Production Experiences from Computation Reuse at Microsoft

Alekh Jindal
Gray Systems Lab
Microsoft
alekh.jindal@microsoft.com

Shi Qiao
Azure Data
Microsoft
shqiao@microsoft.com

Hiren Patel
Azure Data
Microsoft
hirenp@microsoft.com

Abhishek Roy
Gray Systems Lab
Microsoft
abhishek.roy@microsoft.com

Jyoti Leeka
Gray Systems Lab
Microsoft
jyoti.leeka@microsoft.com

Brandon Haynes
Gray Systems Lab
Microsoft
brandon.haynes@microsoft.com

ABSTRACT

Massive data processing infrastructures are commonplace in modern data-driven enterprises. They facilitate data engineers in building scalable data pipelines over shared datasets. Unfortunately, data engineers often end up building pipelines that have portions of their computations common across other pipelines over the same set of shared datasets. Consolidating these data pipelines is therefore crucial for eliminating redundancies and improving production efficiency, thus saving significant operational costs. We had built CloudViews for automatic computation reuse in Cosmos big data workloads at Microsoft. CloudViews added a feedback loop in the SCOPE query engine to learn from past workloads and opportunistically materialize and reuse common computations as part of query processing in future SCOPE jobs — all completely automatic and transparent to the users.

In this paper, we describe our production experiences with CloudViews. We first describe the data preparation process in Cosmos and show how computation reuse naturally augments this process. This is because computation reuse prepares data further into more shareable datasets that can improve the performance and efficiency of subsequent processing. We then discuss the usage and impact of CloudViews on our production clusters and describe many of the operational challenges that we have faced so far. Results from our current production deployment over a two month window show that the cumulative latency of jobs improved by 34%, with a median improvement of 15%, and the total processing time reduced by 37%, indicating better customer experience and lower operational costs for these workloads.

1 INTRODUCTION

Modern data-driven enterprises rely on large-scale data processing infrastructures for deriving business insights. As a result, over the last decade, a plethora of tools have been developed that have democratized scalable data processing for data engineers and data scientists. Examples include MapReduce [14], Spark [5], Hive [4], Presto [34], BigQuery [19], Athena [6], and SCOPE [9, 46]. However, the large-scale data processing infrastructures also incur massive operational costs and therefore improving their efficiency becomes very important in production. Interestingly, easy access to large scale infrastructure often leads data engineers to quickly build data processing pipelines that later end up having portions of computations repeated across one another. Furthermore, these redundancies are hard to discover in large enterprises with thousands of developers spread across different business units. Thus, we need automated tools to consolidate these data pipelines and improve operational efficiency, without impeding the developer productivity in quickly going all the way from raw data to actionable insights.

We see two key challenges when considering approaches for automatic compute reuse. First, it is very tedious to automatically detect the common computations in large volumes of complex analytical queries that are declarative and often include custom user code. Our analysis over a large window of production workloads, consisting of 67 million jobs and 4.3 billion sub-computations (referred to as *query subexpressions*), show that more than 75% of the query subexpressions are repeated. However, not all of the common computations are going to be viable candidates for reuse, e.g., due to very large storage overheads. Therefore, carefully detecting and selecting the common computations for reuse is a challenge. And second, there is a shift towards serverless query processing infrastructures [6, 9, 19], also sometimes referred to as job services, where users simply submit their declarative SQL-like queries without provisioning any infrastructure. As a result, users do not have any spare offline cycles for materializing the common computations before running their actual data analysis. In fact, users want to get started with their analysis quickly and they would rather have any optimizations applied in an online and adaptive manner.

We also see two key opportunities. First, there is a presence of large volumes of shared datasets in enterprises. This is because raw logs and telemetry coming in from various products are extracted and preprocessed into a shape and form that could be easily consumed by thousands of downstream developers. The resulting shared datasets are written once and read many times. Furthermore, they get regenerated periodically without requiring any fine-grained updates. As a result, the shared computations also do not need to be maintained with updates. And second, computation reuse holds the promise of significantly improving both the job performance and the operational efficiency, which are crucial in speeding up the time to insights and to enable developers to do more with the same set of resources (that are likely to have longer procurement cycles).

To address the above computation reuse problem, we had built CloudViews for automatic computation reuse in big data workloads at Microsoft [26]. Specifically, CloudViews optimizes the SCOPE query workloads in Cosmos big data analytics platform, that is used in various business, such as Bing, Windows, Office, Xbox, etc., across the whole of Microsoft. CloudViews identifies the common computations (query subexpressions) across different SCOPE jobs, selects the ones that could lead to more efficiency, materializes them as part of query processing with minimal overhead, reuses them in future jobs — all completely automatic and transparent for the users. In contrast to prior works on materialized views [20] and multi query optimizations [37], CloudViews materializes common query subexpressions as part of query processing, i.e., does not require any offline cycles, and automatically replaces older materialized views with newer ones when the shared datasets are bulk updated, i.e., no update maintenance. However, CloudViews considers only the same logical query subexpressions (with some normalization) for reuse, i.e., it does not consider view containment [21]. Although this is helpful in getting more accurate statistics for estimating the utility and cost of reusing different candidate subexpressions. CloudViews uses these estimates to select the set of subexpressions to materialize such that they provide the maximize reuse within a given storage budget [24].

In this paper, we describe our production experiences with CloudViews. We delve deeper into the data preparation process in Cosmos, referred to as *data cooking*, and discuss how computation reuse naturally augments data cooking by creating more shareable datasets that can boost the performance and efficiency of further downstream processing. While we previously showed the impact of CloudViews on TPC-DS workloads and a small set of production queries in pre-production environment, we now analyze the usage of CloudViews in production and compare the impact along several performance metrics. Apart from performance improvements, CloudViews also introduced an automated feedback loop in the SCOPE query engine, i.e., moving towards a self-tuning model [25, 26]. As a result, there were several operational challenges that we have faced and valuable lessons learned along the way. We discuss these challenges and also reflect back on our journey from research to product.

In summary, we make the following key contributions in this paper:

(1) We provide a detailed description and analysis of data cooking in Cosmos and reason about how compute reuse with CloudViews helps augment the data cooking process. We also discuss the impact of various design decisions that were made in CloudViews. (Section 2)
(2) We present an analysis of the usage and impact of CloudViews on our production workloads. Results from our production deployment show that over a two month window the cumulative latency of jobs improved by 34%, the total processing time reduced by 37%, and the total number of containers used for processing dropped by 36%. We also highlight some of the other non-obvious implications, including smaller inputs (by 36%), less data read (by 39%), and shorter queue lengths (by 13%). (Section 3)
(3) We discuss several operations challenges faced, including challenges in view selection when considering job scheduling, correctness guarantees, dependencies with other components, customer on-boarding, customer expectations, and quantifying the impact over constantly changing workloads. (Section 4)
(4) Finally, we reflect back on the journey from taking a research prototype to production, describe how the ability to create derived data as part of query processing is a powerful mechanism in general, and present insights for many of the open opportunities that we see going forward (Section 5).

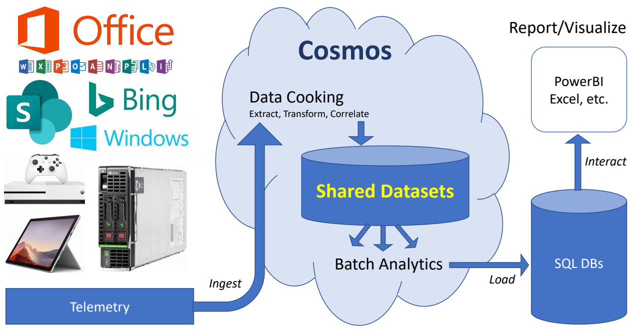


Figure 1: Illustrating typical data cooking in Cosmos.

2 DATA COOKING

Cosmos powers the internal big data analytics at Microsoft, with a massive infrastructure consisting of multiple 50k+ node analytics clusters [35], and processing declarative analytical queries using the SCOPE query engine. Cosmos runs hundreds of thousands of batch SCOPE jobs per day, consuming millions of containers and crunching over several petabytes of data generated per day. Almost 80% of the SCOPE workloads are recurring in nature [28], i.e., similar jobs templates are executed periodically at regular intervals over new data sets and parameters. Furthermore, SCOPE jobs have dependencies across each other. In fact, 80% of the jobs depend on at least one other job, while 68% have dependencies in a recurring fashion [12]. This is because of the presence of large volumes of shared data sets in Cosmos. Below we describe this enterprise pattern in more detail.

2.1 An Enterprise Pattern

Figure 1 illustrates the typical data cooking pipeline in Cosmos, where the raw telemetry data from different Microsoft products and services are ingested into the Cosmos store. Thereafter, the data cooking process extracts the structured data, transforms it into better representation, and correlates across multiple sources. The resulting shared datasets are generated periodically and consumed in multiple downstream batch-oriented analytics. Aggregated data is then loaded into interactive query processing systems like SQL databases for interactive analysis, reporting, and visualization using tools such as PowerBI or Excel. We can see that data cooking and the ability to collect and process large volumes of shared datasets across various developers and even business units is at the core of the above pipeline.

Figure 2 shows cumulative distributions of shared data sets and their consumers in five of our production clusters over a one-week window. We can see that more than half of the datasets are shared across multiple distinct consumers. Furthermore, several datasets are consumed tens to hundreds of times, with few getting reused thousands of times as well. Cluster1 in particular sees more shared data sets since that feeds into the Asimov platform that *implements a new mechanism for user feedback, allow for the testing of new features to gauge user acceptance, track bugs, and easily roll out new functionality and fixes* [33]. In fact, 10% of the inputs on this cluster get reused by more than 16 downstream consumers. For other clusters, 10% of the inputs are consumed by 7 or more downstream consumers. Thus, we see that shared data sets are prevalent in Cosmos.

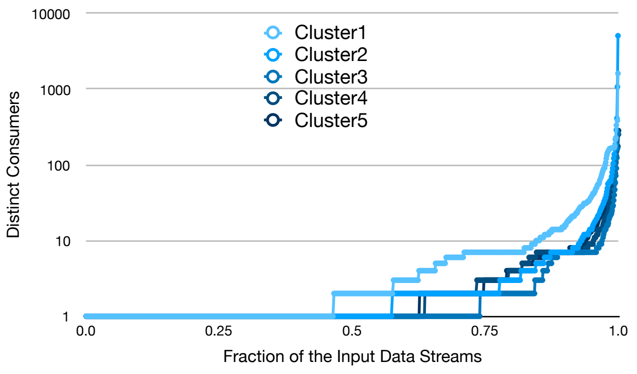


Figure 2: Shared data sets in five production clusters.

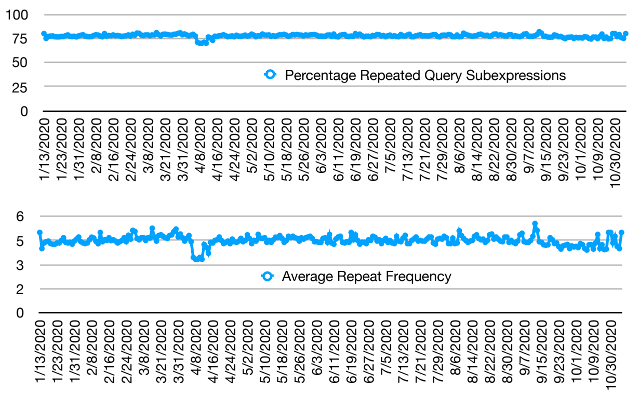


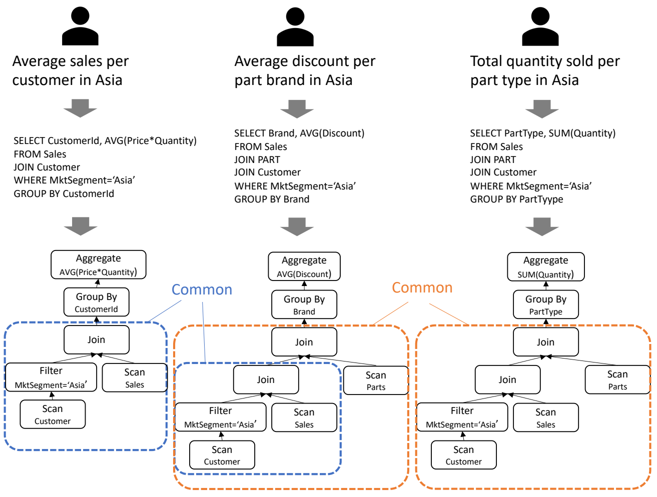
Figure 3: Overlaps in production clusters.

2.2 Augmented Data Cooking

We saw that enterprise data analytics involves cooking massive volumes of data into a form that is consumable by several users. The shared datasets are therefore also a natural habitat for shared data analytics. Furthermore, it turns out that computation reuse is a natural problem in shared data analytics since often there are same sets of transformations that are applied repeatedly by analysts over the same shared datasets. Ideally, these shared computations could be captured in the cooking process itself. However, that is hard practically due to the lack of visibility into the downstream analytics (e.g., by different teams or business units consuming the shared datasets), complexity of SCOPE queries where it is non-trivial to identify the shared computations (portions of declarative large queries, including user defined functions, that may end up getting compiled to the same query sub-plan), or even due to the evolving nature of the analytics (new reports or dashboards being created). In a way, computation reuse can *augment* the handcrafted data cooking process by further fine-tuning the shared datasets with reusable views that are automatically identified, adapted, and created just in time, based on the workloads. Thus, we argue for computation reuse to be a first-class citizen in conjunction with data cooking for shared data analytics.

We analyzed the overlaps in our workloads in five of our production clusters over a 10-month window (January–October, 2020). Overall, there were 67 million jobs consisting of 4.3 billion query subexpressions from 2.5K users and 776 virtual clusters[1].

[1]A virtual cluster represents a sub-cluster that is dedicated for one particular customer or business unit.

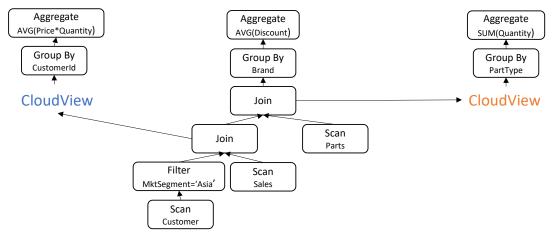


```
SELECT CustomerId, AVG(Price*Quantity)
FROM Sales
JOIN Customer
WHERE MktSegment='Asia'
GROUP BY CustomerId
```

```
SELECT Brand, AVG(Discount)
FROM Sales
JOIN PART
JOIN Customer
WHERE MktSegment='Asia'
GROUP BY Brand
```

```
SELECT PartType, SUM(Quantity)
FROM Sales
JOIN PART
JOIN Customer
WHERE MktSegment='Asia'
GROUP BY PartTyype
```

(a) Query plans with common computations across different users.



(b) Modified query plans with computation reuse across users.

Figure 4: Illustrating computation reuse across three analysts working on the same datasets.

Figure 3 shows that more than 75% of query subexpressions are consistently overlapping over the 10-month window. Furthermore, the average repeat frequency consistently hovers around 5, indicating that materializing and reusing could be helpful for many of these overlapping computations. We refer interested readers to [26] for more fine-grained analysis of overlaps over a single day window.

In summary, computation reuse can fill the gaps in data cooking, and we see significant opportunities for computation reuse in our production workloads.

2.3 CloudViews Overview

Figure 4 illustrates an example scenario of computation reuse across three different users who are analyzing the same shared datasets (which include Customer, Sales, and Parts tables) and analyzing the sales behavior in the same Asia region. We can see how the insights that they are looking for is expressed as SQL queries that get compiled into query plans, and even though the user queries might appear very different, their query plans turn out to have significant portions in common (shown in orange and blue boxes). The analysts may identify that they are all analyzing the same market segment, i.e., the plain sentence version of the insights they are looking for has “In Asia" in common. This may lead them to create indexes or vertical partitions on the customer table. However, the query plans in the bottom of Figure 4a shows much large computations, called *query subexpressions*, are shared across them. Furthermore, different sets of computations are shared across different sets of users. To really discover these opportunities, the analysts would have to understand their analytics more deeply and then coordinate amongst themselves to manually share the common computations, requiring a lot of manual effort that is simply not possible at enterprise scale.

CloudViews addresses the above challenges by introducing automatic computation reuse in an online fashion. Figure 5 shows the overall system architecture of CloudViews and the various steps involved. We briefly summarize them below. CloudViews leverages the presence of large workloads in modern clouds [25]. It extracts the query workload into a denormalized subexpressions table that pre-joins the logical query subexpressions with their runtime metrics as seen in the history. Thereafter, we identify the common subexpressions across queries using a strict subexpression hash, known as *signature*, that uniquely captures a subexpression instance including its inputs used. From the set of all common subexpressions, we select the useful set of expressions to materialize and reuse. Some of the considerations for reuse include storage cost for materialization, processing time saved when reused, saving opportunities per customer, and the presence of concurrent queries that may not benefit from materialization-based reuse. Users can provide storage and other constraints (e.g., maximum number of views to create) for view selection. The view selection output is also made available to customers for insights and expected overall benefits. For the selected views, we collect their corresponding *recurring signatures* that discard time varying attributes like parameter values and input GUIDs, and are likely to remain the same in future instances of the recurring workloads. For easier lookup and control, we generate tags for each of the signatures that help fetch relevant signatures for a given SCOPE job and could also be used for access control. These tagged signatures are then polled by insights service and stored using Azure SQL databases. We also generate a query annotations file with the selected signatures that could be used for quickly debugging any job. For instance, in case of a customer incident, we can reproduce the compute reuse behavior by compiling a job with the annotations file.

At query time, for an incoming SCOPE job, the compiler extracts its tags and fetches the annotations from the insights service. These annotations are then parsed and stored in the optimizer context. During core search, the optimizer tries to match top down (match larger subexpressions first) whether any of the query subexpressions is already materialized. If yes, then it modifies the query plan to reuse the common subexpression with scan over previously materialized subexpression, updates more accurate statistics, and inserts the modified plan into the memo for overall costing. The plan using a materialized subexpression is chosen only if its cost is lower than the plan without the materialized subexpression. In either case, there is a follow-up optimization phase to check (in bottom-up manner) if any of the subexpressions are candidates for materialization. If yes, then an exclusive lock is obtained from the insights service and a spool operator with two consumers is added to that subexpression: one feeds into the rest of the query processing while the other materializes the common subexpression to stable storage. During execution, the job manager makes the view available even before the query finishes (referred to as early sealing in Cosmos), and notifies the insight service to release the view creation lock and start reusing it wherever possible. The modified query plans are surfaced to the users in the query monitoring tool and also logged into the telemetry for future analyses.

We refer interested readers to [26] for more details on each of the components in the above architecture.

2.4 Design Decisions and Limitations

We now highlight some of the key decisions that have served our workloads well while discussing some of the key limitations of our approach. The key things that worked well include:

- **Preserving query boundaries.** One way to reuse computation could be to combine query plans of overlapping SCOPE jobs into merged query plans. However, this is inconvenient since it requires changes to submission system and hard to explain the cost of merge query to different users. Also, fault tolerance becomes more intertwined due to the hard dependency between multiple jobs. CloudViews keeps job boundaries intact while opportunistically reusing computations wherever possible.
- **Online materialization.** CloudViews materializes common computations in an online manner as part of query processing, i.e., it does not require any offline cycles for materialization. As a result, CloudViews is easier to manage operationally without requiring uses to submit additional queries while also hiding the materialization latencies in large complex query DAGs.
- **Just-in-time views.** CloudViews are also materialized just-in-time when the first query hits a particular query subexpression instance. This means that: (i) the storage space for materialized views is consumed only when the views are about to be reused, and (ii) if the workload changes and a selected subexpression is no longer found in the workload then it will automatically stop being materialized.
- **Accurate cost estimates.** Traditional view selection approaches suffer from poor cardinality and cost estimates in query optimizers [40, 42] since they explore alternate query plan expressions that may not have been executed in the past. However, by considering only the same logical subexpressions for reuse, CloudViews is able to leverage the actual runtime statistics seen in the past instances of those subexpressions. As a result, it can make better decisions about the views that could improve performance when reused and the storage costs associated with them.
- **Scalable view selection.** View selection over large workloads is non-trivial since it explodes the search space exponentially. This is because traditional view selection algorithms consider more generalized sets of views, ones that may not have appeared in past query executions, and select the most useful ones from them. However, by restricting to common subexpressions, CloudViews can run subexpressions selection to Cosmos scale by running it as a label propagation problem in a distributed manner [24].
- **Lightweight view matching.** Traditional view matching require expensive containment checks during query optimization to determine whether a query could be answered from a view or not. CloudViews replaces that with lightweight hash equality checks that only require to recursively compute a signature for each subexpression and then match them with the signatures of one or more available views.

Some of the limitations of our approach are as follows:

- **Exact logical subexpression match.** The most obvious limitation of CloudViews is that it can only reuse the exact same logical query subexpressions, although they can have different physical implementations. While there are numerous opportunities for such reuse, there is even

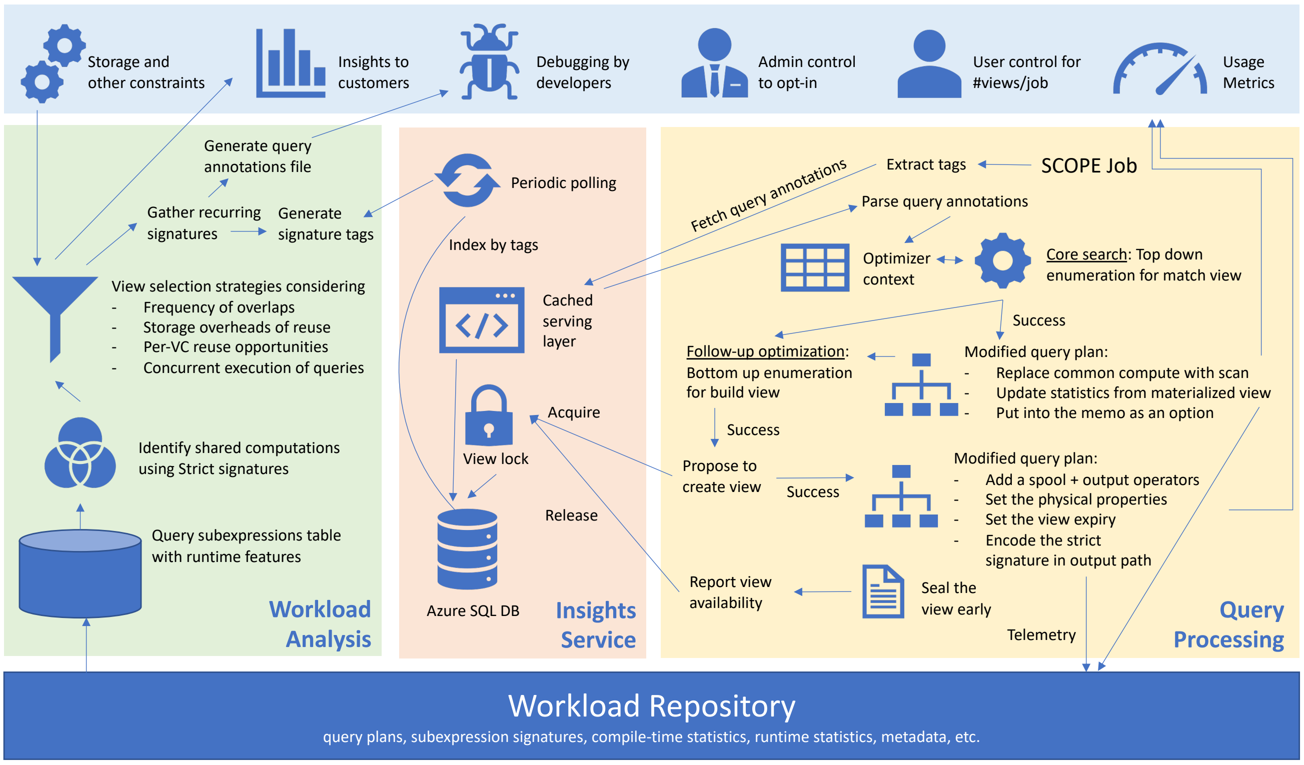

Storage and other constraints
Insights to customers
Debugging by developers
Admin control to opt-in
User control for #views/job
Usage Metrics
Generate query annotations file
Gather recurring signatures
Generate signature tags
View selection strategies considering
- Frequency of overlaps
- Storage overheads of reuse
- Per-VC reuse opportunities
- Concurrent execution of queries
Identify shared computations using Strict signatures
Query subexpressions table with runtime features
Workload Analysis
Periodic polling
Index by tags
Cached serving layer
Acquire
View lock
Release
Azure SQL DB
Insights Service
Fetch query annotations
Extract tags
SCOPE Job
Parse query annotations
Optimizer context
Core search: Top down enumeration for match view
Success
Follow-up optimization: Bottom up enumeration for build view
Modified query plan:
- Replace common compute with scan
- Update statistics from materialized view
- Put into the memo as an option
Success
Propose to create view
Success
Modified query plan:
- Add a spool + output operators
- Set the physical properties
- Set the view expiry
- Encode the strict signature in output path
Report view availability
Seal the view early
Telemetry
Query Processing
Workload Repository
query plans, subexpression signatures, compile-time statistics, runtime statistics, metadata, etc.

Figure 5: The CloudViews architecture for computation reuse in Cosmos.

more potential for creating more generalized views and reusing them in queries that are contained in those views. We discuss this further in Section 5.

- **Concurrent queries.** CloudViews requires materializing common subexpressions before they can be reused in subsequent queries. Although the materialized views are made available as soon as the subexpression portion of the first query finishes (early sealing), still CloudViews cannot help queries that are submitted concurrently unless their submission schedule is altered, which is typically harder in production environments.
- **Not maintained.** CloudViews are treated as cheap throw-away views that are recreated whenever the inputs change. While this eradicates the need for view maintenance strategies, it also results in recreating views over sets of inputs where most have remained unchanged. This is particularly true for recurring queries with a sliding window, e.g., last seven days, where all except the most recent input in the window might remain same.
- **No DDL for user visibility or tuning.** CloudViews are created transparently to the users without registering them as any DDL statements. As a result, users do not have direct visibility into the catalog of available views at any given point nor can they reason about them. They can, however, see the CloudViews-generated files, given that they are stored in user-specified location, and even purge views whenever necessary.
- **User expectations.** CloudViews causes the first query hitting a common subexpression to slow down due to additional materialization overhead. Therefore, the users need to be informed about some queries getting impacted for overall workload efficiency.

Jobs	257,068
Pipelines	619
Virtual Clusters	21
Runtime Versions	12
Views Created	58,060
Views Used	344,966
Latency Improvement	33.97%
Processing Time Improvement	38.96%
Bonus Processing Time Improvement	45.01%
Containers Count Improvement	35.76%
Input Size Improvement	36.38%
Data Read Improvement	38.84%
Queuing Length Improvement	12.87%

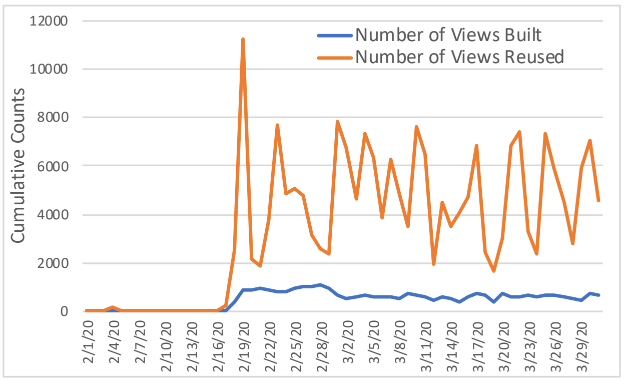
Table 1: Production Impact Summary

3 PRODUCTION IMPACT

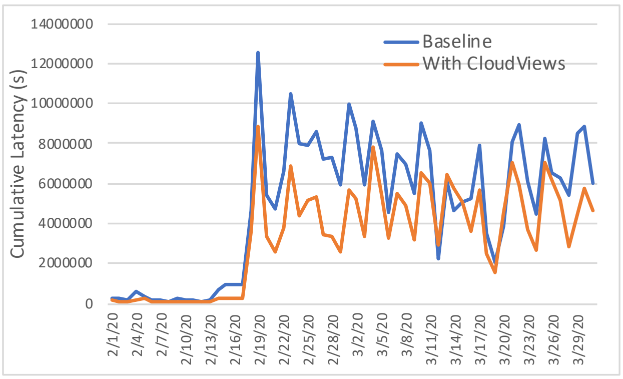
In this section, we describe the impact of CloudViews when deployed for several customers over a two-month window (February–March 2020). Table 1 shows the summary of workload and the performance impact. Below we discuss them in more detail.

3.1 Usage

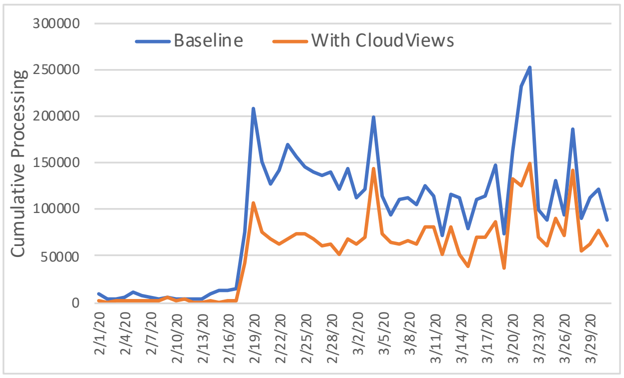
Let us first look at the usage numbers. Our current deployment strategy was *opt-in*, i.e., the feature was made available for customers and they can choose to enable it on their virtual clusters. Overall, we see more than 250k analytical SCOPE jobs across 21 virtual clusters using CloudViews. These jobs were from 619 unique data pipelines and ran over 12 different SCOPE runtime versions. Figure 6a shows the number of views materialized and



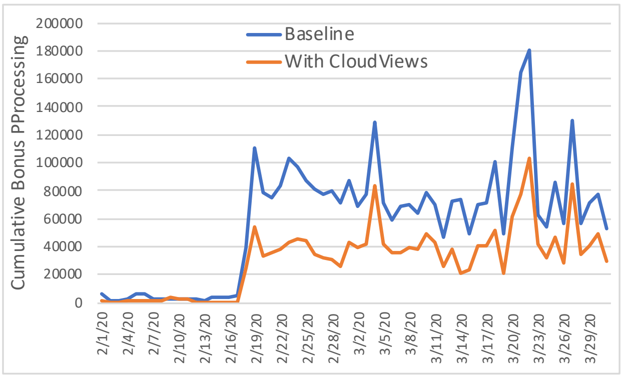
(a) Usage



(b) Latency



(c) Processing Time



(d) Bonus Processing Time

Figure 6: The usage and impact of CloudViews on production workloads.

reused over a two-month window. After an initial customer onboarding period, we see a periodic view creation and reuse pattern. Naturally, much more views are reused than created every day. Overall, approximately 58k views were created and they were reused 350k times, each view being reused almost 6 times on average. Our current eviction policies expire each of the views after one week of creation, thus consuming a fixed amount of storage (that is configured by the customers and affects the number of views selected for reuse) in the stable state.

3.2 Latency

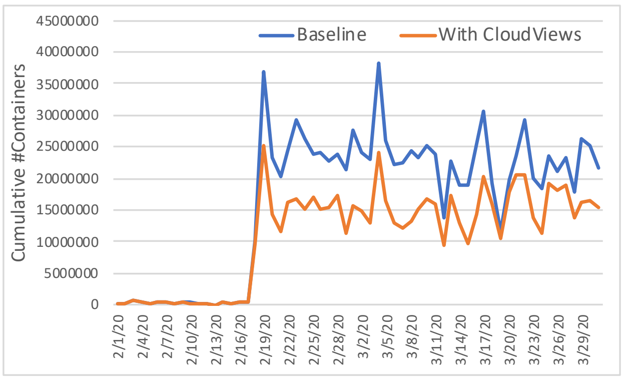
Latency is a crucial metric for customers and so we next explore the impact of CloudViews on job latencies. Figure 6b shows the daily cumulative latencies of jobs when CloudViews was enabled, compared to if it was not enabled, over the same two-month window. We can see that even though view materialization incurs an overhead, overall there is a gain in cumulative latencies of the jobs. This is because we materialize CloudViews in an online fashion in a separate stage that runs in parallel and hence the impact of latency is typically less. At the same time, we also see that the latency improvements are staggered and minimal on several days. This is because computation reuse could only improve latency if the portions of the query graph that was reused lies on the critical path of the job. Given that our view selection strategies do not optimize for that (since the objective function is to maximize for total compute), latency improvements are not guaranteed, especially with large query DAGs where overlapping computations may not be on the critical path. Still, we see a median per-job latency improvement of 15% and a cumulative overall improvement of close to 34%, that is significant for improving the customer experience in production workloads.

3.3 Processing Time

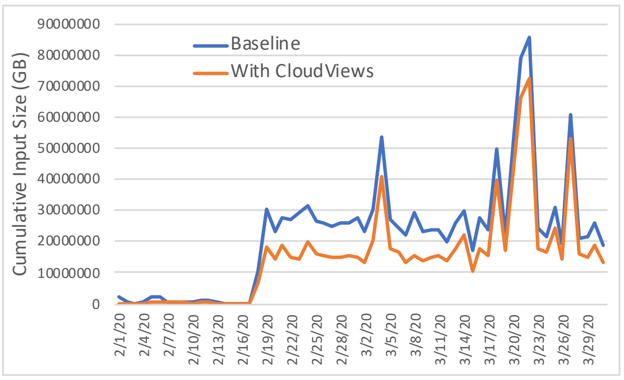
We now look at the impact on total processing time (i.e., the sum of processing time of all the containers used in the jobs) which is often considered a better measure of compute efficiency. Figure 6c shows the cumulative processing costs per day when CloudViews was enabled compared to if it was not enabled. Indeed, in contrast to latency, we can see more distinct change in processing time, with close to 39% improvement overall. This is because processing time savings do not depend on the critical path and any reuse in the query graph contributes to some savings, modulo the time spent in reading the materialized shared computation. Strictly speaking we also need to discount the extra processing time spent in writing the shared computation in the first job. Since we look at the observed processing times in the cluster all of these overheads automatically get accounted for. The processing time savings validate the utility of CloudViews to improve cluster efficiency and to free up spare resources that will let developers do more with the same set of resources.

3.4 Bonus Processing Time

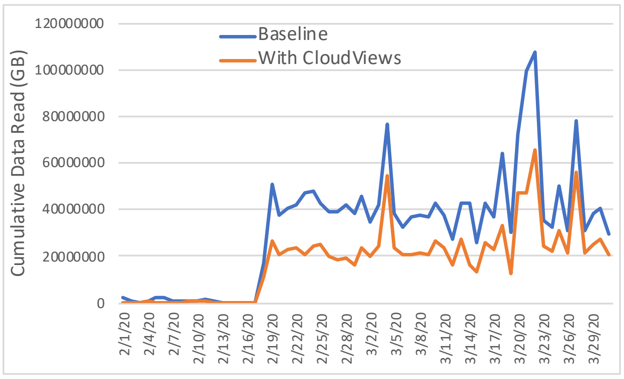
Cosmos employs an opportunistic resource allocation policy to improve cluster utilization [8]. The idea is to allocate unused resources opportunistically to jobs in case they could use them, e.g., one or more stages in a job has more partitions than the number of containers available to the job or stages that could start making



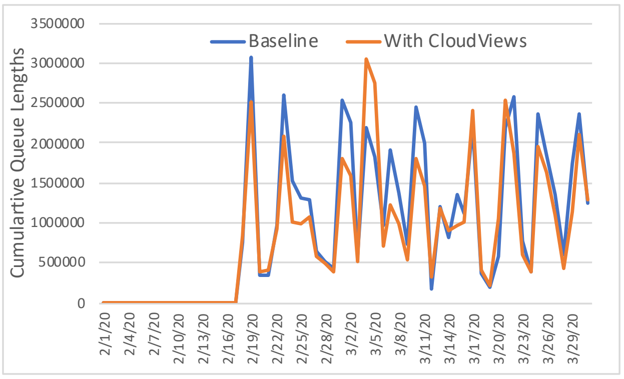
(a) Containers



(b) Input Size



(c) Data Read



(d) Queuing Lengths

Figure 7: Other non-obvious impact of CloudViews on production workloads.

progress in parallel. We record the processing time using opportunistic resources as bonus processing time, and this is helpful to not only improve the utilization but to also improve SCOPE job performance by dynamically leveraging unused cluster capacity. Unfortunately, bonus processing also leads to unpredictability in performance, i.e., the job runtimes may vary a lot based on the unused capacity at any given time in the cluster. Therefore, reducing the reliance on bonus processing is desirable for more predictable job behavior. Interestingly, it turns out that by sharing common computations and not re-executing them with a lot of variance each time, CloudViews can reduce the reliance on bonus processing and hence improve job predictability. Figure 6d shows the reduction in bonus processing time with CloudViews. We can see a significant change in bonus processing time with an overall reduction of 45%, which is important for improving the overall system reliability.

3.5 Containers

While latency and processing are expected to be impacted by computation reuse in CloudViews, we also discover other rather unexpected implications. In particular, the total number of containers used in each SCOPE job is an important measure of resource consumption and computation reuse can also help improve that. Figure 7a shows the change in the cumulative number of containers used with CloudViews. We can see that similar to the total processing time, total number of containers also show significant improvement both for each day (and also for each job) and also overall (36% fewer containers used). This is because eliminating re-computation of large expensive chunks of query DAGs also eliminates the corresponding set of resources used to process those computations. Furthermore, due to the challenges in estimating cardinality over big data workloads, SCOPE query engine often ends up overestimating cardinalities and thus over-partitioning the intermediate outputs, leading to many more containers getting instantiated and each processing relatively smaller amounts of data [42]. Computation reuse automatically circumvents this issue by avoiding re-execution of such common computations in the first place and thus saving resources that would been otherwise be consumed due to mis-estimation. In fact, computation reuse further helps feed more accurate statistics from the previously materialized subexpressions to the rest of the query plan.

3.6 Input

Another less anticipated effect of computation reuse is the size of the input read. Figure 7b shows a sizeable reduction in the total inputs sizes read by all queries in that workload. This is because quite often the inputs datasets are filtered, selectively joined, or aggregated before they are materialized as common subexpressions, which end up being much smaller than the initial input sizes. Smaller input sizes not only reduce IO but also improve the container efficiency since SCOPE jobs are widest at the beginning (due to very large input sizes) and then their width drops significantly, causing lots of allocated containers remaining unused in a large portion of the query [7]. Smaller input sizes avoid such extreme container usage over the course of a job execution. Furthermore, it also eases the pressure on the storage layer, which is increasingly disaggregated from the compute in modern clouds [35], and thereby helps in reducing the throttling during peak load conditions.

3.7 Data Read

Apart from the input data read, let us also look at the total data read. Figure 7c shows the change in the overall data read and it is very similar to the trend of input read, although overall, data read improves by 39%, which is more than the improvements in input read. Reducing data reads eases the overall IO pressure including both the persistent store and also the local temporary store that is used to write intermediate outputs for each of the SCOPE jobs. This is significant because there could be hotspots with significant intermediate IO for large SCOPE DAGs and fewer data read can alleviate some of these hotspots.

3.8 Queue Lengths

SCOPE jobs are processed in a job service form factor, where users submit their jobs and they are queued until there are enough resources available for them to be scheduled. Interestingly, computation reuse can even help reduce the queue length due to less computations being done by each job which causes them to finish faster. Figure 7d shows the cumulative queue lengths seen by all jobs for each day in the workload, with a reduction in queue lengths on several days and an overall reduction of 13%. Shorter queue lengths improve the user experience and reduces their time to insights, enabling tighter SLAs in many cases. It also helps execute queries with more recently optimized query plans, e.g., leveraging more recently generated CloudViews, that would otherwise be missed by jobs sitting in longer queues.

4 OPERATIONAL CHALLENGES

In this section, we describe some of the operational challenges that we have faced. Note that these exclude the multiple rounds of customer feedback and feature improvements we did before deploying CloudViews in production.

- **Schedule-aware views.** Cosmos customers have built several workflow tools to schedule and monitor SCOPE jobs periodically. In some cases, these tools trigger all jobs at the start of every period, with the goal of finishing them before the period ends and to avoid missing their SLAs. Given that CloudViews requires materializing the common computation before it can be reused, jobs that get scheduled (and thus compiled) at the same time cannot benefit from such reuse. One option could be to alter the job submission and add a little lag to jobs that could reuse common computations. However, it turned out to be very hard to convince customers to change their job submission schedules. Instead, we modified our view selection algorithms to account for concurrent job submissions; specifically, we only consider subexpressions that could finish materializing before the start of other consuming jobs.
- **Per-customer view selection.** The data cooking process could create shared datasets across multiple customer virtual clusters (VCs), leading to computation reuse opportunities across multiple VCs as well. However, customers often care about the reuse and performance improvements in each of their individual VCs. This is because they want to benefit from better SLAs and do more processing on a per-VC basis. Furthermore, the cost of storing the common computations could be significant (depending on how much reuse is done) and customers often want to keep it separate for each VC. This means that selecting the views to materialize globally is not enough but rather we need to select them for each virtual cluster. Given that there are thousands of virtual clusters in Cosmos, it is not possible to run view selection separately for each of them. At the same time, a single view selection script that partitions workloads by virtual clusters needs to also consider the constraints per VC (see top left in Figure 5). It also makes it hard to maintain the single script with evolving requirements for different customers.
- **Signature correctness.** The core of CloudViews relies on a signature (i.e., a hash) to identify logical subexpressions. While this is trivial for native operators in the SCOPE engine, things can get murky with the user defined operators (UDOs) and the libraries used in it. In particular, it could be very difficult to compute the signatures in user code that involves recursively dependent libraries (there are extreme cases in Cosmos with very deep dependency chains). Traversing these long chains could slow down the entire compilation process. Furthermore, in some extreme cases, the UDOs may even contain non-determinism by design, e.g., `DateTime.Now`, `UTCNow`, `Guid.NewGuid()`, or `new Random().Next()`. It is not clear what the correct semantics are when computing signatures over such UDOs. Our approach is to exclude such extreme cases to avoid failures or incorrect results, i.e., we skip any computation reuse if the dependency chain is too long or if a UDO is found to contain non-determinism.
- **Impact of changed signatures.** While it is understood that the signatures will change with the workloads, sometimes they also evolve with new SCOPE runtime (due to changes in compilation, optimizer representation, or other code changes). As a result, all existing materialized views get invalidated. Thus, evolving signatures is very tricky since we need to keep track of changes that can affect signatures and re-run any prior workload analysis.
- **Other dependencies.** From Figure 5, we can see that CloudViews depends on other components, such as the offline workload analysis, insights service, the job manager to seal the view early, and the various customer interactions. However, these components often evolve independently and so they need to be kept in sync. For instance, the early sealing had to be ported to new job manager versions, the view storage locations need to be migrated to ADLS [13], and the insights service had to be scaled.
- **Handling GDPR requirements.** The emergence of new privacy regulations, such as [16], mean that we cannot simply look at the input paths but need to also keep track of when the inputs have changed due to *forget* requests and automatically stop consuming them henceforth. We handled this by ensuring that the input GUIDs are updated both with recurring updates and with GDPR related updates, which are handled separately in our storage layer.
- **Opt-in vs opt-out.** Given that CloudViews trades computation reuse for storage costs (and some latency overheads in the first job that hits the common computation), we need to make customers aware of the expected costs and benefits. As a result, we adopted an *opt-in* model of deployment where only the customers who bought in were onboarded. This also helped us address bug fixes and other deployment issues more gradually. However, *opt-in* also requires significant customer interaction eating up a lot of program manager time. As a result, it is not scalable to large number of customers. Therefore, after sufficient

hardening of the CloudViews feature in production, we have now started enabling it using an *opt-out* model, where virtual clusters are grouped into tiers (based on business importance) and they are automatically onboarded tier by tier, starting with the lowest tier.

- **Multi-level control.** We ended up placing several levels of control to enable or disable CloudViews. These include job-level control for individual developers to toggle CloudViews in their jobs, VC-level control for enabling or disabling specific VCs once they decide to onboard or opt-out, cluster-level to make the feature enabled or disabled across the board for the entire cluster instead of doing it for each of the thousands of VCs on each cluster, and insight service level control as the uber control for gate keeping and toggling during customer incidents.
- **Measuring impact.** Finally, while it is easy to measure performance improvements in a pre-production environment by re-running both the baseline and the modified version, it is less simple to measure performance once a feature is deployed in production. This is because it is very difficult to draw the baseline in a constantly changing production environment, e.g., when input sizes in recurring jobs change significantly [40]. It is also not possible to re-run all production jobs with CloudViews disabled to establish a baseline. Therefore, we took the following approach to identify baseline performance: we took previous instances of the queries that qualified for CloudView optimization and collected four weeks' worth of observations before enabling CloudViews on them. Thereafter, we took the 75th percentile value of each of the performance metrics, such as latency and processing time, and compared them with each of the newer instance of that query once CloudViews was enabled.

5 LOOKING BACK AND FORTH

We now reflect back on our journey from research to production and discuss many of the next set of problems we see in the area of computation reuse.

5.1 From Research to Production

CloudViews has come a long way from research to production, starting from our initial research ideas in 2015 and then us spending the next five years for prototyping in the SCOPE codebase, rallying product team support around it, getting valuable customer feedback, fixing many of the bugs that were discovered, onboarding customers first by opt-in and later by opt-out approach, and finally addressing many of the operational challenges. More significantly, however, CloudViews introduced a self-tuning feedback loop to the SCOPE query engine, a significant departure from using just the compile-time estimates to leveraging how things went in the past for query optimization. Along the way, we also learned valuable lessons for doing applied research within a product group setting [23]. A key amongst them is the realization that productization of research ideas is often a much longer and sometimes even a painful journey, that requires perseverance and willingness to adapt. This is because production features need to consider a lot of corner cases. The new DevOps model where there is no dedicated testing team anymore, combined with the cloud service form factor of modern software, acts as a forcing function for product teams to put all requisite safeguards for a consistent user experience and lower maintenance overhead.

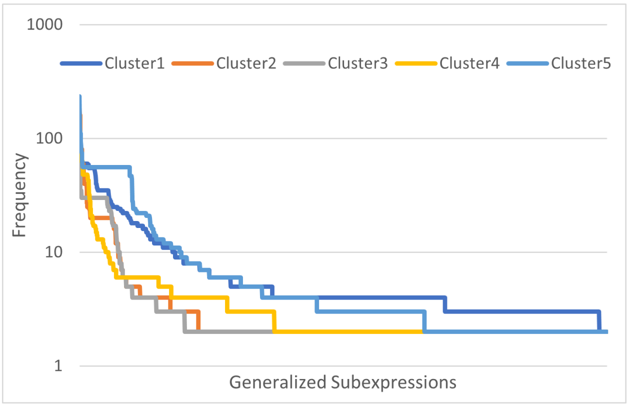


Figure 8: Opportunities for more generalized views: the x-axis shows the subexpressions that join the same sets of inputs, and the y-axis shows their corresponding frequency.

Finally, applied research turns out to be a highly collaborative endeavor, right in the trenches of product teams, and therefore it needs to be appreciated as such.

5.2 Towards Broader Workload Optimization

CloudViews helped open up the area of workload optimization for cloud query engines, leveraging the large workload telemetry that are visible, with appropriate anonymizations, in modern cloud environments [25]. This resulted in a mindset change from optimizing just a query at a time to also consider optimizing the entire workload, something which customers care a lot in order to manage their total cost of ownership (TCO). Specifically, the notion of signatures to uniquely identify query subexpressions turned out to be very helpful not just for computation reuse, but also for applications such as discovering interesting query patterns in the workload, learning high accuracy micro-models for specific portions of the workload [27], compressing workloads into a representative set for pre-production evaluation, and surfacing data and job dependencies for interesting pipeline optimizations. Likewise, the insights service evolved into an independent component that could serve many different kinds of insights, e.g., cardinality [42], cost [40], or resources [39]. The insights could be scaled and bulk loaded upfront to the SCOPE optimizer, with an end to round trip latency of around 15 milliseconds. All of the above resulted in common pieces of infrastructure that could be leveraged across several new optimizations in SCOPE, and even for other query engines like Spark.

5.3 Generalized Reuse

CloudViews identifies view candidates using per-operator signature matching, which establishes syntactic equivalence but does not consider views that differ syntactically but are otherwise logically equivalent (e.g., `SELECT * FROM Sales WHERE CustomerId > 5` and `SELECT * FROM Sales WHERE 2 * CustomerId > 10`). However, while relaxing this constraint may offer additional performance improvements by enabling the reuse of more views, computing logical equivalence is expensive and in general undecidable. Nonetheless, recent work has pushed down this cost to the point where exploiting these opportunities may be feasible for many queries [10, 11, 38, 47, 48]. Evaluating this precise trade-off, and quantifying the number of materialized views that more general

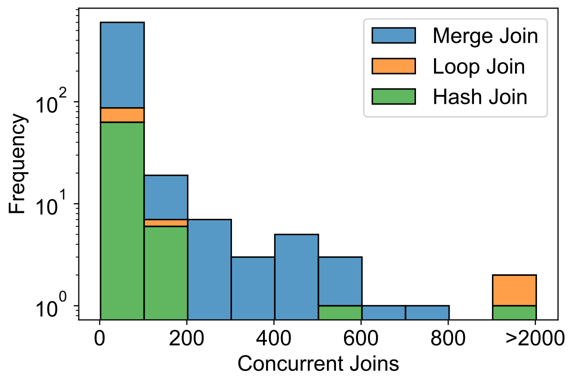


Figure 9: Concurrently executing joins on a Cosmos cluster occurring in a single day. This workload contained two outliers that were concurrently executed 2016 and 23,040 times.

logical equivalence would allow, remains a ripe avenue for future research.

Similarly, equivalence itself (logical or otherwise) is more restrictive than necessary for view reuse, and in many cases containment may offer similar opportunities for reusing computation (e.g., materializing `SELECT * FROM Sales WHERE CustomerId > 5` and using it to answer the query `SELECT * FROM Sales WHERE CustomerId > 6`). Like logical equivalence, containment is a hard problem (in general NP-complete, although polynomial-time algorithms exist for some subproblems), has been extensively explored in the literature, and is expensive to compute. Nonetheless, some recent work has used machine learning to evaluate containment rates [22], and many techniques used to efficiently compute logical equivalence may be applicable to containment as well.

Figure 8 shows the reuse opportunity when considering one kind of generalization: subexpressions that join the same sets of inputs. These subexpressions could still have different projections, selections, or group by operations, which could be merged to create more general materialized views and then later query could be rewritten using containment checks. Figure 8 shows the opportunity over the same five clusters as in Figures 2 and 3, and we see lots of generalized subexpressions with frequencies on the order of 10s to 100s.

5.4 Reuse in Concurrent Queries

CloudViews materializes views for reuse in subsequent queries, which is necessary for queries that are temporally non-overlapping. However, opportunities for reuse exist for *concurrent* queries, which does not require pre-materialization since intermediate results may be directly pipelined. While at first this might seem like an infrequent occurrence, we observed thousands of such opportunities per day in our production workloads. Figure 9 illustrates these opportunities for concurrently-executing joins within a single Cosmos cluster over a single day. We see that several join instances that are found to be concurrent hundreds to thousands of times.

Extending CloudViews to support concurrently executing queries and extending its feedback loop to efficiently learn the trade-offs between immediate reuse and materialization remains a ripe direction for future exploration.

5.5 Reuse in Other Engines

The idea of computation reuse goes beyond the SCOPE query engine. In fact, we adapted the computation reuse ideas to the Spark query engine as part of the SparkCruise project [36]. SparkCruise selects high utility common computations and performs automatic materialization and reuse for Spark SQL queries. Like CloudViews, SparkCruise analyzes past application workload logs to select common subexpressions for reuse. The list of common subexpressions is provided to the Spark query optimizer for future materialization and reuse. All these actions are performed automatically without any active involvement from the customer. On TPC-DS benchmarks, SparkCruise can reduce the running time by approximately 30% [36].

Even though both CloudViews and SparkCruise share the same ideas, there are differences in the target systems and the deployment environments. SCOPE query engine is developed by Microsoft and we added the signatures and optimizer rules deep inside the query optimizer. However, Spark is an open-source project and making any code changes in Spark will tie us to a specific version and delay the upgrade process in the future. Therefore, we use the optimizer extensions API in Spark to add two additional rules to the query optimizer — first for online materialization, and second for computation reuse. We also implemented an event listener for Spark SQL that can log query plans and compute signature annotations on the logical query plan object. The user simply needs to add SparkCruise library and set a couple of configuration parameters. With these changes, we can provide computation reuse in Spark without modifying its code.

Currently, SparkCruise is deployed on Azure HDInsight [32]. Azure HDInsight offers Spark cluster-as-a-service. Users can spin up ephemeral Spark clusters, run their query workloads, and delete the cluster after the job has finished. This scenario requires a fast workload-based feedback loop. To enable this fast feedback loop, we gave the control of the workflow to the end users or the data engineers. The users can schedule the workload analysis and view selection job periodically, or as often as the changes in the query workload. To help users understand their query workloads and decide whether SparkCruise will benefit their workload or not, we provide an interactive Workload Insights Notebook in Python [31]. The Workload Insight Notebook shows the workload statistics in aggregate as well as the redundancies in the workload. The results from the notebook can convince the users to enable the computation reuse feature on their workloads.

CloudViews and SparkCruise show that the benefits of computation reuse are not limited to a specific query engine. We believe that computation reuse should be a fundamental principle, like data locality and fault tolerance, when designing big data processing systems.

5.6 Other Applications of Reuse

The CloudViews mechanism of producing artifacts as part of query execution is useful in many other related applications:

- **Checkpointing.** Computation reuse can be applied for automatic checkpoint and restart in large analytical queries. The idea is to select intermediate subexpressions in a job's query plan to materialize and reuse them in case the job is restarted after a failure. Job failures are common in production clusters, with a small fraction failing every day. There are many reasons for failures including but not limited to storage errors, lack of compute capacity, missing input files, and network timeouts [45]. These transient errors are especially problematic for long running jobs that run for hours and fail towards the end. Typically, the failed jobs are resubmitted after a short delay. However, these jobs execute from the start all over again, thus wasting valuable compute resources and also delaying the final results. CloudViews can be used in this scenario to recover quickly from failures. During the compilation phase, we use query history to find which operators are more likely to fail and add a checkpoint just before them [49]. Then, during the resubmission, CloudViews can load the last available checkpoint thereby avoiding re-computation. Going forward, it would be interesting to select views that increase cluster utilization by maximizing the reuse across queries and minimizing the recovery time at the same time.

- **Pipeline Optimization.** Enterprise data analytics consists of data pipelines where analytical queries are interconnected by their outputs and inputs. Furthermore, the output of each producer query in the pipeline is typically consumed by multiple downstream queries. Unfortunately, the producers are not aware of the right data representations, or physical designs, required by their consumers. As a result, downstream queries need to prepare the data before they can run the actual processing. Therefore, it is crucial to leverage the data dependencies for creating the right physical designs tailored to the downstream queries. This can be done by producing the right physical design as part of query execution of producer job, i.e., a CloudViews that captures the required physical designs needed by downstream jobs.
- **Sampling.** Sampling is a powerful technique used for approximate query execution. Approximate query execution helps lower the latency and cost of running complex analytical queries on large datasets [29]. CloudViews style computation reuse can be applied for reducing the cost of approximate query execution even further. This can be achieved by sampling the views created by CloudViews. Sampled views will particularly help reduce query latency and cost in queries where substantial work happens after the sampler. Likewise, we could create statistics on the common subexpressions to provide insights to data scientists and analysts.
- **Bit-vector Filtering.** Bit-vector filters such as bitmap filters, Bloom filters and similar variants have been proposed by both industry and academia to perform semi-join reductions [15]. Semi-join reductions help filter rows which do not qualify join condition early-on in the query execution plan. Bit-vector filters have a low storage and compute overhead. They are commonly used in hash joins during query processing. CloudViews style computation reuse can be applied for generating bit-vectors during query execution as well. During query execution, a spool operator could be used for generating the bit-vector filter from right child of hash join and reuse it in subsequent queries.

6 OTHER RELATED WORK

Compute reuse is a hot topic in industry and we discuss some of the trends in other major companies below.

Snowflake is a cloud-based data warehouse company. The Snowflake design caches results of every query executed in the past 24 hours [41]. Users can then postprocess the cached result for further analysis. However, caching only the final results has limited applicability. CloudViews on the other hand reuses computation at any point in the query plan and is thus a superset of result set caching performed by Snowflake.

Google BigQuery is an interactive big data system that supports different kinds of caching and computation reuse. In particular, similar to Snowflake, it also supports caching arbitrary query results [18]. However, same as Snowflake, users have to manually rewrite their queries against the cached query results. BigQuery also supports materialized views that are maintained and support automatic query rewriting [17]. However, it only supports queries with aggregate function to simply the query rewriting problem.

Amazon Redshift is cloud data warehouse product from Amazon. Redshift enables materialized views to be automatically refreshed with automatic query rewrites [3]. However, the materialized views are still created manually and unlike online materialization in CloudViews, the responsibility of creating materialized views in Redshift lies with the user.

Alibaba [2] allows users to create materialized views. Once created, the materialized views could be used in queries. However, Alibaba's data warehouse engine requires manual re-writes to reuse materialized views. Again, unlike the automatic materialization and reuse in CloudViews.

Oracle deploys an algorithm called extended covering subexpressions (ECSE) to select materialized views for reuse [1]. It performs pairwise selection of 'join sets' and deploys other heuristics to reduce the search space to polynomial size. The authors test ECSE against a two-node configuration against a small number of queries, and intend to deploy the technique to their cloud in the future. By contrast, CloudViews is not restricted to this smaller class of candidate materialized views and has been continuously executing at extreme scale over hundreds of thousands of queries per day.

Finally, there are several other works that apply computation reuse in other settings. Yuan et al. [44] apply computation reuse on query workloads from Alibaba. Inspired by BigSubs algorithm [24] of CloudViews, they formulate the subexpression selection problem as ILP (Integer Linear Programming) and use deep reinforcement learning to solve the ILP. AStream [30] is a shared computation reuse framework for streaming queries. AStream can dynamically adapt to changing streaming query workloads without affecting the query execution topology. Computation reuse was also applied on Machine Learning workloads by Helix [43]. Helix finds common intermediate computation between iterations and automatically materializes some of them for future iterations.

7 CONCLUSION

Large-scale data processing infrastructures are key to data-driven decisions in modern enterprises. Unfortunately, the scale and complexity of these infrastructures could also make them unwieldy and highly inefficient. In this paper, we describe how large scale data preparation, also referred to as data cooking, in the Cosmos big data infrastructure at Microsoft often leads to redundant computations across different data pipelines and how computation reuse naturally augments the data cooking processing by fine-tuning the cooked datasets with more shareable ones. We show the impact of CloudViews, an automatic computation reuse infrastructure that we had build in the SCOPE query engine, over large production workloads and discuss many of the operational challenges that we faced. CloudViews has not only helped improve operational efficiency (37% less aggregated processing time) and customer experience (34% less aggregated latency), but it has also opened up new avenues and reusable infrastructure for a broad range of feedback-driven workload optimizations — the stepping stones towards a self-tuning intelligent cloud.

REFERENCES

[1] Rafi Ahmed, Randall G. Bello, Andrew Witkowski, and Praveen Kumar. 2020. Automated Generation of Materialized Views in Oracle. *VLDB* 13, 12 (2020), 3046–3058.

[2] alibaba [n.d.]. Create a materialized view. https://www.alibabacloud.com/help/doc-detail/116486.htm.

[3] amazon-redshift [n.d.]. Amazon Redshift announces automatic refresh and query rewrite for materialized views. https://aws.amazon.com/about-aws/whats-new/2020/11/amazon-redshift-announces-automatic-refresh-and-query-rewrite-for-materialized-views/.

[4] apache-hive [n.d.]. Apache Hive. https://hive.apache.org/.

[5] apache-spark [n.d.]. Apache Spark. https://spark.apache.org/.

[6] aws-athena [n.d.]. AWS Athena. https://aws.amazon.com/athena/.

[7] Malay Bag, Alekh Jindal, and Hiren Patel. 2020. Towards Plan-aware Resource Allocation in Serverless Query Processing. In *12th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 20)*.

[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*. 285–300.

[9] 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.

[10] Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. 2018. Axiomatic Foundations and Algorithms for Deciding Semantic Equivalences of SQL Queries. *VLDB* 11, 11 (2018), 1482–1495.

[11] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL. In *CIDR*.

[12] Andrew Chung, Subru Krishnan, Konstantinos Karanasos, Carlo Curino, and Gregory R. Ganger. 2020. Unearthing inter-job dependencies for better cluster scheduling. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 1205–1223.

[13] datalake [n.d.]. Azure Data Lake. https://azure.microsoft.com/en-us/solutions/data-lake/.

[14] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation (OSDI'04)*. USENIX Association, USA, 10.

[15] Bailu Ding, Surajit Chaudhuri, and Vivek Narasayya. 2020. Bitvector-Aware Query Optimization for Decision Support Queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20)*. 2011–2026.

[16] gdpr-art17 [n.d.]. Art. 17 GDPR. Right to erasure ('right to be forgotten'). https://gdpr.eu/article-17-right-to-be-forgotten/.

[17] Google. 2020. *BigQuery Materialized View*. Retrieved November 18, 2020 from https://cloud.google.com/bigquery/docs/materialized-views-intro

[18] Google. 2020. *BigQuery Result Caching*. Retrieved November 18, 2020 from https://cloud.google.com/bigquery/docs/cached-results

[19] google-bigquery [n.d.]. Google BigQuery. https://cloud.google.com/bigquery.

[20] 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.

[21] Alon Y. Halevy. 2001. Answering queries using views: A survey. *VLDB J.* 10, 4 (2001), 270–294.

[22] Rojeh Hayek and Oded Shmueli. 2020. Improved Cardinality Estimation by Learning Queries Containment Rates. In *EDBT*, Angela Bonifati, Yongluan Zhou, Marcos Antonio Vaz Salles, Alexander Böhm, Dan Olteanu, George H. L. Fletcher, Arijit Khan, and Bin Yang (Eds.). 157–168.

[23] Alekh Jindal. 2020. Applied Research Lessons from CloudViews Project. *SIGMOD Record (to appear)* (2020).

[24] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. 2018. Thou Shall Not Recompute: Selecting Subexpressions to Materialize at Datacenter Scale. In *VLDB*.

[25] Alekh Jindal, Hiren Patel, Abhishek Roy, Shi Qiao, Jarod Yin, Rathijit Sen, and Subru Krishnan. 2019. Peregrine: Workload Optimization for Cloud Query Engines. In *SoCC*.

[26] Alekh Jindal, Shi Qiao, Hiren Patel, Zhicheng Yin, Jieming Di, Malay Bag, Marc Friedman, Yifung Lin, Konstantinos Karanasos, and Sriram Rao. 2018. Computation Reuse in Analytics Job Service at Microsoft. In *SIGMOD*.

[27] Alekh Jindal, Shi Qiao, Rathijit Sen, and Hiren Patel. 2020. Microlearner: A fine-grained Learning Optimizer for Big Data Workloads at Microsoft. *Under Submission* (2020).

[28] 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 *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, USA, 117–134.

[29] Srikanth Kandula, Anil Shanbhag, Aleksandar Vitorovic, Matthaios Olma, Robert Grandl, Surajit Chaudhuri, and Bolin Ding. 2016. Quickr: Lazily Approximating Complex Ad-Hoc Queries in Big Data Clusters. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2016)* (proceedings of the acm sigmod international conference on management of data (sigmod 2016) ed.). ACM - Association for Computing Machinery. https://www.microsoft.com/en-us/research/publication/quickr-lazily-approximating-complex-ad-hoc-queries-in-big-data-clusters/

[30] Jeyhun Karimov, Tilmann Rabl, and Volker Markl. 2019. AStream: Ad-Hoc Shared Stream Processing. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 607–622. https://doi.org/10.1145/3299869.3319884

[31] Microsoft. 2020. *Azure SparkCruise Samples*. Retrieved November 18, 2020 from https://github.com/Azure-Samples/azure-sparkcruise-samples

[32] Microsoft. 2020. *SparkCruise on Azure HDInsight*. Retrieved November 18, 2020 from https://docs.microsoft.com/en-us/azure/hdinsight/spark/spark-cruise

[33] Nick. 2014. *Microsoft uses real time telemetry 'Asimov' to build, test and update Windows 9*. Retrieved November 13, 2020 from https://mywindowshub.com/microsoft-uses-real-time-telemetry-asimov-build-test-update-windows-9/

[34] Presto. 2020. *Presto*. Retrieved November 18, 2020 from https://prestodb.io/

[35] Raghu Ramakrishnan, Baskar Sridharan, John R Douceur, Pavan Kasturi, Balaji Krishnamachari-Sampath, Karthick Krishnamoorthy, Peng Li, Mitica Manu, Spiro Michaylov, Rogério Ramos, et al. 2017. Azure Data Lake Store: a hyperscale distributed file service for Big Data analytics. In *SIGMOD*. 51–63.

[36] Abhishek Roy, Alekh Jindal, Hiren Patel, Ashit Gosalia, Subru Krishnan, and Carlo Curino. 2019. SparkCruise: Handsfree Computation Reuse in Spark. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 1850–1853. https://doi.org/10.14778/3352063.3352082

[37] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. 2000. Efficient and Extensible Algorithms for Multi Query Optimization. In *SIGMOD*. 249–260.

[38] Guillem Rull, Philip A. Bernstein, Ivo Garcia dos Santos, Yannis Katsis, Sergey Melnik, and Ernest Teniente. 2013. Query containment in entity SQL. In *SIGMOD*, Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias (Eds.). 1169–1172.

[39] Rathijit Sen, Alekh Jindal, Hiren Patel, and Shi Qiao. 2020. AutoToken: Predicting Peak Parallelism for Big Data Analytics at Microsoft. *PVLDB* 13, 12 (2020), 3326–3339. https://doi.org/10.14778/3415478.3415554

[40] Tarique Siddiqui, Alekh Jindal, Shi Qiao, Hiren Patel, and Wangchao Le. 2020. Cost Models for Big Data Query Processing: Learning, Retrofitting, and Our Findings. In *SIGMOD*. 99–113. https://doi.org/10.1145/3318464.3380584

[41] Snowflake [n.d.]. Caching in Snowflake Data Warehouse. https://community.snowflake.com/s/article/Caching-in-Snowflake-Data-Warehouse.

[42] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, and Sriram Rao. 2018. Towards a Learning Optimizer for Shared Clouds. *PVLDB* 12, 3 (2018), 210–222.

[43] Doris Xin, Stephen Macke, Litian Ma, Jialin Liu, Shuchen Song, and Aditya Parameswaran. 2018. HELIX: Holistic Optimization for Accelerating Iterative Machine Learning. *Proc. VLDB Endow.* 12, 4 (Dec. 2018), 446–460. https://doi.org/10.14778/3297753.3297763

[44] H. Yuan, G. Li, L. Feng, J. Sun, and Y. Han. 2020. Automatic View Generation with Deep Learning and Reinforcement Learning. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1501–1512. https://doi.org/10.1109/ICDE48307.2020.00133

[45] Matei Zaharia. 2019. Lessons from Large-Scale Software as a Service at Databricks. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 101. https://doi.org/10.1145/3357223.3365870

[46] 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.

[47] Qi Zhou, Joy Arulraj, Shamkant Navathe, William Harris, and Jinpeng Wu. 2020. SPES: A Two-Stage Query Equivalence Verifier.

[48] Qi Zhou, Joy Arulraj, Shamkant B. Navathe, William Harris, and Dong Xu. 2019. Automated Verification of Query Equivalence Using Satisfiability Modulo Theories. *VLDB* 12, 11 (2019), 1276–1288.

[49] Yiwen Zhu, Alekh Jindal, Malay Bag, and Hiren Patel. 2020. Phoebe: A Data-Driven Checkpoint Optimizer. *Under Submission* (2020).