Software Development for Information Systems (2019 – 2020)

Antonis Klironomos

Chrysostomos Rampidis

Fotis Ektoras Tavoularis

# Introduction

The following study was executed as part of the course on software development for information systems

in the Notional and Kapodistrian University of Athens. The aim of this study was to create a subset of a database

that manages data entirely in the main memory. The whole task was split in three parts. In the first part

we implemented a Sort Merge Join Algorithm, in the second one we performed a query analysis, similar to the one used

in the SIGMOD 2018 competition, while in the last part we implemented CPU personalization and a query optimizer.

The project is hosted in GitHub (https://github.com/hectortav/Project2020) and contains a Makefile for compilation

and a bash script (testProgram.sh) to easily test the program. To run the executable, produced by the Makefile,

called "final", you can use one or more of the following flags to provide cli arguments:

\* -qr (QueRy)

Run in queries of every batch in parallel

\* -ro (ReOrder)

Run bucket reorder (radix-sort) in parallel

\* -pb (Reorder -> New job Per Bucket) ("-ro" should be provided) Create a new parallel job for each new bucket

\* -qs (QuickSort)

Run quicksorts independently

\* -jn (JoiN)

Run joins in parallel (split arrays)

\* -ft (FilTer)

Runs filters in parallel

\* -pj (ProJection)

Runs Projection checksums in parallel

\* -all (ALL)

Everything runs in parallel

\* -n <threads>

Specify number of threads to run

# Observations and choices during implementation

We implemented and tested two different methods for sorting buckets during the first and second part. One of the was using recursion while the other one was using a loop to break the buckets into smaller ones when needed. After thorough testing we decided two use the recursive one since we managed to achieve better execution times and convolution

while we also managed to keep each thread work on a different memory part at any time, reducing the need for semaphores which would slow the execution further since each thread would have to wait to access the memory.

Another part of the execution that we tried to parallelize was the copy from one tuple to another during each call of the Reorder function (tuplereorder\_parallel @ functions.cpp), but we finally decided not to keep the change since it significantly slowed the program because all the extra threads that would be used to execute the copy could be used more

efficiently by another function that is waiting in the scheduler.

To adapt Join for running on a parallel processing system, we tried several different implementations. First, we tried the one described in class, that is to join areas that have the same prefix. Then, we tried merging some of these areas together so that each new area cluster is assigned to a thread. Finally, we tried to split the relation in equal parts so that each part is assigned to a thread. We noticed that the third method that kept the job count at a minimum while at the same time accomplished a balanced load between threads, produced a better result concerning time and processor usage.

# Time & Memory Statistics

The following table contains the program runtime measurements, CPU and memory usage with various parameters:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 2 threads | 4 threads | 8 threads | 16 threads | 32 threads | 64 threads |
| Serial |  |  |  |  |  |  |
| Run in queries of every batch in parallel |  |  |  |  |  |  |
| Run bucket reorder (radix-sort) in parallel |  |  |  |  |  |  |
| Create a new parallel job for each new bucket |  |  |  |  |  |  |
| Run quicksorts independently |  |  |  |  |  |  |
| Run joins in parallel (split arrays) |  |  |  |  |  |  |
| Runs filters in parallel |  |  |  |  |  |  |
| Runs Projection checksums in parallel |  |  |  |  |  |  |
| Everything runs in parallel |  |  |  |  |  |  |

Unit Testing

To test the execution of our program we used unit testing and especially Cunit. We tried to make unitests for every function regardless of the size or simplicity. Some of the functions tested are the following:

* randomIndex
* swap
* hashFunction
* makeparts
* splitpreds
* optimizepredicates
* predsplittoterms
* sortBucket
* histcreateTest
* psumcreateTest
* tuplesReorderTest
* InputArray::filterRowIds
* InputArray::extractColumnFromRowIds
* IntermediateArray::populate
* IntermediateArray::findColumnIndexByInputArrayId
* IntermediateArray::findColumnIndexByPredicateArrayId
* IntermediateArray::selfJoin