UNIVERSITY OF AGDER

Project report in Dat300

Distributed querying with Apache Solr

Improving performance by splitting complex search expressions

A bachelor thesis by

Mari Næss
Ørjan Hatteberg
Enok Karlsen Eskeland

Supervisors

Folke Haugland
Jaran Nilsen

This Bachelor's Thesis is carried out as a part of the education at the University of Agder and is therefore approved as a part of this education.

University of Agder, 2011

Faculty of Engineering and Science

Department of information- and communication technology

Abstract

Complex taxonomies delivers bad search performance for Integrasco. This report is about troubleshooting the issue and developing a solution based on a hypothesis stating that several smaller taxonomies in sum performs better than one large taxonomy. Testing indicated optimization potential in splitting. This sparked the creation of a query splitter to decrease response time in a sharded environment on Solr. This splitter proved to give improved performance when a taxonomy performs poorly with the regular search. Despite a problem relating to searches with high start offset, this can be a beneficial solution for the problem.

Contents

List of Figures

List of Tables

Definition list

Taxonomy is by Integrasco usage and in this context defined as a complex query.

Document is the basic unit in Lucene indexing. E.g. a single pdf or a book.

Rows is the number of documents in the result set of a query.

Start Offset is the index of the first document you want displayed.

Page Offset is used in pagination, but is the same as start offset.

Iterations are the number of times a taxonomy is queried.

Hit Count is the total number of documents matching the query.

QueryOptimizer library is the solution developed for the problem.

QTime is the time spent generating the in memory response for a query in Solr (milliseconds).

Elapsed Time is QTime plus serializing and de-serializing transmitting in Solr (milliseconds).

Query Time is the time it takes to perform a solr search from QueryOptimizer or the test framework (milliseconds).

Lucene is an open source free text search library from Apache.

Solr is an open source search server utilizing Lucene.

Solrconfig.xml contains the parameters to configure Solr.

QueryResultWindowSize . A window is a section of search results. It can be from 0-49, 50-99 etc. When querying the entire window in which the search match will be returned and loaded into cache. QueryResultWindowSize is the size of these windows.

QueryResultMaxDocsCached is the maximum number of documents a single query can have in cache memory.

Index is a sorted list of terms present in the data set. Contains links for finding the term locations.

Sharded index is an index split in smaller parts possibly on different servers to better cope with scaling issues.

Chapter 1

Introduction

Chapter 2

Theory

2.1 Gaussian Processes for Regression

$$k(x, x^{'}) = \sigma_f^2 exp\left[\frac{-(x - x^{'})^2}{2l^2}\right] + \sigma_n^2 \delta(x, x^{'}) \tag{2.1}$$

$$K = \begin{bmatrix} k(x_1x_1) & k(x_1x_2 & k(x_1x_3 & k(x_1x_4 & k(x_1x_5 \\ k(x_2x_1) & k(x_2x_2 & k(x_2x_3 & k(x_2x_4 & k(x_2x_5 \\ k(x_3x_1) & k(x_3x_2 & k(x_3x_3 & k(x_3x_4 & k(x_3x_5 \\ k(x_4x_1) & k(x_4x_2 & k(x_4x_3 & k(x_4x_4 & k(x_4x_5 \\ k(x_5x_1) & k(x_5x_2 & k(x_5x_3 & k(x_5x_4 & k(x_5x_5 \end{bmatrix} \tag{2.2}$$

$$\begin{aligned} x - x^{'} &= Distancebetweenobservationdata \\ l &= lengthparameter \\ \sigma_f^2 &= Maximumallowablecovariance \end{aligned} \tag{2.3}$$

Chapter 3

Solution

3.1 Fire and Sensor Simulation

3.2 Fire Interpreter

The fire interpreter's job is to receive input from the simulator and calculate values for the received cells. The calculated values are then posted to the visualizer. The advanced calculations are done by the library pyXGPR. This is a Gaussian Process Regression library implemented with Python. It produces a mean and a variance when used correctly. The first input parameter **X** is a list of points which tells where the training data is located. Another parameter **Y** contains the values to the training data. The last interpreter generated parameter **x star** contains the points where we want to find the mean and the variance. In addition to these parameters the library needs to be told what covariance functions pyXGPR should use to calculate the correlation between the cells in **X**, **Y** and **x star**. There is also added parameter values to these functions.

The most basic use of pyXGPR is one dimensional (line regression) where **X** is the location and **Y** is the value. The fire interpreter uses regression in three dimensions where **X** and **Y** are the map coordinates and an additional parameter **t** is for time. **t** is necessary to save earlier sensor data which later are utilized in calculations. It should also be mentioned that before this implementation, this was done by saving the best data. Best data is to be understood as the data which has the lowest variance. Data with lower variance would be applied to the saved map. This hack and the implementation of t is done because previous sensor data is important as long as they are weighted less than the newest sensor data. As time

increases there will be sensor data covering most of the map, but the old sensor data will have less weight and thus giving new sensor data the opportunity to be taken into account. Figure 3.1 illustrates this.

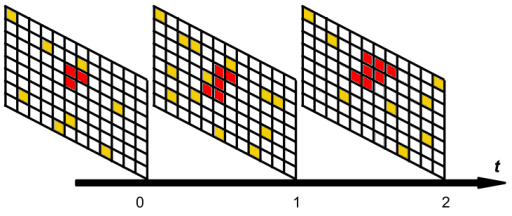

Figure 3.1: The fire interpreter saves the sensor data (yellow dots) with **t**.

The simulation map is 71 X 71 squares. GPR calculates the correlation between all points. This means pyXGPR creates huge matrices. As t increases the the matrices grow larger and thus the execution time rises. The first version of the program used more than 6 minutes to run on a normal laptop. To better the performance it was found necessary to locate some squares which did not need to be calculated. Cells containing sensors was therefore removed from **x star** since the values were already there. With the introduction of t the matrices became larger and execution time rose and once again it was necessary to find some steps to improve the performance. For a cell to be added to **x star** it had to be close to sensor which communicated fire. This was done by finding the euclidean distance between all cells which had no value and all sensors which communicated fire. This reduced the number of points in **x star** significantly. As time increases Another measure taken to reduce the execution time was to save cells which was calculated to be on fire. From these the euclidean distance to all points were calculated and if they were within a certain distance they would be added to **x star**.

Before the interpreter posts the calculated data to the visualizer it converts the mean values to discrete values which is used in the visualizer to decide how intence the fire is burning. A threshold to determine if there should be fire is set, but can be difficult to determine as time progresses. The input values for sensors sensing fire is 0 to 10, while sensors which are not sensing fire is converted from 0 to -1. The reason for the conversion is to get some more distance between fire and not fire. The fire threshold is set low to make sure all the realistic fire is covered.

3.2.1 Changes in pyXGPR kernel to support wind

The simulation use wind as a crucial parameter to decide where the fire is spreading. Wind is therefore clever to use in the interpretation process. To make it a useful parameter the kernel in pyXGPR has been edited. The kernel contains all the different covariance functions and noise functions. The modification has been done in the function which measures the squared distance between two points. The squared distance is measured by creating two matrices for each dimension. In the fire interpret these dimensions are x, y and t. One matrix containing all sensor data is deducted from a matrix containing all the uncalculated data. The dimensional result is multiplied with itself and added to a distance matrix. All dimensional results are added to a distance matrix.

$$\theta = cos^{-1}\left(\frac{vector_{ij} \times windVector}{vector_{ij} \times windVector}\right) \tag{3.1}$$

$$weight = \frac{\theta}{\pi} \tag{3.2}$$

$$\begin{bmatrix} distance_{11} & distance_{1j} \\ distance_{i1} & distance_{ij} \end{bmatrix} \times \begin{bmatrix} weight_{11} & weight_{1j} \\ weight_{i1} & weight_{ij} \end{bmatrix} \tag{3.3}$$

Before these calculations the interpreter implementations in this function creates a weight matrix which has the same column and row values as the distance matrix. This weight is calculated with the forumula in 3.2. The Hadamard product (see 3.3) of these two matrices is returned as the new distance matrix.

Where $vector_{ij}$ is the vector from sensor position i to uncalculated position j. The weight is further normalized to better fit the wind. If $vector_{ij}$ between $90°$ and $180°$ weight will be larger than 1 and less than $45°$ weight will be less than 1. The values in the weight matrix is multiplied with the values in the distance matrix. The $vector_{ij}$ the weight matrix is multiplied with $vector_{ij}$ the distance matrix. This modifies the distance matrix where some values are reduces and some are increased, determined by their vector direction. The correlation of cells looks at the distance between them. Therefore decreased distance gives a cell a higher probability of being on fire. It should be mentioned that sensors which does not detect fire will have a weight of 1.d

3.2.2 Unexpected predictions with wind

Figure 3.2: The left image illustrates when wind is used in predicting fire. The right image illustrates the faults with a pure wind implementation. The blue dots are where the there should have been predicted fire.

The implementation of wind is working, but has some drawbacks. The first in 3.2 illustrates how the fire is predicted when the wind is blowing from the east. The spread starts where the sensor is located and continuous in the shape of a triangle towards west. This looks good in the first image, but the second image illustrates the drawbacks. The wind is still blowing from the east and the two sensors detecting fire are on a line. The right tips of the predicted fire is where the sensors are located. The blue squares is used to highlight where there should be predicted fire. This situation occurs because the blue squares are closest to the second sensor. The vector which goes from the second sensor to the any of blue square has an angle which is more than $45°$ when compared to the wind vector which is $[-1, 0]$. Problems with predicting the middle part of the fire can occur when the the size of the predicted fire is becoming quite large. To solve both these problems there has been developed two solutions which have been tested with varying results. The first solution looks at the correlation between all sensors sensing fire while the second solution uses some techniques found in graphical programming. The best would be to avoid these and instead approach these problems with dynamic parameters. As the predicted fire grew larger the parameters changed. But problems arises with such a solution as well. The edge of the fire is more likely to be smudged out and it would in some cases predict fire in areas where there were no fire.

Solution 1 to unexpected predictions with wind

Burning sensor correlation was a way to solve the problem when applying wind. It creates a list of all sensors which are sensing fire. Each element in the list has vectors to all other sensors communicating fire. In 3.3 the green triangle is an uncalculated point while the numbered red circles are sensors sensing fire. There are vectors from sensor 1 to all other sensors sensing fire and a vector to the green triangle. This is an illustration of one of the entries in this list. The basis for this

theory is that there is probably actual fire between the sensors. The blue vector compares direction with all the black vectors, see 3.1. It finds the black vector which has the most similar direction and check if it is within a certain threshold. If this comes out positive the distance for this point to sensor one will be multiplied with a number less than one, else it would be multiplied with one. This approach

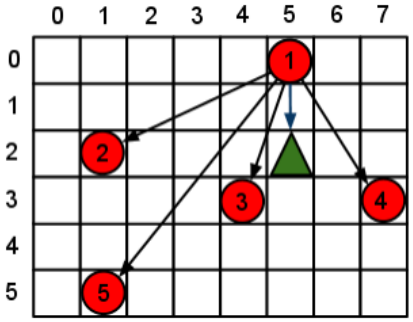

Figure 3.3: All sensors (red) are sensing fire and sensor one has a vector to all other vectors sensing fire. The green triangle is an uncalculated point. Burning sensor correlation looks for the black vector which has the most similar angle to the closest blue vector.

has a fault. When a situation like 3.2 (second image) occurs it would predict fire against the wind. The burning sensor correlation would neutralize the added wind to such a degree that it was removed. The complexity with this solution is high and thus the execution time rises fast as **t** increases and the fire interpreter uses more data.

Solution 2 to unexpected predictions with wind

In the second approach to solve the wind and filling issues, components from computer graphics were used. The outer boundary of the sensors sensing fire is chosen. The shape will be a convex polygon. Uncalculated cells within this polygon will have a higher probability of being on fire in the prediction. As in the burning sensor correlation implementation it is assumed its a higher probability of being fire between sensors sensing fire. To locate all the outer cells Graham's scan [1] was applied. When these points were found Bresenham's line algorithm was used to find the in between cells. The result was all points which the lines covered in figure 3.4 was retrieved. After this a scan line fill algorithm was used to find all the cells within the polygon. These steps resulted in a list of cell positions. This list was used when calculating the wind in accordance to wind direction. If the cell to be evaluated was inside this polygon the weight was multiplied with a

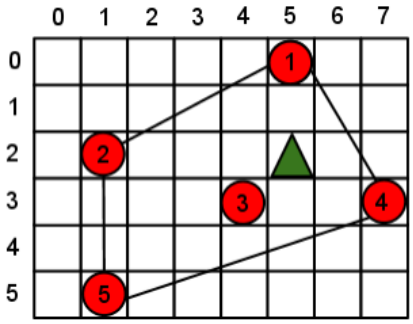

Figure 3.4: Locates all cells within the boundary.

number lower than one. This solution is also much faster than the first one.

Chapter 4

Discussion

4.1 Results

Chapter 5

Conclusion

Acknowledgments

We would like to thank our supervisors Folke Haugland and Jaran Nilsen for their constructive feedback that has led to progress in times when the project was at a stand still. We would also like to thank Integrasco for letting us use their office to store the server and work in, and University of Agder for lending us the server.

02 june 2011
University of Agder

Bibliography

[1] Alejo Hausner, CS Department, Princeton University *http://www.cs.princeton.edu/courses/archive/spr10/cos226/demo/ah/GrahamScan.html (2011).* Prentice Hall