Acceleration of decision tree computation using speculation

A Case Study of accelerating queries using speculation

Jonathan Reymond
jonathan.reymond@epfl.ch

ABSTRACT

Decision trees that involve query results in their decisions are often handled sequentially : before getting to the query execution of the sub-branch, we first wait for the result of the branch. To do that, we allocate to the given query all the computational resources available and use parallelism to accelerate its running time. But, due to the diminishing returns principle, this approach has its limitations.

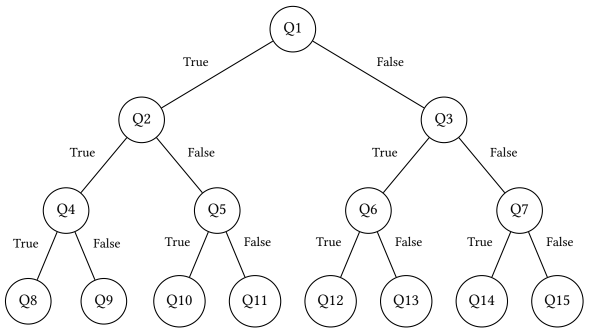
In this paper, we explore a way to accelerate the running time of the overall execution using speculation : instead of using all the resources for one query, we speculate on which branch the program will go and compute beforehand the given query(ies). To reach this goal, we consider only Sql queries. We first do an analysis based on the diminishing returns concept. Then we define a mixed-integer linear program to solve the problem of allocating efficiently the queries regardless of their position in the decision tree. We then convert it into a constraint program and add the specifications of the decision tree and the probability of occurrence of each branch to finally get a solver able to process decision trees of height at most 4.

1 INTRODUCTION

With the expansion of the application based on data analysis, many programs need to process queries over a given data set to be able to decide what to execute next. Not only programs, but also data analysts have to explore the data set by making some queries and infer some results to decide in which direction to continue. For instance, extract some statistics of a population like the average age, then counting how many people having a given disease on the studied group to finally decide whether it is relevant to, for example, develop a new medicine. This data-driven workflow, where the decisions redirect the program into one branch or another one, can be formulated with if-else statements and induces a decision tree. Let Q_i be a query. Let assume that our program consists of if-else statements as follows :

```
if condition(Q1):
    if condition(Q2):
        ...
    else :
        ...
else :
    if condition(Q3):
        ...
    else : ...
```

We could have for instance ***Q1 : "AVG(p.age); SELECT People as p"*** and ***condition(Q1) : "Q1 > threshold"***. Some other computations can naturally be present, but the main idea is that the query results define directly which branch will be executed or not. If we continue up to 15 queries, we get a decision graph like below :



At the end, only one branch from the top node Q_1 to a leaf $\{Q_8$, ..., $Q_{15}\}$ will be executed. A standard execution would allocate all the resources to compute the current query, get its value, and evaluate the condition to determine which will be the next query to compute. But due to Amdahl's law[1], we get what we call a diminishing returns effect : increasing the computational power will not always result in a significant speed-up. For instance, passing from 8 to 16 cores available to compute the given query will not necessarily induce a speed-up factor of 2, and as the number of resources increases, this effect becomes more and more important. So, for instance, if we know that the branch that will be executed is $[Q1, Q2, Q4, Q9]$, then it could be advantageous to split the resources and run a subset of queries in parallel.

Based on that observation, we introduce a task scheduler that, given the probability of a given branch or path to succeed, will output a sequence of batches where the first ones will contain the queries of the most probables paths/branches. These probabilities could be computed beforehand by considering only a sample of the data set, infer its distribution, then computing an approximation of the result of the studied query with a confidence interval to finally infer the probability for this query to satisfy its corresponding condition. The main idea of this scheduler is to use these probabilities to speculate on which branch the program will go, compute at the same time the next potential queries in batches (always based on their probabilities), and if the speculation was incorrect, correct it by computing the next batches. For instance, the scheduler computes $[Q1, Q2, Q4, Q8]$ in one batch thinking that the program will choose this branch, but then the program evaluates $condition(Q4)$ and $Q9$ should be executed instead of $Q8$ whose result already computed is not necessary. So the scheduler will have then to compute $Q9$ to finish the execution of the project.

We will demonstrate that this approach gives a running time improvement in the case when its speculation is correct, but that the penalty it endures when it does a mistake varies greatly depending on the instance studied: on the queries involved, on the structure and size of the decision tree, on the probabilities, etc. During the

experiments we made, some important facts have to been taken into account: the number of partitions we consider to evaluate the queries and also the set of queries we process. Other works using speculation have been made, for instance to process exactly SQL queries having an inner query inside by executing the inner and outer query in parallel[3].
This paper is organized as follows : After describing the setup we used, we present a study to illustrate the diminishing returns effect we can observe in a set of SQL queries. In the section 4, we describe our scheduler optimizer step by step by first solving the sub-problem of allocating the queries altogether without considering the decision tree. To do that we give a MILP formulation as it can solve the problem optimally. We then transform it into a constraint problem for practical reasons that will be explained. Afterwards, we state the main problem and reuse our CP formulation to solve it. The section 5 gives results we obtained by using this scheduler and illustrates with examples also its limitations. Finally, we make a conclusion, share the learning outcomes of the project and describe what could be done to go further on this work. Note that all the implementation of this paper can be found in the following Github repository.

2 EXPERIMENTAL SETUP

The data set we used to do our evaluations was the *Internet Movie Data Base* (IMDB), which stores information related to the cinema area: actors, movies, movie companies information, etc. It has the main advantage to possess large number of tables and to be sufficiently large for our purpose.
For the queries, we have implemented the complete Spark version of a subpart of the Sql queries appearing in the Join Order Benchmark (JOB) introduced by Leis and al.[2], but with slightly different filter parameters to always get a non-empty result. These queries are exclusively based on joins, filter and aggregate operations. So we also have implemented in Spark another set of Sql queries we defined using also *GROUP BY* and *ORDER BY* operations.
For stating the running time of each query given the number of cores and the number of partitions, we do not have considered the time to allocate the tables it used and repeat the operation 4 times to get a good accuracy.
The experiments were made using a Linux machine with the kernel version 4.15. having 16 cores of type Intel Xeon E5-2640 v2 at 2GHz, with 8 cores per socket and 256 GB of main memory.

3 DIMINISHING RETURNS

In the following figure 1, as the number of resources grows, the speed-up gain get lower and lower. For instance, if we consider the processing time for 4 partitions, we pass from 124.47 seconds to 56.40 seconds when we use 2 cores instead of one. So a speed-up of factor of 2.2. But getting 16 cores instead of 8 will only cause a speed-up of 1.2 due to the diminishing returns notion. For the queries we have studied and as shown in the plot, changing the number of partitions will induce some differences in the curves, but the overall behaviour of them will not drastically change when running a single query, which could not be the case when running in parallel multiple queries. To illustrate even more clearly the diminishing returns concept, we do the following experiment on a set of queries: we first

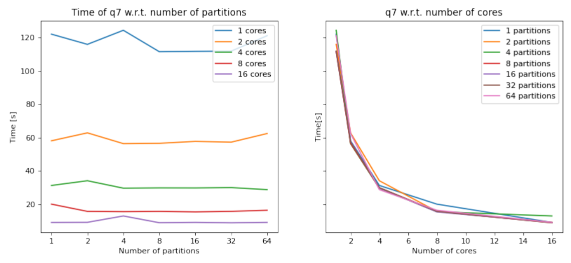


Figure 1: Running time Q7

compute 2 queries in parallel, then we compute them sequentially with the same resources, then we compare the results. For instance, we compute $Q1$, $Q2$ in parallel with 8 cores, then compare with the computing $Q1$ with 8 cores, then $Q2$ with 8 cores. It yields the results shown in figure 2. In the first graph we observe, as the resources

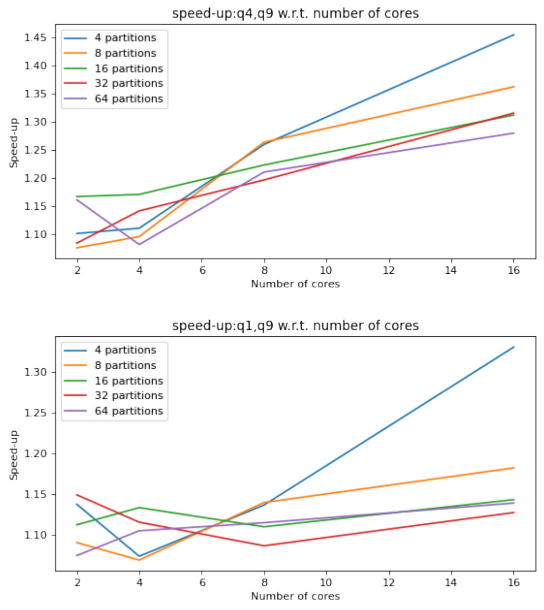


Figure 2: Sequential vs Parallel time

increase, a constant increase of the speed-up when computing the queries in parallel, picturing well the fact that the diminishing returns effect becomes more and more great with respect to the resources available. But as it appears clearly on the second graph, for another instance, the increase of the speed-up with respect to the number of cores is much less important, except for one particular partition. So in the present case, the number of partitions really matter, and we should choose it carefully. Nevertheless, this number depends on numerous factors that are hard to infer beforehand: in this instance as an example, the three queries $Q1$, $Q4$, $Q9$ use almost the same tables (except $Q4$ who uses 2 different tables over 5) and $Q1$ and $Q4$ are both Sql queries using exclusively joins and aggregates, so the number of partitions depends on other factors such as the distribution of the data set or the order of the joins, etc. Nevertheless, in all our experiments, we always get a speed-up when computing the queries in parallel

4 SCHEDULE OPTIMIZER

For the first part of our optimization process, we neglect the probabilities of a given query to be process. Assume that we have already computed the matrix T, where T_{qc} is the running time of query q with c cores. Let C be the total number of cores of our machine and Q the total number of queries. Then our optimization problem consists simply of placing the queries into batches with a given number of cores allocated for each, and minimize the total running time with respect to T. If R is the number of possible runs/batches, then the total running time is defined as the sum of the maximum query running time of each batch r.

4.1 MILP approach

We first introduce mixed-integer linear program of the optimization problem. Let X_{qcr} be a boolean variable that is 1 if the query q runs on the run r with c cores. Then we can formulate our schedule problem as follows :

$$\min \sum_{r=1}^{R} \max_{q} \left(\sum_{c=1}^{c} T_{qc} X_{qcr} \right) \tag{1}$$

$$\sum_{c=1}^{C} \sum_{r=1}^{R} X_{qcr} = 1 \qquad \forall q = 1 \ldots Q \tag{2}$$

$$\sum_{q=1}^{Q} \sum_{c=1}^{C} c X_{qcr} \leqslant C \qquad \forall r = 1 \ldots R \tag{3}$$

The objective function 1 can be reformulated as follows : for each run r, take the maximum running time of the queries that are assigned to it (i.e. where $X_{qcr} = 1$). The first constraint 2 ensure that each query is assigned exactly once in the whole program. Finally, the second constraint 3 ensure that for each batch/run r, the total number of cores allocated to the queries belonging to that batch not exceed the number of cores of the machine. . Theoretically, the maximum number of runs R is equal to the number of queries, but we can set it to $Q/2$ to reduce the number of variables without restricting too much the possibilities. The "max" term in the objective function can be replaced by another variable k with these constraints :

$$\begin{aligned} &\min \sum_{r=1}^{R} k_r \\ &\sum_{c=1}^{c} T_{qc} X_{qcr} \leqslant k_r \qquad \forall q = 1 \ldots Q \end{aligned} \tag{4}$$

This step is mandatory to be able to use any MILP-solver. The solver we used, MOSEK[1], proceed by first relax the problem by saying that $X_{qcr} \in [0, 1]$ and then from this solution compute a feasible solution (i.e. $X_{qcr} \in \{0, 1\}$). Despite giving the optimal solution, it is well known that MILP are NP-hard, and due to the number of variables ($= Q*R*C$), it does not scale for a number of queries larger than 4-5. For this reason, we moved to a constraint-programming formulation.

[1]https://www.mosek.com

4.2 CP approach

A constraint program is similar to a MILP. However, it differs from the latter by the way it is solved. A constraint program is viewed as a logical problem and not as an algebraic problem like the MILP relaxation. CP-solvers do not have a relaxation step, but tries different assignments and check that it respects the constraints. This approach has two main advantages :
1. The algorithm to solve is very parallelizable.
2. All the constraints in an MILP can be reformulated in a CP-program, and there exists other types of constraints.
CP-solvers are well suited for job scheduling, i.e. assign to each job a machine to minimize the running time given some constraints, which is exactly what we try to do. The solver we employed was the CP-SAT solver part of the Or-Tool software developed by Google[2].

4.2.1 CP formulation. The second point we mentioned will allow us to reduce the number of variables and to scale better. We can first express the assignment of a query not with a matrix, but with an integer vector, removing the C dimension :

$$X_{qr} = \begin{cases} c & \text{if query } q \text{ is assigned to } r \text{ with } c \text{ cores} \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

This allows us to simplify the expression $\sum_{c=1}^{c} T_{qc} X_{qcr}$ in 4 to $T_{qX_{qr}}$, and with other appropriate variable we manage to pass from $O(Q * R * C)$ variables to $O(Q * C + Q * R)$.
With this formulation, we can get the optimal solution for any list of queries having a length at most 8. We need therefore some heuristics to process larger instances.

Using the expressiveness of CP programs, we want to get, for the assignment of one query, not a boolean matrix, but a vector of length R named V_q containing directly the number of cores assigned to it in the given run. We therefore need to enforce the constraint that this vector has exactly one non-zero entry to replace the corresponding constraint 2 in the MILP formulation. To this end, let $x_q \in \{1, 2, ..., C\}$ be the number of cores assigned to the non-zero entry. We first define a matrix A where :

$$A_{qr} = \begin{cases} x_q & \text{if } r = R - 1 \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

We introduce then an index vector I_q of length R where $I_{q_i} \in \{0, 1, ..., R-1\}$ s.t.

$$A_q[I_{q_j}] = V_{q_j} \quad \forall j \in \{0, 1, ..., R-1\}$$

and ensure that every I_{q_j} takes a different value. It results that the vector V_q has the value x_q only in the index j where $I_{q_j} = R - 1$. Other ways exist to get this constraint, but this formulation was the one giving the best running time.
At the end, we obtain the following CP program :

[2]https://developers.google.com/optimization

$$
\begin{array}{ll}
\multicolumn{2}{c}{\min \ \sum_r k_r} \\
\sum_{q=1}^{Q} V_{q_r} \leqslant C & \forall r \in \{1,\dots,R\} \\
A\left[I_{q_j}\right] = V_{q_j} & \forall j \in \{0,\dots,R-1\},\ \forall q \in \{1,\dots,Q\} \\
Distinct\left(I_q\right) & \forall q \in \{1,\dots,Q\} \\
k_r = \max_q \left(T_{q,V_{q_r}}\right) & \forall r \in \{1,\dots,R\}
\end{array}
\tag{7}
$$

We also define the matrix T slightly differently by first making sure that all its values are integers, since CP programs support only float values in the objective function. This step will introduce some inaccuracies, and we therefore multiply the matrix by a large constant to reduce these inaccuracies. We also add a zero column for the case we want to know the running time of a query with zero cores. This CP formulation can scale a list of queries of length up to 10. While at first sight it might be not a great improvement, it avoids to get very small sub-set of queries due to the heuristic splitting.

4.2.2 *CP heuristic step.*

The basic idea of this step is to smartly split the query list in two until each sub-list has length at most 10, find the optimal solution of each and then add them to get the final result. The way we process to find a good split is to reuse our CP formulation, set R to 2, and C' to a function of C and Q so that the number of cores we assign to the queries is approximately in the same order as the number of cores we would assign in a normal execution. We also change the inequality in 3 into an inequality to reduce the possibilities (this will also enforce in a way to distribute the queries more evenly).

4.3 Probabilities

Here we introduce the probabilistic approach. In fact, we are primarily interested not in computing all the possible queries, but in selecting a subset of them so that we can terminate quicker. If we take a look again at the decision tree1 and imagine that, finally, the branch taken by the program was [Q1, Q2, Q4, Q9], then, we want our schedule optimizer to output these queries in its first batches. We also want to avoid the execution of the queries not belonging to that branch. In other words, if the optimizer outputs for instance [[Q1, Q4, Q2], [Q8, Q5, Q10], [Q5, Q6, Q13], ...], the user will only have to compute the first two batches and ignore the others, creating therefore an early exit mechanism.

What we want, in expectancy, is that the running time of executing sequentially the program is lower than computing queries in batches, considering the probability of each branch to be executed. This probability can easily be computed by multiplying the probability of each edge of the path to be taken. Let set_p be the set of queries belonging to the path p, $pathTime_p$ the running time after all queries in set_p are processed. For each query q, let $queryTime_q$ the time spend after computing the given query and $indexRun_q$ the index of the run where q is processed. $runTime_r$ the running time passed after having executed the run r. Which results in the following CP program :

$$
\begin{array}{ll}
\min \sum_{p \in \mathcal{P}} proba_p * pathTime_p & \\
\sum_{q=1}^{Q} V_{q_r} \leqslant C & \forall r \in \{1,\dots,R\} \\
A\left[I_{q_j}\right] = V_{q_j} & \forall j \in \{0,\dots,R-1\}, \\
 & \forall q \in \{1,\dots,Q\} \\
Distinct\left(I_q\right) & \forall q \in \{1,\dots,Q\} \\
k_r = \max_q \left(T_{q,V_{q_r}}\right) & \forall r \in \{1,\dots,R\} \\
indexRun_q = r & \text{for } r \text{ s.t. } V_{q_r} > 0,\ \forall q \in \{1,\dots,Q\} \\
queryTime_q = \sum_{r=0}^{indexRun_q} k_r & \forall q = 1,\dots,Q \\
pathTime_p = \max_{q \in set_p} \left(queryTime_q\right) & \forall p \in \mathcal{P}
\end{array}
\tag{8}
$$

Where $proba_p$ is the probability of path p to happen in the decision tree. We can equivalently formulate it as follows :

$$
\begin{array}{ll}
\min \sum_{p \in \mathcal{P}} proba_p * pathTime_p & \\
\sum_{q=1}^{Q} V_{q_r} \leqslant C & \forall r \in \{1,\dots,R\} \\
A\left[I_{q_j}\right] = V_{q_j} & \forall j \in \{0,\dots,R-1\}, \\
 & \forall q \in \{1,\dots,Q\} \\
Distinct\left(I_q\right) & \forall q \in \{1,\dots,Q\} \\
k_r = \max_q \left(T_{q,V_{q_r}}\right) & \forall r \in \{1,\dots,R\} \\
indexRunQuery_q = r & \text{for } r \text{ s.t. } V_{q_r} > 0 \\
runTime_r = \sum_{j=0}^{r} k_j & \forall r = 0,\dots,R-1 \\
indexRunPath_p = \max_{q \in set_p} \left(indexRunQuery_q\right) & \forall p \in \mathcal{P} \\
pathTime_p = runTime\left[indexRunPath_p\right] & \forall p \in \mathcal{P}
\end{array}
\tag{9}
$$

Where $indexRunPath_p$ is the index of the run where the path p terminates and $runTime_r$ the total running time until the run r ends. The objective function describes indeed the expectancy of the total running time of all paths : if we consider the completion of a given path p as a Bernouilli variable $\sim Bern(proba_p)$, then the term $proba_p * pathTime_p$ gives its expected running time. A proposal to minimize all the branches altogether is to simply add their terms. This approach has its limits and other formulations are possible.

An important note to take into account is that in this CP formulation, we treat the probability of a whole path and not for a particular query. So if the two firsts batches given by CP as a result contains all the queries of the shortest path, then this CP will not differentiate if the order of both are switched since it does not change the total running time of its execution. To illustrate, if we imagine that the shortest path is $[Q1, Q2, Q4, Q8]$ in the figure 1 and the CP returns for the first two batches $[Q1, Q2]$ and $[Q4, Q8]$. Returning $[Q4, Q8]$, $[Q1, Q2]$ will also lead to the same cost since this is when both have completed that the first path is completed. So, to fix this issue, during the program execution, we have to first find in which batch is the query we want to execute located and only after that execute the given batch.

4.3.1 *Split procedure.*

We also need to adapt our split algorithm to handle paths. After having executed the same heuristic step but with the new CP formulation stated above. We also introduce a new objective function that minimizes the longest expected runtime of any paths. In other words, we have instead this following CP :

$$
\begin{array}{ll}
\min \ maxPathTime & \\
\sum_{q=1}^{Q} V_{q_r} \leqslant C & \forall r \in \{1, \ldots, R\} \\
A\left[I_{q_j}\right] = V_{q_j} & \forall j \in \{0, \ldots, R-1\}, \\
 & \forall q \in \{1, \ldots, Q\} \\
Distinct\left(I_q\right) & \forall q \in \{1, \ldots, Q\} \\
k_r = \max_q \left(T_{q, V_{q_r}}\right) & \forall r \in \{1, \ldots, R\} \\
indexRunQuery_q = r & \text{for } r \text{ s.t. } V_{q_r} > 0 \\
runTime_r = \sum_{j=0}^{r} k_j & \forall r = 0, \ldots, R-1 \\
indexRunPath_p = \max_{q \in set_p} \left(indexRunQuery_q\right) & \forall p \in \mathcal{P} \\
pathTime_p = runTime[indexRunPath_p] & \forall p \in \mathcal{P} \\
maxPathTime = \max_{p \in \mathcal{P}} proba_p * pathTime_p &
\end{array}
\tag{10}
$$

The first reason why we choose this objective function instead of the other one is because the completion time to solve this CP formulation was, experimentally, more than two times faster for a tree of height 3. Also, for a tree of height 4 the old objective function simply does not scale in contrary to this formulation. The other reason is that if we consider the first objective function, it takes, in a certain way, a weighted average of the running time of all paths where the weights correspond to their probabilities. So the effect is that it will highly consider the paths with high probabilities and give much less importance to the other ones. The other formulation will result in a similar behaviour, but practically the difference of consideration between the most probable and less probable paths will be more attenuated. So, essentially, for the first objective function the runtime of the most probable paths will be lower, but the overall runtime will be larger than the second one.
In a sense, the second one is more stable than the first one. And since the task here is to split the queries into two sub-lists and to avoid early wrong decisions, the more stable objective function is more adapted to this situation.
During the splitting, the set of queries belonging to a path will also need to be updated. Let $queries_{left}$ and $queries_{right}$ be the corresponding sets of queries appearing in the first ($left$) and second run ($right$). Let $paths_{left}$ be the set of paths whose queries only appears in the first run and $paths_{right}$ the paths who complete in the second run. We define

$$remainQueries = queries_{left} \setminus \bigcup_{p \in paths_{left}} set_p$$

the set of queries in $queries_{left}$ not appearing in the paths in $paths_{left}$. We add them in the $queries_{right}$ set since they slow down the completion time of the first run where we have the most probable paths. Finally, we remove from the paths completing in the second runs all the queries appearing in the first run. It is very probable that $remainQueries$ is empty, but depending on how we define the objective function (see next section), it may not.

4.3.2 Objective function discussion. As discussed previously, the choice of the objective function impacts greatly the result of the CP. We have already introduced two objective functions, and it is possible to amplify their effects. For instance, for the first, we could have instead the following :

$$\min \sum_{p \in \mathcal{P}} proba_p^2 * pathTime_p$$

So consider even more the most probable paths and make therefore less stable decision. The contrary can be done by replacing the power of 2 by a $\beta \in]0, 1[$. This is also possible to different objective functions with a linear combination of them. We could also add regularisation terms to the current objective function such as $regFactor * \sum_r k_r$ to also minimize the overall running time, i.e. minimize the risk when we choose the wrong path. Another important way that could potentially handle better the case when only a few probabilities are considerable and the other much less important would be to optimize only over these paths directly by defining a threshold over the probabilities and add a regularization term to minimize the overall time to handle the other paths.
For simplicity, in this project we will keep only the current objective functions as defined above.

5 RESULTS

In this study, to compute the matrix T, we have only considered the case for a fixed number of partitions equals to 16. Furthermore, for the probabilities of each path to happen, we assign them a random value described.
We first illustrate what would be the running time if we execute all the queries of a decision tree of height 2. Let $[q1, q2, ..., q7]$ be the queries inside this tree ordered in the same way as 1 with their running times that we have already pre-computed. The sequential approach to execute all these queries would be to allocate all resources at each step to each single query. On the other side, using our first CP formulation (without probabilities), we get these following diagrams 3.To explain the notation, "q1:16" means that the query $q1$ will run with 16 CPUs in parallel within its batch.

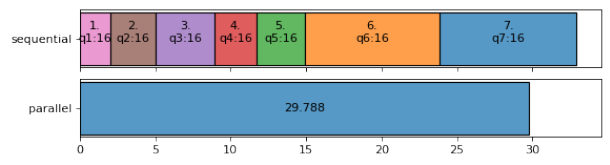


Figure 3: Sequential vs Parallel execution

Indeed, we observe an improvement of the total process time. But naturally, in the sequential process, we only process one branch. If for instance beforehand we know that the probability to take the path $[q1, q2, q5]$ is almost 1 and the other probabilities near zero. Using our CP implementation, we get the following graph 4, where the third part indicates the running time of the paths given by the solution of the CP. In this case, for this simple instance, we get a speed-up of approximatively 10%. An important note to take into account in this graph and the following ones is that the running time of the paths displayed corresponds to the case when all the queries are executed, which won't be done in practice and is therefore lower. For instance, let assume that we have evaluated the first batch and that the statement of $Q2$ is not satisfied, resulting to

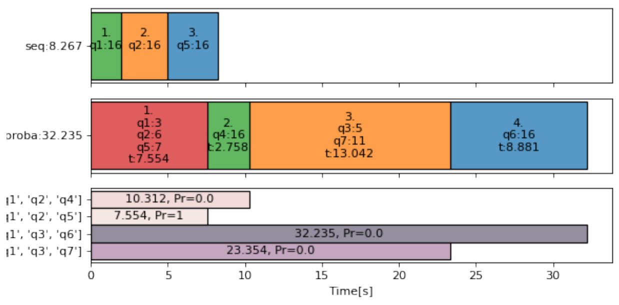


Figure 4: Seq vs 1-Probability execution

the fact that it is not necessarily to execute the path $[Q1, Q2, Q4]$ (please refer to the decision tree graph 1). So it would mean that we do not have to execute the green batch, where $Q4$ is, and therefore the real running time of the two remaining paths is indeed lower. Another remark that shows up, is that due to how we defined our objective function, the second path that has the lower cost is the neighbour of the chosen path, simply because it has only one different query in its set whereas the others have two. This behaviour would be even more noticeable as the height of the tree grows.
But before that, we can take the example where a probability for a given query to satisfy its condition in the if-else block is 1/2. It results that each branch has the same probability to be processed. Comparing the expected completion time of the sequential execution and the first CP formulation yields to the figure 5 where the third part indicates the running time of the paths sequentially.

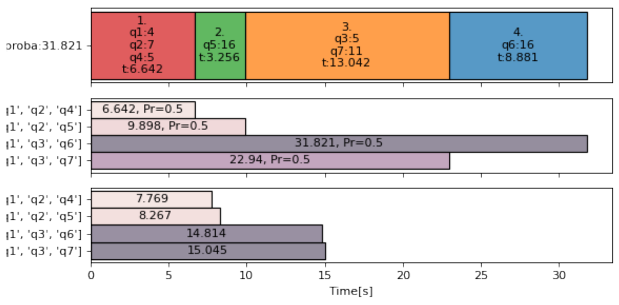


Figure 5: Seq vs 1/2-Probability execution

In this example, we have a speed-up only if the first path [q1, q2, q4] is taken. Indeed, due to how the objective function is defined, the optimal strategy the solver has to take is to give advantage to the potential shortest paths, and since it has to complete all queries, the other paths will suffer from a large completion time. Using another objective function as stated above could indeed help to reduce the overall completion time, but not correct in this case the fact that we have to complete all the queries in the graph.
As we stated the problem, i.e. decide which branch the program will execute and allocating the resources to optimize the time, having a well-balanced decision tree whose branches have a similar probability will always lead to choose the branch with the potential minimum running time. To correct this behaviour, a possibility would be to not consider the entire decision tree to avoid early decisions.

We now study a decision tree with height 4, i.e. with 31 queries inside it. Primarily, we will set the probabilities of the queries to very large values only in the left most side part of the tree (the last three entries in both sub-graphs), note that in the graph the paths are annotated not with the names of the queries, but with the indexes of the nodes. This leads to the picture 6. For two of them we

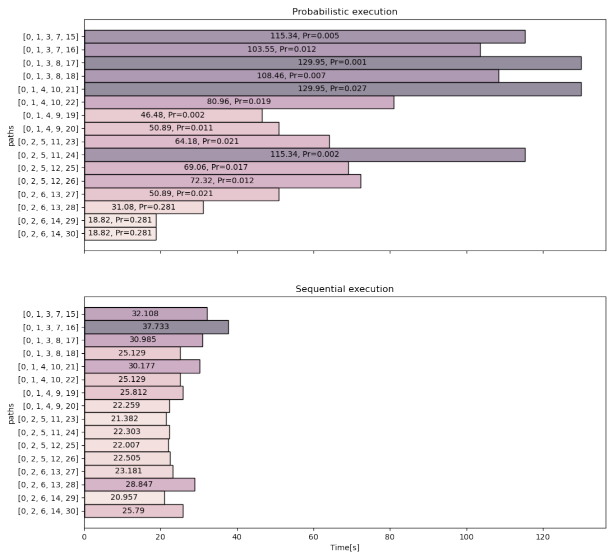


Figure 6: Execution with tree of depth 4

indeed get a speed-up factor of respectively 1.37 and 1.11, and for the third one, the result is not far from the sequential execution. If we compare the average running time, we get 25.2 seconds for the sequential execution and 22.9 for the parallel one. So, even though we had to use some heuristic steps by splitting in part the query list due to the size of the instance, the results remain consistent.
If, on the contrary, we look to the case where we have only large probability values on the two extremities of the decision tree (the first and last entry in the graph) as depicted in the figure 7. We have still an improvement for the right-side part, but for the other one the total time when considering that the first node statement leaded to it is 40.41[s] instead of 32.10[s] for the sequential execution. In such instances where a given node has a high probability to be executed and that its corresponding probability to be true or false near 1/2 (as in here with the root node), it would be more beneficial to not explore entirely its sub-tree, but to define a maximum height to look at.

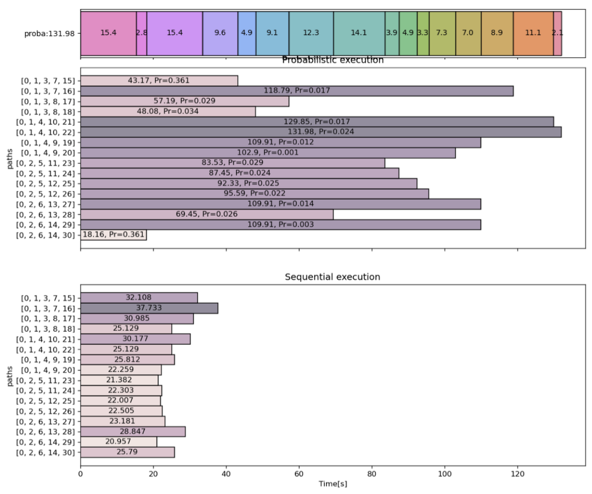


Figure 7: Execution with tree of depth 4

6 CONCLUSIONS

In this report, we presented an algorithm that, given a decision tree who needs for each node to compute a SQL query to choose which branch to go, and the probability of each branch to occurs, allocates the available resources efficiently for each query to minimize the running time of the branches having the highest probabilities. We demonstrated that the approach taken could indeed result in a diminution of the processing time needed to complete the decision tree, in some instances, but the potential gains are highly dependent on the structure of the decision tree, on how great is the diminishing returns effect and even more on the probability distribution of the branches. As a matter of fact, the potential speed-up gain of computing queries in parallel is highly dependent on the running time of the SQL queries executed individually. The results we obtained also do not reflect entirely the case when the program makes the wrong speculation and suffer from a penalty. So for instance if the program computes exactly the path we speculated except the last node, a normal execution would simply allocate all the resources to the remaining query. But in the way we described it, it is possible that the remaining query belongs to a batch containing other non-relevant queries, slowing down the process.

So the first simple improvement we can implement is that when our speculation was false, i.e. the batch we processed contains unnecessarily queries, we could recompute the scheduling for the sub-tree defined from the node where we haven't processed the query yet. As said earlier, the choice of the objective function could also give better results. A last proposition would be to directly not consider the whole decision tree, but only a sub-part of it and stop at the nodes where the probability of their statements is around 1/2.

To go further, we have here only done our experiments using SQL queries, but because of the way the problem is defined, any program containing a decision tree where the evaluation of a node implies a lot of computation and suffers from diminishing returns could potentially use the same optimizer.

For the final thoughts of the project, various misadventures could have been avoided in retrospect and would have saved a lot of time. For instance, using Spark-SQL instead of Spark to avoid loading the data set manually and to explicitly write each SQL query in Spark, also to directly use Python for the optimization part.The learning outcomes were various. Among them, learning the constraint programming paradigm, how to load a data set using Spark only, define how to make two programs in different languages interact, be more fluent in Python/Spark, how to use a cluster, etc.

REFERENCES

[1] Jim Gray and Prashant J. Shenoy. 2000. Rules of thumb in data engineering. *Proceedings of 16th International Conference on Data Engineering (Cat. No.00CB37073)* (2000), 3–10.

[2] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215.

[3] Panagiotis Sioulas, Viktor Sanca, Ioannis Mytilinis, and Anastasia Ailamaki. 2021. Accelerating Complex Analytics using Speculation. In *CIDR*.