Pimp my Taxi Ride

DEBS Grand Challenge 2015

Aimylos Galeos
Computer Engineering and Informatics Department
University of Patras, Greece
galeos@ceid.upatras.gr

Prokopis Gryllos
Computer Engineering and Informatics Department
University of Patras, Greece
gryllos@ceid.upatras.gr

Nikolaos Leventis
Computer Engineering and Informatics Department
University of Patras, Greece
leventis@ceid.upatras.gr

Konstantinos Mavrikis
Computer Engineering and Informatics Department
University of Patras, Greece
mavriki@ceid.upatras.gr

Spyros Voulgaris
Dept. of Computer Science
VU University
The Netherlands
spyros@cs.vu.nl

ABSTRACT

The DEBS Grand Challenge is an annual event in which participants are asked to provide the fastest possible solutions to real-world problems in the area of event-based systems. This paper reports in detail the architecture and algorithms employed in our custom implementation for the 2015 Grand Challenge. We start by laying out the overall architecture of our design, we continue by detailing the inner working and structures for carrying out the challenge, and we conclude with a discussion on lessons learned.

Keywords

Event-based Systems, Data Processing, High Performance Computing, Data Structures

1. INTRODUCTION

The 9th ACM International Conference on Distributed Event-Based Systems (DEBS 2015) defined a Grand Challenge [2] for designing and building the fastest-possible implementation of a stream-processing application.

The application concerns massive data collected under the *FOILing NYC's Taxi Trip Data* project [3]. This project has collected itinerary information on all rides of over 10,000 taxis in a wide area of $150km \times 150km$ around New York City over the whole year of 2013. The collected information includes the (anonymized) taxi ID, (anonymized) taxi license ID, pickup and dropoff coordinates, pickup and dropoff times, and the total cost of the ride divided in basic fare, tax, tolls, and tip.

The Grand Challenge defined a complex problem consisting of two queries. Assuming that the data of a taxi ride becomes available altogether the exact moment it drops the passengers off (i.e., entries in the file are sorted by dropoff time), we should concurrently execute the following two queries:

1. **Frequent Routes:** Considering the area split in a grid of 300×300 cells of $500m \times 500m$ each, compute the running list of the top-10 *most frequent routes* (i.e., ⟨pickup cell, dropoff cell⟩ tuples) in a sliding window of 30 minutes. The top-10 list should be output in a specific format every time it is altered.

2. **Profitable Cells:** By splitting now the area in 600×600 cells of $250m \times 250m$ each, compute the running list of the top-10 *most profitable cells*. A cell's *profitability* is defined as the ratio of the cell's current profit over the number of empty taxis currently residing in the cell. In their turn, the cell's *current profit* is defined as the *median* of fare + tip profits out of all rides that *started* in that cell and ended within the last 15 minutes (irrespectively of the dropoff cell), while the cell's number of *empty taxis* is defined as the number of taxis that had a drop-off in that cell within the last 30 minutes and have not reported any subsequent ride yet.

Note that these two queries have to be computed in parallel, sending their output in separate files.

The criterion for determining the Grand Challenge winner is based on two metrics. The first metric is the *total execution time* for computing both queries, while the second one is the *average processing time per record*. Clearly, both metrics should be optimized in parallel.

In this paper we report on our design and implementation of our solution. More specifically, Section 2 presents an overview of our high-level design decisions. Section 3 details the heavily multithreaded pipelined architecture of our solution. Subsequently, Section 4 and Section 5 detail the data structures and inner workings for computing queries 1

and 2, respectively. Section 6 explains our evaluation techniques and fine-tuning methods, while Section 7 follows with a concluding discussion.

2. SOLUTION OVERVIEW

A number of decisions had to be taken during the brainstorming and design phase of our endeavor. Our starting design decision was on whether we should pursue a solution based on an established stream-processing platform (e.g., Apache Storm [1]), or go for a highly tailored and optimized custom solution.

We opted for a custom solution, for the following reasons. First and foremost, the challenge's goal was clearly stated upfront: performance, performance, and performance. That is, code maintainability, functionality extensibility, and scalability of the application on more than one machines were *not* our goals. That is, a custom-made solution would be able to tailor the executable to optimize on the specific problem at stake. Second, we had to take into consideration that our application would be tested on finite and rather limited resources: a single quad-core machine with 8GB memory. Thus, going for a tailor-made implementation would give us more control over the use of our limited resources.

A second design decision concerned the selection of the programming language and libraries to use. The selection of C/C++ was a clear choice for us, as the lower you get on the "bare metal" the more power you can squeeze out of your CPU(s).

A third decision had to do with the selection of the platform. We opted for Linux and did all our implementation on Ubuntu 14.10 with the Gnome-Shell environment. The reason for that is a rather subjective one: we consider Linux a superb development platform.

Regarding the design of our solution, the two cornerstones of our design are a very efficient multithreaded stream processing architecture, and very highly optimized data structures for the requested computations.

As far as the multithreading part is concerned, it became clear upfront that we were dealing with a highly computationally demanding problem, that certainly required extensive multithreading. We took extra steps for that, and split our program in five pipeline stages with a total of eight threads. The details of the multithreaded solution are given in Section 3.

With respect to the data structures we used, we took extra care to keep complexity as low as possible, with most operations executing at O(1), while we carefully wrote our code to minimize on the constant factor. The details on the structures for Queries 1 and 2 are presented in Sections 4 and 5, respectively.

3. PIPELINED STREAM PROCESSING

We first give an overview and reasoning of our design, and then we dive into the gory details.

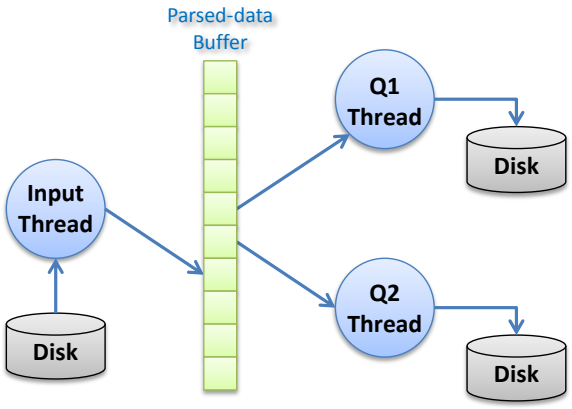

Figure 1: Basic parallelization at a conceptual level.

3.1 Design Rationale

The target application follows a classic execution model: (i) reading input from disk, (ii) performing computations, and (iii) writing output back to disk. Chopping it up in elementary pipeline stages of the right computational granularity is key to harnessing the power of modern multicore CPUs, and speeding up its execution.

At first glance, there is an obvious point of parallelization between the two queries, Query1 and Query2. Although they operate on the same input data, their respective calculations are completely orthogonal to each other, rendering them excellent candidates for being assigned to two separate threads. Handling input, that is, taking care of disk reading and parsing input data, forms a third natural point of parallelization that lets the Q1 and Q2 threads undistracted to their computations. This leads to the conceptual design laid out in Fig. 1.

In order to gain more speed, though, a finer granularity splitting of tasks is required. At a minimum, disk *reading* and *writing* operations should be assigned their own threads, to shield CPU-intensive tasks from disk I/O latency. This gives us a pipeline of four stages: disk reading, parsing of input data, Q1 and Q2 computations (in parallel), and disk writing.

After implementing this scheme and timing the respective stages, we observed that threads Q1 and Q2 were taking roughly the same time (Q2 being slightly heavier), while the parser thread was taking almost twice as much, essentially forming a bottleneck. This came as no surprise, having to process roughly 190 ASCII characters per line and to parse diverse data types, including dates, coordinates, MD-5 keys, and prices. Splitting up the parser was vital to achieving a higher speed.

Splitting up the parser, though, was not straightforward. The variable length of input lines prevented their processing in arbitrary order, as the next line's beginning could not be known before the current line's end had been traced. This imposed a sequential component to parsing the input. We circumvented this hurdle by factoring out the absolutely minimal functionality for which sequential execution could

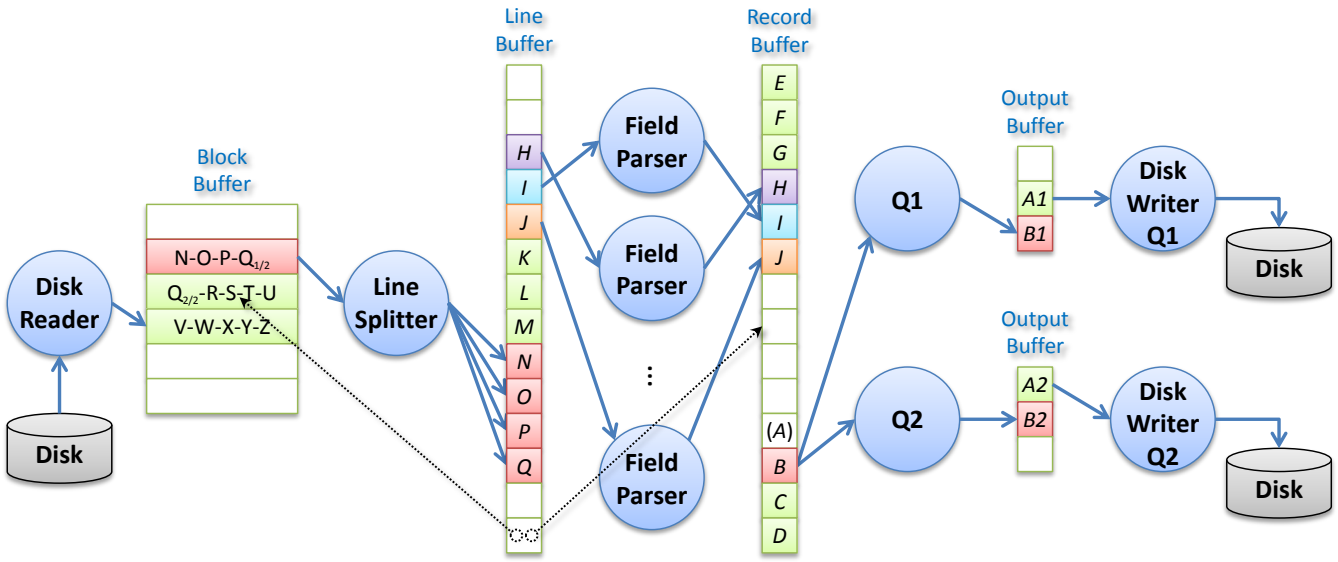

Figure 2: The pipeline architecture for stream processing.

not be prevented, therefore letting aside the bulkier part of the parser's job for parallel execution, as explained in the next section.

This led us to the final architecture of our application, depicted in Fig. 2. It consists of *five pipeline stages*, and a total of *eight threads* (we opted for two field parsers in our final deployment, as discussed later).

3.2 Multithreaded Architecture

Our final application architecture is shown in Fig. 2. It consists of five pipeline stages, glued through four levels of circular buffers.

The DiskReader thread feeds the BlockBuffer with raw data from the input file. The BlockBuffer is organized as a circular buffer of fixed-size blocks allocated in contiguous memory ranges. The DiskReader's operation is, thus, very basic.

As mentioned in the previous section, parsing has been split in two stages. One that takes care of the bare minimum sequential functionality, and a second one that can be parallelized for performance.

In particular, we split the parser in two tasks, the LineSplitter and the FieldParser. As its name suggests, the former is in charge of spotting the beginning of subsequent lines, and storing for each line two pointers in LineBuffer: a pointer to its first character (in the BlockBuffer), and a pointer to the record where that line's fields should be parsed in (in the RecordBuffer).

This allows any number of FieldParser threads to run simultaneously, each picking the next unprocessed LineBuffer entry and parsing the respective line's fields to the associated data record. Thus, two or more lines may be parsed in parallel, while their sequential order is retained in the respective data records.

The RecordBuffer contains records, each holding a line's parsed data. That is, a record is a struct containing the taxi's ID, the pick up and dropoff cell IDs and times, the trip's profit, etc.

Records are then being consumed by Query1 and Query2, which execute in parallel and independently. When either of the query threads determines that its top-10 list has been updated, it stores the corresponding information to be printed in its OutputBuffer, and lets the respective DiscWriter thread take care of disk I/O. This allows the query thread to keep computing its subsequent records.

3.3 Synchronization Pitfalls

Although our architecture appears to be a chain of standard producer-consumer threads, a number of synchronization details lie under hood, requiring fine-grained, delicate handling.

As expected, we have defined exactly *one* mutex per data structure, that is, there is a total of five mutexes in the application (each output buffer is a separate structure).

Let us start with the BlockBuffer, for example. Clearly, the DiskReader thread is the producer, and the LineSplitter thread is the consumer. However, this differs from the typical producer-consumer scenario, as the lock on the data is *not* released by the consumer. In fact, as the FieldParser is reading ASCII characters directly from the BlockBuffer's memory (to avoid redundant memory copying), the buffer's entries can only be unlocked after *all* associated lines have been processed by some of the FieldParsers. The fact that these lines can be processed in an arbitrary order does not simplify matters.

To resolve this, the LineSplitter annotates each processed

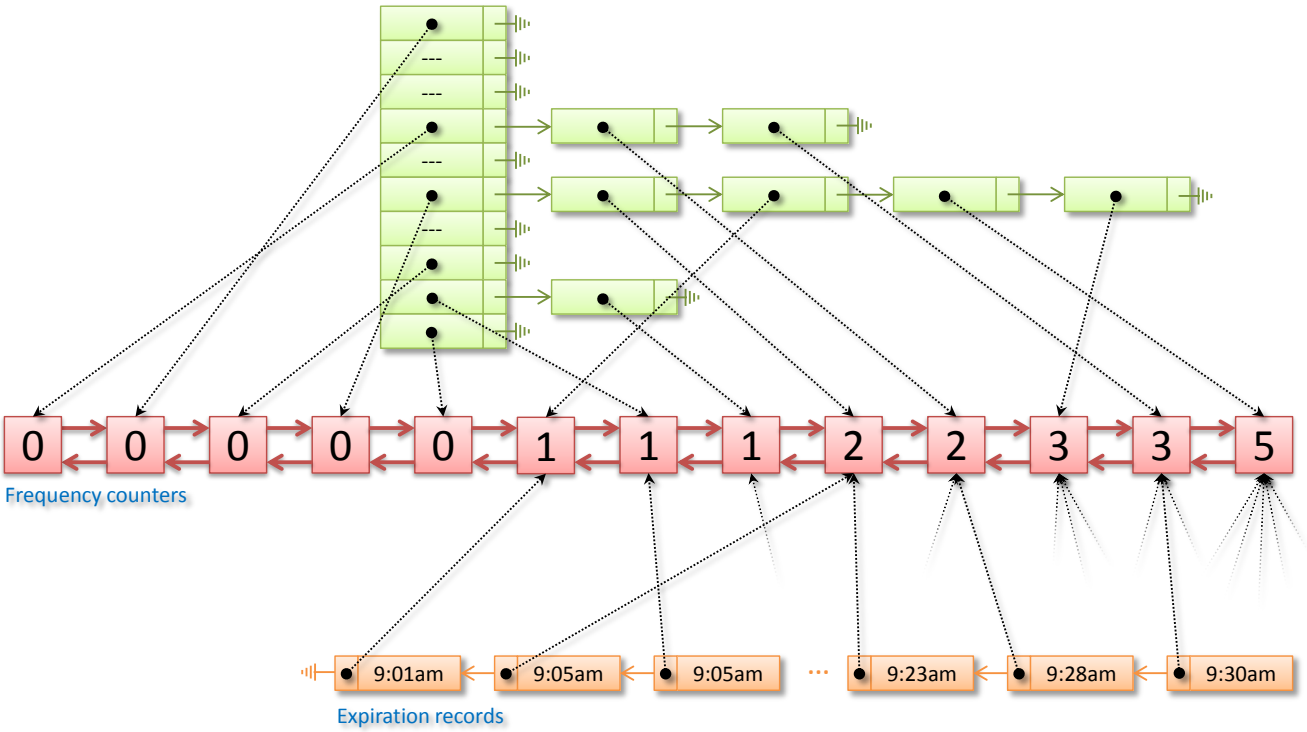

Figure 3: Data structure for Query 1.

BlockBuffer block with the *number* of lines that partially or fully lie on it, as well as with a flag telling whether it's *done* processing that block or not yet (i.e., if *yes*, the authoritative line count for that block is known). In addition, the LineSplitter also annotates each LineBuffer entry with the block (or, in some cases, two blocks) on which the line lies. Finally, when a FieldParser thread completes the parsing of a line, it acquires the respective block's mutex, and increments a counter indicating how many lines of that very block have been parsed. It additionally checks whether the LineSplitter has marked that it's done with that block, and in that case checks whether the total and parsed lines counters for that block match. If yes, it releases that block and notifies the DiskReader thread (which could have been blocked due to a full BlockBuffer) that a block has just been freed.

The LineBuffer and RecordBuffer locking mechanisms have their own peculiarities. The former has one producer (LineSplitter) but potentially many consumers (FieldParser), where each entry is to be consumed precisely *once* by *any one* of the consumers. The latter (RecordBuffer) has potentially many producers (FieldParser), and precisely two consumers (Query1, Query2), where each entry is to be consumed by *both* consumers.

Although these are not as complex as the BlockBuffer locking mechanism described above, all deviations from the standard producer-consumer model deserve very detailed thinking and extensive testing. We invested in both.

The last two OutputBuffer locking mechanisms were the only standard consumer-producer paradigms in our application.

4. QUERY 1: FREQUENT ROUTES

For Query1 we need to compute the top-10 most *frequent routes* during the last 30 minutes. As Query1 considers a grid of 300×300 cells, there is a total of $300^4 = 8.1$ billion distinct routes. Clearly, such a huge array cannot fit in 8GB of memory, but would have anyway been an enormous waste if it did.

Fortunately, we measured that the actual number of distinct routes that appear *at least once* in the full sample file is below one million, therefore it *is* possible to store them all in main memory, and simply index them through a hash table.

So, for each *seen* route we store a *frequency counter* indicating the number of times the route has been taken in the last 30 minutes. This is simple to implement. For each new route, we hash its pickup and dropoff cell IDs and feed the outcome to a hash table to get hold of the route's frequency counter. Once we have reached that counter, we just increment it by one.

Implementing the 30-minute window required an additional structure. In particular, we implemented a FIFO of *expiration records*. Each expiration record contained two fields: the trip's dropoff time, and a pointer to the respective route's frequency counter.

Fig. 3 depicts the data structures used for Query1. Han-

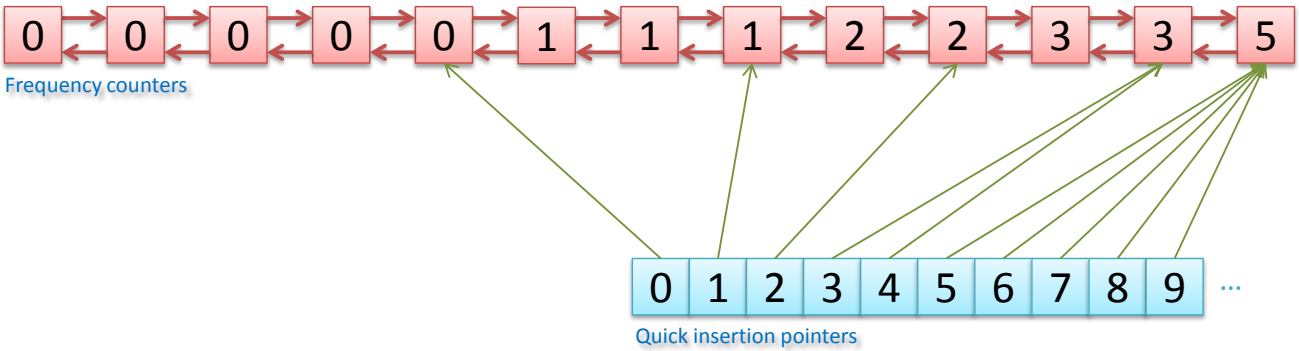

Figure 4: Quick insertion pointers for Query 1.

dling of the 30-minute window should now be clear. Upon registration of the *current trip* (i.e., the one described in the line currently being processed by QUERY1), we add a new expiration record with its dropoff time to the *head* of the aforementioned FIFO. At the same time we check if one or more expiration records at the *tail* of the FIFO have actually expired (i.e., have dropoff time more than 30 minutes older than the current trip's dropoff time), and for each expired record we use its pointer to reach and decrement the respective counter record, and then we evict it from the FIFO.

The mechanisms described so far provide us with up-to-date frequency counters of each route's repetitions in the last 30 minutes, with O(1) complexity both for incrementing a route's counter as well as for handling its expiration. Computing the top-10 list among these counters after every single update, though, is yet another challenge.

For estimating the top-10 in a swift manner, we decided to maintain all counter records *sorted at all times*, through a doubly-linked list. The main observation motivating this design decision is that frequency counters can only change by one. They are either incremented or decremented by one at a time. Therefore, a frequency counter can only move up or down by at most one position in the sorted list of counters, which is done in O(1).

One last observation led to a further optimization of the sorted list operation, illustrated in Fig. 4. There may be lots of routes having the same frequency counter value, especially for routes of low frequency (e.g., lots of routes having counters 0 or 1). When such a route's frequency counter is incremented (or decremented), we have to traverse all same-value counters towards the higher (or lower) frequency routes. To avoid a sequential traversal, we maintain an array of *quick insertion pointers* to the first record of each value in the list. I.e., position i of that array has a pointer to the first record with counter i. This way, when a route's counter changes, we can literally remove it from its current position and embed it in its new spot in the list with complexity O(1).

5. QUERY 2: PROFITABLE AREAS

In QUERY2 we need to find the running top-10 most profitable cells. More specifically, we need to compute the following (running) values for each cell:

1. **Median profit:** The *median profit* of all trips that *started* in that cell, and were completed (at any cell) within the last 15 minutes.

2. **Empty taxis:** The number of trips that *ended* in that cell within the last 30 minutes, and whose taxis have not completed a newer trip yet (in which case they would count as empty taxis for their new drop-off cell).

3. **Profitability:** This is the ratio of the two prior values, that is, a cell's profitability is the cell's median profit divided by the number of empty taxis currently residing in that cell.

Cells' profitabilities should be computed continuously, and the 10 most profitable cells should be output every time the top-10 list changes.

Note that QUERY2 is *cell-centric*, as opposed to QUERY1 being *route-centric*. Given the 600×600 cell grid considered in QUERY2, we have to deal with a maximum of 360,000 *cell records*. In order to avoid excessive memory waste (as many cells lie over the Hudson River and are, thus, never referenced), we introduce a level of indirection, defining an array of 360,000 *pointers* to records. The actual records, therefore, reside in a significantly smaller pool of 100,000 initially allocated cells (Fig. 5(a)), as well as extra dynamically allocated pools should that initial pool run out of available entries.

5.1 Median profit

As we have to compute the median profit *per cell*, we allocate a separate *cell record* for each *seen* cell, as shown in Fig. 5(b).

To compute the running median of a dynamic list of profits for a given cell, we partition them in half across two heaps: H_{low} containing the lower half of that cell's profits, and H_{high} containing the higher half. The former heap, H_{low}, is a *max heap*, i.e., its maximum element floats to its root. In contrast, the latter heap, H_{high}, is a *min heap*.

We also enforce the invariant that the two heaps' sizes cannot differ by more than one. This is easy to accomplish. After adding or removing a profit from the corresponding heap, we check whether the invariant still holds, and if not

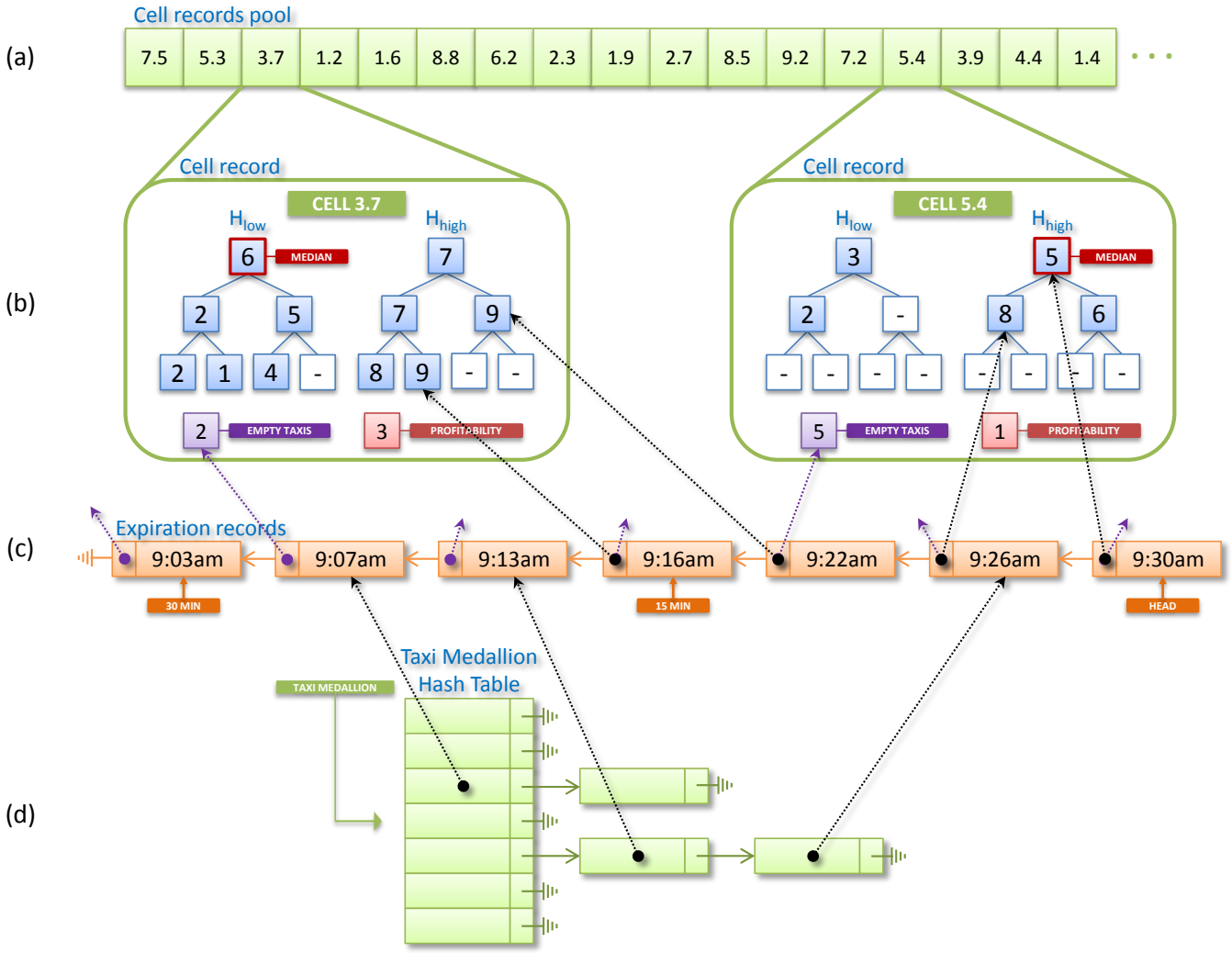

Figure 5: Query 2 data structure.

we remove the larger heap's root (i.e., H_{low}'s max or H_{high}'s min) and we insert it in the other heap, making them equi-sized again.

Now the median's computation is straightforward. If any of the heaps is larger than the other (i.e., by precisely one item), the larger heap's root is the median. Otherwise the median is defined as the two roots' mean. In the special case that both heaps are empty, the median is considered to be zero. Fig. 5(b) presents two median calculation examples.

In order to avoid performance penalties by dynamic memory allocation, we implemented these heaps as fixed-size arrays. We statically allocated a pair of heaps for each cell with a capacity of 256 entries per heap. That is, we can handle a maximum of 512 trips having started at the same pickup cell in a window of 15 minutes. For the rare case that a cell's popularity exceeds that threshold, we transfer its data to a new pair of heaps with higher capacity (and subsequently to even higher should that be exceeded too) which is dynamically allocated. Due to the rarity of this case, performance penalty is negligible.

Removing profits after the 15-minute window has elapsed, bares strong similarities to the FIFO-based handling of the sliding window in Query1. In particular, we employ a FIFO structure again, whose elements store the trip's dropoff time and a reference to the respective profit. Note that, as our heaps are implemented by arrays whose items are moved around during heapify operations, a profit's address is not fixed so it *cannot* be pointed at directly. Therefore, we also maintain a *reverse-indexing array* (not shown in the figures) mapping a profit's value to its current index in the heap arrays. Clearly, the reverse-indexing array has to be updated for each heapify operation, however it saves us from an expensive linear search through the heap for a profit when it has to be removed.

5.2 Empty taxis

In this calculation we need to compute the (running) number of empty taxis in a cell during a sliding window of 30 minutes.

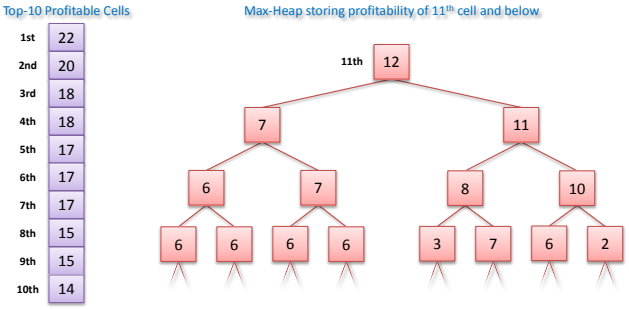

Figure 6: Top-10 calculation for Query 2.

For memory and performance gains, we share the FIFO structure used for managing the 15-minute window of profits. Simply we maintain two pointers, one pointing at the oldest record not exceeding 15 minutes, used for profit expirations, and the other pointing at the oldest record not exceeding 30 minutes, used for empty-taxis expirations. The combined FIFO structure is illustrated in Fig. 5(c).

A noteworthy difference to Query1's FIFO, is that an empty-taxis expiration record in Query2 may also have to be removed at an arbitrary time, when the same taxi has been involved in a newer trip before its window limit has been reached. To handle that, we maintain a hash table indexed by taxi medallions, and pointing at expiration records in the Query2 FIFO. This hash table is presented in Fig. 5(d).

When a new trip is being processed, a new expiration record is added to the head of the Query2 FIFO. Additionally, an entry associating the respective taxi ID with the newly created expiration record is inserted to the hash table. If, however, the hash table already contains an entry for that same taxi pointing at an active expiration record, the respective empty-taxis counter is decremented and that expiration record is marked inactive before the hash table entry is updated to point at the new expiration record. When an expiration record becomes more than 30 minutes old, the empty-taxis counter for the respective cell is decreased only if that expiration record was still active. If it was inactive, it means that the empty-taxis counter decrement has already taken place, due to that taxi's involvement in a newer trip.

Finally, note that the active-inactive flag is considered exclusively for the 30-minute expiration of empty-taxis counters, while it is not taken into consideration by the 15-minute expiration of profits.

5.3 Profitability top-10

After the median profit and/or number of empty taxis in a cell is updated, we trivially also update its profitability by a mere division. However, extracting the (running) top-10 out of a hundred thousand cells requires some extra mechanism.

In particular, we store the top-10 profitability cells *separately* from all the rest. The top-10 cells are stored in a sorted array, while all remaining cells (i.e., from the 11th onwards) in a max-heap structure. When a cell's profitability is updated either upwards or downwards, we move it accordingly either within the top-10 array or within the heap, depending on where it already resided, or we insert it anew into the heap if it had not yet been registered. If either the top-10 array's minimum element or the heap's root was changed during this action, we compare the two, and if the heap's root is higher (or equal, to cater for freshness-based priority) we swap the two and we trigger a *heapify-down* operation for heap restructuring. Upon a change of the top-10 ranking, we also trigger the generation of new output.

6. EVALUATION AND FINE-TUNING

First, let us explicitly state how we measured the requested metrics (Section 1). For the total execution time we used the well-known Linux `/usr/bin/time` tool. For determining per-record delays, for each record we mark the current time in microsecond granularity using the `clock_gettime()` POSIX function right before starting to parse its line of characters[1]. Then, we use the same function to take a second timestamp right before printing the output[2].

Evaluating our implementation was a defacto standalone procedure, due to the lack of alternative implementations to compare against. This led us to devising the following strategy. First, we strived for the leanest and fastest possible implementation of every single pipeline component, splitting the component in parallel stages if possible. Second, we applied precise timing on all components and identified the slowest one, that is, our application's bottleneck. Lastly, we fine-tuned our thread synchronization mechanism to ensure that the slowest component never stops working, i.e., it never has to wait for the next input item.

In our implementation Query2 was identified to be the slowest component, being slightly slower than Query1 (3.073144 average CPU clock ticks per record vs. 2.597875). A major implication of that is that the sizes of buffers between threads (BlockBuffer, LineBuffer, and RecordBuffer in Fig. 2) have a major effect in the overall performance, notably with respect to the second evaluation metric, that is the average processing delay per record (see Section 1).

More specifically, the use of large buffer sizes lets Query1 gradually move far ahead of Query2 in the records they process, and the same applies to the LineSplitter and FieldParser threads. That is, a large number of lines are piled up in long buffers, being processed instantly by Query1 but waiting long before being processed by Query2. This has a detrimental effect on Query2's reported delays, as they accumulate the time records have been laying in intermediate buffers. At the same time, the fast advancement of Query1 does not buy us anything, as the total time is still determined by the slowest of the two.

Our solution to that was to limit the size of buffers after extensive experimentation and benchmarking of our application's execution time and respective average delays.

A second major challenge for fast execution was to minimize the overhead caused by excessive thread switching and frequent blocking on condition variables. On that front, we

[1]file `linesplitter/LineSplitter.cc` line 93
[2]file `dataStructQ1Output.cc` lines 172, 222, 264 and file `dataStructQ2Output.cc` lines:185, 265, 311

came up with another technique, applying *thresholds* on signaling blocked threads.

More specifically, as some producer threads were faster than their respective consumers (e.g., DISKREADER was faster than LINESPLITTER), the respective buffers were practically continuously full, the consumer being blocked on a write. In normal operation, as soon as the consumer read an entry, it signaled the producer than more space is made available for producing more data. This, however, caused continuous switching between the threads. To address that, we defined *thresholds*, which prevent the consumer from signaling the producer unless their intermediate buffer's occupancy has dropped below a certain level. This significantly diminished thread switching, leading to total execution speedup of over 20%, and lower delay times by a factor of two.

Although we cannot compare our solution against any other implementation, it is indicative that our implementation executes *faster* than the well known `wc` (word count) on the same file. Specifically, for a file for which `wc` takes over 5 minutes, our complete QUERY1 and QUERY2 solution terminates with complete output in the respective files in around 4 minutes and 20 seconds.

7. DISCUSSION

Designing and implementing a solution for the DEBS Grand Challenge 2015 gave us an excellent opportunity to work on an interesting topic, and to dive into the gory details of thread synchronization and hacking-level code optimization.

First, the well known saying, that "Early optimization is the root of all evil" was confirmed over and over in our work.

Second, we learned that pipelining very often leads to bottlenecks due to excessive thread-switching. The technique we applied (thresholds) gave impressive results in the delay times without hurting the throughtput time.

Lastly, the whole procedure reminded us of the great development facilitation provided by high-level languages and frameworks, at the expense of extra performance gains. It's all about a tradeoff, and for the DEBS Grand Challenge the criterion was one: great performance!

8. REFERENCES

[1] Apache Storm (http://storm.apache.org/).

[2] DEBS Grand Challenge 2015 (http://www.debs2015.org/call-grand-challenge.html).

[3] FOILing NYC's Taxi Trip Data (http://chriswhong.com/open-data/foil_nyc_taxi/).