数据库专题训练 · Lab4

计01　容逸朗　2020010869

实验目的

1. 进一步了解 SQL 执行流程。

基础实验内容

1. 实现 AndConditionNode 和 OrConditionNode 的 visit 函数

- 首先利用 `lhs_->accept()` 遍历结点的左右子；
- 然后检查左右两子的表名是否相同：
  - 若相同，则左子必在 `table_filter_` 中；
  - 若不同，则右子必在 `table_filter_` 中；
  - 取出对应后代结点，此处简称为**目标节点** `cond`；
- 判断**目标节点**是否为 `AndCondition`，若不是则**目标节点**变为 `AndCondition({cond})`（即把原来的节点加进一个 `AndCondition` 中）
- 注意到结点的右子必为平凡节点（因为 parser 会将连续的 AND 解释为左深树），此时只需要判断右子是否为空且不曾在**目标节点**中出现，若非空空则把结点加入 `AndCondition` 中；
- 最后以右子表名为键值把**目标节点**放入 `table_filter_` 中；
- 两种结点的实现方式类似，对应 OR 的情况只要把上面对应的 AND 改为 OR 即可。

  注：考虑到后续测例中没有过于复杂的条件运算，采用上述方法是合理的。

2. 实现 JoinConditionNode 的 visit 函数

- 同样地，先利用 `lhs_->accept()` 遍历结点的左右子；
- 然后利用 `SystemManager::GetTable` 取得左右表格，并利用 `GetColumnIdx` 取得对应列编号；
- 得到列编号后，可以构造一个 `JoinCondition`，然后把其放入 `{左表名}.{右表名}` 对应的 `table_filter_` 中。

3. 在 Select 的 visit 函数中添加连接算子

- 需要遍历 `table_filter_` 的每一个 `join_condition`；
- 首先检查表名是否含有 delimiter，若无则跳过；
- 在现有测例下，对应结点只可能为 `AndCondition` 或 `JoinCondition`；
- 那么可以利用 `GetConditions` 函数把 `AndCondition` 的所有子节点取出，对于 `JoinCondition` 则不需额外操作；
- 然后对于每一个子节点，判断其是否为 `JoinCondition`，若是则处理之；
- 首先利用 `table_shift_` 取出左右两表的偏移量，并通过 `LeftShift` 和 `RightShift` 更改

`JoinCondition` 的值；
- 然后利用并查集 `uf_set` 查找所有父结点为左（右）表的表格，并记为 set_L (set_R)；
- 计算 set_L 所有表的列数总和 bias，然后对 set_R 的所有表格加上 bias 的偏移；
- 最后，利用并查集把右表并入左表中，然后以左表为键值把 `JoinCondition` 加入 `table_map` 中。

4. 实现 Next 函数并进行连接运算

- 利用 `GetLeft` 和 `GetRight` 取得左、右结点 `node`；
- 然后不断执行 `node->Next()` 取得完整的表格；
- 得到完整的两个 RecordList 后，通过 `Fit` 函数确认每对 Record 是否满足合并条件，并利用嵌套循环连接的方法合并表格即可。

高级功能 1: 实现多种 join 算法

1. 设计方案

- 基础功能已经实现了嵌套循环连接（Nested Loop Join）；
- 除此之外，我还实现了排序归并连接（Sort Merge Join）和哈希连接（Hash Join）；
- **排序归并连接**：首先利用 `std::sort` 对两个数组排序，然后利用归并的方式得到结果；
- **哈希连接**：首先判断储存格的类型，然后构造对应的 `std::unordered_map` 模拟哈希函数即可。

2. 测例生成

- 测试的思路如下，首先建立两张表，然后分别向两张表插入 10000 条数据，然后做一次简单的合并选取，计算用时。我测试了三个合并结果数据量在不同量级的测例，从中可以看到三者的性能差别。
- 具体代码请参考 `test_generator/lab4a.py`，此代码可以按要求生成不同的测例。

3. 测试结果

表格大小相同 ($N = M = 10^4$)

- 对于合并数据结果较小（$P = 10^4$，约 10000 条）的情况，嵌套循环连接的方式最慢，而排序归并连接和哈希连接都有比较好的效果，最快的和最慢的方法有 35 倍的性能差异：

```
dbtrain> dbtrain>    dbtrain> dbtrain>   dbtrain> dbtrain>
Use Nested Loop Join Use Sort Merge Join Use Hash Join
Execute Time: 6153ms Execute Time: 213ms Execute Time: 171ms
```

- 对于合并结果数据量中等（$P = 10^3$，约 10^5 条）的情况，嵌套循环连接的方式最慢，而排序归并连接和哈希连接都有比较好的效果，最快的和最慢的方法约有 8 倍的性能差异：

```
dbtrain> dbtrain>    dbtrain> dbtrain>   dbtrain> dbtrain>
Use Nested Loop Join Use Sort Merge Join Use Hash Join
Execute Time: 6714ms Execute Time: 918ms Execute Time: 901ms
```

- 对于合并结果数据量大（$P = 10^2$，约 10^6 条）的情况，尽管排序归并连接和哈希连接都比嵌套循环连接好，但是性能差异已经低于 2 倍：

```
dbtrain> dbtrain>     dbtrain> dbtrain>     dbtrain> dbtrain>
Use Nested Loop Join  Use Sort Merge Join   Use Hash Join
Execute Time: 26957ms Execute Time: 11376ms Execute Time: 15877ms
```

表格大小相差数量级较大 $(N = 10^5, M = 10^3)$

- 对于合并结果数据量中等（$P = 10^3$，约 10^5 条）的情况，结果是类似的：

```
dbtrain> dbtrain>     dbtrain> dbtrain>     dbtrain> dbtrain>
Use Nested Loop Join  Use Sort Merge Join   Use Hash Join
Execute Time: 6947ms  Execute Time: 1284ms  Execute Time: 1149ms
```

高级功能 2: 实现聚合算子

1. 设计方案

- 首先需要在分析树中加入新的类型 `AggCol` 代替原有的 `Col` ，其参数除了原来 `Col` 所拥有的表名、列名外，还增加了一个用于表示聚合算子类型的值 `AggType`（非聚合算子此项记为 `BASIC`）；
- 同时，我修改了 Select 语句，加入了 groupby_course，其中包含了所有分组字段，在预处理阶段时需要为对应列作标记；
- 除此之外，最主要的改动在 `SelectNode` 中：
  - 若选取的列不包含聚合算子，则使用原来的方法，直接返回 Record List 即可。
  - 否则，我们需要对 RecordList 的数据进行预处理，先遍历一次数据项，然后把分组字段的内容合并为一个键值放入 `std::map` 中作为索引列；
  - 与此同时，遍历每列并利用 `map` 维护各键值对应列的 SUM, COUNT, MAX 和 MIN 结果；
  - 最后，遍历 `map` 中内容，按照各列的 `AggType` 信息恢复内容或计算聚合值；

2. 测试结果

- 为方便展示，我设计了一个固定格式的测例，测例有两个表：
- `sc_sheet` 表：

id	cid	course	score1	score2	score3
1	1	AAA	81	90	35
2	1	AAA	82	55	100
3	1	AAA	53	36	32
1	1	BBB	89	32	99
3	1	BBB	19	32	86
6	2	BBB	77	23	9
2	1	CCC	92	84	34
3	1	CCC	77	43	98
4	2	CCC	93	87	45
5	2	CCC	23	76	34
1	1	DDD	23	79	66
5	2	DDD	23	99	51
6	2	DDD	45	98	33
2	1	EEE	99	54	46
4	2	EEE	35	57	24
5	2	EEE	44	57	88

- `ta_list` 表：

course	ta	salary
AAA	A	1
AAA	B	2
AAA	C	3
BBB	A	3
BBB	B	2
BBB	C	5
CCC	A	3
CCC	C	2
DDD	B	4
EEE	C	5

- 进行如下操作：

```
select id, COUNT(*) from sc_sheet where score1 < 60 group by id;

select MAX(score1), SUM(score2), AVG(score3) from sc_sheet where
course = 'CCC';

select COUNT(*), SUM(score1), AVG(score3), id, cid from sc_sheet group
by id, cid;

select cid, MIN(score2), AVG(score1) from sc_sheet group by cid;

select cid, course, MAX(score1), MIN(score2), AVG(score3), COUNT(*)
from sc_sheet group by cid, course;

select COUNT(*), SUM(ta_list.salary), sc_sheet.id, sc_sheet.cid,
ta_list.ta from sc_sheet, ta_list where sc_sheet.course =
ta_list.course and sc_sheet.score1 > 80 group by sc_sheet.id,
sc_sheet.cid, ta_list.ta;
```

- 运行后得到：

```
-- select id, COUNT(*) from sc_sheet where score1 < 60 group by id;
id | COUNT(*)
1 | 1
3 | 2
4 | 1
5 | 3
6 | 1

-- select MAX(score1), SUM(score2), AVG(score3) from sc_sheet where
course = 'CCC';
MAX(score1) | SUM(score2) | AVG(score3)
93 | 290 | 52.75

-- select COUNT(*), SUM(score1), AVG(score3), id, cid from sc_sheet
group by id, cid;
COUNT(*) | SUM(score1) | AVG(score3) | id | cid
3 | 193 | 66.6667 | 1 | 1
3 | 273 | 60 | 2 | 1
3 | 149 | 72 | 3 | 1
2 | 128 | 34.5 | 4 | 2
3 | 90 | 57.6667 | 5 | 2
2 | 122 | 21 | 6 | 2

-- select cid, MIN(score2), AVG(score1) from sc_sheet group by cid;
```

```
cid | MIN(score2) | AVG(score1)
1 | 32 | 68.3333
2 | 23 | 48.5714

-- select cid, course, MAX(score1), MIN(score2), AVG(score3), COUNT(*) from sc_sheet group by cid, course;
cid | course | MAX(score1) | MIN(score2) | AVG(score3) | COUNT(*)
1 | AAA | 82 | 36 | 55.6667 | 3
1 | BBB | 89 | 32 | 92.5 | 2
1 | CCC | 92 | 43 | 66 | 2
1 | DDD | 23 | 79 | 66 | 1
1 | EEE | 99 | 54 | 46 | 1
2 | BBB | 77 | 23 | 9 | 1
2 | CCC | 93 | 76 | 39.5 | 2
2 | DDD | 45 | 98 | 42 | 2
2 | EEE | 44 | 57 | 56 | 2

-- select COUNT(*), SUM(ta_list.salary), sc_sheet.id, sc_sheet.cid, ta_list.ta from sc_sheet, ta_list where sc_sheet.course = ta_list.course and sc_sheet.score1 > 80 group by sc_sheet.id, sc_sheet.cid, ta_list.ta;
COUNT(*) | SUM(salary) | id | cid | ta
2 | 4 | 1 | 1 | A
2 | 4 | 1 | 1 | B
2 | 8 | 1 | 1 | C
2 | 4 | 2 | 1 | A
1 | 2 | 2 | 1 | B
3 | 10 | 2 | 1 | C
1 | 3 | 4 | 2 | A
1 | 2 | 4 | 2 | C
```

- 不难验算，上面结果是正确的。
- 除此之外，我还设计了测例生成器 `test_generator/lab4b.py`，用于检查实现的正确性。
- 经多次测试，均可得到正确的结果。

总结

- 高级功能 Commit ID：`510467e02cc46b332c01831f4ff5691807c2329f`（位于 `ch4b` 分支）
- 上述的分支包含了已实现的所有功能，即：
  - 基础功能
  - 高级功能：实现多种 Join 算法
  - 高级功能：实现聚合算子

- 用时：
  - 基础功能用时 5 小时；
  - 实现多种 Join 算法用时 3 小时；
  - 实现聚合算子用时 6 小时；
- 合计 14 小时。