Peregrine: Workload Optimization for Cloud Query Engines

Alekh Jindal, Hiren Patel, Abhishek Roy, Shi Qiao, Zhicheng Yin, Rathijit Sen, Subru Krishnan
{firstname.lastname}@microsoft.com
Microsoft

ABSTRACT

Database administrators (DBAs) were traditionally responsible for optimizing the on-premise database workloads. However, with the rise of cloud data services, where cloud providers offer fully managed data processing capabilities, the role of a DBA is completely missing. At the same time, optimizing query workloads is becoming increasingly important for reducing the total costs of operation and making data processing economically viable in the cloud. This paper revisits workload optimization in the context of these emerging cloud-based data services. We observe that the missing DBA in these newer data services has affected both the end users and the system developers: users have workload optimization as a major pain point while the system developers are now tasked with supporting a large base of cloud users.

We present Peregrine, a workload optimization platform for cloud query engines that we have been developing for the big data analytics infrastructure at Microsoft. Peregrine makes three major contributions: (i) a novel way of representing query workloads that is agnostic to the query engine and is general enough to describe a large variety of workloads, (ii) a categorization of the typical workload patterns, derived from production workloads at Microsoft, and the corresponding workload optimizations possible in each category, and (iii) a prescription for adding workload-awareness to a query engine, via the notion of query annotations that are served to the query engine at compile time. We discuss a case study of Peregrine using two optimizations over two query engines, namely Scope and Spark. Peregrine has helped cut the time to develop new workload optimization features from years to months, benefiting the research teams, the product teams, and the customers at Microsoft.

CCS CONCEPTS

• **Information systems → Query optimization**; **Relational parallel and distributed DBMSs**; • **Computer systems organization → Cloud computing**.

ACM Reference Format:
Alekh Jindal, Hiren Patel, Abhishek Roy, Shi Qiao, Zhicheng Yin, Rathijit Sen, Subru Krishnan. 2019. Peregrine: Workload Optimization for Cloud Query Engines. In *ACM Symposium on Cloud Computing (SoCC '19), November 20–23, 2019, Santa Cruz, CA, USA*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3357223.3362726

SoCC '19, November 20–23, 2019, Santa Cruz, CA, USA

ACM ISBN 978-1-4503-6973-2/19/11…$15.00
https://doi.org/10.1145/3357223.3362726

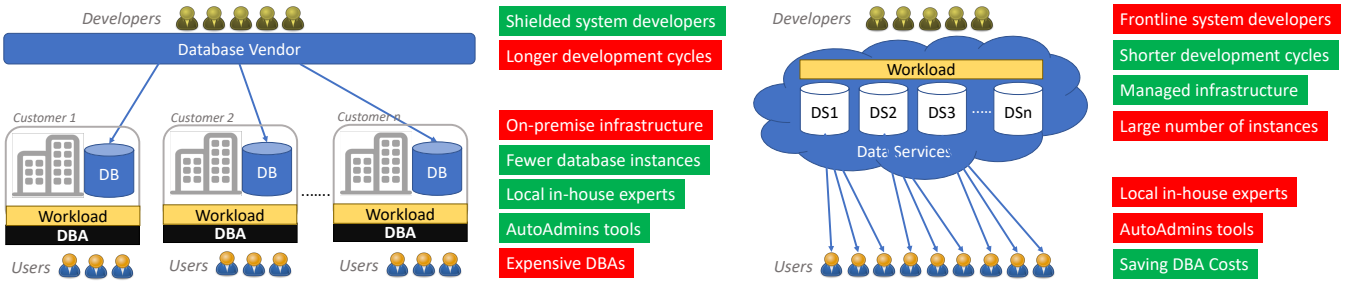
1 INTRODUCTION

Database administrators (DBAs) have long carried the burden of managing database systems. They are the indispensable resources for installing, maintaining, and tuning the database systems of an organization, with business relying on them for their critical operations. Traditional DBAs were domain experts who relied on their experience and best practices to manage the databases. However, vendors and practitioners soon realized the effort and costs associated with manual database administration. This led to the rise of auto-admin tools [22, 26, 28, 29], consisting of algorithms and wizards, to assist in database administration. Auto-admins were client-side tools that used heuristics and other estimates for the tuning decisions, and often involved human DBAs in the loop.

While DBA has primarily been a concern for the database customers, the rapid rise of cloud infrastructures has turned the tables. Databases are now increasingly offered as fully managed services in the cloud, also referred to as *data services*, e.g., Athena [24], BigQuery [25], and ADLA [5], where users simply point to their data and start querying immediately. This is desirable for both the customers and the cloud providers, since the customers are no longer responsible for database installation and maintenance, while the cloud providers can continuously improve the upgrade and tooling experience. Figure 1 contrasts the traditional on-premise databases with the new cloud data services. While the on-premise databases relied heavily on the presence of DBAs to manage and tune databases, unfortunately, the role of a DBA for performance tuning is completely missing in the cloud data services. At the same time, tuning end to end workloads, aka *workload optimization*, is crucial in the cloud environments for reducing the total cost of operations. Lack of a DBA coupled with a lack of user-control in the cloud data services, has caused workload optimization to be a major challenge in these data services.

Interestingly, the cloud data services also open up several new opportunities. First of all, the popularity and ease of cloud services have increased the workload traces available to the cloud provider[1]. This opens up a huge opportunity to leverage these workloads for performance tuning. Second, the cloud providers can not only analyze the massive query workloads, but they can also apply a feedback loop to their managed data services, since the services are under their control anyways. Such self-tuning mechanisms are needed to not only improve customer experience but to also make the life of system developers easier. Third, it is often hard to get the performance tunings right in the first shot and so the cloud provider can try different things or even learn the best tunings across multiple customers, all in one central place. Fourth, given the fast pace of releases and updates in the cloud services, the cloud provider can provide newer performance tunings in a shorter turnaround time. Finally, all major cloud providers offer multiple

[1]These workloads could be anonymized traces, or the logs could be accessed only within privacy boundaries of the customer.



(a) On-premise Databases (b) Cloud Data Services

Figure 1: Comparing on-premise with cloud data services; observation: the role of a DBA for performance tuning is missing in the cloud data services.

query engines and the need for analyzing query workloads and applying feedback actions is common across all of them. Thus, there is an opportunity to build a common platform that could be leveraged across multiple cloud query engines.

Contributions. Motivated by the above observations, in this paper, we present Peregrine, a workload optimization platform for cloud query engines. The goal of Peregrine is to: (i) make it easy to analyze query workloads and build workload optimization features (or *apps*) using them, (ii) define the space of workloads optimizations that is relevant to typical production workloads, and (iii) provide design patterns to add workload awareness to the cloud query engines. In summary, our key contributions are as follows:

(1) We illustrate the problem of a missing DBA from the Big Data analytics infrastructure at Microsoft. We describe the need for workload optimization from our customer experiences and discuss the implications of not having a DBA function in cloud data services. (Section 2)

(2) We present Peregrine, a platform that provides *dba-as-a-service* to fill the workload optimization gap in cloud data services, an engine-agnostic workload optimization layer that can be developed and scaled independently, and a global optimization framework to help cloud users reduce their total costs of operations. (Section 3)

(3) We introduce a novel way of representing query workloads that is agnostic to the specific query language and general enough to describe a large variety of workloads. A workload thus described can generate intermediate representations (IRs) that can be used for quickly building workload optimization features later on. (Section 4)

(4) We present a categorization of the typical workload patterns, derived from production workloads at Microsoft, and discuss their corresponding workload optimizations possible in each category, thus providing a good starting point to researchers and practitioners for building workload optimization applications. (Section 5)

(5) We describe providing workload feedback to the query engines, via the notion of *query annotations*. We discuss how the query annotations are served from a feedback service and how the query engines are enabled to act on the workload feedback. Together, these add workload awareness to a query engine. (Section 6)

(6) We present a detailed case study on how Peregrine helped in (i) building multiple workload optimizations, namely CloudViews [41] and CardLearner [64], for the Scope query engine, and (ii) building the CloudViews optimization for multiple query engines, namely Scope [67] and Spark. (Section 7)

(7) Finally, we discuss the road ahead for Peregrine, particularly in terms of extending it along various dimensions, including more optimizations and query engines, and improving the end to end deployment experience. (Section 8)

2 THE MISSING DBA IN THE CLOUD

We now illustrate the problem of missing DBA using the big data analytics infrastructure at Microsoft, which is used across the whole of Microsoft for business units such as Bing, Office, Windows, Xbox, Skype, etc. Microsoft's big data infrastructure consists of hundreds of thousands of machines and uses Scope as the primary query engine. The Scope job service processes hundreds of thousands of analytical jobs authored by thousands of developers across Microsoft. These jobs process massive volumes of data, several exabytes in total across all of the jobs, to analyze and improve different Microsoft products. To do this, Scope exposes a SQL-like declarative query interface, where users specify their business logic and the system automatically figures out how to execute that in a distributed environment. Essentially, this means that the users don't have to worry about provisioning any machines (aka *serverless*) and focus only on their data processing logic at hand. Therefore, it is easy to see how such a managed cloud data service can offer a tremendous business proposition.

The size and volume of the Scope job service infrastructure makes workload optimization critical. This is because even few percentage point improvements could lead to millions of dollars of savings in operational costs. At the same time, by being a managed service, Scope users have little control over the query processing infrastructure. Consequently, a lot of prior research has looked at automatic tuning the performance of Scope query engine [21, 41]. Scope query engine further exposes a number of tuning knobs, including dozens of hints at the script, data, and plan level. However, only a small pool of Scope users are able to leverage these hints while a large fraction are non-experts who need help on a regular basis. In fact, our internal survey shows customers asking for better

tooling and support to improve their Scope queries. Thus, workload optimization remains a major pain point for Scope customers.

The above customer pain often ends up as support requests or incidents reports to the service provider. Our analysis over the incident management data from the Scope job service reveals tens of thousands of incidents per year with almost ten times the number of incidents as the number of system developers on support calls at any time, and more than hundred times the number of users as the total number of system developers. Scope business leaders are further keen to improve performance in order to reduce the total cost of operations. In fact, there is an ongoing effort to reduce the total processing time by several percentage points year over year. The situation becomes worse when new query engines are added, e.g., Microsoft's big data infrastructure now also supports Spark query engine, which has very different performance characteristics, in addition to Scope. Not to mention the growth of the workload on existing ones, e.g., approximately ten percent increase in the number of Scope jobs in the first quarter of this year (2019). Therefore, it is not possible for system developers to cope up with the increasing support load.

Although we have described the problem of missing DBA in the context of the Scope job service infrastructure above, it is easy to imagine a similar situation for other cloud services. In fact, our internal discussions at Microsoft reveal very similar pain points in other parts of Azure as well. In summary, the lack of a DBA has the following key implications for cloud data services:

1. Performance tuning becomes harder for cloud data services. The new class of big data systems, with their declarative query interfaces, are quite complex and require a lot of tuning for good performance. Furthermore, the cloud infrastructures add more dimensions by allowing resources to be provisioned dynamically and on demand, thereby making performance tuning even harder.

2. Reducing costs becomes critical with managed cloud services. Cloud data services users are typically billed in a pay-as-you-go manner and so performance efficiency is important for lowering the dollar bill. Given that reducing the operational costs is often a key motivation for enterprise customers to move to the cloud, higher dollars costs may simply not be acceptable.

3. Lack of user control. Managed data services do not provide much control to the users. This is for a variety of reasons, including privacy and security, impact on other customers due to multi-tenancy, and most importantly for deliberately abstracting away the low-level internal details for the ease of use. However, by not having much control, users cannot tune performance even if they had the expertise to do so. This puts the onus of performance tuning squarely on the cloud providers.

4. Long tail of non-expert users. Given the ease of spinning up database instances in the cloud, managed data services have attracted a wide variety of users, many of whom having little or no expertise. Typically, in the absence of a DBA, the non-expert users rely on self-help through discussion forums and other support groups to resolve their performance related issues. However, many of these issues end up being repetitive or even trivial, and unless these forums are monitored and curated with the accurate information, they may end up confusing people.

5. System developers are the new virtual DBAs. Finally, managed services make the lives of system developers harder. They routinely see performance issues popping up as service incidents or support requests and need to handhold the users for their performance tuning tasks. This is counter-productive and often frustrating for the system developers since it eats up a lot of their precious feature development time. It is also not a scalable situation since *the number of cloud users far outnumber the number of system developers*, and it is not possible for the system developers to cope up with the deluge of incident reports or support tickets.

3 WORKLOAD OPTIMIZATION PLATFORM

Cloud computing is changing the way users interact with databases, i.e., via data services, and therefore we need to reinvent the DBA function for this data services world. We see the following key requirements in doing so:

DBA-as-a-service. Given the service-oriented approach in the cloud, offering DBA as a service is a natural extension of the DBA to this new data services world. This makes sense since cloud providers are responsible for tuning their managed data services. They also have the unique advantage of observing workloads in their cloud infrastructure, and deriving customized tuning decisions for each workload that are likely to be far more efficient and reliable.

Engine-agnostic. Workload optimization needs to be agnostic to the underlying query engine so that it could be developed and scaled independently, the cloud providers can apply the tuning decisions based in a fast feedback loop, and optimizations built once could be used multiple times across several engines. Therefore, it is crucial to not tie workload optimization to any specific query engine.

Global optimization. Workload optimizations need to have a global view of the workload, i.e., consider the logs and the metadata across the end to end data processing pipeline, in order to make global optimizations decisions that are crucial for reducing the total cost of operations, which is also a primary motivation for workload optimization in the first place.

To address the above requirements, we present the Peregrine workload optimization platform. Figure 2 shows the high-level architecture of Peregrine. It consists of three major components.

(i) Workload Representation. A workload representation component, shown just below the query engines in Figure 2, takes as input the anonymized logs from the query engine as well as related runtime information from the underlying platforms (job scheduler, job manager, resource manager, storage service, etc.). The output is one or more intermediate workload representations that are common across workloads and query engines. These intermediate representations are then input to the engine-agnostic optimization algorithms. Workload representation alleviates much of the pain associated with parsing and interpreting the raw query logs and turning them into actionable data.

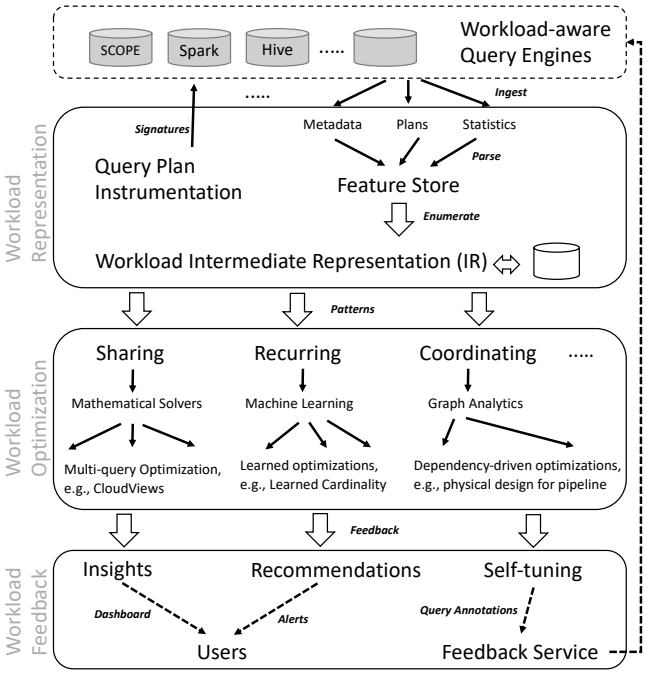


Figure 2: The Peregrine architecture, consisting of the workload representation, the workload optimization, and the workload feedback layers.

(ii) Workload Optimization. A workload optimization component, shown in the middle in Figure 2, mines the query workload for interesting patterns and runs optimization algorithms to tune those patterns. Identifying patterns and optimizing for them makes workload optimization more practical and less open-ended. Currently, Peregrine focuses on the three typical class of patterns derived from our production workloads at Microsoft, however, more patterns could be easily added. Each of these pattern classes enable several workload optimizations.

(iii) Workload Feedback. Finally, a workload feedback component shown at the bottom in Figure 2, collects the output of the optimization algorithms and converts them into actionable feedback that could be either consumed by the users in the form of insights and recommendations or fed back to the query engines for self-tuning. For self-tuning, the feedback is encoded as query annotations, loaded onto a feedback server, and requires some (minor) query engine changes for taking actions (Section 6).

The key strengths of the Peregrine platform are as follows. First, it provides a clean interface (only need to understand the query logs) which could easily be implemented for different query engines. Second, it provides an extensible infrastructure wherein more instrumentation, parsers, patterns, optimizations, and feedback could be added based on evolving workload needs. Third, it provides library implementations of each of the above which act as the starting point for covering several common scenarios. And finally, it has multi-faceted endpoints (insights, recommendations, feedback) which can cater to different user expectations.

In the rest of the paper, we discuss the above three components of Peregrine, namely the workload representation, optimization, and feedback. Thereafter, we present a case study and discuss the road ahead with Peregrine.

4 WORKLOAD REPRESENTATION

A core capability of Peregrine is to allow developers to describe query workloads in an engine agnostic manner. Workloads thus described could be then optimized using a common set of algorithms. This is useful because workloads from different query engines need to be described only once, taking away the pain of parsing, processing, and wrangling the low-level system logs over and over again for every new optimization. Below we first describe query plan instrumentation for capturing and subsequently affecting different traits of a query plan. Then we describe the feature store to collect the universal set of all features from the query plans. Finally, we present a query workload IR for efficiently running workload optimization algorithms on top.

4.1 Query Plan Instrumentation

Query optimizers have long been responsible for making the optimization decisions in a query plan, including multi-query optimization. However, with optimizers becoming increasingly complex, the preferred choice is to make minimal deep changes in the system, i.e., instead of in-lining new tuning features within the query optimizer, offload them to an external component that could be developed, managed, and scaled independently. Peregrine fills precisely this gap: it can run expensive optimizations over large workloads offline and provide hints for optimizing future queries. However, Peregrine still needs to be aware of the query plan traits in order to make optimization decisions outside the query optimizer. Most query engines log the query plans, so one option could be to reconstruct the optimizer internal state, e.g., the memo for a Cascade optimizer [36], using the query plan logs. However, this reconstructed state could be lossy since the optimizer states are not dumped entirely for practical reasons (e.g., there could be thousands of columns and so the column reference list might be pruned to reduce the logging overhead) or security concerns.

Peregrine provides an instrumentation mechanism for capturing query plan traits and logging them as *signatures* in the query logs. For each node in the query plan, the signatures capture the internal optimizer state, corresponding to different query plan traits (e.g., operator names, attribute references, required properties, etc.), into fixed sized hashes and output them as part of the query logs. These signatures also ease some of the featurization pain in the high dimensional plan space, e.g., for user-defined operators that may end up with lots of one-hot encoded features. Furthermore, they could be analyzed externally to generate optimization hints when similar query traits are seen in the future. Capturing signatures as part of query plans and taking actions based on the signatures in future queries allows Peregrine to externalize workload optimization with minimal changes to the query optimizer. Signatures could be of different types to capture different query plan traits, e.g., capturing only the operators, capturing the entire subexpression rooted at each operator, capturing physical properties, etc. We have implemented multiple such signatures for Scope query plans, they

could be composed to identify combined traits, and they could also be used across multiple engines. In summary, signatures are hash identifiers of different granularities for each node in the query plan that (i) capture the query subexpressions and related properties from the optimizer's internal memo, (ii) have a cheap and convenient way to featurize query plans into a flat structure, (iii) reduce the dimensionality of the workload, e.g., user-defined operators that are very difficult to featurize otherwise, (iv) use them to cluster subqueries with similar characteristics (more on this in Section 5), and (v) leverage them for invoking query optimization rules once the feedback is provided (more on this in Section 6). While lightweight, signatures are currently limited to equality comparisons and do not support more advanced containment checks. Likewise, plan properties cannot be reconstructed from the the signatures. Exploring some of these would be part of future work.

Finally, Peregrine makes it easy to extend the signatures to capture more kinds of query plan traits for newer optimization scenarios. The signature API takes a plan subexpression as input and returns the list of signatures as output.

4.2 Feature Store

We need to parse and analyze the features from the past query logs before we can optimize query workloads. Unfortunately, however, parsing and transforming query logs is a significant effort since developers typically start analyzing the logs from scratch for every new optimization. Therefore, one of our first goals in Peregrine is to make this process easier by creating a common data platform, on top of the query logs, which can be used to collect the universal set of features and quickly build the optimizations on top. It consists of two steps: (i) an ingestion step to collect the logs from different query engines, and (ii) a parsing step to transform the logs into a common set of features. We describe these two below.

The ingestion pipeline starts with query workload logs from different query engines. These logs are anonymized as mandated by the specific scenario, for example, whether the logs are stored within the cluster or in an outside service. Peregrine provides connectors to typical cloud-based storage services, such as Azure Data Lake Storage [57], Azure Data Explorer [4], and Azure SQL DB [8]. The ingestion layer can also access the raw logs directly for standalone deployment. Collecting and persisting the workloads logs into a persistent store is an important step for enabling analysis over the past workload.

Once the workload logs are accessed, we need to extract the relevant entities from the anonymized query logs for further analysis. There are three pieces of information that we mostly care about: (i) *query metadata*, including flags and parameters provided with the query, user and account names, query submit, start, and end times, available resources, etc., (ii) *query plans* including the logical (input, analyzed, optimized), the physical, and the execution plans for the query, and (iii) *runtime statistics* from the executed data flow, including row counts, latency, CPU time, I/O time, memory usage, etc. Since these get generated at different points in query execution, they may be present in different places in the workload log and even in different log files. We collect this query information from specific subsets of the log and call it the query traces.

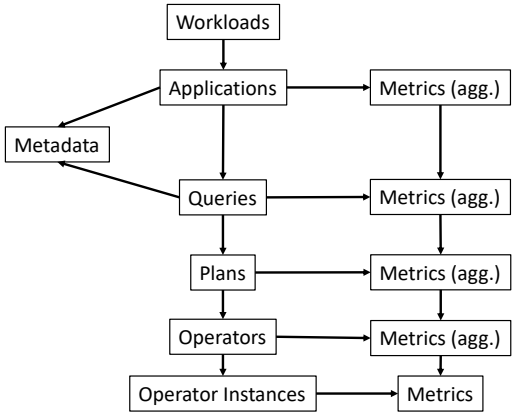


Figure 3: Example entities in the feature store.

Finally, we parse the query traces from the ingestion pipeline. For this, Peregrine provides a set of parsers to parse the query metadata, plans, and runtime statistics from various query engines in different formats, e.g., JSON, XML, or plain text. We need a new parser for each engine but can share much of parsing code for different formats of the same engine, e.g., parsing an operator node might be common. In any case, we believe that the combination of engines and formats will only be a small finite set of parsers that could be leveraged repeatedly. The parsers output a set of common workload features along with the relationships, as illustrated in Figure 3. A query workload can consist of multiple applications, each with metadata, metrics, and multiple queries. Queries can have one or more plans each with a set of operators. Multiple instances of an operator run in a distributed setting and the metrics collected by the application link to each operator instance. Query engines such as Scope and Spark pre-aggregate metrics across operator instances, e.g., average latency of an operator. We can further aggregate at the metrics for plans, queries, or the application level.

The feature store described above takes care of preprocessing the workload logs, which is often the biggest pain point in optimizing query workloads and requires considerable effort. The feature store also has an extensible design to add more query engines, extract other pieces of information from the log, add new parsers for custom formats, and add newer workload features as they emerge. Still, we expect the set of customizations to be limited and hence we could share many of the pieces for a new data service and save considerable effort.

4.3 Workload IR

The entities shown in Figure 3 are quite descriptive, however, we still need to generate an efficient representation for running optimization algorithms. Therefore, Peregrine allows creating more efficient intermediate representations (IRs), which generalize across query processors, and could be used to run various optimization algorithms on top. Figure 4 shows three such denormalizations, the plans IR, the subexpressions IR, and the operator instances IR, which de-normalizes the workload entities for more efficient processing later on. Table 1 illustrates a subexpressions IR, where each row is a subexpression with associated attributes and metrics.

QueryID	Signature	Operator	EstCardinality	RowLength	EstSubexprCost	EstOperatorCost	PartitioningType	Partitions	Cardinality	SubexprRuntime	OperatorRuntime
1	155	RangeScan	1837160	251	104.736	104.736	Range	2	15891	3.203	0.016
2	155	Exchange	1837160	251	743.709	638.973	RoundRobin	250	15891	3.203	0.016
3	326	UDF	1837160	251	744.077	0.368	RoundRobin	250	15891	66.966	55.16

Table 1: Illustrating an subexpressions IR instance. Each row in the IR corresponds to one subexpression in one of the queries. The columns include query level attributes (only QueryID here), compile-time attributes (e.g., Estcardinality, RowLength, etc.), and run-time attributes (e.g., Cardinality, OperatorRuntime, etc.).

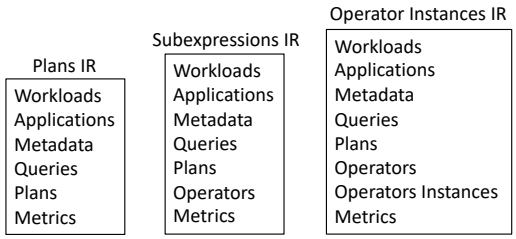


Figure 4: Example denormalized IRs.

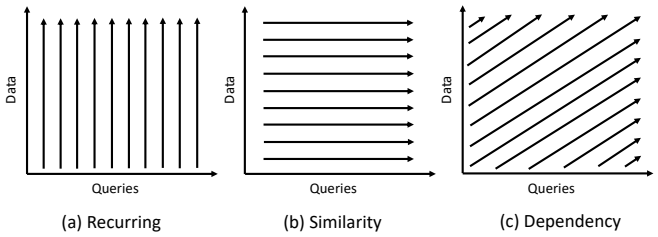


Figure 5: Typical patterns observed from production workloads at Microsoft. The lines correspond to the pattern direction, e.g., vertical lines indicate that recurring queries are same queries templates running over different datasets, similar queries are ones having overlaps over same datasets, while depending queries consume the outputs of queries over previous dataset instances.

Applications can create and use on or more of these IRs depending on the granularity of the information they need. IRs could be stored in various storage backends, using the same set of connectors as used for accessing the query workload logs, and are very useful for quickly drawing insights and building optimizations over a query workload. Once created, IRs could also be shared across multiple workload optimization applications. In particular, researchers at Microsoft have found subexpressions IR to be useful in a number of applications, including finding subexpressions to materialize [40], learning cardinalities over recurring workloads [64], and mining physical design hints. IRs further generalize across query engines as well, e.g., researchers have used the same subexpressions IR for both Scope and SparkSQL.

Thus, Peregrine helps encapsulate all engine specific knowledge from the query workloads and exposes only the generalized IRs to the optimization algorithms, in order to speed up the time to build workload optimization features. We discuss these workload optimizations below.

5 WORKLOAD OPTIMIZATION

Workload optimization is very broad topic and there could be numerous ways to optimize query workloads. To make the problem space more tractable, below we first identify the key workload patterns from production workloads at Microsoft. Then, we describe the class of optimizations that we have identified and the tools that could be applied for each of these workload patterns.

5.1 Patterns

We now discuss the key patterns that we have identified from the Scope job service workloads. Given that Scope job service runs analytics over all of the products at Microsoft, such as Bing, Office, Windows, Skype, XBox, etc., we believe it constitutes a fairly representative mix of enterprise data analytics. Figure 5 depicts the three major workload patterns that we have identified. The points on horizontal axis represent a query template that has to be provided with specific dataset versions, while the points on vertical axis represent the changes in data with time. Therefore, each execution of the query is captured by a point on the graph. The lines on the graph indicate one of the multiple relationships between queries and data and between queries themselves.

Recurring. Production workloads are often repetitive, i.e., same queries executed periodically with new inputs and parameters. To illustrate, more than half of big data workloads at Microsoft consists of recurring queries [43]. Such recurring queries often do bulk of the preprocessing to generate cooked data, which could then be used for final results. The left plot in Figure 5 visualizes the recurring queries as vertical bars, i.e., same queries executing over newer versions of data. Recurring queries indicate the predictive nature of the workload, thereby allowing us to analyze the past workloads and inferring the future ones.

Similarity. In addition to recurring, queries in the same recurring intervals are also often similar. The horizontal bars in Figure 5 represent the similarity between different queries on the same version of data. This similarity is because the queries are written by multiple users who access the same sets of inputs in the cloud infrastructure. A typical similarity pattern is common subexpressions or overlaps across queries. More than half of production big data queries at Microsoft have overlaps with at least one other query [41]. The similarity workload patterns indicate the interactions within the workload and therefore we can build multi-query optimizations to optimize this interaction.

Coordinating. Finally, production workloads often consist of data pipelines where the output of one query (from the previous recurring interval) is consumed by a subsequent query (in the subsequent recurring interval). Such dependencies introduce SLA requirements

and are crucial to be aware of while optimizing. We depict such dependencies across data instances using diagonal bars in the right most plot in Figure 5. Dependency patterns indicate the connectivity in the workload and gives rise to a whole new set of dependency driven analytics [50].

The above workload patterns could be leveraged to enable a large set of workload optimizations. We describe some of these below.

5.2 Optimizations

We identified three broad classes of optimizations, one for each of the workload patterns describe above. We refer to them as *learned*, *multi-query*, and *dependency-driven* optimizations respectively. Below we describe these classes with several specific examples in each one of them.

5.2.1 Learned Optimizations. Recurring workload patterns provides opportunities to learn from the past, i.e., build models over it, and apply them again in the future. Some examples in this direction include topics around learning optimizer, which has gained a lot of traction lately. For instance, cardinality estimation, i.e., the size of the intermediate outputs at each point in a query plan, is to key picking a query plan in a query optimizer. Unfortunately, however, cardinality estimation remains the Achilles heels of query optimization [46] due to inaccurate operator selectivities, correlations between different portions of the data, and exponential error propagation up the query DAG. To address this problem, our recent work analyzes past Scope query workloads and learns cardinality models for similar subexpressions seen in the past [64]. We implemented this system using Peregrine. Others have looked at building more general models for range predicates [33, 65]. Going a step further, some have even proposed to directly pick the query plans using neural networks [48, 54]. Recurring workloads could help in learning such models.

Performance prediction is another example where recurring query workloads could be exploited. In fact, a lot of work has been done on prediction the performance (latency, throughput, etc.) of a query workload [23, 35] and improving job predictability [43]. Going further, we used Peregrine to learn the cost models that Scope query optimizer uses to pick the best physical execution plan, i.e., the plan with cheapest cost [59]. This is useful because cost models try to estimate the performance of a query plan which is very difficult in a distributed big data system. Existing cost models use heuristics to model the most important queries and are often way off for other queries. Therefore, instead of trying to build the perfect analytical model to estimate query costs, the past recurring workloads could be analyzed to learn cost models that predict the costs for future queries.

Cloud computing allows to pick resources dynamically, however, picking the right set of resources is a challenge. Recurring patterns are useful to build resource models from past workload and use them for future prediction [45, 56]. Learning resource models is further useful for combining query and resource optimization. Current big data query processing systems first choose a query plan and then pick the resources (number of containers, size of containers, etc.) required to run the plan. However, the choice of query plan itself depends on the resources used, e.g., hash join will be better with small number of large containers, while sort-merge join will do better with exactly the opposite. It turns out that resource-aware query planning can results in plans that are twice as fast and yet consume half the resources [63]. Analyzing recurring workloads and providing it as a feedback to the query engine helps make resource-aware decisions [59].

Finally, there is a recent trend of building machine learning models for data distributions and using them for more efficient data structures, e.g., indexes [44], bloom filters [52], etc. Again, recurring workloads nicely supports learning such data distributions.

5.2.2 Multi-query Optimizations. The similarity pattern we see in Figure 5 is an obvious candidate for multi-query optimizations. The overlapping computations across queries result in the same partial computations being executed multiple times, thereby leading to a significant wastage of compute resources. Analyzing the query workloads and creating materialized views to reuse overlapping computation across queries could lead to significant resource savings and performance improvements. CloudViews, one of the first features built using the Peregrine platform, performs automatic computation reuse in the Scope and HDI [6] environments [40, 41, 58]. Multi-query optimization has been extensively research and several other approaches could be applied to cloud data services, e.g., MRShare [53], PigReuse [27], etc.

Apart from reusing common subexpressions, the similarity patterns provides significant opportunities to also cache data at different layers in the data service. For instance, caching hot data that is accessed frequently into a fast storage tier [39]. Likewise, we could also cache query plans, partially or fully, in order to improve the compilation time. This is useful because query compilation is increasingly becoming an overhead in large complex queries, potentially accessing hundreds of inputs. Plan caching is also helpful because we know the actual costs of the partial or full plans seen before [32], and hence the query optimizer can: (i) make more informed decisions in picking the best plan, and (ii) significantly prune the search space by discarding poor plans.

5.2.3 Dependency-driven Optimizations. The dependency patterns could be used to run a number of dependency driven analytics [50]. Example includes computing the relative importance of queries in a data pipeline and scheduling them according to their importance [30]. Another example is creating physical designs for analytical data pipelines: enterprise data analytics often consists of a pipeline of queries that have data dependencies between them, i.e., the output of a producer query could be used in a subsequent consumer query. However, the producer query is often not aware of how its outputs are being consumed and so it may not create the right physical designs (partitioning, sorting, etc.). Our analysis from Scope workloads indicate that more than half of the structured outputs from producer queries end up re-partitioned or re-sorted down in the consumer queries. Analyzing workloads and mining frequently applied physical design could help to move those designs to the producer query itself. The physical designs could be created in the base output or as additional materialized views. However, note that it is hard for users to mine and come up with such complex physical design hints. In fact, we have already received several

requests from Scope customers to automatically apply physical design hints to their Scope jobs. Leveraging the dependency patterns, extracted from Peregrine feature store, makes this possible.

5.3 Tools

We need a core set of algorithmic tools to run the optimizations discussed in the previous Section. Interestingly, different optimization classes map naturally to different sets of tools (not comprehensive by any means), as discussed below. The learned optimizations, for instance, would need the machine learning libraries to run typical learning tasks, including feature engineering, model training, and model inference. Popular machine learning libraries include Scikit-learn [55], ML.Net [16], TensorFlow [20], PyTorch [19], Keras [15], etc. These libraries could run in a standalone manner or scaled via distributed processing platforms such as Spark, Scope, etc. The resulting learned models could be managed using tools such as MLFlow [66], ONNX [17], etc.

Multi-query optimizations could be often formulated as linear programs and so we need an industry strength mathematical solver, one that can scale to large number of variables and can apply a number of pruning. Popular examples include CPLEX [11], Gurobi [14], and OpenSolver [49].

Finally, the dependency-driven optimizations could be often mapped to graph problems, naturally or as approximations, and so we need a graph processing engine, e.g, Neo4j [51], to run such optimizations. These optimizations could be further scaled out using vertex-centric graph processing, e.g., Giraph [2], GraphLab [47], etc., which is a popular paradigm to divide and conquer graph analytics over massive datasets. Vertex-centric graph processing has also been shown to run efficiently within a database system [42], which avoids introducing one more system for running the optimization algorithms.

6 WORKLOAD FEEDBACK

Once the workload has been analyzed and optimizations decisions have been gathered, we need to provide the feedback for actions. Peregrine provides three kinds of feedback, namely, the *insights*, the *recommendations*, and the *self-tunings*. Insights are essentially summaries and reports over the workload IRs to help users understand their workload and take any appropriate tuning actions based on their interpretation. Peregrine generates summaries over the subexpressions IR and makes them available to the Scope customers for driving insights. Recommendations are outputs of the optimization algorithms that are provided as hints to the users. Users can apply these hints using the tuning knobs provided by the query engines, e.g., Scope query engine provides dozens of tunings knobs such as row count hint, operator algorithm hint, and forcing join order hint. Recommendations involve users in the tuning process since they are responsible for applying the hints on their own. We have enabled physical design hints, mined from the workload optimization in Peregrine, to the Scope customers. Self-tuning is more complex since it requires integration with the query engine, without requiring users in the loop. This is highly desirable compared to recommendations because most of the Scope job service users are non-experts, they do not have bandwidth to control the various optimizations, and the process is too dynamic to get right manually.

Below we describe how we built the self-tuning infrastructure for the Scope query engine. We first introduce the notion of *query annotations* for describing the self-tuning feedback, then we describe the feedback service to serve the query annotations, and finally we describe adding workload-awareness to the query engines by consuming the query annotations in the query engines.

6.1 Query Annotations

A self-tuning query engine needs to interpret and act upon the optimization decisions described in the previous Section. However, making special modifications to the query engine for every new optimization is not productive. Therefore, we encode the workload optimization decisions into a common format that is (i) extensible to add more optimizations, and (ii) could be integrated with multiple query engines. We call these encodings the *query annotations*. Query annotations provide a clear interface (or contract) between the workload optimization feedback and the changes in the query engines to consume that feedback. These two are often owned by separate teams within a company and so having a clear contract helps to develop and maintain the workload optimization algorithms and the query engines actions separately.

We define query annotations as a set of the following triplets: $Annotation(signature, action, parameters)$. Signatures are the query plan identifiers as described in Section 4.1. Actions are the names of the self-tunings performed by the query engine, e.g., the configuration to apply, or the tuning knob to set, or the query optimizer rule to invoke. Parameters provide the information needed for the action, e.g., the configuration value or the optimizer rule parameters. Note that a given signature may have several annotation actions, and a given annotation action might be applied to several signatures. Thus, the query annotations specify the self-tuning actions using the parameters and conditioned upon the query plan signatures.

Below we first describe the feedback service to serve the annotations to the query engines and then we describe making query engines workload aware.

6.2 Feedback Service

The query annotations need to be consumed by the query engine during compilation and optimization. The annotations need to be further consumed by multiple query engines. Therefore, Peregrine provides a feedback service to lookup the annotations from anywhere, and to scale annotation serving independently. We describe this below.

The workload optimization algorithms output their query annotations into a file in a cloud storage location. The feedback service periodically polls this location for new annotations and bulk loads any new annotation file present. In case of data corruption or failures, the feedback service could always be re-initialized from this cloud storage location. The feedback service is backed by relational storage, such as an Azure SQL Server instance, and so all annotations are loaded into SQL tables.

The service provides APIs to lookup annotations by their signatures, e.g., it can return all annotations for a given signature. Each annotation is associated with a customer account, and queries

from a customer account can load only the annotations associated with that account. Note that a query typically contains several signatures (recall that each node in the query plan corresponds to a signature) and there could be feedback for several of them. Looking up the feedback service for each individual signature can result in significant compilation overheads. Therefore, the feedback service allows to add tags to annotations and batch lookup all annotations for given tag(s). Example tag includes recurring job name, i.e., a periodic job that appears with a similar name each time. For such jobs, the compiler could load all annotations corresponding to the recurring job name in a single lookup.

We apply a few optimizations to improve the performance of feedback lookup. First, we create an index on the signatures and the tags to make the point and batch lookups faster. We always bulk load the annotations for a customer account, thereby not having to update the indexes incrementally. The feedback service further caches annotations in the application layer since many queries have common subexpressions and hence common annotations as well. The typical lookup overhead from the feedback is in the order of a few to tens of milliseconds, which is generally acceptable for analytical workloads where compilation could run into several seconds. Still for more latency sensitive workloads, we could co-locate the feedback service with the query engine, or even query the SQL Server backend directly.

In addition to serving the query annotations, the feedback service could also help coordinate optimizations across multiple queries. For example, CloudViews [41] used the service to obtain exclusive locks when materializing common subexpressions for future reuse. Thus, the feedback service can act as the synchronization point across multiple queries for such optimizations.

Finally, the feedback service expires annotations when new annotations for the same signatures and the same actions are available. In this case, the new annotations override the older ones. Alternatively, we consider the hash collision probability of the signatures and expire them well before any likely collision is possible. For Scope job service, we have found a few weeks to be a good heuristic in the worst case. Peregrine also provides admin control for purging all annotations for a given customer account and for enabling or disabling (without actually purging) the feedback for an account.

6.3 Workload-aware Query Engine

We now discuss making query engines workload aware, i.e., learning from how things went in the past workload and taking optimization actions for future queries. Figure 6 shows the query engine centric view of the workload feedback in Peregrine. The enhanced query engine has two key additions: (i) the mechanisms to load the query annotations feedback from the past workloads, and (ii) the capability to take optimization decisions based on that feedback. We discuss these below.

There are several ways of loading the query annotations. We could look up the annotations for each signature in the optimizer (the point lookup described in Section 6.2). Alternatively, we could pre-load the relevant signatures, using the tags defined in the feedback service, upfront in the compiler. Or, in case of smaller self-contained applications, we could also load all available signatures

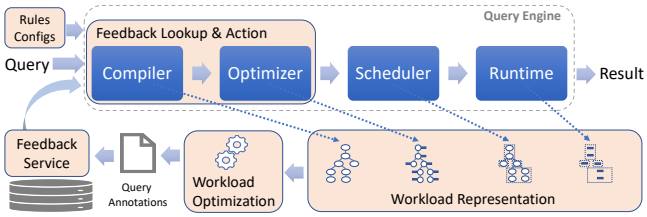


Figure 6: End to end workload-driven feedback when using Peregrine.

in the compiler (or optimizer) and use them wherever applicable. The default behavior is to load the query annotations from the feedback service. However, we also support loading the annotations file directly for debugging purposes: developers can quickly create and test new annotation feedback without having to go via the feedback service. This can also serve as a hot fix for customers in case they have some imminent issues with their workload optimizations.

The engine also needs to apply multiple workload optimization features developed by different teams. Many optimization features can be independently applied if they are developed for different stages or different workloads. For optimizations that interact on the same workload, the query planner that triggers the rules decides the order in which they are applied. For instance, a learned cardinality model will be applied earlier in the logical plans before learned cost models are applied in the physical plans. Furthermore, the optimization decisions could be taken at different stages in the query engine, namely compilation, optimization, scheduling, or even the runtime. As a result, the logic for taking the actions could be in the form of optimizer rules, configuration values, or even learned models for any of the numerous system heuristics.

Finally, while we have mostly considered offline workload analysis and feedback in this paper, the workload feedback illustrated in 6 could also be applied in a faster feedback loop or run in an online manner for quicker adaptivity to the recently observed workload.

Depending on the query engine, the workload awareness described above could either require changes to the engine or could be provided as one or more extensions from outside. Below we discuss both of these scenarios for the Scope and the Spark query engines respectively.

7 CASE STUDY

We now discuss some of the scenarios that are enabled by Peregrine. In particular, we discuss how Peregrine helps develop multiple optimizations quickly, apply the same optimizations across multiple engines, and democratizes workload optimization wherein several developers and researchers can participate. Section 7 shows the Peregrine instantiation for this case study. We will walk through this instantiation below.

7.1 Multiple Optimizations

The CloudViews project [41] was started for workload optimization in Scope job service, and the goal was to analyze and reuse common subexpression computations across the query workload. The CloudViews team added two signatures to the Scope query engine,

namely the *strict* and the *recurring* signatures, to capture the subexpression instances and templates respectively. The signatures are emitted along with the query plans into the workload repository, as shown in Figure 7. Peregrine then connects to the workload repository, parses the query logs, and also links the runtime execution statistics with operators in query plans. The workload is finally represented as subexpressions IR (see Section 4.3). CloudViews mines the common subexpressions from the IR and runs the view selection algorithm [40] on top of it. The selected views are served to the query engine via the feedback service. CloudViews adds a couple of optimizer rules and compiler flags, which require modifications to the Scope compiler and optimizer. See [41] for more details.

CloudViews opened the door for the use of common subexpressions in other optimizations as well. In particular, query statistics such as cardinality estimation could be improved over these subexpressions. Therefore, in the CardLearner project [64] cardinality models are learned from past workload to optimize future queries. Interestingly, the team could build upon the same workload IR and invoke the machine learning libraries over the previously mined common subexpressions. The learned models were loaded into the same feedback service as new annotations for the same signatures but different actions (cardinality model) and parameters (the model string). Of course, the optimizer changes were required to use the cardinality predictions from the models wherever applicable. However, by reusing the workload representation, the pattern mining, and the feedback service, the total feature development time came down from years to months. More importantly, the developer or researcher could focus more on the core feature of learning the cardinality model, rather than going through the painful process of analyzing workloads right from the raw logs.

7.2 Multiple Engines

CloudViews turned out to be relevant for other query engines as well. In particular, CloudViews was extended to the Spark query engine as part of the SparkCruise project [58]. Two signatures - *strict* and *recurring* were added in Spark via an event listener that computes the signatures and logs the optimized query plan at the end of query execution. We implemented the parsers to parse the Spark logs and collect the same set of entities, as from the Scope engine, into the feature store. Thereafter, the same enumerator generated the subexpressions workload IR as for Scope. Once the subexpressions IR were generated, we could leverage the same workload optimization algorithms, the same query annotations, and the same feedback service. This is fortuitous since the view selection algorithm could be improved independently and leveraged for multiple query engines automatically.

The Spark query engine needs additional optimizer rules to use the CloudViews annotations from the feedback service. While Scope required engine changes, Spark allows additional rules as extensions from outside. SparkCruise team added these rules in the Peregrine runtime library and referenced them in the Spark optimizer. The Spark optimizer applies these rules during optimization of logical query plan. SparkCruise thus developed was able to reduce the total running time of TPC-DS benchmark queries by 30% on HDI [6] test clusters.

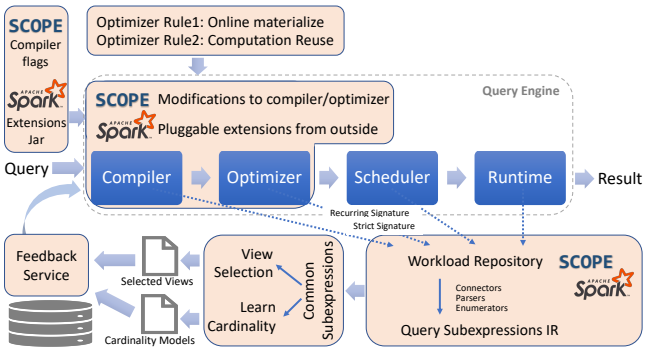


Figure 7: Peregrine instantiation for the case study, illustrating (i) multiple optimizations (materializing common subexpressions and learning cardinality models), and (ii) multiple query engines (SCOPE and Spark).

Thus, we see how with Peregrine the same workload optimizations could be easily brought to multiple query engines, again bringing down the feature development time from years to months.

7.3 Multiple Developers

In addition to building multiple optimizations and using them across multiple query engines, Peregrine has also enabled multiple developers to be more productive. In fact, since we started building Peregrine a year ago, over a dozen workload optimizations, or *apps* as we call them, have been built by different developers, researchers, and interns. This is because Peregrine abstracts many of the painful steps, e.g., workload representation into an IR, which need not be duplicated and allow people to build on each other's efforts, e.g., same view selection algorithm available to multiple query engines. This has made our own lives easier internally since it is now far easier to onboard new people, and to even bring in their expertise from another query processing domain. Even externally, our conversations with friends across companies such as Hortonworks, Databricks, Flink, Lyft, Netflix, etc. reveal that this could be a useful platform for collaboration. As a result, Peregrine is under active development and we are working towards open sourcing.

8 THE ROAD AHEAD

We now discuss the road ahead for Peregrine. We see two main pillars of development in Peregrine going forward, namely extending it with more capabilities and improving the end to end deployment experience. We elaborate on these two below.

8.1 Extensions

There is a significant room to extend Peregrine along several dimensions. First of all, new data services are rapidly mushrooming in public clouds and we would like to represent their workloads using Peregrine. Azure alone has more than a dozen data services publicly available and having common workload representation could enable several optimizations. This becomes further important when customers move from one data service to another (e.g., their application may be better supported by the other data service) or

migrate from on-premise to cloud infrastructure (e.g., to leverage managed hardware and software infrastructure), without losing the workload optimization capabilities. Effectively, an engine agnostic optimization platform like Peregrine also enables workload migration to the most suitable query engine.

In addition to adding more query engines, there is also room to develop the feedback end points. In particular, insights and recommendations are challenging because they need to be both understandable as well as actionable. Insights are often helpful in a visual form and there is a lot of prior work on visual analytics and explanations [37, 61]. Likewise, recommendations are more helpful when their impact is quantified. At the same time, the primary purpose of recommendations is to involve user judgement in the loop and not rely solely on the machine computed impact. Therefore, making effective use of user judgement when serving recommendations is a challenge.

Finally, although Peregrine currently focuses on query engines, workload optimization can be applied more broadly to other layers in the data processing stack. Components such as job scheduler, resource manager, storage system, physical hardware, or even the end user applications can all benefit from workload optimization. There are several existing works in this direction, e.g., self-tuning efforts for task schedulers [34], adaptivity in the storage layer [39], etc. The goal will be to investigate whether these could be integrated or the optimizations applied across multiple layers in the stack.

8.2 Experiences

There is a need to enhance the deployment experience of Peregrine in several ways. Many of these would require integrating with existing tools (with possible extensions) while others may need building new ones. First of all, building workload optimization apps involve a number of steps which need to be orchestrated. Therefore, we need to make it easier to compose these steps into pipelines, execute these pipelines efficiently, and be able to debug and troubleshoot them in case of errors. Popular orchestration tools include AirFlow [1], Celery [9], etc. The goal would be to integrate them with Peregrine for a unified experience.

Second, feedback serving needs to be differentiated for different environments. Apart from the current web service form factor, feedback could also be served as a micro-service or even as a library within an application, depending upon the latency requirements, privacy concerns, etc. Feedback serving becomes further challenging with machine learning models, since it is tedious to serialize and deserialize the models from different libraries. There are a lot of prior tools and technologies for model serving [17, 60] and we need to leverage the most appropriate ones for different scenarios.

Third, given that workload optimization is a continuous process, we need to track the lifecycle of the feedback (e.g., which feedback was gathered, which queries was it applicable for, was the it helpful in improving performance, etc.?). Incidentally, there is a plethora of recent works on model management, e.g., ModelDB [62], MLFlow [66], etc., and we need to extend it for more general feedback management. Tracking is needed to trace back the feedback in case of failures (e.g., which feedback led to performance regression) or for regulatory concerns (e.g., did optimization algorithms run on sensitive data?).

Finally, we need to build the user interfaces for better productivity, e.g., integration with the IDEs used by the developers, testing and replaying queries with and without feedback and providing more general configuration management for switching various knobs in the system. These self-serving tools can significantly improve the developer experience and enhance productivity.

9 OTHER RELATED WORK

Workload optimization is gaining a lot of traction with the major cloud providers and below we provide a brief overview of some of these efforts. While some believe that aggressive resource provisioning can solve the performance tuning by dynamically adding more hardware on demand [10], new trends like serverless computing [3, 7, 13] are gaining popularity since they hide many of the tuning and provisioning decisions from the user, putting the onus on the cloud provider instead. Therefore, cloud providers are increasingly looking at newer tools and technologies to improve the customer experience. In particular, people have turned towards AI for solving the hard workload optimization problems, such as automatically creating indexes in Microsoft SQL Server [31] or determining the best query paths using the IBM Db2 AI for z/OS. Likewise, Oracle autonomous database aims to provide automated (or self-driving) experience for much of the database management and tuning [18]. Huawei's GaussDB is another such effort to leverage AI for making database administration and tuning easier for enterprise customers [12]. Thus, across the board, we see the interest in AI for solving database problems. However, most of these efforts are for a specific query engine and often for specific optimizations as well.

The only work we are aware of that considers common infrastructure across engine is the database agnostic framework for labeling query strings from Snowflake [38]. However, it is limited in terms of the optimizations it can support, since it does not exploit query plans or runtime statistics. It can be used in coarse grained applications such as workload sampling and security auditing.

10 CONCLUSION

In this paper, we presented Peregrine, a platform for optimizing query workloads in cloud query engines. Key features of Peregrine include an engine agnostic way of describing query workloads, an intermediate representation (IR) to easily featurize the workload and build a shared set of optimizations, a classification of workload patterns and their corresponding optimization techniques to make the space of possible optimizations tractable, mechanisms to serve workload feedback, and a prescription for adding workload awareness to a query engine. Peregrine has been used at Microsoft to quickly implement, test, and deploy multiple workload optimizations such as CloudViews and CardLearner. Peregrine also provides a write once, run everywhere model for workload optimization across multiple engines such as Scope and Spark, which helped us bring CloudViews to Spark as well. In summary, this paper ushers data systems into the era of intelligent public clouds, essentially filling the gap of missing DBAs in these clouds.

Acknowledgements. We would like to thank the SCOPE and the HDInsight teams for helping us understand, build, and deploy workloads optimizations over their workloads. Thanks to several of our Scope job service customers for providing scenarios to apply workload optimizations. And finally, special thanks to our PM team for being supportive all along.

REFERENCES

[1] Apache airflow. https://airflow.apache.org. Accessed: 2019-06-08.
[2] Apache Giraph Project. http://giraph.apache.org.
[3] Aws serverless. https://aws.amazon.com/serverless.
[4] Azure data explorer. https://azure.microsoft.com/en-us/services/data-explorer. Accessed: 2019-06-08.
[5] Azure data lake analytics. https://azure.microsoft.com/en-us/services/data-lake-analytics. Accessed: 2019-06-08.
[6] Azure HDinsight. https://azure.microsoft.com/en-us/services/hdinsight. Accessed: 2019-06-08.
[7] Azure serverless. https://azure.microsoft.com/en-us/overview/serverless-computing.
[8] Azure sql database. https://azure.microsoft.com/en-us/services/sql-database. Accessed: 2019-06-08.
[9] Celery: Distributed task queue. http://www.celeryproject.org. Accessed: 2019-06-08.
[10] Cloud databases: The advantage of no more performance tuning. https://datometry.com/resources/cloud-express-articles/cloud-databases-advantages-no-more-performance-tuning. Accessed: 2019-06-08.
[11] Cplex optimizer. https://www.ibm.com/analytics/cplex-optimizer. Accessed: 2019-06-08.
[12] Gaussdb. https://techcrunch.com/2019/05/14/huawei-cloud-database.
[13] Gcp serverless. https://cloud.google.com/serverless.
[14] Gurobi optimization. http://www.gurobi.com. Accessed: 2019-06-08.
[15] Keras: The python deep learning library. https://keras.io. Accessed: 2019-06-08.
[16] Ml.net. https://dotnet.microsoft.com/apps/machinelearning-ai/ml-dotnet. Accessed: 2019-06-08.
[17] Open neural network exchange. https://onnx.ai. Accessed: 2019-06-08.
[18] Oracle autonomous database. https://www.oracle.com/database/autonomous-database.html.
[19] Pytorch. https://pytorch.org. Accessed: 2019-06-08.
[20] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, 2016.
[21] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. Re-optimizing Data-parallel Computing. In *NSDI*, 2012.
[22] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. In *VLDB*, 2000.
[23] M. Akdere, U. Çetintemel, M. Riondato, E. Upfal, and S. B. Zdonik. Learning-based query performance modeling and prediction. In *ICDE*, 2012.
[24] Amazon Athena. https://aws.amazon.com/athena/.
[25] Google BigQuery. https://cloud.google.com/bigquery.
[26] N. Bruno, S. Chaudhuri, A. C. König, V. R. Narasayya, R. Ramamurthy, and M. Syamala. AutoAdmin Project at Microsoft Research: Lessons Learned. *IEEE Data Eng. Bull.*, 2011.
[27] J. Camacho-Rodríguez, D. Colazzo, M. Herschel, I. Manolescu, and S. R. Chowdhury. Reuse-based optimization for pig latin. In *CIKM*, 2016.
[28] S. Chaudhuri and V. Narasayya. Automating Statistics Management for Query Optimizers. *IEEE TKDE*, 2001.
[29] S. Chaudhuri and V. Narasayya. Self-tuning Database Systems: A Decade of Progress. In *VLDB*, 2007.
[30] A. Chung, C. Curino, S. Krishnan, K. Karanasos, P. Garefalakis, and G. R. Ganger. Peering through the dark: An owl's view of inter-job dependencies and jobs' impact in shared clusters. In *SIGMOD Demonstration*, 2019.
[31] B. Ding, S. Das, R. Marcus, W. Wu, S. Chaudhuri, and V. Narasayya. Ai meets ai: Leveraging query executions to improve index recommendations. In *SIGMOD*, 2019.
[32] B. Ding, S. Das, W. Wu, S. Chaudhuri, and V. Narasayya. Plan stitch: harnessing the best of many plans. In *VLDB*, 2018.
[33] A. Dutt, C. Wang, A. Nazi, S. Kandula, V. Narasayya, and S. Chaudhuri. Selectivity Estimation for Range Predicates using Lightweight Models. In *VLDB*, 2019.
[34] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy. Dhalion: self-regulating stream processing in heron. In *VLDB*, 2017.
[35] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 592–603. IEEE, 2009.
[36] G. Graefe. The Cascades Framework for Query Optimization. In *IEEE Data Engineering Bulletin*, volume 18, pages 19–29, 1995.
[37] K. Hu, M. A. Bakker, S. Li, T. Kraska, and C. Hidalgo. Vizml: A machine learning approach to visualization recommendation. In *CHI*, 2019.
[38] S. Jain, J. Yan, T. Cruane, and B. Howe. Database-agnostic workload management. In *CIDR*, 2019.
[39] V. Jalaparti, C. Douglas, M. Ghosh, A. Agrawal, A. Floratou, S. Kandula, I. Menache, J. S. Naor, and S. Rao. Netco: Cache and i/o management for analytics over disaggregated stores. In *Proceedings of the ACM Symposium on Cloud Computing*, 2018.
[40] A. Jindal, K. Karanasos, S. Rao, and H. Patel. Thou Shall Not Recompute: Selecting Subexpressions to Materialize at Datacenter Scale. In *VLDB*, 2018.
[41] A. Jindal, S. Qiao, H. Patel, Z. Yin, J. Di, M. Bag, M. Friedman, Y. Lin, K. Karanasos, and S. Rao. Computation Reuse in Analytics Job Service at Microsoft. In *SIGMOD*, 2018.
[42] A. Jindal, P. Rawlani, E. Wu, S. Madden, A. Deshpande, and M. Stonebraker. Vertexica: your relational friend for graph analytics! In *VLDB*, 2014.
[43] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. n. Goiri, S. Krishnan, J. Kulkarni, and S. Rao. Morpheus: Towards automated slos for enterprise clusters. In *OSDI*, 2016.
[44] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In *SIGMOD*, 2018.
[45] J. Li, A. C. König, V. Narasayya, and S. Chaudhuri. Robust estimation of resource consumption for sql queries using statistical techniques. In *VLDB*, 2012.
[46] SIGMOD Blog. http://wp.sigmod.org/?p=1075.
[47] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. In *VLDB*, 2012.
[48] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul. Neo: A learned query optimizer. *arXiv preprint arXiv:1904.03711*, 2019.
[49] A. J. Mason. Opensolver-an open source add-in to solve linear and integer progammes in excel. In *Operations Research Proceedings 2011*. Springer, 2012.
[50] R. Mavlyutov, C. Curino, B. Asipov, and P. Cudré-Mauroux. Dependency-driven analytics: A compass for uncharted data oceans. In *CIDR*, 2017.
[51] J. J. Miller. Graph database applications and concepts with neo4j. In *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA*, 2013.
[52] M. Mitzenmacher. A model for learned bloom filters and related structures. *arXiv preprint arXiv:1802.00884*, 2018.
[53] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. Mrshare: Sharing across multiple queries in mapreduce. *PVLDB*, 2010.
[54] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi. Learning state representations for query optimization with deep reinforcement learning. In *DEEM*, 2018.
[55] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 2011.
[56] K. Rajan, D. Kakadia, C. Curino, and S. Krishnan. PerfOrator: eloquent performance models for resource optimization. In *SoCC*, 2016.
[57] R. Ramakrishnan, B. Sridharan, J. R. Douceur, P. Kasturi, B. Krishnamachari-Sampath, K. Krishnamoorthy, P. Li, M. Manu, S. Michaylov, R. Ramos, N. Sharman, Z. Xu, Y. Barakat, C. Douglas, R. Draves, S. S. Naidu, S. Shastry, A. Sikaria, S. Sun, and R. Venkatesan. Azure Data Lake Store: A Hyperscale Distributed File Service for Big Data Analytics. In *SIGMOD*, 2017.
[58] A. Roy, A. Jindal, H. Patel, A. Gosalia, S. Krishnan, and C. Curino. Sparkcruise: Handsfree computation reuse in spark. In *VLDB*, 2019.
[59] T. Siddiqui, A. Jindal, S. Qiao, H. Patel, and W. Le. Learned cost models for optimizing big data queries. *Under Submission*, 2019.
[60] A. Team. Azureml: Anatomy of a machine learning service. In *Conference on Predictive APIs and Apps*, 2016.
[61] M. Vartak, S. Rahman, S. Madden, A. Parameswaran, and N. Polyzotis. Seedb: efficient data-driven visualization recommendations to support visual analytics. In *VLDB*, 2015.
[62] M. Vartak, H. Subramanyam, W.-E. Lee, S. Viswanathan, S. Husnoo, S. Madden, and M. Zaharia. Model db: a system for machine learning model management. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, 2016.
[63] L. Viswanathan, A. Jindal, and K. Karanasos. Query and resource optimization: Bridging the gap. In *ICDE*, 2018.
[64] C. Wu, A. Jindal, S. Amizadeh, H. Patel, S. Qiao, W. Li, and S. Rao. Towards a Learning Optimizer for Shared Clouds. In *VLDB*, 2019.
[65] Z. Yang, E. Liang, A. Kamsetty, C. Wu, Y. Duan, X. Chen, P. Abbeel, J. M. Hellerstein, S. Krishnan, and I. Stoica. Selectivity estimation with deep likelihood models. *arXiv preprint arXiv:1905.04278*, 2019.
[66] M. Zaharia, A. Chen, A. Davidson, et al. Accelerating the machine learning lifecycle with mlflow. *Data Engineering*, 2018.
[67] J. Zhou, N. Bruno, M.-C. Wu, P.-Å. Larson, R. Chaiken, and D. Shakib. SCOPE: parallel databases meet MapReduce. *VLDB J.*, 21(5):611–636, 2012.