Database Optimization for the Cloud: Where Costs, Partial Results, and Consumer Choice Meet

Willis Lang, Rimma V. Nehme, Ian Rae
Microsoft Gray Systems Lab
{wilang, rimman, ianrae}@microsoft.com

ABSTRACT

Database services in the cloud (DBaaS) allow users to convert the fixed costs of deploying and maintaining a database system to variable costs that are incurred in a pay-as-you-go manner. Considerable research focus has been given to the cost of the elasticity (of performance) and reliability of a DBaaS. This is because in the cloud, users are able to modulate these characteristics of a DBaaS by paying more or less to the provider. However, the one invariant has always been that the user will receive a complete and correct result. In this paper we consider another possibility; that users may be willing to accept different quality results from a DBaaS that aren't complete and correct if the price is right. Recently, there has been research classifying "partial results" produced using incomplete input (due to failures) while processing a query. These classifications provide a broad and general way to understand a partial result using semantic guarantees that can be made about the result. In this paper, we consider a database system in the cloud that allows users to control the cost of a query by defining the sorts of partial results that are acceptable.

1. INTRODUCTION

Many major cloud providers have begun selling Databases-as-a-Service (DBaaS), where customers pay for access to database systems hosted and managed by the provider. These DBaaS offerings are attractive to customers for two reasons. First, due to economies of scale, the hardware and energy costs incurred by users are likely to be much lower when they are paying for a share of a service rather than running everything themselves. Second, the costs incurred in a well-designed DBaaS will be proportional to the actual usage (a "pay-per-use" model). In this paper, we consider DBaaS as a "utility service," where the cloud provider can meter the DBaaS services (compute and storage) and charge customers for their exact usage, similar to how we pay for utilities like water and gas.

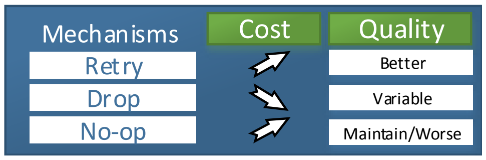
Users of DBaaS systems have taken advantage of this flexibility to build sophisticated parallel database infrastructures. In particular, many customers using Microsoft SQL Azure partition their data across thousands (or even tens of thousands) of independent SQL Azure instances, and they combine data across partitions by sending "fan-out" queries across all of these instances. Unfortunately, when executing queries, the default behavior for DBaaS systems like SQL Azure is to be "all or nothing."

This means that the system either returns a complete, correct result, or, in the case of failure, it returns no result whatsoever. This approach is certainly reasonable—after all, one can hardly criticize a data processing system for correctness—but it has some negative aspects. For instance, if a failure occurs, the user may still be charged the expense of processing a portion of the query since these SQL Azure instances are billed separately, despite getting no real value from the answer.

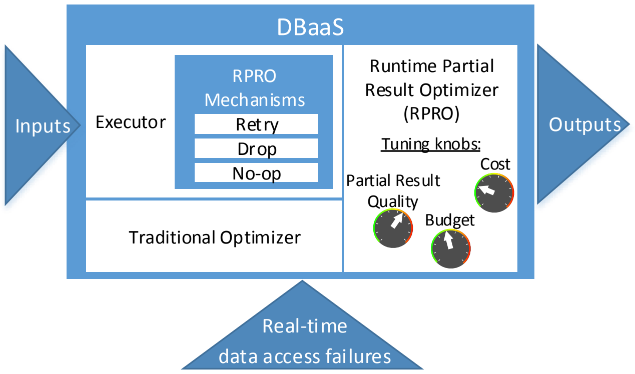
Previous work addressed this issue by creating a set of semantics for "partial results" [10]. Under this model, when the DBaaS system encounters a failure, the system returns whatever results it has computed thus far, using the partial results semantics to provide certain guarantees about the quality of the result. Consequently, users who are able to make use of an answer that is partially correct can get at least some return on their investment for running the query, even if a perfect result is not attainable. While this work is effective in improving the system's response to failures, it raises an interesting question: would a user who is willing to accept a partial result in the face of failure also be willing to accept a partial result *even if no failure has occurred*?

Although this may seem strange, it makes a certain amount of economic sense. In order to provide a perfect result, the DBaaS system must contact all servers holding partitions of the base data, and the user must pay for the processing time spent on every server. If a user is willing to accept a partial result, the DBaaS system can eliminate some partitions by never contacting the individual database instance or by terminating execution for a partition early, translating directly into cost savings for the user. We assert that, just as previous work explored the idea that some users would accept an approximate aggregate [1, 6], partial result sets are sufficient for some users, and furthermore, these users are being overcharged by traditional DBaaS offerings.

Accordingly, the goal of this work is to enable DBaaS systems to proactively adjust the processing power (and the associated cost) necessary to execute a query, using requirements on result quality given by users in terms of partial result semantics. Optimization for performance in distributed database systems using economic approaches has been studied, but has either not considered partial results [19] or the system changes to enable such trade-offs [13]. Furthermore, our proposal is more general than prior work in sampling-based approximate querying systems [1, 6] as (1) we propose operating over the entire SQL surface, and (2) our optimizations are much more generalized (e.g., we may completely ignore certain partitions of the input *midway* through query execution while fully computing the results on others.) We propose accomplishing this by integrating



(a)



(b)

Figure 1: (a) Three mechanisms to trade-off *cost* and *quality*. (b) A database system in the cloud that includes a new "Runtime Partial Result Optimizer" as well as our executor mechanisms.

a *Runtime Partial Result Optimizer* (RPRO) into the DBaaS query processor. The responsibility of the RPRO is to modify the behavior of query pipelines and operators during execution to control execution cost and result quality.

In order to manage this balance, the partial result optimizer utilizes three novel mechanisms (see Figure 1a): *retry*, *drop*, and *no-op*. *Retry* repeats work to improve result quality, *drop* eliminates some tuples to save cost, and *no-op* passes tuples through without processing them in order to preserve result quality. By determining when to invoke these mechanisms, the RPRO can have a significant impact on the execution of a query and dramatically change both the quality and the cost of a result.

Of course, care must be taken when attempting to quantify the level of result quality that the user would find acceptable. For example, suppose a user indicates to the system that they are willing to accept a result set that is potentially missing some tuples. One possible strategy a data processing system could use to answer this user is to simply do nothing (i.e., *drop* all of the input data) and always return an empty result set. Such a system would be both very fast and very cheap, but it's easy to see that its users would not be very happy!

This shows that we must have some way of balancing the user's value of a correct result against the cost incurred to produce the result. To address this, we rely on the user to provide a penalty function that relates the various result quality levels, allowing users to express a preference for the type of partial result they received. We use this penalty model to formulate an optimization problem that minimizes execution cost and quality penalties, subject to a total cost budget specified by the user, which the RPRO attempts to optimize as query execution progresses (see Figure 1b).

This paper provides the following main contributions:

- A new cloud querying paradigm where users can opt to pay less for a partial result.
- Methods for enhancing physical operators in the DBaaS system to improve quality or save cost at runtime.
- A description of various optimization problems involving cost, quality preferences, and budget.

The rest of the paper is organized as follows. We discuss our assumptions about the DBaaS system and briefly describe partial result semantics in Section 2. We describe our mechanisms and various potential optimization problems in Section 3. Finally, we present related work in Section 4, and we conclude in Section 5.

2. PRELIMINARIES

In this section we provide the necessary background and definitions required for the remainder of this paper. In particular, we discuss the assumptions we make about DBaaS system that our work operates in, a brief overview of semantics for partial results, and a description of our cost model.

2.1 System

We can take any modern parallel database system (Microsoft SQL Server PDW, HP Vertica, IBM DB2, Pivotal GPDB, etc.) and install it across a cluster of virtual machines in the cloud to form the foundations of a DBaaS. Users of these systems then pay-as-they-go, incurring charges based on the amount of storage and computational resources they utilize. We assume that the DBaaS pricing model is such that users are charged for their exact usage, rather than at a coarser granularity (such as per hour or per day). This pricing model is more precise than the typical DBaaS offering in the market today, but we argue that the competition amongst DBaaS providers will pressure them to provide more precise usage metering over time.

Most of these database systems are able to query over data stored in data sources that are "external" to the cluster where the computation occurs. An example of such functionality would be the ability for SQL Server PDW to query over remote Hadoop Distributed File System (HDFS) data or query a "linked-server" database. In this work, we assume that users are issuing queries over many such external data sources and combining the results in some way. In fact, in Azure, Microsoft currently has customers that have horizontally sharded their tables across thousands to tens of thousands of independent SQL Azure databases. Querying these external sources typically incurs additional costs, as these databases are independent and not locally stored with the parallel database cluster. Furthermore, these external, loosely coupled data sources may not always be available due to failures, congestion, or maintenance.

2.2 Partial Results

Here, we briefly revisit the concepts of partial result semantics and classification when processing over data sources that may fail, which were described in our recent work [10]. These partial results semantics are defined with respect to the "true result" which a parallel database system would return if no failures have occurred. These partial result semantics provide guarantees on two specific aspects of a result set: cardinality of the set and correctness of the values.

We describe the *Cardinality* of a result set using one of four possible states:

Complete The result set is guaranteed to have exactly the same number of tuples as the true result.

Incomplete The result set may have fewer tuples than the true result set.

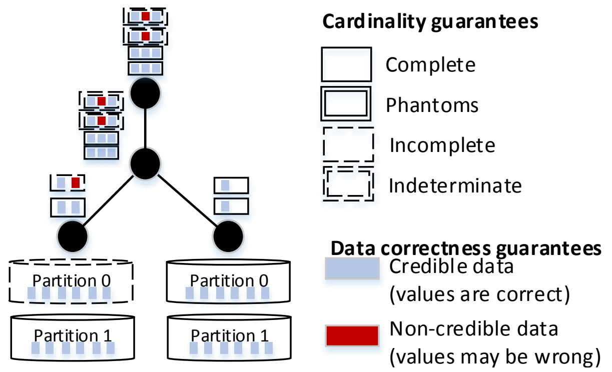


Figure 2: An example of partial result classification when input is *Incomplete*.

Phantom The result set may have more tuples than the true result set. These extra tuples may be produced by non-monotone operators or predicates over *Non-credible* values (see below).

Indeterminate The result set may have fewer tuples or more tuples than the true result set (i.e., both *Incomplete* and *Phantom*).

We describe the correctness (or *Credibility*) of values in a result set using one of two possible states:

Credible The value was directly read from persistent storage or it was computed from a *Complete* set of Credible values.

Non-credible The value was computed from a partial result. In other words, a value computed from a non-credible value or computed from a result set whose cardinality was not *Complete*.

As we show in Figure 2, the framework for the partial result analysis and classification of tuple sets relies on studying the way operators transform and propagate the classification of their input data to their output tuple sets. We can change the precision of our classification by changing the granularity at which we classify data (e.g., classifying tightly bounded regions of a result set). In the figure, at the "partition level" of analysis, we can classify different horizontal partitions of a tuple set with different *Cardinality* and *Credibility* semantics.

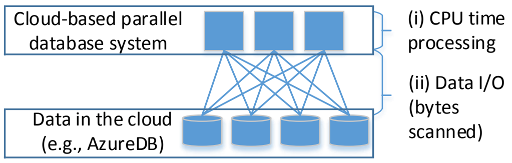
2.3 Query Processing Cost Model

Parallel database systems typically model the cost of query processing using metrics such as data pages read and written, CPU-time spent processing, and working memory consumed. In our discussion of a cloud-based data warehouse, we consider only a simplified query cost model using the CPU-time spent by the parallel database system running in the cloud and the amount of table data retrieved from the cloud-based storage (shown in Figure 3). This cost model also represents the dollar cost to the user of running their query. As we will see later, our optimization objective function will be the minimization of a function that relies on this cost model:

$$\text{cost}(query) = \text{cost}(plan) + \text{cost}(data\ retrieved) \qquad (1)$$

Here the term $\text{cost}(plan)$ can basically be thought of as a traditional query optimization cost function. Since we consider only the CPU time spent, this can be calculated by summing the CPU time per tuple processed across all of the operators of the query plan.

The second major component of our *cost* model is the cost of data retrieved from storage (the egress cost): $\text{cost}(data\ retrieved)$. The amount of table data retrieved from the cloud storage subsystem may be the volume of all of the relations involved in the query if the storage system is a basic distributed file system (e.g., Azure blob store, Amazon S3, HDFS). In certain cases, the data may be filtered at the source to reduce this egress cost (e.g., if the data sources are themselves independent databases). Finally, we also consider relations horizontally partitioned across multiple data sources such that we may selectively scan partitions of the relations.



Cloud-based parallel database system
Data in the cloud (e.g., AzureDB)
(i) CPU time processing
(ii) Data I/O (bytes scanned)

Figure 3: The query cost model consists of (i) the CPU time spent processing the query, and (ii) the amount of table data retrieved from the cloud-based storage.

3. RUNTIME OPTIMIZATION OF PARTIAL RESULTS IN THE CLOUD

When we think of traditional query plan optimization, we generally think of join order enumeration, plan shape, physical operator choices, etc. [8]. One approach to improving the utility of DBaaS systems is to add partial results optimization into this set; if we were to introduce our new optimization constraints and objectives into these optimization phases, there would undoubtedly be instances where the optimizer would choose a different plan. However, to increase modularity and minimize the complexity of the changes to the database system, we leave this as future work. Instead, our discussions will focus on a new database component that we call the **Runtime Partial Result Optimizer** (RPRO).

Using the models we described in Section 2, the RPRO continuously optimizes a partial results optimization problem during query processing, similar to dynamic re-optimization [9]. At runtime, the optimizer modifies the behavior of query pipelines and operators to accommodate failures, the accumulating cost of processing, and the partial result classification of the query. In order to accomplish this, the RPRO has three new executor mechanisms at its disposal (see Figure 1), which we will introduce more fully in Section 3.2. However, before we delve into our descriptions of the system, we will first define our problem through the inputs and outputs to the new optimizer (shown in Figure 4).

Given the following inputs:

- W – A SQL query (or a workload of multiple queries).
- D – An input dataset.
- B – *Optional*: The maximum cost budgeted for the query based on the cost model for processing the query (Section 2.3).
- P – *Optional*: A partial result penalty model as we will describe in Section 3.4.

We wish to execute W with potential data source access failures, and produce the following outputs:

- R – A partial result tuple set.
- L – A partial result classification.
- C – The cost of producing R and L (i.e., price to the user).

With these inputs and outputs, with the possibility of failures and receiving partial results, there are numerous constraints that can

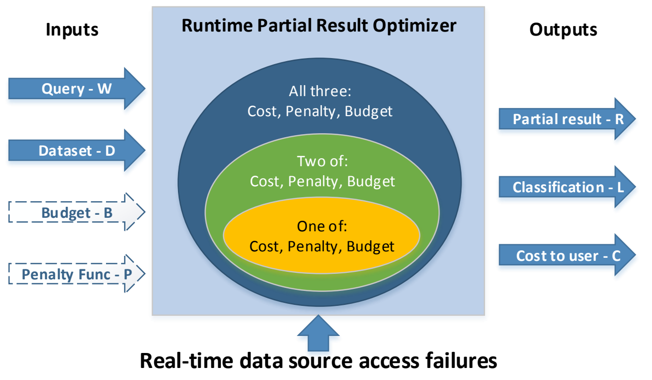


Figure 4: Given the inputs and potential data access failures, the runtime partial result optimizer will produce three outputs.

be applied and objectives to minimize during query optimization and processing. In this paper, we consider three of these objectives/constraints that will become components of our optimization problem:

Cost Cost of the query as defined in Section 2.3. – $\text{cost}(query)$

Penalty User-defined partial result penalties that we will define in Section 3.4. – $\text{penalty}(root)$

Budget The maximum cost that a query can accrue before query termination. – B (see optional input above)

Our discussion in Section 3.5 (organized according to the diagram in Figure 4) will highlight the pitfalls when we ignore any one of these components and thus show the importance of incorporating all three components during query optimization and processing.

3.1 Assumptions

First, our parallel data warehouse in the cloud has the architecture described in Section 2.1, and we use the concrete scenario of a traditional parallel database system like Microsoft SQL Server PDW running in Azure virtual machines against SQL Azure. Furthermore, the base table data is horizontally sharded across many (thousands or more) loosely coupled, independent data sources (like Azure Blob store or SQL Azure databases). Finally, we optimize and process left-deep query plans where the leaf operators connect to and scan the data sources.

For our failure model, we assume that failures occur only while accessing the base data sources and not in the database system virtual machines. These failures can occur at any time. Our cost model for running queries in the cloud is defined in Section 2.3. We assume that the dollar cost that we pass onto the customer is solely defined by this cost model (compute and egress).

We will introduce a new partial result penalty model below to accompany the cost model for optimization. The partial result penalty cost model guides optimization, but is not directly translated to dollar cost to the user and does not represent cost incurred by the cloud provider.

3.2 New Mechanisms for the Query Executor

In this paper, we introduce three new mechanisms: *retry*, *drop*, and *no-op*. With these mechanisms, the RPRO can react to data access failures as well as intentionally reduce the amount of processing work to control costs.

***Retry* mechanism:** The *retry* mechanism provides the ability of restarting an operator pipeline. Recall from our assumptions in

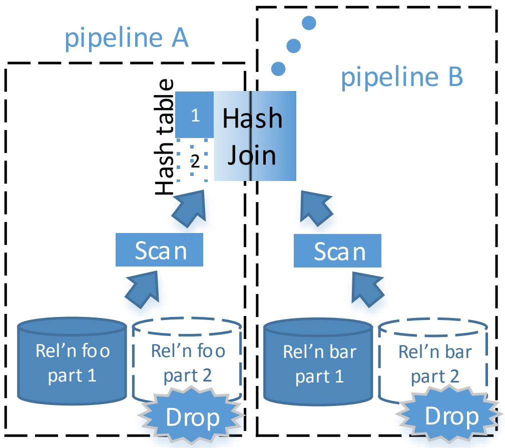


Figure 5: Pipeline A triggers the *drop* mechanism for partition 2 and doesn't build the corresponding hash table entries. In this case, pipeline B's *drop* mechanism is subsequently triggered so partition 2 is not read.

Section 3.1 that our query plans are all left-deep plans with leaf operators scanning data from SQL Azure data sources. If the *retry* mechanism is invoked, perhaps (but not necessarily) due to a data access failure at the leaf of the pipeline, then the results at the root of the pipeline (a blocking operator or the top of the plan tree) will be discarded and the entire pipeline will start from scratch. Furthermore, in the case of horizontally partitioned input data, the *retry* mechanism can target specific partitions of the input to be retried, allowing us to temper the cost of retrying a pipeline. *This mechanism allows us to produce results with better partial result classifications, but at higher cost.*

***Drop* mechanism:** The *drop* mechanism provides the ability of a pipeline to ignore tuples that it receives at the leaf operator from the data source and "drop them on the floor." In particular, with partition-level classification of partial results (see Section 2.2), the *drop* mechanism allows us to ignore all of the tuples of a particular partition (see Figure 5). This figure also shows that if the mechanism is invoked during the execution of an operator pipeline, it can cause a cascade onto the subsequent pipelines that have yet to be executed, since those operators will not receive the dropped partition in their input. *This mechanism allows us to reduce the dollar cost of processing a query, but may also reduce the quality of the final partial result classification.*

***No-op* mechanism:** The *no-op* mechanism provides a dual-like ability to the *drop* mechanism in that a pipeline can now choose to ignore filter predicates (`SELECT` operators) when processing tuples. Similar to the *drop* mechanism above, this mechanism can be triggered at the partition level whereby we choose to allow all of the tuples of a particular partition to pass the predicate.

A reason that one may want to trigger the *no-op* mechanism is if the predicate is going to be evaluated over *Non-credible* values (see Section 2.2). In this case, if we apply the predicate, the resulting partial result level immediately becomes *Indeterminate* (as we show in Figure 6), since we cannot be sure that the tuples which passed the predicate should have been produced and vice versa. On the other hand, if we trigger the *no-op* mechanism, then we will have allowed all of the tuples through and we will have only the possibility of *Phantom* tuples, since we know that we will not have introduced any erroneously missing tuples by filtering them out. In other words, it prevents us from mistakenly removing tuples from the result.

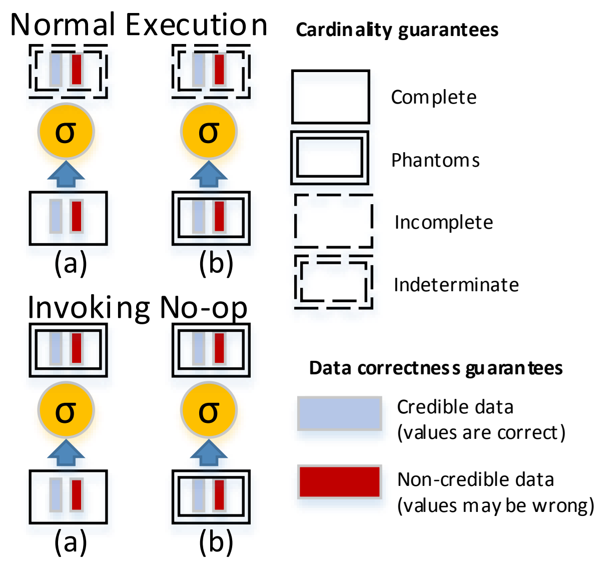


Figure 6: *No-op* (a) minimize quality loss; (b) preserve quality.

This mechanism allows us to potentially produce results with better partial result classifications, but possibly at a higher cost since there is no data reduction from the predicate.

3.3 Control and Triggering Mechanisms

The mechanisms described above must be controlled by some part of the execution engine. Certain parallel database systems like SQL Server PDW execute queries in a discrete series of operations [17] dispatched serially by a centralized "*Engine*" process. We propose that the RPRO is implemented within such centralized *Engine* processes, since it is already tasked with coordinating the execution of operations across all of the cluster's compute nodes. The RPRO has the ability to trigger the mechanisms at the appropriate pipelines (for the *retry* and *drop* mechanisms) and the `SELECT` operators (for the *no-op* mechanism). As we described above, the *drop* mechanism may cause cascading *drop* mechanisms across many pipelines, also ensured by the RPRO.

During pipeline processing, as the RPRO monitors the execution across the cluster, there are many instances in which the RPRO may trigger one of our mechanisms. The *retry* mechanism may be triggered (i) for the currently executing pipeline if a failure has been detected when reading data or (ii) for prior pipelines, if the output from the current pipeline will be of unacceptably poor quality (defined by a penalty model like that described in Section 3.4). In both of these cases, the cost of *retrying* a pipeline is taken into account by the objective function of our optimizer.

The *drop* and *no-op* mechanisms may be triggered after any operator in the pipeline produces a tuple. When the *cost* of processing creeps too high, the *drop* mechanism may be triggered to reduce further processing cost. Alternatively, if the projected quality of the output of the operator is low, then the *no-op* mechanism may be triggered to avoid *Indeterminate* results.

Essentially, the RPRO must be continuously re-evaluating its objective function by updating its estimated cost with the actual amount of data read and processed, as well as the resulting quality loss if (a) failures have occurred or (b) any *drop*/*no-op* has been triggered. When the RPRO is calculating cost, it uses traditional cardinality estimates of the tuples yet to be processed as well as the number of tuples processed at runtime. This allows the RPRO to

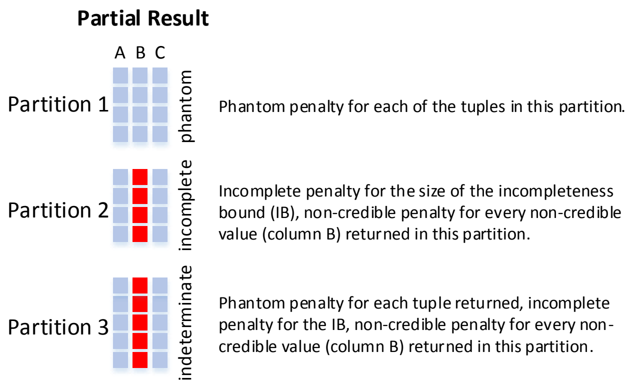

Partial Result
A B C
Partition 1
phantom
Phantom penalty for each of the tuples in this partition.
Partition 2
incomplete
Incomplete penalty for the size of the incompleteness bound (IB), non-credible penalty for every non-credible value (column B) returned in this partition.
Partition 3
indeterminate
Phantom penalty for each tuple returned, incomplete penalty for the IB, non-credible penalty for every non-credible value (column B) returned in this partition.

Figure 7: Assessing penalties to a partial result returned. Each partition of the result may be penalized differently.

make *a priori* decisions about how to process a given partition, since it can compare the expected cost of normal processing against the expected effect of invoking one of the partial results mechanisms.

Although this serves as a reasonable baseline, there are many open questions as to how the RPRO predicts the impact of future failures on the quality of the results. In this paper, we assume the naïve optimizer that assumes that no additional failures beyond those that have been observed will occur. For example, if a *retry* is triggered, it is assumed that the retry will succeed, and the RPRO will not encounter further failures. We leave incorporating the likelihood of failures into the model (i.e., a predictive optimizer) for future work.

3.4 Partial Result Penalty Model

Since certain partial result classifications may be more useful and acceptable to a user than others, we rely on the user to guide the RPRO through a *partial result penalty model*. The partial result penalty model allows the user to specify their preference for particular partial result classifications by imposing varying penalties on the RPRO's objective function depending on which partial result classifications it returns. Consequently, in our complete optimization problem formulation, the optimizer will attempt to return "better" partial results (as defined by the penalty function) so long as the cost for doing so does not exceed the penalty.

To inform the RPRO's decision-making, we extend the partial results model of [10] with an *Incompleteness bound*. When *Incomplete* data is read from a data source (due to a failure or due to a *drop* invocation), we assume that we know how many tuples we should have read from that source and this provides a maximum bound on how *Incomplete* we are. We can propagate this bound up the query plan through all unary operators (as well as binary operators like `UNION ALL`). In the case of a join, the *Incompleteness bound* is re-calculated by the size of the Cartesian product. For every operator, we have the number of input rows (processed and estimated remaining) and the *Incompleteness bound* on this input.

In Figure 7, we illustrate how partial result penalties can be assessed onto a partial result returned. We first consider partition 1 that is classified *Phantom* with all of its values labelled *Credible*. Partition 1 will be assessed the user-defined *Phantom* penalty for every tuple in the partition since any one of them could be an erroneously present tuple.

Partition 2 is *Incomplete* and has a column that is *Non-credible*. This partition will have a nonzero *Incompleteness bound* and the user-defined *Incomplete* penalty will be assessed against the size of the bound. Furthermore, for every tuple returned, one of the values is *Non-credible* and so the user-defined *Non-credible* penalty will

Cost	Penalty	Budget	Strategy
X			Never read anything, always return *Incomplete*.
	X		Never return a partial result; retry if failures occur.
		X	Execute until query complete or budget exhausted; return whatever results available at that time.
X	X		Trade off penalty and cost based on estimated cardinalities.
X		X	Never read anything, always return *Incomplete*.
	X	X	Use only retry until query complete or budget exhausted; return whatever results available at that time.
X	X	X	Trade off penalty and cost based on estimated cardinalities and remaining budget.

Table 1: Strategies for different formulations of the optimization problem.

be assessed for each of these *Non-credible* values.

Partition 3 is *Indeterminate* with a *Non-credible* column. The penalty due to partition 3 is the sum of the *Phantom* penalty, the *Incomplete* penalty on the *Incompleteness bound*, and the *Non-credible* penalty on all of the suspect values returned. Intuitively, since partition 3 has the poorest semantic guarantees, in incurs the most penalties for the optimizer to take into account.

Finally, if an optimizer is to be able to make decisions based on both the traditional *Cost* of the execution and the *Penalty* of the partial result, then the penalty model must be normalized to the *Cost*. One way we can achieve this is if the penalty model is calculated as a function of the cost of the plan (see $\mathrm{cost}(plan)$ in Section 2.3.) For instance, *Phantom* partitions that are produced are already charged a cost based on computation and data egress, but with a user-specified penalty (i.e., a multiplier) for the *Phantom* classification, we will also impose an additional multiple on the compute and egress cost. *Incomplete* partitions are trickier, but if we calculate the *Penalty* based on a user-provided *Incomplete*-multiple on the *Incompleteness bound*, then we can still determine the penalty for such a partition. Similarly, we can calculate a penalty for *Non-credible* values within partitions.

3.5 Optimization Problem Formulations

Earlier, in Figure 4, we showed how runtime partial result optimization may be performed over an optimization space. In this section, we present different formulations for the optimization problem using three components: *Cost*, *Penalty*, and *Budget*. The strategy taken by the RPRO changes significantly depending on which of these components are used, and consequently we discuss each formulation in turn (see Table 1 for a summary).

3.5.1 Only Cost, Only Penalty, or Only Budget

The simplest formulations for the RPRO include only one component, either only cost, only penalty, or only budget.

Optimizing for *Cost* only. This formulation simply consists of the minimization of the cost model described in Section 2.3 with no constraints. Since there is no instruction from the user as to what type of partial results are better than others, an empty *Incomplete* result is just as good as a *Complete* one. Therefore, the logical optimization for our optimizer is to *scan nothing* from the data sources (i.e., trigger the *drop* mechanism,) resulting in a cost of zero.

Optimizing for *Penalty* only. A naïve alternative to the *Cost*-only formulation is to minimize the penalty on the plan. Here the opposite problem occurs. Without anything to temper the cost of computation, in the event of a data access failure, the optimizer may continuously trigger the *retry* mechanism in an attempt to produce a *Complete* and *Credible* partial result. This may result in extreme (or infinite) cost, and the query may never complete.

Only subject to *Budget*. To combat any possibility of infinite cost, the RPRO may be instructed to adhere to a budget constraint on the cost of processing the query without minimizing either total cost or penalty. In this case, as long as sufficient budget remains, the optimizer will not invoke any of the partial result mechanisms, since the formulation does not distinguish between partial results and complete results. However, once the budget is exhausted, if the query is not yet complete, the optimizer will trigger *drop* across all partitions in all remaining (active) pipelines, returning whatever result is available. This means that the results produced may be useless, since the point of budget exhaustion is likely to be arbitrary.

3.5.2 Cost/Penalty, Cost/Budget, or Penalty/Budget

Due to the downsides of the one-component formulations, here we consider formulations with two components.

Optimizing for *Cost* & *Penalty*. The main pitfall to optimizing solely for *Cost*, was that there was no counterweight to incurring no cost and producing an empty tuple set. In this formulation, the RPRO will not attempt to improve a partial result classification if the penalty for doing so is less than the estimated cost of retrying. However, as there is no budget constraint, the overall cost of the query may be quite high if the number of retries is large, and consequently this formulation may result in expensive partial results.

Optimizing for *Cost* & *Budget*. Including the *budget* constraint to the cost model allows the user to specify the maximum acceptable cost for producing the result. As with the *Cost*-only formulation, without the *penalty* component in the objective function, the optimizer will always find it preferable to return a low-cost, *Incomplete* (empty) result set.

Optimizing for *Penalty* & *Budget*. To round out the two-component formulations, we consider the formulation that minimizes the partial result *penalty* while adhering to a cost budget. While this objective function does not have the unbounded cost and poor partial result classification limitations of the previous functions, it has the problem that the RPRO will always *retry* pipelines up until the budget is met. Without the *cost* component, the optimizer has no concern of how much it will cost to *retry* the pipeline, and as a result, it is likely that the optimizer with this formulation will exhaust its budget in a manner similar to the *Budget*-only formulation.

3.5.3 Cost, Penalty, and Budget

The main optimization problem that we propose to use for the Runtime Partial Result Optimizer is one that includes all three components: *cost*, *penalty*, and *budget*.

$$\min\{\mathrm{cost}(query) + \mathrm{penalty}(plan) \mid \mathrm{cost}(query) \leq B\} \quad (2)$$

In our prior discussion, we listed a number of (undesirable) strategies that the system may take when optimizing for a problem with fewer components. We saw that, without all three components, the logical strategies resulted in unbounded cost or poor quality results. Here, we have put all three components together, ensuring that a

Runtime Partial Result Optimizer tasked with this objective function simultaneously optimizes for better quality results (by minimizing penalty) while simultaneously minimizing cost and remaining below a specified budget limit.

With this three-component formulation, we force the RPRO to balance the cost of processing the query against the quality of the partial result classification with a maximum cost bound for the query. In this paper, we don't consider hard constraints on the quality of the partial result or a soft cap on the cost bound. Both of these alternatives introduce very interesting possibilities, but we leave such problem formulations as future work.

3.6 Implications and Open Questions

The failure-tolerant nature of the system, coupled with the budget-based compute constraint means that our proposed system behaves very differently from traditional and approximate query systems. Consider the fact that different partitions of the input data (and output result) may receive dramatically skewed amounts of processing "effort" (i.e., computational resources.) For a single query, it may be possible for the system to fully process some input partitions without any failures and then encounter a failure on some problematic partition and *retry* without success until the budget is exhausted. Furthermore, with the *drop* and *no-op* mechanisms, different partitions of an input relation may not even be processed by the same execution plan due to failures and the decisions of the Runtime Partial Result Optimizer. Finally, there may be circumstances where, even though no access failures occurred while reading the base relations for the query, in the end, the RPRO may still cause the system to return a partial result due to the user's penalty model and the resulting lower cost for the query.

There are many open questions requiring further exploration:

- If the query is parallelizable over its input data, do we process all of the input partitions simultaneously or do we process them serially? If we estimate the possibility of failures and the prospect of running out of budget, then there is a potential difference in the runtime partial result optimization under concurrent processing versus serially processing partitions.

- The policy with which we allocate the query budget is also flexible. We may process the query such that the *Cost* is drawn from the budget pool in a straight-forward manner until the entire pool is exhausted. On the other hand, another way to manage the budget is to divide it "vertically" amongst the partitions in some fashion so that no partition can "starve" another partition from compute resources due to failures and continuous *retries*. Conversely, we can divide the budget "horizontally" across different pipelines of left-deep query plans so each relation in the query gets a fair share of the budget for *retries*. These options lend themselves to the notion of "local" optimization vs "global" optimization.

- How does the RPRO scale when it needs to control large-scale data processing across large clusters that run concurrent queries? Given that the RPRO will need to gather runtime information from all of the system nodes as it performs optimizations on a per-query basis, this will be a significant bottleneck. One way of load balancing this optimization work across the cluster is to assign a different cluster node the RPRO responsibilities (and pipeline dispatching duties) for every query. Other means of sharing these RPRO duties may include communication strategies where the RPRO node receives runtime information from the cluster in a tree-topology fashion.

4. RELATED WORK

The related work covers many different areas of database research such as partial answers [4], online aggregation [6, 14], dynamic re-optimization for performance [9], multi-objective query optimization [20], and cost optimization in the cloud [11, 12, 16]. Other types of "partial" querying and result quality that we may consider include (in)consistent querying [2], optimization and querying over sampled data [1, 5, 7, 18], uncertain querying [3], and probabilistic querying [15].

The high-level differentiator of our work and the contribution of this paper is the combination of the cloud pay-as-you-go paradigm with the return of partial results that may not be the same as those produced by a traditional on-premise database system. As we described in Section 3.6, there are many open opportunities for further exploration and we believe that the various areas of the state-of-the-art are complementary to our proposed system.

5. CONCLUSION

In this paper, we described a new problem of optimization in the cloud to better serve users that are willing to receive results with weaker quality guarantees in exchange for a lower cost. We propose that a DBaaS can provide this feature through the use of a new Runtime Partial Result Optimizer that utilizes three novel mechanisms for modifying the execution of pipelines and operators to manage the balance between cost and quality. This paper represents an initial discussion of this idea, but we have only touched the surface of a rich area for future work. We expect that deeper research is needed across many facets of this type of system, including issues such as alternative notions of result quality and partiality, traditional (compile-time) query plan optimization, more sophisticated failure models, more sophisticated and nuanced penalty modeling, and local vs. global optimization policies.

6. REFERENCES

[1] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *EuroSys*, 2013.

[2] M. Arenas, L. Bertossi, and J. Chomicki. Consistent Query Answers in Inconsistent Databases. In *PODS*, 1999.

[3] O. Benjelloun, A. Das Sarma, A. Halevy, M. Theobald, and J. Widom. Databases with Uncertainty and Lineage. *VLDB Journal*, 2008.

[4] P. Bonnet and A. Tomasic. Partial Answers for Unavailable Data Sources. *INRIA Technical Report*, 1997.

[5] P. B. Gibbons and Y. Matias. New Sampling-Based Summary Statistics for Improving Approximate Query Answers. In *SIGMOD*, 1998.

[6] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online Aggregation. In *SIGMOD*, 1997.

[7] Y. Hu, S. Sundara, and J. Srinivasan. Supporting Time-constrained SQL Queries in Oracle. In *VLDB*, 2007.

[8] M. Jarke and J. Koch. Query optimization in database systems. *ACM Comput. Surv.*, 16(2), 1984.

[9] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD*, 1998.

[10] W. Lang, R. V. Nehme, E. Robinson, and J. F. Naughton. Partial Results in Database Systems. In *SIGMOD*, 2014.

[11] W. Lang, S. Shankar, J. Patel, and A. Kalhan. Towards multi-tenant performance slos. In *ICDE*, 2012.

[12] Z. Liu, H. Hacigumus, H. J. Moon, Y. Chi, and W.-P. Hsiung. Pmax: Tenant placement in multitenant databases for profit maximization. In *EDBT*, 2013.

[13] H. Mendelson and A. N. Saharia. Incomplete information costs and database design. *ACM Trans. Database Syst.*, 11(2), 1986.

[14] V. Raman and J. M. Hellerstein. Partial Results for Online Query Processing. In *SIGMOD*, 2002.

[15] C. Ré and D. Suciu. Approximate Lineage for Probabilistic Databases. In *VLDB*, 2008.

[16] J. Schaffner, T. Januschowski, M. Kercher, T. Kraska, H. Plattner, M. J. Franklin, and D. Jacobs. Rtp: Robust tenant placement for elastic in-memory database clusters. In *SIGMOD*, 2013.

[17] S. Shankar, R. Nehme, J. Aguilar-Saborit, A. Chung, M. Elhemali, A. Halverson, E. Robinson, M. S. Subramanian, D. DeWitt, and C. Galindo-Legaria. Query Optimization in Microsoft SQL Server PDW. In *SIGMOD*, 2012.

[18] M. A. Soliman, I. F. Ilyas, and S. Ben-David. Supporting Ranking Queries on Uncertain and Incomplete Data. *VLDB Journal*, 2010.

[19] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: A wide-area distributed database system. *VLDB Journal*, 5(1), 1996.

[20] I. Trummer and C. Koch. Approximation Schemes for Many-objective Query Optimization. In *SIGMOD*, 2014.