SQLite3源程序分析之查询处理及优化-CSDN博客

blog.csdn.net/weixin_30608503/article/details/98840621

前言

查询处理及优化是关系数据库得以流行的根本原因，也是关系数据库系统最核心的技术之一。SQLite的查询处理模块很精致，而且很容易移植到不支持SQL的存储引擎(Berkeley DB最新的版本已经将其完整的移植过来)。

查询处理一般来说，包括词法分析、语法分析、语义分析、生成执行计划以及执行计划几个部分。SQLite的词法分析器是手工写的(比较简单)，语法分析器由Lemon生成，语义分析主要是进行语义方面的一些检查，比如table是否存在等。而执行计划的生成及执行是最核心的两部分，也是相对比较复杂、有点技术含量的部分。SQLite的执行计划采用了虚拟机的思想，实际上，这种基于虚拟机的思想并非SQLite所独有，但是，SQLite将其发挥到了极致，它生成的执行计划非常详细，而且易读(不得不佩服D. Richard Hipp在编译理论方面的功底)。

1、语法分析——语法树

语法分析的主要任务是对用户输入的SQL语句进行语法检查，然后生成一个包含所有信息的语法树。对于SELECT语句，这个语法树最终由结构体Select表示：

```
struct Select {
  ExprList *pEList;      /* The fields of the result 结果的字段 */
  u8 op;                 /* One of: TK_UNION TK_ALL TK_INTERSECT TK_EXCEPT */
  char affinity;         /* MakeRecord with this affinity for SRT_Set */
  u16 selFlags;          /* Various SF_* values */
  SrcList *pSrc;         /* The FROM clause */
  Expr *pWhere;          /* The WHERE clause */
  ExprList *pGroupBy;    /* The GROUP BY clause */
  Expr *pHaving;         /* The HAVING clause */
  ExprList *pOrderBy;    /* The ORDER BY clause */
  Select *pPrior;        /* Prior select in a compound select statement 在一个复合选择语
句中的优先选择 */
  Select *pNext;         /* Next select to the left in a compound */
  Select *pRightmost;    /* Right-most select in a compound select statement 在一个复合
选择语句中的最右边的选择*/
  Expr *pLimit;          /* LIMIT expression. NULL means not used. */
  Expr *pOffset;         /* OFFSET expression. NULL means not used. */
  int iLimit, iOffset;   /* Memory registers holding LIMIT & OFFSET counters */
  int addrOpenEphm[3];   /* OP_OpenEphem opcodes related to this select */
};
```

该结构体中，pEList是结果列的语法树；pSrc为FROM子句的语法树；pWhere为WHERE部分的语法树。

select语法分析最终在sqlite3SelectNew中完成：

```
Select *sqlite3SelectNew(
  Parse *pParse,        /* Parsing context 解析上下文 */
  ExprList *pEList,     /* which columns to include in the result 在结果中包含哪些列 */
  SrcList *pSrc,        /* the FROM clause -- which tables to scan */
  Expr *pWhere,         /* the WHERE clause */
  ExprList *pGroupBy,   /* the GROUP BY clause */
  Expr *pHaving,        /* the HAVING clause */
  ExprList *pOrderBy,   /* the ORDER BY clause */
  int isDistinct,       /* true if the DISTINCT keyword is present */
  Expr *pLimit,         /* LIMIT value.  NULL means not used */
  Expr *pOffset         /* OFFSET value.  NULL means no offset */
){
  Select *pNew;
  Select standin;
  sqlite3 *db = pParse->db;
  pNew = sqlite3DbMallocZero(db, sizeof(*pNew) );
  assert( db->mallocFailed || !pOffset || pLimit ); /* OFFSET implies LIMIT */
  if( pNew==0 ){
    pNew = &standin;
    memset(pNew, 0, sizeof(*pNew));
  }
  if( pEList==0 ){
    pEList = sqlite3ExprListAppend(pParse, 0, sqlite3Expr(db,TK_ALL,0));
  }
  pNew->pEList = pEList;
  pNew->pSrc = pSrc;
  pNew->pWhere = pWhere;
  pNew->pGroupBy = pGroupBy;
  pNew->pHaving = pHaving;
  pNew->pOrderBy = pOrderBy;
  pNew->selFlags = isDistinct ? SF_Distinct : 0;
  pNew->op = TK_SELECT;
  pNew->pLimit = pLimit;
  pNew->pOffset = pOffset;
  assert( pOffset==0 || pLimit!=0 );
  pNew->addrOpenEphm[0] = -1;
  pNew->addrOpenEphm[1] = -1;
  pNew->addrOpenEphm[2] = -1;
  if( db->mallocFailed ) {
    clearSelect(db, pNew);
    if( pNew!=&standin ) sqlite3DbFree(db, pNew);
    pNew = 0;
  }
  return pNew;
}
```

以上函数主要是将之前得到的各个子语法树汇总到Select结构体，并根据该结构，进行语义分析及执行计划的生成等工作。

示例(贯穿全文)**：**

```
explain select s.sname,c.cname,sc.grade from students s join sc join course c on
s.sid=sc.sid and sc.cid = c.cid;
0|Trace|0|0|0||00|
1|Goto|0|35|0||00|
//(1)////
2|OpenRead|0|3|0|2|00|students    #打开students表
3|OpenRead|1|7|0|3|00|sc          #打开sc表
4|OpenRead|3|8|0|keyinfo(2,BINARY,BINARY)|00|sqlite_autoindex_sc_1 #sc的索引
5|OpenRead|2|5|0|2|00|course      #打开course表
6|OpenRead|4|6|0|keyinfo(1,BINARY)|00|sqlite_autoindex_course_1    #course的索引
//(2)//
7|Rewind|0|29|0||00|               #将游标p0定位到students表的第一条记录
8|Column|0|0|1||00|students.sid    #取出第0列,写到r1
9|IsNull|1|28|0||00|
10|Affinity|1|1|0|d|00|
11|SeekGe|3|28|1|1|00|             #将游标p3定位到sc索引>=r1的记录处
12|IdxGE|3|28|1|1|01|
13|IdxRowid|3|2|0||00|
14|Seek|1|2|0||00|
15|Column|3|1|3||00|sc.cid         #读取sc.cid到r3
16|IsNull|3|27|0||00|
17|Affinity|3|1|0|d|00|
18|SeekGe|4|27|3|1|00|             #将游标p4定位到course索引>=r3的记录处
19|IdxGE|4|27|3|1|01|
20|IdxRowid|4|4|0||00|
21|Seek|2|4|0||00|
///(3)//
22|Column|0|1|5||00|students.sname #从游标p0取出第1列 (sname)
23|Column|2|1|6||00|course.cname   #从游标p2取出第1列 (cname)
24|Column|1|2|7||00|sc.grade       #从游标p1取出第2列 (grade)
25|ResultRow|5|3|0||00|
///(4)///
26|Next|4|19|0||00|
27|Next|3|12|0||00|
28|Next|0|8|0||01|
29|Close|0|0|0||00|
30|Close|1|0|0||00|
31|Close|3|0|0||00|
32|Close|2|0|0||00|
33|Close|4|0|0||00|
//(5)//
34|Halt|0|0|0||00|
35|Transaction|0|0|0||00|
36|VerifyCookie|0|7|0||00|
37|TableLock|0|3|0|students|00|
38|TableLock|0|7|0|sc|00|
39|TableLock|0|5|0|course|00|
40|Goto|0|2|0||00|
```

该SQL语句生成的语法树如下：

FROM部分：

第一个表项：

表名zName =”stduents”，zAlias=”s”，
jointype = 0

第二个表项：

jointype = 1(JT_INNER)

第三个表项：

jointype = 1(JT_INNER)

WHERE部分(结点类型为Expr的一棵二叉树)**：**

2、生成执行计划(语法树到OPCODE)

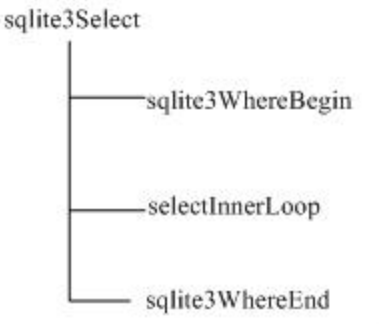
Select的执行计划在sqlite3Select中完成：

```
int sqlite3Select(
  Parse *pParse,         /* The parser context */
  Select *p,             /* SELECT语法树 */
  SelectDest *pDest      /* 如何处理结果集 */
)
```

该函数先对SQL语句进行语义分析，再进行优化，最后生成执行计划。

对于上面的SQL语句，生成的执行计划(虚拟机opcode)大致分成5部分，前4部分都在sqlite3Select()中生成，它主要调用了以下几个函数：

其中(1)、(2)在sqlite3WhereBegin()中生成，(2)即所谓的查询优化处理；(3)在 selectInnerLoop中生成；(4)在sqlite3WhereEnd中生成；(5)在sqlite3FinishCoding中完成。



1）sqlite3WhereBegin

该函数是查询处理最为核心的函数，它主要完成where部分的优化及相关opcode的生成。

```
WhereInfo *sqlite3WhereBegin(
  Parse *pParse,        /* The parser context */
  SrcList *pTabList,    /* A list of all tables to be scanned 要扫描的所有表的列表*/
  Expr *pWhere,         /* The WHERE clause */
  ExprList **ppOrderBy, /* An ORDER BY clause, or NULL */
  u16 wctrlFlags        /* One of the WHERE_* flags defined in sqliteInt.h */
)
```

pTabList是由分析器对FROM部分生成的语法树，它包含FROM语句中的表的信息；pWhere是WHERE部分的语法树，它包含WHERE中所有表达式的信息；ppOrderBy对应ORDER BY子句。

SQLite的查询优化简单而精致，在sqlite3WhereBegin函数中，即可完成所有的优化处理。查询优化的基本理念就是**嵌套循环**(nested loop)，SELECT语句的FROM子句的每个表对应一层循环(INSERT和UPDATE语句对应只有一个表)。例如：

```
SELECT * FROM t1, t2, t3 WHERE ...;
```

进行如下操作：

```
foreach row1 in t1 do       \    Code generated
  foreach row2 in t2 do      |-- by sqlite3WhereBegin()
    foreach row3 in t3 do   /
      ...
    end                     \    Code generated
  end                        |-- by sqlite3WhereEnd()
end                         /
```

而对于每一层的优化，基本的理念就是分析WHERE子句中是否有表达式能够使用该层循环的表的索引。

SQLite有三种基本的扫描策略：

① 全表扫描，这种情况通常出现在没有WHERE子句时；

② 基于索引扫描，这种情况通常出现在表有索引，而且WHERE中的表达式又能够使用该索引的情况；

③ 基本rowid的扫描，这种情况通常出现在WHERE表达式中含有rowid的条件。(该情况实际上也是对表进行扫描，SQLite以rowid为聚簇索引)

第一种情况比较简单，第三种情况与第二种情况没有本质的差别。下面就第二种情况进行详细讨论。

以下为**sqlite3WhereBegin**的关键**代码：**

```
/*分析where子句的所有表达式
**如果表达式的形式为X <op> Y,则增加一个Y <op> X形式的虚Term,并在后面进行单独分析
*/
exprAnalyzeAll(pTabList, pWC);
WHERETRACE(("*** Optimizer Start ***\n"));
//优化开始
for(i=iFrom=0, pLevel=pWInfo->a; i<nTabList; i++, pLevel++){
    WhereCost bestPlan;         /* Most efficient plan seen so far 迄今为止最有效的计划
*/
    Index *pIdx;                /* Index for FROM table at pTabItem */
    int j;                      /* For looping over FROM tables 从表循环*/
    int bestJ = -1;             /* The value of j */
    Bitmask m;                  /* Bitmask value for j or bestJ */
    int isOptimal;              /* Iterator for optimal/non-optimal search 优化/非优化
搜索的迭代器 */

    memset(&bestPlan, 0, sizeof(bestPlan));
    bestPlan.rCost = SQLITE_BIG_DBL;

    /*进行两次扫描:*/
   //如果第一次扫描没有找到优化的扫描策略,此时,isOptimal==0,bestJ==-1,则进行第二次扫描
    for(isOptimal=1; isOptimal>=0 && bestJ<0; isOptimal--){
      //第一次扫描的mask==0,表示所有表都已经准备好
      Bitmask mask = (isOptimal ? 0 : notReady);
      assert( (nTabList-iFrom)>1 || isOptimal );

      for(j=iFrom, pTabItem=&pTabList->a[j]; j<nTabList; j++, pTabItem++){
        int doNotReorder;    /* True if this table should not be reordered 如果该表不应
该被重新排序为True */
        WhereCost sCost;     /* Cost information from best[Virtual]Index() */
        ExprList *pOrderBy;  /* ORDER BY clause for index to optimize */

        //对于左连接和交叉连接,不能改变嵌套的顺序
        doNotReorder =  (pTabItem->jointype & (JT_LEFT|JT_CROSS))!=0;

        if( j!=iFrom && doNotReorder ) //如果j==iFrom,仍要进行优化处理(此时,是第一次处理
iFrom项)
            break;
        m = getMask(pMaskSet, pTabItem->iCursor);
        if( (m & notReady)==0 ){//如果该pTabItem已经进行处理,则不需要再处理
          if( j==iFrom )
              iFrom++;
          continue;
        }
        pOrderBy = ((i==0 && ppOrderBy )?*ppOrderBy:0);

        {
          //对一个表(pTabItem),找到它的可用于本次查询的最好的索引,sCost返回对应的代价
          bestBtreeIndex(pParse, pWC, pTabItem, mask, pOrderBy, &sCost);
        }
          if( (sCost.used&notReady)==0
         && (j==iFrom || sCost.rCost<bestPlan.rCost)
```

```
        ){
          bestPlan = sCost;
          bestJ = j; //如果bestJ>=0,表示找到了优化的扫描策略
        }
        if( doNotReorder ) break;
      }//end for
    }//end for
    WHERETRACE(("*** Optimizer selects table %d for loop %d\n", bestJ,
           pLevel-pWInfo->a));

    if( (bestPlan.plan.wsFlags & WHERE_ORDERBY)!=0 ){//不需要进行排序操作
      *ppOrderBy = 0;
    }
   //设置该层选用的查询策略
   andFlags &= bestPlan.plan.wsFlags;
   pLevel->plan = bestPlan.plan;

   //如果可以使用索引,则设置索引对应的游标的下标
   if( bestPlan.plan.wsFlags & WHERE_INDEXED ){
      pLevel->iIdxCur = pParse->nTab++;
   }else{
      pLevel->iIdxCur = -1;
   }
   notReady &= ~getMask(pMaskSet, pTabList->a[bestJ].iCursor);
   //该层对应的FROM的表项,即该层循环是对哪个表进行的操作
   pLevel->iFrom = (u8)bestJ;

}
//优化结束
WHERETRACE(("*** Optimizer Finished ***\n"));
```

优化部分的代码的基本算法如下：

```
foreach  level  in all_levels
bestPlan.rCost = SQLITE_BIG_DBL
foreach table in tables that not handled
{
    //计算where中表达式能使用其索引的策略及代价rCost
    If(sCost.rCost < bestPlan.rCost)
      bestPlan = sCost
}
level.plan = bestPlan
```

该算法本质上是一个贪婪算法(greedy algorithm)。其中，bestBtreeIndex(pParse, pWC, pTabItem, mask, pOrderBy, &sCost)是pParse对应的表针对where子句的表达式分析查询策略的核心函数。

对于之前的示例，经过以上优化处理后，得到的查询策略分3层循环，最外层是students表，全表扫描；中间层是sc表，利用索引sqlite_autoindex_sc_1，即sc的key对应的索引；内层是course表，利用索引sqlite_autoindex_course_1。

之后，开始生成(1)、(2)两部分opcode。

其中**(1)的opcode由以下代码生成：**

```
//生成打开表的指令
if( (pLevel->plan.wsFlags & WHERE_IDX_ONLY)==0
    && (wctrlFlags & WHERE_OMIT_OPEN)==0 ){
    //pTabItem->iCursor为表对应的游标下标
    int op = pWInfo->okOnePass ? OP_OpenWrite : OP_OpenRead;
    sqlite3OpenTable(pParse, pTabItem->iCursor, iDb, pTab, op);
}

//生成打开索引的指令
if( (pLevel->plan.wsFlags & WHERE_INDEXED)!=0 ){
    Index *pIx = pLevel->plan.u.pIdx;
    KeyInfo *pKey = sqlite3IndexKeyinfo(pParse, pIx);
    int iIdxCur = pLevel->iIdxCur; //索引对应的游标下标
    sqlite3VdbeAddOp4(v, OP_OpenRead, iIdxCur, pIx->tnum, iDb,(char*)pKey,
P4_KEYINFO_HANDOFF);
    VdbeComment((v, "%s", pIx->zName));
}
```

而**(2)的opcode由以下代码生成：**

```
  notReady = ~(Bitmask)0;
  for(i=0; i<nTabList; i++){
    //核心代码,从最外层向最内层,为每一层循环生成opcode
    notReady = codeOneLoopStart(pWInfo, i, wctrlFlags, notReady);
    pWInfo->iContinue = pWInfo->a[i].addrCont;
  }
```

其中codeOneLoopStart(pWInfo, i, wctrlFlags, notReady)函数，根据优化分析得到的结果生成每层循环的opcode：

```
static Bitmask codeOneLoopStart(
  WhereInfo *pWInfo,   /* Complete information about the WHERE clause */
  int iLevel,          /* Which level of pWInfo->a[] should be coded */
  u16 wctrlFlags,      /* One of the WHERE_* flags defined in sqliteInt.h */
  Bitmask notReady     /* Which tables are currently available */
)
```

codeOneLoopStart针对5种不同的查询策略，**生成**各自不同的**opcode:**

```
if( pLevel->plan.wsFlags & WHERE_ROWID_EQ ){    //rowid的等值查询
...
}else if( pLevel->plan.wsFlags & WHERE_ROWID_RANGE ){    //rowid的范围查询
...
}else if( pLevel->plan.wsFlags & (WHERE_COLUMN_RANGE|WHERE_COLUMN_EQ) ){    //使用索引
的等值/范围查询
...
}if( pLevel->plan.wsFlags & WHERE_MULTI_OR ){    //or
...
}else{    //全表扫描
...
}
```

其中，**全表扫描**如下：

```
static const u8 aStep[] = { OP_Next, OP_Prev };
static const u8 aStart[] = { OP_Rewind, OP_Last };
pLevel->op = aStep[bRev];
pLevel->p1 = iCur;
pLevel->p2 = 1 + sqlite3VdbeAddOp2(v, aStart[bRev], iCur, addrBrk); //生成
OP_Rewind/OP_Last指令
pLevel->p5 = SQLITE_STMTSTATUS_FULLSCAN_STEP;
```

示例中最外层循环students是全表扫描，生成指令7。

其中，**利用索引的等值/范围查询：**

对于示例：中间循环sc表，用到索引，指令8~14是对应的opcode。
内层循环course表，也用到索引，指令15~21是对应的opcode。

在通用数据库中，连接操作会生成所谓的结果集(用临时表存储)。而SQLite不会生成中间结果集，例如示例中，会分别对students、sc和course表各分配一个游标，每次调用接口sqlite3_step时，游标根据where条件分别定位到各自的记录，然后取出查询输出列的数据，放到用于存放结果的寄存器中(如示例(3)中的opcode)。所以在SQLite中，必须不断调用sqlite3_step才能读取所有记录。

2）selectInnerLoop

该函数主要生成输出结果列的opcode，即示例(3)中的opcode。

3）sqlite3WhereEnd

该函数主要完成嵌套循环的收尾工作的opcode的生成，为每层循环生成OP_Next/OP_Prev，以及关闭表和索引游标的OP_Close。

3、SQLite的代价模型

再看bestBtreeIndex函数，其完成查询代价的计算以及查询策略的确定。

SQLite采用基于代价的优化。根据处理查询时CPU和磁盘I/O的代价，主要考虑以下一些因素：

A、查询读取的记录数；

B、结果是否排序(这可能会导致使用临时表)；

C、是否需要访问索引和原表。

```
static void bestBtreeIndex(
  Parse *pParse,              /* The parsing context */
  WhereClause *pWC,           /* The WHERE clause */
  struct SrcList_item *pSrc,  /* The FROM clause term to search */
  Bitmask notReady,           /* Mask of cursors that are not available */
  ExprList *pOrderBy,         /* The ORDER BY clause */
  WhereCost *pCost            /* Lowest cost query plan */
)
```

该函数的主要工作就是输出pCost，它包含查询策略信息及相应的代价。

其**核心算法**如下**：**

```
//遍历其所有索引,找到一个代价最小的索引
for(; pProbe; pIdx=pProbe=pProbe->pNext){
    const unsigned int * const aiRowEst = pProbe->aiRowEst;
    double cost;                /* Cost of using pProbe */
    double nRow;                /* Estimated number of rows in result set */
    int rev;                    /* True to scan in reverse order */
    int wsFlags = 0;
    Bitmask used = 0;    //该表达式使用的表的位码

    int nEq;             //可以使用索引的等值表达式的个数
    int bInEst = 0;      //如果存在 x IN (SELECT...),则设为true
    int nInMul = 1;      //处理IN子句
    int nBound = 100;    //估计需要扫描的表中的元素,100表示需要扫描整个表,范围条件意味着只需要扫
描表的某一部分
    int bSort = 0;       //是否需要排序
    int bLookup = 0;     //如果对索引中的每个列,需要对应的表进行查询,则为true

    /* Determine the values of nEq and nInMul */
    //计算nEq和nInMul值
    for(nEq=0; nEq<pProbe->nColumn; nEq++){
      WhereTerm *pTerm; /* A single term of the WHERE clause */
      int j = pProbe->aiColumn[nEq];
      pTerm = findTerm(pWC, iCur, j, notReady, eqTermMask, pIdx);
      if( pTerm==0 ) //如果该条件在索引中找不到,则break
          break;
      wsFlags |= (WHERE_COLUMN_EQ|WHERE_ROWID_EQ);
      if( pTerm->eOperator & WO_IN ){
        Expr *pExpr = pTerm->pExpr;
        wsFlags |= WHERE_COLUMN_IN;
        if( ExprHasProperty(pExpr, EP_xIsSelect) ){ //IN (SELECT...)
          nInMul *= 25;
          bInEst = 1;
        }else if( pExpr->x.pList ){
          nInMul *= pExpr->x.pList->nExpr + 1;
        }
      }else if( pTerm->eOperator & WO_ISNULL ){
        wsFlags |= WHERE_COLUMN_NULL;
      }
      used |= pTerm->prereqRight; //设置该表达式使用的表的位码
    }

    //计算nBound值
    if( nEq<pProbe->nColumn ){//考虑不能使用索引的列
      int j = pProbe->aiColumn[nEq];
      if( findTerm(pWC, iCur, j, notReady, WO_LT|WO_LE|WO_GT|WO_GE, pIdx) ){
        WhereTerm *pTop = findTerm(pWC, iCur, j, notReady, WO_LT|WO_LE, pIdx);
        WhereTerm *pBtm = findTerm(pWC, iCur, j, notReady, WO_GT|WO_GE, pIdx);//>=

        //估计范围条件的代价
        whereRangeScanEst(pParse, pProbe, nEq, pBtm, pTop, &nBound);
        if( pTop ){
          wsFlags |= WHERE_TOP_LIMIT;
```

```
      used |= pTop->prereqRight;
    }
    if( pBtm ){
      wsFlags |= WHERE_BTM_LIMIT;
      used |= pBtm->prereqRight;
    }
    wsFlags |= (WHERE_COLUMN_RANGE|WHERE_ROWID_RANGE);
  }
}else if( pProbe->onError!=OE_None ){//所有列都能使用索引
  if( (wsFlags & (WHERE_COLUMN_IN|WHERE_COLUMN_NULL))==0 ){
    wsFlags |= WHERE_UNIQUE;
  }
}

if( pOrderBy ){//处理order by
  if( (wsFlags & (WHERE_COLUMN_IN|WHERE_COLUMN_NULL))==0
    && isSortingIndex(pParse,pWC->pMaskSet,pProbe,iCur,pOrderBy,nEq,&rev)
  ){
    wsFlags |= WHERE_ROWID_RANGE|WHERE_COLUMN_RANGE|WHERE_ORDERBY;
    wsFlags |= (rev ? WHERE_REVERSE : 0);
  }else{
    bSort = 1;
  }
}

if( pIdx && wsFlags ){
  Bitmask m = pSrc->colUsed; //m为src使用的列的位图
  int j;
  for(j=0; j<pIdx->nColumn; j++){
    int x = pIdx->aiColumn[j];
    if( x<BMS-1 ){
      m &= ~(((Bitmask)1)<<x); //将索引中列对应的位清0
    }
  }
  if( m==0 ){//如果索引包含src中的所有列,则只需要查询索引即可
    wsFlags |= WHERE_IDX_ONLY;
  }else{
    bLookup = 1;//需要查询原表
  }
}

//估计输出行数,同时考虑IN运算
nRow = (double)(aiRowEst[nEq] * nInMul);
if( bInEst && nRow*2>aiRowEst[0] ){
  nRow = aiRowEst[0]/2;
  nInMul = (int)(nRow / aiRowEst[nEq]);
}

//代价为输出的行数+二分查找的代价
cost = nRow + nInMul*estLog(aiRowEst[0]);

//考虑范围条件影响
```

```
    nRow = (nRow * (double)nBound) / (double)100;
    cost = (cost * (double)nBound) / (double)100;

    //加上排序的代价:cost *log (cost)
    if( bSort ){
      cost += cost*estLog(cost);
    }

    //如果只查询索引,则代价减半
    if( pIdx && bLookup==0 ){
      cost /= (double)2;
    }

    //如果当前的代价更小
    if( (!pIdx || wsFlags) && cost<pCost->rCost ){
      pCost->rCost = cost; //代价
      pCost->nRow = nRow;  //估计扫描的元组数
      pCost->used = used; //表达式使用的表的位图
      pCost->plan.wsFlags = (wsFlags&wsFlagMask); //查询策略标志(全表扫描,使用索引进行扫描)
      pCost->plan.nEq = nEq; //查询策略使用等值表达式个数
      pCost->plan.u.pIdx = pIdx; //查询策略使用的索引(全表扫描则为NULL)
    }

    //如果SQL语句存在INDEXED BY,则只考虑该索引
    if( pSrc->pIndex ) break;

    /* Reset masks for the next index in the loop */
    wsFlagMask = ~(WHERE_ROWID_EQ|WHERE_ROWID_RANGE);
    eqTermMask = idxEqTermMask;
  }
```

SQLite的代价模型比较简单，而通用数据库一般是将基于规则的优化和基于代价的优化结合起来，更为复杂。