Computation Reuse in Analytics Job Service at Microsoft

Alekh Jindal, Shi Qiao, Hiren Patel, Zhicheng Yin, Jieming Di, Malay Bag, Marc Friedman, Yifung Lin, Konstantinos Karanasos, Sriram Rao

Microsoft

{aljindal,shqiao,hirenp,zhyin,jiedi,malayb,marc.friedman,yifungl,kokarana,sriramra}@microsoft.com

ABSTRACT

Analytics-as-a-service, or analytics *job service*, is emerging as a new paradigm for data analytics, be it in a cloud environment or within enterprises. In this setting, users are not required to manage or tune their hardware and software infrastructure, and they pay only for the processing resources consumed per job. However, the shared nature of these job services across several users and teams leads to significant overlaps in partial computations, i.e., parts of the processing are duplicated across multiple jobs, thus generating redundant costs. In this paper, we describe a computation reuse framework, coined CloudViews, which we built to address the computation overlap problem in Microsoft's SCOPE job service. We present a detailed analysis from our production workloads to motivate the computation overlap problem and the possible gains from computation reuse. The key aspects of our system are the following: (i) we reuse computations by creating materialized views over *recurring workloads*, i.e., periodically executing jobs that have the same script templates but process new data each time, (ii) we select the views to materialize using a *feedback loop* that reconciles the compile-time and run-time statistics and gathers precise measures of the utility and cost of each overlapping computation, and (iii) we create materialized views in an *online* fashion, without requiring an offline phase to materialize the overlapping computations.

CCS CONCEPTS

• **Information systems → Query optimization**; • **Computer systems organization → Cloud computing**;

KEYWORDS

Materialized Views; Computation Reuse; Shared Clouds

ACM Reference Format:
Alekh Jindal, Shi Qiao, Hiren Patel, Zhicheng Yin, Jieming Di, Malay Bag, Marc Friedman, Yifung Lin, Konstantinos Karanasos, Sriram Rao. 2018. Computation Reuse in Analytics Job Service at Microsoft. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3183713.3190656

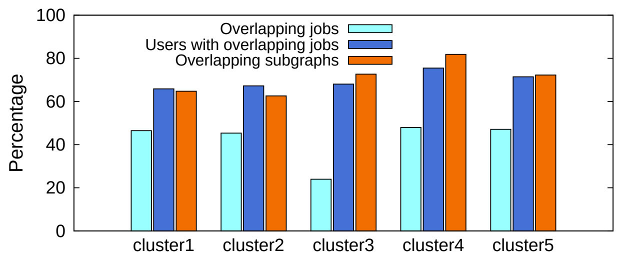


Figure 1: Overlap in different production clusters at Microsoft.

1 INTRODUCTION

1.1 Background

There is a recent trend of offering analytics-as-a-service, also referred to simply as *job service*, by major cloud providers. Examples include Google's BigQuery [15], Amazon's Athena [3], and Microsoft's Azure Data Lake [5]. Similar job services are employed for the internal needs of large enterprises [11, 49]. These services are motivated by the fact that setting up and running data analytics is a major hurdle for enterprises. Although platform as a service (PaaS), software as a service (SaaS), and more recently database as a service (DBaaS) [4, 6] have eased the pain of provisioning and scaling hardware and software infrastructures, users are still responsible for managing and tuning their servers. A job service mitigates this pain by offering *server-less* analytics capability that does not require users to provision and manage servers. Instead, the service provider takes care of managing and tuning a query engine that can scale instantly and on demand. Users can get started quickly using the all familiar SQL interface and pay only for the processing used for each query, in contrast to paying for the entire provisioned server infrastructure irrespective of the compute resources actually used.

1.2 Problem

Given the above shift from provisioned resources to actually consumed resources, enterprises naturally do not want to duplicate their resource consumption and pay redundant costs. However, this is a major challenge in modern enterprise data analytics which consists of complex data pipelines written by several users, where parts of the computations end up running over and over again. Such computation overlap not only adds to the cost, but it is also really hard for the developers or even the administrators to detect these overlaps across different scripts and different users.

To illustrate the problem, consider SCOPE [11, 52], which is the equivalent of Azure Data Lake for internal data analytics at Microsoft. SCOPE is deployed over hundreds of thousands of machines, running hundreds of thousands of production analytic jobs per day that are written by thousands of developers, processing several exabytes of data per day, and involving several hundred petabytes of I/O. Almost 40% of the daily SCOPE jobs have computation overlap with one or more other jobs. Likewise, there are millions of overlapping subgraphs that appear at least twice. These overlaps are incurred by 70% of the total user entities (humans and machines) on these clusters. Figure 1 shows the cluster-wise computation overlap in five of our clusters. We can see that all clusters, except cluster3, have more than 45% of their jobs overlapping. Likewise, more than 65% of users on all clusters end up having computation overlap in their jobs and the percentage of subgraphs appearing at least twice could be as high as 80%. While the ideal solution would be for the users to modularize their code and reuse the shared set of scripts and intermediate data, this is not possible in practice as users are distributed across teams, job functions, as well as geographic locations. Thus, we need an automatic cloud-scale approach to computation reuse in a job service.

1.3 Challenges

There is a rich literature for materializing views [19, 20, 22, 30, 33, 44, 46, 53] and for reusing intermediate output [10, 12, 18, 23, 36–38, 50]. However, there are a number of new challenges in building a computation reuse framework for the SCOPE job service.

First, enterprise data analytics often consists of **recurring jobs** over *changing* data. The SCOPE job service has more than 60% of the jobs in its key clusters as recurring [25]. With recurring jobs, scheduling and carefully materializing views over the new data is crucial, which was not an issue in traditional view selection. Incremental maintenance would not work because data might be completely new. SCOPE jobs are further packed in tight data pipelines, i.e., multiple jobs operate in a given time interval with strict completion deadlines. Tight data pipelines leave little room to analyze the recurring workload over the new data in each occurrence.

Second, we need a **feedback loop** to analyze the previously executed workload and detect overlapping computations. Given the large volume of overlaps, materializing all of them for reuse is simply not possible. Typical methods to select the interesting overlaps (or views) depend on the utility and cost of each overlap, i.e., the runtime savings and the storage cost of each overlap. Unfortunately, however, the optimizer estimates for utility and costs are often way off due to a variety of factors (unstructured data, inaccurate operator selectivities, presence of user code, etc.) [17, 29, 31]. Thus, the feedback loop needs to reconcile the logical query trees with the actual runtime statistics to get more precise measures of utility and cost of each overlap.

Third, a job service is always **online** and there is no offline phase available to create the materialized views, which is expected with traditional materialized views. Halting or delaying recurring jobs to create materialized views is not an option, as it carries the risk of not meeting the completion deadlines and affecting downstream data dependency. Thus, we need to create materialized views just in time and with minimal overheads. This is further challenging because multiple jobs can now compete to build views (build-build interaction), and they depend on each other for the availability of views (build-consume interaction).

Finally, we need an **end-to-end system** for computation reuse that has a number of requirements, including automatic reuse and transparency to the end users, that are inspired from our production environments.

1.4 Contributions

In this paper, we describe why and how we built an end-to-end system for automatically detecting and reusing overlapping computations in the SCOPE job service at Microsoft. Our goal is to allow users to write their jobs just as before, i.e., with zero changes to user scripts, and to automatically detect and reuse computations wherever possible. We focus on exact job subgraph matches, given that exact matches are plentiful and it makes the problem much simpler without getting into view containment complexities. Although we present our ideas and findings in the context of the SCOPE job service, we believe that they are equally applicable to other job services. Our core contributions are as follows.

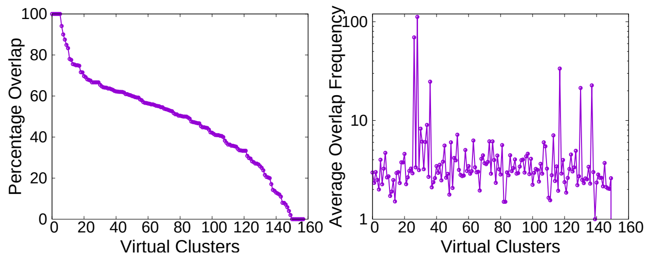
First, we present a detailed analysis of the computation reuse opportunity in our production clusters to get a sense of the magnitude of the problem and the expected gains. Our analysis reveals that computation overlap is a major problem across almost all business units at Microsoft, with significant runtime improvements to be expected with relatively low storage costs. We also note that the overlaps often occur at shuffle boundaries, thereby suggesting that the physical design of the materialized view is important (**Section 2**).

Then, we discuss enabling computation reuse over recurring jobs. The key idea is to use a combination of normalized and precise hashes (called *signatures*) for computation subgraphs. The normalized signature matches computations across recurring instances, while the precise signature matches computations within a recurring instance. Together these two signatures enable us to analyze our workload once and reuse overlapping computations over and over again (**Section 3**).

We provide an overview of our CLOUDVIEWS system, an end-to-end system for computation reuse in a job service, along with our key requirements and the intuition behind our approach. The CLOUDVIEWS system consists of an offline CLOUDVIEWS analyzer and an online CLOUDVIEWS runtime. To the best of our knowledge, this is the first work to present an industrial strength computation reuse framework for big data analytics (**Section 4**).

We describe the CLOUDVIEWS analyzer for establishing a feedback loop to select the most interesting subgraphs to materialize and reuse. The CLOUDVIEWS analyzer captures the set of interesting computations to reuse based on their prior runs, plugs in custom view selection methods to select the view to materialize given a set of constraints, picks the physical design for the materialized views, and also determines the expiry of each of the materialized views. We further describe the admin interface to trigger the CLOUDVIEWS analyzer (**Section 5**).

We describe the CLOUDVIEWS runtime which handles our online setting for computation reuse. Key components of the runtime include a metadata service for fetching the metadata of computations relevant for reuse in a given job, an online view materialization mechanism as part of the job execution, a synchronization mechanism to avoid materializing the same view in parallel, making materialized views available early during runtime, automatic query rewriting using materialized views, and job coordination hints to maximize the computation reuse (**Section 6**).



(a) Percentage overlap (b) Average overlap frequecy

Figure 2: Overlap in one of the largest SCOPE clusters

Thereafter, we present an experimental evaluation of CloudViews. We present the impact over production workloads at Microsoft, both in terms of latency and CPU hours. Our results show an average and overall latency improvement of 43% and 60% respectively, as well as an average and overall CPU hour improvement of 36% and 54% respectively. We further show evaluation over the TPC-DS benchmark. Our results show 79 out of the 99 TPC-DS queries having improvements with CloudViews, with an overall runtime improvement of 17%. We also discuss the various overheads of CloudViews, including the cost of workload analysis, the metadata lookup, and the impact on compiler runtime (**Section 7**).

Finally, we discuss the lessons learned from the CloudViews project. (**Section 8**)

2 THE REUSE OPPORTUNITY

We saw the overall overlap across our SCOPE clusters in Figure 1. Below we analyze the overlap at different granularity levels.

2.1 Overlap within a Cluster

We analyze one of the largest clusters, in terms of the number of jobs, to better understand the overlap. Figure 2(a) shows the percentage of jobs overlapping in each of the virtual clusters[1] (VCs) in this physical cluster. Our analysis shows that while some VCs have no overlapping jobs, 54% of the VCs have more than 50% of their jobs overlapping, and few others have 100% of their jobs overlapping. Figure 2(b) shows the average overlap frequencies across different virtual clusters. The average overlap frequency ranges from 1.5 to 112 (median 2.96, 75^{th} percentile 3.82, 95^{th} percentile 7.1). The key lesson here is that computation overlap is a cluster-wide phenomenon and not limited to specific VCs or workloads.

Our interactions with the internal SCOPE customers reveal two main reasons for the prevalence of computation overlap seen above: (i) users rarely start writing their analytics scripts from scratch, rather they start from other people's scripts and extend/modify them to suit their purpose, (ii) there is a data producer/consumer model at play in SCOPE, where multiple different consumers process the same inputs (generated by the producers), and they often end up duplicating the same (partial or full) post-processing over those inputs.

2.2 Overlap within a Business Unit

We further analyze one of the largest business units, in terms of the number of jobs, in the above cluster. Figure 3 shows the overlapping computations from all VCs in this business unit. Note that business unit is a meaningful granularity because VCs within a business unit compose a data pipeline, with some VCs cooking the data (producers) and some VCs processing the downstream data (consumers).

Figures 3(a)–3(d) show the cumulative distributions of per-job, per-input, per-user, and per-VC overlaps. Surprisingly, we see that most of the jobs have $10s$ to $100s$ of subgraphs that overlap with one or more other jobs. This suggests that there are significant opportunities to improve the data pipelines in order to reduce the redundancy. Apart from reusing computations, one could also consider sharing computations in the first place. We make a similar observation from per-input overlap distribution, where we see that more than 90% of the inputs are consumed in the same subgraphs at least twice, 40% are consumed at least five times, and 25% are consumed at least ten times. In terms of users, we again see $10s$ to $100s$ of overlaps per user, with top 10% having more than 1500 overlaps. These heavy hitters could be consulted separately. Lastly, for VCs, we see at least three groups having similar number of overlaps. Overall, computation overlap is widespread across jobs, inputs, users, and VCs, and it needs to be addressed in a systematic manner.

2.3 Operator-wise Overlap

We now analyze the operator-wise overlap, i.e., the root operator of the overlapping computation subgraph. Figure 4(a) shows the operator distribution for the overlaps shown in Figure 3. We can see that sort and exchange (shuffle) constitute the top two most overlapping computations. This is interesting because these two are typically the most expensive operations as well and so it would make sense to reuse these. In contrast, the next three most overlapping operators, namely Range (scan), ComputeScalar, and RestrRemap (usually column remapping), are expected to be much cheaper to re-evaluate, since they are closer to the leaf-level in a query tree. Among other operators of interest, we see group-by aggregate, joins, and user defined functions (including process, reduce, and even extractors) having significant overlaps.

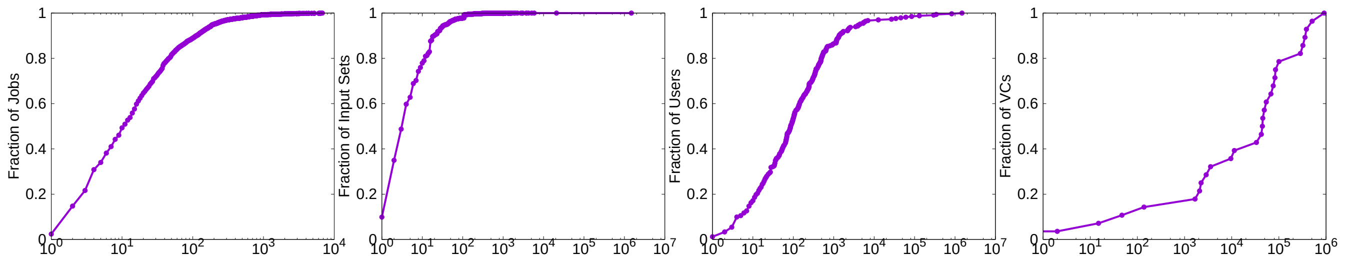
Figures 4(b)- 4(d) show the cumulative overlap distribution for three of the operators, namely shuffle, filter, and user-defined processor. Even though we show the shuffle operator to be more overlapping, only a small fraction of the shuffles have high frequency. This changes for the filter operator, where the cumulative distribution grows more flat, meaning that more number of filters have higher frequency. Finally, for user defined operators in Figure 4(d), the curve is more flatter. This is because user defined operators are likely to be shared as libraries by several users and teams.

2.4 Impact of Overlap

In the previous section, we saw the computation overlaps in different clusters, VCs, and operators. We study the impact of these overlaps along several dimensions. Figures 5(a)–5(d) show the cumulative distributions of frequency, runtime, output size, and relative costs (i.e., view-to-query cost ratio) of the overlapping computations in one of our largest business units (same as from Section 2.2).

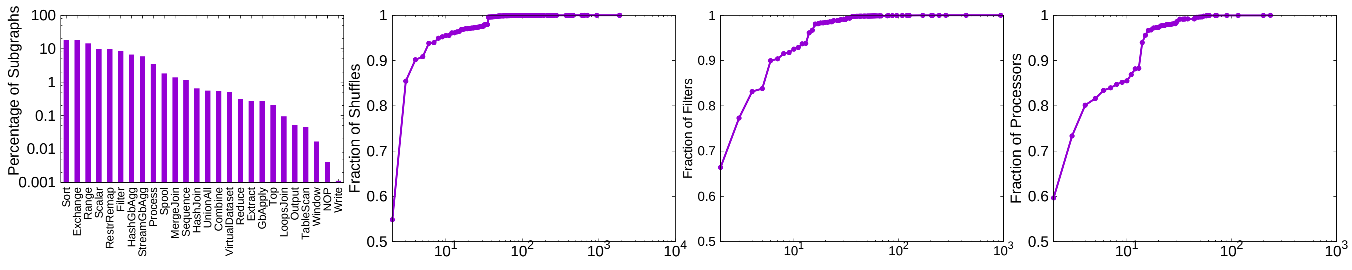
In terms of frequency, there are close to a million computations appearing at least twice, with tens of thousands appearing at least 10 times, few hundreds appearing at least 100 times, and some appearing at least 1000 times — all in one single day! The average

[1]A virtual cluster is a tenant having an allocated compute capacity, called tokens, and controlling access privileges to its data.



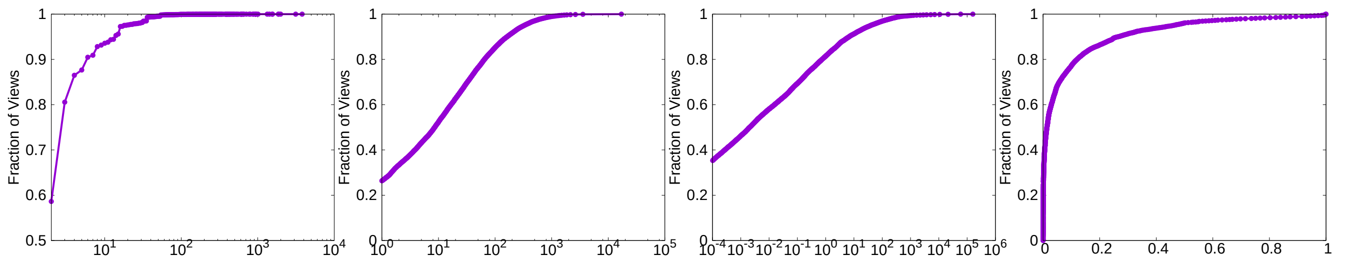
(a) Overlap by jobs (b) Overlap by inputs (c) Overlap by users (d) Overlap by VCs

Figure 3: Cumulative distributions of overlap in one of the largest business units.



(a) Operator breakdown (b) Shuffle (c) Filter (d) Processor

Figure 4: Operator-wise overlaps in 4(a); Per-operator cumulative distributions of overlap in 4(b)– 4(d).



(a) Frequency (b) Runtime (s) (c) Size (GB) (d) View-Query Cost Ratio

Figure 5: Quantifying the impact of overlap in one of the largest business units.

overlap frequency however is 4.2 (median 2, 75^{th} percentile 3, 95^{th} percentile 14, and 99^{th} percentile 36). Thus, computation overlap frequencies are heavily skewed and we need to be careful in picking the views to materialize.

In contrast to the frequency, the runtime and the output size distributions have much less skew. Interestingly, 26% of the overlaps have runtime of $1s$ or less, indicating there are opportunities to prune many of the reuse candidates, while 99% of the overlaps have runtime below $1000s$. In terms of output size, 35% of the overlaps have size below $0.1MB$, which is good (in case they are useful) for storage space, and 99% have size below $1TB$. Lastly, view-to-query cost ratio is an interesting metric to understand the relative importance of a view to a query. We note that 46% of the overlapping computations have view-to-query cost ratio of 0.01 (1%) or less. These overlaps will not result in significant savings in latency, although their cumulative resource consumption savings may still be of interest to the customer. Overall, this is again a highly skewed distribution with only 23% of the overlaps having a view-to-query cost ratio of more than 0.1, and just 4% having a ratio of more than 0.5.

3 REUSE OVER RECURRING WORKLOADS

Our goal is to materialize overlapping computations over recurring jobs in SCOPE, i.e., jobs that appear repeatedly (hourly, daily, weekly, or monthly), have template changes in each instance, and operate over new data each time. Prior works require the workload to be known a-priori in order to analyze the workload and select the views to materialize. However, with recurring jobs changing and running over new data in each instance, the exact workload is not available until the next recurring instance, e.g., the next hour. Running the workload analysis to select the views to materialize within the same recurring instance, before running the actual jobs, is simply not possible.

To handle recurring jobs, we collect a combination of two signatures for each subgraph computation: one which identifies the computation precisely and one which normalizes the precise signature by the recurring changes, e.g., data/time predicates, input names, etc. These signatures are created during compilation and they are similar to plan signatures or fingerprints in prior works [1].

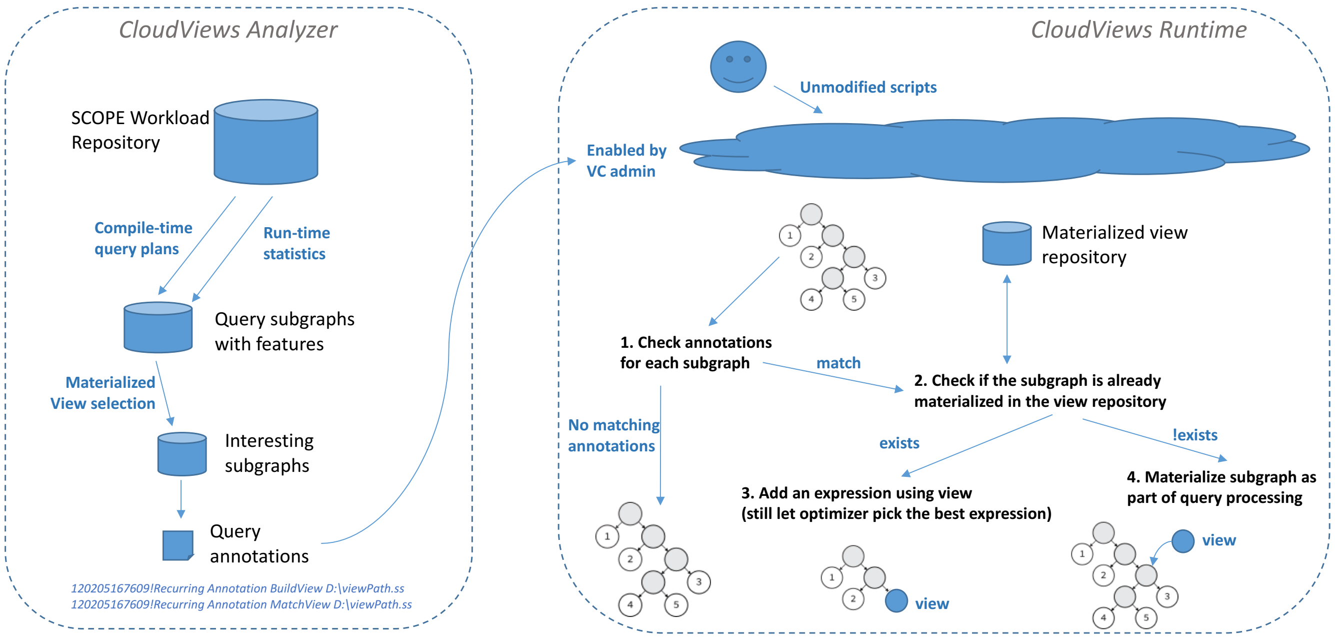

CloudViews Analyzer
SCOPE Workload Repository
Compile-time query plans
Run-time statistics
Query subgraphs with features
Materialized View selection
Interesting subgraphs
Query annotations
120205167609!Recurring Annotation BuildView D:\viewPath.ss
120205167609!Recurring Annotation MatchView D:\viewPath.ss
CloudViews Runtime
Unmodified scripts
Enabled by VC admin
Materialized view repository
1. Check annotations for each subgraph
match
2. Check if the subgraph is already materialized in the view repository
No matching annotations
exists
!exists
3. Add an expression using view (still let optimizer pick the best expression)
4. Materialize subgraph as part of query processing
view
view

Figure 6: CloudViews System Architecture.

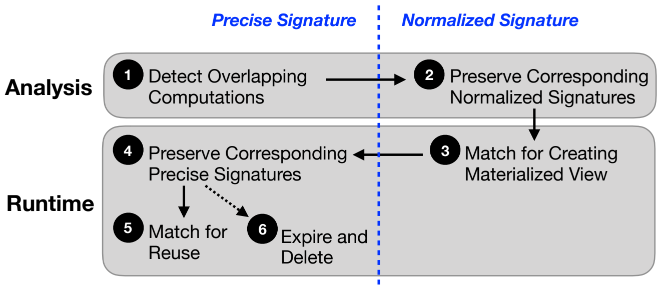

Precise Signature
Normalized Signature
Analysis
Runtime
1 Detect Overlapping Computations
2 Preserve Corresponding Normalized Signatures
3 Match for Creating Materialized View
4 Preserve Corresponding Precise Signatures
5 Match for Reuse
6 Expire and Delete

Figure 7: Use of precise and normalized signatures for computation reuse over recurring workloads.

However, we extended the precise signature to further include the input GUIDs, any user code, as well as any external libraries used for custom code. The normalized signature ensures that we capture a normalized computation that remains the same across different recurring instances. Figure 7 shows the use of these two signatures in our approach. We analyze any recurring instance from the workload history and select frequent computations (views) based on their precise signatures (Step 1), and collect their corresponding normalized signatures into our metadata service (Step 2). This analysis needs to be run periodically, only when there is a change in the workload, thereby removing the need to run workload analysis within each recurring instance. Later during runtime, we materialize subgraphs based on their normalized signatures (Step 3), but we also record the precise signature of each of the materialized view into the physical path of the materialized files (Step 4). The precise signatures are used to match future computations for reuse (Step 5), as well as for expiring a materialized view (Step 6). In summary, the normalized signature identifies subgraphs *across* recurring instances (for materialization) while the precise signature matches subgraph *within* a recurring instance (for reuse). Together, they make computation reuse possible over recurring workloads.

4 CLOUDVIEWS OVERVIEW

In this section, we give a brief overview of the CloudViews system. Our key goals derived from our engagement with product teams are as follows:

(1) Automatic: We need minimal manual intervention, since it is really hard for developers to coordinate and reuse overlapping computations amongst themselves. Thus, overlapping computations should be detected, materialized, reused, and evicted automatically.

(2) Transparent: With hundreds of thousands of jobs, it is simply not possible to make changes in user scripts or their libraries.

(3) Correct: Computation reuse should not introduce incorrect results, i.e., data corruption. This is especially challenging due to the presence of parameters, users code, and external libraries.

(4) Latency-sensitive: SCOPE users cannot afford to slow down their data pipelines and hence computation reuse should offer better or same performance. This requires accurate estimates on the cost/benefit of materialized views, and the optimizer should still be able to discard a view in case it turns out to be too expensive.

(5) Maximize reuse: The obvious goal is to do the computation reuse wherever possible. This is hard because overlapping jobs may arrive concurrently and so views materialized in one job may not end up being reused.

(6) Debuggability: SCOPE has a rich debuggability experience and computation reuse should preserve that. Specifically, customers (and operations team) should be able to replay the job, see which materialized views are created or used, trace the jobs which created any of the views, and even drill down into why a view was selected for materialization or reuse in the first place.

(7) Reporting: Finally, we need to report the impact of computation reuse on job performance, i.e., the storage costs and runtime savings, as well as make all metadata related to the overlapping computations queryable.

Traditional materialized view technologies typically have three components, an offline view selection component, an offline view building component, and an online view matching component. In our approach, we have two *online* components: a periodic workload analyzer to mine overlapping computations, and a runtime engine to materialize and reuse those computations.

Figure 6 shows the high-level architecture of CloudViews. The left side shows the periodic workload analyzer that is used to analyze the SCOPE workload repository. Admins can choose to include or exclude different VCs for analysis. The output of this analysis is a set of annotations telling future jobs the subgraph computations that must be materialized and reused. The right side of Figure 6 shows the runtime component of CloudViews. Here, each incoming job can be processed in one of three ways: (i) exactly same as before in case none of the job subgraphs are materialized or are deemed to be too expensive by the optimizer, (ii) modified job graph that reads from a materialized view (i.e., there is a matching subgraph annotation and it is materialized) and reading from the materialized view is considered more efficient than recomputing it by the optimizer, and (iii) modified job graph that spools and materializes the output of a subgraph (i.e., there is a matching subgraph annotation but it is not materialized). While the analyzer part of CloudViews could be triggered explicitly by the user or scheduled as another recurring job, the runtime part is triggered by providing a command line flag during job submission. The job scripts of the end users remain unchanged.

5 CLOUDVIEWS ANALYZER

In this section, we describe the analyzer component of CloudViews. The key features of this component include: (i) providing feedback loop for runtime statistics, (ii) picking the physical design for the selected views to materialize, (iii) determining the expiry of a materialized view, and (iv) providing a user interface to tune and visualize the workload analysis. We describe each of these below.

5.1 The Feedback Loop

Picking the right set of views to materialize is a hard problem. State-of-the-art approaches rely on *what-if optimization* to estimate the expected improvements if the view were to be materialized [2]. Unfortunately, the optimizer cost estimates are often way off due to the presence of complex DAGs and user code. The problem becomes even more severe in a distributed cloud setting where virtual hardware and scheduling issues make it even harder to model the actual gains in terms of job latencies. As a result, the actual improvements from a materialized view may be much lower while its actual materialization costs may be much higher than the estimated ones. Thus, we need higher confidence on which views to materialize and do not want to materialize a view which later ends up not being used, thereby wasting customer money in a job service. This gets further challenging with dynamic resource allocation within a job graph as well as with opportunistic resource allocation in SCOPE [8].

We handle the above issues by providing a feedback loop that reconciles compile-time estimates with run-time statistics, as depicted

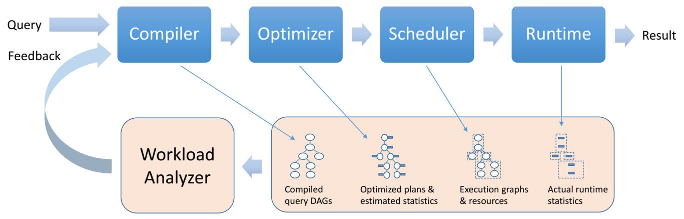

Query
Feedback
Compiler
Optimizer
Scheduler
Runtime
Result
Workload Analyzer
Compiled query DAGs
Optimized plans & estimated statistics
Execution graphs & resources
Actual runtime statistics

Figure 8: Feedback loop for computation reuse.

in Figure 8. Our feedback mechanism goes beyond learning from the same query, as in LEO [45], and considers arbitrary fine-grained commonalities across multiple jobs. We do this by enumerating all possible subgraphs of all jobs seen within a time window in the past, e.g., a day or a week, and finding the common subgraphs across them. Though this is more restricted than considering generalized views[2], the subgraphs considered have actually been used in the past (likely to be also used in the future) and there are runtime statistics available from those previous runs (can cost them more accurately).

In order to use the runtime statistics from the previous runs, we connect the job data flow (one which actually gets executed on a cluster of machines) back to the job query graph (the tree representation of the input user query). We do this by linking the operators executed at every stage in the data flow to operators in the query graph. Then, for every query subgraph, we extract corresponding runtime statistics from the data flow. These include latency (time taken to execute the subgraph), cardinality (number of output rows in the subgraph), data size (in bytes), and resource consumption (CPU, memory, parallelism, IO, etc.). In cases where several operators are pipelined in a data flow, we attribute runtime statistics such as resource consumption to individual operators, e.g., by sub-dividing the total resource consumption of all pipelined operators based on the exclusive runtime of each operator in the pipeline.

Our feedback loop has several key benefits. First, there is an inevitable duplication of analysis in user scripts, due to common data preparation needed in multiple analyses or simply due to the fact that developers often start from someone else's script before adding their own logic. With the feedback loop in our job service, users do not have to worry about de-duplicating their scripts; the system takes care of doing it automatically at runtime. Second, the runtime statistics provide more predictable measures of view materialization costs and benefits, thereby giving the customer a better idea of how much he will pay and how much he will save with this feature. Third, the feedback loop makes it more likely that the selected (and materialized) subgraphs will actually end up being used in future jobs, in contrast to picking materialized views based on cost estimates and later finding them not useful if the estimates turn out to be incorrect. Fourth, our feedback loop considers job subgraphs without considering merging two or more subgraphs, as in more general view selection. This ensures that materializing a view never requires additional computation (and hence additional money) than that would anyways be done by a job using that view. And finally, the runtime statistics observed from the subgraphs of one job get shared across all future queries having any of those subgraphs. In fact, for any new job that comes in, the system may already know

[2]Queries $Q1$ and $Q2$ reading attributes (A, B) and (A, C), respectively, would generate a view (A, B, C) as candidate, even though it is neither a subgraph of $Q1$ nor of $Q2$.

the costs of its several subgraphs and may decide to not recompute them.

5.2 Selecting Subgraphs to Materialize

As mentioned before, we simplify the view selection problem by restricting ourselves to common subgraphs. Although this is more limited than generalized view selection, we are able to capture precise utility and cost estimates, since the subgraphs have been executed in the past. In addition, during query rewriting we simply scan the materialized view, without incurring any other post-processing, and hence the gains are more predictable.

We consider two kinds of approaches to select the subgraphs to materialize:
(i) selecting the top-k subgraphs using one or more heuristics, e.g., total subgraph utility, or total utility normalized by storage cost, or limiting to at most one subgraph per-job, etc. The system allows users to plug in custom heuristics to narrow down to the subgraphs of their interest.
(ii) packing the most interesting subgraphs (or *subexpressions*) given a set of constraints, e.g., storage constraints, view interaction constraints, etc. We investigated the subexpression packing problem in more details in a companion work [24].

5.3 Physical Design

One of the early lessons we learned in this project was the importance of view physical design. The physical design of materialized views is typically not paid much attention, i.e, views and their physical design are typically not selected at the same time. However, we observed that materialized views with poor physical design end up not being used because the computation savings get over-shadowed by any additional repartitioning or sorting that the system needs to do. This happens because with massively large datasets and massively parallel processing in SCOPE, repartitioning and sorting are often the slowest steps in the job execution.

CLOUDVIEWS, therefore, pays close attention to view physical design. To do so, we extract the output physical properties (partitioning type, partitioning columns, number of partitions, sort columns, sort direction) of each of the subgraph while enumerating them. The output physical properties are good hints for view physical design as they are expected by subsequent operators in the job graph. In case of no explicit physical properties at the subgraph root, we infer them from the children, i.e., we traverse down until we hit one or more physical properties. Depending on how an overlapping subgraph is used in different jobs, there may be multiple sets of physical properties for the same subgraph. The default strategy is to pick the most popular set. However, in case of no clear choice, we treat multiple physical designs (of the same view) as different views and feed them to the view selection routine.

5.4 Expiry and Purging

Although storage is cheap, the storage space used by materialized views still needs to be reclaimed periodically. A simple heuristic is to remove all views from the previous recurring instance. However, discussions with our customers revealed that output of hourly jobs could also be used in weekly jobs or monthly jobs. Therefore, removing views after each hour/day could be wasteful. A better option is to track the lineage of the inputs of the view, i.e., for each of the view input, check the longest duration that it gets used by any of the recurring jobs. The maximum of all such durations gives a good estimate of the view expiry. Apart from using standard SCOPE scripts, this type of lineage tracking could also be facilitated using provenance tools such as Grok [43] and Guider [35], or Goods [21]. The view expiry thus obtained is encoded into the physical files, and our Storage Manager takes care of purging the file once it expires.

Cluster admins could also reclaim a given storage space by running the same view selection routines as described in Section 5.2 but replacing the *max* objective function with a *min*, i.e., picking the views with minimum utility. In the worst case, the materialized view files can be simply erased from the cluster. Both of the above operators, however, require cleaning the views from the metadata service first before deleting any of the physical files (to ensure that jobs consuming any of those inputs do not fail).

5.5 User Interfaces

CLOUDVIEWS provides a few ways to interact with the workload analyzer. First, there is a command line interface to run the analyzer over user specific cluster, VCs and time ranges. Users can also provide their custom constraints, e.g., storage costs, latency, CPU hours, or frequency, to filter down the overlapping computations. Then, there is a Power BI [39] dashboard to look at various summaries from computation overlap analysis, as well as drill down into the top-100 most overlapping computations in more detail. Together, the goal is to help users understand the computation overlap in their workloads and to tailor computation reuse for their needs.

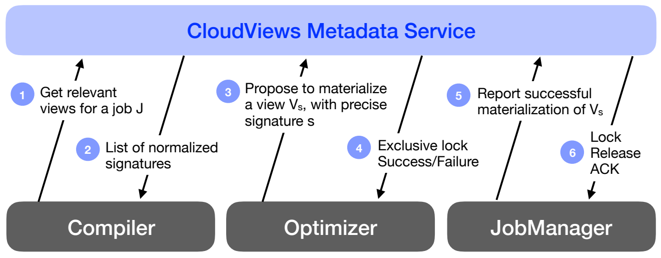
6 CLOUDVIEWS RUNTIME

In this section, we describe the various components that make computation reuse possible during query processing. We collectively refer to them as the CLOUDVIEWS runtime, which consists of: (i) a metadata service to query the relevant overlaps in each incoming job, (ii) an online view materialization capability to materialize views as part of query processing, (iii) a synchronization mechanism to prevent concurrent jobs materializing the same view, (iv) an early materialization technique to publish a materialized view even before the job producing it completes, (v) automatic query rewriting to use materialized views wherever possible, and (vi) hints to the job scheduler in order to maximize the computation reuse.

6.1 Metadata Service

The goal of the metadata service is to provide the lookup for overlapping computations and to coordinate the materialization and reuse of those computations. Recall that we have an *online* setting, i.e., data batches and jobs arrive continuously, and hence view materialization and reuse is a dynamic activity. Therefore, instead of simply looking up the views in the compiler, multiple SCOPE components interact with the metadata service at runtime, as illustrated in Figure 9.

First, the compiler asks the metadata service for overlapping computations (views) for a given job J (Step 1). The naïve approach would be for the compiler to lookup each subgraph individually to check whether or not this is an overlapping computation. However, the number of lookup requests can explode since SCOPE job graphs can be quite large, thereby leading to higher compilation overhead



CloudViews Metadata Service
1 Get relevant views for a job J
2 List of normalized signatures
3 Propose to materialize a view V_s, with precise signature s
4 Exclusive lock Success/Failure
5 Report successful materialization of V_s
6 Lock Release ACK
Compiler
Optimizer
JobManager

Figure 9: CLOUDVIEWS metadata service interactions with different SCOPE components.

as well as higher throughput requirements from the metadata service. Instead, we make one request per-job and fetch all overlaps that could be relevant for that job. This is done by creating an inverted index as follows. For each overlapping computation instance, we extract tags from its corresponding job metadata. We normalize the tags for recurring jobs and create an inverted index on the tags to point to the corresponding normalized signatures. The metadata service returns the list of normalized signatures relevant to J to the compiler (Step 2). The signatures returned by the metadata service may contain false positives, and the optimizer still needs to match them with the actual signatures in the query tree.

Second, when the optimizer tries to materialize an overlapping computation, it proposes the materialization to the metadata service (Step 3). The metadata service tries to create an exclusive lock to materialize this view. Due to large number of concurrently running jobs, the same view could be already materialized by another job, i.e., the lock already exists. In this case, the service returns a failure message, otherwise, it returns success (Step 4). Note that we mine the average runtime of the view subgraph from the past occurrences, and use that to set the expiry of the exclusive lock. Once the exclusive lock expires, and if the view is still not materialized, another job could try to create the same materialized view. This gives us a fault-tolerant behavior for view materialization.

Finally, the job manager reports the successful materialization of a view to the metadata service (Step 5) and the service acknowledges the lock release (Step 6). The metadata service now makes the materialized view available for other jobs to reuse, i.e., it may appear the next time the compiler asks for relevant views for a job (Step1).

We deployed our metadata service using AzureSQL as the backend store. The metadata service periodically polls for the output of CLOUDVIEWS analyzer and loads the set of selected overlapping computations whenever new analysis is available. We purge expired computations at regular intervals.

6.2 Online Materialization

Traditional materialized views require an offline process where the database administrator is responsible to first create all relevant materialized views, i.e., the preprocessing step, before the database becomes available for running the query workload. This is not possible with recurring jobs which run in tight data pipelines with strict completion deadlines, where there is little room to do the preprocessing for creating the materialized views. Preprocessing blocks the recurring jobs, thereby causing them to miss their completion deadlines. Recurring jobs also have data dependency between them, i.e., the result of one recurring job is used in subsequent recurring jobs. Thus, missing completion deadline for one recurring jobs affects the entire data pipeline.

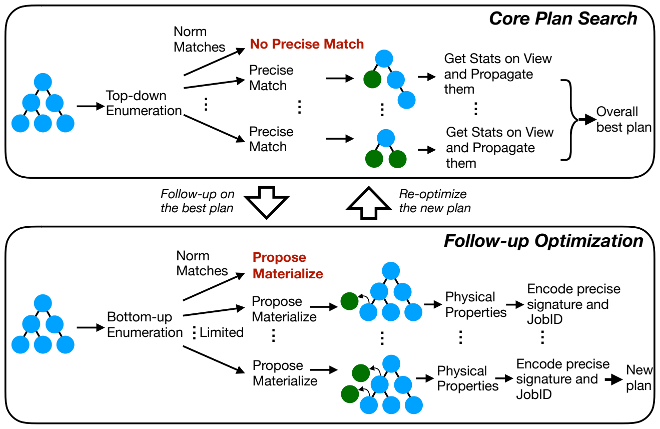


Figure 10: Illustrating online materialization and query rewriting mechanisms in the SCOPE query optimizer.

We introduce a mechanism for creating and reusing materialized views as part of the query processing, as depicted in Figure 10. After fetching the relevant normalized signatures from the metadata service, the compiler supplies them as annotations to the query optimizer. The compiler also preserves the annotations as a job resource for future reproducibility. The optimizer first checks for all reuse opportunities in the plan search phase before trying to materialize one or more views in a *follow-up optimization* phase, shown in lower half of Figure 10. This ensures that views already materialized (and available) are not attempted for materialization. During follow-up optimization, the optimizer checks whether the normalized signature of any of the subgraphs matches with the ones in the annotation. We match the normalized signatures in a bottom-up fashion (materializing smaller views first as they typically have more overlaps) and limit the number of views that could be materialized in a job (could be changed by the user via a job submission parameter). In case of a match, the optimizer proposes to materialize the view (Step 3 in Figure 9). On receiving success from the metadata service, the optimizer adjusts the query plan to output a copy of the sub-computation to a materialized view, while keeping the remainder of the query plan unmodified as before. The new output operator also enforces the physical design mined by the analyzer for this view. The optimizer takes care of adding any extra partitioning/sorting operators to meet those physical property requirements. The optimizer stores the precise signature of each materialized view as well as the ID of the job producing the materialized view (for tracking the view provenance) into the physical path of the materialized file.

The salient features of our approach are as follows. First, we introduce a mechanism to create materialized views with minimal overhead as part of the query processing, without requiring any up-front preprocessing that would block the recurring queries. Second, our approach causes the first query that hits a view to materialize it and subsequent queries to reuse it wherever possible. As a result, we materialize views, and hence consume storage, just when they are to be needed, instead of creating them a priori long before they would ever be used. Third, we do not need to coordinate between the query which materializes the view (as part of its execution), and the queries which reuse that materialized view; in case of multiple queries arriving at the same time, the one which finishes first materializes the view. Fourth, in case there is a change in query workload starting from a given recurring instance, then the view materialization based on the the previous workload analysis stops automatically as the signatures do not match anymore. This avoids paying for and consuming resources for redundant views that are not going to be used after all. This also indicates that it is time to rerun the workload analysis. Finally, our approach does not affect any of the user infrastructure in their analytics stack. This means that the user scripts, data pipelines, query submission, job scheduling, all remain intact as before.

For traditional users with enough room for upfront view materialization, e.g., weekly analytics, CLOUDVIEWS still provides an offline view materialization mode. In this mode, the optimizer extracts the matching overlapping computation subgraph while excluding any remaining operation in the job. The resulting plan materializes only the views and could be executed offline, i.e., before running the actual workload. The offline mode can be configured at the VC level in the metadata service, and later the annotations passed to the optimizer are marked either online or offline depending on the metadata service configuration.

6.3 Query Rewriting

To rewrite queries using materialized views, we added an additional task in the Volcano style plan search [16]. This additional task, as shown in the upper half of Figure 10, matches the normalized signatures retrieved from the metadata service with the normalized signatures of each of the query subgraphs in a top-down fashion, i.e, we match the largest materialized views first. In case of a match, the optimizer matches the precise signature as well. Only if the precise signature matches then the materialized view could be reused. In such a scenario, the optimizer adds an alternate subexpression plan which reads from the materialized view. We do not limit the number of materialized views that could be used to answer a query. Once all applicable materialized views have been added as alternate subexpressions, the optimizer picks the best plan based on the cost estimates, i.e., one or more materialized views may end up not being used if their read costs are too high. The plan that reads from the materialized view also loads the actual statistics (for that sub-computation) and propagates those statistics up the query tree. This gives more confidence in deciding whether the plan using the materialized view is actually a good one or not. Overall, we provide fully automatic query rewriting using views, with zero changes to user scripts.

6.4 Synchronization

We have two goals in terms of synchronization: (i) *build-build* synchronization, i.e., not having multiple jobs materialize the same view, and (ii) *build-use* synchronization, i.e., reuse a computation as soon as it is materialized. We handle the build-build synchronization by trying to reuse computations before trying to materialize them, as described in Section 6.2. For concurrent jobs, we also create exclusive locks via the metadata service, as described in Section 6.1. Given that the service is backed by AzureSQL, it provides consistent locking, and only a single job can actually materialize a view at a time. To handle the build-use synchronization, we modified the SCOPE job manager to publish the materialized view as soon as it is available. This means that the materialized view output is available even before the job that produces it finishes. We refer to this as *early materialization*. Early materialization is a semantic change as it breaks the atomicity of SCOPE jobs, however, it very useful because the views could be a much smaller subgraph of the overall job graph. Furthermore, the materialized view is not a user output, but is rather treated as a system output, and therefore we do not affect the user contract. Finally, early materialization also helps in case of jobs failures, since the job can restart from the materialized view now, i.e., early materialization acts as a checkpoint.

6.5 Job Coordination

The perfect scenario for computation reuse is when one of the jobs with overlapping computation is scheduled before others, so that the view could be computed exactly once and reused by all others. However, in reality, multiple jobs containing the same overlapping computation could be scheduled concurrently. In this case, they will recompute the same subgraph and even attempt to materialize it (though only one will prevail). We mitigate this problem by reordering recurring jobs in the client job submission systems[3]. To do this, in addition to selecting the interesting computations to materialize, the CLOUDVIEWS analyzer also provides the submission order of the recurring jobs, that contain those computations, which will give the maximum benefit. We do this by grouping jobs having the same number of overlaps (job with multiple overlaps can appear in multiple groups), and picking the shortest job in terms of runtime, or least overlapping job in case of a tie, from each group. The deduplicated list of above jobs will create the materialized views that could be used by all others, and so we propose to run them first (ordered by their runtime and breaking ties using the number of overlaps). Such an ordering can be enforced using the SCOPE client-side job submission tools. Future work will look into how view-awareness could be handled centrally by the job scheduler itself.

7 EVALUATION

In this section, we present an experimental evaluation of CLOUDVIEWS. We break down our evaluation into three parts, answering each of the following questions: (i) what is the impact on performance over production jobs at Microsoft? (ii) what is the impact on traditional TPC-DS benchmark? and (iii) what are the overheads involved in CLOUDVIEWS? Below we address each of these.

7.1 Impact on Production Jobs

We first present performance evaluation results from our production clusters. Given the capacity constraints and costs involved in running experiments on these clusters, we carefully picked a small set of job workload for our evaluation, as described below.

Workload. We ran CLOUDVIEWS analyzer over one day worth of jobs from one of the largest business units at Microsoft. We narrowed down the overlapping computations to those appearing at least thrice,

[3]There are multiple client side tools developed and maintained by different business units at Microsoft to create workflows on top of our job service.

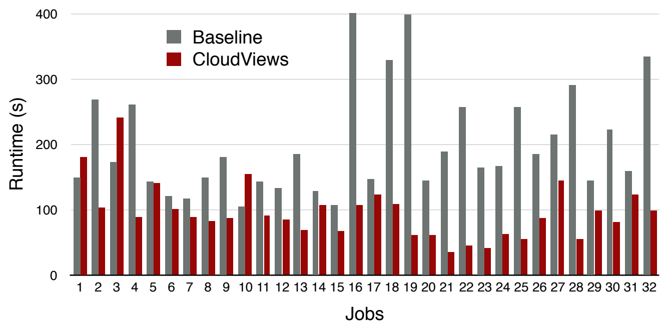


Figure 11: Production jobs: end-to-end latency.

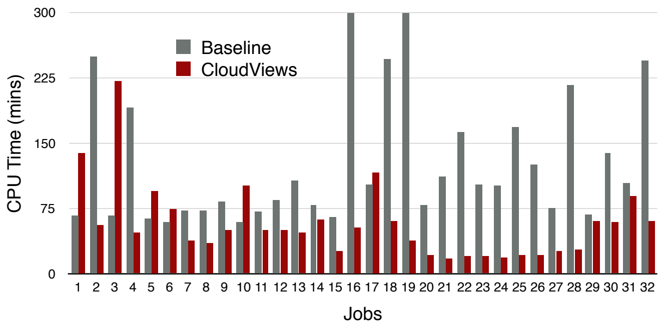


Figure 12: Production jobs: resource consumption.

whose cost is at least 20% of the overall job cost, and considering at most one overlapping computation per job. From the resulting set of overlapping computations, we picked the top-3 computations (views) based on their total utility, i.e., frequency times the average runtime of the overlapping computations. For each of these computations, we looked up the jobs relevant to those computations to construct our workload, consisting of a total of 32 jobs — 16, 12, and 4 jobs respectively for the three overlapping computations.

Setup. We ran the above jobs in a pre-production environment and over production data, but with output redirection as described in [1]. For each view, we ran the relevant jobs in a sequence, in the same order as they arrived in the past workload. The first job in the sequence materializes the overlapping computation, while the remaining jobs reuse it. We executed each job twice, once with and once without CLOUDVIEWS enabled. In order to make the two runs comparable, we used the same number of machine instances and disabled opportunistic scheduling [8]. We also validated the outputs of the two runs to ensure that there is no data corruption.

Results. Figure 11 shows the end-to-end job latency. While there is latency improvement in all but three jobs that create the materialized view, the actual improvements vary (maximum of 91% speedup and 48% slowdown, and average speedup of 43%). This is due to a number of factors, including: (i) materialized view read costs could be significant and variable based on the parallelism used at runtime, (ii) accurate estimates are propagated only in the subexpression that uses a view and the estimates are still way off in other cases (often over-estimated to avoid failures in big data systems), (iii) there could be additional partitioning or sorting applied by the optimizer to satisfy the required physical properties of the parent subexpressions (very hard to get the best view physical design for every query), and (iv) latency improvements depend on the degree to which the overlap is on the critical path. Still, the overall workload sees a total latency improvement of 60% and even though we evaluated some of the most overlapping jobs for this particular customer, it demonstrates the effectiveness of CLOUDVIEWS in speeding up analytical jobs in our job service.

Figure 12 shows the resource consumption in terms of the total CPU-hours for each of the jobs in our workload. Similar to latency, CPU-hour improvements are also variable (maximum of 95% speedup, minimum of 230% slowdown, and average speedup of 36%). In particular, increased parallelism to read and write the materialized view, which could be often large, affects the overall resource consumption. Overall, however, there is a 54% drop in CPU time for the entire workload. Again, this is quite significant for reducing operational costs in our clusters.

7.2 TPC-DS Experiments

We now present results from TPC-DS benchmark [48]. Even though TPC-DS does not really model the recurring workloads with producer/consumer behavior that we have in SCOPE, it is still helpful to evaluate our system on a more widely used benchmark.

Workload. We generated $1TB$ of TPC-DS dataset and considered all of the 99 queries in the benchmark. We ran all TPC-DS queries once without using CLOUDVIEWS. Then, we ran the CLOUDVIEWS analyzer to detect and select top-10 overlapping computations, similar to what was described in Section 7.1. Note that this is a very conservative selection of overlapping computations, and much higher gains could be realized by using more sophisticated view selection methods proposed in the literature [33].

Setup. We ran the above workload in a test environment using the CLOUDVIEWS runtime. We use our job coordination hints to run one of the jobs containing an overlap first (to create the materialized view) and the other jobs containing the same overlap after that (to use the materialized view). We ran each query with 100 machine instances and disabled opportunistic scheduling [8] in order to make the performance comparable.

Results. Figure 13 shows the runtime improvements with CLOUDVIEWS in each of the TPC-DS queries. We can see that even with our conservative selection of overlapping computations, most of the queries (79 out of 99) see an improvement in performance. Both the peak improvement as well as the peak slowdown is close to 62%. Overall, the average runtime improves by 12.5%, while the total workload runtime improves by 17%. These would translate to significant cost savings in a job service where users pay for the resources used, which is proportional to the runtime, per query.

7.3 Overheads

Finally, we discuss some of the overheads associated with CLOUDVIEWS. First, we have an overhead of running the CLOUDVIEWS analyzer. A typical run to analyze all jobs (several tens of thousands) in a cluster takes a couple of hours. However, since we analyze the recurring templates, we only need to analyze once in a while when there are changes in workload. We detect changes in workload by monitoring changes in the number of materialized views created over time.

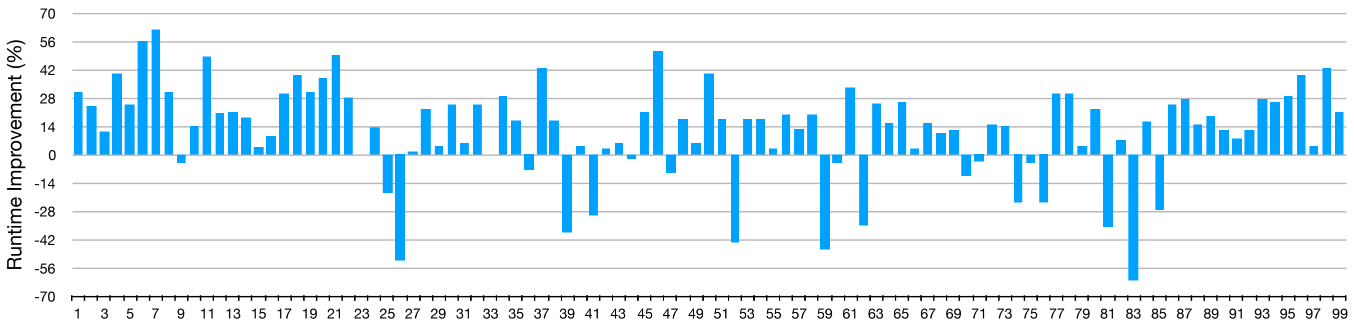


Figure 13: TPC-DS queries: percentage runtime improvements.

Then, there is compile time overhead to lookup the metadata service and to do additional work during query optimization. We measured the latency added due to metadata service lookup and it turned out to be 19ms on average with a single thread and 14.3ms on average when using 5 threads in the metadata service. This is reasonable given that the overall compilation time for TPC-DS queries was in the range of 1-2 minutes. Likewise, we measured the query optimization overhead with CLOUDVIEWS over TPC-DS queries. Interestingly, while the optimization time increased by 28% on average when creating a materialized view, the optimization time decreased by 17% on average when using the view. This is because the query tree becomes smaller when using the view and so any follow-up optimizations become faster.

8 LESSONS LEARNED

In this section, we outline experiences from deploying CLOUDVIEWS to our production clusters. The CLOUDVIEWS analyzer is available as an offline tool for VC admins, while the CLOUDVIEWS runtime ships with the most recent SCOPE release. The technology is currently in preview and available to our customers in an opt-in mode, i.e., each VC admin can enable CLOUDVIEWS either for the entire VC or for certain jobs in that VC. Eventually, the goal is to make CLOUDVIEWS opt-out, i.e., overlapping computations are reused wherever possible, but customers can explicitly turn it off in special cases, e.g., SLA sensitive jobs. Below we summarize the key lessons learned from the CLOUDVIEWS project.

Discovering hidden redundancies. Data analytics jobs have hidden redundancies across users (or sometimes even for the same user), and it is really hard to detect and mitigate these redundancies manually at scale. Most of the customers we talked to already expected to have computation overlaps in their workloads, and it was interesting for them to see the exact jobs and the overlapping computations present in them. While some of the customers were willing to take the pain of manually modifying their scripts to prevent overlaps, most preferred to use our automatic reuse approach instead.

Improving data sharing across VCs. SCOPE workloads are typically organized as data pipelines, with dependencies across VCs that are fulfilled via explicit data materialization. With CLOUDVIEWS, we could help customers detect the most efficient of these materializations, better than those from the manual best effort and that could speedup downstream processing. This is an interesting side-effect of CLOUDVIEWS and would be a subject for future work.

Extracting static computations. In many cases, we saw that there were overlapping computations even across multiple recurring instances of the same job, i.e., even with different inputs. This was because portions of the job were unchanged across multiple instances, i.e., the inputs to those portions were still the same while other portions of the job had different inputs. CLOUDVIEWS was therefore effective in detecting such static computations across multiple job instances.

Reusing existing outputs. In several other cases, a subgraph rooted at an output operator was common across jobs. This means that multiple jobs were producing the same output without ever realizing it. CLOUDVIEWS was helpful to consolidate such redundant outputs by materializing the common computation once and reusing it wherever possible; we separately asked the owners of those jobs to remove the redundant output statements in their jobs.

Discarding redundant jobs. In multiple cases, entire jobs were detected as overlapping. This was because of two reasons: (i) given that jobs are recurring, some of the jobs end up scheduled more frequently than the new data arrival, and (ii) there were rare cases of plain redundancy where multiple users unknowingly submit the same job. CLOUDVIEWS helped in detecting such redundancies.

Utility of view physical design. Our workloads had explicit data dependencies across jobs, but the users had little idea on how to set the physical designs of the output from one job that needs to be consumed by another job. With CLOUDVIEWS, we not only capture many of these dependencies across jobs, but also pick the best physical designs for those dependency outputs.

Better reliability. By materializing the shared computations across jobs, CLOUDVIEWS not only provides better performance, but it also reduces the failure rates as fewer tasks are scheduled in subsequent jobs hitting the same overlapping computation. Thus, view materialization acts as a checkpoint providing better reliability. This is further useful when the first job that hits an overlapping computation fails, since the overlapping portion may be already materialized due to early materialization in CLOUDVIEWS runtime.

Better cost estimates. As mentioned before, view materialization improves the cost estimates since we can collect exact statistics from the materialized output. Given that we materialize computations that are frequent as well as expensive, better estimates over those computations are even more significant.

User expectations. It was important to manage the user expectations in CLOUDVIEWS project. This includes VC admin expectations to see the cost of overlaps in their workload and expected gains with CLOUDVIEWS, end-user expectations to know whats going on in their job and reacting accordingly, and the operational support team expectations to be able to reproduce and debug the jobs submitted with CLOUDVIEWS enabled.

Updates & privacy regulations. Finally, any updates in the input data results in a different precise signature, thus automatically invaliding any older materialized view for reuse. This is crucial for privacy reasons when the customers explicitly request to stop using their personal data, as provisioned in the new EU GDPR [13].

9 RELATED WORK

Traditional materialized views. Selecting views to materialize has been a long standing topic of research in databases. Given a set of queries, view selection deals with the problem of selecting a set of views to materialize to minimize some cost function (such as query evaluation and/or view maintenance cost) under some constraints (e.g., space budget) [33]. Several approaches have been proposed, especially in the context of data warehouses [19, 46] and data cubes [22]. These include modeling the problem as a state optimization problem and using search algorithm to find the most appropriate view set [46], using AND/OR to model the alternatives in a single DAG [19], or using a lattice to model data cube operations [22]. MQO [42] is similar to view selection, with the difference that views are typically only transiently materialized for the execution of a given query set. [41] describes how to incorporate MQO with a Volcano-style optimizer. It uses an AND/OR DAG and proposes heuristic algorithms for choosing intermediate results to materialize (with no space budget). Recycling intermediate results has also been proposed in the context of MonetDB [23] and pipelined query evaluation [36].

Views are more generic than subexpressions, as they can consider computation that does not appear in the logical query plan. This increases the space of possible solutions, and complicates query containment and answering queries using views [20]. Subexpression selection has also been considered in SQL Server [53]. Other related works have looked at common subexpressions within the same job script [44].

All of the above works have focussed on traditional databases with few tens to hundreds of queries. In contrast, the SCOPE job service processes tens of thousands of jobs per cluster per day. Thus, scalability is a major concern in our setting. In this paper, we described a system that can create and reuse materialized views at our scale. In a companion work, we looked at scalable view selection for our workload size [24].

Computation reuse in big data platforms. Reusing computation has received particular attention in big data platforms, since (i) there is a lot of recurring computation, (ii) optimization time is relatively short compared to the execution time of the jobs, and (iii) performance and resource benefits can be significant. ReStore [12], for instance, considers the caching of map-reduce job outputs, given a space budget. Others have looked at history-aware query optimization with materialized intermediate views [38] and at allocating the cache fairly amongst multiple cloud tenants [28]. Still others have looked at multi-query optimization in the context of map-reduce [37, 50]. PigReuse [10] addresses MQO for Pig scripts. It creates an AND/OR graph using the nested algebra representation of the Pig jobs, and then uses an ILP solver to select the least costly plan that can answer all queries. Most of these works consider sharing opportunities only for map and reduce operators, and hence their applicability is limited. Nectar [18] considers caching intermediate result in a more generalized DAG of operators. It uses heuristics, based on lookup frequency and the runtime/size of the intermediate results, to decide on the cache insertion. Still, an intermediate result is typically the output of an operator pipeline (i.e., consisting of multiple operators), without considering the outputs of all possible subexpressions. Finally, Kodiak [30] applies the traditional database approach of selecting and materializing views, while ensuring that queries meet their SLA and the total view storage is within a budget.

Our approach is different from the above works, since we consider computation reuse over recurring jobs in a job service that is always online, i.e., there is no offline phase for view creation. Furthermore, our end-to-end system includes establishing a feedback loop to ensure that computation reuse is actually effective.

Recurring and progressive query optimization. Both recurring and progressive optimization focus on the problem of inaccurate or missing statistics in query optimization, and not on reusing common subexpressions across jobs. In particular, recurring optimization (such as DB2's LEO [45] and more recently Scope's RoPE [1]) collects actual statistics of query subexpressions at runtime, and uses them in future executions of the same subexpression to improve statistics, and hence the quality of the optimized plan. Progressive optimization has been studied both in the traditional query optimization setting [7, 26, 34], and for big data clusters [9, 27]. These systems observe statistics at runtime and can change the query plan mid-flight in case the observed statistics are significantly different from the estimated ones. We borrow the concept of signatures from [9] to efficiently identify common subgraphs across jobs.

Shared workload optimization. A lot of works have looked at building a common query plan for a set of queries to share operators, such as scans [40, 54] or joins [32]. A global optimization approach to find the overall best shared plan is presented in [14]. Work sharing has also been explored in big data systems. Examples include scan sharing in MapReduce [37], Hive [47] and Pig [51], Unlike such approaches, we opted to keep each job separate: operator sharing in pay-as-you-go job services makes billing and accounting tedious, while it introduces artificial dependencies between jobs, which become even worse in the case of failures.

10 CONCLUSION

In this paper, we presented a case for computation reuse in an analytics job service. We motivated the problem via a detailed analysis from production SCOPE workloads at Microsoft, and described the CLOUDVIEWS system for automatically reusing overlapping computations in SCOPE. The CLOUDVIEWS system addresses several novel challenges including recurring workloads, establishing a feedback loop, and an online setting. Overall, computation overlap is a problem across almost all business units at Microsoft and CLOUDVIEWS can automatically reuse computations wherever possible, resulting in significant potential cost savings.

REFERENCES

[1] Sameer Agarwal, Srikanth Kandula, Nicolas Bruno, Ming-Chuan Wu, Ion Stoica, and Jingren Zhou. 2012. Re-optimizing data-parallel computing. In *NSDI*.

[2] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. 2000. Automated Selection of Materialized Views and Indexes in SQL Databases. In *VLDB*. 496–505.

[3] Amazon Athena 2018. https://aws.amazon.com/athena/. (2018).

[4] Amazon RDS 2018. https://aws.amazon.com/rds/. (2018).

[5] Azure Data Lake 2018. https://azure.microsoft.com/en-us/solutions/data-lake/. (2018).

[6] Azure SQL 2018. https://azure.microsoft.com/en-us/services/sql-database/. (2018).

[7] Shivnath Babu, Pedro Bizarro, and David J. DeWitt. 2005. Proactive Re-optimization. In *SIGMOD Conference*. 107–118.

[8] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. 2014. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In *OSDI*.

[9] Nico Bruno, Sapna Jain, and Jingren Zhou. 2013. Continuous Cloud-Scale Query Optimization and Processing. In *VLDB*.

[10] Jesús Camacho-Rodríguez, Dario Colazzo, Melanie Herschel, Ioana Manolescu, and Soudip Roy Chowdhury. 2016. Reuse-based Optimization for Pig Latin. In *CIKM*.

[11] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. 2008. SCOPE: easy and efficient parallel processing of massive data sets. *PVLDB* 1, 2 (2008), 1265–1276.

[12] Iman Elghandour and Ashraf Aboulnaga. 2012. ReStore: Reusing Results of MapReduce Jobs. *PVLDB* 5, 6 (2012).

[13] EU GDPR 2018. https://www.eugdpr.org/. (2018).

[14] Georgios Giannikis, Darko Makreshanski, Gustavo Alonso, and Donald Kossmann. 2014. Shared Workload Optimization. *PVLDB* 7, 6 (2014).

[15] Google BigQuery 2018. https://cloud.google.com/bigquery. (2018).

[16] Goetz Graefe. 1995. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.* 18, 3 (1995), 19–29.

[17] Zhongxian Gu, Mohamed A. Soliman, and Florian M. Waas. 2012. Testing the Accuracy of Query Optimizers. In *DBTest*. 11:1–11:6.

[18] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. 2010. Nectar: Automatic Management of Data and Computation in Datacenters. In *OSDI*.

[19] Himanshu Gupta and Inderpal Singh Mumick. 2005. Selection of Views to Materialize in a Data Warehouse. *IEEE Trans. Knowl. Data Eng.* 17, 1 (2005), 24–43.

[20] Alon Y. Halevy. 2001. Answering queries using views: A survey. *VLDB J.* 10, 4 (2001).

[21] Alon Y. Halevy, Flip Korn, Natalya Fridman Noy, Christopher Olston, Neoklis Polyzotis, Sudip Roy, and Steven Euijong Whang. 2016. Goods: Organizing Google's Datasets. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. 795–806.

[22] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. 1996. Implementing Data Cubes Efficiently. In *ACM SIGMOD*.

[23] Milena Ivanova, Martin L. Kersten, Niels J. Nes, and Romulo Goncalves. 2009. An architecture for recycling intermediates in a column-store. In *SIGMOD*.

[24] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. 2017. Thou Shall Not Recompute: Selecting Subexpressions to Materialize at Datacenter Scale. *Under Submission* (2017).

[25] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Íñigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. 2016. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *OSDI*. 117–134.

[26] Navin Kabra and David J. DeWitt. 1998. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. In *SIGMOD Conference*. 106–117.

[27] Konstantinos Karanasos, Andrey Balmin, Marcel Kutsch, Fatma Ozcan, Vuk Ercegovac, Chunyang Xia, and Jesse Jackson. 2014. Dynamically optimizing queries over large scale data platforms. In *SIGMOD*.

[28] Mayuresh Kunjir, Brandon Fain, Kamesh Munagala, and Shivnath Babu. 2017. ROBUS: Fair Cache Allocation for Data-parallel Workloads. In *SIGMOD*. 219–234.

[29] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *PVLDB* 9, 3 (2015), 204–215.

[30] Shaosu Liu, Bin Song, Sriharsha Gangam, Lawrence Lo, and Khaled Elmeleegy. 2016. Kodiak: Leveraging Materialized Views For Very Low-Latency Analytics Over High-Dimensional Web-Scale Data. *PVLDB* 9, 13 (2016).

[31] Guy Lohman. 2014. http://wp.sigmod.org/?p=1075. (2014).

[32] Darko Makreshanski, Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. 2016. MQJoin: Efficient Shared Execution of Main-Memory Joins. *PVLDB* 9, 6 (2016).

[33] Imene Mami and Zohra Bellahsene. 2012. A survey of view selection methods. *SIGMOD Record* 41, 1 (2012), 20–29.

[34] Volker Markl, Vijayshankar Raman, David E. Simmen, Guy M. Lohman, and Hamid Pirahesh. 2004. Robust Query Processing through Progressive Optimization. In *SIGMOD*. 659–670.

[35] Ruslan Mavlyutov, Carlo Curino, Boris Asipov, and Philippe Cudré-Mauroux. 2017. Dependency-Driven Analytics: A Compass for Uncharted Data Oceans. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*.

[36] Fabian Nagel, Peter A. Boncz, and Stratis Viglas. 2013. Recycling in pipelined query evaluation. In *ICDE*.

[37] Tomasz Nykiel, Michalis Potamias, Chaitanya Mishra, George Kollios, and Nick Koudas. 2010. MRShare: Sharing Across Multiple Queries in MapReduce. *PVLDB* 3, 1 (2010).

[38] Luis Leopoldo Perez and Christopher M. Jermaine. 2014. History-aware query optimization with materialized intermediate views. In *ICDE*.

[39] Power BI 2018. https://powerbi.microsoft.com. (2018).

[40] Lin Qiao, Vijayshankar Raman, Frederick Reiss, Peter J. Haas, and Guy M. Lohman. 2008. Main-memory scan sharing for multi-core CPUs. *PVLDB* 1, 1 (2008).

[41] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. 2000. Efficient and Extensible Algorithms for Multi Query Optimization. In *SIGMOD*.

[42] Timos K. Sellis. 1988. Multiple-Query Optimization. *ACM Trans. Database Syst.* 13, 1 (1988), 23–52.

[43] Shayak Sen, Saikat Guha, Anupam Datta, Sriram K. Rajamani, Janice Y. Tsai, and Jeannette M. Wing. 2014. Bootstrapping Privacy Compliance in Big Data Systems. In *IEEE Symposium on Security and Privacy*. 327–342.

[44] Yasin N. Silva, Per-Åke Larson, and Jingren Zhou. 2012. Exploiting Common Subexpressions for Cloud Query Processing. In *ICDE*.

[45] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. 2001. LEO - DB2's LEarning Optimizer. In *VLDB*.

[46] Dimitri Theodoratos and Timos K. Sellis. 1997. Data Warehouse Configuration. In *VLDB*.

[47] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: a warehousing solution over a map-reduce framework. *PVLDB* 2, 2 (2009).

[48] TPC-DS Benchmark 2018. http://www.tpc.org/tpcds. (2018).

[49] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *EuroSys*.

[50] Guoping Wang and Chee-Yong Chan. 2013. Multi-Query Optimization in MapReduce Framework. *PVLDB* 7, 3 (2013).

[51] Xiaodan Wang, Christopher Olston, Anish Das Sarma, and Randal Burns. 2011. CoScan: Cooperative Scan Sharing in the Cloud. In *SOCC*. 11:1–11:12.

[52] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Åke Larson, Ronnie Chaiken, and Darren Shakib. 2012. SCOPE: parallel databases meet MapReduce. *VLDB J.* 21, 5 (2012), 611–636.

[53] Jingren Zhou, Per-Åke Larson, Johann Christoph Freytag, and Wolfgang Lehner. 2007. Efficient exploitation of similar subexpressions for query processing. In *ACM SIGMOD*. 533–544.

[54] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. 2007. Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS. In *VLDB*.