PJ2 report

17300180070 马逸君

系统与源码理解

PostgreSQL 中的**所有运算操作都是由系统目录**(也称数据字典)**来定义的。**在我看来，这是它与传统关系型数据库最大的不同——在后者中，系统目录是用来存储数据库信息、表信息、列信息的，但在 PostgreSQL 中，它存储的信息远远不止这些，还存储了数据类型、函数、访问路径等。

PostgreSQL 的所有运算操作都基于系统目录，而系统目录是允许用户修改的，这就给 PostgreSQL 带来了**无与伦比的可扩展性。**在传统关系型数据库中，用户对其进行扩展的仅有方式是：修改数据库系统的源码，或者加载由 DBMS 供应商特别编写的模块。而在 PostgreSQL 中，我们可以动态扩展，用户只需指定目标编码文件（对 C 语言函数来说，是共享库(.so)文件；对 SQL 函数来说，更是可以通过 SQL 文件或者直接在命令行中完成加载），然后 PostgreSQL 就可以在运行中动态加载它。这种能够"on-the-fly"地修改数据库运算操作的能力使得 PostgreSQL 能够非常方便地支持新应用和新存储结构。

值得一提的是，用户在 PostgreSQL 中**扩展函数时需要遵循严格的格式**，需要在扩展函数语句中指定函数名，指定参数个数及各自类型、是输入还是输出参数/名称/默认值，指定返回值类型，指定源码的语言类型，指定输入为 NULL 时的处理方式，其结果的可变性（immutable/stable/volatile），是否有副作用（leakproof），是否支持并行……当然其中大部分内容都是可选的，但给予了用户自行声明这些属性的权利，进一步体现了 PostgreSQL 的高自由度。

初步设计及实现

本次我们需要实现两个用户函数 levenshtein_distance(varchar, varchar)和 jaccard_index(varchar, varchar)，用来计算给定的两个字符串的 levenshtein 距离和 jaccard 系数。因为对 C 语言最为熟悉，我选择了用 C 语言来实现。

首先我们需要声明一个符合"Version 1"调用规则的函数框架，即 C 源码中的返回值类型和参数类型都用系统给定的宏来代替，而不直接写出；利用系统定义的函数来读参数指针和返回结果；用取得的参数指针调用系统定义的另外一些函数，来读取参数内部的文本内容和取得参数的长度。

这之后需要实现函数的主体部分。

对于 levenshtein 距离，朴素算法是递推。用 f(i, j)表示 第一个字符串 a 的前 i 个字符构成的子串 和 第二个字符串 b 的前 j 个字符构成的子串 的 levenshtein 距离，递推公式如下：

$$f(i,j)=\begin{cases} max(i,j), \text{如果} i=0 \text{或} j=0 \\ min\begin{cases} f(i-1,j)+1 \\ f(i,j-1)+1 \\ f(i-1,j-1)+1_{(a_i\neq b_j)} \end{cases} \text{,其它情况} \end{cases}$$

其中 $1_{(ai\neq bj)}$的意思是，若 $a_i \neq b_j$，则取值 1，否则取 0.

按照这个思路容易写出函数的主体部分（一个二重循环）。

对于 Jaccard 系数，因为我们并不关心每个 bigram 的具体内容，只需要判断它们是否在两个字符串中都出现，所以我采用了哈希的思想：用一个 128*128 的二维 bool 数组作为哈希表，bigram 两个字符的 ASCII 码值分别作为第一维第二维下标。这个哈希是一个完美哈希，不必考虑冲突。

所以我们只需要把两个字符串都从头到尾遍历一遍，一边遍历一边就可以取得所需的 bigram。声明两个变量，分别统计两个字符串的 bigram 的交的个数 ins 和并的个数 uni。访问第一个字符串的时候，把出现的每个 bigram 在哈希表中都标为 true，然后给 uni 加 1；访问第二个字符串的时候，若这个 bigram 在第一个字符串中出现过，即在哈希表中已经是 true，就把 ins 加 1，否则给 uni 加 1。最后的返回值是(double)ins/uni。

关键代码说明

见源代码的注释。

性能优化及实验结果

算法角度：设 M、N 分别表示两个字符串的长度。

对 levenshtein_distance()函数，朴素算法时间复杂度为 O(MN)，存在优化空间。

我主要研究了双向搜索这样一种优化算法。以两个字符串 “听说马上就要放假了” 和 “你听说要放假了” 为例，说明优化思路：

f	听	说	马	上	就	要	放	假	了
我	**1**	2	3	4	5	6	7	8	9
听	**1**	2	3	4	5	6	7	8	9
说	2	**1**	**2**	**3**	**4**	5	6	7	8
要	3	2	2	3	4	**4**	5	6	7
放	4	3	3	3	4	5	**4**	5	6
假	5	4	4	4	4	5	5	**4**	6
了	6	5	5	5	5	5	6	5	**4**

我们发现，在递推过程中，绝大部分的中间结果是无用的，图中只有加粗的几个值是最终的答案路径。它的特点是：从左上向右下走，每次选择的方向都是三个方向（向下、向右、向右下）里相对最优的一个，但如果有相等的结果，则所有相等的方向都要计算，直到在推

进面上出现一个相对更优的方案，则舍弃相对较差的所有方案。这样一来，递推过程实际有用的就是图上灰格子里的信息，可以看到我们少计算了很多冗余信息。为了缩小推进面的大小，可以从首尾两个方向同时开搜。

这样一来就得到一种比较浅显的优化方式：双向搜索。用一个二维数组记录答案，二维数组的含义与朴素算法相同；从字符串的首端和尾端同时开始匹配，过程中永远只保留推进面上的最小值（但所有相等的值都要保留下来），当两个方向的搜索在横坐标或纵坐标上相遇时即停止，统计答案。

理论上时间复杂度为 O(m(d+1))，m 是较短的串长度，d 是最终的编辑距离。对于高相似度的串，理论上提升明显。但在编写过程中，我发现这种算法存在一个很严重的问题：在开始计算之前必须把 f 都初始化为 INF，这样一来实际复杂度仍然是 O(mn)，实际运行中甚至发现仅初始化的时间就已经和朴素算法耗时差不多；再考虑搜索要创建节点、要用队列等一系列增大常数的因素，很可能还没有朴素算法快。

考虑到我们的字符串长度都不超过 200，可以预测：一方面单次字符串匹配的时间并不是我们的查询过程的瓶颈，另一方面这么短的字符串也优化不了多少。但双向搜索的代码复杂度相比朴素算法却是大大提升，遂放弃继续实现这一算法。

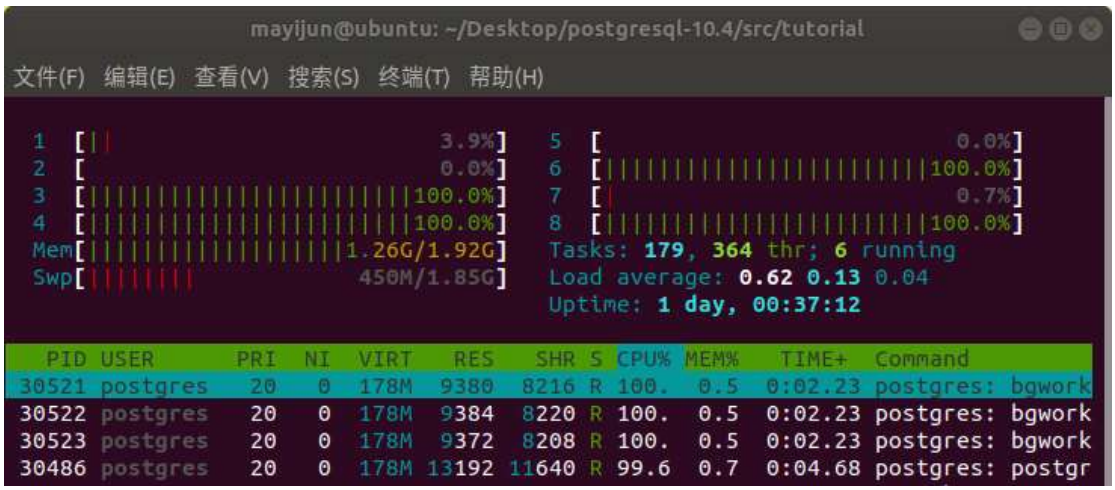
除了这种只验证对角线的算法之外，还可以通过数学方法[1]或只求近似解[2]来进行优化，但这些算法就远远超出我的知识水平了。

对 jaccard_index()函数，时间复杂度为 O(M+N)，已经是理论复杂度的下界了，所以不需要进行算法的优化。

数据库角度：指并行。

首先考虑查询实现方式上的优化。因为数据上索引的情况是可以按需更改的，所以没有固定的最优解；而且 PostgreSQL 的查询优化器已经帮我们选好了最佳的查询实现方式，所以我们不必手动指定具体使用哪一种方式完成查询。

这样一来，在数据库角度可以进行的优化就是并行查询了。具体方法是，在测试用的 sql 文件开头，把 min_parallel_table_scan_size[3]这个系统变量设置为 1，并在 CREATE FUNCTION 语句中把函数声明为 parallel safe。这样一来，在执行查询的时候，查询优化



器就会自动启用并行，我们可以通过 Linux 下 htop 这个工具一睹多核并行的快感。当然，在文件结尾记得把这个系统变量恢复默认值。

值得一提的是，在我的电脑上一开始只能并行使用 8 个核心中的 3 个。查阅数据文件夹下的 postgresql.conf 这个文件，我们发现 max_parallel_workers_per_gather 这个系统变量的默认值为 2。把它改成 8 以后，就可以并行使用 4 个核心了，速度进一步提升。把相关变量全部改成 16 仍然只能用 4 个核心，至于为什么不能使用全部的 8 个核心，这就是查询优化器的事情了。

我们对并行的优化效果进行了测试：

```
explain analyze select count(*) from restaurantphone rp, addressphone ap where levenshtein_distance(rp.phone, ap.phone) < 4;

explain analyze select count(*) from restaurantaddress ra, restaurantphone rp where levenshtein_distance(ra.name,rp.name) < 3;

explain analyze select count(*) from restaurantaddress ra, addressphone ap where levenshtein_distance(ra.address, ap.address) < 4;

explain analyze select count(*) from restaurantphone rp, addressphone ap where jaccard_index(rp.phone, ap.phone) > 0.6;

explain analyze select count(*) from restaurantaddress ra, restaurantphone rp where jaccard_index(ra.name, rp.name) > 0.65;

explain analyze select count(*) from restaurantaddress ra, addressphone ap where jaccard_index(ra.address, ap.address) > 0.8;
```

结果如下。其中“时间”一栏的结果为运行三次的平均值，单位为 ms；本机 CPU 为 Intel Core i5-8250U，内存大小 8GB。

朴素算法：

	1	2	3	4	5	6
返回值	3252	2130	2592	1639	3453	2542
时间	2536.683	4673.884	9301.783	2169.9	2703.659	3372.455

并行：

	1	2	3	4	5	6
返回值	3252	2130	2592	1639	3453	2542
时间	922.208	1621.547	3149.083	810.006	906.1	1109.61

我们发现，4 核并行并不能使执行时间缩短到原来的 1/4。这是因为并行本身是会增加开销的，启用并行查询收集数据并将“收集”的数据进行聚合会带来额外的开销。每增加一个并行，开销也随之增大。因而有时更多的并行并不能改善查询性能。[4]事实上，系统中设定表大小小于 min_parallel_table_scan_size 时一般不启用并行，就是出于这种考虑：表较小时并行查询的性能优化有限。（但我们一方面为了执行速度冲高，一方面为了体现并行的威力，强行修改此变量开启了并行。）

开发过程花絮

首先我要严厉谴责一个非常不负责任的 CSDN 博主：sustccs2。我在查找快速求解 levenshtein 距离的方法的时候，找到了这样一篇文章《计算字符串相似度 Levenshtein 的优化》[5]。这篇文章提供了一个似是而非的 Java 源代码，并在结尾处说“匆忙写完之后仅用了几个数据检验，不排除有没排查到的错误，再次抱拳。”我一开始是信任这篇文章的。

但在我把给的 Java 源码翻译成 C 语言并运行之后，发现运行前面的那个测试查询会报错。调试发现计算过程中可能会越界。我加上了越界处理，这时可以正常运行了，但返回的结果是偏小的。又经过很长时间的调试我发现文中有逻辑错误：对于三个方向中有两个或三个出现相等值的情况，它的处理极其草率：优先选择向下走。这样是得不到最优解的，甚至可以轻而易举地被一个"aaa"和"aaaa"的测试数据卡掉。难怪查询返回值偏小。

发现这个算法有逻辑漏洞后，我花了一整天时间调试的快速算法泡汤了。看来我国的搜索引擎和学术博主的水平有待提升啊……（若要把这个算法修补正确，就必须像我刚才说的那样双向搜索了，但因为对长度在 200 以下的字符串效果实在难说，加上代码量巨大，我就没有接着实现这种算法了。）

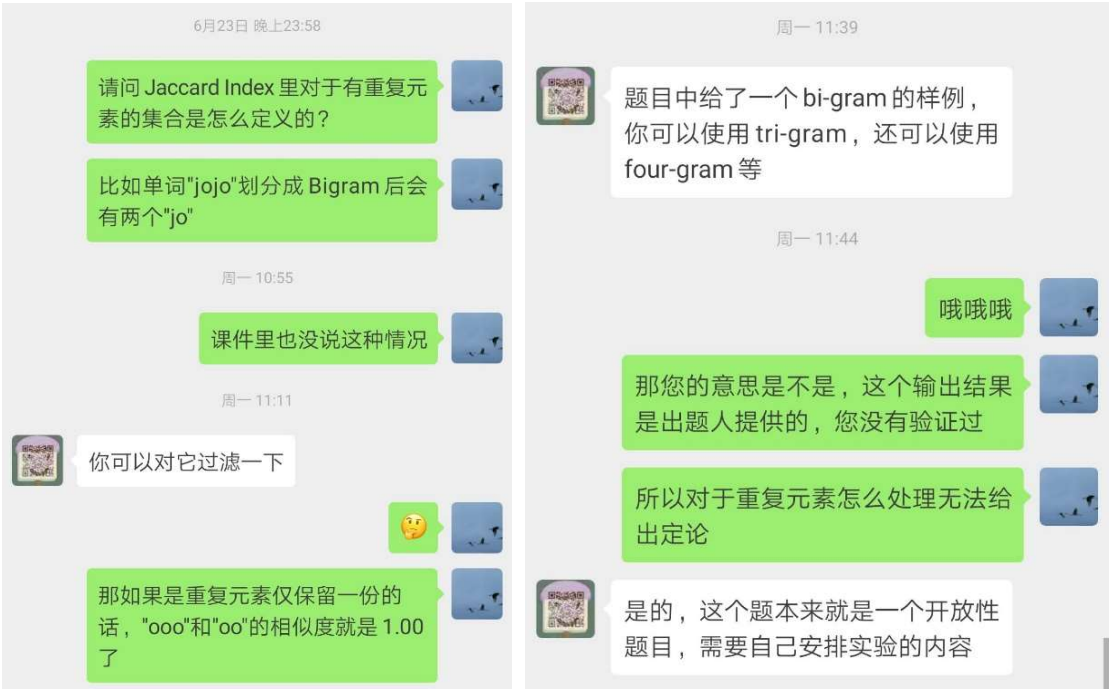
算法优化除了降低复杂度，还有一条可行的途径是卡常数。我进行过这种尝试。我写了个 C++程序，用 getline 读入了所有的数据，统计一共有多少种字符，希望减少哈希表的大小。结果显示：

```
!#&'()+,-./0123456789:;ABCDEFGHIJKLMNOPQRSTUVWXYZ[abcdefghijklmnopqrstuvwxyz|~
```

一共有 81 种字符。把哈希表大小从 128*128 优化到 81*81 的效果并不明显，还会增加一层哈希函数的时间开销，遂放弃。

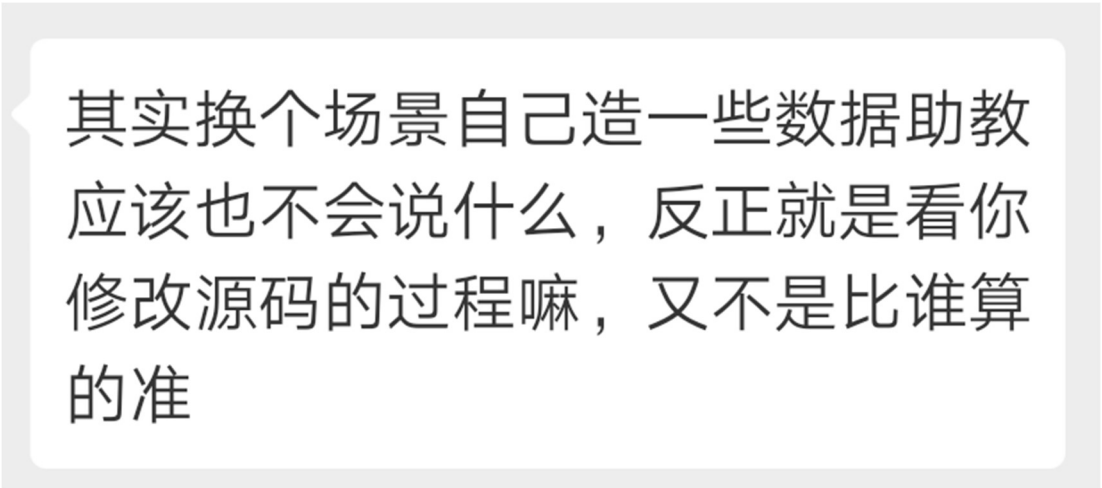
之后是关于 PPT 里给出的标准答案的问题。它给出的 Levenshtein 第三笔测资的标准答案是 2592，但我一开始算出的答案总是 2582，和一些同学对答案以后发现他们也是一样的，直到后来李泳桦同学告诉我 2592 是不区分大小写时的结果，我才明白问题，在此向他致谢。

PPT 给出的 Jaccard 第一笔测资标准答案为 1653，我的答案则是 1639。Jaccard 算法其实有很多待讨论的地方，比如我询问助教的问题：把字符串视为 Bigram 的集合，重复元素应该怎么处理？是严格去重，还是视为可重集？助教也承认这是一个没有严格定论的问题。



为了继续探寻 1653 这个答案是怎么得到的，我首先尝试用可重集来实现，答案为 1488，差距较大，说明标算应该是去重而非可重集。之后我试着把填充在串首串尾的字符由'$'改成空格，答案变成 1660，仍有差距。和同学交流，他们的答案也是各异，一个同学说是 1647。我还尝试把返回值类型改成 float，或者把分子分母直接作为整数返回并把 jaccard_index<0.6 这样的查询条件改成用乘法判定结果（完全避免精度问题），仍然不对。最后我利用 pg_similarity 这个标准库[6]里的 Jaccard 算法验证了答案，连标准库也有误差，输出结果为 1675。

有个同学跟我说：



此言得之。之后我就不再纠缠答案误差的问题了。

此外，开发的时候还遇到了这些插曲：在自己用户的.profile 里设置了 PATH，启动服务器的时候报不知名的错误，后来发现是必须在所有用户下都设置好 PATH 才能启动；加载一个函数后，如果修改了这个函数，必须用\q 关闭连接然后重连，才能正常地重新加载这个函数（否则仍然是修改前的效果；这个问题在开发之初坑了我不少时间，如果不是我发现

怎么改函数它的返回值都没变，可能还发现不了这个问题；不知这个问题是不是 PostgreSQL 的一个 BUG)；不知道怎么解压下载下来的压缩包（gunzip postgresql-10.8.tar.gz；tar xf postgresql-10.8.tar)，直接用 Ubuntu 自带的归档文件管理器解压了，结果根本找不到安装要用的文件夹；这是我第一次编译以 MB 为单位的源码，编译了好长时间……

总之，本次 PJ 是我第一次接触自由度如此之高的、开源的数据库系统，增长了不少见识，也为日后数据库方面的进一步学习做了铺垫。

参考文献

[1] http://www.berghel.net/publications/asm/asm.php
Hal Berghel, David Roach. An Extension of Ukkonen's Enhanced Dynamic Programming ASM Algorithm. 1996.1.8.
[2] https://onak.pl/papers/focs_2010-approximating_edit_distance.html
Alexandr Andoni, Robert Krauthgamer, Krzysztof Onak.
Polylogarithmic Approximation for Edit Distance and the Asymmetric Query Complexity. 2010.
[3] https://www.cnblogs.com/kuang17/p/8311071.html
狂神 314　PostgreSQL 系统参数调整及并行设置（转） 2018.1.18
[4] https://www.cnblogs.com/baisha/p/8309852.html
emplay PostgreSQL9.6 的新特性并行查询 2018.1.18
[5] https://blog.csdn.net/sustccs2/article/details/51909535
sustccs2 计算字符串相似度 Levenshtein 的优化 2016.7.14
[6] https://github.com/eulerto/pg_similarity
eulerto. pg_similarity. 2019.5.22