READER            -> OPEN                [pager_unlock]
**
**   READER            -> WRITER_LOCKED       [sqlite3PagerBegin]
**   WRITER_LOCKED     -> WRITER_CACHEMOD     [pager_open_journal]
**   WRITER_CACHEMOD   -> WRITER_DBMOD        [syncJournal]
**   WRITER_DBMOD      -> WRITER_FINISHED     [sqlite3PagerCommitPhaseOne]
**   WRITER_***        -> READER              [pager_end_transaction]
**
**   WRITER_***        -> ERROR               [pager_error]
**   ERROR             -> OPEN                [pager_unlock]
**
```

我們可以注意到，在 file 被 locked 之後，pager 就會打開一個 journal file，之後才會進行寫入的操作。接下來我們來看看 pager.c 中 pager_open_journal()是怎麼宣告的:

```
/*
** This function is called at the start of every write transaction.
** There must already be a RESERVED or EXCLUSIVE lock on the database
** file when this routine is called.
**
** Open the journal file for pager pPager and write a journal header
** to the start of it. If there are active savepoints, open the sub-journal
** as well. This function is only used when the journal file is being
** opened to write a rollback log for a transaction. It is not used
** when opening a hot journal file to roll it back.
**
** If the journal file is already open (as it may be in exclusive mode),
** then this function just writes a journal header to the start of the
** already open file.
**
** Whether or not the journal file is opened by this function, the
** Pager.pInJournal bitvec structure is allocated.
**
** Return SQLITE_OK if everything is successful. Otherwise, return
** SQLITE_NOMEM if the attempt to allocate Pager.pInJournal fails, or
** an IO error code if opening or writing the journal file fails.
*/
static int pager_open_journal(Pager *pPager){
```

這邊就可以驗證前面所提到的 locking，因為只有需要 write 的時候才需要準備 journal file，所以註解中的一開始就寫到只有 RESERVED 及 EXCLUSIVE 兩種狀況會需要打開 journal file，在第三段中有提到，若 journal file 已經被打開，那 pager 只會寫入那個已經被打開的檔案的 journal header 中。

另外，還有一種特殊狀況下 rollback journal 會被稱為 hot journal，其需要滿足以下的條件:

1. journal 存在
2. journal 的大小>512bytes
3. journal header 不是 0 形式是良好的(well-formed)
4. super-journal 存在
5. 對於當前的 file，沒有其他的 reserved locking

仔細檢查上面的五個條件，我們可以發現 hot journal 的目的是為了讓 database 面臨崩潰或斷線等等莫名其妙的情況時的恢復機制，如果我們在寫入 file 時未完成寫入就遇到莫名中斷的時候，當前寫入的 journal file 就會被標記成 hot hournal，這樣做的目的是，當我們下次重新打開 database，SQLite 就會馬上去檢查被標記為 hot journal 的檔案，並且 hot journal 中的內容套用回資料庫中，以保持 database 的 consistency。

Journal file 是用來記錄寫入的過程，若成功 commit 則將 journal file 的變更存入原本的 file 中，hot journal 則是 journal file 異常終止的特例。

b. use examples to explain concurrency control/isolation in SQLite

i. when is the change done by an operation visible to other operations (which operations)

ii. when will nondeterministic happen (that is to say, in what condition we can not know what will happen in advance)

(我並沒有真正找到這題的答案在哪裡所以只能附上 chatgpt 的解答，抱歉)

(source:chatgpt)

i. 操作對其他操作的更改何時對其可見（哪些操作）：

在 SQLite 中，操作對其他操作的更改在特定的時間點對其可見，這取決於事務的隔離級別和完成操作的時機。以下是幾個例子：

1. 事務隔離級別：READ COMMITTED：

- 假設有兩個操作 A 和 B，在 READ COMMITTED 隔離級別下。
- 如果操作 A 修改了某一行的數據並提交事務，那麼操作 B 在事務開始之前不會看到操作 A 所做的更改。
- 但是，一旦操作 A 提交了事務，操作 B 在開始後的任何時間點都能看到操作 A 的更改。

2. 事務隔離級別：SERIALIZABLE：

- 假設有兩個操作 A 和 B，在 SERIALIZABLE 隔離級別下。

- 如果操作 A 修改了某一行的數據並提交事務，那麼操作 B 在事務開始之前和之後都無法看到操作 A 所做的更改。

- 在 SERIALIZABLE 隔離級別下，操作 B 必須等待操作 A 完成並提交事務後，才能看到操作 A 的更改。

根據不同的隔離級別，對於其他操作何時能看到變更有不同的規則和限制。

ii. 非確定性發生的條件（即，什麼條件下我們無法預先知道會發生什麼）：

在 SQLite 中，非確定性（Nondeterminism）通常與並發操作和競爭條件相關。以下是一個例子：

假設有兩個操作 A 和 B，它們同時修改相同的數據行。操作 A 檢查數據行的值，並根據結果進行更新，而操作 B 同樣也在同一時間對該數據行進行更新。

在這種情況下，由於兩個操作同時進行且相互競爭，無法確定哪個操作會先完成。這導致結果的非確定性，因為最終的結果取決於操作 A 和操作 B 執行的順序，並且我們無法預先知道哪個操作會先完成。

非確定性在

並發操作和共享數據的情況下可能發生，需要使用適當的並發控制機制（例如鎖定）來確保數據的一致性和可預測性。SQLite 提供了不同的並發控制機制和隔離級別，以幫助處理這些問題。

Code base exploration:

我選擇的是 component 中的 tokenizer，也就是將 SQL 分成 token 並送給 parser 進行分析的工具。利用 tokenizer 的幫助，SQLite 可以更精準的理解 SQL 的內容，也可以避免一些無關緊要的字元影響到整個 SQL 的解讀(比方說換行或是空白)。Tokenizer 的功能大多寫在 tokenize.c 中，接下來我會以 tokenize.c 中的程式碼來進行功能的解釋。

1. 在 21-59 行，因為需要隊後續會使用到的 aiClass 陣列做 binary search，所以必需要將字元對應到整數，所以這邊定義了一些字元到整數的 mapping。
2. 在 61-100 行中，則是根據所使用的字元編碼格式的不同(可能是 ASCII 或是 EBCDIC)，建立相對應的 aiClass[]。
3. 111-135 行則是將大寫轉換成小寫的轉換器，利用這樣的方式，才可以達到我們在下 SQL 時習慣的 case-insensitive 的功能。
4. 148 行 include 了 keywordhash.h 使得 tokenizer 可以利用其中的函數來判斷現在分出來的字元是 keyword 還是正常的字。
5. 167-187 行宣告了 idChar(x)的函數，可以判斷當前的字元 X 是不是可以做為 idetifier，若可以則回傳 true，否則回傳 false。
6. 197-214 行則是從字串中得到下一個 token 的函數，其中利用到了 sqlite3GetToken 以及 sqlite3ParserFallBack 兩個函數來幫忙完成。
7. 246-267 行則是判斷當前讀到的 over、window、filter 這三個詞是作為 identifier 還是 keyword，因為這些東西的判斷要根據前後文才能決定，所以需要特別拉出來討論。
8. 273-562 行利用 sqlite3GetToken 這個函式來計算當前指標指到的 token 的長度。
9. 567-718 行宣告了 sqlite3RunParser 這個函數，他可以對特定的 SQL 語句執行 parser，並且回傳執行 parser 時遇到的錯誤數量。
10. 726-730 行的函數可以檢查當前的 SQL 語句是否由一個 identifier 結尾，若是則將一個 space 貼到 SQL 的尾端。
11. 737-852 行則是對一個 SQL 語句進行標準化(normalized)的函數。

從上面的這些函數可以發現，整個 tokenizer 所要做的事情其實並不難理解，就是將我們所輸入的 SQL 語句轉換成 SQLite 可以更容易理解的格式，我們也許很習慣在使用 SQL 時大小寫都可以的這個原則，但應該很少有人實際去知道說每個 DBMS 到底是怎麼時做這樣的功能的。另外，SQL 之所以可以跨行做輸入，也是因為有 parser 的幫助，所以每當我們出現了換行或空白這些無謂的字元時，tokenizer 都會幫我們將我們輸入的 SQL 整理成最標準的 SQL 語句，這大大的增加了我們使用 SQL 的便利性，也可以理解到 tokenizer 是如何來辨識輸入的字串是 identifier 還是 keyword，是個讓 SQLite 的使用更為便利不可或缺的工具。