Cost-Effective Dynamic Optimisation for Multi-Cloud Queries

Damien T Wojtowicz, Shaoyi Yin, Franck Morvan, Abdelkader Hameurlain

Cost-Effective Dynamic Optimisation for Multi-Cloud Queries

Damien T. Wojtowicz, Shaoyi Yin, Franck Morvan and Abdelkader Hameurlain
IRIT Laboratory
Paul Sabatier University, Toulouse, France
{damien.wojtowicz,yin,morvan,hameurlain}@irit.fr

***Abstract*—The provision of public data through various Database-as-a-Service (DBaaS) providers has recently emerged as a significant trend, backed by major organisations. This paper introduces Nebula, a non-profit middleware providing multi-cloud querying capabilities by fully outsourcing its users' queries to the involved DBaaS providers. First, we propose a quoting procedure for those queries, whose need stems from the pay-per-query policy of the providers. Those quotations contain monetary cost and response time estimations, and are computed using provider-generated tenders. Then, we present an agent-based dynamic optimisation engine that orchestrates the outsourced execution of the queries. Agents within this engine cooperate in order to meet the quoted values. We evaluated Nebula over simulated providers by using the Join Order Benchmark (JOB). Experimental results showed Nebula's approach is, in most cases, more competitive in terms of monetary cost and response time than existing work in the multi-cloud DBMS literature.**

***Index Terms*—Multi-Cloud, Database-as-a-Service, Dynamic Query Optimisation, Service-Level Agreement, Middleware**

I. Introduction

Public availability of scientific data, may it be raw observational records or derived products, is central to scientific research. In areas such as astronomy, biology or Earth science, they form the bedrock of research endeavours aiming at processing and crossing those data in order to unveil new knowledge. Various data sharing methods co-exist. While file provisioning is still prevalent and historically ingrained, there is a trending shift to the cloud[1] backed by major organisations such as the European Space Agency [1] or the United States' National Oceanic and Atmospheric Administration [2]. Leveraging the Database-as-a-Service (DBaaS) model is also an option. In this case, access rights to cloud-hosted databases are granted to the public[2] and, for a fee, users – often called 'tenants' in this context – are offered the opportunity to formulate queries using the provider's services (a multi-tenant DBMS). Alternatively, users can download the wanted datasets and run queries on self-administrated databases, implying they have both sufficient skills and resources.

This growing availability of data in the cloud raises the issue of their cross-analysis and by extension the problem of data integration when sources are hosted on various providers' infrastructure. This is part of the broader topic of multi-cloud data management, first mentioned about a decade ago [3], which is now acknowledged as an area of importance to both academia and the industry[3] for the foreseeable future [4]. Several multi-cloud DBMSs have been developed so far, typically as components of a broader service (e.g. a collaborative document editing tool [5]). Rather than leveraging DBaaS, they spawn instances of their system over many virtual machines hosted on a variety of Infrastructure-as-a-Service (IaaS) providers. They are designed to support various data models, may they be associative arrays [5]–[7], relational [8] or heterogeneous [9], [10]. Some are also designed to leverage several cloud file storage services [11]–[13]. Most of these works focus on data placement and replication policies so as to offer security and availability guarantees. Multi-cloud aspect to querying is reduced to the choice of one provider perceived as the best fit to execute the query with respect to a given objective (e.g. monetary cost or performance).

With the aim of fostering the analysis of cloud-hosted relational public data by using the infrastructure of their DBaaS providers, akin to Analytics-as-a-Service [14] but without the pursuit of profit, this paper introduces Nebula, a system offering multi-cloud querying capabilities. We adapt the traditionally vertical mediator-wrapper [15] architecture (which recently evolved towards the polystore systems [16]) in the following way: while the interactions with the users and the overall control flow remain centralised, the execution is fully outsourced to the cloud providers on a pay-per-use basis. Hence, Nebula responds to two key challenges stemming from this context: (i) query pricing in order to define a quotation for a multi-cloud query and (ii) multi-cloud query optimisation.

Quotations are a necessity given the monetary implications of cloud computing and the variety of billing models. Indeed, while some providers, such as Google[4], charge per query input relations size, more complex policies defined into Service-Level Agreements (SLA) may exist [17] (e.g. based on the complexity of the query or the computational resources used during its processing). We propose a quoting method that abstracts the aforementioned pricing policy heterogeneity. It consists of decomposing the queries into a directed acyclic graph (DAG) of sub-queries that will be estimated by their

[1]See examples at Amazon (https://registry.opendata.aws/) or Google (https://cloud.google.com/public-datasets).

[2]Such datasets are available on Google's BigQuery. (https://cloud.google.com/bigquery/public-data).

[3]Integrated solutions facilitating inter-provider data exchange, such as Google's Omni, are emerging at this very moment. https://cloud.google.com/blog/products/data-analytics/introducing-bigquery-omni

[4]See BigQuery's fares at https://cloud.google.com/bigquery/pricing.

target providers. These estimates are further aggregated and corrected in order to present a price and a response time to the user, with a given error margin. Nebula acts as a broker between the providers and its users, who will be directly billed. The error margin is important to build a trustworthy reputation because it relaxes users' accuracy expectations while being reassuring.

Our system also needs to optimise the multi-cloud queries. Thanks to cloud computing capabilities, host databases are seen as black-boxes. Thus, Nebula solely needs to solve a derived version of the join order problem by minimising the overall monetary cost of the multi-cloud query while keeping in mind the estimated response time from the quotation. In distributed systems, dynamic optimisation engines, especially agent-based ones, are known to perform better than traditional, optimise-then-execute engines thanks to their ability to proactively compensate sub-optimalities at execution time [18]. Their decentralised nature is not only philosophically consistent with the way data is stored, but is also an asset in terms of performance since re-optimisation is performed by agents that solely consider their potential future load. Therefore, Nebula's multi-cloud query optimisation engine uses agents that cooperate in order to progressively orchestrate sub-queries execution and inter-provider data transfers in order to meet the quotation.

This paper is organised as follows. In Section II, we provide a review of the literature about relational multi-cloud DBMSs as well as the related topics of personalised SLA generation and dynamic query optimisation. We then proceed with the description of Nebula's architecture (Section III) and its two core components: its query quotation calculation method (Section IV) and its agent-based dynamic optimisation engine (Section V). Experiments are explained and their results discussed in Section VI. Finally, we conclude in Section VII.

II. Related Work

Work on multi-cloud data management led to the development of several DBMSs. In this section, we provide a brief survey of those designed for relational databases. Moreover, since personalised SLA generation in the context of database queries shares similarities with the idea of quotation, we briefly survey the literature on this topic. We also touch upon dynamic query optimisation within distributed systems, which are essentially similar to a multi-cloud environment notwithstanding monetary matters.

A. Relational Multi-Cloud DBMSs

Relational multi-cloud DBMSs are federations of instances of these systems deployed on virtual machines hosted by various IaaS providers. To our knowledge, there are two systems capable of managing relational databases.

SCOPE [19] is originally designed as a column store, and had been extended to support data model heterogeneity [10]. SCOPE serves as the basis for a SaaS application, tied by SLA to its tenants. It is usually hosted by a single provider, and its multi-cloud capabilities act as a spillway by outsourcing the excess workload. Data and task placement choices are multi-objective: they take into account performance and monetary cost.

The SHAMC [8] system has been designed as a relational DBMS that focuses on system availability in the context of cloud-hosted, encrypted databases. It relies on homomorphic encryption, which led to the development of algorithms able to perform relational algebra operations on encrypted data. Data is fully replicated on all servers in order to (i) dispatch user queries evenly across all the available databases, and (ii) to avoid costly inter-provider communications.

None of those systems tackle the multi-cloud query optimisation problem: queries are dispatched to be fully executed by a single instance of the systems. The goal is to optimise the system as a whole rather than each single query. This stance stems from the IaaS nature of the cloud resources involved in both systems.

B. Personalised SLA for DB Queries

The idea behind personalised SLA in the context of DBaaS is that queries or batches of queries should be priced according to their complexity as well as user-defined constraints. As an example, *ceteris paribus*, tenants would pay less for less complex queries, and providers would not put their profits at jeopardy when dealing with more complex ones. Although previous work considered SLA generation for a specific querying workload, these differ from our proposal in that (i) they are designed for a cloud provider or services hosted on a single cloud (ii) they are not designed to generate SLAs for a single query and (iii) their scope is broader than Nabula's quotations. Since this topic emerged lately, its literature is quite sparse.

The first method [20] has been designed for the deployment on the cloud of the Myria polystore. It is tailored to facilitate a choice by the users between several deployment configurations of Myria called *tiers* that will ultimately use a single-cloud IaaS environment. Those tiers have a price and are showcased with a a workload example and its corresponding predicted response time.

The second method [17] generates SLA using query templates according to a user-defined expected query completion time and a tolerance threshold. The idea is that users are fine with waiting a bit more for the query to be executed, granted that they will pay less, hence a price function is defined according to the users expectations. The base price value of the function is determined using selectivity intervals.

These two approaches are designed to form the basis of billing policies. Therefore, they do not seem appropriate in a multi-cloud environment, since a multi-cloud middleware would not be able to control its expanses and performances in the same way a cloud provider does.

C. Dynamic Query Optimisation

Within a DBMS, a query is compiled into an execution plan in the form of a relational algebra tree. Its performance are correlated to the amount of input data of each operator. Hence, an optimiser is involved in the compilation process,

aiming at finding an optimal plan – mostly given estimations of its operators' output cardinality (i.e., number of tuples) made by a cost model. The latter are often inaccurate [21], hence sub-optimal plans are generated. When executed, they entail poor performances and – in the context of cloud computing – avoidable monetary costs.

Dynamic query optimisation consists of progressively executing a query and re-optimising its execution plan using information gathered at execution [22] so as to mitigate the effects of estimation errors. Dynamic query optimisation models fall into two categories. They are either centralised, i.e. carried out by a single process, or decentralised, i.e. carried out by a set of cooperating processes that may be executed on various machines. Re-optimisation can be performed at different levels of query processing.

Techniques applied at the inter-operator level have been developed. Their are of interest in a multi-cloud environment because of the black-box approach to the providers. Two kinds of approaches can be identified.

First, query scrambling [23] consists of rescheduling or recomposing pipelines in a query's execution plan when an operator fails to deliver on time. This technique was designed with the aim of reducing waiting times in the execution plan, which are a lesser issue in a multi-cloud environment, granted that idling-induced monetary losses – storage fees – are low.

Second, turning operators of the execution plan into mobile agents across a distributed DBMSs' sites [18] can lead to a cooperative execution of the query. By sharing intelligence about intermediate results they produced, agents trigger re-estimations of the remaining operators by their counterparts. The latter may take new migration decisions, which are also propagated and may dramatically change the execution plan.

Other techniques, such as tuple routing [24], [25], time slicing [26], query decomposition [27], mobile agent-based joins [28] or operator inflation [29], are either applied at tuple-level or at intra-operator level. These strategies are unfortunately not tailored to multi-cloud query optimisation. Indeed, the levels at which they operate are the providers' preserve because of the black-box approach induced by DBaaS.

We take inspiration from the multi-agent method: by incorporating monetary costs and changing the objective function, agents should be able to meet response time and monetary cost targets.

III. Overall System Design

Nebula is a middleware that centralises the users interactions as well as the orchestration of the operations on the providers. This centralisation is explained by the DBaaS model assumed for the providers: we suppose they are solely able to execute queries and administrative DBMS commands. As a side note, there is no need for the data nor the provider to be public: a private cloud can also be incorporated to the sources pool.

Fig. 1 depicts the overall architecture of Nebula, as well as the information flows it hosts. The multi-cloud schema aggregates the public schema from the data sources. Namely,

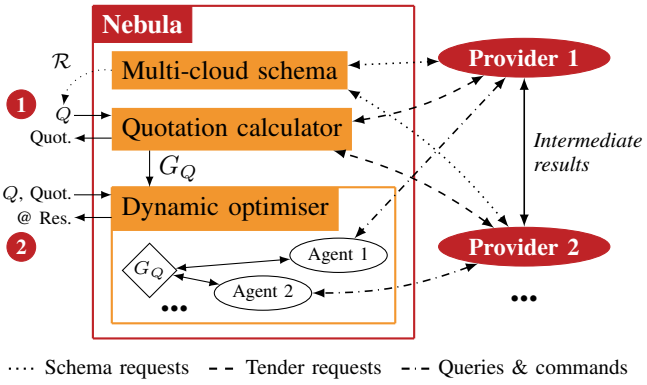


Figure 1. Overall design of the Nebula system.

it is the set $\mathcal{R}$ of couples $<R, P>$ linking a relation R with its host provider P.

The quotation calculator generates, for a given multi-cloud query Q written in SQL, several quotations modelled as a couple $<\mathcal{M}, \delta>$ with $\mathcal{M}$ a monetary cost and δ an estimated response time. This quotation is computed by decomposing Q into a directed acyclic graph G_Q whose vertices model sub-queries affected to a provider that will be asked to produce a tender for the sub-queries. Those are aggregated to compute the quotations for Q. If the user agrees to one of those quotations, then Nebula may proceed with Q's execution.

The execution engine uses agents that communicate with each other using the query's graph G_Q generated by the quotation calculator and send queries and commands to the providers. Those agents seek to minimise the monetary cost of the multi-cloud query, while still keeping in mind the quoted response time. Therefore, the performance constraints usually applied to optimisers are relaxed: the system does not need to seek the cheapest nor the fastest execution plan, but rather the one that yields a monetary cost and a response time closer to the quotation.

IV. Quoting Procedure and Billing Policy

Nebula's quoting procedure consists of three steps: (i) query decomposition into sub-queries, (ii) sub-queries costing using provider-generated tenders and (iii) identifying several possible quotes. The general idea of this procedure had been briefly outlined in previous paper [30]. It is presented more thoroughly hereafter and extended by introducing the calculation of several quotes for a query as well as their correction.

A. Query Decomposition

Inspired by historical query decomposition techniques [27] and recent work on multi-objective query optimisation applied to IaaS cloud environments [31], our quoting procedure starts by decomposing a multi-cloud query Q into a *directed acyclic graph* (DAG) $G_Q = <V, E>$: each vertex $v \in V$ models a sub-query involving a maximal amount of clauses for a given combination of providers; each directed edge $e \in E$ represents dependencies between sub-queries.

All queries, may they be multi-cloud queries or sub-queries within a DAG, are assumed to be of Selection-Projection-Join type. They are seen as a set of clauses C, and each clause $c \in C$ is a tuple $<\varphi, t, A, P>$ where φ is a predicate involving the set of attributes A from relations hosted by providers from the set P. The clauses are of type $t \in \{\Pi, \sigma\}$, Π standing for projection (i.e. SQL's `SELECT` predicates) and σ for selection (i.e. SQL's `WHERE` predicates). The relational algebra is voluntarily limited to those operators since they are sufficient to generate the SQL code of the sub-queries by deducing the involved relations from A. Moreover, there is no need to formally encode the joins within Nebula because the providers will ultimately make their own plans and choices of algorithms for the execution of the sub-queries.

Algorithm 1 is the procedure that takes the clauses from a multi-cloud query Q and generates its DAG G_Q, with $v \in V = <P, \mathcal{P}, C^{(v)}>$ a vertex, and $Perm(k, X)$ all the k-permutations of a set X. P is a vertex's assigned provider, $\mathcal{P}$ lists all the providers whose data is used in the attributes of the clauses set $C^{(v)}$. It consists of progressively building the DAG layer after layer by (i) grouping clauses by the set of providers storing their input data, (ii) adding projections for the attributes that will be needed to run subsequent sub-queries, (iii) transforming these groups into vertices in the DAG and (iv) linking them together. In order to limit its size, the query graph is generated so that the oriented spanning anti-trees rooted in the DAG's leaves are left or right linear. An example of the input C and output G_Q for a query is provided in Fig. 2. Two types of sub-queries can be identified: single-provider queries (i.e. involving a set of clauses using data from a single provider) and inter-provider queries (involving clauses using data originally hosted on at least two providers).

Algorithm 1: Query graph generation

Input: C the set of clauses from query Q.
Result: The directed acyclic graph $G_Q = <V, E>$

$V, E \leftarrow \emptyset$;
Let M map a set of providers $K = \{c.P \mid c \in C\}$ to a set of clauses as $M : k \rightarrow \{c \mid c \in C, c.P = k\}$;
foreach $k \in K$ *by ascending order of* $|k|$ **do**
  `/* Adding projections required by further sub-queries */`
  $C^{(k)} \leftarrow M(k) \cup \left\{<\text{`}a\text{'}, \Pi, \{a\}, Prov(a)> \mid a \in c.A, \forall c \in M(x)\ \forall x \in K\ :\ k \subset x\ \wedge\ Prov(a) \in k\right\}$;
  `/* Iteration over all permutations of k to add new vertices and edges */`
  foreach $k' \in Perm(|k|, k)$ **do**
    $v \leftarrow <k'_0, k', C^{(k)}>$;
    if $|k| > 1$ **then**
      `/* Linkage with v's predecessors */`
      $B \leftarrow \bigcup_{w \in \{k'_0, k'_{|k|}\}} <w, w'>$ with $w' \in V \wedge w'.\mathcal{P} = k - w$;
      foreach $<a, b> \in B$ **do**
        $v' \leftarrow v$;
        Add v' to V ;
        Add $<a, v'>$ and $<b, v'>$ to E;
    else
      Add v to V ;
$G_Q \leftarrow <V, E>$;

B. Costing the DAG's Components

Next step in quoting computation is sub-queries costing. The latter starts by, for each sub-query SQ_v whose SQL code is generated from $C^{(v)} : v \in V$, asking its affected provider P for a tender. It is typically defined as a triple $<\mathcal{M}, \delta, S>$ with $\mathcal{M}$ a monetary cost, δ a response time and S an estimated output size, but some of those components may be missing because of the nature of the various pricing policies used by the providers. After being computed, values from this triple are added to v's tuple. As a consequence, Nebula estimates the missing ones.

If the price component is missing from the P provider's estimation, it may be because it follows a transparent, *à la* BigQuery, pricing policy where $\mathcal{M}$ is derived directly from the sum of the input relations size. The system fills this gap by multiplying the sum of the size of the sub-query's input relation by the provider's querying price factor $\varepsilon^{(P)}$ in euro per gigabyte. There is also the possibility to use slots in a pre-paid bundle, where n queries have already been paid for a fee m. In this case, the sub-query's price is trivially $\mathcal{M}_v = \frac{m}{n}$.

Another missing component could be the response time δ. Nebula predicts this value by leveraging a 5-neighbours regressor from the literature [32] that takes as a feature the DBMS's optimiser cost. We adapt this model in Nebula in an online-learning fashion using the `river` library [33]. The main interest of online learning is its adjusting capability that strengthens extrapolations of the model thanks to the ingestion of new data points, allowing a cold-start.

The last costing step consists of evaluating inter-provider transfers as well as intermediate results storage costs. Each edge $e = <I, O> \in E$ of the query graph G_Q – with I its input vertex, $I.S$ the estimated size of its intermediate results and O its output – is modelled as a couple $<\mathcal{M}_e, \delta_e>$. The latter is computed as depicted in (1) and (2), with $\varepsilon^{(E)}_{P_v}$ the export cost of a provider P_v (associated to a vertex v) expressed in euro per gigabyte, $\varepsilon^{(S)}_{P_v}$ the storage cost and b_P the network bandwidth of a provider.

$$\mathcal{M}_e = \begin{cases} I.S \times \varepsilon^{(S)}_{P_I} & \text{if } P_I = P_O \\ I.S \times \left(\varepsilon^{(E)}_{P_I} + \varepsilon^{(S)}_{P_O}\right) & \text{otherwise} \end{cases} \quad (1)$$

$$\delta_e = \begin{cases} 0, & \text{if } P_I = P_O \\ I.S \times min(b_{P_I}, b_{P_O}) & \text{otherwise} \end{cases} \quad (2)$$

C. Quotations' Components Computation

When looking at G_Q through the lenses of flow networks, vertices modeling maximal single-provider sub-queries are sources and the set $W \subset V$ of vertices without successors

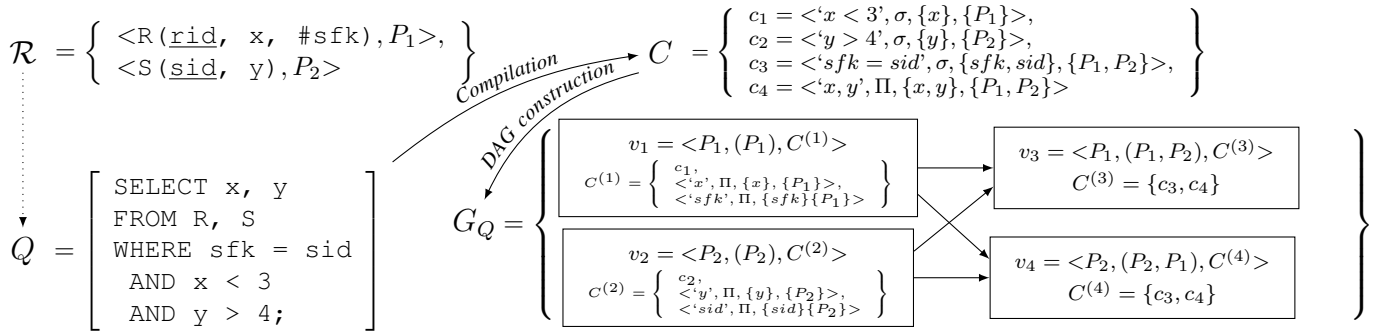


Figure 2. Example of query decomposition for a query Q – formulated over a multi-cloud schema $\mathcal{R}$ – and its set of clauses C into a DAG G_Q. Within G_Q, vertices v_1 and v_2 represent single-provider queries; v_3 and v_4 represent join queries.

(i.e. the last sub-queries to be executed in order to produce Q's results) are sinks. Let $\mathcal{T}_Q$ be the set of oriented spanning anti-trees over G_Q where each $T_w = <V_{T_w}, E_{T_w}> \in \mathcal{T}_Q$ is rooted in a sink vertex $w \in W$. The monetary cost $\mathcal{M}_{T_w}$ of a given T_w is defined as the sum of the costs induced by its sub-queries, data storage and inter-provider transfers. Its response time δ_{T_w} is trivially the δ-maximal source-to-sink path into the tree.

As to let its users make their own compromise between response time and monetary cost, a set of quotations $\mathcal{Q}_Q$ is computed by Nebula. The latter only contains competitive quotes: for a given one, other quotations that are both more expansive and yield a longer response time are not considered. We formally define $\mathcal{Q}_Q$ in (3), with $\lambda \geq 1$ a coefficient that helps eliminate less competitive execution plans with a similar δ. Units are assumed to be cents and milliseconds: when values are small, differences between candidate quotations may be outside a λ range and yet remain insignificantly different with regard to the units, hence the roundings to the nearest integer.

$$\mathcal{Q}_Q = \left\{ \begin{array}{l} <\mathcal{M}_t, \delta_t> \mid t \in \mathcal{T}_Q, \\ \quad \nexists\, t' \in \mathcal{T}_Q : t \neq t' \\ \quad\quad \wedge\ \lfloor \mathcal{M}_{t'} \rceil \leq \lfloor \mathcal{M}_t \rceil \wedge \lfloor \delta_{t'} \rceil < \lfloor \lambda \delta_t \rceil \end{array} \right\} \quad (3)$$

D. Quotation Post-Processing

Ultimately, our quotation is the outcome of calculations based on estimates from the providers' query optimisers. Those are possibly dramatically wrong [21]; they might therefore lead to calculation errors in the aforementioned steps.

The Nebula execution engine (presented in Section V) is inspired by dynamic optimisation techniques, which are known for their ability to reduce the difference between estimations and the reality of an execution plan. Thus, we postulate that there will be a quasi-linear relationship between the quotation and the actual values.

As a consequence, all the prices within $\mathcal{Q}_Q$ are corrected using a linear regression f. The latter takes as an input the quoted monetary cost, the total size of the query's input relations S_t and its estimated response time δ_t. This regression is implemented in the manner of online learning. This approach of machine learning eliminates the need for a large set of training data: linear regressions will be perpetually refined with new queries. Likewise, a 5-neighbours regressor using the same arguments as f is used to post-process the estimated response time of the quotation.

E. Reputation and Billing Policy

Offering strong guarantees over the actual monetary cost and response time of a query would be unrealistic. Indeed, early estimation errors in the quotation computation are snowballed in the later sub-queries'. Moreover, Nebula has virtually no control over the optimisation of the sub-queries since providers act according to their global profit targets or commercial objectives. Hence, all quotations $<\mathcal{M}, \delta> \in \mathcal{Q}_Q$ are in fact used to define the upper bounds of intervals $[0, (1+\Lambda)\mathcal{M}]$ and $[0, (1+\Lambda)\delta]$, with Λ a fixed tolerance error ratio. Those represent the range in which the response time and the monetary cost are expected to be.

Since Nebula is designed as a non-profit broker that connects its users to the cloud providers, not a penny actually passes through our system: providers bill directly our users. As a consequence, compensations cannot be offered when quotations are not met at execution time. Rather than offering strong guarantees that would require an SLA, we rely on reputation by transparently providing the users the past queries history. The administrator-defined Λ ratio helps building the reputation: by choosing a fair value – say 0.1 – users accuracy expectations towards the quotations are toned down yet trust towards Nebula is enabled. When the actual monetary cost or response time or both for a query exceeds the upper bound of the interval, the reputation of the Nebula system will be degraded.

V. Multi-Cloud Dynamic Optimisation

The distributed database literature shows that agent-based, decentralised dynamic execution proved its ability to progressively produce a more efficient execution plan than a traditional optimiser's [18], [28]. The key behind their better performances is their proactivity in avoiding bad operator order within the execution plans by triggering re-estimations using newly gathered knowledge about the intermediate results. Nebula's optimisation engine is inspired by those works.

It relies on an agent-based model (ABM) in which control agents are entrusted with the orchestration of the multi-cloud query execution. Their environment as well as their behaviour is described hereafter.

A. Agent-Based Model Setup

An ABM within Nebula consists of a set of cooperative agents $\mathcal{A}$ deployed to orchestrate the outsourced execution of a multi-cloud query Q given a quotation $<\mathcal{M}_Q, \delta_Q>$. $\mathcal{A}$ is formally defined as $\mathcal{A} = \{A_v \mid v \in V, N_v^+ = \emptyset\}$ with N_v^+ the set of immediate predecessors of a vertex $v \in V$ in G_Q. Namely, an agent is spawned for each vertex associated to a maximal single-provider sub-query. Upon creation, they immediately command the execution of their vertex's sub-query and thus start their cyclic behaviour. As a consequence of this initialisation, an agent is actually affected to a provider.

The environment in which agents operate is the DAG G_Q recycled from the quotation computation process of Q. It is already weighted by estimated monetary costs and response time of (i) each sub-query associated to the vertices and (ii) storage and inter-provider transfers modelled by edges. Each vertex modeling an inter-provider query is also equipped with a two-slots synchronisation barrier that makes its attached execution agent wait for the completion of all the prerequisite to the sub-query execution (e.g. an inter-provider data transfer). Single-provider queries' vertex have an already risen barrier.

B. Agents' Behaviour

Agents within our ABM communicate by altering their environment: estimated $\mathcal{M}_v$ and δ_v are overwritten either with real values or re-estimations when new knowledge is gathered. When an agent $A_v \in \mathcal{A}$ associated to a vertex v chooses a new vertex $v' \in N_v^-$ (N_v^- being the set of immediate successors of a vertex v) to attach to, it communicates its decision removing from G_Q all the vertices now unreachable by the agents.

1) Agents' Goal: During the execution orchestration process, each agent $A_v \in \mathcal{A}$ moves forward into G_Q by choosing a new vertex $v' \in N_v^- \cap V_T$ to attach to, with N_v^- being the set of immediate successors of a vertex v and $T = <V_T, E_T> \in \mathcal{T}_Q$ the tree returned by the function defined in (4). Given a tolerance threshold Λ, it consists of minimising the response time while not exceeding a price tolerance range. If this minimisation appears to be impossible, then agents try to reproduce the quoted ratio between monetary cost and response time in an attempt to minimise both the response time and the monetary cost.

$$BestTree \mapsto \min_{t \in \mathcal{T}_Q} \begin{cases} \delta_t \text{ if } \exists t : \mathcal{M}_t \leq (1+\Lambda)\mathcal{M}_Q \\ \qquad\qquad \wedge\ \delta_t \leq (1+\Lambda)\delta_Q \\ \left| \frac{\delta_t}{\mathcal{M}_t} - \frac{\delta_Q}{\mathcal{M}_Q} \right| \text{ otherwise} \end{cases} \tag{4}$$

2) Environment Updates: Updates within the DAG G_Q can be performed by an agent A_v at two occasions. The first one immediately follows the completion of their sub-queries, by asking the providers to re-estimate all the sub-queries attached to the set $\mathcal{N}_v^-$ of all the reachable vertices from v. Those new estimations are also used to re-compute the monetary cost and duration associated to the relevant G_Q's edges. Note that in the early execution of Q, this step may be long because of the amount of vertices' sub-query to evaluate.

The second case in which an agent modifies the DAG is just after the choice of its next vertex to associate to. When such a decision occurs, it means that a part of the search space was overlooked. It is therefore superfluous, and should be plainly deleted from G_Q. Considering an agent A_v who chooses v', the set of vertices to be deleted $V_\ominus^{(v')}$ contains all the vertices outside the set of trees within $\mathcal{T}_Q$ that do not contain v'.

DAG trimming is how agents communicate their choices: deadlocks induced by diverging quasi-simultaneous decisions are prevented, hence removing the need of a direct inter-agent communication and negotiation protocol. Moreover, it turns out that these alterations reduce the search space to explore for each decision, thus limiting the algorithmic complexity of the agents' behaviour.

3) Agents' Algorithm: Agents are formalised in Algorithm 2, assuming a function $Subquery : C^{(v)} \mapsto SQL$ generating a SQL query from a vertex's v set of clauses $C^{(v)}$. Their cyclic behaviour starts by commanding the execution of the sub-query modelled by their vertex. It follows by updating G_Q by triggering re-estimations for subsequent sub-queries. Then, the agent chooses v' the next vertex to attach to, which puts him on the path towards what is perceived as the best execution plan. An inter-provider transfer is ordered if v's provider is different from v''s. Finally, the synchronisation barrier of v' is notified so that the ABM can move forward with the dynamic optimisation of the multi-cloud query.

Fig. 3 exemplifies how the ABM evolves during the processing of a query. Agents A_1, A_2 and A_3 are respectively affected to sub-queries SQ_1, SQ_2 and SQ_3 : they order their execution by their host provider (e.g. A_1 commands provider P_1 to execute SQ_1). In step A, SQ_2's execution had completed, triggering re-estimations of $<\mathcal{M}_e, \delta_e, S_e>$ for all $e \in \mathcal{N}_2^-$. By applying (4), A_2 chose SQ_{21} as a next step and cleaned G_Q from now-unreachable edges. The agent waits for A_1 to complete its tasks. In step B, A_1 is done and A_2 can order the execution of SQ_{21} while A_3 still waits for SQ_3 to be completed. In step C, SQ_3 has been executed by its host provider, A_3 moved the data to SQ_{213}'s provider and marked it as ready to be handled by A_2. In further steps, A_2 will ultimately order the execution of SQ_{213}.

VI. Experiments and Results

Experiments carried out in order to evaluate both our quotation procedure and our dynamic execution model led to the implementation of a prototype for the Nebula system as well as simulated providers. We used real-world data from the IMDb dataset and real-world queries from the Join Order Benchmark (JOB) [21] in order to realistically test Nebula. Comparisons were made with a complete dataset download approach, as well as a full replication approach inspired from the literature (e.g. the SHAMC system [8]).

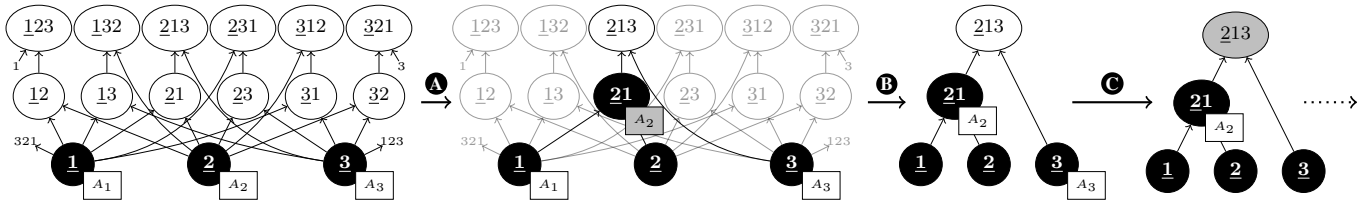


Figure 3. Evolution of an ABM during the execution of a query involving three providers. Underlined digits are the code of the sub-query's host provider.

Algorithm 2: Dynamic Execution

Input: $G_Q =< V, E >$ the query's graph, v the node to which the agent A is attached, and $A.P$ the provider managed by A

while $v.P = A.P$ **do**
  Wait for v's barrier to rise ;
  $<\mathcal{M}'_v, \delta'_v> \leftarrow v.P.execute\big(Subquery(C^{(v)})\big)$;
  `/* Knowledge sharing and re-estimations */`
  Under mutex on G_Q **do**
    $\mathcal{M}_v, \delta_v \leftarrow \mathcal{M}'_v, \delta'_v$;
    foreach $e \in E_v^-$ **do** Re-compute $\mathcal{M}_e$ and δ_e;
    foreach $w \in \mathcal{N}_v^-$ *by ascending order of depth* **do**
      Re-estimate $\mathcal{M}_w$, δ_w and S_w ;
      foreach $e \in E_w^-$ **do** Re-compute $\mathcal{M}_e$ and δ_e;
  `/* Next node choice and` G_Q `trimming */`
  Under mutex on G_Q **do**
    $v' \leftarrow N_v^- \cap BestTree$;
    $V \leftarrow V \setminus V_{\ominus}^{(v')}$;
    $E \leftarrow \big\{e \mid e = <I, O> \in E, \{I, O\} \cap V_{\ominus}^{(v')} = \emptyset\big\}$;
  if $v.P \neq v'.P$ **then**
    $v'.P$ loads intermediate results ;
  $v \leftarrow v'$;
  Notify v's barrier ;

A. Experimental Setting

Nebula is implemented as a Python `flask` API. SQL queries formulated over the multi-cloud schema are compiled with the `sly` library[5]. All graph-related aspects from Nebula are handled using `networkx` [34]. Agents are implemented using the threads provided by the standard Python library.

1) Simulated Providers: During our experiments, cloud providers are simulated. Those are also implemented as a Python `flask` API[6], and use PostgreSQL as a back-end DBMS. Provider's fares – listed in Table I – are inspired by Google's BigQuery. Their economical model is pay-per-query, and all the costs, may they be induced by querying, export or storage, are proportional to the amount of data those operations handle. Storage costs are usually expressed in monetary units per gigabyte per month, but we choose for simplicity reasons to drop the time dimension in this pricing. Thus, storage will be slightly over-priced as compared with a real setting.

Table I
PROVIDERS' SPECIFICATIONS. 'E' STANDS FOR 'EXPORT', 'Q' FOR 'QUERYING' AND 'S' FOR 'STORAGE'.

Prov.	Price (ct/GB)			Computer specifications
	E	***Q***	***S***	
P_1	8.50	1.75	0.35	Intel Core i5-7440HQ @ 2.80 GHz RAM 1 × 8 GB DDR4 2400 MHz SSD NVMe M.2
P_2	6.00	1.00	0.20	Intel Core i3-3227U @ 1.90 GHz RAM 2 × 4 GB DDR3 1600 MHz SSD SATA
P_3	2.50	0.25	0.075	Intel Core i5-2520M @ 2.50 GHz RAM 1 × 4 GB DDR3 1333 MHz HDD SATA

Providers' tenders are triples $<\mathcal{M}, \delta, S>$. $\mathcal{M}$ is the product of the querying cost and the sum of the size of the input relations S is computed according to the estimated output cardinality returned by PostgreSQL's `EXPLAIN` command. δ is estimated using a cost-based model adapted from the literature [32], leveraging a k-neighbours regressor (with $k = 5$). For a given query, the query's total cost is extracted from its execution plan fetched from PostgreSQL's `EXPLAIN` command, and is used to predict a query's execution time. This cost is the sum of the cost of all the plan's operators executed locally.

2) Benchmark Suite: We used the Join Order Benchmark (JOB) [21], a set of ad-hoc queries formulated over a database built using data from IMDb[7] . Both the database and the queries aim at reproducing real-world cases of querying over skewed data, pushing the query optimisers' selectivity estimators to their limits. The tables from this database are distributed thematically over three providers as depicted in Table II.

3) Comparison Points: Nebula's quotations and execution orchestration are compared to three strategies. The first point of comparison is the static execution approach. For each quotation computed for a multi-cloud query, its underlying execution plan is executed in order to evaluate uncorrected

[5]https://sly.readthedocs.io/

[6]https://flask.palletsprojects.com/en/1.1.x/

[7]Data was retrieved using the procedure detailled at https://github.com/gregrahn/join-order-benchmark/ published alongside the JOB article.

Table II
IMDB DATASET DISPATCHMENT

Provider	Theme	Nb. of tables	Size (MB)
P_1	Actors	7	5337
P_2	Movies	7	3501
P_3	Companies	3	314

Table III
FULL REPLICATION FOLLOWED BY SINGLE-CLOUD EXECUTION TOTAL MONETARY COST COST AND RESPONSE TIME.

Prov.	Cost (€)			Time (min)		
	Repl.	*Queries*	*Total*	*Repl.*	*Queries*	*Total*
P_1	0.98	3.95	6.85	110.5	6.15	117.1
P_2	0.97	2.26	5.16	95.0	8.5	103.5
P_3	0.95	0.56	3.46	> 600	465.4	> 600

quotations accuracy as well as dynamic optimisation worthwhileness.

The second point of comparison mimics the full replication strategies followed by single-cloud query execution implemented in the multi-cloud DBMS literature [8]. Because of their reliance on IaaS, existing systems do not make use of existing available data hosted by their underlying providers, and are therefore replicated on all sites including its original host. As a consequence, the replication cost for a provider P – computed as in (5) – takes into account this behaviour. This approach is tested over the three simulated providers.

$$\mathcal{M}^{(P)}_{\text{F-rep}} = \sum_{P' \in \mathcal{P}} \sum_{R \in \mathcal{R}_{P'}} Size(R) \times (\varepsilon^{(Q)}_{P'} + \varepsilon^{(E)}_{P'} + \varepsilon^{(S)}_{P}) \quad (5)$$

As depicted in Table III, replication is a big part of the total cost. Moreover, its duration eclipses the response time of the workload.

The third point of comparison is a full download of the dataset, matching an approach where the user downloads the dataset and subsequently analyses it on their infrastructure. Its total cost, depicted by (6) (with $\mathcal{P}$ the set of all involved providers, $\mathcal{R}_P$ the set of relations of a provider P, $\varepsilon^{(Q)}_P$ and $\varepsilon^{(E)}_P$ respectively its billing coefficient for querying and export, both in euros per GB), is $\mathcal{M}_{\text{DL-all}} = 0.95$ €. For the sake of our experiments, and in all fairness to the other points of comparison, the response time of each query is assumed to be the same as P_1 in Table III (i.e. the most efficient one), and the underlying costs are assumed to be the same as P_3 (i.e. the cheaper one). The total response time of the workload $\delta_{\text{DL-all}} = 212$ min is the sum of the export time for all providers and the fastest response time of a simulated provider.

$$\mathcal{M}_{\text{DL-all}} = \sum_{P \in \mathcal{P}} \sum_{R \in \mathcal{R}_P} Size(R) \times (\varepsilon^{(Q)}_P + \varepsilon^{(E)}_P) \quad (6)$$

However, the latter strategy is difficult to model generically. Indeed, it implies sufficient locally available resources and skills for the users. Their cost, depending on the dataset and the complexity of the workload, may range from a couple hundred euros for a laptop to a several millions euros private cloud. Those are hidden costs that are hard to take into account in our experiments. The execution time of the benchmark suite also heavily depends on the infrastructure.

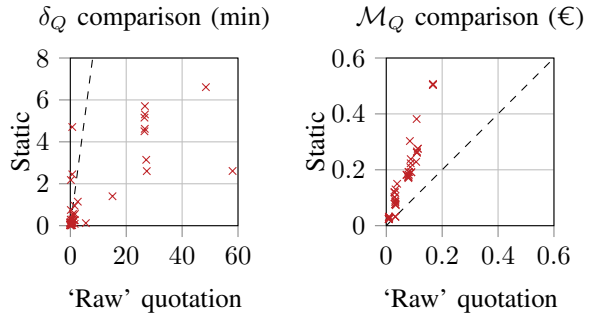


Figure 4. Comparison between the not-yet corrected quotations and the execution values of their underlying execution plan.

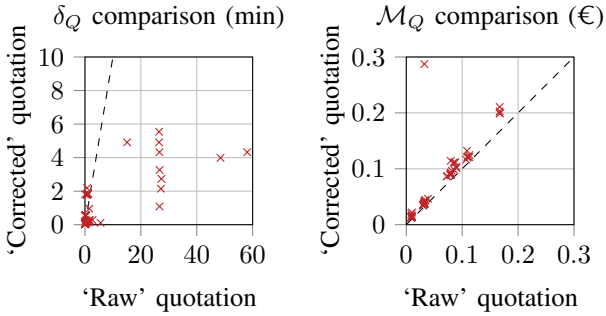


Figure 5. Comparison of the quotations before and after their correction.

B. Results and Discussion

Experimental results are described and explained hereafter. We first show the strengths and limits of our quotation computation method as well as the benefits of its post-processing. Then, we show that dynamic optimisation manages to be cheaper than a static approach. Last but not least, we show that Nebula's approach is faster and less expansive when leveraging already available data hosted in the cloud compared to the multi-cloud DBMS literature [8].

Note that the words 'static' and 'dynamic' in the subsequent figures are short forms referring to the nature of the optimisation technique used. Consistently with previously used notations, δ_Q refers to the response time of a query Q and $\mathcal{M}_Q$ its monetary cost.

1) Quotation Accuracy: As Fig. 4 shows, the quotation calculation method poorly estimates the performances of an execution plan. When comparing the estimations with its execution values, the response times are overestimated by up to an order of magnitude. On the other hand, monetary costs are underestimated.

These results are unsurprising, as δ and $\mathcal{M}$ are derivatives of estimates from the simulated providers' underlying DBMS.

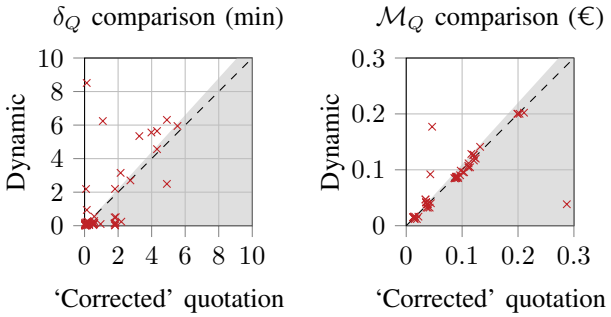


Figure 6. Comparison between the corrected quotations and the execution values of resulting of dynamic optimisation. The grey area corresponds to the tolerance interval with $\Lambda = 0.1$.

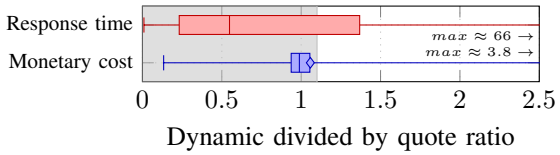


Figure 7. Ratio of the dynamic execution values to their quotation. The grey area corresponds to the tolerance interval with $\Lambda = 0.1$.

Such systems' optimiser are known to be error prone [21], occasionally making serious estimation errors, snowballing into the execution plans and leading to poor cost estimations. Ultimately, those errors come from selectivity estimation miscalculations – a known and well documented problem that is yet to be solved. Post-processing those estimates using online learning methods, as depicted in Fig. 5, manages to lower the response time estimates to more realistic values, and marginally changes the monetary cost.

As illustrated in Figures 6 and 7, Nebula manages to produce execution plans that respect the tolerance interval – with $\Lambda = 0.1$ – for 65 % of the benchmark's queries when it comes to response time, and for 78 % in terms of monetary cost. Notwithstanding possible different optimisation choices because of (4), raising Λ to 0.25 could lead to respective rates of 69 % and 93 %. These high rates are a positive confidence booster in our system, as it is quite unlikely that a user will pay more than the quotation's price.

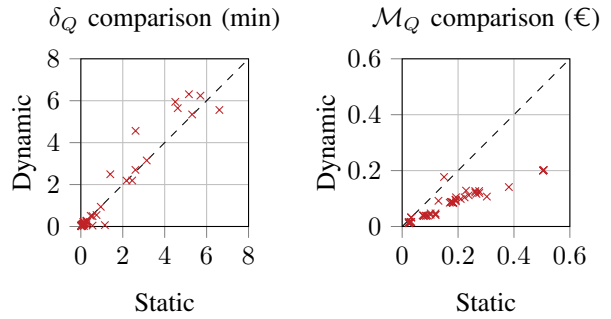


Figure 8. Queries response time δ_Q and monetary cost $\mathcal{M}_Q$ comparison between their static and dynamic execution.

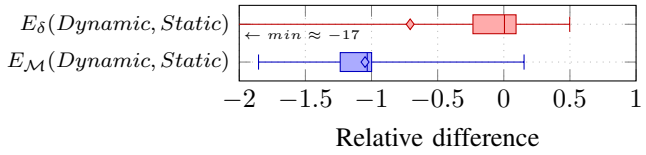


Figure 9. Relative difference $E(x, y) = \frac{x-y}{x}$ of response time δ and monetary cost $\mathcal{M}$ between queries' dynamic and static execution plan.

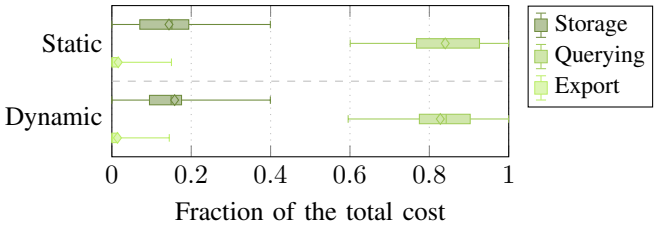
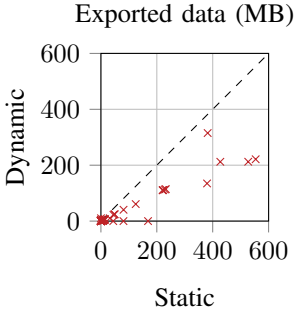


Figure 10. Distribution of the cost breakdown between export, querying and storage, presented as a ratio of the total cost of the queries.

2) Re-optimisation impact: As shown in Fig. 8, for equivalent performance, dynamic optimisation systematically produces execution plans that are cheaper to execute than statically optimised queries. When looking at the relative difference in execution time and monetary cost between these two methods (Fig. 9), it can be seen that in fact half of the queries actually perform better when re-optimised. However, those are often comparatively quick to execute: positive relative difference are typical of several minutes long queries. The extra monetary cost of static execution is on average the double of the dynamic execution's, can be as high as 170 %, and is rarely significantly less expansive.

Examining the nature of the monetary costs during execution, Fig. 10 reveals that the proportion of querying costs is slightly lower in re-optimised execution plans than in static ones. The opposite trend can be seen for storage costs. It is explained by the lower amount of data transferred by the re-optimised plans, as shown in Fig. 11. Indeed, the ratio of data moved by the static method to the dynamic method is, for three quarters of the queries, higher than 2.

In a multi-cloud environment with DBaaS providers, the



Statistics	$S_S^{(E)}/S_D^{(E)}$
Average	2.73
Min	0.01
Q_1	0.23
Median	0.55
Q_3	1.37
Max	66.42

Figure 11. Comparison between the exported amount of data between the static ($S_S^{(E)}$) and the dynamic ($S_D^{(E)}$) strategy for each query. The table displays some statistics about the ratio $S_S^{(E)}/S_D^{(E)}$.

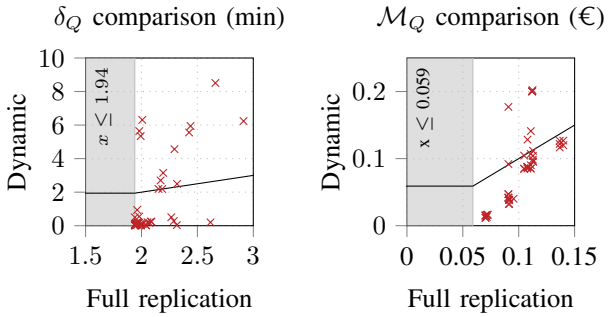


Figure 12. Comparison between the dynamically re-optimised execution values and their counterparts with a single-cloud execution approach on P_2. The grey areas are offset added because of the full replication costs.

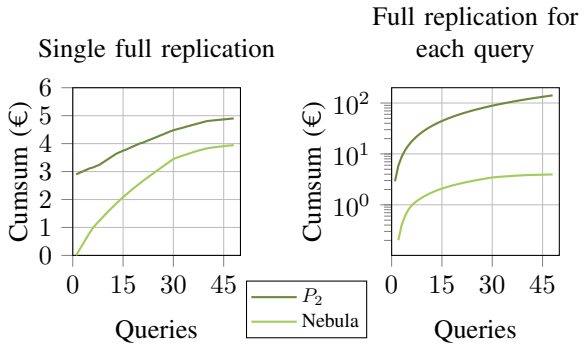


Figure 13. Cumulative sum of Nebula's monetary cost $\mathcal{M}_Q$ for a query Q and their counterpart on P_2, by ascending order of $\mathcal{M}_Q$.

monetary cost of queries is proportional to the total amount of data they handle and transfers of data are billed twice (at export and storage). Hence, such results come as no surprise, since agent-based dynamic optimisation methods are known for their ability to reduce data transfers over the network.

Response time of queries, however, is not derivable solely from the quantity of data queried but rather from the combination of the selectivity factor of each of their operators. As the latter cannot be estimated in a robust way, orchestration agents may indeed take bad decisions with respect to their response time minimisation target, especially when a provider is significantly cheaper and slower than the others.

3) Comparison with a full replication followed by a single-cloud execution: The total cost of the benchmark execution by Nebula is, depending on the target quotation chosen when several are generated for a query, between 3.94 € and 3.95 €. The total duration is between 66.4 min and 66.6 min. Comparing these figures with the single-cloud execution results for the provider P_2 previously presented in Table III, it can be seen that using Nebula allows for up to 58 % better performance for a lower or equivalent price.

Fig. 12 shows that the main reason why a single-cloud execution approach is outperformed by Nebula is the data replication process. However, its amortisation is workload-dependant, which can erode the advantages of a multi-cloud approach. On the other hand, the more data to replicate, the more competitive Nebula is. It is also worth noticing that a multi-cloud querying approach may, when data origin and intermediate queries' selectivity factor are favourable, be cheaper than a single-cloud execution approach. These results echo the comparison between centralised and distributed DBMS: data distribution enables parallelism, which may leads to better performance.

Last but not least, data replication raises the issue of updating the replicates. Fig. 13 depicts the evolution of the total price of the workload as it goes assuming two extreme cases: when the data is up-to-date during the whole process and when the data changes between each query (therefore requiring a new replication). When assuming infrequent updates, results suggest that Nebula may become more expansive at some point. However, if there are changes in the data sources, then either the full replication must be performed again, or update mechanism must be engineered. The latter implies further design work on the multi-cloud schema's metadata. The total cost for the considered workload could be as high as two orders of magnitude higher than Nebula's, notwithstanding performances degradation. In any case, a multi-cloud querying approach eliminates this needs and constantly offers access to the freshest readily available data at all times. This observation is reminiscent of the comparison between virtual data integration systems and data warehouses, the latter needing to find an equilibrium between data freshness and update costs.

VII. Conclusion

In this article, we introduced Nebula, a prototype offering querying capabilities over several relational databases readily available on various DBaaS infrastructure. Their pay-per-query model influenced the design of our system, leading to the development of a quotation computation method as well as a multi-cloud query optimiser. The former implements an exhaustive search strategy and relies on providers-generated tenders in order to estimate the intermediate queries' monetary cost and execution time; the latter implements an agent-based dynamic optimisation strategy. We showed that Nebula does manage to assess the monetary cost of the multi-cloud queries it orchestrates, hence enabling trustworthiness of its users *vis-à-vis* the quotations. The dynamic optimisation strategy offers better performances for a better price than the replication and execution model from the multi-cloud DBMSs literature, especially when the database is frequently updated.

These encouraging results open up new research possibilities regarding query optimisation in a multi-cloud environment. From a technical point of view, in order to overcome the inherent limits of the exhaustive search, recent moves towards integration of reinforcement learning techniques to solve the Join Order Problem [35] could be an inspiration. Mechanisms to leverage opportunities stemming from the market or to protect the users from malicious providers should also be designed. Finally, transcending the relational model to offer support for heterogeneous data sources, in order to push the polystore systems in a multi-cloud environment, is an exciting perspective at a time when diversity is the rule for public data.

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 (Grant number 44086TD), Amadeus program 2020 (French-Austrian Hubert Curien Partnership – PHC). We also thank the International Cooperation & Mobility (ICM) of the Austrian Agency for International Cooperation in Education and Research (OeAD-GmbH).

REFERENCES

[1] F. Vuolo, M. Żółtak, C. Pipitone, L. Zappa, H. Wenng, M. Immitzer, M. Weiss, F. Baret, and C. Atzberger, "Data Service Platform for Sentinel-2 Surface Reflectance and Value-Added Products: System Use and Examples," *Remote Sensing*, vol. 8, no. 11, p. 938, Nov. 2016.

[2] J. D. L. Beaujardière, "NOAA Environmental Data Management," *Journal of Map & Geography Libraries*, vol. 12, no. 1, pp. 5–27, Jan. 2016.

[3] M. Vukolić, "The Byzantine Empire in the Intercloud," *SIGACT News*, vol. 41, no. 3, p. 105–111, Sep. 2010.

[4] D. Abadi, A. Ailamaki, D. Andersen, P. Bailis, M. Balazinska, P. Bernstein, P. Boncz, S. Chaudhuri, A. Cheung, A. Doan, L. Dong, M. J. Franklin, J. Freire, A. Halevy, J. M. Hellerstein, S. Idreos, D. Kossmann, T. Kraska, S. Krishnamurthy, V. Markl, S. Melnik, T. Milo, C. Mohan, T. Neumann, B. Chin Ooi, F. Ozcan, J. Patel, A. Pavlo, R. Popa, R. Ramakrishnan, C. Ré, M. Stonebraker, and D. Suciu, "The Seattle Report on Database Research," *ACM SIGMOD Record*, vol. 48, no. 4, pp. 44–53, Feb. 2020.

[5] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha, "SPANStore: Cost-Effective Geo-Replicated Storage Spanning Multiple Cloud Services," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. Farminton, USA: ACM, Nov. 2013, pp. 292–308.

[6] D. Bermbach, M. Klems, S. Tai, and M. Menzel, "MetaStorage: A Federated Cloud Storage System to Manage Consistency-Latency Tradeoffs," in *2011 IEEE 4th International Conference on Cloud Computing*. Melbourne, Australia: IEEE Computer Society, Jul. 2011, pp. 452–459.

[7] D. Dobre, P. Viotti, and M. Vukolić, "Hybris: Robust Hybrid Cloud Storage," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC '14. New York, USA: ACM, Nov. 2014, pp. 1–14.

[8] L. Wang, Z. Yang, and X. Song, "SHAMC: A Secure and Highly Available Database System in Multi-Cloud Environment," *Future Generation Computer Systems*, vol. 105, pp. 873–883, Apr. 2020.

[9] M. A. Alzain, B. Soh, and E. Pardede, "MCDB: Using Multi-clouds to Ensure Security in Cloud Computing," in *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*. Sydney, Australia: IEEE Computer Society, Dec. 2011, pp. 784–791.

[10] A. Rafique, D. Van Landuyt, E. Truyen, V. Reniers, and W. Joosen, "SCOPE: self-adaptive and policy-based data management middleware for federated clouds," *Journal of Internet Services and Applications*, vol. 10, no. 1, p. 2, Jan. 2019.

[11] T. G. Papaioannou, N. Bonvin, and K. Aberer, "Scalia: An adaptive scheme for efficient multi-cloud storage," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, Salt Lake City, USA, Nov. 2012, pp. 1–10.

[12] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, "DepSky: Dependable and Secure Storage in a Cloud-of-Clouds," *ACM Transactions on Storage*, vol. 9, no. 4, pp. 12:1–12:33, Nov. 2013.

[13] M. Li, C. Qin, and P. P. C. Lee, "CDStore: Toward Reliable, Secure, and Cost-Efficient Cloud Storage via Convergent Dispersal," in *USENIX ATC 15*, Santa Clara, USA, 2015, pp. 111–124.

[14] P. P. Jayaraman, C. Perera, D. Georgakopoulos, S. Dustdar, D. Thakker, and R. Ranjan, "Analytics-as-a-service in a multi-cloud environment through semantically-enabled hierarchical data processing," *Software: Practice and Experience*, vol. 47, no. 8, pp. 1139–1156, 2017.

[15] G. Wiederhold, "Mediators in the architecture of future information systems," *Computer*, vol. 25, no. 3, pp. 38–49, Mar. 1992.

[16] J. Duggan, J. Kepner, A. J. Elmore, and S. Madden, "The BigDAWG Polystore System," *SIGMOD Record*, vol. 44, no. 2, p. 6, 2015.

[17] S. Yin, A. Hameurlain, and F. Morvan, "SLA Definition for Multi-Tenant DBMS and its Impact on Query Optimization," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 11, pp. 2213–2226, Nov. 2018.

[18] F. Morvan, M. Hussein, and A. Hameurlain, "Mobile agent cooperation methods for large scale distributed dynamic query optimization," in *14th International Workshop on Database and Expert Systems Applications, 2003. Proceedings.*, Sep. 2003, pp. 542–547.

[19] A. Rafique, D. Van Landuyt, V. Reniers, and W. Joosen, "Towards an Adaptive Middleware for Efficient Multi-Cloud Data Storage," in *Proceedings of the 4th Workshop on CrossCloud Infrastructures & Platforms*, ser. Crosscloud'17. Belgrade, Serbia: ACM, Apr. 2017, pp. 1–6.

[20] J. Ortiz, V. T. de Almeida, and M. Balazinska, "Changing the Face of Database Cloud Services with Personalized Service Level Agreements," in *CIDR*. Acilomar, USA: CIDR, 2015, p. 13.

[21] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?" *Proceedings of the VLDB Endowment*, vol. 9, no. 3, pp. 204–215, Nov. 2015.

[22] J. M. Hellerstein, M. J. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, and M. A. Shah, "Adaptive Query Processing: Technology in Evolution," *IEEE Bulletin of the Technical Committee on Data Engineering*, vol. 23, no. 2, pp. 7–18, Jun. 2000.

[23] L. Amsaleg, A. Tomasic, M. Franklin, and T. Urhan, "Scrambling query plans to cope with unexpected delays," in *Fourth International Conference on Parallel and Distributed Information Systems*. Miami Beach, USA: IEEE Computer Society, Dec. 1996, pp. 208–219.

[24] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. Patterson, and K. Yelick, "Cluster I/O with River: making the fast case common," in *Proceedings of the sixth workshop on I/O in parallel and distributed systems*, ser. IOPADS '99. New York, USA: ACM, May 1999, pp. 10–22.

[25] R. Avnur and J. M. Hellerstein, "Eddies: continuously adaptive query processing," in *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '00. Dallas, USA: ACM, May 2000, pp. 261–272.

[26] I. Trummer, J. Wang, D. Maram, S. Moseley, S. Jo, and J. Antonakakis, "SkinnerDB: Regret-Bounded Query Evaluation via Reinforcement Learning," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD '19. New York, USA: ACM, Jun. 2019, pp. 1153–1170.

[27] E. Wong and K. Youssefi, "Decomposition — a Strategy for Query Processing," *ACM Transactions on Database Systems (TODS)*, vol. 1, no. 3, pp. 223–241, Sep. 1976.

[28] J.-P. Arcangeli, F. Morvan, A. Hameurlain, and F. Migeon, "Mobile Agents Based Self-Adaptive Join for Wide-Area Distributed Query Processing," *Journal of Database Management*, vol. 15, no. 4, pp. 25–44, 2004.

[29] D. Agrawal, S. Chawla, B. Contreras-Rojas, A. Elmagarmid, Y. Idris, Z. Kaoudi, S. Kruse, J. Lucas, E. Mansour, M. Ouzzani, and P. Papotti, "RHEEM: Enabling Cross-Platform Data Processing," in *Proceedings of the VLDB Endowment*, vol. 11. Rio de Janeiro, Brazil: VLDB Endowment, Aug. 2018, p. 14.

[30] D. T. Wojtowicz, S. Yin, and F. Morvan, "SLA definition for multi-cloud queries," in *Actes de la conférence BDA 2020*, Paris (online), France, Oct. 2020, p. 80.

[31] E. Tsamoura, A. Gounaris, and K. Tsichlas, "Multi-objective optimization of data flows in a multi-cloud environment," in *Proceedings of the Second Workshop on Data Analytics in the Cloud*, ser. DanaC '13. New York, USA: ACM, Jun. 2013, pp. 6–10.

[32] A. Kleerekoper, J. Navaridas, and M. Lujan, "Can the Optimizer Cost be Used to Predict Query Execution Times?" *arXiv:1905.00774 [cs]*, May 2019.

[33] J. Montiel, M. Halford, S. M. Mastelini, G. Bolmier, R. Sourty, R. Vaysse, A. Zouitine, H. M. Gomes, J. Read, T. Abdessalem, and A. Bifet, "River: machine learning for streaming data in Python," *arXiv:2012.04740 [cs]*, Dec. 2020.

[34] A. Hagberg, P. Swart, and D. S Chult, "Exploring network structure, dynamics, and function using networkx," Los Alamos National Laboratory, Los Alamos, USA, Tech. Rep. LA-UR-08-05495; LA-UR-08-5495, Jan. 2008.

[35] S. Krishnan, Z. Yang, K. Goldberg, J. Hellerstein, and I. Stoica, "Learning to Optimize Join Queries With Deep Reinforcement Learning," *arXiv:1808.03196 [cs]*, Jan. 2019.