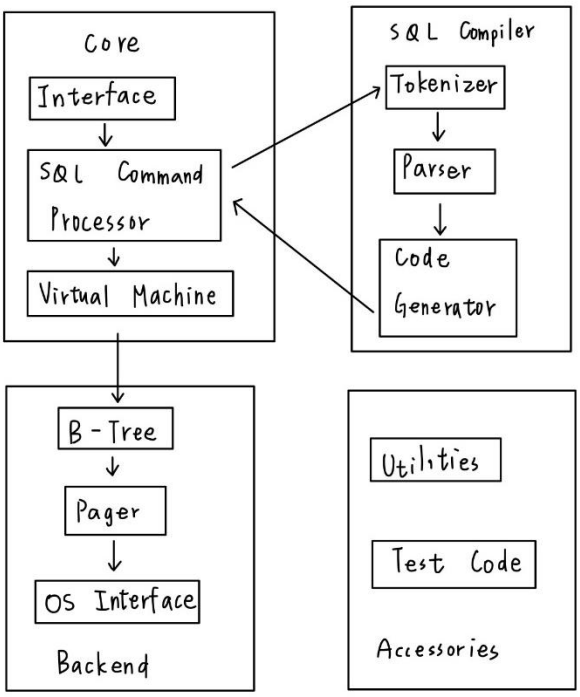
2-A

1. (a)



(b) • Interface

大多數 C 語言的 interface 可以在 source file 中的 main.c、legacy.c 及 vdbeapi.c 中找到，雖然有一些 routines 分散在其他 files，但它們可以 access to data structures with file scope

• Tokenizer

當執行一個包含 SQL statements 的 string 時，要先把這個 string 傳遞給 tokenizer。Tokenizer 會把 string 分割成很多個 tokens，將這些 tokens 一個個傳遞給 parser。Tokenizer 位於 tokenize.c 中。

• Parser

Parser 根據上下文賦予 token 意義。SQLite 的 parser 使用 Lemon parser generator，Lemon 做的工作與 YACC/BISON 相同，但它使用不同的輸入句法使其更不易出錯。Lemon 產生可重入並且線程安全的 generator。Lemon 定義了 non-terminal destructor 的概念，因此遇到語法錯誤時不會有 memory leak。驅動 Lemon 的 file 在 parse.y 中。

• Code Generator

parser 把 token 組裝成 parse tree 後，就用 code generator 去分析 parse tree 以及產生 bytecode 去執行 SQL statement。Code generator 包含許多 file：attach.c、auth.c、build.c、delete.c、expr.c、insert.c、pragma.c、select.c、trigger.c、update.c、vacuum.c 和 where.c。expr.c 處理 SQL 中

表達式的代碼生成。where.c 處理 SELECT、UPDATE 和 DELETE 語句中 WHERE 的代碼生成。文件 attach.c、delete.c、insert.c、select.c、trigger.c、update.c 和 vacuum.c 處理同名 SQL 語句的代碼生成。其他 SQL 語句的代碼由 build.c 生成。File auth.c 實現 sqlite3_set_authorizer() 的功能。

- Bytecode Engine

Code generator 生成的 bytecode program 由 virtual machine 來執行，virtual machine 整個被包含在一個單一的 source file vdbe.c 中. vdbe.h 定義一個介於 virtual machine 和剩下的 SQLite library 的 interface，vdbeInt.h 定義 private the virtual machine 的 structures 及 interfaces。vdbe*.c 是 virtual machine 的幫助. vdbeaux.c 包含了被 virtual machine 使用的 utilities 以及被剩下的 library 用來建造 VM programs 的 interface modules。vdbeapi.c 包含 sqlite3_bind_int()、sqlite3_step()等 interfaces。各自的值被存在叫做 internal object 中，由 vdbemem.c implement。

- B-Tree

SQLite database 使用 B-tree 的形式存在 disk，B-樹的 implement 位於 btree.c 中。database 中的每個 table 和 index 使用一棵單獨的 B-tree，所有的 B-tree 存放在同一個 disk file 中。B-tree 子系統的 interface 以及剩下的 SQLite library 定義於 btree.h 中。

- Page Cache

B-tree module 以固定大小的 page 從 disk 請求訊息，默認的 page size 是 4096 bytes，但是可以是介於 512 和 65536 的 2 的冪次方。Page cache 負責讀、寫和緩存 pages。Page cache 還提供 rollback 和 atomic commit abstraction，並且管理 database file 的鎖定。B-tree driver 從 page cahce 中請求特定的 page，在想修改頁面、提交或回滾當前修改時，它也會通知 page cache。Page cache 處理所有麻煩的細節，以確保請求能夠快速、安全而有效地被處理。主要的 Page cache 由 pager.c implement。

- OS Interface

為了在 OS 之間提供移植性，SQLite 使用 VFS，每個 VFS 提供在 disk 上開檔、讀、寫及關檔的方法，及用於其他 OS-specific task，如查找當前時間或獲取隨機性以初始化內建的 pseudo-random number generator。SQLite 提供 VFSes 給 unix ，位於 os_unix.c file，以及 Windows，位於 os_win.c file。

- Utilities

Memory allocation, caseless string comparison routines, portable text-to-number conversion routines 位於 util.c 中。parser 使用的 symbol tables 由 hash tables maintain，可在 hash.c 中找到。utf.c 包含 Unicode conversion subroutines。printf.c 包含 SQLite 自己的 printf() implement。

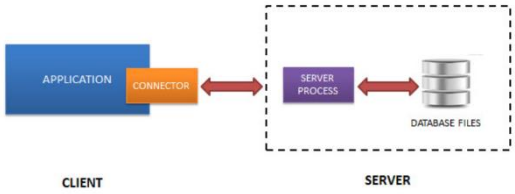
random.c 包含 SQLite 自己的 pseudo-random number generator。

- Test Code

Source file 中名字以 test 開頭的文件只用於測試，不包含在 standard library 中

(c)

- SQLite 是 severless 的，直接重 disk 上存取 database file 進行 read、write; 而 mysql 則是需要有 server 與 client，需要有 TCP/IP protocol to send and receive requests，為 client/server architecture



- SQLite 支援的 datatypes: Blob, Integer, Null, Text, Real;
  MySQL 支援的 datatypes: Tinyint, Smallint, Medi umint, Int, Bigint, Double, Float, Real, Decimal, Double precision, Numeric, Timestamp, Date, Datetime,Char, Varchar, Year, Tinytext, Tinyblob, Blob, Text, MediumBlob, MediumText, Enum, Set, Longblob, Longtext.
- SQLite 沒有特別的 user management functionality 因此不適合同時有 multiple user access。MySQL 有 well-constructed user management system，因此可以處理 multiple users 及給予不同等級的權限。
- SQLite 沒有 inbuilt authentication mechanism，因此任何人都可以 access database files。而 MySQL 有許多 inbuilt security features，包括 authentication with a username、 password 及 SSH.

2. (a)

- Tokenizer

  當執行一個包含 SQL statements 的 string 時，要先把這個 string 傳遞給 tokenizer。Tokenizer 會把 string 分割成很多個 tokens，將這些 tokens 一個個傳遞給 parser。Tokenizer 位於 tokenize.c 中。

- Parser

  Parser 根據上下文賦予 token 意義。SQLite 的 parser 使用 Lemon parser generator，Lemon 做的工作與 YACC/BISON 相同，但它使用不同的輸入句法使其更不易出錯。Lemon 產生可重入並且線程安全的 generator。Lemon 定義了 non-terminal destructor 的概念，因此遇到語法錯誤時不會有 memory leak。驅動 Lemon 的 file 在 parse.y 中。

- Code Generator

  parser 把 token 組裝成 parse tree 後，就用 code generator 去分析 parse tree 以及產生 bytecode 去執行 SQL statement。Code generator 包含許

多 file：attach.c、auth.c、build.c、delete.c、expr.c、insert.c、pragma.c、select.c、trigger.c、update.c、vacuum.c 和 where.c。expr.c 處理 SQL 中表達式的代碼生成。where.c 處理 SELECT、UPDATE 和 DELETE 語句中 WHERE 的代碼生成。文件 attach.c、delete.c、insert.c、select.c、trigger.c、update.c 和 vacuum.c 處理同名 SQL 語句的代碼生成。其他 SQL 語句的代碼由 build.c 生成。File auth.c 實現 sqlite3_set_authorizer()的功能。

(b)(i) • where

Sqlite3WhereBegin()分析 where 子句

```
WhereInfo *sqlite3WhereBegin(
  Parse *pParse,          /* The parser context */
  SrcList *pTabList,      /* FROM clause: A list of all tables to be scanned */
  Expr *pWhere,           /* The WHERE clause */
  ExprList *pOrderBy,     /* An ORDER BY (or GROUP BY) clause, or NULL */
  ExprList *pResultSet,   /* Query result set.  Req'd for DISTINCT */
  u16 wctrlFlags,         /* The WHERE_* flags defined in sqliteInt.h */
  int iAuxArg             /* If WHERE_OR_SUBCLAUSE is set, index cursor number
                          ** If WHERE_USE_LIMIT, then the limit amount */
)
```

WhereClauseInit()初始化 whereClause

```
void sqlite3WhereClauseInit(
  WhereClause *pWC,        /* The WhereClause to be initialized */
  WhereInfo *pWInfo        /* The WHERE processing context */
){
  pWC->pWInfo = pWInfo;
  pWC->hasOr = 0;
  pWC->pOuter = 0;
  pWC->nTerm = 0;
  pWC->nSlot = ArraySize(pWC->aStatic);
  pWC->a = pWC->aStatic;
}
```

Sqlite3WhereSplit()根據 where 中的 op(AND 及 OR)將每個條件分開來放入 slot[]中，slot 大小會增加到可以存放所有條件的 size

```
 WHERE  a=='hello' AND coalesce(b,11)<10 AND (c+12!=d OR c==22)
        \_______/     \________________/     \________________/
         slot[0]           slot[1]               slot[2]
```

```
void sqlite3WhereSplit(WhereClause *pWC, Expr *pExpr, u8 op){
  Expr *pE2 = sqlite3ExprSkipCollateAndLikely(pExpr);
  pWC->op = op;
  if( pE2==0 ) return;
  if( pE2->op!=op ){
    whereClauseInsert(pWC, pExpr, 0);
  }else{
    sqlite3WhereSplit(pWC, pE2->pLeft, op);
    sqlite3WhereSplit(pWC, pE2->pRight, op);
  }
}
```

Generate code for the start of the iLevel-th loop in the WHERE clause

```
Bitmask sqlite3WhereCodeOneLoopStart(
  Parse *pParse,       /* Parsing context */
  Vdbe *v,             /* Prepared statement under construction */
  WhereInfo *pWInfo,   /* Complete information about the WHERE clause */
  int iLevel,          /* Which level of pWInfo->a[] should be coded */
  WhereLevel *pLevel,  /* The current level pointer */
  Bitmask notReady     /* Which tables are currently available */
)
```

Generate code to construct the Index object for an automatic index

```
static void constructAutomaticIndex(
  Parse *pParse,              /* The parsing context */
  WhereClause *pWC,           /* The WHERE clause */
  struct SrcList_item *pSrc,  /* The FROM clause term to get the next index */
  Bitmask notReady,           /* Mask of cursors that are not available */
  WhereLevel *pLevel          /* Write new index here */
)
/* Create the automatic index */
  assert( pLevel->iIdxCur>=0 );
  pLevel->iIdxCur = pParse->nTab++;
  sqlite3VdbeAddOp2(v, OP_OpenAutoindex, pLevel->iIdxCur, nKeyCol+1);
  sqlite3VdbeSetP4KeyInfo(pParse, pIdx);
  VdbeComment((v, "for %s", pTable->zName));
```

Call exprAnalyze on all terms in a WHERE clause.

```
void sqlite3WhereExprAnalyze(
  SrcList *pTabList,       /* the FROM clause */
  WhereClause *pWC         /* the WHERE clause to be analyzed */
```

```
){
  int i;
  for(i=pWC->nTerm-1; i>=0; i--){
    exprAnalyze(pTabList, pWC, i);
  }
}
```

Return TRUE if the given operator is one of the operators that is allowed for an indexable WHERE clause term. The allowed operators are "=", "<", ">", "<=", ">=", "IN", "IS", and "IS NULL"

```
static int allowedOp(int op){
  assert( TK_GT>TK_EQ && TK_GT<TK_GE );
  assert( TK_LT>TK_EQ && TK_LT<TK_GE );
  assert( TK_LE>TK_EQ && TK_LE<TK_GE );
  assert( TK_GE==TK_EQ+4 );
  return op==TK_IN || (op>=TK_EQ && op<=TK_GE) || op==TK_ISNULL || op==TK_IS;
}
```

- between

  當發生 expr1 BETWEEN expr2 AND expr3 時，添加兩個 virtual terms

  expr1 > = expr2 AND expr1 <= expr3

  virtual terms 僅用於分析，不會導致任何 byte-code 產生。

```
/* If a term is the BETWEEN operator, create two new virtual terms
 ** that define the range that the BETWEEN implements.  For example:
 **
 **      a BETWEEN b AND c
 **
 ** is converted into:
 **
 **      (a BETWEEN b AND c) AND (a>=b) AND (a<=c)
 **
 ** The two new terms are added onto the end of the WhereClause object.
 ** The new terms are "dynamic" and are children of the original BETWEEN
 ** term.  That means that if the BETWEEN term is coded, the children are
 ** skipped.  Or, if the children are satisfied by an index, the original
 ** BETWEEN term is skipped.
 */
 else if( pExpr->op==TK_BETWEEN && pWC->op==TK_AND ){
   ExprList *pList = pExpr->x.pList;
```

```
  int i;
  static const u8 ops[] = {TK_GE, TK_LE};
  assert( pList!=0 );
  assert( pList->nExpr==2 );
  for(i=0; i<2; i++){
    Expr *pNewExpr;
    int idxNew;
    pNewExpr = sqlite3PExpr(pParse, ops[i],
                            sqlite3ExprDup(db, pExpr->pLeft, 0),
                            sqlite3ExprDup(db, pList->a[i].pExpr, 0));
    transferJoinMarkings(pNewExpr, pExpr);
    idxNew = whereClauseInsert(pWC, pNewExpr, TERM_VIRTUAL|TERM_DYNAMIC);
    testcase( idxNew==0 );
    exprAnalyze(pSrc, pWC, idxNew);
    pTerm = &pWC->a[idxTerm];
    markTermAsChild(pWC, idxNew, idxTerm);
  }
}
```

- like

  在 like 的右邊必須是 string literal 或是 a parameter bound to a string literal 且不以 wildcard character 開頭，在 like 的左邊必須是 indexed column 的名字且具有 TEXT affinity，如果使用 ESCAPE，則 ESCAPE character 必須為 ASCII 或 a single-byte character in UTF-8。case_sensitive_like 可用來區分大小寫

  The LIKE optimization might occur if the column named on the left of the operator is indexed using the built-in BINARY collating sequence and case_sensitive_like is turned on. Or the optimization might occur if the column is indexed using the built-in NOCASE collating sequence and the case_sensitive_like mode is off. These are the only two combinations under which LIKE operators will be optimized. 如果滿足了條件，可以增加一些條件來減少 LIKE 的查詢範圍

```
/* Add constraints to reduce the search space on a LIKE or GLOB
 ** operator.
 **
 ** A like pattern of the form "x LIKE 'aBc%'" is changed into constraints
 **
 **          x>='ABC' AND x<'abd' AND x LIKE 'aBc%'
 **
```

```
** The last character of the prefix "abc" is incremented to form the
** termination condition "abd".  If case is not significant (the default
** for LIKE) then the lower-bound is made all uppercase and the upper-
** bound is made all lowercase so that the bounds also work when comparing
** BLOBs.
*/
if( pWC->op==TK_AND
 && isLikeOrGlob(pParse, pExpr, &pStr1, &isComplete, &noCase)
){
  Expr *pLeft;       /* LHS of LIKE/GLOB operator */
  Expr *pStr2;       /* Copy of pStr1 - RHS of LIKE/GLOB operator */
  Expr *pNewExpr1;
  Expr *pNewExpr2;
  int idxNew1;
  int idxNew2;
  const char *zCollSeqName;     /* Name of collating sequence */
  const u16 wtFlags = TERM_LIKEOPT | TERM_VIRTUAL | TERM_DYNAMIC;

  pLeft = pExpr->x.pList->a[1].pExpr;
  pStr2 = sqlite3ExprDup(db, pStr1, 0);

  /* Convert the lower bound to upper-case and the upper bound to
  ** lower-case (upper-case is less than lower-case in ASCII) so that
  ** the range constraints also work for BLOBs
  */
  if( noCase && !pParse->db->mallocFailed ){
    int i;
    char c;
    pTerm->wtFlags |= TERM_LIKE;
    for(i=0; (c = pStr1->u.zToken[i])!=0; i++){
      pStr1->u.zToken[i] = sqlite3Toupper(c);
      pStr2->u.zToken[i] = sqlite3Tolower(c);
    }
  }

  if( !db->mallocFailed ){
    u8 c, *pC;       /* Last character before the first wildcard */
    pC = (u8*)&pStr2->u.zToken[sqlite3Strlen30(pStr2->u.zToken)-1];
```

```
      c = *pC;
      if( noCase ){
        /* The point is to increment the last character before the first
        ** wildcard.  But if we increment '@', that will push it into the
        ** alphabetic range where case conversions will mess up the
        ** inequality.  To avoid this, make sure to also run the full
        ** LIKE on all candidate expressions by clearing the isComplete flag
        */
        if( c=='A'-1 ) isComplete = 0;
        c = sqlite3UpperToLower[c];
      }
      *pC = c + 1;
    }
    zCollSeqName = noCase ? "NOCASE" : sqlite3StrBINARY;
    pNewExpr1 = sqlite3ExprDup(db, pLeft, 0);
    pNewExpr1 = sqlite3PExpr(pParse, TK_GE,
           sqlite3ExprAddCollateString(pParse,pNewExpr1,zCollSeqName),
           pStr1);
    transferJoinMarkings(pNewExpr1, pExpr);
    idxNew1 = whereClauseInsert(pWC, pNewExpr1, wtFlags);
    testcase( idxNew1==0 );
    exprAnalyze(pSrc, pWC, idxNew1);
    pNewExpr2 = sqlite3ExprDup(db, pLeft, 0);
    pNewExpr2 = sqlite3PExpr(pParse, TK_LT,
           sqlite3ExprAddCollateString(pParse,pNewExpr2,zCollSeqName),
           pStr2);
    transferJoinMarkings(pNewExpr2, pExpr);
    idxNew2 = whereClauseInsert(pWC, pNewExpr2, wtFlags);
    testcase( idxNew2==0 );
    exprAnalyze(pSrc, pWC, idxNew2);
    pTerm = &pWC->a[idxTerm];
    if( isComplete ){
      markTermAsChild(pWC, idxNew1, idxTerm);
      markTermAsChild(pWC, idxNew2, idxTerm);
    }
  }
```

(ii) SQL 是一種 declarative language，而非 procedural language，當下一個 SQL

statement 可以有許多種 algorithm 來得到正確的答案，query planner 會嘗試找出最有效率的 algorithm 來完成該 SQL statement，但前提為 query planner 需要有由 programmer 所添加的 indices 才能運作。

```
** 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.
```

```
if( pSrc->pIBIndex ){
    /* An INDEXED BY clause specifies a particular index to use */
    pProbe = pSrc->pIBIndex;
  }else if( !HasRowid(pTab) ){
    pProbe = pTab->pIndex;
  }else{
    /* There is no INDEXED BY clause.  Create a fake Index object in local
    ** variable sPk to represent the rowid primary key index.  Make this
    ** fake index the first in a chain of Index objects with all of the real
    ** indices to follow */
    Index *pFirst;                  /* First of real indices on the table */
```

```
    memset(&sPk, 0, sizeof(Index));
    sPk.nKeyCol = 1;
    sPk.nColumn = 1;
    sPk.aiColumn = &aiColumnPk;
    sPk.aiRowLogEst = aiRowEstPk;
    sPk.onError = OE_Replace;
    sPk.pTable = pTab;
    sPk.szIdxRow = pTab->szTabRow;
    sPk.idxType = SQLITE_IDXTYPE_IPK;
    aiRowEstPk[0] = pTab->nRowLogEst;
    aiRowEstPk[1] = 0;
    pFirst = pSrc->pTab->pIndex;
    if( pSrc->fg.notIndexed==0 ){
      /* The real indices of the table are only considered if the
      ** NOT INDEXED qualifier is omitted from the FROM clause */
      sPk.pNext = pFirst;
    }
    pProbe = &sPk;
  }
  rSize = pTab->nRowLogEst;
  rLogSize = estLog(rSize);

#ifndef SQLITE_OMIT_AUTOMATIC_INDEX
  /* Automatic indexes */
  if( !pBuilder->pOrSet      /* Not part of an OR optimization */
   && (pWInfo->wctrlFlags & WHERE_OR_SUBCLAUSE)==0
   && (pWInfo->pParse->db->flags & SQLITE_AutoIndex)!=0
   && pSrc->pIBIndex==0      /* Has no INDEXED BY clause */
   && !pSrc->fg.notIndexed   /* Has no NOT INDEXED clause */
   && HasRowid(pTab)         /* Not WITHOUT ROWID table. (FIXME: Why not?) */
   && !pSrc->fg.isCorrelated /* Not a correlated subquery */
   && !pSrc->fg.isRecursive  /* Not a recursive common table expression. */
  ){
    /* Generate auto-index WhereLoops */
    WhereTerm *pTerm;
    WhereTerm *pWCEnd = pWC->a + pWC->nTerm;
    for(pTerm=pWC->a; rc==SQLITE_OK && pTerm<pWCEnd; pTerm++){
      if( pTerm->prereqRight & pNew->maskSelf ) continue;
```

```
      if( termCanDriveIndex(pTerm, pSrc, 0) ){
        pNew->u.btree.nEq = 1;
        pNew->nSkip = 0;
        pNew->u.btree.pIndex = 0;
        pNew->nLTerm = 1;
        pNew->aLTerm[0] = pTerm;
```

SQLite 亦可用 indices 去完成 the ORDER BY clauses 因此 indices 可用來加快 sorting 及 searching 的速度，但當沒有合適的 indices 時，ORDER BY clauses 就必須個別進行 sort

```
** We say the WhereLoop is "order-distinct" if the set of columns from
** that WhereLoop that are in the ORDER BY clause are different for every
** row of the WhereLoop.  Every one-row WhereLoop is automatically
** order-distinct.   A WhereLoop that has no columns in the ORDER BY clause
** is not order-distinct. To be order-distinct is not quite the same as being
** UNIQUE since a UNIQUE column or index can have multiple rows that
** are NULL and NULL values are equivalent for the purpose of order-distinct.
** To be order-distinct, the columns must be UNIQUE and NOT NULL.
**
** The rowid for a table is always UNIQUE and NOT NULL so whenever the
** rowid appears in the ORDER BY clause, the corresponding WhereLoop is
** automatically order-distinct.
*/
```

```
/* IN terms are only valid for sorting in the ORDER BY LIMIT
        ** optimization, and then only if they are actually used
        ** by the query plan */
        assert( wctrlFlags & WHERE_ORDERBY_LIMIT );
        for(j=0; j<pLoop->nLTerm && pTerm!=pLoop->aLTerm[j]; j++){}
        if( j>=pLoop->nLTerm ) continue;
      }
      if( (pTerm->eOperator&(WO_EQ|WO_IS))!=0 && pOBExpr->iColumn>=0 ){
        Parse *pParse = pWInfo->pParse;
        CollSeq *pColl1 = sqlite3ExprNNCollSeq(pParse, pOrderBy->a[i].pExpr);
        CollSeq *pColl2 = sqlite3ExprCompareCollSeq(pParse, pTerm->pExpr);
        assert( pColl1 );
        if( pColl2==0 || sqlite3StrICmp(pColl1->zName, pColl2->zName) ){
          continue;
```

```
      }
      testcase( pTerm->pExpr->op==TK_IS );
    }
    obSat |= MASKBIT(i);
  }

  if( (pLoop->wsFlags & WHERE_ONEROW)==0 ){
    if( pLoop->wsFlags & WHERE_IPK ){
      pIndex = 0;
      nKeyCol = 0;
      nColumn = 1;
    }else if( (pIndex = pLoop->u.btree.pIndex)==0 || pIndex->bUnordered ){
      return 0;
    }else{
      nKeyCol = pIndex->nKeyCol;
      nColumn = pIndex->nColumn;
      assert( nColumn==nKeyCol+1 || !HasRowid(pIndex->pTable) );
      assert( pIndex->aiColumn[nColumn-1]==XN_ROWID
                          || !HasRowid(pIndex->pTable));
      isOrderDistinct = IsUniqueIndex(pIndex)
                          && (pLoop->wsFlags & WHERE_SKIPSCAN)==0;
    }

    /* Loop through all columns of the index and deal with the ones
    ** that are not constrained by == or IN.
    */
    rev = revSet = 0;
    distinctColumns = 0;
    for(j=0; j<nColumn; j++){
      u8 bOnce = 1; /* True to run the ORDER BY search loop */

      assert( j>=pLoop->u.btree.nEq
          || (pLoop->aLTerm[j]==0)==(j<pLoop->nSkip)
      );
      if( j<pLoop->u.btree.nEq && j>=pLoop->nSkip ){
        u16 eOp = pLoop->aLTerm[j]->eOperator;

/* Get the column number in the table (iColumn) and sort order
```

```
** (revIdx) for the j-th column of the index.
*/
if( pIndex ){
  iColumn = pIndex->aiColumn[j];
  revIdx = pIndex->aSortOrder[j] & KEYINFO_ORDER_DESC;
  if( iColumn==pIndex->pTable->iPKey ) iColumn = XN_ROWID;
}else{
  iColumn = XN_ROWID;
  revIdx = 0;
}

/* An unconstrained column that might be NULL means that this
** WhereLoop is not well-ordered
*/
if( isOrderDistinct
 && iColumn>=0
 && j>=pLoop->u.btree.nEq
 && pIndex->pTable->aCol[iColumn].notNull==0
){
  isOrderDistinct = 0;
}

/* Find the ORDER BY term that corresponds to the j-th column
** of the index and mark that ORDER BY term off
*/
isMatch = 0;
for(i=0; bOnce && i<nOrderBy; i++){
  if( MASKBIT(i) & obSat ) continue;
  pOBExpr = sqlite3ExprSkipCollateAndLikely(pOrderBy->a[i].pExpr);
  testcase( wctrlFlags & WHERE_GROUPBY );
  testcase( wctrlFlags & WHERE_DISTINCTBY );
  if( (wctrlFlags & (WHERE_GROUPBY|WHERE_DISTINCTBY))==0 ) bOnce = 0;
  if( iColumn>=XN_ROWID ){
    if( pOBExpr->op!=TK_COLUMN ) continue;
    if( pOBExpr->iTable!=iCur ) continue;
    if( pOBExpr->iColumn!=iColumn ) continue;
  }else{
    Expr *pIdxExpr = pIndex->aColExpr->a[j].pExpr;
```

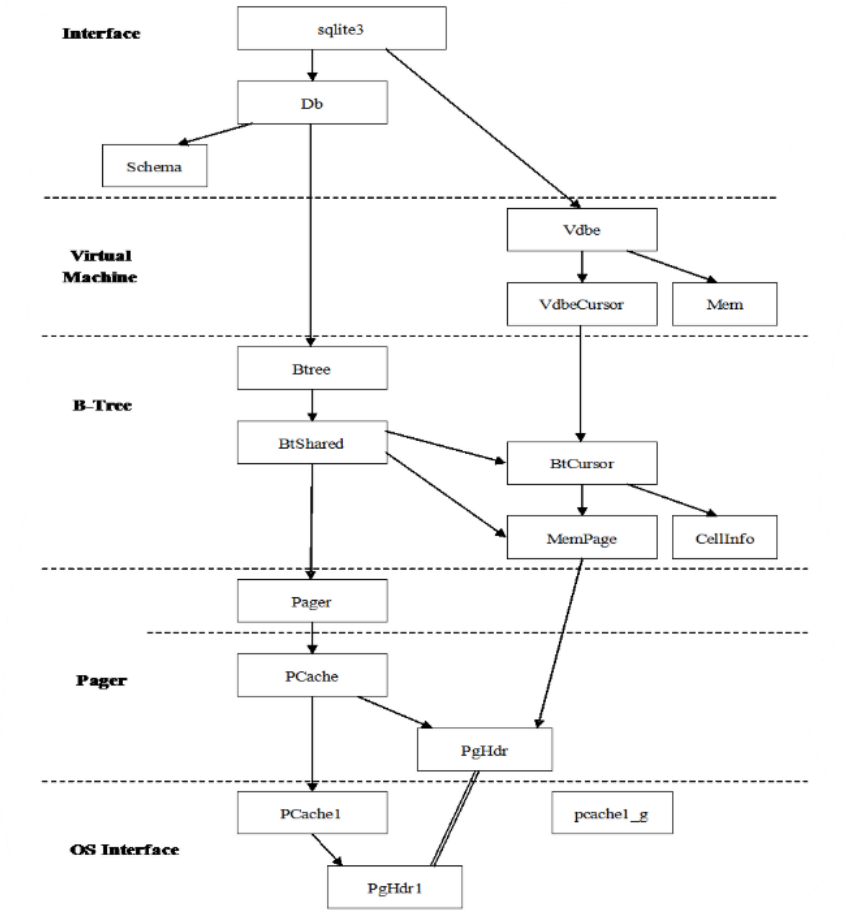
```
          if( sqlite3ExprCompareSkip(pOBExpr, pIdxExpr, iCur) ){
            continue;
          }
        }
        if( iColumn!=XN_ROWID ){
          pColl = sqlite3ExprNNCollSeq(pWInfo->pParse, pOrderBy->a[i].pExpr);
          if( sqlite3StrICmp(pColl->zName, pIndex->azColl[j])!=0 ) continue;
        }
        if( wctrlFlags & WHERE_DISTINCTBY ){
          pLoop->u.btree.nDistinctCol = j+1;
        }
        isMatch = 1;
        break;
      }
      if( isMatch && (wctrlFlags & WHERE_GROUPBY)==0 ){
        /* Make sure the sort order is compatible in an ORDER BY clause.
        ** Sort order is irrelevant for a GROUP BY clause. */
        if( revSet ){
          if( (rev ^ revIdx)!=(pOrderBy->a[i].sortFlags&KEYINFO_ORDER_DESC) ){
            isMatch = 0;
          }
        }else{
          rev = revIdx ^ (pOrderBy->a[i].sortFlags & KEYINFO_ORDER_DESC);
          if( rev ) *pRevMask |= MASKBIT(iLoop);
          revSet = 1;
        }
      }
      if( isMatch && (pOrderBy->a[i].sortFlags & KEYINFO_ORDER_BIGNULL) ){
        if( j==pLoop->u.btree.nEq ){
          pLoop->wsFlags |= WHERE_BIGNULL_SORT;
        }else{
          isMatch = 0;
        }
      }
      if( isMatch ){
        if( iColumn==XN_ROWID ){
          testcase( distinctColumns==0 );
          distinctColumns = 1;
```

```
        }
        obSat |= MASKBIT(i);
      }else{
        /* No match found */
        if( j==0 || j<nKeyCol ){
          testcase( isOrderDistinct!=0 );
          isOrderDistinct = 0;
        }
        break;
      }
    } /* end Loop over all index columns */
    if( distinctColumns ){
      testcase( isOrderDistinct==0 );
      isOrderDistinct = 1;
    }
  } /* end-if not one-row */

  /* Mark off any other ORDER BY terms that reference pLoop */
  if( isOrderDistinct ){
    orderDistinctMask |= pLoop->maskSelf;
    for(i=0; i<nOrderBy; i++){
      Expr *p;
      Bitmask mTerm;
      if( MASKBIT(i) & obSat ) continue;
      p = pOrderBy->a[i].pExpr;
      mTerm = sqlite3WhereExprUsage(&pWInfo->sMaskSet,p);
      if( mTerm==0 && !sqlite3ExprIsConstant(p) ) continue;
      if( (mTerm&~orderDistinctMask)==0 ){
        obSat |= MASKBIT(i);
      }
    }
  }
}
```

3.

(a)



SQLite 中每個 database 存在 disk file 中，因為 B-tree 對 data 的查找、刪除、添加速度快，所以這些 data 以 B-tree 的形式存在 disk 上。所有 B-tree 存在相同的 disk file 中。

B-tree 維護著多個 page 之間的關係，使其能夠快速找到有關聯的 data。一個 database 由多個 B-tree 組成的。

B-Tree Page Format:避免 overread 及讓 binary search 盡可能更快。每一個 b-tree page 都有一個 fixed-size header 及 variable-sized footer，將一些 header data 移動到 footer 的目的是為了防止從 corrupt b-tree page 中讀取 varint field 時發生 buffer overread。Pager 幫助 B-tree 管理頁面。

Page header : 1 byte - flags.

Internal nodes only: 4 bytes - Page number of rightmost child node.

Page Footer: Starting from the end of the page

Fields : 2 bytes - Number of cells on this page.

2 bytes - Total free space on page, in bytes.

2 bytes - Offset of first byte after last cell.

2 bytes for each cell - the offset to the start of the cell

```
/*
** Create a new BTree table.  Write into *piTable the page
** number for the root page of the new table.
**
** The type of type is determined by the flags parameter.  Only the
** following values of flags are currently in use.  Other values for
** flags might not work:
**
**     BTREE_INTKEY|BTREE_LEAFDATA     Used for SQL tables with rowid keys
**     BTREE_ZERODATA                  Used for SQL indices
*/
static int btreeCreateTable(Btree *p, int *piTable, int createTabFlags){
  BtShared *pBt = p->pBt;
  MemPage *pRoot;
  Pgno pgnoRoot;
  int rc;
  int ptfFlags;          /* Page-type flage for the root page of new table */

  assert( sqlite3BtreeHoldsMutex(p) );
  assert( pBt->inTransaction==TRANS_WRITE );
  assert( (pBt->btsFlags & BTS_READ_ONLY)==0 );
```

```
/* Creating a new table may probably require moving an existing database
    ** to make room for the new tables root page. In case this page turns
    ** out to be an overflow page, delete all overflow page-map caches
    ** held by open cursors.
    */
    invalidateAllOverflowCache(pBt);
```

```
/*
** Invalidate the overflow page-list cache for all cursors opened
** on the shared btree structure pBt.
*/
static void invalidateAllOverflowCache(BtShared *pBt){
  BtCursor *p;
  assert( sqlite3_mutex_held(pBt->mutex) );
  for(p=pBt->pCursor; p; p=p->pNext){
    invalidateOverflowCache(p);
```

```
  }
}
```

(b)

Open file:

```
/*
** Allocate and initialize a new Pager object and put a pointer to it
** in *ppPager. The pager should eventually be freed by passing it
** to sqlite3PagerClose().
**
** The zFilename argument is the path to the database file to open.
** If zFilename is NULL then a randomly-named temporary file is created
** and used as the file to be cached. Temporary files are be deleted
** automatically when they are closed. If zFilename is ":memory:" then
** all information is held in cache. It is never written to disk.
** This can be used to implement an in-memory database.
**
** The nExtra parameter specifies the number of bytes of space allocated
** along with each page reference. This space is available to the user
** via the sqlite3PagerGetExtra() API.  When a new page is allocated, the
** first 8 bytes of this space are zeroed but the remainder is uninitialized.
** (The extra space is used by btree as the MemPage object.)
**
** The flags argument is used to specify properties that affect the
** operation of the pager. It should be passed some bitwise combination
** of the PAGER_* flags.
**
** The vfsFlags parameter is a bitmask to pass to the flags parameter
** of the xOpen() method of the supplied VFS when opening files.
**
** If the pager object is allocated and the specified file opened
** successfully, SQLITE_OK is returned and *ppPager set to point to
** the new pager object. If an error occurs, *ppPager is set to NULL
** and error code returned. This function may return SQLITE_NOMEM
** (sqlite3Malloc() is used to allocate memory), SQLITE_CANTOPEN or
** various SQLITE_IO_XXX errors.
*/
int sqlite3PagerOpen(
```

```
  sqlite3_vfs *pVfs,         /* The virtual file system to use */
  Pager **ppPager,           /* OUT: Return the Pager structure here */
  const char *zFilename,     /* Name of the database file to open */
  int nExtra,                /* Extra bytes append to each in-memory page */
  int flags,                 /* flags controlling this file */
  int vfsFlags,              /* flags passed through to sqlite3_vfs.xOpen() */
  void (*xReinit)(DbPage*) /* Function to reinitialize pages */
){
  u8 *pPtr;
  Pager *pPager = 0;         /* Pager object to allocate and return */
  int rc = SQLITE_OK;        /* Return code */
  int tempFile = 0;          /* True for temp files (incl. in-memory files) */
  int memDb = 0;             /* True if this is an in-memory file */
```

- read

為了讓使用者能從 database 中拿到 data，SQLite 在某些時候需要從 database 中讀取 data，通常 data 是以 aligned blocks of page-size bytes 被讀取的當 data 被讀出來後，SQLite 將其 raw page 存在 page cache 中，當上層需要時就會查詢 page cache 中是否包含了當前 database connection 中所儲存的 page 的副本

```
/*
** Return a pointer to the data for the specified page.
*/
void *sqlite3PagerGetData(DbPage *pPg){
  assert( pPg->nRef>0 || pPg->pPager->memDb );
  return pPg->pData;
}

/*
```

- write

為了確保 write transaction 是獨立的，在修改 database file 的內容之前 SQLite 需要先獲得該 database file 的 exclusive lock，在 write transaction 完成之前不會 unlock 掉該鎖以確保在發生 write transaction 時不會有其他的連線在 read data。當發生 write transaction 時會修改 database page 的內容、將新的 database page 加到 database file image 中，或是在 database file 的最後面丟掉 database page，之後再將 write transaction 的結論及更動的部分永久提交到 database 中

```c
/*
** 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);
  }
}
```

- close
  在完成該 transaction 時，SQLite 只需 unlock 在 database file 中打開的 file 的鎖，之後再關掉打開的 VFS file-handle 並釋放 in-memory page cache 有關的資源

```
/*
** Shutdown the page cache.  Free all memory and close all files.
**
** If a transaction was in progress when this routine is called, that
** transaction is rolled back.  All outstanding pages are invalidated
** and their memory is freed.  Any attempt to use a page associated
** with this page cache after this function returns will likely
** result in a coredump.
**
** This function always succeeds. If a transaction is active an attempt
** is made to roll it back. If an error occurs during the rollback
** a hot journal may be left in the filesystem but no error is returned
** to the caller.
*/
int sqlite3PagerClose(Pager *pPager, sqlite3 *db){
  u8 *pTmp = (u8*)pPager->pTmpSpace;
  assert( db || pagerUseWal(pPager)==0 );
  assert( assert_pager_state(pPager) );
  disable_simulated_io_errors();
  sqlite3BeginBenignMalloc();
  pagerFreeMapHdrs(pPager);
  /* pPager->errCode = 0; */
  pPager->exclusiveMode = 0;
```

4. (a) Pager module 處理 locking 及 concurrency control，為了使 SQLite 有 Atomic、Consistent、Isolated 及 Durable.

- locking

  總共分為 5 個 state，分別為 UNLOCKED、SHARED、RESERVED、PENDING 及 EXCLUSIVE。

  - UNLOCKED 狀態時，不能讀也不能寫，為 default state;
  - SHARED 狀態時只能讀但不能寫，且可以有很多 process 同時有 SHARED lock;

```
/*
** This function is called to obtain a shared lock on the database file.
** It is illegal to call sqlite3PagerGet() until after this function
** has been successfully called. If a shared-lock is already held when
** this function is called, it is a no-op.
**
```

```
** The following operations are also performed by this function.
**
**   1) If the pager is currently in PAGER_OPEN state (no lock held
**      on the database file), then an attempt is made to obtain a
**      SHARED lock on the database file. Immediately after obtaining
**      the SHARED lock, the file-system is checked for a hot-journal,
**      which is played back if present. Following any hot-journal
**      rollback, the contents of the cache are validated by checking
**      the 'change-counter' field of the database file header and
**      discarded if they are found to be invalid.
**
**   2) If the pager is running in exclusive-mode, and there are currently
**      no outstanding references to any pages, and is in the error state,
**      then an attempt is made to clear the error state by discarding
**      the contents of the page cache and rolling back any open journal
**      file.
**
** If everything is successful, SQLITE_OK is returned. If an IO error
** occurs while locking the database, checking for a hot-journal file or
** rolling back a journal file, the IO error code is returned.
*/
int sqlite3PagerSharedLock(Pager *pPager){
```

– RESERVED 狀態時代表某個 process 正準備在某個時間點發生 write 的動作，但目前只有 read，當有 RESERVED lock 時仍可以獲得 SHARED lock;

```
 /* Race condition here:  Another process might have been holding the
    ** the RESERVED lock and have a journal open at the sqlite3OsAccess()
    ** call above, but then delete the journal and drop the lock before
    ** we get to the following sqlite3OsCheckReservedLock() call.  If that
    ** is the case, this routine might think there is a hot journal when
    ** in fact there is none.  This results in a false-positive which will
    ** be dealt with by the playback routine.  Ticket #3883.
    */
    rc = sqlite3OsCheckReservedLock(pPager->fd, &locked);
    if( rc==SQLITE_OK && !locked ){
      Pgno nPage;                 /* Number of pages in database file */

      assert( pPager->tempFile==0 );
      rc = pagerPagecount(pPager, &nPage);
```

```
int sqlite3OsCheckReservedLock(sqlite3_file *id, int *pResOut){
  DO_OS_MALLOC_TEST(id);
  return id->pMethods->xCheckReservedLock(id, pResOut);
```

– PENDING 狀態時代表該 process 即將發生 write，在 SHARED lock 被 unlock 後即可獲得 EXCLUSIVE lock，當有 PENDINFG lock 時不能獲得新的 SHARED lock，只能將目前擁有的 SHARED lock 執行完;

– EXCLUSIVE 狀態為可以 write database file，EXCLUSIVE lock 不與其他種 lock 同時存在。

```
/*
** This function may only be called while a write-transaction is active in
** rollback. If the connection is in WAL mode, this call is a no-op.
** Otherwise, if the connection does not already have an EXCLUSIVE lock on
** the database file, an attempt is made to obtain one.
**
** If the EXCLUSIVE lock is already held or the attempt to obtain it is
** successful, or the connection is in WAL mode, SQLITE_OK is returned.
** Otherwise, either SQLITE_BUSY or an SQLITE_IOERR_XXX error code is
** returned.
*/
int sqlite3PagerExclusiveLock(Pager *pPager){
  int rc = pPager->errCode;
  assert( assert_pager_state(pPager) );
  if( rc==SQLITE_OK ){
    assert( pPager->eState==PAGER_WRITER_CACHEMOD
         || pPager->eState==PAGER_WRITER_DBMOD
         || pPager->eState==PAGER_WRITER_LOCKED
    );
    assert( assert_pager_state(pPager) );
    if( 0==pagerUseWal(pPager) ){
      rc = pager_wait_on_lock(pPager, EXCLUSIVE_LOCK);
    }
  }
  return rc;
}
```

pager module 只會 track 除了 PENDING 以外的 4 種 locking state

- The Rollback Journal

  當某個 process 想要更改 database file 時，先將原本的 database content 記錄在 rollback journal 中。rollback journal 為與 database file 放在同一個

目錄底下的 disk file，且有與 disk file 相同的名稱，並帶有-journal suffix。Rollback journal 還記錄 database 的 initial size，以便 database file 增大時，可以在 rollback 時將其改為原始大小。當 SQLite 同時使用多個 database 時，每個 database 都有其自己的 rollback journal。如果需要 rollback 以恢復 database 的 integrity，則 rollback journal is said to be hot。當 process 在 database 更新過程中系統崩潰或電源故障導致更新不成功時，會 create hot journal。

A journal is hot if...

- It exists, and
- Its size is greater than 512 bytes, and
- The journal header is non-zero and well-formed, and
- Its master journal exists or the master journal name is an empty string, and
- There is no RESERVED lock on the corresponding database file.

```
/*
** Playback the journal and thus restore the database file to
** the state it was in before we started making changes.
**
** The journal file format is as follows:
**
**  (1)  8 byte prefix.  A copy of aJournalMagic[].
**  (2)  4 byte big-endian integer which is the number of valid page records
**       in the journal.  If this value is 0xffffffff, then compute the
**       number of page records from the journal size.
**  (3)  4 byte big-endian integer which is the initial value for the
**       sanity checksum.
**  (4)  4 byte integer which is the number of pages to truncate the
**       database to during a rollback.
**  (5)  4 byte big-endian integer which is the sector size.  The header
**       is this many bytes in size.
**  (6)  4 byte big-endian integer which is the page size.
**  (7)  zero padding out to the next sector size.
**  (8)  Zero or more pages instances, each as follows:
**        +  4 byte page number.
**        +  pPager->pageSize bytes of data.
**        +  4 byte checksum
**
** When we speak of the journal header, we mean the first 7 items above.
** Each entry in the journal is an instance of the 8th item.
```

```
**
** Call the value from the second bullet "nRec".  nRec is the number of
** valid page entries in the journal.  In most cases, you can compute the
** value of nRec from the size of the journal file.  But if a power
** failure occurred while the journal was being written, it could be the
** case that the size of the journal file had already been increased but
** the extra entries had not yet made it safely to disk.  In such a case,
** the value of nRec computed from the file size would be too large.  For
** that reason, we always use the nRec value in the header.
**
** If the nRec value is 0xffffffff it means that nRec should be computed
** from the file size.  This value is used when the user selects the
** no-sync option for the journal.  A power failure could lead to corruption
** in this case.  But for things like temporary table (which will be
** deleted when the power is restored) we don't care.
**
** If the file opened as the journal file is not a well-formed
** journal file then all pages up to the first corrupted page are rolled
** back (or no pages if the journal header is corrupted). The journal file
** is then deleted and SQLITE_OK returned, just as if no corruption had
** been encountered.
**
** If an I/O or malloc() error occurs, the journal-file is not deleted
** and an error code is returned.
**
** The isHot parameter indicates that we are trying to rollback a journal
** that might be a hot journal.  Or, it could be that the journal is
** preserved because of JOURNALMODE_PERSIST or JOURNALMODE_TRUNCATE.
** If the journal really is hot, reset the pager cache prior rolling
** back any content.  If the journal is merely persistent, no reset is
** needed.
*/
static int pager_playback(Pager *pPager, int isHot){
  sqlite3_vfs *pVfs = pPager->pVfs;
  i64 szJ;                 /* Size of the journal file in bytes */
  u32 nRec;                /* Number of Records in the journal */
  u32 u;                   /* Unsigned loop counter */
  Pgno mxPg = 0;           /* Size of the original file in pages */
```

```
int rc;                  /* Result code of a subroutine */
int res = 1;             /* Value returned by sqlite3OsAccess() */
char *zMaster = 0;       /* Name of master journal file if any */
int needPagerReset;      /* True to reset page prior to first page rollback */
int nPlayback = 0;       /* Total number of pages restored from journal */
u32 savedPageSize = pPager->pageSize;

/* Figure out how many records are in the journal.  Abort early if
** the journal is empty.
*/
assert( isOpen(pPager->jfd) );
rc = sqlite3OsFileSize(pPager->jfd, &szJ);
if( rc!=SQLITE_OK ){
  goto end_playback;
}

/* Read the master journal name from the journal, if it is present.
** If a master journal file name is specified, but the file is not
** present on disk, then the journal is not hot and does not need to be
** played back.
**
** TODO: Technically the following is an error because it assumes that
** buffer Pager.pTmpSpace is (mxPathname+1) bytes or larger. i.e. that
** (pPager->pageSize >= pPager->pVfs->mxPathname+1). Using os_unix.c,
** mxPathname is 512, which is the same as the minimum allowable value
** for pageSize.
*/
zMaster = pPager->pTmpSpace;
rc = readMasterJournal(pPager->jfd, zMaster, pPager->pVfs->mxPathname+1);
if( rc==SQLITE_OK && zMaster[0] ){
  rc = sqlite3OsAccess(pVfs, zMaster, SQLITE_ACCESS_EXISTS, &res);
}
zMaster = 0;
if( rc!=SQLITE_OK || !res ){
  goto end_playback;
}
pPager->journalOff = 0;
needPagerReset = isHot;
```

```
/* This loop terminates either when a readJournalHdr() or
** pager_playback_one_page() call returns SQLITE_DONE or an IO error
** occurs.
*/
while( 1 ){
  /* Read the next journal header from the journal file.  If there are
  ** not enough bytes left in the journal file for a complete header, or
  ** it is corrupted, then a process must have failed while writing it.
  ** This indicates nothing more needs to be rolled back.
  */
  rc = readJournalHdr(pPager, isHot, szJ, &nRec, &mxPg);
  if( rc!=SQLITE_OK ){
    if( rc==SQLITE_DONE ){
      rc = SQLITE_OK;
    }
    goto end_playback;
  }

  /* If nRec is 0xffffffff, then this journal was created by a process
  ** working in no-sync mode. This means that the rest of the journal
  ** file consists of pages, there are no more journal headers. Compute
  ** the value of nRec based on this assumption.
  */
  if( nRec==0xffffffff ){
    assert( pPager->journalOff==JOURNAL_HDR_SZ(pPager) );
    nRec = (int)((szJ - JOURNAL_HDR_SZ(pPager))/JOURNAL_PG_SZ(pPager));
  }

  /* If nRec is 0 and this rollback is of a transaction created by this
  ** process and if this is the final header in the journal, then it means
  ** that this part of the journal was being filled but has not yet been
  ** synced to disk.  Compute the number of pages based on the remaining
  ** size of the file.
  **
  ** The third term of the test was added to fix ticket #2565.
  ** When rolling back a hot journal, nRec==0 always means that the next
  ** chunk of the journal contains zero pages to be rolled back.  But
```

```
** when doing a ROLLBACK and the nRec==0 chunk is the last chunk in
** the journal, it means that the journal might contain additional
** pages that need to be rolled back and that the number of pages
** should be computed based on the journal file size.
*/
if( nRec==0 && !isHot &&
    pPager->journalHdr+JOURNAL_HDR_SZ(pPager)==pPager->journalOff ){
  nRec = (int)((szJ - pPager->journalOff) / JOURNAL_PG_SZ(pPager));
}

/* If this is the first header read from the journal, truncate the
** database file back to its original size.
*/
if( pPager->journalOff==JOURNAL_HDR_SZ(pPager) ){
  rc = pager_truncate(pPager, mxPg);
  if( rc!=SQLITE_OK ){
    goto end_playback;
  }
  pPager->dbSize = mxPg;
}

/* Copy original pages out of the journal and back into the
** database file and/or page cache.
*/
for(u=0; u<nRec; u++){
  if( needPagerReset ){
    pager_reset(pPager);
    needPagerReset = 0;
  }
  rc = pager_playback_one_page(pPager,&pPager->journalOff,0,1,0);
  if( rc==SQLITE_OK ){
    nPlayback++;
  }else{
    if( rc==SQLITE_DONE ){
      pPager->journalOff = szJ;
      break;
    }else if( rc==SQLITE_IOERR_SHORT_READ ){
      /* If the journal has been truncated, simply stop reading and
```

```
          ** processing the journal. This might happen if the journal was
          ** not completely written and synced prior to a crash.  In that
          ** case, the database should have never been written in the
          ** first place so it is OK to simply abandon the rollback. */
          rc = SQLITE_OK;
          goto end_playback;
        }else{
          /* If we are unable to rollback, quit and return the error
          ** code.  This will cause the pager to enter the error state
          ** so that no further harm will be done.  Perhaps the next
          ** process to come along will be able to rollback the database.
          */
          goto end_playback;
        }
      }
    }
  }
  /*NOTREACHED*/
  assert( 0 );

end_playback:
  if( rc==SQLITE_OK ){
    rc = sqlite3PagerSetPagesize(pPager, &savedPageSize, -1);
  }
```

(b)(i)

Example :假設有兩個 database connection X 和 Y。X 使用 BEGIN 和 SELECT statement 啟動 read transaction。然後 Y 出現並 run UPDATE statement 以修改 database。X 隨後可以對 Y 修改的記錄執行 SELECT，但 X 將看到較舊的資料，因為當 X 持有該 read transaction 時看不到 Y 的更動。如果 X 要查看 Y 所做的更改，則 X 必須結束該 read transaction 並開始一個新的 transaction

因此 the change done by a commit is visible to other operations

```
/*
** Generate VDBE code for a COMMIT or ROLLBACK statement.
** Code for ROLLBACK is generated if eType==TK_ROLLBACK.  Otherwise
** code is generated for a COMMIT.
*/
void sqlite3EndTransaction(Parse *pParse, int eType){
  Vdbe *v;
```

```
  int isRollback;

  assert( pParse!=0 );
  assert( pParse->db!=0 );
  assert( eType==TK_COMMIT || eType==TK_END || eType==TK_ROLLBACK );
  isRollback = eType==TK_ROLLBACK;
  if( sqlite3AuthCheck(pParse, SQLITE_TRANSACTION,
       isRollback ? "ROLLBACK" : "COMMIT", 0, 0) ){
    return;
  }
  v = sqlite3GetVdbe(pParse);
  if( v ){
    sqlite3VdbeAddOp2(v, OP_AutoCommit, 1, isRollback);
  }
}
```

(ii) Example :X 使用 BEGIN 和 SELECT 啟動 read transaction，然後 Y 使用 UPDATE 對 database 進行更改。然後 X 嘗試使用 UPDATE 對 database 進行更改。X 嘗試將 read transaction 升級成 write transaction 但失敗，並出現 SQLITE_BUSY_SNAPSHOT error，因為 X 看到的 database snapshot 不再是最新版本的 database。如果允許 X 寫入，它將 fork database file 的歷史記錄，而 SQLite 不支援。為了讓 X 寫入 database，它必須首先 release snapshot，然後藉由 BEGIN 開始新的 transaction。
因此當一開始有一個 process 對 database 做更改時，在期結束前若有其他 process 也對該 database 做更改就會發生 nondeterministic 的情況。

```
/*
** All of the code in this file may be omitted by defining a single
** macro.
*/
#ifndef SQLITE_OMIT_AUTHORIZATION

/*
** Set or clear the access authorization function.
**
** The access authorization function is be called during the compilation
** phase to verify that the user has read and/or write access permission on
** various fields of the database.  The first argument to the auth function
** is a copy of the 3rd argument to this routine.  The second argument
```

```
** to the auth function is one of these constants:
**
**       SQLITE_CREATE_INDEX
**       SQLITE_CREATE_TABLE
**       SQLITE_CREATE_TEMP_INDEX
**       SQLITE_CREATE_TEMP_TABLE
**       SQLITE_CREATE_TEMP_TRIGGER
**       SQLITE_CREATE_TEMP_VIEW
**       SQLITE_CREATE_TRIGGER
**       SQLITE_CREATE_VIEW
**       SQLITE_DELETE
**       SQLITE_DROP_INDEX
**       SQLITE_DROP_TABLE
**       SQLITE_DROP_TEMP_INDEX
**       SQLITE_DROP_TEMP_TABLE
**       SQLITE_DROP_TEMP_TRIGGER
**       SQLITE_DROP_TEMP_VIEW
**       SQLITE_DROP_TRIGGER
**       SQLITE_DROP_VIEW
**       SQLITE_INSERT
**       SQLITE_PRAGMA
**       SQLITE_READ
**       SQLITE_SELECT
**       SQLITE_TRANSACTION
**       SQLITE_UPDATE
**
** The third and fourth arguments to the auth function are the name of
** the table and the column that are being accessed.  The auth function
** should return either SQLITE_OK, SQLITE_DENY, or SQLITE_IGNORE.  If
** SQLITE_OK is returned, it means that access is allowed.  SQLITE_DENY
** means that the SQL statement will never-run - the sqlite3_exec() call
** will return with an error.  SQLITE_IGNORE means that the SQL statement
** should run but attempts to read the specified column will return NULL
** and attempts to write the column will be ignored.
**
** Setting the auth function to NULL disables this hook.  The default
** setting of the auth function is NULL.
*/
```

```c
int sqlite3_set_authorizer(
  sqlite3 *db,
  int (*xAuth)(void*,int,const char*,const char*,const char*,const char*),
  void *pArg
){
#ifdef SQLITE_ENABLE_API_ARMOR
  if( !sqlite3SafetyCheckOk(db) ) return SQLITE_MISUSE_BKPT;
#endif
  sqlite3_mutex_enter(db->mutex);
  db->xAuth = (sqlite3_xauth)xAuth;
  db->pAuthArg = pArg;
  if( db->xAuth ) sqlite3ExpirePreparedStatements(db, 1);
  sqlite3_mutex_leave(db->mutex);
  return SQLITE_OK;
}

/*
** Write an error message into pParse->zErrMsg that explains that the
** user-supplied authorization function returned an illegal value.
*/
static void sqliteAuthBadReturnCode(Parse *pParse){
  sqlite3ErrorMsg(pParse, "authorizer malfunction");
  pParse->rc = SQLITE_ERROR;
}
```

2-B

• 我選擇 Utility 中的 printf.c，因為通常在寫 SQL 的時候都只有寫 select，但當 query 執行完後又會 print 東西出來，因此我想知道為何 select 後不需寫 print。Printf.c 在 SQLite 的角色為類似 c 語言中的 printf

• Related source code files : “sqlite/src/printf.c”

• sqlite3_mprintf（）和 sqlite3_vmprintf（）routines 將其結果寫入從 sqlite3_malloc64（）獲得的 memory 中。這兩個 routines 返回的 string 應由 sqlite3_free（）釋放。如果 sqlite3_malloc64（）無法分配足夠的內存來容納結果字符串，則這兩個 routines 都將 return NULL pointer。

sqlite3_snprintf（）routine 類似 C language 中的“ snprintf（）”。結果被寫入作為第二個參數提供的 buffer，第二個參數的大小由第一個參數給定。請注意，前兩個參數的順序與 snprintf（）相反，必須在不破壞向後兼容性的情況下才能

解決。sqlite3_snprintf（）返回指向其 buffer 的 pointer，而不是實際寫入 buffer 的 character 數量。
只要 buffer 大小大於零，sqlite3_snprintf（）就會保證 buffer 始終以零結尾。第一個參數“ n”是 buffer 的總大小，包括 zero terminator 的空間。因此，可以完全寫入的最長字符串為 n-1 個字符。
sqlite3_vsnprintf（）例程是 sqlite3_snprintf（）的 varargs 版本。

```
/*
** "*val" is a double such that 0.1 <= *val < 10.0
** Return the ascii code for the leading digit of *val, then
** multiply "*val" by 10.0 to renormalize.
**
** Example:
**     input:     *val = 3.14159
**     output:    *val = 1.4159    function return = '3'
**
** The counter *cnt is incremented each time.  After counter exceeds
** 16 (the number of significant digits in a 64-bit float) '0' is
** always returned.
*/
static char et_getdigit(LONGDOUBLE_TYPE *val, int *cnt){
  int digit;
  LONGDOUBLE_TYPE d;
  if( (*cnt)<=0 ) return '0';
  (*cnt)--;
  digit = (int)*val;
  d = digit;
  digit += '0';
  *val = (*val - d)*10.0;
  return (char)digit;
}
```

```
/*
** Allocate memory for a temporary buffer needed for printf rendering.
**
** If the requested size of the temp buffer is larger than the size
** of the output buffer in pAccum, then cause an SQLITE_TOOBIG error.
** Do the size check before the memory allocation to prevent rogue
** SQL from requesting large allocations using the precision or width
```

```
** field of the printf() function.
*/
static char *printfTempBuf(sqlite3_str *pAccum, sqlite3_int64 n){
  char *z;
  if( pAccum->accError ) return 0;
  if( n>pAccum->nAlloc && n>pAccum->mxAlloc ){
    setStrAccumError(pAccum, SQLITE_TOOBIG);
    return 0;
  }
  z = sqlite3DbMallocRaw(pAccum->db, n);
  if( z==0 ){
    setStrAccumError(pAccum, SQLITE_NOMEM);
  }
  return z;
}
```

```
/*
** Print into memory obtained from sqliteMalloc().  Use the internal
** %-conversion extensions.
*/
char *sqlite3VMPrintf(sqlite3 *db, const char *zFormat, va_list ap){
  char *z;
  char zBase[SQLITE_PRINT_BUF_SIZE];
  StrAccum acc;
  assert( db!=0 );
  sqlite3StrAccumInit(&acc, db, zBase, sizeof(zBase),
                      db->aLimit[SQLITE_LIMIT_LENGTH]);
  acc.printfFlags = SQLITE_PRINTF_INTERNAL;
  sqlite3_str_vappendf(&acc, zFormat, ap);
  z = sqlite3StrAccumFinish(&acc);
  if( acc.accError==SQLITE_NOMEM ){
    sqlite3OomFault(db);
  }
  return z;
}
```

```
/*
** Print into memory obtained from sqliteMalloc().  Use the internal
```

```
** %-conversion extensions.
*/
char *sqlite3MPrintf(sqlite3 *db, const char *zFormat, ...){
  va_list ap;
  char *z;
  va_start(ap, zFormat);
  z = sqlite3VMPrintf(db, zFormat, ap);
  va_end(ap);
  return z;
}
```

```
/*
** Print into memory obtained from sqlite3_malloc().  Omit the internal
** %-conversion extensions.
*/
char *sqlite3_vmprintf(const char *zFormat, va_list ap){
  char *z;
  char zBase[SQLITE_PRINT_BUF_SIZE];
  StrAccum acc;

#ifdef SQLITE_ENABLE_API_ARMOR
  if( zFormat==0 ){
    (void)SQLITE_MISUSE_BKPT;
    return 0;
  }
#endif
#ifndef SQLITE_OMIT_AUTOINIT
  if( sqlite3_initialize() ) return 0;
#endif
  sqlite3StrAccumInit(&acc, 0, zBase, sizeof(zBase), SQLITE_MAX_LENGTH);
  sqlite3_str_vappendf(&acc, zFormat, ap);
  z = sqlite3StrAccumFinish(&acc);
  return z;
}
```

```
/*
** Print into memory obtained from sqlite3_malloc()().  Omit the internal
** %-conversion extensions.
```

```
*/
char *sqlite3_mprintf(const char *zFormat, ...){
  va_list ap;
  char *z;
#ifndef SQLITE_OMIT_AUTOINIT
  if( sqlite3_initialize() ) return 0;
#endif
  va_start(ap, zFormat);
  z = sqlite3_vmprintf(zFormat, ap);
  va_end(ap);
  return z;
}
```

```
/*
** sqlite3_snprintf() works like snprintf() except that it ignores the
** current locale settings.  This is important for SQLite because we
** are not able to use a "," as the decimal point in place of "." as
** specified by some locales.
**
** Oops:  The first two arguments of sqlite3_snprintf() are backwards
** from the snprintf() standard.  Unfortunately, it is too late to change
** this without breaking compatibility, so we just have to live with the
** mistake.
**
** sqlite3_vsnprintf() is the varargs version.
*/
char *sqlite3_vsnprintf(int n, char *zBuf, const char *zFormat, va_list ap){
  StrAccum acc;
  if( n<=0 ) return zBuf;
#ifdef SQLITE_ENABLE_API_ARMOR
  if( zBuf==0 || zFormat==0 ) {
    (void)SQLITE_MISUSE_BKPT;
    if( zBuf ) zBuf[0] = 0;
    return zBuf;
  }
#endif
  sqlite3StrAccumInit(&acc, 0, zBuf, n, 0);
  sqlite3_str_vappendf(&acc, zFormat, ap);
```

```
  zBuf[acc.nChar] = 0;
  return zBuf;
}
char *sqlite3_snprintf(int n, char *zBuf, const char *zFormat, ...){
  char *z;
  va_list ap;
  va_start(ap,zFormat);
  z = sqlite3_vsnprintf(n, zBuf, zFormat, ap);
  va_end(ap);
  return z;
}
```

```
/*
** This is the routine that actually formats the sqlite3_log() message.
** We house it in a separate routine from sqlite3_log() to avoid using
** stack space on small-stack systems when logging is disabled.
**
** sqlite3_log() must render into a static buffer.  It cannot dynamically
** allocate memory because it might be called while the memory allocator
** mutex is held.
**
** sqlite3_str_vappendf() might ask for *temporary* memory allocations for
** certain format characters (%q) or for very large precisions or widths.
** Care must be taken that any sqlite3_log() calls that occur while the
** memory mutex is held do not use these mechanisms.
*/
static void renderLogMsg(int iErrCode, const char *zFormat, va_list ap){
  StrAccum acc;                          /* String accumulator */
  char zMsg[SQLITE_PRINT_BUF_SIZE*3];    /* Complete log message */

  sqlite3StrAccumInit(&acc, 0, zMsg, sizeof(zMsg), 0);
  sqlite3_str_vappendf(&acc, zFormat, ap);
  sqlite3GlobalConfig.xLog(sqlite3GlobalConfig.pLogArg, iErrCode,
                           sqlite3StrAccumFinish(&acc));
}
```