Smarter Warehouse

Nikolay Laptev, Meta, nlaptev@fb.com
Wenbo Tao, Meta, wenbo0@fb.com
Caner Komurlu, Meta, ckomurlu@fb.com
Jason Xu, Meta, zhenggang@fb.com
Deke Sun, Meta, deke@fb.com
Thomas C.H. Lux, Meta, tlux@fb.com
Luo Mi, Meta, miluo@fb.com

***Abstract*—Warehouse users often have to make too many decisions about their queries, pipelines, workflows and data to optimize the resources they use as well as the quality and the availability of their data. For example, whether to use Spark or Presto, how to best partition their data or what hyper-parameters to tune to resolve various query or pipeline problems. Furthermore, warehouse users are often unaware of big performance opportunities around data skew, multi-query optimization, query materialization and more. In this paper we describe the Smarter Warehouse initiative that aims to automate or simplify many of these optimization decisions. Our long term vision is for a large portion of the Smarter Warehouse optimizations to be seamlessly incorporated into the compute and I/O layers of the stack, leading to a simpler warehouse user experience and large amounts of resource savings.**

I. Introduction

We often rely on Warehouse users to optimize their queries and tune an increasing number of knobs in order to realize efficiency gains. What if the warehouse could automate its optimization process while simplifying the user experience?

In this paper, we describe the Smarter Warehouse project the goal of which is to make the warehouse more efficient and easier to use by replacing heuristics with automated solutions. We focus on sustainable efficiency wins in terms of compute savings while tracking usability/developer efficiency including '*hours of tuning effort*', '*maintenance*', '*days until stability*' and more.

The motivation for the Smarter Warehouse comes from the fact that data warehouse users have to make too many decisions about their queries, pipelines, workflows and data that significantly impact the resources they use as well as the quality and the availability of their data:

1) Which engine to use (e.g., Presto vs. Spark),
2) Various pipeline characteristics (e.g., decide whether to store in a managed table or use a view),
3) Which optimizations to apply to a given query (e.g., tune query hyper-parameters),
4) Perform Root Cause Analysis and other exploratory analysis (e.g., if a pipeline or workflow fails, figure-out why and fix),
5) Re-do the cycle when data volume or pipeline complexity changes.

The current trend in the data warehouse is unsustainable due to its fast growth of compute and storage required, and it is not surprising that capacity & efficiency is becoming a top priority in the industry. Everywhere that we give users the ability to make choices we raise a risk that many of them will make the wrong ones, often resulting in efficiency losses. The ideal smart infrastructure simplifies these decisions and pushes them "*under the hood*", leaving the users to focus on the query or ML model logic while still allowing for a rare manual tuning if desired. In this model, tuning, data storage, optimizations and fault tolerance are handled by the infrastructure itself which is seamlessly:

- deciding how to run jobs
- optimizing the usage of compute resources
- optimizing physical and logical storage of data
- optimizing queries
- monitoring, diagnosing and recovering from faults

These are ambitious goals and we propose to take an incremental approach, starting from automating simpler decisions and moving on to more advanced settings as we learn more and develop system expertise. The advantage of such a journey is that every step along the way can result in significant developer and system efficiency wins.

The industry is trending towards warehouse automation [3], [6], [8]–[10], [15]. For example authors in [2], [11], [19] explore the benefit of tunable parameters in a database system and emphasize the need for a system that learns from the parameters used by engineers. Authors in [16] discuss how leveraging data collected from previous query runs can be used to train ML models and recommend configurations that are as good or better than ones generated by human experts. Authors in [4] demonstrate how adaptive sampling of database parameters can be used to find good parameter values for improving memory distribution, I/O, query plans, parallelism and many aspects of logging. While these and other ongoing efforts tackle a part of the warehouse automation space, there does not yet exist a holistic approach that focuses on joint automation of warehouse queries, systems, data and pipelines.

By realizing the vision of making the warehouse smarter we have the potential to save significant resources in terms of memory, I/O and compute. This estimate is based on our initial results from hyper-parameter tuning, presented in Section II-A and observations such as the fact that 90% of our pipelines run daily, while the majority of produced tables are not queried daily.

A smarter warehouse will not only have impact on compute but also on engineering productivity, reducing the number of warehouse-related decisions that engineers have to make. While it is challenging to measure engineer efficiency wins, ultimately at the company level, people's efficiency matters far more than compute: the savings on people always have a much

2473-3490/22/$31.00 ©2022 IEEE
DOI 10.1109/ICDEW55742.2022.00005

Pillar	Description	Initiatives
Systems	Make core compute engines easier to use.	Hyper-parameter tuning
Queries	Focus engineers on query logic instead of engine-specific optimizations. Give engineers fast feedback and preempt inefficient (or destined-to-fail) queries.	Query Optimization
Pipelines	Pipelines run continuously and therefore have a lot of history in terms of data accesses that can be leveraged to perform additional optimizations.	Table materialization and virtualization
Data	Instead of only focusing on query history when performing optimizations, focus also on the underlying data to optimize partitioning, bucketing and more.	Auto-partitioning, auto-bucketing.
Insights	By analyzing a large set of computationally expensive queries, insights around common patterns can be learned.	Learning patterns from repeated query runs.

TABLE I
SUMMARY OF THE SMARTER WAREHOUSE PILLARS

higher multiplication factor and higher impact. Nevertheless, some of the ways we aim to measure people efficiency include tracking the number of relevant user group posts, number of code changes for a pipeline after launch, or number of pipelines per year per engineer.

These resource savings are distributed across data consumers and data producers. Data consumers benefit from the Smarter Warehouse in terms of higher data availability. Data producers benefit in terms of less complexity (e.g., when writing pipelines) and less service support load. Finally, warehouse providers benefit from the reduced resource spend on power, hardware and overall maintenance and planning.

Note that a Smarter Warehouse does not imply that we no longer need manual optimization tools. We still need to understand trends and there will still be optimization patterns that require a user's knowledge of the problem and intent. Specific pillars within the Smarter Warehouse include Smarter Systems, Queries, Pipelines, Data and Insights, a summary of which is provided in Table I and are described in more detail, with their applications in subsequent sections.

II. SMARTER SYSTEMS

Having learnable system components is important to achieving good performance in our datastore systems. One example of a system component that must be made learnable is query and application hyper-parameters, current progress of which is described in Section II-A. This is particularly challenging due to having many configuration knobs such as the number of shuffle partitions or the size of off-heap memory in Spark. Furthermore, the number of knobs in systems like Spark and Presto is constantly increasing with the new versions released. Many of these knobs are not independent, which means that changing one knob may affect the optimal setting for another one. It is hard enough for humans to understand the impact of one knob, let alone the interactions between multiple knobs. Lastly, one often cannot reuse the same configuration from one application/pipeline to the next. This is because the optimal configuration depends heavily on the pipeline's workload and the database server's underlying hardware. The best configuration for one pipeline may not be the best for another. Section II-A provides more details around Presto and Spark hyper-parameter tuning. There are other learnable components that we consider under the Smarter System umbrella including modifying the query scheduler with learned rules instead of a static collection of rules.

A. Hyper-Parameter Tuning (HPT)

While there are generally good heuristics for DBMS parameter tuning, these do not always provide best parameters for all systems and hardware configurations. Although one can rely on these rules for a set of DBMS, they are not universal. This is one of the reasons why companies resort to hiring expensive DB admins in order to perform knob-tuning. One common approach that such DBAs take is to copy the workload and perform offline experiments by tuning a set of knobs and observing performing. This trial-and-error is expensive and tedious for many reasons including (1) many of the knobs are not independent, (2) the values for some knobs are continuous, (3) one often cannot reuse the same configuration from one application to the next, and (4) DBMSs are always adding new knobs [18]. While there has been extensive research in the area of hyper-parameter tuning, our work differs from the past in terms of a tighter compute engine integration, specifically with Presto and Spark, as well as the focus on leveraging compute engine query plan to determine relevant parameters to optimize.

Compute engines such as Presto and Spark have plenty of knobs, also known as hyper-parameters, that affect query processing performances. Often these knobs are set to suboptimal values, which then yields drastically lower performance. This creates a need to tune knobs for performance improvements. Nevertheless, the configuration space is exponentially large in the number of knobs. Moreover, these knobs are often not independent from each other. Therefore changing the value of one knob may affect some other knobs' effect in query processing performance. Another difficulty is that a knob tuning that improves the processing performance of some queries may deteriorate performance on others.

In general, knob tuning relies on human expertise. Even in the best case scenario where an expert tunes a subset of knobs while having a very clear understanding of how they affect the processing performance, they will choose to validate performance improvements by rigorous experimentation. Typically, one needs to run a controlled experiment where a control group of queries are run with existing parameter settings while a test group is run with new parameter values. Unfortunately, multiple steps of this process, including but not limited to designing and running the experiment, collecting and analyzing the results require human intervention which renders the whole process too cumbersome to repeat considering the large space

of potential knob settings.

We started investigating a room for improvement for Presto and Spark engines. Our estimations have shown that the majority of parameters for both engines are set to default values and we can potentially have significant savings in terms of power. This estimate was based on creating a system that automatically generates code changes for pipelines where query-level hyper-parameters can be updated by running a shadow test which leverages Bayesian search to explore parameter search space efficiently.

This was then extended to auto set certain parameters based on historical statistics, such as *partial aggregation* and *join distribution type* in Presto, for all queries. The idea of partial aggregation is to run the aggregate's state transition function over different subsets of the input data independently, and then to combine the state values resulting from those subsets to produce the same state value that would have resulted from scanning all the input in a single operation [5]. Partial aggregation will have an unnecessary cost if the output size is not effectively reduced. Since most of the recurring batch processing jobs are very stable in terms of the amount of data processed from run to run, we leveraged the historical statistics of output to input size ratio to decide if the partial aggregation can be disabled automatically based on certain threshold.

For join distribution type optimization, although Presto already has cost based optimization (CBO), in a lot of cases the lack of good cardinality/size information of intermediate or even input tables makes it hard to make optimal decisions when broadcast join is applicable. Similarly, we also leveraged the historical statistics of input data size of join operators to guide the join type determination. One limitation of this strategy is that currently Presto only allows parameter tuning on the query level (*ie*, enable/disable partial aggregation for all the aggregation operators in one query). One possible resolution is replacing the operator-level cardinality estimates with historical statistics and then letting the optimizers themselves make better decisions.

We also experimented with setting cluster level parameters for Spark, which is in contrast to the query level parameter setting. For example we noticed that Spark queries spend 1/5 of their processing resources on garbage collection (GC) operations. We aimed to improve cluster level GC performance by experimenting with GC-related configurations such as GC-algorithm and GC-parallelism. While our search resulted in parameter settings that reduced GC time by 20%, we have not observed improvements in total processing times beyond our measured noise level.

We are also exploring leveraging open source parameter search initiatives to integrate with our Presto and Spark compute engines. For example the prototype system (BAO: Bandit optimizer) showed impressive results on both single node DBMS [12] and big data workload [14]. This system does not modify the optimizers directly. Instead, it steers the existing optimizers and rules to choose more promising query plans from ML predictions. We are exploring plugging it in Presto as a standalone service to generate more efficient query plans on the per query basis to achieve efficiency/latency wins and then extending it to work for Spark as well. We also plan to try leveraging the predictive model to mine more optimization insights for queries with certain patterns.

TABLE II
TESTED GARBAGE COLLECTION PARAMETERS

Algorithm	Parameter	Description	Values Tried
Parallel GC	ParallelGCThreads	Number of threads that do GC in parallel	2, 4 (default), 6, 8, 10, 12
G1 GC	InitiatingHeapOccupancyPercent	Maximum percentage of heap occupied before GC cycle starts	25, 35
	ConcurrentGCThreads	Number of concurrent GC threads	2, 4
	ParallelGCThreads	Same as above	6, 8

We have noticed that Presto and Spark hyper-parameter tuning can be unified behind a single interface of a standalone parameter tuning engine that we term *Warehouse HPT*. In the initial stage, the engine will conduct controlled experiments and analyze results to find insights that are statistically significant and stable improvement for at least a significant subset of queries. It will then leverage the experiment data to mine rules (e.g., via a simple decision tree model) to be applied to queries in production. The rule will be in the format of "if a query Q meets criteria A and B, then apply parameter value x, otherwise y". Thus *Warehouse HPT* will enable engine experts to list past parameter value trials with their analyses, and to seamlessly invoke new parameter value trials with a few clicks, without dealing with the details of preparing and conducting controlled experiments, collecting results, running an analysis on the results, and trying to come up with rules that allow to yield maximum benefit from the parameter change. Overall, by extending our Presto and Spark engines with the open source HPT techniques we believe that parameter setting for our warehouse will be made easier and more efficient for both our systems and our customers.

For Presto, where we tuned per query partial aggregation and join distribution type parameter, we had seen initial results of $\sim$3% CPU improvement and 2-3% latency wins over the entire production workload. For Spark Garbage Collection (GC) tuning we have observed improvements in GC time but not in overall CPU. The details of tuned parameters are presented in Table II.

Table II shows a subset of algorithms and parameters with their values we experimented to improve garbage collection for Spark. Our analyses showed that particularly ParallelGC with ParallelGCThread equal 8 improves GC time by around 20%, yet we could not secure a consistent improvement in CPU usage.

III. SMARTER QUERIES

Classical query optimization focuses on crafting heuristic rules around best join type or join ordering to use or how to perform predicate push-down [17]. While these optimizations are still relevant to the *Smarter Warehouse*, our goal is to automatically learn the best optimizations to apply in order to create the most optimal query plan.

Engineers writing a query often have to worry about executing the given query on Spark, Presto or another compute engine each with a different set of parameters (see Smarter Systems section above). Instead, our vision is to have a common query optimizer that not only decides which batch compute engine to use, but may also decide whether to perform some early computation in WWW/logging stack, in streaming prior to ingestion, at ingestion time, or in NRT/micro-batch allowing for easier query/pipeline authoring. This will extend the current rule-and-heuristic-based optimizer with learned components for more efficient query plan generation.

An example of where the common query optimization layer will be useful is in interactive data analysis, which is often inundated with common computations across multiple queries. These redundancies considerably increase the resource cost for interactive query sessions. Reusing common computations could lead to substantial cost savings, but it is difficult for the users to manually detect and reuse the common computations themselves (especially during ad hoc analyses).

Most of our batch workload consists of the same queries running day after day, which makes learning query optimizations possible. In fact, the majority of today's workload is from pipelines over a year ago. Thus we (a) have plenty of data to train a query-specific optimizer and (b) the resource savings we attain will accrue over a long period of time. Sections III-A, III-B, III-C and III-D demonstrate our initial prototype for query optimization.

A. Query Optimization Overview

Traditionally, query optimizations are done in database engines (i.e. Presto and Spark). While eventually we want to enable smarter warehouse optimizations within these engines, our current query optimization efforts focus on a variety of SQL-to-SQL rewrites outside of the engines. The reasons for this are four-fold.

First, query optimizers in the database engines are conventionally cost-based, and thus do not take into account historical stats which can be helpful in query optimization. For example, computationally expensive join orderings can potentially be corrected by looking at past runs. As another example, looking at the frequency of subqueries in the past is helpful in deciding what materialized views to create [1].

Second, we can afford running shadow tests to see if an optimization works. This can be worthwhile because most queries are recurring, including dashboard queries and hourly/daily/weekly ETL jobs.

Third, we can propose bolder and riskier code change optimizations to the users, and only ship the optimizations if users accept the code changes. For example, the use of the UNION clause without the ALL clause requires a computationally expensive deduplication process. Oftentimes this is not intended. Users just forget to add ALL. However, it is still not 100% safe to perform a transparent optimization that adds ALL regardless. In this case, we assign a ticket to the query owner to let them double check if deduplication is really desired.

Lastly, proposing query optimizations as code changes has a great benefit of facilitating better engineering practices. Given the scale of the warehouse at large companies, it is easy to find an existing piece of code and reuse it in a different context. This is a common practice adopted by users, especially new engineers. As a result, anti-patterns propagate rapidly through copy-and-paste engineering. By automatically proposing code changes and asking the users to review, we are able to raise the awareness of anti-patterns and promote good practices.

B. Example Optimization Patterns

We are currently working on extensive single and multi-query optimization patterns. Here we describe some examples of the SQL-to-SQL rewrite patterns.

Missing Date Filter. In the data warehouse, tables are often partitioned on the date column by default. A query usually accesses only one or a few date partitions by specifying a filter on the date column. Sometimes users forget to specify the date filter, leading to computationally expensive scans of all data. We detect the missing date filters and assign tickets to users asking them to check whether this is intended. Most of the time, the users add a date filter after seeing the ticket and it saves a lot of computing resources.

Inefficient Expressions. In many cases, a computationally expensive expression can be replaced/removed without affecting downstream application logic. For example, users would write the ORDER BY clause in an ETL INSERT query, which is often the result of copy & paste and not necessary. Now, consider an example of "risky" optimization that needs user confirmation. COUNT DISTINCT is very computationally expensive, but in many cases an exact count of distinct values is not needed. An approximate count offered by APPROX_DISTINCT can be sufficient and much more efficient.

Unnecessary Sharding. Query sharding is a "trick" well-known in the company to avoid out-of-memory errors in Presto. Basically, users can breakdown a computationally expensive INSERT query into smaller ones by appending WHERE predicates to it. For example, users would rewrite *INSERT INTO a SELECT * FROM b* as 40 smaller queries *INSERT INTO a SELECT * FROM b WHERE id % 40 = i,* $0 \leq i < 40$. Presto can potentially be architectured to support this pattern better, but this trick has become so popular that sometimes it is the default thing to do whenever the user feels that a query is going to run out of memory, even though it may not run out of memory at all. We detect such patterns and run shadow tests to show to the users that the query sharding is not necessary. Users would often accept our suggestion. Based on our initial results we have saved on average over 60% of compute by leveraging these rewrite strategies for this pattern.

C. Action Recommendations for Warehouse Users

Data warehouse users spend time on a variety of internal tooling systems. Ideally, we want action recommendations based on the optimization patterns (e.g. add missing date filter) to be visible as much as possible. While we have a few systems

presenting action recommendations in a variety of ways such as assigning tickets, UI popups and linting suggestions, they are far from unified, and present an inconsistent and fragmented user experience. Looking ahead, we envision a future where a centralized engine dispatches action recommendations that cover every user surface possible, enabling a coherent and consistent user experience and eliminating anti-patterns to the greatest extent possible.

D. Query Cost Prediction

Predicting query execution time is useful in many database management issues including admission control, query scheduling, progress monitoring, and system sizing [20]. As part of the Smarter Warehouse we are also working on a machine learned model for query cost prediction. Our initial results show that using a very simple and fast machine learning model, we can predict 20% of all query failures with 95% precision (before the query is executed). If deployed only to our interactive query operations, this model might reduce the amount of time the average user spends every day waiting on failed queries by over 10 minutes.

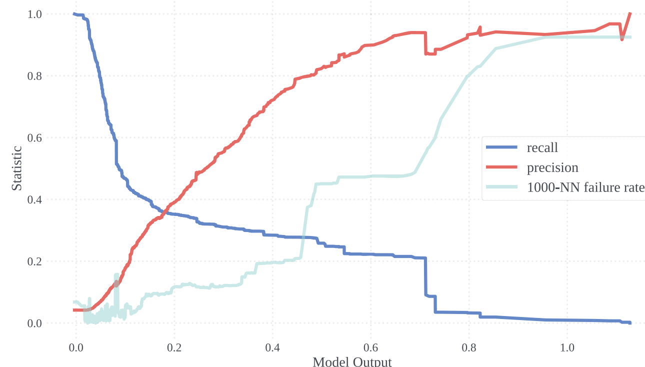
Predicting when a query will succeed or fail enables us to warn users, saving time they would otherwise lose to waiting (that amounts to tens of minutes per user-day and thousands of engineering hours every day). Alternatively, we can redirect the destined-to-fail queries to a different queue in the engine to reduce the side effects of the hundreds of thousands of CPU hours a day spent on failed interactive queries.

Before a SQL query gets executed by our query engines (Presto and Spark), it is first compiled into a "query plan" that represents the set of basic operators (scan a file, join on a key, ...) for execution. These plans take very little time to generate compared to the cost of running the query ($\ll$1%). In addition to the query plan, we use a meta-store to get information about the size of the input tables and other such statistics. These two sources contribute the only features currently in use. Importantly, no features related to the SQL, join ordering, or filters that a user applied in the query are utilized, which are the features that intuitively would be best at predicting success and failure (we plan to use them in the future).

After training the model we are seeing a peak classification accuracy of >95% on test data, but what is more important is observing the model's ability to separate those queries that are likely to succeed from those that are likely to fail. When applying the model we produce a query failure likelihood (using nearest neighbor over the model outputs on the training data) that can be used for binning and making warnings for users. Figure 1 shows the the precision and recall curves for this initial model predicting query failures.

The current prototype model is using a very small set of features, so we are hopeful that this already-promising performance can be significantly improved. For training we sampled one hundred thousand interactive queries from a two week window, and fit a multilayer perceptron (MLP) with 10 layers and 128 nodes per layer (totaling 150K parameters at

Fig. 1. Precision, recall, and 1000 nearest-neighbor query failure rate curves based on model output on test data (where output being larger means a query is more likely to fail). A very simple model based on query plans and input table sizes alone can capture 20% of failures with 95% precision. This model was trained on 100K randomly sampled queries over a two week time period and tested on 100K queries from the second day after the training time period. In addition to confidently capturing 20% of failures, the 2-norm neighborhoods around model outputs are relatively homogeneous and allow for "failure likelihood" estimation via nearest neighbor over the training data.



a size of 600KB). A single forward pass (prediction) takes only 7 milliseconds. To test the model we collected a new set of one hundred thousand of queries from two days after the training data window, which prevents any feature spillage and ensures we have time to train a model similarly in production.

We have deployed this model as part of a service to make the predictions accessible to internal tools. The service takes as input the query plan JSON and the list of input table names for the query. From there it parses out the features and passes them into a model, returning a prediction that includes the likelihood of success. By making the model available through a service, it will allow rapid testing and integration with compute engines and other tools in our warehouse.

IV. Smarter Pipelines

Pipelines at large companies often consist of multiple queries that are currently executed completely independently even though there might be shared data, predicates and other logic among them. For example, previous work observed significant overlaps in the computations performed by user jobs in modern shared analytics clusters [4]. Naively computing the same sub-expressions multiple times results in wasting cluster resources and longer execution times. Given that these shared cluster workloads consist of tens of thousands of jobs, identifying overlapping computations across jobs is of great interest to both cluster operators and users. Therefore learning to select common parts of job plans and materializing them to speed-up the evaluation of subsequent jobs will be useful in making our pipelines smarter. *LINT* checks can also suggest if a new pipeline tries to repeat an already existing (sub)query.

Another item that is part of the smarter pipelines umbrella is learning to use views instead of managed tables. Using views instead of managed tables is an important database optimization. For example a lot of the times table T is created but it is only read by the downstream task in the same pipeline

and never again. For cases like this, having a view for *T* is likely better in terms of both storage as well as potentially I/O and CPU. Initial estimate of the number of pipeline-generated tables that can be replaced by views is $> 5\%$. This estimate is based on the table access frequency in the last 30 days. Replacing tables with views may allow for not only a reduced I/O but also a reduced storage and compute. A similar parallel direction would be for Presto or Spark to recognize that a materialized view could better serve a query with a matching filter predicate, rather than use the base table in Hive. In other words, materialized views are often preferred in cases where the cardinally of the query output reduces substantially compared to the input [13] due to filtering or aggregation and we aim to automatically determine such opportunities with the Smarter Warehouse. See section IV-A for more details on our initial prototype.

Smarter pipeline root cause analysis (RCA) is another item in this pillar. Currently engineers need to scan potentially hundreds of metrics to find out performance issue symptoms. Certain metrics are highly correlated and rapid fault propagation in databases renders them anomalous almost simultaneously making debugging more challenging due to the added anomaly noise. As a result, pipeline debugging with various metric fluctuation patterns likely leads to complex relationships with diverse root causes. Another pipeline optimization opportunity involves mining pipelines with a high failure rate and auto-pausing them until addressed by the user or recommending switching to a different engine (e.g., from Presto to Spark to avoid Out-Of-Memory errors (OOMs)). To discover and untangle such relationships, developing smarter pipeline analysis tools is our first step in this direction.

A. Infrequently Accessed Table Optimization

In this section we describe an infrequently accessed table optimization based on virtual views, which is orthogonal to a materialized view optimization. Often data warehouse users create infrequently accessed/temp tables in their pipelines or create many tables that are largely based on the same query but with a different filter. These tables consume storage resources and potentially increase consuming query complexity.

The scale of the problem is quite large. For example there are thousands of tables that have been accessed less than 30 times in the last month. Completely removing such tables may be challenging due to downstream dependencies. Therefore, as part of the Smarter Warehouse direction, we want to explore a scalable way of replacing some of these tables by *virtual views* thereby reducing complexity by hiding the underlying physical storage semantics and improving storage efficiency by not materializing the table.

The idea of using views instead of tables is very important in database optimization. For example often table *T* is created but it is only read by the downstream task in the same pipeline and never again. For cases like this, having a view for *T* is likely better in terms storage. Views can also help in complexity reduction: if we have tables $T_1 ... T_k$ generated using queries $Q_1 ... Q_k$ that are similar, then we can use a single view that can potentially replace all or some of $Q_1 ... Q_k$. Thus, by leveraging views, the logic which would otherwise appear as boilerplate code can now be consolidated, centralized, and manageable. Industry-wide, views are starting to gain popularity, but they are still not widely adopted, partly due to the lack of awareness.

Our goal is to explore automated table to view conversion in selected instances to reduce complexity as well as improve storage efficiency. One challenge that we also had to overcome to make Virtual Views truly a first class citizen in the warehouse is to add lineage support. Lineage information is useful to understand how data is flowing through view tables. With this lineage support, we have more confidence to push higher adoption of view tables in the warehouse. This adoption can reduce physical storage and sensitive information for infrequently accessed tables.

As part of this effort, we have landed multiple pipeline rewrites. These rewrites remove the need to persist the intermediate table and instead use a *VIRTUAL_VIEW*. By using a *VIRTUAL_VIEW* we have saved many petabytes of logical storage across the many clusters. Finally, we are closely monitoring the updated pipeline performance and expect the compute usage to remain largely unchanged. The reason for this is because in the updated pipeline *Table A* is used to generate *Table B* and when we convert *Table A* into a view, thereby reducing *Table A*'s computationally cost to 0, the increased resource cost of generating *Table B* is offset by the computational savings we got from converting *Table A* into a view.

V. Smarter Data

While it is common to leverage previous query runs to learn potential query optimization patterns [21] leveraging the underlying data is less explored and is often beneficial for modeling query cardinality without incurring expensive query execution costs.

Smarter data implies learning optimal physical data layout which includes bucketing, partitioning, column ordering, data compression and auto data type selection to improve query performance based on the historical workload [7]. For example partition tuning, i.e., selecting partition columns that are appropriate for the workload, is a crucial task that is often set to a default value by users. However, selecting the right partition is a very difficult optimization problem: there exists a very large number of candidate partitions for a given schema, a given partition may benefit some parts of the workload and also incur storage overhead and other complexity in terms of the number of directories created due to partition columns selected.

Therefore a learning system may choose to generate partition recommendations by analyzing the workload online (i.e., in parallel with query processing), which allows the recommendations to adapt to shifts in the running workload. The pipeline owner should also be able to request a recommendation at any time and is responsible for selecting the

partition at the time of pipeline creation. Another partition-based optimization is to compute the partition access ratio and warn users if this ratio is too low (e.g., a given partition is not frequently accessed). As with previous learning components of the Smarter Warehouse, the engineer should be able to refine the automated recommendations by passing indirect domain knowledge to the tuning algorithm.

VI. Query Insights & Other Optimizations

Smarter warehouse exploration and experimentation tools will allow us to understand how engineers are using the warehouse and will allow us to bring together the many disparate datasets into a unified view in order to analyze the various warehouse patterns to better plan and prioritize opportunities towards making our warehouse smarter. This includes building tools that allow for an easier way to explore engine, query, pipeline and data characteristics to understand what parts of it may need tuning. While this may not lead to immediate measurable gains, building such tools is important as it will allow engineers to explore how the warehouse is used and to understand optimization gaps and ways to potentially address them [8]. For example this would involve exploring warehouse component usage and deprecating components that are infrequently used or simplifying frequently used components (e.g., reducing the number of required operator flags).

Other Optimizations: In addition to the major Data Warehouse efficiency-related initiatives (e.g., HPT) mentioned earlier in this manuscript, many additional anti-patterns exist. For these anti-patterns that are not directly and explicitly covered in the top-down Smarter Warehouse roadmap, an alternative bottom-up approach is taken to enable Warehouse power users to report these inefficiencies and to help with remediation efforts. The Smarter Warehouse's role here is to help streamline this entire process.

A. Tables with Low Daily Partition Access Ratio

New partitions are generated daily for most tables in the majority of data warehouses. However, many of these partitions are never accessed, which wastes computation and storage. We used the ratio of daily partitions that were accessed to detect this anti-pattern. We term this the "partition access ratio." We automatically send a message to owners of tables that are resource intensive to compute and/or store that have a ratio below a threshold (e.g., 0.3) advising them to write out partitions less frequently (e.g., every other day).

B. Error-Prone Pipelines

Pipeline owners in the data warehouse are advised when their pipelines fail to complete successfully due to a system error. If an owner ignores this message repeatedly, it's a signal that the pipeline is no longer producing business-critical data. In these cases as well, we recommend a reduction in pipeline frequency.

C. Pipelines with High Resource Consumption Failures

Resource failures (e.g. Out-Of-Memory, exceed time limit) are often transient errors in the Data Warehouse and, if given enough retries, may eventually go away. These eventually successful runs can give the owners a false sense of pipeline robustness leaving them unaware of the large CPU waste incurred from the previous failed runs. In these cases, we surface pipelines that are associated with particularly high failure CPU and recommend appropriate remediation methods (e.g., switching from Presto to Spark).

VII. Discussion and Conclusion

The focus of this workshop paper is to introduce the vision for the *Smarter Warehouse* and the initial steps we are taking in this direction. Along this journey, there are a number of lessons that we have learned:

- *Importance of computational savings estimation*: having a reliable way to estimate potential wins of an optimization direction allows for easier prioritization.
- *Unify common project frameworks*: Unifying common parts of the hyper-parameter tuning (HPT) library as well as query optimization library for Presto and Spark will allow us to move faster.
- *Compute engine alignment*: There was a lot of discussion around implementing HPT / query optimization within the engine vs. outside of the engine. It is important for us to align on this across the compute engines.
- *Breadth vs. depth strategy to maximize impact*: Warehouse space has huge opportunities for machine learning impact. To make sure we are working on the most impactful projects, we must have dedicated time to explore riskier projects such as query cost prediction.
- *Focusing on resource savings as an important metric of progress*: To align with company/org priorities and needs we must align on efficiency wins versus measuring usability.
- *How to systematically source new projects and scope*: So far we have been choosing Smarter Warehouse projects in an ad hoc way, however the goal is to make the process more systematic by leveraging internal programs to find insights in our query authoring workflows that can benefit from machine learning optimization.
- *Aligning with cross-functional teams*: It is important to make sure we can push out intelligence to the end user in a scalable way and to that end our partnership with internal compute engine and efficiency teams is important.

While we have introduced a number of initiatives in this workshop paper, ultimately it is important to unify the presented optimizations in order to have a holistic solution for the *Smarter Warehouse*. This includes bridging the potential conflict between HPT, smarter queries, systems & pipelines, which may be applicable to specific use-cases (e.g., interactive and batch queries may need different set of optimizations and a "one size fits all" approach will likely not be applicable).

The progress we have made so far has already resulted in sizable power savings and we aim to share further progress with the Database community in the near future.

References

[1] D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek. Efficient view maintenance at data warehouses. *SIGMOD Rec.*, 26(2):417–427, jun 1997.

[2] S. Chaudhuri and G. Weikum. Rethinking database system architecture: Towards a self-tuning risc-style database system. VLDB '00, page 1–10, 2000.

[3] S. Chaudhuri and G. Weikum. Foundations of automated database tuning. SIGMOD '05, page 964–965, 2005.

[4] S. Duan, V. Thummala, and S. Babu. Tuning database configuration parameters with ituned. *Proc. VLDB Endow.*, 2(1):1246–1257, aug 2009.

[5] T. P. G. D. Group. *Documentation PostgreSQL 10.3*, 2018.

[6] M. Haynie. A dbms for large design automation databases. SIGMOD '88, page 269–276, New York, NY, USA, 1988. Association for Computing Machinery.

[7] B. Hilprecht, A. Schmidt, M. Kulessa, A. Molina, K. Kersting, and C. Binnig. Deepdb: Learn from data, not from queries! 13(7):992–1005, mar 2020.

[8] G. Li, X. Zhou, and L. Cao. Machine learning for databases. AIML-Systems 2021, 2021.

[9] G. Li, X. Zhou, J. Sun, X. Yu, Y. Han, L. Jin, W. Li, T. Wang, and S. Li. Opengauss: An autonomous database system. *Proc. VLDB Endow.*, 14(12):3028–3042, jul 2021.

[10] G. M. Lohman and S. S. Lightstone. Smart: Making db2 (more) autonomic. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, page 877–879. VLDB Endowment, 2002.

[11] L. Ma, W. Zhang, J. Jiao, W. Wang, M. Butrovich, W. S. Lim, P. Menon, and A. Pavlo. *MB2: Decomposed Behavior Modeling for Self-Driving Database Management Systems*, page 1248–1261. Association for Computing Machinery, New York, NY, USA, 2021.

[12] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska. Bao: Making learned query optimization practical. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1275–1288, 2021.

[13] H. Mistry, P. Roy, K. Ramamritham, and S. Sudarshan. Materialized view selection and maintenance using multi-query optimization. *CoRR*, cs.DB/0003006, 2000.

[14] P. Negi, M. Interlandi, R. Marcus, M. Alizadeh, T. Kraska, M. Friedman, and A. Jindal. Steering query optimizers: A practical take on big data workloads. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2557–2569, 2021.

[15] S. Schelter, D. Lange, P. Schmidt, M. Celikel, F. Biessmann, and A. Grafberger. Automating large-scale data quality verification. 11(12):1781–1794, aug 2018.

[16] K. Schnaitter and N. Polyzotis. Semi-automatic index tuning: Keeping dbas in the loop. 5(5):478–489, jan 2012.

[17] R. Sethi, M. Traverso, D. Sundstrom, D. Phillips, W. Xie, Y. Sun, N. Yegitbasi, H. Jin, E. Hwang, N. Shingte, and C. Berner. Presto: Sql on everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1802–1813, 2019.

[18] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic database management system tuning through large-scale machine learning. SIGMOD '17, page 1009–1024, New York, NY, USA. Association for Computing Machinery.

[19] G. Weikum, A. Moenkeberg, C. Hasse, and P. Zabback. Self-tuning database technology and information services: From wishful thinking to viable engineering. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, page 20–31. VLDB Endowment, 2002.

[20] W. Wu, Y. Chi, S. Zhu, J. Tatemura, H. Hacigümüs, and J. F. Naughton. Predicting query execution time: Are optimizer cost models really unusable? In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 1081–1092, 2013.

[21] Z. Zolaktaf, M. Milani, and R. Pottinger. Facilitating sql query composition and analysis. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 209–224, New York, NY, USA, 2020. Association for Computing Machinery.