DBMS 期末实验报告

章凌豪 13307130225

实验环境

虚拟机：VMware Workstation 9.0
系　统：Debian 8
处理器：2.3GHz 单核
内　存：512 MB

提交文件说明

src/execnodes.h	修改的代码文件
src/funcapi.c	
src/nodeNestloop.c	
src/nodeNestloop.h	
src/pg_proc.h	
src/postgresql.conf	数据库配置文件
test_p1/combined.txt	Part1 的运行结果
test_p1/jaccard.txt	
test_p1/levenshtein.txt	
test_p1/p1test1.sql	Part1 的测试语句
test_p1/p1test2.sql	
test_p1/p1test3.sql	
test_p2/p2test.sql	Part2 的测试语句
test_p2/reset_stat.sql	查看 pg_stat_statements 的语句
test_p2/show_stat.sql	
script.txt	实验过程中用到的命令，内有详细说明

Part 1: Similarity Join

实现

① 在 **src/include/catalog/pg_proc.h** 中添加以下代码

```
DATA(insert OID = 4375 ( levenshtein_distance PGNSP PGUID 12 1 0 0 f f f t f i 2 0 20 "25 25" _null_ _null_ _null_ _null_ levenshtein_distance _null_ _null_ _null_ ));
DESCR("levenshtein_distance");
DATA(insert OID = 4376 ( jaccard_index PGNSP PGUID 12 1 0 0 f f f t f i 2 0 700 "25 25" _null_ _null_ _null_ _null_ jaccard_index _null_ _null_ _null_ ));
DESCR("jaccard_index");
```

② 在 **src/backend/utils/fmgr/funcapi.c** 中实现所要求的两个函数。其中值得注意的是，**jaccard_index()**要返回 **FLOAT4**，**levenshtein_distance()**要返回 **INT64**，这是与 **pg_proc.h** 中的定义一致的，否则会出错。

③ 阅读文档可以知道，对于 PostgreSQL 的内建类型 text，可以通过以下代码获取相关信息：

```
// 获取参数指针
text *ARG_A = (text *)PG_GETARG_TEXT_P(0);
text *ARG_B = (text *)PG_GETARG_TEXT_P(1);
// 获取参数长度
int N = VARSIZE(ARG_A) - VARHDRSZ;
int M = VARSIZE(ARG_B) - VARHDRSZ;
// 获取内容指针
char *A = VARDATA(ARG_A);
char *B = VARDATA(ARG_B);
```

以下均用 A、B 表示两个字符串，N、M 表示它们的长度。

④ 由于要求进行不区分大小写的比较，对所有字符都取了 **toupper()**。

算法

① **jaccard_index**

算法的主要步骤如下：

1. 将 A 和 B 分别拆成 bigram，这里要去重
2. 统计 A 和 B 各自有几个 bigram
3. 统计 A 和 B 有几个共同的 bigram
4. 根据公式计算 Jaccard Index

考虑到对字符串进行操作比较繁琐，同时因为取的 token 是 bigram，而一个 bigram 可以映射到一个大小为字符集平方的空间。在这里我们可以假设输入均为 ASCII 字符，所以可以将一个 bigram 映射到 0~65535(256^2 - 1)的整数，从而方便后续的统计。

进一步地，我们可以用两个 Bitmap 来分别记录 A 和 B 有哪些 bigram。这里选择使用 2048 个 int 来实现一个 65536 位的 Bitmap。设置或检查 Bitmap 的某一位都可以用位运算完成，速度非常理想。

```
// Hash 函数，将一个 bigram 的两个字符映射到 0~65535 的空间
#define IDX(x, y) (((unsigned int)(toupper(x)) << 8) + (unsigned int)(toupper(y)))
// 设置 Bitmap 的第 idx 位为 1
#define MAP_SET(map, idx) map[(idx) >> 5] |= (1U << ((idx) & 0x1F))
// 检查 Bitmap 的第 idx 位是否为 1
#define MAP_CHK(map, idx) !!(map[(idx) >> 5] & (1U << ((idx) & 0x1F)))

unsigned int MAP_A[2049] = {0};
unsigned int MAP_B[2049] = {0};
```

有了这两个 Bitmap 之后，我们可以先对 A 进行统计，再对 B 进行统计，当一个 bigram 第一次出现在 B 中而它又同时在 A 中出现时，就增加共同 bigram 的计数。

由于长度为 N 的字符串只有 N+1 个 bigram，而且算法只涉及两次对字符串的扫描，所以算法的时间复杂的是 **O(N)** 的。又因为对 Bitmap 的操作使用了位运算，算法实际运行时的常数也很理想。

② **levenshtein_distance**

算法主要就是一个简单的 DP，转移方程如下：

$$Dist[i][j] = \begin{cases} Dist[i-1][j-1] & (A[i] = B[i]) \\ Min\{Dist[i-1][j], Dist[i][j-1], Dist[i-1][j-1]\} + 1 & (A[i] \neq B[i]) \end{cases}$$

初始条件为$Dist[i][0] = 0$、$Dist[0][j] = 0$。最后$Dist[N][M]$即为结果。

算法的时间复杂度为 **O(N^2)**。

测试

测试结果如下，具体输出见提交的结果文件：

测试序号	结果行数	所用时间
#1 levenshtein	82 行	4.60s
#2 jaccard	31 行	0.66s
#3 combined	33 行	7.27s

Part 2: Block Nested Loop Join

实现

① 首先阅读 **src/backend/executor/nodeNestloop.c**，可以知道对 Nested Loop Join 的处理主要是在 **ExecNestLoop()** 中进行。具体来说，每次通过 ExecProcNode(outerplan)
获得一个 outer tuple，再进行一次 inner scan。而我们要做的就是将这个过程变成每次从一个已经创建好的 Block 中获取 outer tuple，在读完一个 Block 时再接着创建一个新的。

② 实现一个函数 **GetOuterBlock()**，在其中创建 BLOCK_SIZE 个 TupleTableSlot，每个 TupleTableSlot 对应原先的一个 outer tuple。函数返回起始的 TupleTableSlot 指针。这里要用到 **src/backend/executor/execTuples.c** 中定义的 **MakeTupleTableSlot()** 和 **ExecCopySlot()**。

```
TupleTableSlot *GetOuterBlock(PlanState *plan) {
    TupleTableSlot **block = palloc(sizeof(TupleTableSlot *) * (BLOCK_SIZE + 1));
    TupleTableSlot *tuple;
    int i;

    // 标识 Block 的末尾
    block[BLOCK_SIZE] = NULL;

    for (i = 0; i < BLOCK_SIZE; ++ i) {
        tuple = ExecProcNode(plan);
        if (TupIsNull(tuple)) {
            block[i] = NULL;
            if (!i) {
                // 正好占满上个 Block
                return NULL;
            }
            else {
                return block;
            }
        }

        // 将 Tuple 复制一份
        block[i] = MakeSingleTupleTableSlot(tuple->tts_tupleDescriptor);
        ExecCopySlot(block[i], tuple);
    }

    return block;
```

```
}
```

③ 有了 **GetOuterBlock()**之后就可以一次性获取一整个 Block 的数据了。现在还需要在 **src/include/nodes/execnodes.h** 中的 ExprContext 这个 struct 里添加两个指针 ecxt_outer_block_ptr 和 ecxt_outer_block_head，分别用于记录当前 Block 的当前 Tuple 地址和当前 Block 的起始地址（释放 Block 用）。

有了这些信息以后，在 **ExecNestLoop()** 的主循环中，我们就可以通过 ***(++ econtext->ecxt_outer_block_ptr)** 获取下个 tuple。当这个指针为空时，说明当前 Block 已取完或是所有 outer tuple 已取完。第一种情况下我们要释放上个 Block 占用的内存，同时调用 **GetOuterBlock()**获取新的 Block。这里要用到 execTuples.c 中定义的 **ExecDropSingleTupleTableSlot()**；第二种情况下就可以结束 JOIN。

```
    // 从当前 Block 中获取下个 Tuple
    if (econtext->ecxt_outer_block_ptr) {
        outerTupleSlot = *(++ econtext->ecxt_outer_block_ptr);
    }

    // 当前 Block 已取完
    if (TupIsNull(outerTupleSlot)) {
        // 释放上个 Block 占用的空间
        if (econtext->ecxt_outer_block_head) {
            TupleTableSlot **ptr = (TupleTableSlot
**)econtext->ecxt_outer_block_head;
            void *tmp = ptr;
            for (; !TupIsNull(*ptr); ++ ptr) {
                ExecDropSingleTupleTableSlot(*ptr);
            }
            pfree(tmp);
            // 上个 Block 的实际 Tuple 数量不足 BLOCK_SIZE 个，说明 Outer Tuple 已取完
            if (econtext->ecxt_outer_block_ptr - (TupleTableSlot
**)econtext->ecxt_outer_block_head != BLOCK_SIZE) {
                return NULL;
            }
        }

        // 获取新 Block
        econtext->ecxt_outer_block_ptr = GetOuterBlock(outerPlan);
        // 没有下个 Block 了，说明 Outer Tuple 已取完，结束 JOIN
        if (!econtext->ecxt_outer_block_ptr) {
            return NULL;
        }
        // 记录新 Block 的起始地址
        econtext->ecxt_outer_block_head = econtext->ecxt_outer_block_ptr;
```

```
        outerTupleSlot = *econtext->ecxt_outer_block_ptr;
    }
```

④ 最后，在 **src/include/executor/nodeNestloop.h** 里，要定义常量 BLOCK_SIZE 的大小。

测试

首先要说明，简单地使用以下的 SQL 语句是不能触发 Nested Loop Join 的：

```
SELECT COUNT(*)
FROM restaurantaddress ra, restaurantphone rp
WHERE ra.name = rp.name;
```

这样的语句显然会被 Optimizer 优化掉从而绕过 Nested Loop Join。通过查看日志也证实了这个判断。

为了避免这种情况，一个方法是给 WHERE 添加一个 Optimizer 认为不能优化的条件，比如这样：

```
SELECT COUNT (*)
FROM restaurantaddress ra, restaurantphone rp
WHERE ra.name = rp.name OR RANDOM() > 1;
```

这样就可以触发 Nested Loop Join，也不影响最终结果，对性能也没有太大影响。

用这条语句进行测试，会发现不论是第一次还是第二次运行，耗时都在 7~8s 左右，甚至将 BLOCK_SIZE 的值进行改变，重新编译安装后，还是同样的情形。一个合理的猜测是，因为测试数据量太小，可以完整存进内存，所以 Block Nested Loop Join 的耗时与普通的 Nested Loop Join 几乎一致，BLOCK_SIZE 也基本上对耗时没有影响。本来，Block Nested Loop Join 就是在那种数据过大，无法完整存进内存时才有意义，因为在那种情形下磁盘 IO 会成为显著的瓶颈。而在这次试验中就不能太好的表现出这一点。

此外，在实现了 Block Nested Loop Join 后，还可以用 Part1 中的 SQL 语句测试实现的正确性。在执行这些测试时，发现与不使用 Block Nested Loop Join 时结果一致，耗时也相差不大，这也印证了上述猜测。

关于 Shared Block Read 的统计，PostgreSQL 官方推荐使用的是 pg_stat_statements 这个 extension。根据文档，安装步骤如下：

1. 在 contrib/pg_stat_statements 目录下执行 make，将得到的 pg_stat_statements.so 复制到 pgsql/lib/ 目录下
2. 在 pgsql/data/postgresql.conf 中添加：
   ```
   shared_preload_libraries = 'pg_stat_statements'
   ```
3. 连接数据库，执行：
   ```
   CREATE EXTENSION pg_stat_statements;
   ```

   得到统计数据后可以发现，出于同样的原因，第二次运行同一条查询时的 Shared Block Read 数均为 0。