High Power Perl : two case studies

German Perl Workshop, Cologne
3-7 April 2018

Laurent Rosenfeld
(paris.pm)

My work activity

- I was a free-lance consultant for a major telecommunication operator in France
- Working on data quality
  - Notably comparing data from various applications of the operator's information system
  - Finding differences in order to fix data discrepancies
  - Large data volumes (hundreds of millions of records, files typically 4 to 16 GB)

A real performance problem (1)

- Comparing data of two applications, *A* and *F*
  - *A* is the operator's main business application managing many many things; *F* is a specialized helper application, with a much narrower scope
  - 35 million customer records on each database
  - Extracting flat CSV files from each database
  - Requirement to convert data from the business concepts of *A* into *F* tariff plans
  - Necessity to use on *A* a proprietary language *G* (similar to PL-SQL) to access the *A* database

A real performance problem (2)

- Conversion matrix (ca. 70,000 lines)
  - 4 pieces of data from *A* lead to a Tariff Plan in *F*:
    - 1 or 2 billing services (among 20 to 30 for one client)
    - The business segment (1 per customer)
    - The platform (also unique for one customer)
  - A matrix record may have some fields empty
  - Several records may match, a system of priorities is used to find the right tariff plan in *F* 's parlance

Example of the conversion matrix

- The matrix is a CSV file (separator : ";")

Priority	Tariff *F*	Platform *A*	Seg-ment *A*	Billing services *A*
1000	T1	Plt 1	S1	P1, P2
1001	T2	Plt 1	S1	P3
1002	T4	Plt 1	*	P1, P2
1003	T3	Plt 1	S1	*
1010	T2	Plt 1	S2	P1, P2
1011	T1	Plt 1	S2	P3
1012	T2	Plt 1	S2	*

Combinatorial explosion

- For each customer, we need to :
  - Find the business segment (unique)
  - Find the active billing services (20 to 30 per client)
  - Test each billing service and each pair of billing services, with or without a segment, in the matrix
  - Find the combination having the lowest priority number

Conclusion to the first case

- Perl is slower than a compiled language, but you can get better performance if you make good use of its strengths
  - I might never have thought of using hashes, greps, and so on if I had been using, say, C
- The team converted to Perl for many tasks
- Use a hash (or several) when it makes sense

Combinatorial explosion (2)

- Very, very slow program in the *G* language:
  - The first version would have run in 160 days
  - Improved version would have taken 60 days
  - Not acceptable: 3 days would do, perhaps 5, not 60

Using Perl to profile the G program

- I used Perl 5 to profile the G program:
  - Wrote a small Perl profiling program to:
    - read and parse the G program source code,
    - add time recording statements before and after each procedure call,
    - compile and run the modified G code, and
    - measure the time taken by each of the procedures.

Profiling results

- Results: more than 99% of the time was taken traversing the conversion matrix
  - about 59.5 days out of 60 days, and 12 hours for the rest (extraction and so on)
- No further optimizing was deemed to be possible on the conversion part in G

Rewriting the conversion in Perl

- The profiling analysis led to the decision to rewrite the conversion part in Perl
  - The data extraction is done in G (extracting all BS)
  - The extracted file is reprocessed in Perl
  - Removing the sequential reads of the Matrix, replacing them with hash lookups

Rewriting the conversion in Perl

- The new Perl program uses a series of hashes to store the matrix data:
  - 2 hashes: seg. → priority and single BS → priority
  - 2 HoH: seg. + 1 BS → priority; 2 BS → priority
  - 1 HoHoH : segment + 2 BS → priority
  - One array : priority → F tariff plan
  - One auxilliary hash: a list of all BS actually existing in the matrix

The algorithm is now a few lookups

- For each *A* customer, we :
  - Filter the useful billing services (simple grep to prune the BS not used in the matrix)
  - Lookup business segment alone
  - Lookup each remaining billing service alone
  - Lookup each remaining BS with segment
  - Lookup each pair of BS with segment,
  - Lookup each pair of BS without segment
  - Keep the combination with highest priority,
  - Which leads to the *F* tariff plan.

Results

- Overall **run time divided by 110**
  - 13 hours instead of 60 days
  - 12 hours for the data extraction in *G* and 1 hour for the Perl conversion program
  - Conversion proper went down from 59.5 days to 1 hour, i.e. it was about **1,400 times faster**.
- This meant completely changing the algorithm
  - A somewhat complex mecanism made very simple thanks to the expressivity of Perl
  - 210 code lines, 2 days for coding and unit testing

Second case : matching ranges

- A problem with a genetics program
- We have a list of millions of ranges in a hash :

```
(
   'seq1' => { start => 0,    end => 50,  },
   'seq2' => { start => 75,   end => 150, },
   'seq3' => { start => 200,  end => 300, },
   'seq4' => { start => 450,  end => 550, }, # ...
)
```

- We want to know if a given region (a range from another data source) overlaps with any range of the list

Performance problem

- The program was doing something like this :

```
foreach my $key (sort { … } keys %{ref_data}) {
   my $start = $ref_data->{$key}{start};
   my $end   = $ref_data->{$key}{end};
   if ( ( $ref->[0] >= $start and $ref->[0] <= $end )
     or ( $ref->[1] <= $end and $ref->[1] >= $start ) ) {
         return 'IN';
  }
}
return 'OUT';
```

- This took 12 seconds for checking one region
- There were hundreds of thousands regions to check – so about 330 hrs for 100,000 regions

Some problems with the code

- The program is doing a sequential search through all the keys of the hash
- The sorting of the keys is done for each region to be checked
- Sorting the keys is in fact useless (removing it speeds up the program by a factor of 4)
- There is really no point to use a hash if you're not doing lookups but a simple sequential search

CS basics: binary search (1)

- If the ranges are sorted, we can implement a binary search rather than a sequential search
- Binary search is how you look for a word in a dictionary or a name in a phone book
- Or a number-guessing game : pick a number between 1 and 64 –

```
Is it 32, less than 32, or more than 32? - Less
Is it 16, less than 16, or more than 16? - More
Is it 24, less than 24, or more than 24? - Less
Is it 20, less than 20, or more than 20? - Less
Is it 18, less than 18, or more than 18? - More
Then, it is 19!
```

CS basics: binary search (2)

- Example binary search implementation:

```
sub search_min {
my ($first, $last, $start_reg, $sorted_ref) = @_;
  return $first if $first >= $last – 20; #shortened search
  my $pivot = int (($first + $last)/2);
  if ($start_reg > $sorted_ref->[$pivot][1]) {
    return search_min ($pivot + 1, $last, $start, $sorted);
  } elsif  ($start_reg < $sorted_ref->[$pivot][1]) {
    return search_min ($first, $pivot - 1, $start, $sorted);
    } else {
      return $pivot;
    }
}
```

Sorting the reference data and implementing a binary search

- With dummy data (1 M ranges, 20 regions),
  - Sequential search (no sort) took 21 seconds
  - A binary search on sorted data took 0 second!
  - Adding a loop to do the search 5,000 times (so 100,000 searches) took 5 seconds
- So : 1 region per sec. versus 20,000 per sec.
  - Worst case is $(\log_2 1E6) + 1 \approx 21$ iterations
  - Average case: $(\log_2 1E6 + 1) - 1 \approx 19$
- Same output

Results with the real process

- The huge speed-up obtained relates only to the search and to dummy data
  - It does not count the time to enter the reference data and to sort the ranges
  - The actual program first has to populate and sort the reference data, so the improvement is diluted
- The actual overall performance enhancement is by a factor of about 35 (from 28 hours to slightly less than 50 minutes).

Conclusion

- A good data structure can give a huge gain on performance by allowing a better algorithm
- Perl gives a great flexibility for constructing such data structures and use them efficiently
- But we need to think hard about the best data structure, depending on the specific need
  - It may be a collection of hashes, or it may be something else su as an array with a binary search approach

Questions?

- Thank you for listening. Any questions?

- These slides are available on Github :
  - https://github.com/LaurentRosenfeld/
  - They can be used under the terms of the Creative Commons Attribution ShareAlike License (CC-BY-SA)