**CS3223 Assignment 2 – Hash Join**

## a) Team members:

Le Vu Nguyen Chuong – A0088552W

Huynh Ngoc Tai – A0074310W

b) Platform used: Solaris

## c) Edited files:

The following list contains the files that we have edited. Every new introduction of code is prefixed with a comment cs3223.

hashjoin.h, globals.c, guc.c, nodeHashjoin.c, explain.c, nodeHash.c

## d) Hash Functions:

We support all primitive data types (int, double, varchar, etc) in our hash functions implementation. In hashjoin.h, we introduced several new attributes to struct HashJoinTableData, which are:

An int pointer (bitvector) to serve as an array of integer, each of which stores 32 bits of the bitvector. This is to ensure that we takes care of one bit of the bitvector using only one bit in the memory. The number of elements in a bitvector can be dynamically allocated, depends on bitvector\_size. The number of bits that needs to be allocated will therefore be:

#bits = bitvector\_size \* 1024 \* 8;

The maximum #bits is limited by 2^32, which implies the maximum user input bitvector\_size is 524288.

We instead of applying our hash functions directly to a Datum representing a tuple value, we decided to apply the hash functions to the hash value returned from Postgres hash function, i.e we hash each tuple twice – once using Postgres hash function, once using one of our hash functions. This is a sound fix to a problem of supporting various data type (int, double, varchar, etc). Note that especially for the case of supporting varchar, which has taken us a lot of time to handle, is the main reason why we decided to delegate the first hashing duty to Postgres. Eventually, we have found that Postgres hash function can deal with varchar safely and correctly. Delegating the work of capturing different data types to Postgres, thought to be time inefficient, turns out to be reasonably good in terms of performance: we manage to improve the overall join performance and achieve generally good bitvector effectiveness.

Hash Method 1: We use a simple modulo operation (by bitvector\_size) as our hash function.

Hash Method 2: FNV32 (Noll, n.d.). The key note of FNV32 is to process an octet\_of\_data at a time.

## e) Experiment results:

### Part 1 of Assignment: Benchmarking Postgres Hash Join

|  |  |  |  |
| --- | --- | --- | --- |
| X | Query Q1  Time (ms) | Query Q2  Time (ms) | Query Q3  Time (ms) |
| 12,500 | 5540.335 | 6143.764 | 6564.537 |
| 25,000 | 6085.037 | 5858.864 | 6328.897 |
| 37,500 | 6335.665 | 6447.458 | 6605.340 |
| 50,000 | 6669.804 | 7080.800 | 6566.842 |
| 75,000 | 7906.391 | 8966.027 | 7272.314 |

Table A

|  |  |  |  |
| --- | --- | --- | --- |
| work mem (kB) | Query Q1  Time (ms) | Query Q2  Time (ms) | Query Q3  Time (ms) |
| 192 | 7059.599 | 7547.430 | 7000.713 |
| 320 | 6922.775 | 6420.682 | 6337.042 |
| 576 | 6757.650 | 6595.618 | 6240.067 |
| 1088 | 6524.940 | 5633.162 | 5542.180 |
| 2112 | 4854.253 | 3538.010 | 3965.769 |
| 4160 | 4397.563 | 3770.369 | 3559.010 |

Table B

### Part 3 of Assignment: Benchmarking the new Implementation

#### Experiment C1:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **X** | Hash Method 1 | | Hash Method 2 | |
| Effectiveness (%) | Time (ms) | Effectiveness (%) | Time (ms) |
| 12500 | 99.70% | 4217.546 | 99.69% | 3699.763 |
| 25000 | 99.43% | 4203.306 | 99.41% | 4484.544 |
| 37500 | 99.17% | 5302.949 | 99.09% | 5664.047 |
| 50000 | 98.85% | 6330.350 | 98.81% | 6261.706 |
| 75000 | 98.20% | 7811.196 | 98.37% | 8115.489 |

#### Experiment C2:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **X** | Hash Method 1 | | Hash Method 2 | |
| Effectiveness (%) | Time (ms) | Effectiveness (%) | Time (ms) |
| 12500 | 99.70% | 3598.461 | 99.69% | 3956.416 |
| 25000 | 99.43% | 4178.219 | 99.41% | 4427.690 |
| 37500 | 99.17% | 5452.872 | 99.09% | 5530.924 |
| 50000 | 98.85% | 6019.564 | 98.81% | 6224.585 |
| 75000 | 98.20% | 7627.230 | 98.37% | 8073.968 |

#### Experiment C3:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **X** | Hash Method 1 | | Hash Method 2 | |
| Effectiveness (%) | Time (ms) | Effectiveness (%) | Time (ms) |
| 12500 | 99.70% | 7884.609 | 99.69% | 7602.415 |
| 25000 | 99.43% | 6953.326 | 99.41% | 6924.912 |
| 37500 | 99.17% | 6914.813 | 99.09% | 7740.874 |
| 50000 | 98.85% | 7904.096 | 98.81% | 7708.404 |
| 75000 | 98.20% | 7697.256 | 98.37% | 8605.057 |

#### Experiment D1:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N (kB)** | Hash Method 1 | | Hash Method 2 | |
| Effectiveness (%) | Time (ms) | Effectiveness (%) | Time (ms) |
| 128 | 95.40% | 6482.918 | 95.21% | 6482.357 |
| 256 | 97.67% | 5699.613 | 97.63% | 6512.913 |
| 512 | 98.85% | 5869.672 | 98.81% | 6184.667 |
| 1024 | 99.39% | 6320.000 | 99.37% | 6314.265 |
| 2048 | 99.70% | 6376.002 | 99.71% | 6239.951 |
| 4096 | 99.86% | 6152.919 | 99.85% | 6672.367 |

#### Experiment D2:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N (kB)** | Hash Method 1 | | Hash Method 2 | |
| Effectiveness (%) | Time (ms) | Effectiveness (%) | Time (ms) |
| 128 | 95.40% | 6593.090 | 95.21% | 6707.854 |
| 256 | 97.67% | 5640.286 | 97.63% | 6237.798 |
| 512 | 98.85% | 6499.758 | 98.81% | 6638.321 |
| 1024 | 99.39% | 6257.268 | 99.37% | 6073.526 |
| 2048 | 99.70% | 5964.972 | 99.71% | 6573.886 |
| 4096 | 99.86% | 6612.984 | 99.85% | 6182.455 |

#### Experiment D3:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N (kB)** | Hash Method 1 | | Hash Method 2 | |
| Effectiveness (%) | Time (ms) | Effectiveness (%) | Time (ms) |
| 128 | 95.40% | 8249.151 | 95.21% | 8053.033 |
| 256 | 97.67% | 7573.266 | 97.63% | 8163.586 |
| 512 | 98.85% | 7249.650 | 98.81% | 7817.489 |
| 1024 | 99.39% | 7573.315 | 99.37% | 8276.949 |
| 2048 | 99.70% | 7315.877 | 99.71% | 8346.545 |
| 4096 | 99.86% | 8712.772 | 99.85% | 8477.323 |

#### Experiment E1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **JF** | Hash Method 1 | | Hash Method 2 | |
| Effectiveness (%) | Time (ms) | Effectiveness (%) | Time (ms) |
| 5 | 98.85% | 3669.873 | 98.81% | 3757.190 |
| 10 | 98.85% | 6601.442 | 98.81% | 6588.432 |
| 20 | 98.85% | 13444.463 | 98.81% | 13095.267 |
| 40 | 98.85% | 24272.769 | 98.81% | 23290.451 |
| 80 | 98.85% | 50260.357 | 98.81% | 48750.646 |

#### Experiment F1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **||R||** | Hash Method 1 | | Hash Method 2 | |
| Effectiveness (%) | Time (ms) | Effectiveness (%) | Time (ms) |
| 25,000 | 99.70% | 3582.832 | 99.69% | 3651.069 |
| 50,000 | 99.43% | 4290.718 | 99.41% | 4451.695 |
| 75,000 | 99.17% | 5587.530 | 99.09% | 5920.960 |
| 100,000 | 98.85% | 5928.545 | 98.81% | 5919.045 |
| 200,000 | 97.66% | 9374.361 | 97.67% | 9992.620 |

## f) Brief Evaluation

Through the usage of Bloom Filter, the performance can be improved (compare Table A with Tables C1, C2, C3). The performance of Bloom Filter depends on the hash methods used and on the size of the bitvector. In experiment C and F, we note a trend that when X gets larger, the bitvector effectiveness gets decreasing. This can be explained as the number of tuples in the inner relation gets larger when X is larger and therefore affects the bitvector effectiveness. In experiment D, on the other note, bitvector effectiveness is generally higher when bitvector\_size is larger, and large enough to cater different tuple’s values inside the table. Lastly, from experiment E, the bitvector effectiveness does not depend on the number of tuples (it depends rather on the different values of the tuples); but since the number of tuples increases, the time to process join definitely increases. In conclusion, by accompanying Bloom filer, we can project more than just the tuples that meet the predicate condition, thus implying we can achieve join result with less cost by performing join.

# References

Noll, L. C. (n.d.). *FNV Hash*. Retrieved March 30, 2014, from http://www.isthe.com/chongo/tech/comp/fnv/