Software Development for Information Systems (2019 – 2020)

Antonis Klironomos

Chrysostomos Rampidis

Fotis Ektoras Tavoularis

Contents

1. 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**: Each query is scheduled as a single Job and runs in parallel with the other Jobs.
- **-ro**: If "**-pb**" argument is not provided **and** if current shift of hash function is even, the reorder task of the "overflowed" bucket is scheduled as a single Job and runs in parallel with the other Jobs.
- **-pb**: If "**-ro**" argument is provided, the reorder task of each "overflowed" bucket is scheduled as a single Job and runs in parallel with the other Jobs.
- **-qs**: Each quicksort task is scheduled as a single Job and runs in parallel with the other Jobs.
- **-jn**: If "-jnthreads" argument is not provided, each job is responsible for joining areas that have the same prefix.
- **-jnthreads**: if "**-jn**" argument is provided, the array involved in **each** join is split evenly according to the thread pool's size and the join of each of its parts with the other array is scheduled as a single Job and runs in parallel with the other Jobs.
- **-pj**: The result array is split evenly according to the thread pool's size and the calculation of the sums of each of its parts is scheduled as a single Job and runs in parallel with the other Jobs.
- **-ft**: Each time a filter is handled, the current array of row ids is split evenly according to the thread pool's size and the filtering of each of its parts is scheduled as a single Job and runs in parallel with the other Jobs.
- **-all**: Equivalent to the following combination of arguments: **-qr -ro -qs -jn -pj -ft**
- **-n <# of threads>**: Size of the thread pool to be created by the scheduler.
- **-optimize**: Join enumeration is used to reorder join predicates. **Note**: If this argument is **not** provided, join predicates are not reordered but filter predicates are relocated before the join predicates.
- **-h**: Displays a help message which contains a short description of each argument.

2. Code Description

2.1. *Description of job scheduler and jobs (class JobScheduler abstract class Job)*

- JobScheduler consists of a queue (JobQueue) in which any kind of Job (: abstract class) is inserted.
- When the JobScheduler gets initialized, a thread pool is created, the size of which is defined by the user.
- When a new Job (with a new unique id) is inserted into the queue, an available thread or the first thread that will be available handles the Job.
- Job insertion and obtainment is handled with mutual exclusion in order to avoid race conditions.
- If the Job is a Radix-sort Job (for parallel Radix-sort) or a Quicksort Job (for parallel Quicksort), jobsCounter[queryIndex] is **incremented by 1** when the Job is inserted into the queue. This is helping the parent thread in knowing when to proceed in joining the results of the 2 relations' reordering. (queryIndex is the index/id of the current query and it is used for the case of parallel run of queries)
- When **any** Job is completed, jobsCounter[queryIndex] is **decremented by 1**. If jobsCounter[queryIndex] equals to **0**, a condition variable is signaled which indicates that all the jobs of the current part of the program are done, so the parent thread can continue its execution.
- When the JobScheduler gets destroyed, it creates ExitJobs (with **id=-1**), the count of which is equal to the thread pool's size. When a thread handles an ExitJob, it exits immediately.

2.2. *Description of intermediate result (class IntermediateArray)*

- IntermediateArray consists of the row ids of each array (of the predicate) that took part in previous joins. Each column is identified by the corresponding predicate array's id.
- After the first join, an IntermediateArray is created having as contents the 2 columns that resulted from the join.
- After each consecutive join, which involves a column (with row ids) of the IntermediateArray and a column (with row ids) of a first-time-used predicate array, a new IntermediateArray is created. This newly created IntermediateArray has **row count** equal to the last join result's row count and **column count** equal to the previous IntermediateArray's column count **plus 1**. Its contents are the joined row ids taken from the previous IntermediateArray and the joined row ids of the firstly used predicate array.
- A join between 2 arrays that have already been involved in a previous join, is handled as inner-join/self-join of the current IntermediateArray.

2.3. *Description of handlepredicates(), the main function of the program*

The flow of handlepredicates() is the following:

(1) Filter predicates are relocated before the join predicates. If "**-optimize**" argument is provided, join predicates are reordered according to an algorithm of **join enumeration**.
(2) For each predicate array id (which corresponds to an original input array (InputArray object)), a new InputArray object is created which contains **only the row ids** of the original input array. This is done because it prevents the existence of duplicated data.
(3) For each predicate:
   (a) If the predicate is **filter** or **self-join/inner-join** of an input array, its row ids (InputArray object) are filtered and transferred to a newly created InputArray which replaces the old row ids.
   (b) If the predicate contains 2 predicate array ids both of which exist in the current IntermediateArray (both participated in a previous join), this case is handled as a **self-join/inner-join** of the current IntermediateArray. So, the current IntermediateArray is filtered and transferred to a newly created IntermediateArray which becomes the current one.
   (c) In any other case, the predicate is handled as a typical join between 2 arrays. Specifically:
      i. For each of the 2 pairs of predicate array id and field id, an object of type relation is created by retrieving data from the corresponding original input array. The **payloads** are retrieved from the rows the ids of which exist **(1)** in the corresponding (filtered) row-ids-array (InputArray object) **or (2)** in the current IntermediateArray if the predicate array id participated in a previous join. In case **(1)**, the **keys** of the relation are the row ids of the original input array. In case **(2)**, the **keys** of the relation are the row ids of the current IntermediateArray.
      ii. For each predicate array id and field id, if the combination of them participated in the last join, the above corresponding extracted relation does **not** get reordered, because it already is. In the opposite case, the relation gets reordered with **radix-sort**.
      iii. The 2 above ordered relations are joined in a list which is then converted to a 2-column array each of its columns contains the row ids of either the original input array or the current IntermediateArray as mentioned in step (i).
      iv. If the result contains 0 entries, then the function does the necessary memory deallocation and returns NULL which indicates no results.
      v. In any other case, a new IntermediateArray is created which consists of only the above 2-column result if this is the first join, **or** of the contents

of the previous IntermediateArray **plus** the column with the row ids of the first-time-joined InputArray. This new IntermediateArray becomes the current one.

(d) If the last IntermediateArray is not NULL and its size is greater than 0, the function returns it. In any other case, NULL is returned which indicates no results.

2.4. *Map data structure used for the optimization algorithm*

For the implementation of the optimization algorithm, a map data structure was used. This map uses a "Key" as a key and returns a "Value". A map is a data structure that uses some information as identification for some other information. In our case, it uses a "PredicateArray" to produce another "PredicateArray" with some stats, all described in the "Value" class.

The Key acts like a normal key in any other map, being unique and is the point of reference to returning a Value. A value acts like the content of a specific key. A value also contains a 2d array with statistics used by the algorithm.

The functions insert and retrieve were implemented to make use of the map. Also another helping function,exists, was used to check if a specific key already exists in the map. Constructors and destructors were also implemented for correct memory management and efficient usage of the map.

This map is exclusively used for calculating the best order of the predicates given, meaning that it finds the combination of predicates with the least total cost, by using formulas and specific metrics provided by the instructors of this course.

3. Implementation Observations

We implemented and tested two different methods for sorting buckets during the first and second part. One of them 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 to 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.

We noticed that we had great time improvements when we used smaller and continues memory allocations. For example by turning histogram and psum from 2D arrays to 1D we managed to cut the execution time to 60% of what we had with the model described in class. The reasoning behind this great improvement is that a big part of the array can be kept in cache and we don't have to worry about loading times that take a long time.

Another observation was the difference between the small and medium inputs. We noticed that the multi threading was better exploited with the medium sized input, where we saw greater time and CPU usage improvements. The explanation is that the time that it takes to split the jobs and wait for an available thread is better justified when the inputs are greater and the time relation of the thread organization processes to the time of the rest of the processes tend to zero.

While running the program through a profiler, we noticed that partitioning during predicate optimization, consumed 37% of the overall time. That means there are many data in a small number of buckets. That way we do not have the optimal outcome we could accomplish with an other optimization policy.

Another observation we made is that the list functions (e.g. insert) consume 17% of the time. If we had another data structure that would not require constant checks and processes to allocate new memory blocks we would achieve better times with probably an allocation of unneeded memory as a negative aspect.

4. Time Statistics

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

4.0.1. *Small Sized Input*

	1 thread (serial)	2 threads	4 threads	8 threads	16 threads	32 threads	64 threads
No predicate optimization	0.775	N/A	N/A	N/A	N/A	N/A	N/A
Predicate optimization	0.720	N/A	N/A	N/A	N/A	N/A	N/A
(No predicate optimization) Parallel queries	N/A	0.856	0.662	0.645	0.671	0.671	0.727
(No preddicate optimization) Run the following in parallel (Radix-sort, Quicksort, Join, Filters, Projection)	N/A	0.841	0.837	0.856	0.890	0.914	0.906
(No predicate optimization) Everything runs in parallel (Queries, Radix-sort, Quicksort, Join, Filters, Projection)	N/A	0.889	0.806	0.744	0.719	0.737	0.785
(Predicate optimization) Parallel queries	N/A	0.780	0.645	0.673	0.670	0.631	0.649
(Preddicate optimization) Run the following in parallel (Radix-sort, Quicksort, Join, Filters, Projection)	N/A	0.788	0.772	0.802	0.833	0.983	0.861
(Predicate optimization) Everything runs in parallel (Queries, Radix-sort, Quicksort, Join, Filters, Projection)	N/A	0.825	0.738	0.707	0.652	0.690	0.701

Table 1.

Table 2. Time in seconds

4.0.2. *Medium Sized Input*

	1 thread (serial)	2 threads	4 threads	8 threads	16 threads	32 threads	64 threads
No predicate optimization	47.211	N/A	N/A	N/A	N/A	N/A	N/A
Predicate optimization	48.311	N/A	N/A	N/A	N/A	N/A	N/A
(No predicate optimization) Parallel queries (No preddicate optimization)	N/A	44.903	24.862	20.573	20.471	20.844	21.023
Run the following in parallel (Radix-sort, Quicksort, Join, Filters, Projection) (No predicate optimization)	N/A	41.879	38.320	37.520	38.142	36.670	38.874
Everything runs in parallel (Queries, Radix-sort, Quicksort, Join, Filters, Projection) (Predicate optimization)	N/A	47.957	35.500	29.447	20.235	20.767	22.356
Parallel queries (Preddicate optimization)	N/A	47.957	25.203	20.961	20.235	20.767	22.356
Run the following in parallel (Radix-sort, Quicksort, Join, Filters, Projection) (Predicate optimization)	N/A	46.076	23.769	20.072	20.854	21.316	21.008
Everything runs in parallel (Queries, Radix-sort, Quicksort, Join, Filters, Projection)	N/A	38.684	32.924	30.747	30.871	31.293	36.182

Table 3.

Table 4. Time in seconds

5. Time Results Conclusions

- Without the "-optimize" flag the program runs without query optimization **but with the filters of each query being placed first**. This is a reason why sometimes the program runs quicker without the "-optimize" flag.
- When the small dataset is given as input, the size of the data is so small that the overhead of Join Enumeration adds a **small** delay to the program compared to its execution without the "-optimize" flag.
- When the **medium** dataset is given as input, the size of the data is bigger, so Join Enumeration increases the speed of the program.
- Parallelism increases the speed of the program significantly because the execution load is split so that each thread is handling a different Job at each given moment.
- When the program's thread pool has a size which is larger than the maximum number of threads that the system can provide, there is only a small or no amount of increase at the program's speed compared to using a smaller thread pool.

6. Parallelization Observations

At the third and last part of the project, parallelization was added to the program. The effect this transition had on the program is immediately apparent when looking at the overall execution times. Adding CPU parallelization led to a decline of 60% concerning execution time and a increase of 70% concerning CPU usage which means computer resources are better utilized and the overall outcome is maximized.

7. Version Control Observations

During the whole project we used git as a version control system. With git we managed to track bugs that emerged during development and then easily solve them. Git also helped our team to share code and communicate better that led to better organization and management. Finally, a git aspect we found greatly useful was branching since each team member could work in separate features without causing problems and interference.

8. Unit Testing

To test the execution of our program we used unit testing and especially Cunit. We tried to make unit tests 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

Overall we concluded that unit testing in an easy way to test and validate the solidity of a big project, that would otherwise require many hours of manual testing. Also we found that creating unit tests before the project is a valid way to ensure the smooth development of a program as well as keeping the team in track and necessitate the usage of a common API.

9. Conclusion

During this assignment we used various technologies and researched a plethora of ideas and theories. We appreciated concepts like parallel programming, unit testing, version control and of course software development for information systems. Parallel programming is a useful asset today since all systems have multiple real and virtual CPU cores that are not often utilized correctly. Unit testing is a respectful way to ensure that programs & algorithms are valid and ready for production and version control is an easy way to track bugs and work with teams. Information systems and databases are the core of today's computing and leaving in an age where data are abundant, we must explore new and more efficient ways to store, find and index them. Last but not least, we learned the importance of pre-planning, estimating possible obstacles and chances during development of such last projects.