**代码生成**

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

**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 \_ \_ \_

**数据流图**



其中词法分析器和语法分析器由 chidb 提供

**具体实现**

**1. 读取 Schema 表**

**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

**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

**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

**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

**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)

)

1. 包含 where 子句的查询

Project([columns or '\*'],

Select(column op value,

Table(table\_name)

)

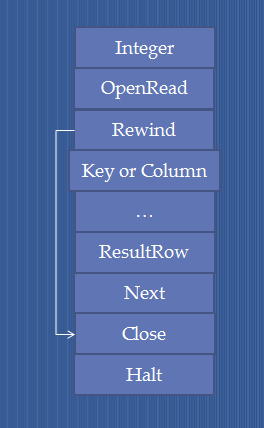
)

**2.1 错误检查**

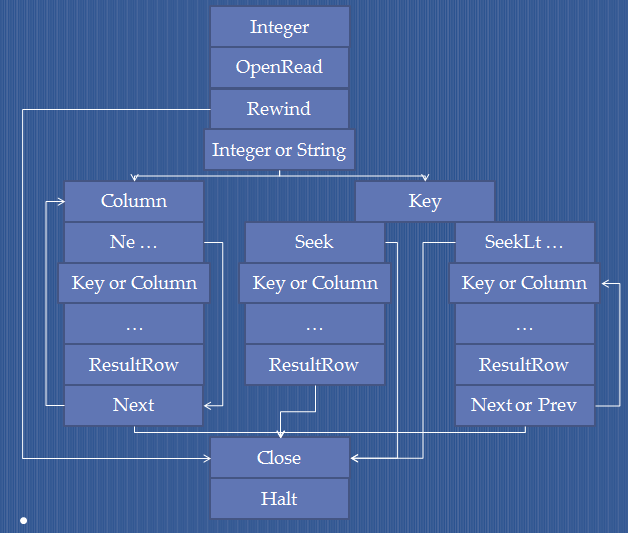
在进行具体的代码生成之前, 需要先对解析后的结果进行检查, 检查包含三个步骤:

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

**2.2 代码生成**



Figure



Figure

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. 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:

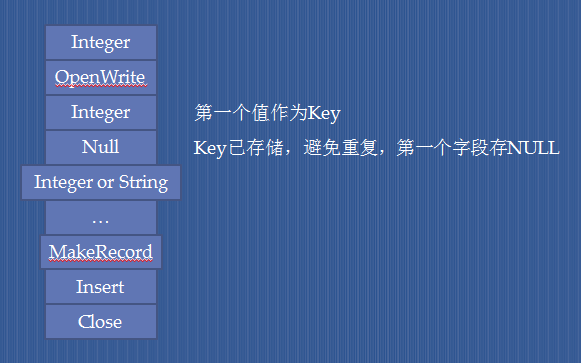
1. Insert 语句总是包含所有列的值, 没有默认值
2. 表名之后不跟随若干列名, 换而言之, 只支持 Insert Into table\_name Values(values...);
3. 插入的值只支持整型和字符串类型

**3.1 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

**3.2 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

**4. Create Table 语句的代码生成**

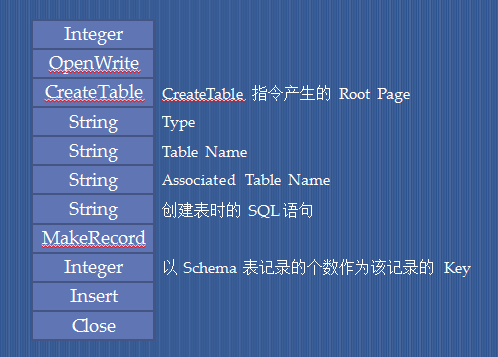
在本实践中, 只实现一个受限的 Create Table 语句的子集, 包含以下限制:

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

**4.1 错误检查**

只检查一项, 即要创建的表名是否已经存在

**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