1. B 树(B-Tree)

实现的功能

1. 打开/关闭 chidb 文件
2. 从文件从读取 B 树的节点
3. 创建 B 树节点和将 B 树节点写入文件
4. 获取和插入 Cell 到 B 树节点中
5. 在 B 树中查找值
6. 向 B 树中插入 Cell

1.1 chidb 文件格式

1.1.1 逻辑组织

chidb 文件包含以下内容：

- 文件头。
- 0 个或更多表。
- 0 个或多个索引。

数据库文件中的每个表都存储为 B + -Tree（我们将它们简称为表 B-Trees）。该树的条目是数据库记录(对应于表中一行的值的集合)。每个条目的键将是其主键。由于表是 B + -Tree，因此内部节点不包含条目，而是用于导航树。

数据库文件中的每个索引都存储为 B 树（我们将其称为索引 B 树）。假设我们有关系 R(pk,…,ik,…) 哪里 pk 是主键，并且 ik 是我们要在其上创建索引的属性，索引 B-Tree 中的每个条目将是一个 (k1,k2) 一对，哪里 k1 是一个值 ik 和 k2 是主键的值（pk）在记录中 ik=k1。

1.1.2 物理组织

chidb 文件分为若干大小的页面 PageSize，从 1 开始编号。每个页面用于存储表或索引 B-Tree 节点。

页面包含带有页面相关元数据的页面标题,例如页面类型。标头未使用的空间可用于存储 单元，这些单元用于存储 B 树条目：

- 叶表单元格：⟨Key,DBRecord⟩，在哪里 DBRecord 是数据库记录，并且 Key 是它的主键。
- 内部表格单元：⟨Key,ChildPage⟩，在哪里 ChildPage 是包含键小于或等于的条目的页面的编号 Key。

- 叶索引单元格：⟨KeyIdx,KeyPk⟩，在哪里 KeyIdx 和 KeyPk 是 k1 和 k2，如先前定义。
- 内部索引单元格：⟨KeyIdx,KeyPk,ChildPage⟩，在哪里 KeyIdx 和 KeyPk 如上定义，并且 ChildPage 是包含键小于的条目的页面数 KeyIdx。

数据库中的页面 1 很特殊，因为文件头使用了它的前 100 个字节。因此，存储在页面 1 中的 B 树节点只能使用 (PageSize-100) 个字节。

B 树节点必须具有 n 键和 n+1 指针。但是，使用单元格，我们只能存储 n 指针。给定一个节点 B，则需要一个额外的指针来存储包含该节点的页面号 B′ 键大于所有的键 B。这个额外的指针存储在页面标题中，称为 RightPage。

表 B 树的逻辑和物理视图如图 Figure 1 所示。

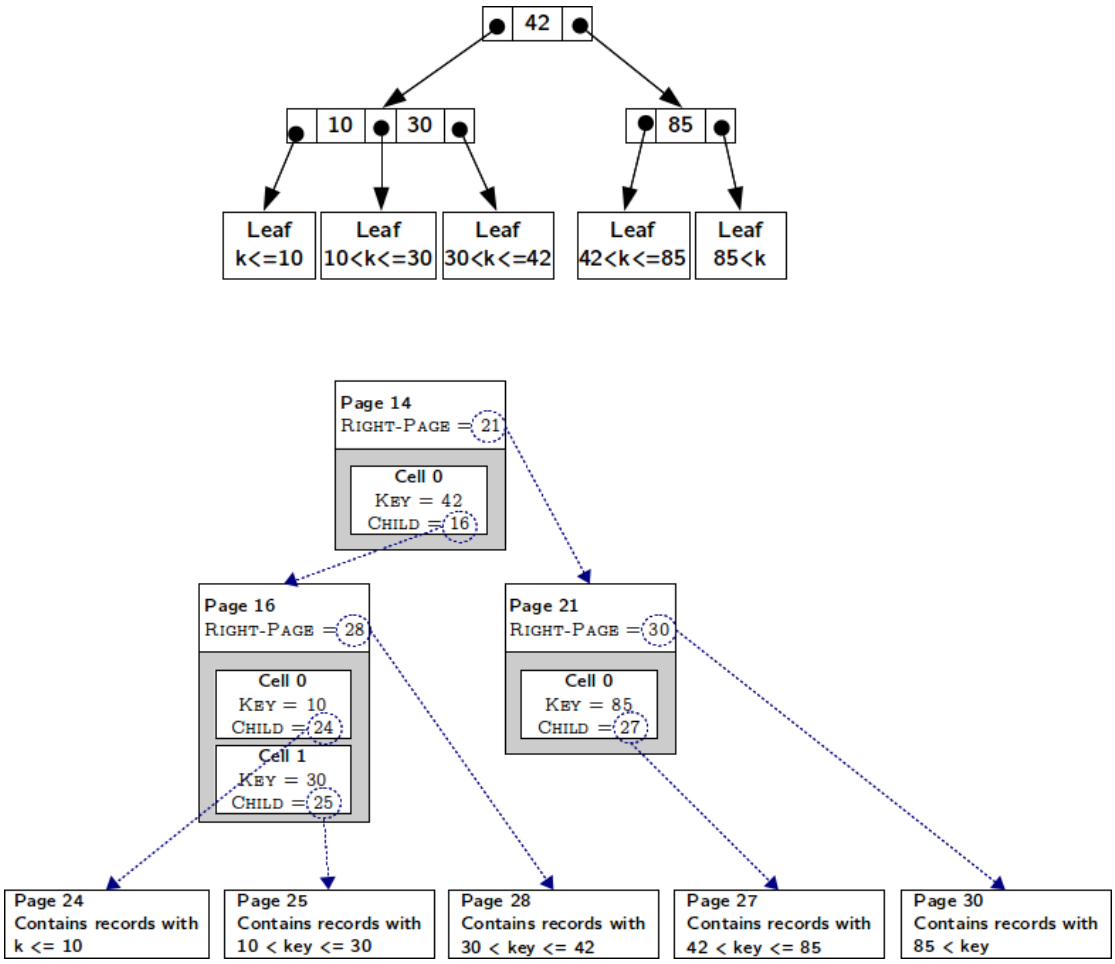


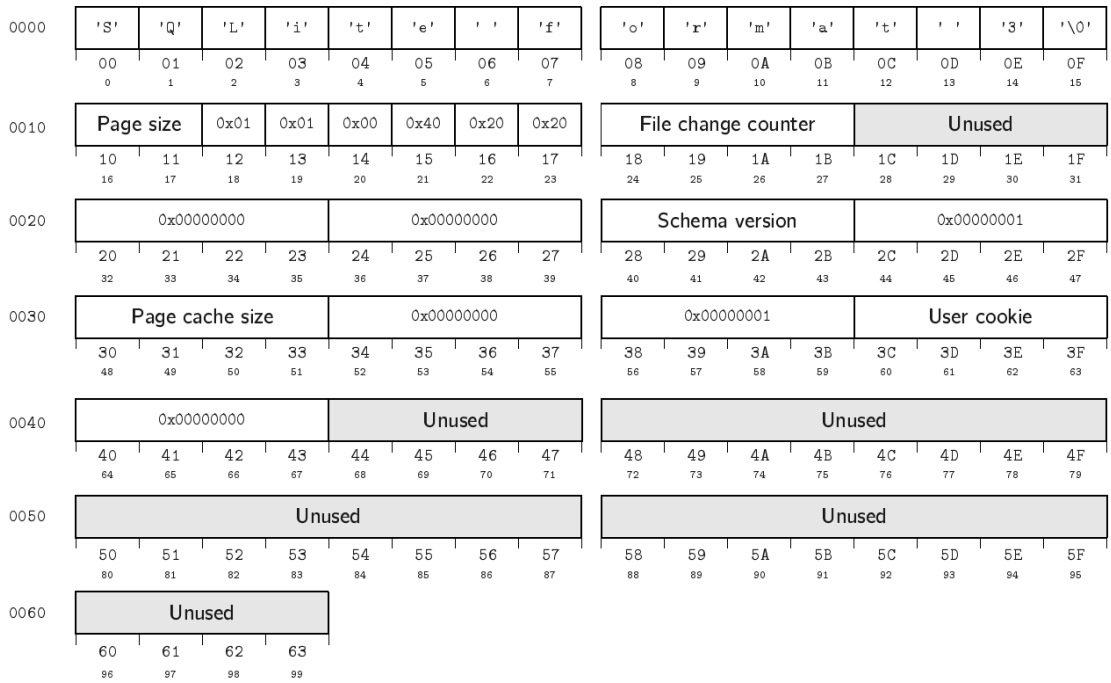
Figure 1

1.1.3 数据类型

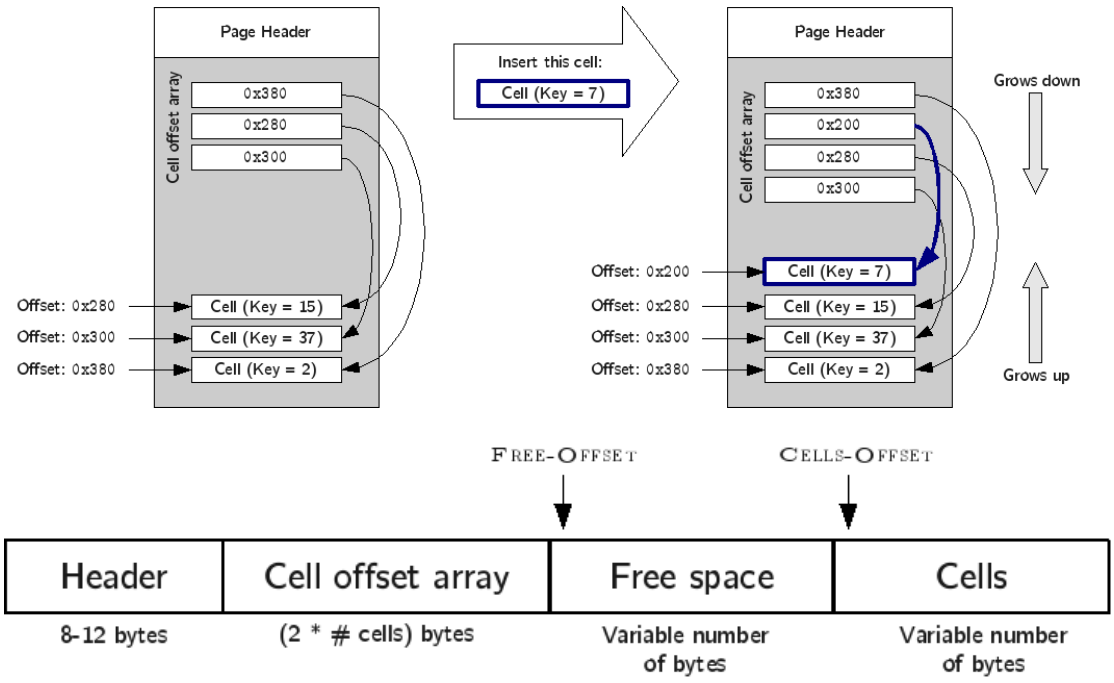
名称	描述	范围
`uint8`	无符号1字节整数	$0 \leq x \leq 255$
`uint16`	无符号2字节整数	$0 \leq x \leq 65,535$
`uint32`	无符号4字节整数	$0 \leq x \leq 2^{32} - 1$
`int8`	有符号的1字节整数	$-128 \leq x \leq 127$
`int16`	有符号2字节整数	$-32768 \leq x \leq 32767$
`int32`	有符号的4字节整数	$-2^{31} \leq x \leq 2^{31} - 1$
`varint8`	无符号1字节varint	$0 \leq x \leq 127$
`varint32`	无符号4字节varint	$0 \leq x \leq 2^{28} - 1$
`string(n)`	零终止的字符串	字符数 $\leq$ `n`
`raw-string(n)`	字符数组	字符数 $\leq$ `n`

1.1.4 文件头

chidb 文件的前 100 个字节包含带有该文件元数据的标头。该文件头使用与 SQLite 相同的格式，并且由于 chidb 中不支持许多 SQLite 功能，因此大多数头都包含常量值。标头的布局如下图所示。

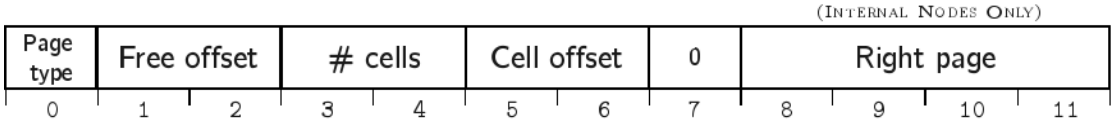


1.1.5 表页



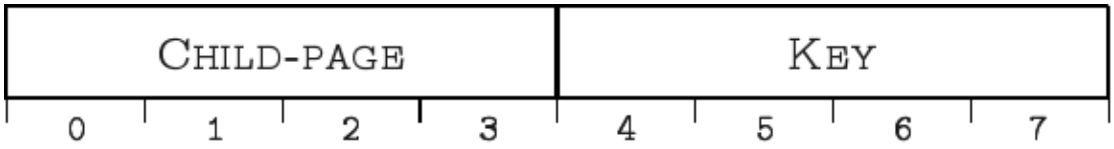
- 在页头位于页面的顶部，并包含有关页面的元数据。页头的确切内容将在后面说明。
- 所述 Cells 偏移数组在页头之后的位置。每个条目都存储为一个 uint16。因此，Cells 偏移数组的长度取决于页面中单元格的数量。
- Cells 位于页面的结尾。
- Cells 偏移数组与 Cells 之间的空间是可用空间 ，Cells 偏移数组可以增长(向下)，Cells 可以增长（向上)。

下图和表格总结了页头的布局：



1.1.6 Table Cell

下图和表中指定了内部表格单元的布局：



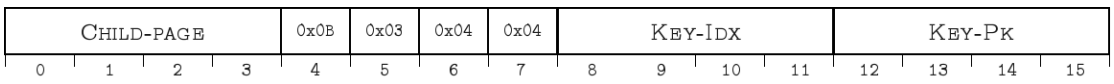
下图和表中指定了叶表单元格的布局：

1.1.7 索引

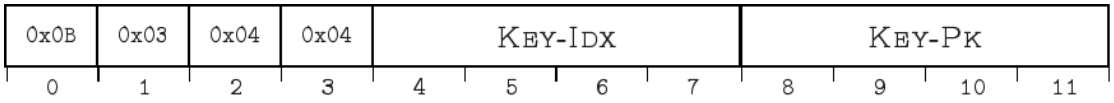
索引 B 树非常类似于表 B 树，因此我们解释过的重新映射表 B 树的大部分内容都适用于索引。主要区别如下：

- PageType 字段必须设置为适当的值（0x02 对于内部页面和 0x0A 叶子页面）
- 表存储为 B + -Tree（记录仅存储在叶节点中）时，索引存储为 B-Tree（记录存储在所有级别)。但是，索引不存储数据库记录，而是(KeyIdx,KeyPk)对

下图和表中指定了内部索引单元的布局：



下图和表中指定了叶索引单元的布局：



1.2 实现的功能及函数

1.2.1 打开/关闭 chidb 文件

```
/** 打开一个 B 树文件
 *
 * 这个函数打开一个数据库文件并且验证其文件头是否正确.
 * 如果文件是空的(或者文件不存在), 则
 * 1) 通过默认的 page size 初始化文件头
 * 2) 在页 1 上创建一个空的表叶节点
*/
int chidb_Btree_open(const char *filename, chidb *db, BTree **bt);
```

```
/** 关闭一个 B 树文件
 *
 * 这个函数关闭一个数据库文件, 释放内存中的资源, 比如 pager
*/
int chidb_Btree_close(BTree *bt);
```

1.2.2 从文件从读取 B 树的节点

```
/** 从硬盘中加载一个 B 树结点
 *
 * 从硬盘中读取一个 B 树结点. 所有关于结点的信息都被存储在 BTreeNode 结构体中.
 * 任何影响在 BTreeNode 变量上的改变直到 chidb_Btree_writeNode 被调用才会
 * 在数据库中起作用.
*/
int chidb_Btree_getNodeByPage(BTree *bt, npage_t npage, BTreeNode **node);
```

```
/** 释放分配给 B 树结点的内存
 *
 * 释放分配给 B 树结点的内存, 以及内存中存储的页.
*/
int chidb_Btree_freeMemNode(BTree *bt, BTreeNode *btn);
```

1.2.3 创建 B 树节点和将 B 树节点写入文件

```
/** 创建一个新的 B 树节点
 *
 * 在文件中分配一个新页面，并将其初始化为 B-Tree 节点。
*/
int chidb_Btree_newNode(BTree *bt, npage_t *npage, uint8_t type);
```

```
/** 初始化 B 树节点
 *
 * 初始化数据库页面以包含一个空的 B-Tree 节点。假定 Page 已存在，并且已经由 Pager 分配。
*/
int chidb_Btree_initEmptyNode(BTree *bt, npage_t npage, uint8_t type);
```

```
/** 将内存中的 B-Tree 节点写入磁盘
 *
 * 将内存中的 B-Tree 节点写入磁盘。为此，我们需要根据 chidb 页面格式更新内存页面。
 * 由于 Cell 偏移数组和单元格本身是直接在页面上修改的，
 * 因此唯一要做的就是将 “type”，“free_offset”，
 * “n_cells”，“cells_offset”
 * 和 “right_page” 的值存储在内存页。
*/
int chidb_Btree_writeNode(BTree *bt, BTreeNode *node);
```

1.2.4 获取和插入 Cell 到 B 树节点中

```
/** 读取 Cell 的内容
 *
 * 从 BTreeNode 读取单元格的内容，并将其存储在 BTreeCell 中。 这涉及以下内容：
 * 1. 找出所需 Cell 的偏移量。
 * 2. 从内存页面中读取 Cell，然后解析其内容。
*/
int chidb_Btree_getCell(BTreeNode *btn, ncell_t ncell, BTreeCell *cell);
```

```
/** 将新 Cell 插入 B 树节点
 *
 * 在指定位置 ncell 处将新单元格插入 B 树节点。这涉及以下内容：
 * 1. 将单元格添加到单元格区域的顶部
 * 2. 修改 BTreeNode 中的 cells_offset 以反映单元区域中的增长
 * 3. 修改 cells 偏移数组，以使位置 >= ncell 中的所有值在数组中向后移动一个位置
 *    然后，将位置 ncell 的值设置为新添加的 Cell 的偏移量
```

```
*/
int chidb_Btree_insertCell(BTreeNode *btn, ncell_t ncell, BTreeCell *cell);
```

1.2.5 在 B 树中查找值

```
/** 在表 B 树中查找条目
 *
 * 在表 B-Tree 中查找与给定键关联的数据
*/
int chidb_Btree_find(BTree *bt, npage_t nroot, key_t key, uint8_t **data, uint16_t
*size);
```

1.2.6 向 B 树中插入 Cell

```
/** 将条目插入表 B 树
 *
 * 它需要一个键和数据，并创建一个 BTreeCell，可以将其传递给 chidb_Btree_insert。
*/
int chidb_Btree_insertInTable(BTree *bt, npage_t nroot,
                              key_t key, uint8_t *data, uint16_t size);
```

```
/** 将条目插入索引 B 树
 *
 * 它使用一个 KeyIdx 和一个 KeyPk，并创建一个 BTreeCell，可以将其传递给
chidb_Btree_insert。
*/
int chidb_Btree_insertInIndex(BTree *bt, npage_t nroot, key_t keyIdx, key_t keyPk);
```

```
/** 将 BTreeCell 插入 B 树
 *
 * chidb_Btree_insert 和 chidb_Btree_insertNonFull 函数
 * 负责将新条目插入 B 树，chidb_Btree_insertNonFull 实际执行插入
 * chidb_Btree_insert，首先检查根是否必须拆分（拆分操作不同于拆分其他任何节点）
 * 如果是这样，则在调用 chidb_Btree_insertNonFull 之前调用 chidb_Btree_split
*/
int chidb_Btree_insert(BTree *bt, npage_t nroot, BTreeCell *btc);
```

```
/** 将 BTreeCell 插入有空闲空间的 B-Tree 节点
 *
 * chidb_Btree_insertNonFull 将 BTreeCell 插入到一个
 * 假设未满（即不需要拆分）的结点上。
 * 如果节点是叶节点，根据其键的位置将单元格直接添加到适当的位置
 * 如果该节点是内部节点，
```

```
 * 则函数将确定必须将其插入哪个子节点，
 * 并且在该子节点上递归调用自身。
 * 但是，在这样做之前它将检查子节点是否已满。
 * 如果是这样，则必须先将其拆分。
*/
int chidb_Btree_insertNonFull(BTree *bt, npage_t npage, BTreeCell *btc);
```

```
/** 切分 B 树结点
 *
 * 拆分 B 树节点 N. 这涉及以下内容：
 * - 在 N 中找到处于中间位置的 Cell
 * - 创建一个新的 B 树节点 M
 * - 将中间位置 Cell 之前的单元格移至 M（如果该单元格是表格叶单元格，那么中间位置的 Cell 也将移动）
 * - 使用中键和 M 的页码将一个单元格提升到父结点中。
*/
int chidb_Btree_split(BTree *bt, npage_t npage_parent, npage_t npage_child,
                                 ncell_t parent_cell, npage_t *npage_child2);
```

2. 数据库机器(Database Machine)

2.1 数据流

在本项目中，数据库机是实际操作数据库文件的部分，DBM 相当于一个 cpu，用于逐条处理 dbm 指令，操作寄存器和游标来返回 sql 语句的结果。

dbm 对用户透明，用户使用 SQL 操作数据库，而真正操作具体数据的是 dbm 指令。

其输入输出如图 2-1-1 所示

图 2-1-1

2.2 DBM 中的数据结构,

如图 2-2-1 所示，一个 dbm 中含有无穷的寄存器和游标，在 dbm 运行时 pc 从 op 这个数组中逐条取出命令并执行，每条指令都可能对某个寄存器或游标进行操作

① 寄存器可以储存整数，字符串或者无类型的比特串

② 游标指向某个表的具体项，并且可以操作 cursor 快速移动，

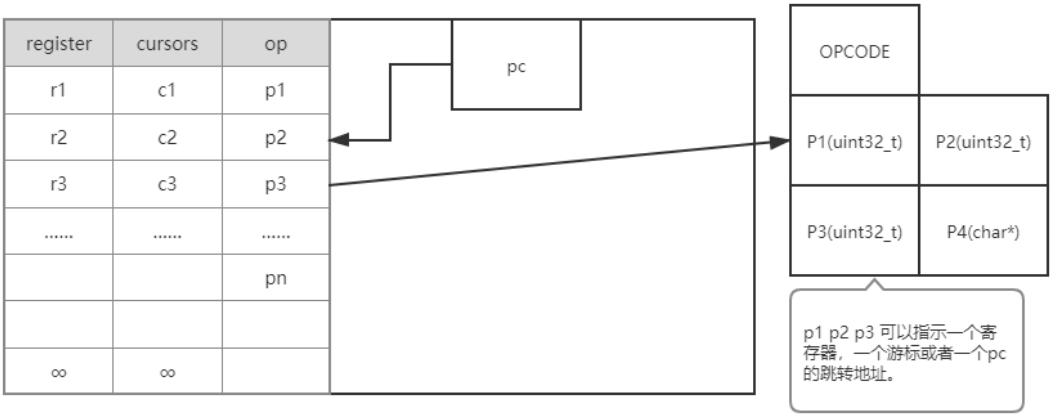


图 2-2-1

游标的结构如图 2-2-2 所示，每个游标都具体指向表中某元组(current_cell),并且储存该表对应 B 树的根节点(root_page)和表所含列数(n_cols)

最后是一个用于记录从根节点到含有该游标所指元组的路径的一个链表 trail,在 trail 中，

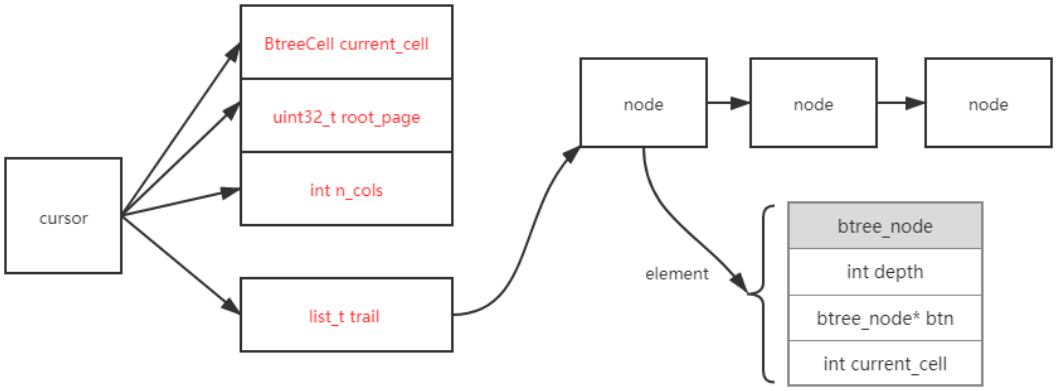


图 2-2-2

其元素记录了当前 B 树深度(depth),所在的 B 树节点以及用于指示路径的 current_cell 标识

2.3 cursor 移动的实现

虽然在一个表中的每一项记录在逻辑上是连续的,但在物理储存结构上两条记录可能不相连,所以在对游标进行移动操作时会涉及到切换结点的问题,在游标中使用链表记录从根节点到对应 B 树叶节点的路径。根据链表中的记录,能够快速找到其父节点,从而能够快速完成叶节点的切换。

游标进行 next 或 prev 的流程如图 2-3-1 所示

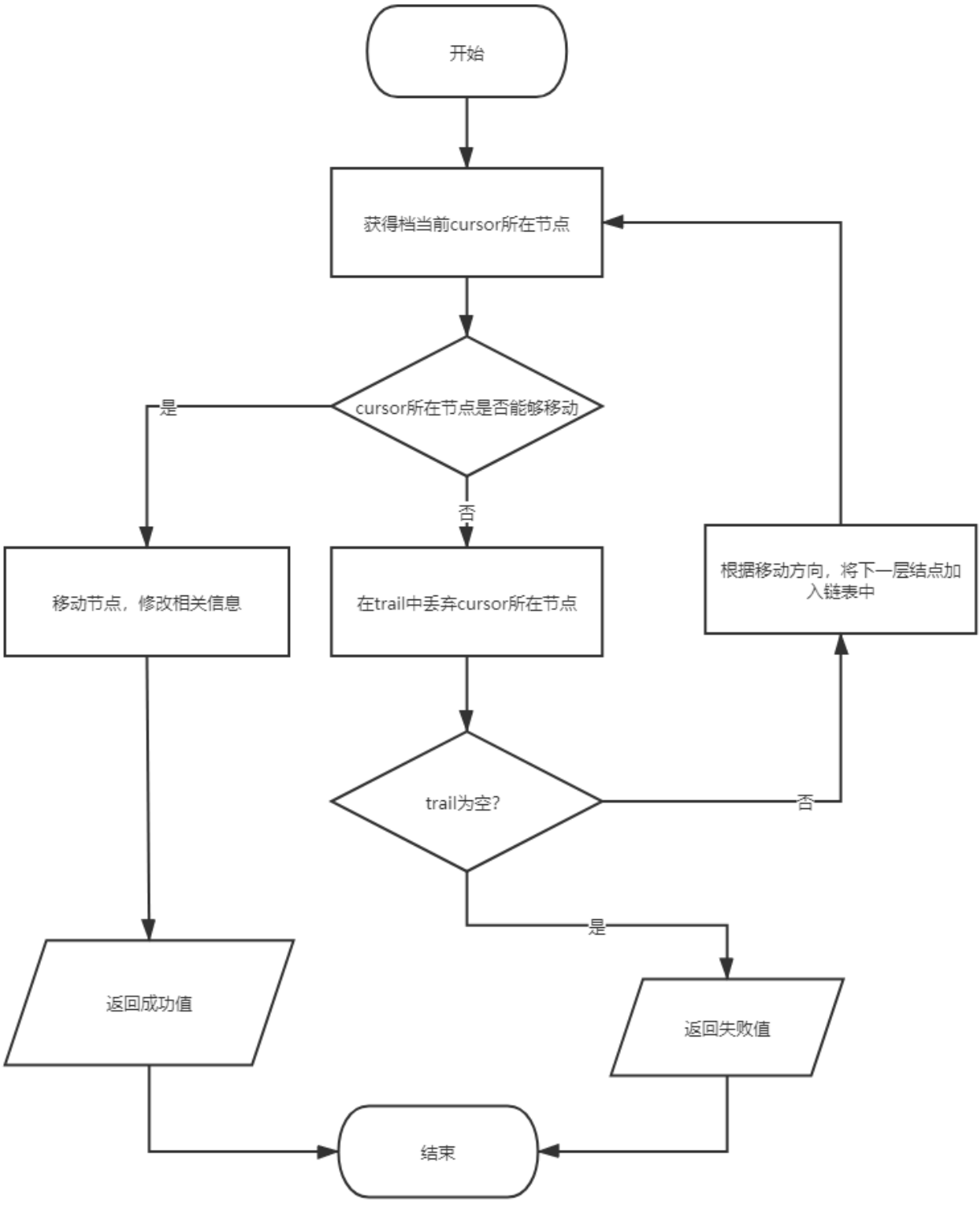


图 2-3-1

对 cursor 进行 seek 操作的流程图如图 2-3-2 所示

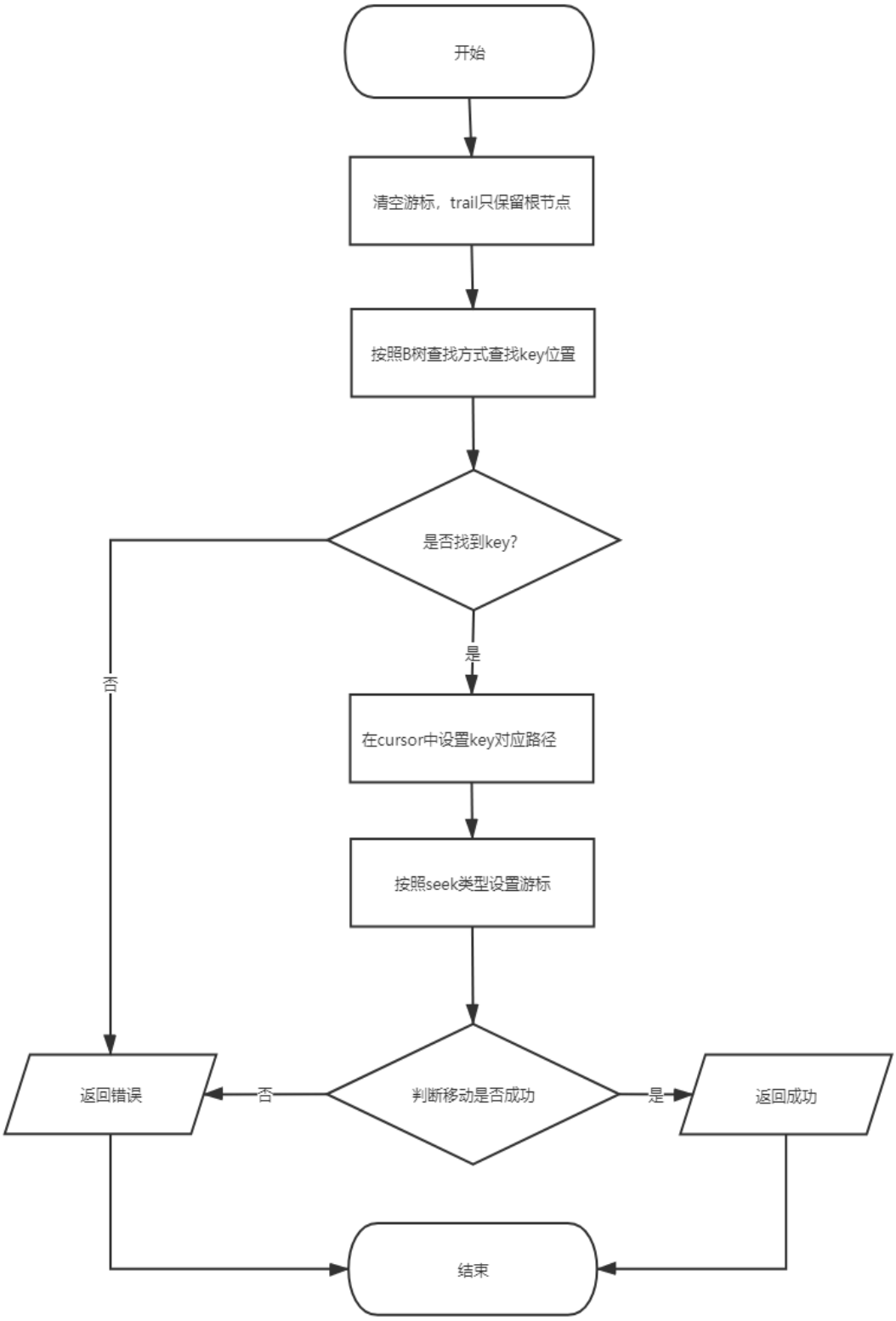


图 2-3-2

2.4 dbm 向上提供的接口函数及参数

所有的 dbm 操作函数都只有两个参数 stmt 和 op 但不同的指令格式不同

//通过 op->opcode 返回执行对应命令的函数

```
int chidb_dbm_op_handle (chidb_stmt *stmt, chidb_dbm_op_t *op)
```

/*寄存器赋值操作类

*Integer String Null Copy SCopy

*与数据库无关，仅仅是根据命令参数向对应的寄存器赋值

*Integer op->p1 value op->p2 r

*将 p1 中的值放入 指示的寄存器中

*String op->p1 length op->p2 r op->p4 string

*将长度为 Length 的字符串存入寄存器 r

*/

```
int chidb_dbm_op_Integer (chidb_stmt *stmt, chidb_dbm_op_t *op)
int chidb_dbm_op_String (chidb_stmt *stmt, chidb_dbm_op_t *op)
int chidb_dbm_op_Null (chidb_stmt *stmt, chidb_dbm_op_t *op)
```

/*寄存器操作函数

*op->p1 r1 op->p2 r2

将寄存器 r1 中的内容拷贝到 r2 中 Scopy 是浅拷贝/

```
int chidb_dbm_op_Copy (chidb_stmt *stmt, chidb_dbm_op_t *op)
int chidb_dbm_op_SCopy (chidb_stmt *stmt, chidb_dbm_op_t *op)
```

/*指令跳转相关函数

*Eq Ne Lt Le Gt Ge

*与数据库无关，仅仅是对于指令是否按照给定的值跳转

*op->p1 r1 op->p3 r2 op->p2 address_j

比较 r1 r2 两个寄存器中的值，如果比较结果与对应命令相同，则将指令跳转到 address_j 的位置/

```
int chidb_dbm_op_Eq (chidb_stmt *stmt, chidb_dbm_op_t *op)
int chidb_dbm_op_Ne (chidb_stmt *stmt, chidb_dbm_op_t *op)
int chidb_dbm_op_Lt (chidb_stmt *stmt, chidb_dbm_op_t *op)
int chidb_dbm_op_Le (chidb_stmt *stmt, chidb_dbm_op_t *op)
int chidb_dbm_op_Gt (chidb_stmt *stmt, chidb_dbm_op_t *op)
int chidb_dbm_op_Ge (chidb_stmt *stmt, chidb_dbm_op_t *op)
```

/*根据给定的 page 打开\关闭游标

*op->p1 cursor 1 op->p2 column_number

打开指定游标，并记录对应的行数/

```
int chidb_dbm_op_OpenRead (chidb_stmt *stmt, chidb_dbm_op_t *op)
int chidb_dbm_op_OpenWrite (chidb_stmt *stmt, chidb_dbm_op_t *op)
int chidb_dbm_op_Close (chidb_stmt *stmt, chidb_dbm_op_t *op)
```

/*重置游标,使其指向第一个单元

```
*Rewind
*如果游标对应的 B 树为空，则执行跳转
*op->p1 cursor c op->p2 address_j*/
int chidb_dbm_op_Rewind (chidb_stmt *stmt, chidb_dbm_op_t *op)

/*移动游标
*Next Prev Seek SeekGt SeekLt SeekGe SeekLe
*op->p1 cursor c op->p2 address_j op->p3 key
*按照给定的 key 对游标进行一次移动操作
*移动失败则进行指令跳转*/
int chidb_dbm_op_Next (chidb_stmt *stmt, chidb_dbm_op_t *op)
int chidb_dbm_op_Prev (chidb_stmt *stmt, chidb_dbm_op_t *op)
int chidb_dbm_op_Seek (chidb_stmt *stmt, chidb_dbm_op_t *op)
int chidb_dbm_op_SeekGt (chidb_stmt *stmt, chidb_dbm_op_t *op)
int chidb_dbm_op_SeekGe (chidb_stmt *stmt, chidb_dbm_op_t *op)
int chidb_dbm_op_SeekLt (chidb_stmt *stmt, chidb_dbm_op_t *op)
int chidb_dbm_op_SeekLe (chidb_stmt *stmt, chidb_dbm_op_t *op)

/*以索引为判断条件的跳转函数
*op->p1 cursor c op->p2 address_j op->p3 Idkey
*判断成功后执行跳转*/
int chidb_dbm_op_IdxGt (chidb_stmt *stmt, chidb_dbm_op_t *op)
int chidb_dbm_op_IdxGe (chidb_stmt *stmt, chidb_dbm_op_t *op)
int chidb_dbm_op_IdxLt (chidb_stmt *stmt, chidb_dbm_op_t *op)
int chidb_dbm_op_IdxLe (chidb_stmt *stmt, chidb_dbm_op_t *op)
int chidb_dbm_op_IdxPKey (chidb_stmt *stmt, chidb_dbm_op_t *op)

/*返回表中的结果
*Column op->p1 cursor c op->p2 column n op->p3 register r
*将游标 c 所指的某个单元格中第 n 列数据存入寄存器 r 中 (n 从 0 开始)*/
int chidb_dbm_op_Column (chidb_stmt *stmt, chidb_dbm_op_t *op)

/*返回 cursor 所指项的关键字值
*op->p1 cursor c op->p3 register r
*如果 cursor 所指向的项是索引项，则返索引对应的值 Pkey*/
int chidb_dbm_op_Key (chidb_stmt *stmt, chidb_dbm_op_t *op)

/*
*记录要返回的其实行号和行数
*op->p1 StartRow op->p2 nRR*/
int chidb_dbm_op_ResultRow (chidb_stmt *stmt, chidb_dbm_op_t *op)

/*
*向表中添加记录
```

*op->p1 startregister r op->p2 number n op->p3 register r2
向表中添加从 r1 到 r1+n-1 中所记录的数据并且将这些记录存入 r2 中/
int chidb_dbm_op_MakeRecord (chidb_stmt *stmt, chidb_dbm_op_t *op)

/*向表中添加一个新的记录
*op->p1 cursor op->p2 register r1 op->p3 r2
*获得 r1 所存的数据，并将其存入 r2 所指示的关键字的表中，并且 cursor 指向新的单元
*/
int chidb_dbm_op_Insert (chidb_stmt *stmt, chidb_dbm_op_t *op)
int chidb_dbm_op_IdxInsert (chidb_stmt *stmt, chidb_dbm_op_t *op)

/*创建表
*op->p1 register r1
在 B 树中添加一个新的节点，并且将这个节点对应的页号存入 r1 中/
int chidb_dbm_op_CreateTable (chidb_stmt *stmt, chidb_dbm_op_t *op)
int chidb_dbm_op_CreateIndex (chidb_stmt *stmt, chidb_dbm_op_t *op)

//中断 DBM，退出程序
int chidb_dbm_op_Halt (chidb_stmt *stmt, chidb_dbm_op_t *op)

3. 代码生成(Code Generation)

完成以下功能:

1. 实现读取 Schema 表
2. 实现简单 Select 语句到 DBM 指令的代码生成
3. 实现简单 Insert 语句到 DBM 指令的代码生成
4. 实现 Create Table 语句到 DBM 指令的代码生成

3.1 例子

```
CREATE TABLE products(code INTEGER PRIMARY KEY, name TEXT, price INTEGER);
```

翻译为

```
Integer      1  0  _  _
OpenWrite    0  0  5  _

CreateTable  4  _  _  _

String       5  1  _  "table"
String       8  2  _  "products"
String       8  3  _  "products"
String       73 5  _  "CREATE TABLE products(code INTEGER PRIMARY KEY, name TEXT,
price INTEGER)"

MakeRecord   1  5  6  _
Integer      1  7  _  _

Insert       0  6  7  _

Close        0  _  _  _
```

3.2 数据流图

其中词法分析器和语法分析器由 chidb 提供

3.3 具体实现

3.3.1. 读取 Schema 表

3.3.1.1 定义 Schema Item 类型

在 chidbInt.h 中定义 chidb_schema_item_t 类型, 结构定义如下

```
typedef struct
{
    char *type;
    char *name;
    char *assoc;
    int root_page;
    chisql_statement_t *stmt;
} chidb_schema_item_t;
```

其包含了 Schema 表中每一行的记录信息

3.3.1.2 修改 chidb 结构体

在 chidbInt.h 中修改 struct chidb 的定义

从

```
struct chidb
{
    BTree   *bt;
};
```

修改为

```
typedef list_t chidb_schema_t;

struct chidb
{
    BTree   *bt;
    chidb_schema_t schema;
    int need_refresh;
};
```

在 chidb 结构体中增加了一个包含记录的列表和表示是否需要更新的成员 need_refresh, 其中 need_refresh 会在 Create Table 语句之后置为 1, 会导致重新读取 Schema 表

3.3.1.3 实现读取 Schema 表

在 src/libchidb/api.c 中声明函数

```
int load_schema(chidb *db, npage_t nroot);
```

函数定义步骤如下:

1. 读取页码为 nroot 的页，遍历页中所有的 cells
2. 如果当前结点是表内部结点，对其 child_page 调用 load_schema
3. 如果当前结点是叶子结点，则解析其包含的数据，加入到 schema 中
4. 遍历完成之后，如果结点非叶子结点，则对其 right_page 调用 load_schema

3.3.1.4 修改打开和关闭文件时的过程

修改 src/libchidb/api.c 中函数 chidb_open 和 chidb_close 的定义
分别添加读取 Schema 和释放 Schema 的步骤

3.3.2. 简单 Select 语句的代码生成

chisql 中，sql 语句会被解析成 chisql_statement_t 类型的数据，其中 Select 语句的 type 字段值为 STMT_SELECT，而其结构被解析成递归定义的 SRA 结构，存储在 chisql_statement_t.select 中，SRA 结构的递归定义如下

```
data SRA = Table TableReference
         | Project SRA [Expression]
         | Select SRA Condition
         | NaturalJoin [SRA]
         | Join [SRA] (Maybe JoinCondition)
         | OuterJoin [SRA] OJType (Maybe JoinCondition)
         | Union SRA SRA
         | Except SRA SRA
         | Intersect SRA SRA

data OJType = Left
            | Right
            | Full

data ColumnReference = ColumnReference (Maybe String) String
data TableReference = TableName String (Maybe String)
data JoinCondition = On Condition
                   | Using [String]
```

在本实践中，只实现一个受限的 Select 语句的子集，包含以下限制：

1. 查询只包含一个单独的表
2. 查询包含若干个列或者是 '*'
3. 查询可以包含 where 子句，但是 where 只会包含一个单独的条件，并且其格式为 column op value，其中运算符只可能是 =，>，>=，< 或者 <=，以及值的类型只能是整型或字符串类型

由于上述限制的存在，本实践中可解析得到的 SRA 结构只会是以下两种形式：

1. 不包含 where 子句的查询

```
Project([columns or '*'],
    Table(table_name)
)
```

2. 包含 where 子句的查询

```
Project([columns or '*'],
    Select(column op value,
        Table(table_name)
    )
)
```

3.3.2.1 错误检查

在进行具体的代码生成之前，需要先对解析后的结果进行检查，检查包含三个步骤:

1. 要查询的表必须存在
2. 要查询的列必须存在于要查询的表中
3. 如果 where 子句存在，列的类型必须和所给的值的类型相同

3.3.2.2 代码生成

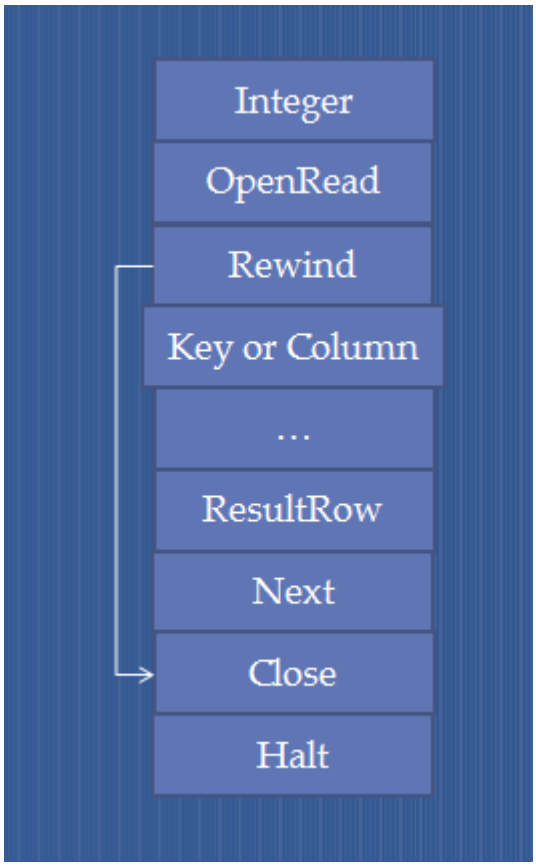


Figure 2

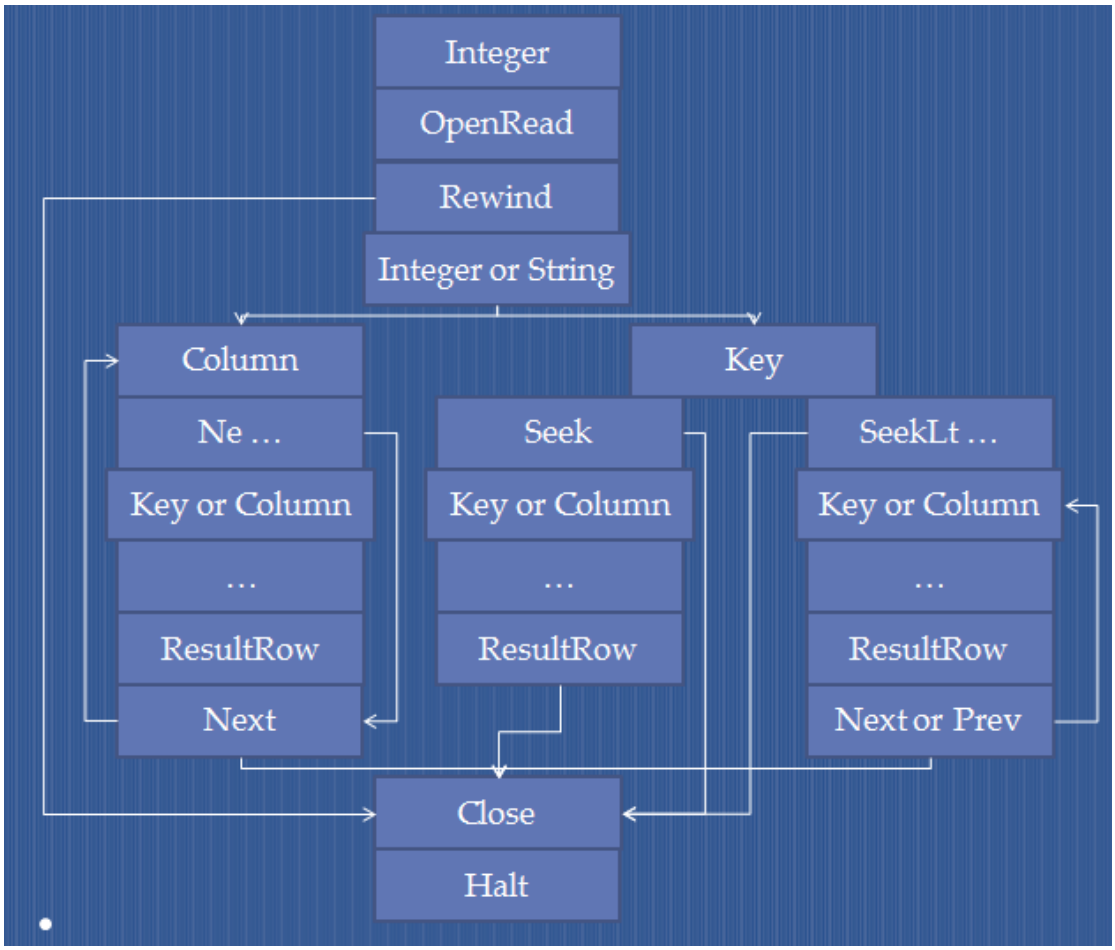


Figure 3

Figure1 为无 Where 子句时生成的示例代码，Figure2 为有 Where 子句时示意图。

1. 根据表名查找到表所在的根页码 nroot，生成 Integer 指令存储 nroot 的值到寄存器并生成 OpenRead 指令读取出 nroot 并以只读模式打开其对应的页与游标相对应
2. 创建 Rewind 指令，当表为空时跳转到结束指令（Close），但目前其跳转目标暂时不确定，记为 n1
3. 若 where 子句存在，将比较中的值通过相应指令（Integer 或 String）存储到寄存器中，若列为第一列则生成 Key 指令，并通过 Seek 指令簇 完成比较，跳转目标记为 n2，标记 after_next 为 1，否则通过 Column 指令获取对应的列，并通过 Eq 等指令完成比较，生成的比较指令的跳转目标记为 n2，需要跳转到 Next 或者 Prev 指令，同时将后续可能生成的 Next 或 Prev 指令的跳转目标记为 n3，当列为第一列时 n3 为比较指令的下一条，否则 n3 指向 Column 指令，特殊的，当列为第一列且比较为相等比较时，只产生 Seek 指令且不后续不产生 Next 或 Prev 指令，故 n3 设为 -1
4. 遍历要查询的列名，并获取其在表中的位置，通过 Column 指令存储到寄存器中
5. 通过 ResultRow 指令将上一步存储到寄存器中的值生成一条记录存储在寄存器中
6. 当 n3 为 -1 时，不需要 Next 或 Prev 指令，当第 3 步产生的比较指令是 SeekLe 或 SeekLt 时产生一条 Prev 指令，否则产生 Next 指令，其跳转目标为

n3，当 after_next 值为 1 时，n2 指向 Next 或 Prev 指令之后，否则指向 Next 或 Prev 指令

7. 生成 Close 指令关闭游标关联的页
8. 生成 Halt 指令停机

在上述步骤完成之后定义结果集，设置结果集的起始寄存器和寄存器的个数为查找的列数，为列申请空间并拷贝列名

3.3.3. 简单 Insert 语句的代码生成

在本实践中，只实现一个受限的 Insert 语句的子集，包含以下限制：

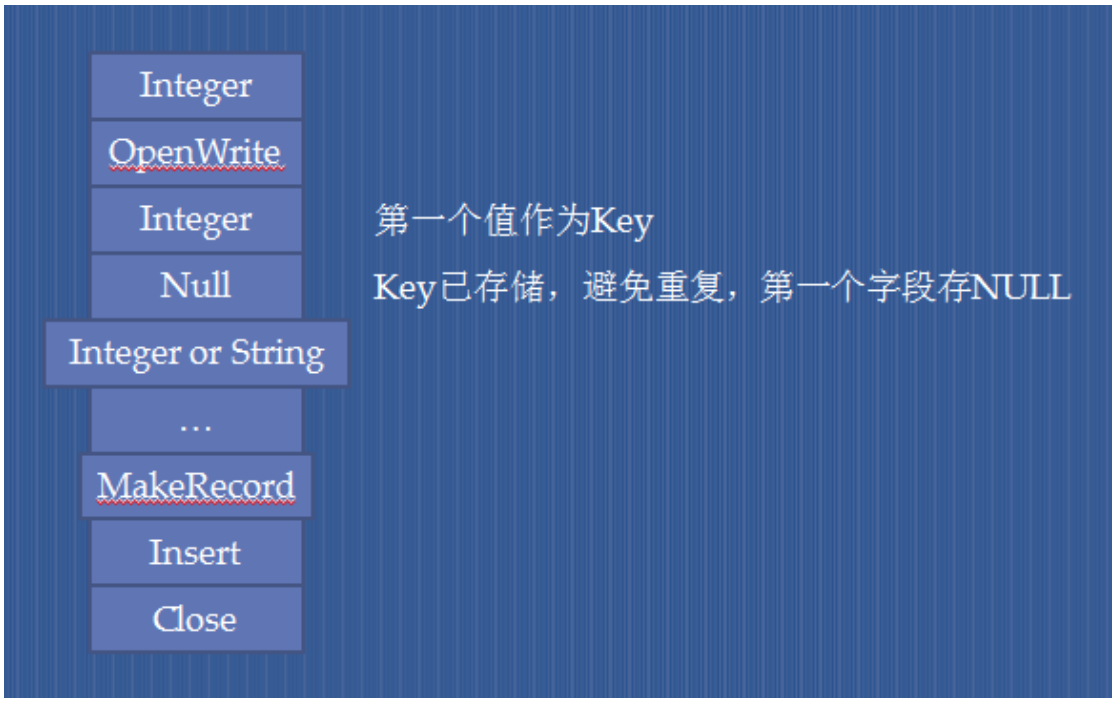
1. Insert 语句总是包含所有列的值，没有默认值
2. 表名之后不跟随若干列名，换而言之，只支持 Insert Into table_name Values(values...);
3. 插入的值只支持整型和字符串类型

3.3.3.1 错误检查

在进行具体的代码生成之前，需要先对解析后的结果进行检查，检查包含两个步骤：

1. 要插入的值的表必须存在
2. 值的类型必须和列的类型相匹配

3.3.3.2 代码生成



1. 生成 Integer 和 OpenWrite 指令以读写模式打开要插入值的表所在的根页
2. 第一列为 Key，生成 Integer 指令，因为 Key 与第一列重复，故为记录中第一列值生成 Null 指令
3. 遍历剩余的值，通过与值类型对应的指令（Integer 或 String）存储在连续的寄存器上
4. 将上述的值通过 MakeRecord 指令生成一行记录存储在寄存器中
5. 生成 Insert 指令将记录与 Key 插入到对应的表上
6. 生成 Close 指令关闭打开的页

3.3.4. Create Table 语句的代码生成

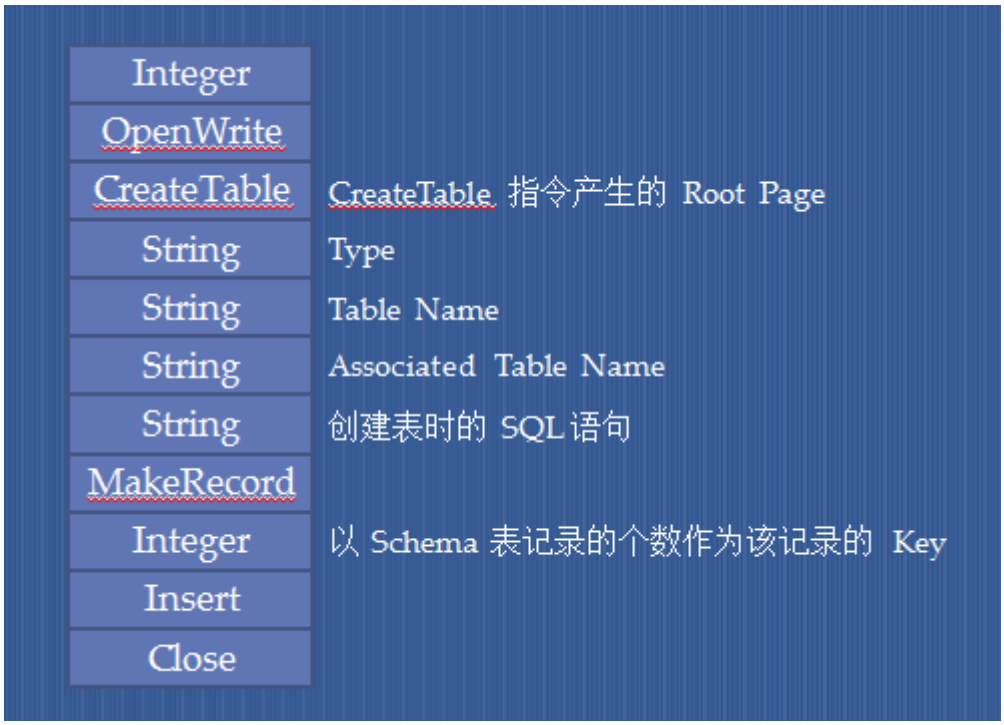
在本实践中，只实现一个受限的 Create Table 语句的子集，包含以下限制：

1. 每一列都以 列名 类型 的格式定义，类型可以是整型或字符串类型
2. 第一列总是主键约束的整型 INTEGER PRIMARY KEY
3. 表中不再有其他的约束

3.3.4.1 错误检查

只检查一项，即要创建的表名是否已经存在

3.3.4.2 代码生成



1. 生成 Integer 和 OpenWrite 指令以读写模式打开 Schema 表

2. 通过 CreateTable 指令新建一个表，并将其所在页码存储在寄存器 4 上
3. 按照顺序分别将创建的类型("table")，表名，关联名，SQL 语句 存储在寄存器 1，2，3，5 上
4. 生成 MakeRecord 指令将寄存器 1-5 上的值生成一条记录
5. 生成 Integer 指令存储 Schema 记录的个数作为新插入记录的 Key
6. 生成 Insert 指令将记录和 Key 插入进 Schema 表中
7. 生成 Close 指令关闭打开的页

4. 查询优化(Query Optimization)

4.1 优化条件

将选取操作尽量后移，实现的查询优化仅针对以下情况：

即当查询语句的 SRA 形如

```
Project([*],
    Select(t.a > int 10,
        NaturalJoin(
            Table(t),
            Table(u)
        )
    )
)
```

即查询语句格式为 SELECT (columns or *) from table1 |><| table2 where table1.column op value; 时，可将得到的 SRA 优化为如下结构：

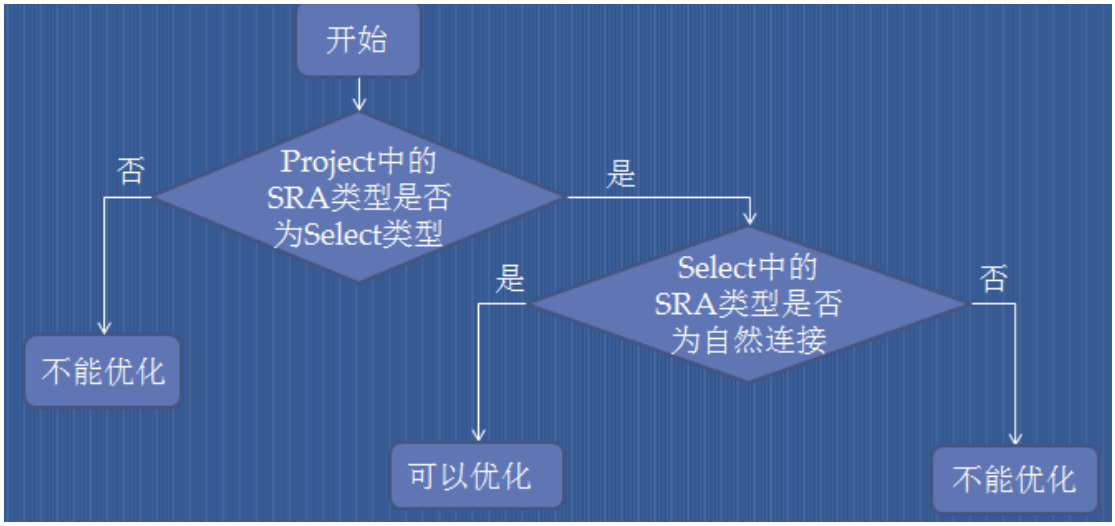
```
Project([*],
    NaturalJoin(
        Select(t.a > int 10,
                Table(t)
        ),
        Table(u)
    )
)
```

即将先自然连接后选取优化为先进行选取操作后再进行自然连接操作

4.2 条件检查

判断解析 SQL 语句得到的 SRA 结构是否满足优化条件，若满足则进行选取操作后移优化，不满足则直接复制原 SRA 结构

按照以下流程进行条件检查：



4.3 选取后移 (Sigma Push)

1. 获取原 SRA 中 Select 中条件中的左边值所处的列引用
2. 获取原 SRA 中的自然连接中的两个表引用
3. 与列引用中相同的表名记为 table1，另一个记为 table2
4. 为优化后的 SRA 及成员分配空间
5. 新 SRA 中的自然连接的右边设置为 Table 类型的 SRA，其连接的表名为 table2
6. 新 SRA 中的 Select 部分的条件与原 SRA 中的条件相同，Select 中的 Table 连接的表名为 table1