Query Optimizer as a Service: An Idea Whose Time Has Come!

Alekh Jindal*
alekh@keebo.ai

Jyoti Leeka
jyoti.leeka@microsoft.com

ABSTRACT

Query optimization is a critical technology that is common across all modern data processing systems. However, it is traditionally implemented in silos and is deeply embedded in different systems. Furthermore, over the years, query optimizers have become less understood and rarely touched pieces of code that are brittle to changes and very expensive to maintain, thus slowing down the pace of innovation. In this paper, we argue that it is time to think of query optimizer as a service in modern cloud architectures. Such a design can help build a common set of well-maintained optimizations that are externalized from the query engines and that could be learned and improved using the large workloads present in modern clouds. We present, *Oasis*, a reference architecture for query optimizer as a service and describe our success in deploying the early version of it in Cosmos. Finally, we discuss the risks and responsibilities involved with *Oasis* to ensure it is a win-win for everyone.

1. INTRODUCTION

Query optimization has been the bread and butter of data processing engines for improving performance and reducing the cost of declarative user queries. It is now also becoming critical piece in modern clouds where data systems are pervasive, user expertise is minimal, and lowering operational costs is paramount. Consequently, we are seeing a lot of effort and investment in building advanced query optimization capabilities in cloud-native data systems, such as Spark [27], Greenplum [10], Snowflake [25], F1 [23], Azure SQL [2], SCOPE [7], Spanner [26], Big Query [4], RedShift [20], Athena [1], CockroachDB [9], among others.

Unfortunately, the current approach to building query optimization capabilities results in re-implementing similar techniques in different systems over and over again. For example, many systems, including Spark, Calcite, Greenplum, Snowflake, F1, SQL Server, and SCOPE, have implemented Cascades-style query optimization. As a result, there is a lot of repeated effort to get query optimization right in each of these systems. Instead, unifying the query optimization capabilities into a common platform will help serve these customers better by advancing the state-of-the-art across the board.

Apart from the redundant effort across systems, getting query optimization right just for a given system is hard by itself. This is evidenced by decades of research on improving strategies for query plan search, models for estimating query costs, and models for intermediate cardinalities and other statistics. In fact, most production systems have been hardened over the years based on countless user scenarios, performance regressions, and customer incidents seen by the product teams over time. This has also resulted in query optimizers turning into massive black boxes that are not just hard to understand or tune but also something that the product teams are highly skeptical to touch or change significantly.

The heart of the problem in current query optimizers is the deep coupling of the optimizer within the query engine, a design decision that is now a big problem for development costs and is hard to advance the query optimizer to fast changing needs of the newer query processing engines. Therefore, it is time to decouple the query optimizer from the query engine into a separate platform. This will help consolidate the efforts and advance the state-of-the-art rapidly. This is also akin to how different components in the Hadoop stack, including the file system (HDFS [5]), resource manager (YARN [30]), task scheduler (Tez [29]), etc., were carved out into independent layers for more agility and standardization. It is time to do the same for the query optimizer.

In this paper, we rethink the traditional query optimizer and present a service-oriented architecture for query optimization in modern clouds. Our core philosophy is not to build yet another query optimizer that replaces the existing ones, something not practical for the large number of systems deployed out there, but rather to *externalize* the various query optimizer components to an external service and look them up when optimizing each new incoming query. A key enabler for such external-

*Work done while the author was at Microsoft.

ization is the plethora of recent machine learning based techniques to improve various query optimizer components, such as cardinality, cost model, and query planner. These learning-based approaches could train over large cloud workloads and specialize for various databases or workload instances. The learning algorithms and models can then accrue towards a common query optimization platform that can be shared across and developed much more rapidly. We have taken a series of incremental steps in recent years towards realizing this architecture at Microsoft, and this paper puts these efforts together into a coherent vision.

In the rest of the paper, we first describe a rethinking of query optimizers from traditional deeply embedded one to an external and service-oriented one (Section 2). We present, *Oasis*, a reference architecture for optimizer as a service (Section 3), and discuss our production successes in implementing *Oasis* by externalizing different components of the SCOPE query optimizer (Section 4). Finally, we touch upon several risks that an externalized and ML-based optimizer service carries and our responsibilities towards them (Section 5).

2. RETHINKING QUERY OPTIMIZER

In this section, we describe our rethinking of query optimizers from traditional to a service-oriented one.

2.1 Traditional Query Optimizer

Traditionally, a query optimizer consists of three core components, namely the cardinality estimator, the cost model, and the query planner, that are embedded inside a query engine. Figure 1(a) illustrates this simplified architecture. We describe each of the three components in more detail below.

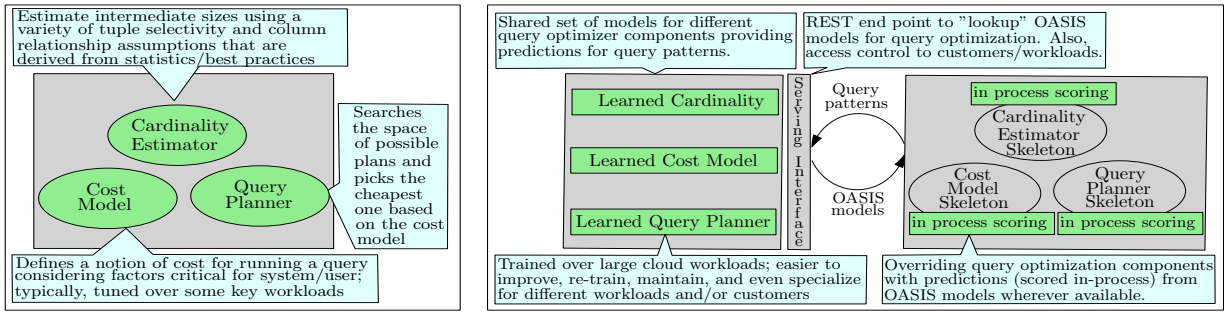
The query planner explores the space of possible query plans for the given user query. Various query plan searching approaches have been proposed in the literature, including bottom-up planner that starts from the input and builds up to the final, top-down planner that starts from the final required plan and implements it down to the base inputs, and randomized planner that explores permutations in the neighborhood to discover more optimal plans. In particular, the Cascades optimizer, a top-down query planner, has become widespread in industry with several engines such as SCOPE, SQL Server, Spark, Snowflake, Greenplum, F1, and others adopting it. The goal of the query planner is to produce the cheapest query plan to execute based on the cost model.

The cost model estimates the cost of a candidate query plan. It uses the cardinality or the intermediate size after each operator in the query plan as a core ingredient to estimate the CPU, IO, latency, and other costs into a single numerical value for each candidate query plan. Naturally, this numerical value does not correspond to any standard metric such as seconds but rather it gives a relative value to compare how good different query plan candidates are. For multi-core and distributed query processors, parallelism is another key factor since the costs are going to be very different based on the number of threads or the number of nodes that can process data in parallel. This becomes tricky because the system may not want to use all available parallelism due to other concurrent queries [22] and also because the performance may not consistently improve at higher parallelism due to increased data movement and IO [19]. As a result, picking the right parallelism is very challenging and yet very important for parallel and distributed plans. In practice, cost models are highly sensitive components of the query optimizer which are typically tuned over some key customers or benchmarks and rarely touched henceforth. Any changes in the cost model could not only take years in development but are also hard to deploy due to their large and unknown impact on the query plans [24].

The cardinality estimator estimates the output number of rows (or cardinality) from each operator in the query plan. It is a key ingredient when estimating the cost of a query subexpression and further combining those costs recursively for the entire query. A cardinality estimator relies on statistics such as histograms, sketches, min/-max, distinct value counts, etc., on the base tables, operator semantics to determine the output size given the inputs, e.g., key foreign key join, cross join, etc., and heuristics derived from customer experience for many other unknowns, e.g., the selectivity of table and scalar UDFs. Cardinality estimator also suffers from limited or stale statistics due to their cost of collection, exponential error propagation which grows worse with larger query plans [17], and the black box nature of user defined operators that are typically present in production workloads. Consequently, cardinality estimation is highly challenging and is routinely found to be off by orders of magnitude in production systems [32].

Some open-source projects have attempted to abstract out the query optimizer into more modular code structure that could be applied to different query processing engines. Most notably, the Apache Calcite [6] project provides a generic and extensible planner framework, with adapters built for many different query engines. Unfortunately, getting the cardinality estimator and the cost model right are the bare essentials of the query optimizer, and these two highly depend on the execution environment and the workload instances at hand. Therefore, Calcite requires users to plug-in these things from their environments which is non-trivial.

In summary, traditional query optimizers have been built over the years with significant effort and using domain knowledge that were relevant at some point in time. However, they have now turned into system components



(a) Traditional Query Optimizer

(b) Service-oriented Query Optimizer

Figure 1: Contrasting traditional and service-oriented query optimizer architectures.

that are too sensitive to touch and too brittle to change.

2.2 Service-oriented Query Optimizer

Given the limitations of traditional query optimizers, let us now consider an alternative service-oriented approach to query optimization. Figure 1(b) illustrates this new architecture. The first thing to notice is the externalization of the three main components of the query optimizer, namely the cardinality estimator, the cost model, and the query planner, from the deeply embedded query optimizer to an external service. Second, the externalized components are hosted as learned ML-models trained over the past query workloads seen by a given set of customers in a given environment. For each incoming query, relevant models are loaded to the query engine, based on the patterns in the query, over a REST endpoint. The query engine still contains its default query optimizer; however, its components are overridden by learned models loaded from the external service.

The above service-oriented design for query optimizer a.k.a. QO, has several advantages:

- We do not introduce yet another QO but rather a practical approach for better query optimizer development.
- The service-oriented approach is decoupled yet completely integrated end-to-end, unlike Calcite that requires to plug-in cardinality estimator and cost model.
- It opens up the black box to see how good different components are, and which ones really matter.
- It constantly improves the different components of query optimizer via better machine learning techniques.
- The system gets better with more usage on the cloud since more diverse training data becomes available.
- Externalization allows exploring better optimization possibilities without worrying about overloading QO.
- There is a natural instance optimization for specific workloads and patterns, something which is desirable.
- The decoupled approach allows scaling query engine and externalized optimization decisions separately.
- The hosted models and techniques are shareable across different databases or even engines since models learn the characteristics of the underlying data anyways.
- We essentially build a single knowledge base that could be easily instantiated for different settings.
- We can deploy several versions of the optimizer at the same time; users can choose which version they want.
- It is easier to debug and investigate the issue with respect to each of the components.
- It is easier to roll back for a specific customer.
- We can easily fall back to default behavior if required.
- System developers can focus more on training the core query optimizer components independently, thus accelerating the software engineering life cycle.
- Finally, the service-oriented design makes the query optimizer future proof for the scale, variety, and complexity of the cloud workloads; future workloads simply means training newer and better ML models, e.g., better featurization, better model selection, or parameter tuning over the new workload.

In summary, the service-oriented approach opens a new way to think about query optimization, bringing in several notable advantages of doing so. It is also the most natural design in modern clouds where complex systems are typically broken down into smaller and simpler services that could be managed independently. With this, in the following section, we present a reference architecture for turning a traditional query optimizer into a service-oriented one.

3. *Oasis*

We now present the *Oasis* reference architecture. This is derived from our prior work on building workload optimizations for cloud query engines [12, 21], however, we tailor the discussion for building *Oasis* below. Our goal is not to replace the traditional query optimizer with a new one, since that is not possible in production setting, but rather to transform it into a service-oriented one. Below we present a seven-step recipe for turning a given query optimizer into *Oasis*.

3.1 Signatures

The first step to creating an *Oasis* is to implement *signatures*, or hashes, for every query subexpression (or query sub-tree).A signature essentially captures a query sub-pattern whose behavior could be learned across the entire workload. Therefore, it could also be seen as an extremely lightweight featurization, which is very useful given the sub-second end-to-end time spent in many typical query optimizers. One could also create multiple signatures for different granularity of the sub-patterns. Finally, as we will see in this section below, signatures facilitate training smaller micromodels and trivially distributing the training pipeline over large clusters.

3.2 Observability

The query engine needs to emit the signatures along with its corresponding runtime metrics telemetry. This observability helps understanding the query performance in a given runtime environment, i.e., it can help build better cardinality and cost models for these workloads. Some of the key things to consider include keeping the overheads of observability minimal on the query performance and making sure the telemetry store can handle the volume of incoming data, e.g., by making telemetry collection asynchronous and scaling the back-end store to support fast ingestion. Furthermore, we need to preserve the customer privacy by anonymizing any business sensitive or personally identifiable information.

3.3 Training Input

Once the telemetry is collected, we need the training input for different optimizer components. This is a cumbersome process that involves extracting the telemetry into a tabular structure, imputing missing data values, denormalizing the data into a flat table that is more amenable to training, and applying data cleaning techniques to it. Given the complexity, we want to do this step once and share the resulting training data for building all models. This is also the step where we can bring in data from other sources, e.g., resource manager or the machine counters, and combine them with the query optimizer telemetry to produce a rich training input.

It is noteworthy that, as workloads and underlying data in shared cloud infrastructures change we repeat this step periodically. We do it once for each instance, depending on the rate of change of data and workload.

3.4 Micromodels

The crux of *Oasis* is micromodels [13], or per-signature models which are lightweight (typically linear models), way smaller in size, and much more specialized than large general-purpose models. As a result, they are faster to train and easy to replace quickly. Micromodels trade the generality of traditional query optimizers, wherein they had to produce the exact same behavior for every query and every customer, with specialized behavior that is ultra-local to a given customer and a given workload, i.e., each customer can essentially have their own cardinality estimator, cost model, and query planner customized for their workloads. Furthermore, micromodels do not see a customer workload as monolithic but rather splits it into fine-grained chunks that could be learned independently and then combined later for the overall query optimization.

3.5 Model Lookup

One of the key aspects of *Oasis* is how query optimizer components are loaded into the query optimizer. We can load models by customers, query templates, or signatures. We can even choose to load the models selectively for subset of customers (e.g., pilot customers or those who have signed up), while still providing the default behavior to others. Such access control makes deployment and onboarding of *Oasis* far easier. This change requires establishing a contract between query engine and externalized service. Over and above that model lookup is fast, for Cosmos users we found lookup time to be 10-15 msec, we achieved this by caching.

3.6 In-process Scoring

Oasis scores models in-process within the query optimizer. This is to reduce the overhead of making thousands of external invocations for model scoring. Instead, *Oasis* loads the model into the optimizer and scores them wherever needed. While this requires changes to the optimizer context, the models themselves have low footprint since they are way smaller in size, and the actual scoring is fast since they are typically linear models.

3.7 Overriding Default Behavior

Finally, *Oasis* needs to override the default optimizer behavior. This could be done either by additional condition checks for the presence of available models or existing mechanisms for overriding the default behavior, e.g., row counts hints could be used to substitute cardinality from default estimator with those obtained from the learned models. This is a one-time deep change in the existing query optimizers that is useful to control the behavior from the externalized service.

4. PRODUCTION SUCCESS

In this section, we describe our production success in deploying the *Oasis* architecture for Cosmos big data platform in Microsoft. Below we discuss how we went about externalizing each of the three components in the SCOPE query optimizer and the lessons learned so far. Sharing the same infrastructure with other query engines like Spark and SQL DW [28] is still an ongoing effort.

4.1 Externalized Cardinality

Cardinality estimation in SCOPE could be orders of

magnitude off – from up to tens of thousand times under-estimation to up to a million times over-estimation [32]. This is due to user defined operations that end up as black boxes for query optimizer. Fortunately, SCOPE workloads have many subexpression patterns that could be generalized across queries. As a result, we were able to achieve up to six orders of magnitude more accuracy. Unfortunately, better cardinality does not necessarily translate to better performance. Therefore, we had to identify scenarios that lead to consistently better performance with externalized cardinality models. Via extensive experimentation, we identified a range of heavy over-estimation, which when fixed leads to lower processing time. This was an important step in enabling externalized cardinality in production, and it was made possible due to externalization since the cardinality models could be independently trained, tested, validated, and experimented over and over again. In a nutshell, *externalization* helped achieve quick deployment and *common subexpression* helped achieve accuracy.

Still, many gaps remain in current version of externalized cardinality. For instance, combining cardinality from learned and default models could lead to expensive plans. Our solution so far has been to leverage retraining. Improving cardinality models remains ongoing work and something that is facilitated by *Oasis*.

4.2 Externalized Cost Models

Cost models in the SCOPE query engine have remained relatively static for several years. There was a significant effort to replace the original cost model, however, it never got enabled by default due to unknown implications. As such it remains optional for customers to try out [24]. Our learning when trying to build learned cost model was that one of the biggest influences of cost estimates in SCOPE is on resource selection. Specifically, we realized that the number of containers to be used for each query is often mis-calculated, resulting in either wasted resources or poor performance [22]. Furthermore, query and resource optimization decisions are often made independently, even though they affect each other [31]. Therefore, we focused on extending the cost models to also make resource decisions in the query optimizer. Our deployed solution was to avoid over-allocation by predicting peak resources for a query plan. This resonated well with customers since they could get more work done with the same amount of resources.

Several follow-ups are possible to our initial deployment of externalized cost models. Approaches we tried include predictively giving up resources in later parts of the query plan [3]. These approaches have gaps in terms of production readiness, e.g., how do we avoid ending up under-allocating. However, the externalized architecture of *Oasis* provides a solid foundation for iteratively improving these approaches.

4.3 Externalized Query Planning

The SCOPE query planner consists of 256 rules. Unfortunately, one third of the rules are not enabled in any of the current workloads [18]. This is because even though a significant effort is spent in writing new rules [16], it is very hard to understand their implications on wider workloads, leading them to remain optional and unused. Our approach to externalizing the query planner was to identify which set of rules make sense for a query, thereby steering its search space towards efficient plans. Additionally, it will be inefficient for optimizers to steer the search space without externalization because workloads as well as data keeps evolving, which requires frequent changes to the optimizer code, and the need to wait for optimizer's next release cycle to be deployed.

To keep things explainable, our current production deployment only changes the space of rules by one at a time, thereby incrementally steering queries to paths that are also understandable to engineers and customers.

Other than steering the search space for a given query, multi-query optimization is also an important part of the query planner. CloudViews [8, 14] is our approach to reuse common computations across queries, whereby we externalize which computations could be reused and load them via *Oasis*. CloudViews has been deployed in production and given its automatic and self-discovering nature, customers have found it very useful. Furthermore, the externalized aspect naturally allows for experimenting with newer view selection algorithms that can be readily plugged into [11].

Finally, physical layouts (partitioning, etc.) of tables are important for avoiding expensive shuffles. This is a multi-query optimization problem since choosing the right layouts depend on all consumers of the dataset. Given that datasets in Cosmos have producer-consumer relationships, our approach is to analyze all consumers of a given dataset to decide on the layout that the producer should be creating. However, when deploying in production we realized that making this automatic can have unexpected implications on some consumers. Therefore, our current version simply recommends layouts for the owners of the producer jobs, and they can choose to apply them. Again, providing these multi-query optimization decisions from an external service allows them to be refined and improved over time.

5. RISKS & RESPONSIBILITIES

Applying machine learning for building better systems has emerged as a hot trend in recent years. It is motivated by the fact that modern cloud systems have become too complex and unwieldy to manage or optimize, coupled with rapid advances and the ease of use of machine learning ecosystem. As a result, it is attractive to leverage machine learning over large work-

loads in modern data systems to derive better performance, lower costs, and add more automation for a productive yet cost-effective experience for a large body of data users out there who may not have the expertise to achieve many of these things on their own.

Even though it makes sense to apply machine learning for building better query optimizers, it is well known that machine learning also comes with its own set of dangers. The question is whether by applying machine learning for query optimization, we are importing those dangers as well. Therefore, in this paper, we also make one of the very first attempts to study the risks and responsibilities of applying ML to query optimizers. We believe this will help us as a community of data system builders to be more responsible. While in the ML community, ethical and responsible AI emerged as later topics, we want to start this conversation early in the data systems world to avoid the pitfalls and suggest best practices before they get baked too deeply in system stack.

Some of the critical questions that we raise include what is the deeper impact of ML on query optimizers and for ML-for-systems? Are we making systems strictly better? Is new system behavior still aligned with the intended one? Should users be aware of a new goal setting? Should system developers be aware of changes in how users are served? Our goal is to provide pointers and suggestions on making ML-for-systems more responsible, one that is more ethical, fairer, and considers the overall good of the user community.

5.1 Are We Serving All Customers?

Rich Gets Richer. One of the biggest risks of using ML for query optimizers is that customers with larger workloads benefit more, both in terms of performance and efficiency, since they can train better models to make the optimization decisions. In contrast, the cloud provider has a responsibility to serve all customers and so they need to devise techniques to transfer the learnings across.

Marginalizing Small Players. Learning-based optimizations affect specific instances rather than improve the overall system, small players risk getting seriously marginalized with outdated system behavior with little incentive for the provider to take them into account given their smaller size. Therefore, the cloud providers still need to find mechanisms to democratize *Oasis* for all players.

5.2 Are We enabling New Workloads?

Penalizing the Explorer. Customers who try new analysis risk creating diverse workloads that may not provide enough learnable patterns to *Oasis*, thus penalizing the exploring and rewarding repetitive work with better performance and lower costs. The question then is how to incentivize explorers with good performance without asking them to become more predictable.

Differentiating the Workload. Applying ML to query optimizers is also often considered as instance optimization [15], which by definition, seeks to optimize different portions of the workload differently. The question then is how do we divide the workload into different portions and how to make this a fair division.

5.3 Are We Building Better Systems?

Opaque Systems. By externalizing and replacing parts of the query optimizer with ML models, we risk substituting transparent, well understood components with models that could be hard to explain or reason about, i.e., are we leading to fundamentally better systems?

The Curse of Convenience. It is convenient to learn what is already being observed, instead of emitting new telemetry, and feedback using existing mechanisms, instead of building new ones, thus, missing the big picture. For instance, learning cardinality over the past workloads ends up biasing over known plans and potentially not guiding the optimizer into unknown spaces.

5.4 Are We Helping System Developers?

MLOps without ML Background. Modern clouds have a devops model, where developers are part of the operational process as well. Unfortunately, ML-for-Systems now puts the developers on the MLOps path without them having the necessary ML background.

Hammer for Every Nail. Developers can easily fall into the trap of fixing everything in the system software via ML, bypassing better coding and system building practices in the first place. The challenge though is that they now need to maintain both the default base system and augmented part of it in *Oasis*.

5.5 Are We Generating Value?

Learning vs Improving. It is non-trivial to understand the improvements due to a learned component beforehand. In fact, big improvements in accuracy may not translate to improvement in performance or other metrics. Therefore, much effort must be put in before the real value is understood or derived.

Throwaway Work. Quickly retraining a large number of ML models that can inform various aspects of the query optimizer implies creating more work of lesser value. The question is whether the resources could instead be used for work whose value is longer lasting.

6. CONCLUSION

This paper opens up a fundamentally new debate in query optimization — to build query optimizer as a service. We explored this radically new design from several practical aspects and presented a reference architecture, *Oasis*, as the first step. We also shared lessons from early deployments of *Oasis* in Cosmos and also introspected risk and responsibilities associated with it.

REFERENCES

[1] AWS Athena. https://aws.amazon.com/athena/, 2021.

[2] Azure SQL. https://azure.microsoft.com/en-us/products/azure-sql/.

[3] M. Bag et al. Towards plan-aware resource allocation in serverless query processing. In *USENIX HotCloud*, 2020.

[4] Google BigQuery. https://cloud.google.com/bigquery.

[5] D. Borthakur et al. Hdfs architecture guide. *Hadoop apache project*, 53(1-13), 2008.

[6] Apache Calcite. https://calcite.apache.org/, 2021.

[7] R. Chaiken et al. SCOPE: easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2), 2008.

[8] CloudViews Project. https://www.microsoft.com/en-us/research/project/cloudviews/, 2021.

[9] CockroachDB. https://www.cockroachlabs.com/, 2021.

[10] Greenplum. https://greenplum.org/, 2021.

[11] A. Jindal et al. Selecting subexpressions to materialize at datacenter scale. *PVLDB*, 11, 2018.

[12] A. Jindal et al. Peregrine: Workload optimization for cloud query engines. In *SoCC*, 2019.

[13] A. Jindal et al. Microlearner: A fine-grained learning optimizer for big data workloads at microsoft. In *ICDE*, 2021.

[14] A. Jindal et al. Production experiences from computation reuse at microsoft. In *EDBT*, 2021.

[15] T. Kraska. Towards instance-optimized data systems. *PVLDB*, 14(12), 2021.

[16] J. Leeka et al. Incorporating super-operators in big-data query optimizers. 13(3), 2019.

[17] V. Leis et al. How good are query optimizers, really? *PVLDB*, 9(3), 2015.

[18] P. Negi et al. Steering query optimizers: A practical take on big data workloads. In *SIGMOD*, 2021.

[19] A. Pimpley et al. Optimal resource allocation for serverless queries. 2021.

[20] Amazon Redshift. https://aws.amazon.com/redshift/.

[21] A. Roy et al. Sparkcruise: Workload optimization in managed spark clusters at microsoft. *PVLDB*, 14(12), 2021.

[22] R. Sen et al. Autotoken: Predicting peak parallelism for big data analytics at microsoft. *PVLDB*, 13(12), 2020.

[23] J. Shute et al. F1: A distributed sql database that scales. In *VLDB*, 2013.

[24] T. Siddiqui et al. Cost models for big data query processing: Learning, retrofitting, and our findings. In *SIGMOD*, 2020.

[25] Snowflake. https://www.snowflake.com/, 2021.

[26] Google Cloud Spanner. https://cloud.google.com/spanner, 2021.

[27] Apache Spark. http://spark.apache.org/, 2021.

[28] Dedicated SQL Pool. https://docs.microsoft.com/en-us/azure/synapse-analytics/sql-data-warehouse/sql-data-warehouse-overview-what-is, 2021.

[29] Apache Tez. https://tez.apache.org/, 2021.

[30] V. K. Vavilapalli et al. Apache hadoop yarn: Yet another resource negotiator. In *SoCC*, 2013.

[31] L. Viswanathan et al. Query and resource optimization: Bridging the gap. In *ICDE*. IEEE, 2018.

[32] C. Wu et al. Towards a learning optimizer for shared clouds. *PVLDB*, 12(3), 2018.