Multi-Cloud Query Optimisation with Accurate and Efficient Quoting

Damien T Wojtowicz, Shaoyi Yin, Jorge Martinez-Gil, Franck Morvan, Abdelkader Hameurlain

Multi-Cloud Query Optimisation with Accurate and Efficient Quoting

Damien T. Wojtowicz*, Shaoyi Yin*, Jorge Martinez-Gil†, Franck Morvan* and Abdelkader Hameurlain*
* IRIT Laboratory, Paul Sabatier University, Toulouse, France, {firstname.lastname}@irit.fr
† Software Competence Center Hagenberg GmbH, Hagenberg, Austria, jorge.martinez-gil@scch.at

***Abstract*—A recent trend among major organisations is to release their datasets in the cloud over various Database-as-a-Service (DBaaS) providers' premises, creating a use case for multi-cloud querying. As identified in the literature, middlewares with such capabilities should quote the monetary cost and the response time of the queries in order to gain the trust of their users, and also optimise the queries so as to avoid cost overruns and meet the quotations. Considering those requirements, this paper introduces an accurate cost model and an efficient execution plan search strategy for dealing with large-scale multi-cloud queries. The former is an ensemble learning stack leveraging online machine learning models, and the latter is a randomised method inspired by iterative improvement. We evaluated our middleware over simulated providers by using the Join Order Benchmark. Experiments showed that the cost model manages to correct the estimations from the providers. The randomised strategy can produce more efficiently execution plans that yield better performances and a lower monetary cost compared to an exhaustive approach from previous work.**

***Index Terms*—Multi-Cloud, Cost Model, Online Machine Learning, Query Optimisation, Database-as-a-Service.**

I. Introduction

The increasing use of the cloud is leading public data producers to release their datasets in the cloud. Such datasets are of utmost importance in several fields, and becomes readily available for analysis over the various services hosted by the cloud providers. They may also be published as relational databases available on Database-as-a-Service (DBaaS) infrastructures, where data storage as well as querying is available on a pay-per-use basis. Cross-analysing data from such databases by leveraging the computing capabilities of their host providers is therefore possible, but requires systems providing multi-cloud querying capabilities.

The topic of multi-cloud data management raised interests in the industry as well as in the DB research community (as highlighted by the Seattle Report [1]). Specialised middlewares for multi-cloud database integration have consequently been introduced [2], [3] to let users write multi-cloud queries independently from provider-arranged multi-cloud federations. In previous work [3], we identified two main design requirements for such a middleware. First, it should be able to quote the monetary cost of the multi-cloud queries, so as to let its users control their expenses. Quotations must be as accurate as possible because the reputation of the middleware is at stake. Second, it should be able to optimise the multi-cloud queries in a multi-objective way, by considering response time and monetary costs in accordance with the quotation. A sub-optimal multi-cloud execution plan leads to extra monetary costs to users; the middleware should therefore correct poor optimisation choices whenever possible.

With the middleware presented in [3], we succeeded to meet these requirements for a small number of providers. Experimental results encouraged us to try with many more simulated providers, and therefore to overcome some limitations of the original proposal. First, taking into account the independence of DBaaS providers, we computed the quotations based on the raw output cardinality estimation from the DBaaS providers' DBMS. When there are many providers, their initial estimation errors may propagate exponentially [4] and lead to a discrepancy of several orders of magnitude between the quotation and the real expense. Second, we also sought optimal execution plans and competitive quotations by fully enumerating all the possible linear plans, a strategy only suitable for small-scale queries, because this operation is known to be NP-hard [5]. In this paper, we aim to produce accurate quotations efficiently for large-scale multi-cloud queries.

Therefore, we introduce a multi-cloud cost model that reprocesses the estimates from the DBaaS providers' DBMS, based on an ensemble learning stack. Our learning stack does not only estimate the output cardinality, but also the response time of the sub-queries. Inspired by recent encouraging results [6], we decided to use online machine learning models [7]. Our choice is reasoned by (i) their lightweightnesss, (ii) their cold-start ability and (iii) their robustness w.r.t. statistical shifts, may they be data distribution changes or design changes in the database engine. This cost model serves as an input to both a quotation calculator and a query optimiser.

We see those as the two faces of the same software component, that should aim at solving efficiently both problems. Indeed, we suggest that the optimiser should not only be used to orchestrate the outsourced execution of the multi-cloud queries, but should also generate quotations using estimates from the cost model. In order to provide querying capabilities over queries involving a large number of providers, we designed an optimiser that uses a single-phase approach and an iterative-improvement-inspired search space exploration strategy so as to avoid never-ending quotation calculation time and prohibitive optimisation costs. This type of strategy is beneficial when the middleware needs to re-optimise queries during their execution in order to better meet quotations, as is in our case.

This paper is organised as follows. In Section II, we review

related work on multi-cloud query processing, cost-models and search space exploration. We present our multi-cloud cost model in Section III and the search space exploration strategy in Section IV. Finally, we evaluate and discuss our proposals in Section V and conclude in Section VI.

II. Related Work

The idea of using the readily available computing capabilities of cloud providers hosting public data for distributed analytics originally arose for MapReduce. Although not directly applicable to relational databases available through DBaaS, it has nevertheless been shown that using multi-cloud resources for processing is cost-effective - in the sense that it can be less expensive than single-cloud or local processing [2].

We reached the same conclusions while working on a middleware that processes SQL queries written on a multi-cloud relational schema aggregating databases from different DBaaS providers [3]. We identified two key design requirements for such a middleware. It should indeed (i) compute quotations to let the users be in control of their expenses, and to (ii) optimise those queries in order to meet the quotation so as to safeguard the reputation of the middleware. In accordance with the literature [4], we observed that accurate estimations of the sub-queries' output cardinality is vital in order to produce quotations including both the monetary cost and the response time of the queries as well as to properly optimise the multi-cloud queries.

DB2's LEO optimiser introduced history-based output cardinality estimation [8]; this paradigm is now embodied by machine learning. Several deep learning models [9] have been proposed so far, achieving high accuracy. However, their applicability to a multi-cloud data integration middleware is questionable because of the important monetary cost entailed by the construction of a training dataset large enough for them. Recent findings showed that output cardinality estimation correction can be achieved with a high level of accuracy using online machine learning models [10].

This approach to output cardinality estimation exploits the DBMS's knowledge of its database while improving the accuracy of its estimates. Besides, this approach has a lesser need for a training dataset while having, in the long run, both the possibility of obtaining precise estimates and adaptability to statistical shifts thanks to its constant ingestion of new data points. In addition to the cardinality estimation, we also studied some methods which predict query response times. In the literature, the most accurate results so far were obtained using a k-neighbours regression [11]. It was adapted in an online learning fashion in our middleware to post-process the response time component of the quotations [3], along with an online linear regression to post-process the monetary component. Nevertheless, both models can make errors, occasionally by an order of magnitude. This is a well-known phenomenon in DBMSs, whose remedy involves correcting the estimates as soon as possible, which we suggest to achieve with a multi-cloud cost model based on an ensemble learning stack.

Another important aspect of these works is the exploration of the search space for the construction of a training dataset and the evaluation of the estimations, seeking a more efficient learning [9]. Some methods require an exhaustive exploration of the search space [12]. This exhaustiveness leads to the execution of many supplementary queries, which is not desirable in our context. We therefore choose to adapt iterative improvement [13], a textbook randomised strategy method that is well known to find good execution plans and avoids a costly, NP-hard exhaustive search space exploration.

III. Accurate Costs Estimation

For a given query, the providers' DBMS can estimate the output cardinality of its results from which it is possible to derive the size, and thus their storage and transfer monetary costs. These estimates are produced by a cost model, and relying on them blindly would be a mistake as these components are known to be error-prone [4]. In order to improve the accuracy of the quotations as well as the quality of the optimisation process of multi-cloud queries, and thus to protect the users of our middleware from unexpected expenses and delays, we propose a method to refine these output cardinality estimates. The approach we follow is that of ensemble learning [14]: several regression models are stacked, and a classifier is responsible for determining, given the features of the query, which of the estimations should be the more accurate.

The statistical distribution of these characteristics is expected to shift, possibly due to the evolution of the providers' cost model or to changes in the content of the databases. Inspired by previous work [10], we propose to design a cost model using online machine learning techniques. Thanks to the forgetting capabilities of those techniques, we expect that our cost model will not suffer from these shifts. Moreover, they enable the models to be updated in constant time with each new observation available, we therefore expect that the optimisation costs will not be increased but marginally.

A. Sub-queries Featurisation

In some work in the literature, query feature representation tries to capture its semantics [12] and the database structure [15]. However, in a multi-cloud environment, the schema cannot be assumed to be static: users may add new data sources and source DBs may evolve. Such changes would involve an engineering overhead, since an history should be maintained and the models unpredictably retrained. This approach would practically need to use way larger feature vectors in the cost model, thus requiring more examples to be trained on in order to achieve good accuracy. Notwithstanding the additional costs incurred by the imprecision of the model over a long period, it would ultimately be rendered useless but for workloads with a large amount of queries. As a consequence, we suggest to featurise the queries by encoding the metadata of their execution plans.

Multi-cloud queries are decomposed into a set of clauses C that are ultimately combined to produce sub-queries. For each sub-query q, a tender can be asked to the provider, from which

Feature	Description
x_0	Estimated output cardinality
x_1	Estimated response time
x_2	Join count
x_3	Output tuple size
x_4	Execution plan depth
x_5	Sum of the size of the input relations

TABLE I: Features of the input vector X_q extracted from provider-generated tenders and execution plans

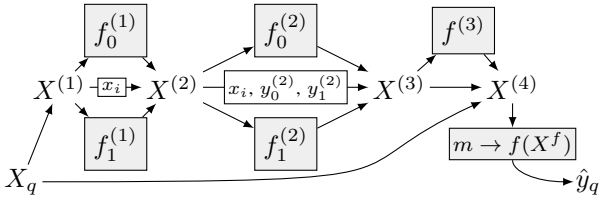

Fig. 1: Ensemble learning stack $\mathbf{M}$.

a feature vector $X_q = [x_0, \ldots, x_5]$ detailed in Table I can be derived. Those features directly encode the amount of data handled by the query (x_0, x_3, x_5) as well as its computational complexity (x_2). Optimisation choices from the provider are also indirectly encoded (x_1, x_4).

B. Ensemble Learning Stack

Our model $\mathbf{M} : \mathbb{R}^6 \mapsto \mathbb{R}^*$ is depicted in Figure 1, and further explained hereafter. Architecture choices were made empirically. It takes as an input a feature vector X_q. It is noted $\mathbf{M}^{(SE)}$ and returns, for a given query q, $\hat{y}_q^{(SE)}$ when instantiated for output cardinality estimation and $\mathbf{M}^{(RT)}$ returning $\hat{y}_q^{(RT)}$ for response time estimation. In the following, x_i should be read as x_0 for $\mathbf{M}^{(SE)}$ and as x_1 for $\mathbf{M}^{(RT)}$.

$\mathbf{M}$ is a multilayer stacking ensemble regression model. Its first layer consists of a linear regression $f_0^{(1)} : \mathbb{R}^3 \mapsto \mathbb{R}^*$ returning $y_0^{(1)}$ and a k-neighbours regressor $f_1^{(1)} : \mathbb{R}^3 \mapsto \mathbb{R}^*$ returning $y_1^{(1)}$, both taking as an input $X^{(1)} = [x_0, x_1, x_2]$. The second layer is a blending layer, whose models takes as an input $X^{(2)} = [x_i, y_0^{(1)}, y_1^{(1)}]$. It consists of a function $f_0^{(2)} : (\mathbb{R}^*)^3 \mapsto \mathbb{R}^*$ returning the average estimate $y_0^{(2)}$ from the first layer and the provider's, and of a k-neighbours regressor $f_1^{(2)} : (\mathbb{R}^*)^3 \mapsto \mathbb{R}^*$ returning $y_1^{(2)}$. The third layer consists of a function $f^{(3)} : (\mathbb{R}^*)^5 \mapsto \mathbb{R}^*$ taking as an input $X^{(3)} = [x_i, y_0^{(1)}, y_1^{(1)}, y_0^{(2)}, y_1^{(2)}]$ and returning $y^{(3)}$ the average value of all the previous estimates.

Finally, $\mathbf{M}$ has a meta model that consists of a Hoeffding tree classifier (i.e. a decision tree that needs relatively small samples to split its nodes) $m : \mathbb{R}^{11} \mapsto \{f^{(0)}, f_0^{(1)}, f_1^{(1)}, f_0^{(2)}, f_1^{(2)}, f^{(3)}\}$ ($f^{(0)}$ being the provider's cost model returning x_i), that takes as an input $X^{(4)} = X_q \parallel [y_0^{(1)}, y_1^{(1)}, y_0^{(2)}, y_1^{(2)}, y^{(3)}]$ and determines the model f that should produce the best estimation. The estimation of f is then returned by $\mathbf{M}^{(SE)}$ as $\hat{y}_q^{(SE)}$ or $\mathbf{M}^{(RT)}$ as $\hat{y}_q^{(RT)}$.

Since early-life estimations of the cost model have poor generalisation capabilities due to the small amount of data it had been trained on, we suggest to ignore its results while the number of ground-truth data points is under a threshold. In this case, the providers' estimate x_i is returned instead.

The recursive nature of execution plans is leveraged to gather more query execution data by learning not only from the whole sub-query but also from its intermediate results.

C. Final Sub-Query Costs Estimation

The monetary costs of querying q on a provider P_i can be calculated as follows. Let $\epsilon_{P_i}^{(Q)}$, $\epsilon_{P_i}^{(E)}$ and $\epsilon_{P_i}^{(S)}$ respectively be the querying, export and storage billing factors in euro per gigabyte of provider P_i. The querying cost of q is defined as $M_{P_i}^{(Q)}(q) = \epsilon_{P_i}^{(Q)} \times x_5^{(q)}$; the storage cost of its intermediate results as $M_{P_i}^{(S)}(q) = \epsilon_{P_i}^{(S)} \times \hat{y}_q^{(SE)}$ and the export cost of those results to a provider P_j as $M^{(E)}(q, P_i, P_j) = \hat{y}_q^{(SE)} \times (\epsilon_{P_i}^{(E)} + \epsilon_{P_j}^{(S)})$ (if $P_i \neq P_j$, elsewhere trivially $M^{(E)}(q, P_i, P_j) = 0$).

The cost model is also able to estimate network transfer times. Let $\mathbf{N}$ be a matrix where each element is a linear regression $g_{ij} : \mathbb{R}_+ \mapsto \mathbb{R}_+$ defined as $g_{ij}(x) = a_{ij}x + b_{ij}$ that predicts the transfer time of a given amount of data from provider i to provider j. All those functions are first initialised with a constant value for a (in GB/seconds) extracted from P_i's and P_j's documentation, with $b = 0$, and are updated whenever an inter-provider data transfer occurs.

IV. Efficient Search Space Exploration

The query optimiser uses the estimates produced by the aforementioned cost model to quote the multi-cloud queries and to orchestrate their outsourcing to the DBaaS providers. It computes two quotations for each multi-cloud query, one minimising the response time and the other the monetary cost. If the user agrees with a quotation, the execution of the multi-cloud query proceed. The optimisation process relies on the randomised search strategy described hereafter.

A. Multi-Cloud Queries Representation

In the following, queries are assumed to be of Select-Project-Join (SPJ) type of acyclic class (i.e. tree queries or chained queries). Since rewriting cyclic queries into equivalent acyclic ones is a NP-hard problem by its own which received attention in the context of distributed query processing [16], this topic is out of scope of this paper.

Figure 2 exemplifies the multi-cloud query execution plan generation process. Let C be the set of clauses of a multi-cloud query Q. A clause $c \in C$ is defined as either a conjunction of selection predicates sharing the same input relation set $\mathcal{R}_c$ or a scan of an input relation. Each relation $R \in \mathcal{R}_c$ is hosted on a provider P_R. Dependency relationship between the clauses is modelled with a graph $D_Q = <V, E>$, where each vertex $v \in V$ has a counterpart in C, and where edge $e \in E$ (with $e = <s, t>$) represents a dependency between the clauses.

Within D_Q, all connected sub-graphs whose clauses' provider set are the same are merged into D'_Q. Each vertex $v \in V$ has therefore a clause set C_v that can be seen as the selections of a maximal sub-query from Q.

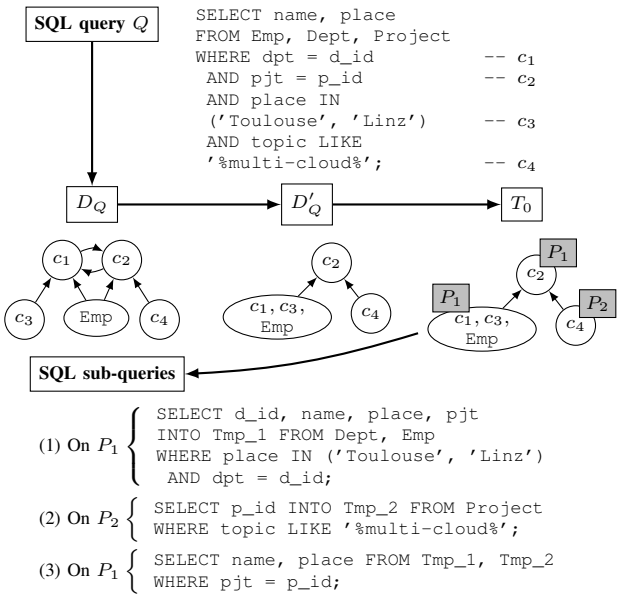

Fig. 2: Generating an execution plan T_0 from a multi-cloud SQL query, assuming relations `Emp(eId, name, #dpt, #pjt)` and `Dept(d_id, place)` hosted on a provider P_1 and `Project(p_id, topic)` on P_2.

A directed spanning tree T_0 is randomly extracted from D'_Q, representing a multi-cloud execution plan. In order to produce a complete execution plan, projection clauses are added to all the vertices so that their subsequent sub-queries can be properly executed later and the projection clause of the multi-cloud query can be satisfied. Then, sub-queries are randomly affected to one of the providers storing the input relations of its clauses. SQL code can be generated from each sub-query.

B. Neighbour Plans Generation

The randomly generated execution plan T_0 is not likely to be optimal, may it be w.r.t. monetary cost or response time. Its purpose is to serve as a starting point for the optimisation process. Therefore, we propose to explore alternatives by deriving several other plans using the transformation rules.

The first rule ϱ_1 enables the exploration of different placement schemes over the providers. Namely, for each vertex $v \in V$, placed on a provider P_v, whose involved provider set is noted as $\mathcal{P}_v = \bigcup_{c \in C_v} \bigcup_{R \in \mathcal{R}_c} P_R$, a new plan is generated with a different P_v for each provider in $\mathcal{P}_v - \{P_v\}$. This rule may discover execution plans with smaller inter-provider data transfers as compared to the deriving plan while preserving the distributed processing of the multi-cloud query.

The second rule ϱ_2 is akin to considering a different join order. Namely, for each of the pairwise successions of vertices containing inter-provider clauses, an execution plan is generated where v_1 and v_2 are swapped. This rule may discover execution plans handling a smaller total amount of data as compared to the derived plan.

The third rule ϱ_3 considers a compromise between distributed parallelism and locality by merging sub-queries hosted on the same provider. Namely, for each couple of successive vertices placed on the same provider, a new execution plan is generated where v_1 and v_2 are merged into a vertex v' having a clause set $C_{v'} = C_{v_1} \cup C_{v_2}$. This rule may discover execution plans which are less storage-intensive and which read the same tuples fewer times.

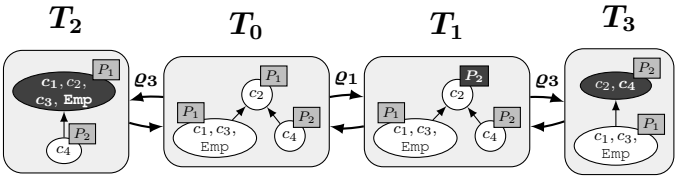

Fig. 3: Search space graph generated from execution plan T_0 for the example depicted in Figure 2. Changes applied by neighbour generation rules ϱ_i are darkened.

C. Exploration Strategy

Our search space exploration strategy is inspired by iterative improvement techniques [13]. The idea is to take a random execution plan as an input and navigate towards a locally-optimal plan by progressively changing it. So as to increase the chances of finding the global minimum without having to perform the NP-hard task of exploring the entire search space, the exploration can be further pursued after the discovery on a local minimum. In order to bound the algorithm to prevent unwanted wandering, a maximal number of improvements towards a minimum n is defined as well as a number of steps beyond the local minimum p. Figure 3 illustrates the procedure, assuming $n = 2$ and $p = 0$: T_0 is the initial execution plan. At step 1, T_1 is generated using ϱ_1 and T_2 using ϱ_3. Assuming T_1 was a local optimum, T_3 is generated at step 2 using ϱ_3. Assuming T_1 is more optimal than T_3, the search for a local optimum stops since n was reached and no step further the local minimum was defined, and T_1 is returned.

The strategy progressively explores the search space by generating, using aforementioned rules ϱ_1, ϱ_2 and ϱ_3, neighbourhood plans set V' of the current plan T and picking the best plan among $V' + T$ w.r.t. ω, until it finds a local minimum. Once a local minimum is found, the process may restart using an execution plan from $V \setminus H$ as a new starting point. In order to compute the quotations, we define several objective functions : (i) the response time $\omega^{(RT)}(T)$, i.e. its maximal sum of the response time among all the leaf-to-root paths, (ii) the monetary cost $\omega^{(M)}(T)$, i.e. the sum of the export, querying and storage costs, and (iii) the monetary cost over response time ratio $\omega^{(RT/M)}(T)$.

V. Experiments and Discussion

In this section, we describe the experiments we carried on in order to validate our cost model and our execution plan search strategy. We compare our results with the literature and discuss the perspectives they draw.

Name	Model	Hyperparameter
$f_0^{(1)}$	`LinearRegressor`	`optimizer`: `AdaDelta` `loss`: `Cauchy` ; `l2`: 2.95
$f_1^{(1)}$	`KNNRegressor`	`n_neighbors`: 6 ; `p`: 1 `aggregation_method`: `weighted_mean`
$f_1^{(2)}$	`KNNRegressor`	`n_neighbors`: 3 ; `p`: 1 `aggregation_method`: `weighted_mean`
m	`HoeffdingTreeClassifier`	`grace_period`: 20 `split_confidence`: 0.55 `split_criterion`: `gini`

TABLE II: Non-default hyperparameters of $\mathbf{M}$

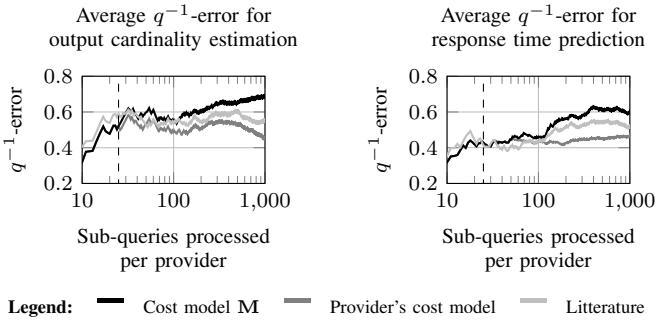

Fig. 4: Evolution of the average q^{-1}-error during the experiments of our cost model $\mathbf{M}$ compared with the providers' DBMS and the models from the litterature.

A. Experimental Setting

In order to evaluate our proposal, we used the Join Order Benchmark (JOB) [4]. Experiments were carried out using the testbed provided by the French national grid computing platform Grid'5000. Comparisons for response time estimation, output cardinality estimation and multi-cloud querying middlewares are provided.

Our middleware is implemented as a Python `flask` API. SQL queries formulated over the multi-cloud schema are compiled with the `sly` library. The multi-cloud cost model described in Section III is implemented using the River Python library [17]. Table II lists the non-default hyper-parameters used for our models, that we determined empirically. Model m's split confidence is intentionally set up on a fairly high value so as to force the decision tree to split quickly whenever a new class is present in the ground-truth; the aim is to counter its early poor accuracy stemming from the cold-start. All graph-related aspects are handled using `networkx` [18]. Clause dependency graphs, execution plans and the space search graph are all implemented as `DiGraphs`. Distributed remote execution of sub-queries over the providers is handled by Python's threads.

During our experiments, cloud providers were simulated. They are implemented as a Python `flask` API, and use PostgreSQL as a back-end DBMS. Each had its own copy of the IMDb database, which were considered independent in order to reproduce the involvement of a variable number of providers for each query in the JOB. As many as 16 simulated providers were deployed, each on a node on a share-nothing cluster, spread on the 8 sites of the Grid'5000 testbed. For a given query, they estimate the response time as the sum of the cost of all its operators divided by 225[1]. They compute the monetary costs as described in Section III-C, with a cost ratio $\epsilon_{P_i}^{(Q)} = 1.75$ ct/GB for querying, $\epsilon_{P_i}^{(E)} = 8.5$ ct/GB for data export and $\epsilon_{P_i}^{(S)} = 3.5$ ct/GB for storage.

In order to assess estimation accuracy, we used the q-error [19] defined in (1), with y an estimation, y' the actual value and $\gamma = 10^{-4}$ (used to not divide by zero).

$$q(y, y') = max\left(\frac{y+\gamma}{y'+\gamma}, \frac{y'+\gamma}{y+\gamma}\right) \quad (1)$$

We compare our cost model with an online implementation of a k-NN regression [11]. Following its introductory paper, we considered parameters k=5 and as a sole feature the DBMS cost returned by the simulated providers.

We compared the response time and monetary cost of the execution plans as well as the optimisation costs of our current proposal with the previous middleware [3]. They were both deployed and tested in similar conditions over the testbed, and underwent the JOB in the same order. We defined a timeout of 10 min for quotation computation and of 20 min for query execution.

B. Results and Discussion

Experimental results are presented and analysed hereafter. We first show that our cost model does manage to improve the accuracy of the estimations the optimiser works on. Then, we show that the randomised search strategy does manage to quote complex queries in a timely manner. Finally, we show that the neighbour generation rules lead to the discovery of more efficient execution plans.

Figure 4 compares the accuracy of our cost model $\mathbf{M}$ with the providers' DBMS's (later referred to as the baseline) and the k-NN regressor from the literature by depicting their average q^{-1}-error (instead of the q-error for readability reasons) as the experiments progress. Once the metamodel m is allowed to make its own choices starting at the 25th query it learned on, it can be noticed that for both metrics the cost model manages to be more accurate than the providers' DBMS', to match and then outreach the accuracy of the cost model from the literature. The difference between our model and the literature stem from their imbrication and show the efficiency of the metamodel: the model in the literature has actually the same nature as f_1^1 and f_1^2 in our stack, hence their accuracy is partially leveraged by m when it chooses the estimation of a model from the stack.

As depicted by Figure 5 and as expected by the theory, the randomised search space exploration strategy does allow

[1]This number had been determined empirically

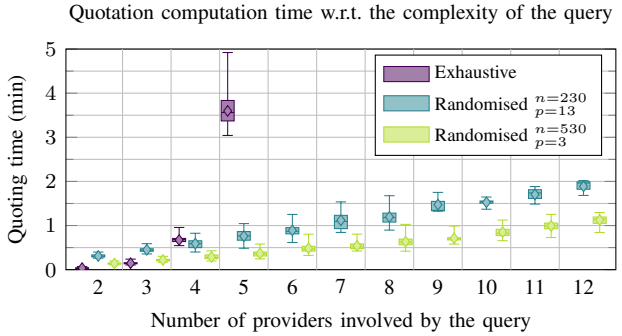

Fig. 5: Comparison of the time spent to compute the quotations for a multi-cloud query w.r.t. its complexity by both an exhaustive strategy and the randomised strategy.

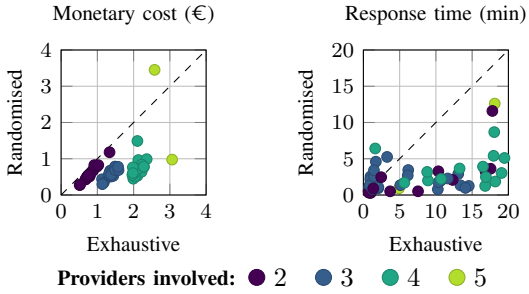

Fig. 6: Comparison of the monetary cost and the response time of the queries yielded using the exhaustive search method and the randomised one.

to compute quotations for complex queries involving many providers. In contrast, the exhaustive method timed-out when there are more than 5 providers. With regard to the randomised strategy, several bounding parameters were tested, showing that the performances of the strategy are proportional to the number of exploration steps looking beyond the local minimum and the number of providers involved by the query, regardless of the actual amount of execution plans estimated.

Finally, the comparison between execution results yielded by the exhaustive space search exploration method with the randomised one, as depicted in Figure 6, shows the latter is way more efficient than the former, managing to increasingly reduce the monetary costs w.r.t. the complexity of the query and generally finding execution plans with a better response time. This can be explained because the randomised method does not limit its exploration to linear execution plans, hence avoiding costly and slow inter-cloud data transfers and therefore reducing the idle time between the sub-queries.

VI. Conclusion

In this paper, we introduced a multi-cloud learned cost-model in order to improve the reliability of both the multi-cloud queries quotation and their optimisation process. It uses online machine learning models. We also introduced a randomised space exploration strategy to quickly optimise the queries. Experimental results showed that the cost model manages to correct the providers' DBMS' estimates, and that the randomised search strategy is able to produce execution plans that are less expensive to execute for better performances.

As experiments using simulated providers showed our approach is promising, there is now a case to transform what is currently a proof-of-concept into an actual middleware supporting a real-world setting, with real providers, real users and actual use cases.

Acknowledgements

This work has been funded by (i) the LabEx CIMI through the MCD project and (ii) the French Ministries of Europe and Foreign Affairs and of Higher Education, Research and Inovation through the EFES project (No 44086TD), Amadeus program 2020 (French-Austrian PHC). We also thank the International Cooperation & Mobility (ICM) of the Austrian Agency for Education and Internationalisation (OeAD-GmbH). The Grid'5000 testbed is supported by Inria, CNRS, RENATER and several Universities as well as other organisations.

References

[1] D. Abadi, A. Ailamaki *et al.*, "The Seattle Report on Database Research," *SIGMOD Record*, vol. 48, no. 4, pp. 44–53, Feb. 2020.
[2] S. Imai, S. Patterson, and C. A. Varela, "Cost-Efficient High-Performance Internet-Scale Data Analytics over Multi-cloud Environments," in *CCGrid*. IEEE & ACM, May 2015, pp. 793–796.
[3] D. T. Wojtowicz, S. Yin *et al.*, "Cost-Effective Dynamic Optimisation for Multi-Cloud Queries," in *CLOUD*. Chicago (Online), USA: IEEE, Sep. 2021, pp. 387–397.
[4] V. Leis, A. Gubichev *et al.*, "How good are query optimizers, really?" *Proc. VLDB Endow.*, vol. 9, no. 3, pp. 204–215, Nov. 2015.
[5] T. Ibaraki and T. Kameda, "On the optimal nesting order for computing *N*-relational joins," *TODS*, vol. 9, no. 3, pp. 482–502, Sep. 1984.
[6] M. Halford, P. Saint-Pierre, and F. Morvan, "Selectivity estimation with attribute value dependencies using linked bayesian networks," *TLDKS*, no. XLVI, pp. 154–188, 2020.
[7] L. Bottou and Y. Le Cun, "Large Scale Online Learning," in *NIPS*, vol. 16. Vancouver, BC, Canada: MIT Press, 2003, pp. 217 – 224.
[8] M. Stillger, G. Lohman *et al.*, "LEO – DB2's LEarning Optimizer," *Proc. VLDB Endow.*, vol. 1, pp. 19–28, Sep. 2001.
[9] C. Wu, A. Jindal *et al.*, "Towards a learning optimizer for shared clouds," *Proc. VLDB Endow.*, vol. 12, no. 3, pp. 210–222, Nov. 2018.
[10] M. Halford, P. Saint-Pierre, and F. Morvan, "Selectivity correction with online machine learning," in *BDA*, Paris (Online), France, Sep. 2020.
[11] A. Kleerekoper, J. Navaridas, and M. Lujan, "Can the Optimizer Cost be Used to Predict Query Execution Times?" *arXiv:1905.00774 [cs]*, May 2019.
[12] R. Marcus, P. Negi *et al.*, "Neo: A Learned Query Optimizer," *Proc. VLDB Endow.*, vol. 12, no. 11, pp. 1705–1718, Jul. 2019.
[13] A. Swami, "Optimization of large join queries: combining heuristics and combinatorial techniques," in *SIGMOD*. New York, NY, USA: ACM, 1989, pp. 367–376.
[14] D. H. Wolpert, "Stacked generalization," *Neural Networks*, vol. 5, no. 2, pp. 241–259, Jan. 1992.
[15] J. Ortiz, M. Balazinska *et al.*, "Learning State Representations for Query Optimization with Deep Reinforcement Learning," in *DEEM*. Houston, TX, USA: ACM, Jun. 2018, p. 4.
[16] Y. Kambayashi, M. Yoshikawa, and S. Yajima, "Query processing for distributed databases using generalized semi-joins," in *SIGMOD*. New York, NY, USA: ACM, 1982, pp. 151–160.
[17] J. Montiel, M. Halford *et al.*, "River: machine learning for streaming data in Python," *JMLR*, vol. 22, no. 110, pp. 1–8, 2021.
[18] A. Hagberg, P. Swart, and D. S Chult, "Exploring network structure, dynamics, and function using networkx," LANL, Los Alamos, USA, Tech. Rep. LA-UR-08-05495, 2008.
[19] G. Moerkotte, T. Neumann, and G. Steidl, "Preventing bad plans by bounding the impact of cardinality estimation errors," *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 982–993, Aug. 2009.