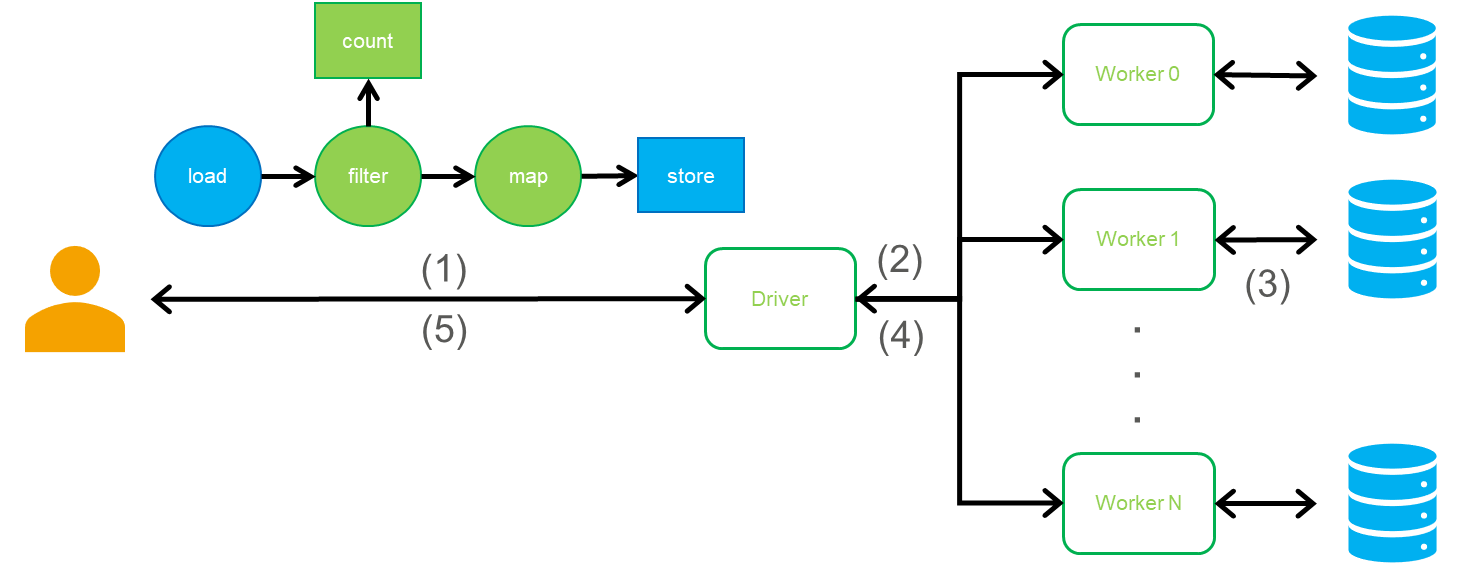
Semantic Cache – Design Documentation

# Introduction

Distributed compute engines typically accept jobs from users in the form of Distributed Acyclic Graphs (DAGs). A driver module is responsible for dividing the jobs in multiple tasks. Then, the driver sends the tasks to worker modules for execution. The whole process can be summarized at ***Figure 1***.



***Figure 1****: Distributed Compute Engine. (1) User sends job request to Driver in the form of a DAG. (2) Driver creates an optimized execution DAG from input DAG and divides it to multiple tasks. Each task can be executed in a single Worker. Then, tasks are sent to Workers for execution in specific order. (3) Workers load initial data from a storage, process it, and store the output back to storage if necessary. (4) Workers send acknowledgement or result related to the execution of the corresponding task back to Driver. (5) Driver sends acknowledgement or result related to the whole execution of the job back to User when all tasks are finished.*

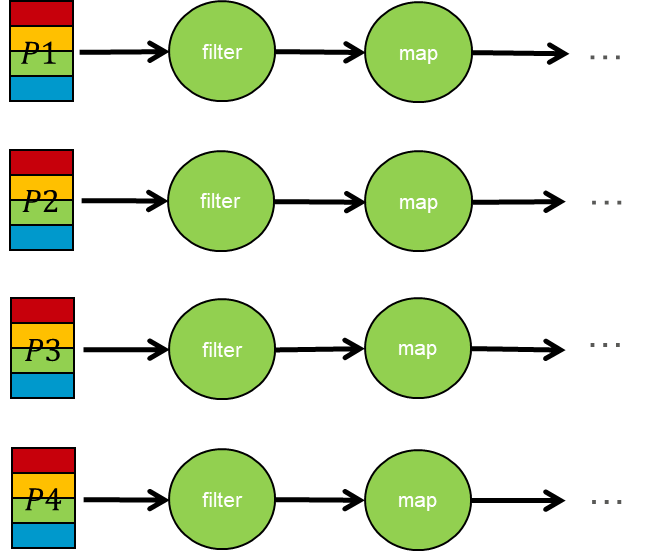
Data analysis jobs typically utilize large data sets with millions/billions of records. However, the results may depend on a very small fraction of the original records. For example, a dataset formed of records taken from sensors deployed in houses over a large area at a frequent rate, can be extremely huge. However, a particular job’s result might be impacted only by a small portion of these records. A typical example of such a job would be the following:

This job loads data from a group of CSV files. Then it filters (associated operation color-coded with green) out all records that do not have a hot temperature (. Notice, that only records in areas with warm climate during warm periods would affect the final count.

Distributed compute engines for big data workloads (e.g. Spark), adopt a naïve approach for tackling workloads like the previous one. The user provides a job in the form of a Directed Acyclic Graph (a simple example is the one we gave above, ). Then, it will create an optimized execution plan from the user-provided DAG based on the available resources and workload characteristics. Using the above example again, an execution plan like the one shown in ***Figure 2*** can be constructed. Let us assume that the compute engine divides the dataset to four different partitions (). A portion of the data inside the partitions has records with hot temperatures (annotated with red color), and the rest of the data belongs to distinct different temperature ranges (warm is annotated with yellow, cool with green and cold with blue color). For each partition, a process loads the corresponding data, filters in only the red portion, and continues the ensuing operations. Although only small portion of the original data, affects the result, this strategy leads to:

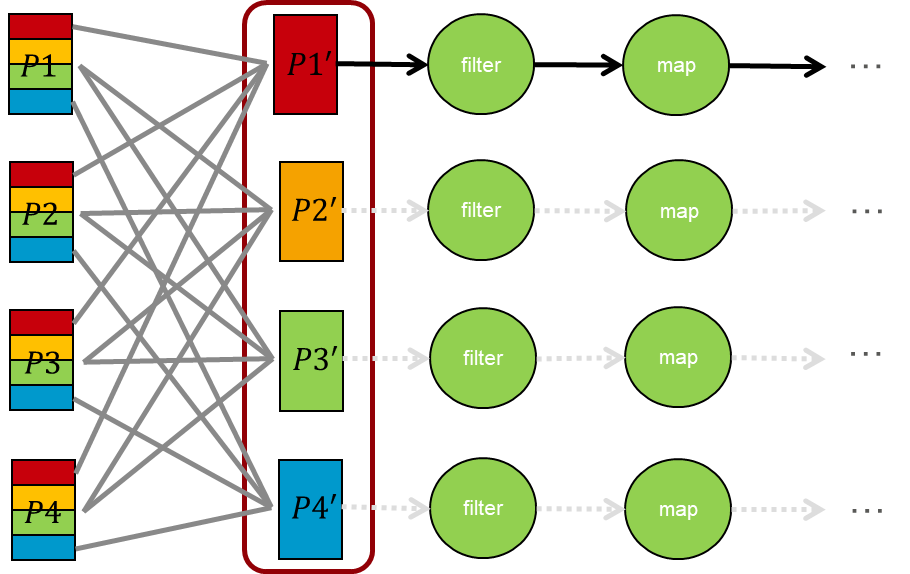
* Storage I/O overhead analogous to the size of the initial dataset.
* Network I/O overhead between the compute (where the data is processed) and storage side (where the data is stored typically in HDD/SSD) analogous to the size of the original dataset.
* Usage of computation and memory resources (DRAM) for the compute engine cluster analogous to the size of the initial dataset.
* Large execution times.

There are a few different techniques trying to solve this issue, that have been proposed in the literature. The common theme amongst them is that they sacrifice some additional memory (DRAM, primary) or disk (HDD/SSD, secondary) space for better overall performance. A cache can be used to store this extra data, since it does not need to be replicated or persisted for fault-tolerance. If the contents are not found in the cache, compute engines can revert to the baseline strategy (inefficient but correct).



***Figure 2****: Baseline plan. This DAG is constructed after the submission of the job we use as an example. This job executes an operation that filters in only records with red values. Finally, the job continues with the ensuing operations (e.g. map).*

*Adaptive partitioning* is the process in which data is dynamically restructured based on the workload characteristics. For example, if there are a lot of jobs which have a filter operation involving the temperature attribute value (like the one in the example), it might be beneficial to sort and split the data between partitions based on the temperature attribute value. Such an approach can be seen in ***Figure 3***. The initial dataset is going to be resorted and the new partitions () correspond to a unique color (distinct temperature ranges). These new partitions can be stored using extra memory or storage space depending on its size and the H/W. When a job like the one we use for our example is executed, the compute engine is going to load only partition, which is the only one that contains red records with hot temperature values (*partition-pruning*). As it can be seen in the figure, a smaller portion of the data is loaded from storage and transferred through the network (one fourth in our example), the compute engine utilizes fewer tasks (one instead of four in our example) with fewer total compute and memory resources, and the execution in most cases is going to be considerably faster. However, the extra overhead from repartitioning data is considerable in terms of extra space and computation.

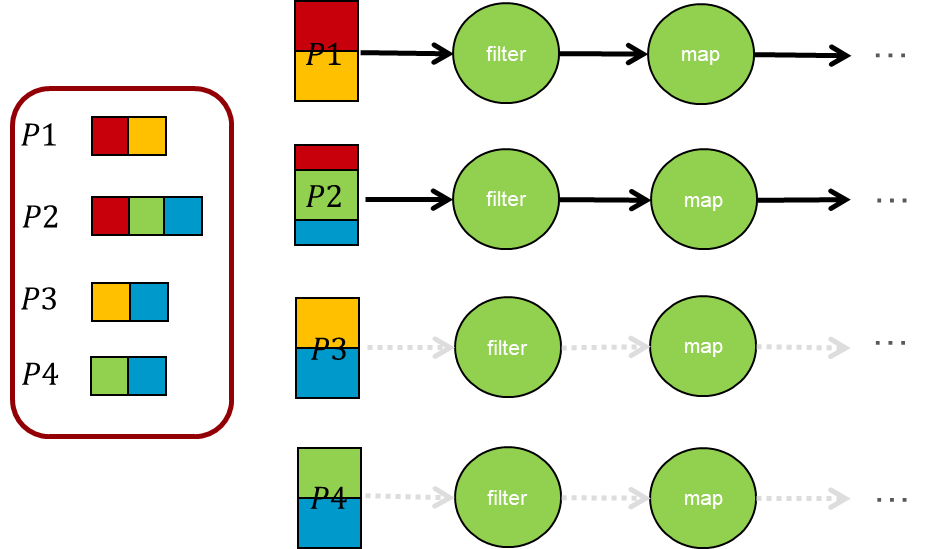


***Figure 3****: Adaptive Partitioning. Repartition original data in terms of temperature and store it again. When a job with a filter operation involving the temperature attribute, loads only partitions that contain at least one filtered-in record (partition-pruning), and continue with the ensuing operations. Notice that the filter operation is still executed on the compute side since in the loaded partitions there can still be records that are eventually filtered out.*

Another technique frequently used to solve the described problem is Data-Skipping. The storage side (or the compute side) maintains information about secondary attribute values (not main partition attributes) per partition. Common data structures used are the following:

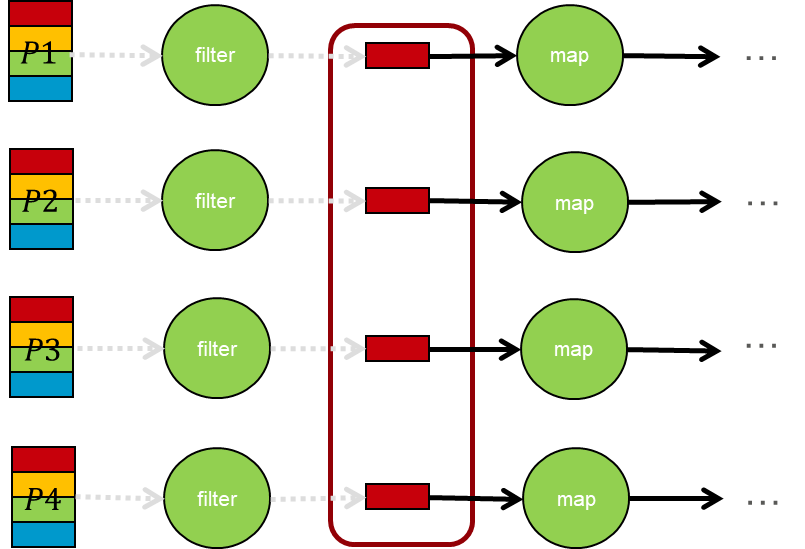
* Min/Max value for each partition of a numeric attribute.
* Lists with all the values inside a single partition of a category attribute.
* Bloom filters for values inside a single partition of an attribute.

A partition can be pruned if Data-Skipping information ensures that eliminating the partition does not lead to a different result. In our example at ***Figure 4***, let us assume that all the red records are in partitions. Then using a list for data-skipping metadata, a compute engine can safely detect that there are no red values in partitions and thus, it can prune them.



***Figure 4****: Data Skipping. A list with all the possible values of an attribute can be maintained. Red value is only present on P1 and P2 partitions. When an operation that filters in only red values is executed then P3 and P4 partitions can be pruned since they do not contain a single record with a red value.*

Another technique used to solve the described problem is Intermediate Data Caching. The compute engine can store the result of an operation and then if the same or similar job is executed, these results can be reused. In our example at ***Figure 5***, when the first job with the specified filter operation (filter-in red records) is executed, it caches the results of the filter in memory or storage. When another job with the same filter operation is executed, it can skip the filter operation and the initial loading from storage, and it can directly load cached data.



***Figure 5****: Intermediate Data Caching. After the filter is executed, the red portion of the data is cached in memory or storage. The rest of the job is then executed. When a subsequent job with the same filter operation (filter-in hot temperature records) is executed, it can directly load the cached data and do not even execute the filter operation.*

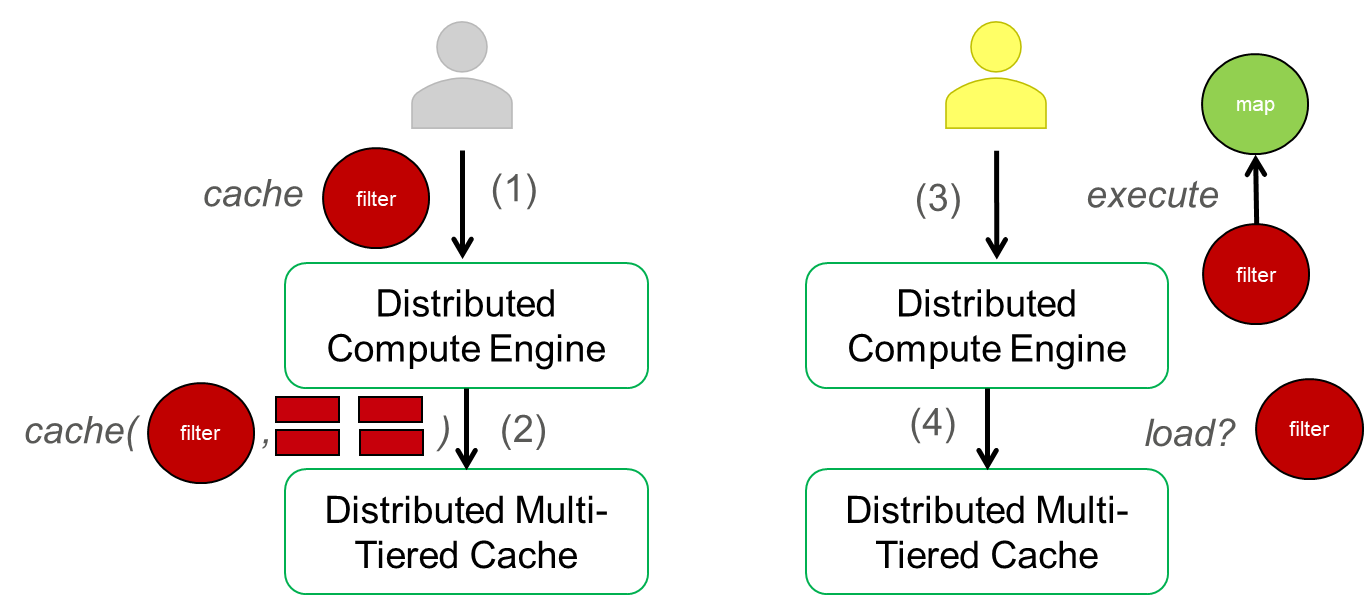
Semantic Cache offers the following functionality:

1. Caches all these different types of useful content.
2. Provides shared read access to content.
3. Semantic-aware capabilities (users do not have to know what the content is for, since optimization according to the current contents is automatic).

# Background

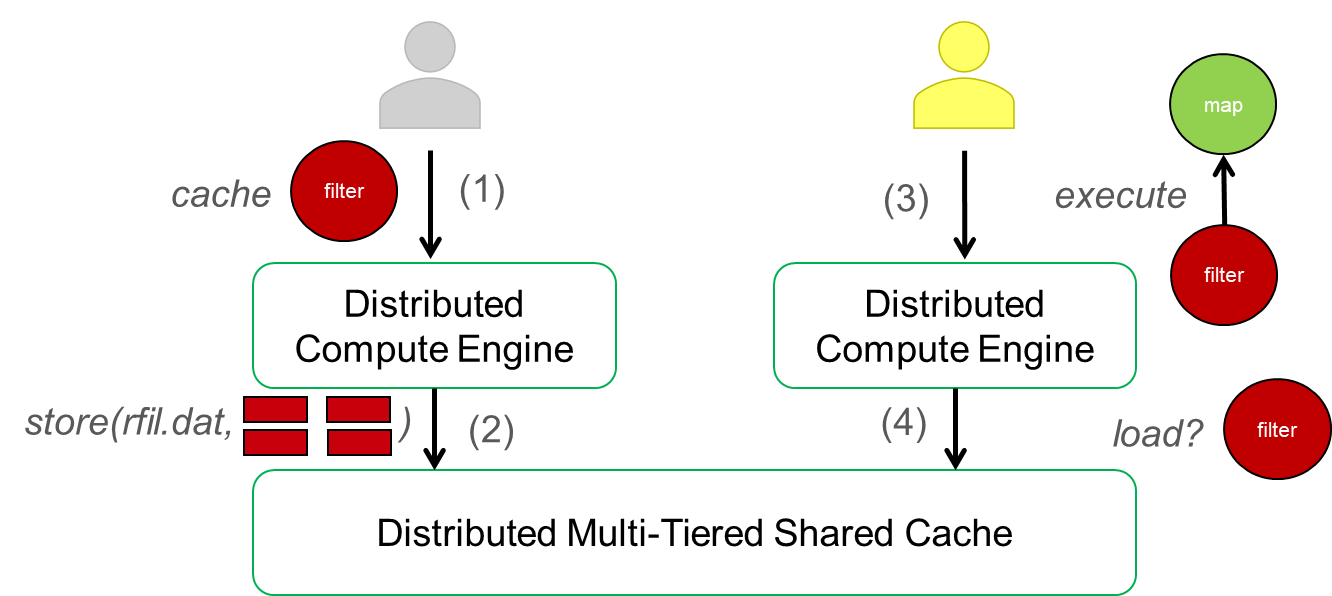
In this section two other options are described.

Spark cache offers a cache solution, but it has a few limitations. It is not shared between users (sessions). As a result, users cannot benefit from other user’s cache contents. ***Figure 6*** summarizes this approach.



***Figure 6****: Spark Cache.*

Alluxio cache is another solution, but the cache itself does not provide semantic information about the content, so the content cannot be utilized unless the users have knowledge about its origins (initially created by them). ***Figure 7*** summarizes the approach taken by Alluxio.



***Figure 7****: Alluxio Cache.*

# Semantic Cache

## Overview

Semantic Cache is a centralized service that maintains semantic information about the cached content, which is used for optimizing queries on top of source data. This metadata is maintained in memory (DRAM) at Semantic Cache. However, Semantic Cache must be paired with one or more distributed baseline stores, like Alluxio or Hadoop Distributed File System (or HDFS) since it does not have the ability to maintain content on its own.

Semantic Cache can partially recognize when the content should be utilized to optimize queries which make it a fitter cache candidate compared to Alluxio or other cache technologies that are not semantic-aware. For example, Alluxio can be used for a shared cache, but users cannot utilize the content written by other users unless they know what the content is about (through naming convention or some other external communication between users).

The content of Semantic Cache is supposed to be managed by a single entity (cache administrator or automatic management by Semantic Cache itself). However, the content can be utilized to optimize queries from multiple users. For management, we plan to support the following three different cases:

1. Manual, cache administrator is responsible for maintaining cache contents. Users should manually ask for cached content (Alluxio fits this option with the sole exception of cached content related to source data). Semantic Cache does not operate in this mode.
2. Manual cache management, cache administrator is responsible for maintaining the cache content (addition, evictions, etc.). However, users’ queries would be automatically adjusted from Semantic Cache to read from the cached contents.
3. Semi-automatic, cache administrator provides useful content candidates for addition to cache and Semantic Cache can automatically evict content based on a cache eviction policy (can be customized, LRU by default). Users’ queries would be automatically adjusted from Semantic Cache to read from the cached contents.
4. Fully automatic, Semantic Cache fully controls cache management based on a utility-based planner (can be customized, default method to be decided). Users’ queries would be automatically adjusted from Semantic Cache to read from the cached contents.

Only option 1 is provided for Semantic Cache at 1st quarter of 2021. The third option should be provided at 2nd quarter of 2021. The second option should be provided soon.

Semantic Cache contents can be anything useful that might:

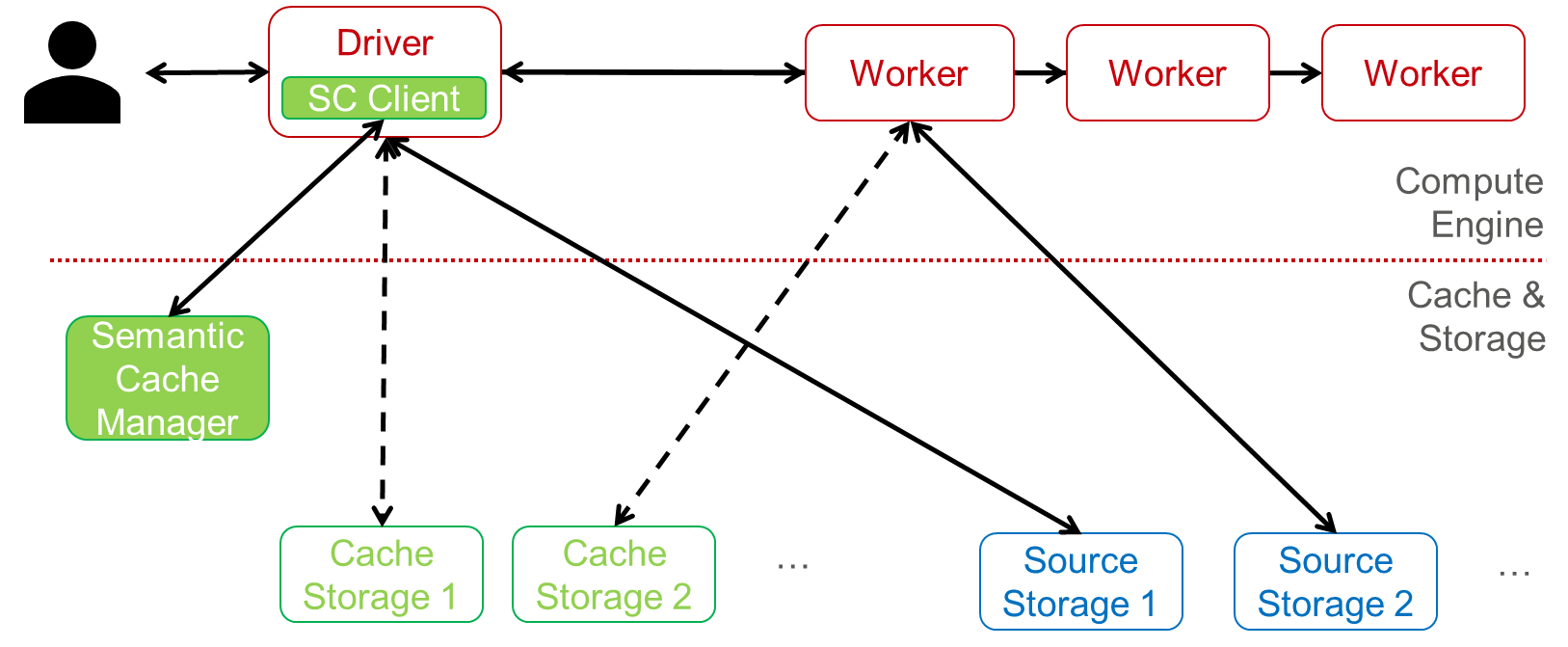
1. Decrease queries’ execution time.
2. Utilize less resources to execute queries.
3. Minimize the communication between the storage side (source data and cached content maintained in storage servers) and the compute side (where the processing happens).
4. Decrease the cost of the whole platform.

As of now, the contents can be:

* Partitioning of source data.
* FIle-Skipping Indices of source data.
* Cached Intermediate Data.

At ***Figure 8***, the overall approach is shown. There are multiple components:

1. Compute Engine Driver: A centralized node that is responsible for fully executing a user’s program by using Compute Engine’s resources.
2. Compute Engine Workers: The Driver utilizes them to execute small portions of the job (tasks) for parallel processing. Their number can scale up/down according to the job’s needs.
3. Semantic Cache Client (org.openinfralabs.caerus.cache.client): Client plugin for Driver that helps Driver communicate with Semantic Cache Manager and write or access cached content.
4. Semantic Cache Manager (org.openinfralabs.caerus.cache.manager): Centralized service that maintains semantic information and size of cached content.
5. Cache Storage: Storage used for cached content. As of now, only technologies that expose Hadoop Distributed File System (HDFS) API are supported. There are four different tiers that are supported (we assume disaggregated architectures with compute and storage cluster separated):
   1. COMPUTE\_MEMORY: Memory cache near the compute cluster.
   2. COMPUTE\_DISK: Disk cache near the compute cluster.
   3. STORAGE\_MEMORY: Memory cache near the storage cluster.
   4. STORAGE\_DISK: Disk cache near the storage cluster.
6. Source Storage: Storage where source data lies. Semantic Cache Manager has no access to it.



*Figure 8: Semantic Cache Architecture. Full lines denote direct communication between nodes, while dashed ones have to be indirect communications through Semantic Cache Manager. Semantic Cache Client is denoted as SC Client in the Figure.*

## Assumptions

Some assumptions are made about the modules and the overall platform. Correctness is ensured if and only if these assumptions hold.

1. The network between the Semantic Cache Client and Manager must be reliable (achieved via using [gRPC](https://scalapb.github.io/docs/grpc/)). This means that messages are not corrupted, partially delivered, or delivered multiple times. Essentially, a message is either delivered once or fails.
2. The Driver does not fail during the execution of a job.
3. The source data stored in Source Storage(s) is immutable (is not modified).
4. The cached content in Cache Storage is always available and cannot be corrupted or modified without Semantic Cache Manager approving the modification.
5. Semantic Cache Manager, although it is a centralized service is always available and cannot fail.
6. Access control works ideally, meaning that a user has access to the cached content if and only if the user also has access to all the source files from which the cached content derived from.

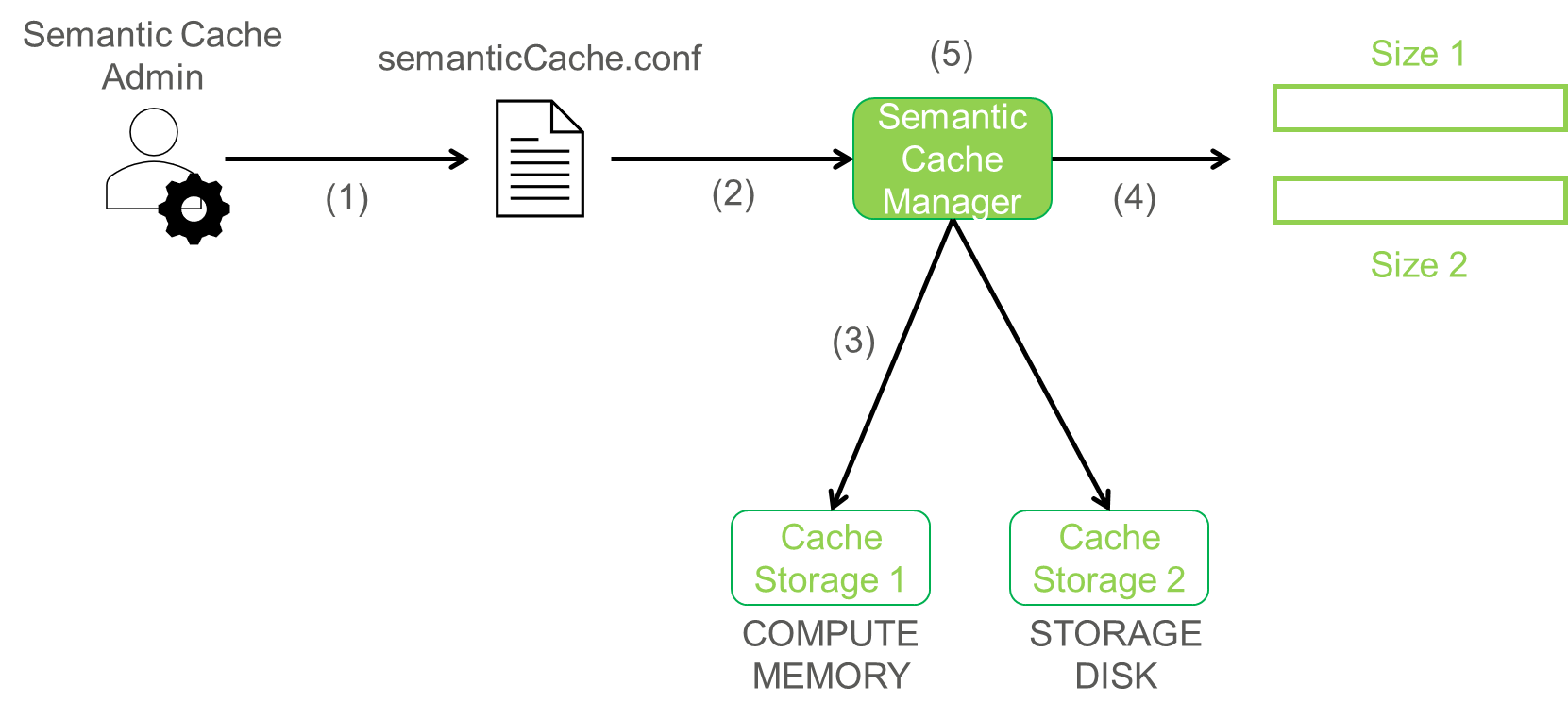
All assumptions are going to be lifted or weakened to more realistic assumptions by technologies which are going to be later introduced in this document.

## Initialization

### Semantic Cache Manager

***Figure 9*** shows how the Semantic Cache Manager is initialized. The steps are as follows:

1. Semantic Cache Admin writes the configuration, that includes URIs and a capacity (in bytes) for each cache tier that is defined by the user. Optionally, it might include the port in which the manager needs to listen to (default: 35001).
2. When initiated, Semantic Cache Manager pulls the configuration from the configuration file.
3. Semantic Cache Manager sets up size variables to prevent cache overflow for each successfully registered baseline cache.
4. Semantic Cache Manager starts listening for connections (start gRPC server) using the configured port value.



*Figure 9: Semantic Cache Manager Initialization*

### Semantic Cache Client

Semantic Cache Client should be initiated inside the application user’s code. After a session/application is specified for the compute engine, then the Semantic Cache Client can be initiated (session/application information, host and port of Semantic Cache Manager should be provided). Finally, Semantic Cache Client tries to connect to server (start gRPC client). If successful, Semantic Cache Client modifies the session/application to intervene in the query optimization process in the Driver.

## Semantic Cache Write (Only for Mode 1)

Semantic Cache supports three different types of write candidates:

1. Repartition (source, attribute)
2. DataSkippingIndex (source, attribute)
3. IntermediateDataCaching (plan)

The procedure is as follows:

1. Cache Admin writes a command to cache a candidate to Semantic Cache Client module. It also provides a name and ties it with the cached content. Finally, the Cache Admin provides the tier to store the content to.
2. Semantic Cache Client (potentially using the Driver’s size estimation) estimates the size needed for the specified content. It forwards it along with the other parameters to Semantic Cache Manager. If there is enough space, the Semantic Cache Manager reserves the estimated size.
3. Semantic Cache Manager sends the output path in the corresponding baseline cache (determined by the tier) to Semantic Cache Client. Notice that the contents are not updated yet. Thus, the content is not available yet for read.
4. Semantic Cache Client initiates the operation for the corresponding candidate through the Driver.
5. If successful, it notifies the Semantic Cache Manager to confirm that the cached content is stored (cached content becomes available for reads). The notification also includes the actual size of the cached content and Semantic Cache Manager readjusts the space required for the reservation. Otherwise, Semantic Cache Client notifies the Semantic Cache Manager to remove the reservation and release the space for this reservation.
6. Semantic Cache Client sends the corresponding reply (success or failure) to Semantic Cache Admin.

Notice that the estimated size needs to be an overestimation of the actual size for this to work. Otherwise, the behavior of the Semantic Cache Manager would be erroneous. ***Figure 10*** shows the write procedure.

## 

Figure 10: Semantic Cache successful write.

## Delete (Only for Mode 1)

The procedure is as follows:

1. Semantic Cache Admin sends delete request to Semantic Cache Client associated with the name of the cached content.
2. Semantic Cache Client propagates the request to Semantic Cache Manager.
3. If content with this name is present, the content is marked for deletion (unavailable for reads). Otherwise, the deletion fails. The Semantic Cache Manager sends a reply to Semantic Cache Client.
4. Semantic Cache Client replies to Semantic Cache Admin.
5. When there is not a current session/application currently utilizing this content, then the content is deleted, and size is updated in Semantic Cache Manager.

## Status (Only for Mode 1)

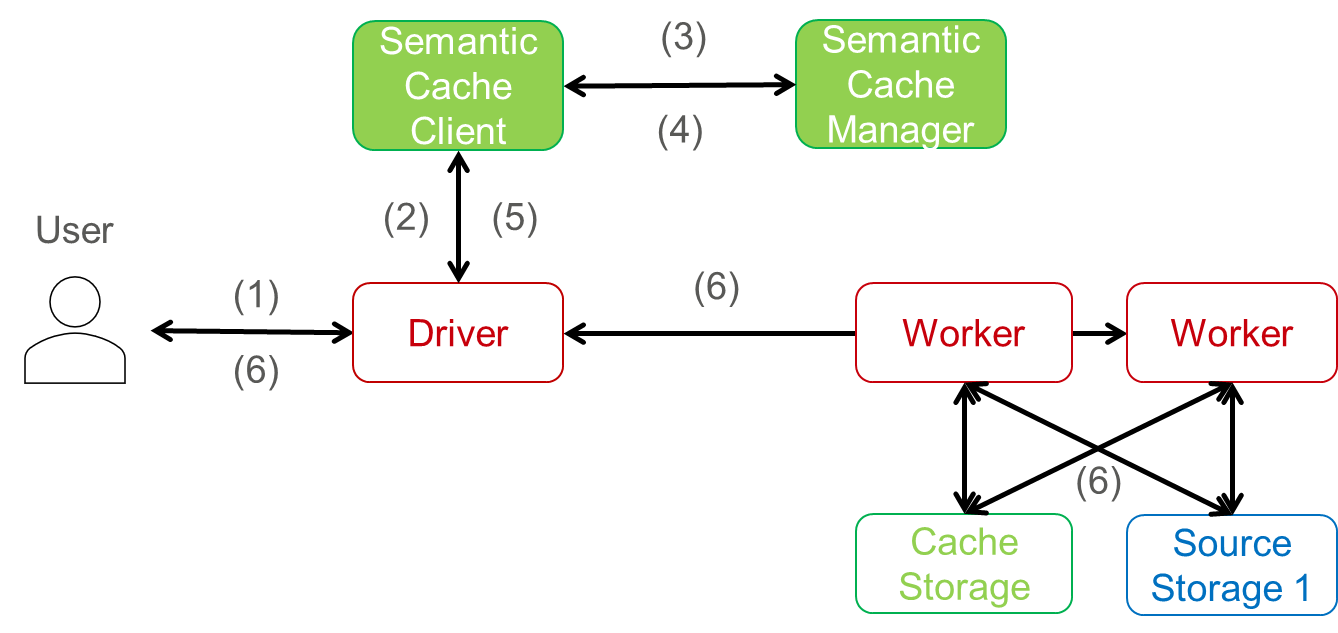
The Semantic Cache Admin could ask the Semantic Cache Manager through the Semantic Cache Client the status of the contents (reservations, currently utilized contents, contents marked for deletion) along with the respective remaining size of the baseline cache(s). This functionality is not supported as of now.

## Semantic Cache Read (Only for Mode 1)

The procedure is as follows:

1. User submits a job/query to the driver.
2. Driver optimizes the plan for the job/query using Semantic Cache Client’s optimization, that we added during initialization phase of Semantic Cache Client to the corresponding application/session.
3. Semantic Cache Client transforms the plan from its initial Compute Engine format to the native Semantic Cache format and sends the plan to the Semantic Cache Manager.
4. Semantic Cache Manager optimizes the transformed plan according to the contents, marks any relevant contents as used by the specified session/application and sends the optimized plan back to the Semantic Cache Client. Semantic Cache Manager should not delete the specified contents until the session/application is terminated.
5. Semantic Cache Client transforms back the plan to its initial Compute Engine format at and sends it back to Driver (thus finishing the Semantic Cache optimization step).
6. Driver continues with the optimization and execution of the query/job and sends back the results to user.

***Figure 11*** shows the read procedure.



*Figure 11: Semantic Cache Read*

Step 4 describes the optimization step**. Figure 12** shows the optimization process in detail. The optimization targets to prune the plan (tree structure) as much as possible. Provided that the lookup for related content is optimal (or fixed), then this process outputs the smallest plan (tree with fewer nodes) that returns the same result as the original. Initially, the shuffle and filter attributes should be void.

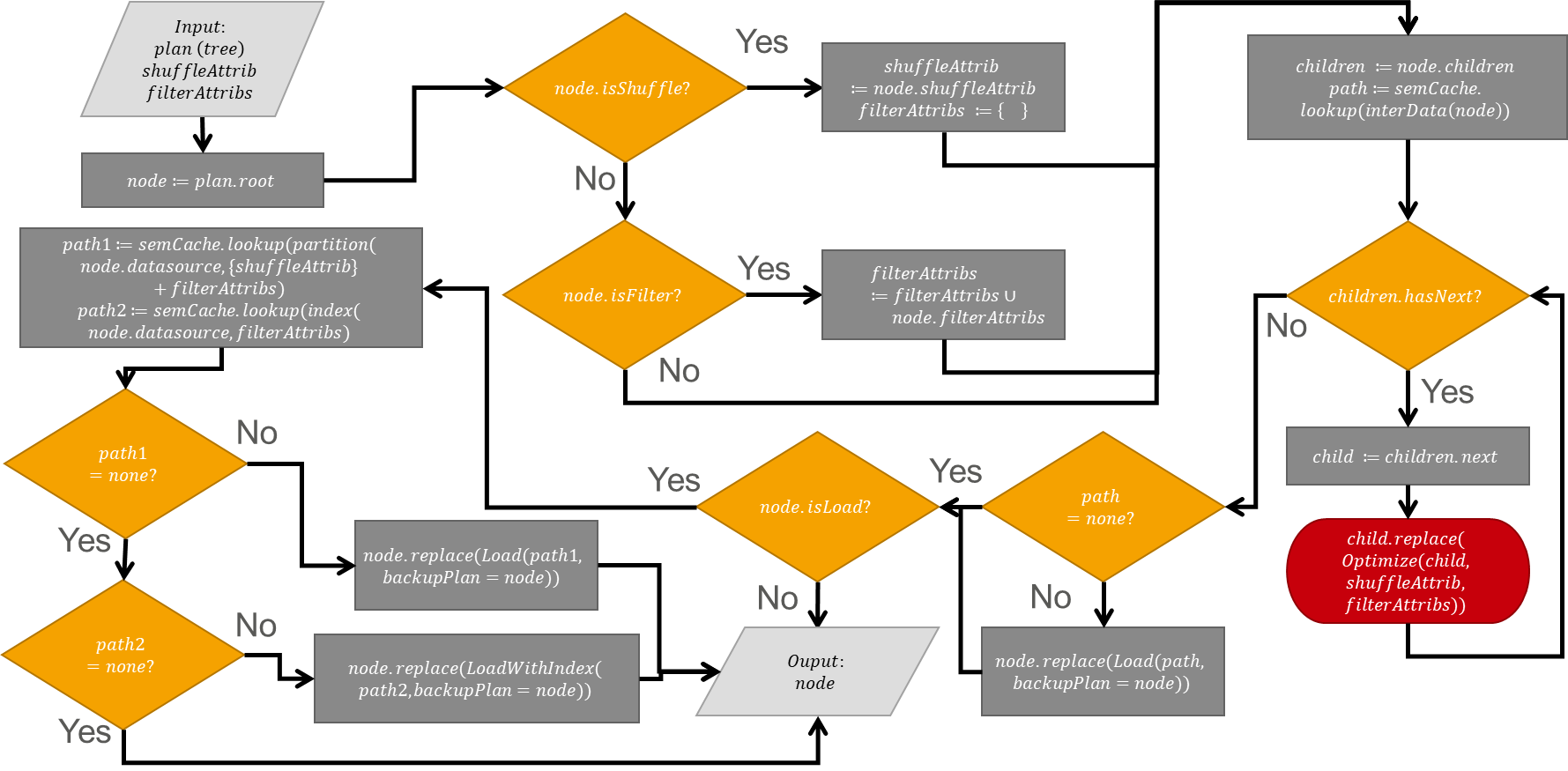


Figure 12: Optimization process through a flow diagram.

## Semantic Cache Planning (Only for Mode 3)

When Semantic Cache is on Mode 3, Semantic Cache Client and Manager work slightly differently than before. The procedure is as follows:

1. User submits a job/query to the driver.
2. Driver optimizes the plan for the job/query using Semantic Cache Client’s optimization, that we added during initialization phase of Semantic Cache Client to the corresponding application/session.
3. Semantic Cache Client transforms the plan from its initial Compute Engine representation (Native Plan) to the Semantic Cache representation (Caerus Plan).
4. Candidate Selector takes the Caerus Plan and picks write candidates for the cache (e.g. repartitioning, file-skipping indexing, caching).
5. Candidates are passed to Size Estimator. For each candidate, Size Estimator estimates the size that the candidate occupies in the cache (write size) and potential size that is loaded when the specific candidate is utilized.
6. Both the Caerus Plan and the Candidates are serialized and are sent to Semantic Cache Manager over the network via RPC.
7. Semantic Cache Manager deserializes Caerus Plan and Candidates.
8. The Caerus Plan is passed on Predictor to aid it with future prediction. Predictor responds with Future Caerus Plans.
9. Semantic Cache Manager utilizes the planner (using existing contents in cache, current plan, write candidates, and future predicted plans) and gets an optimized plan which potentially includes writes and reads from baseline cache(s).
10. Semantic Cache Manager serializes the optimized plan and sends back the reply to Semantic Cache Client.
11. Semantic Cache Client deserializes the optimized plan, transforms back the plan to its initial representation, serializes it and sends it back to Driver (thus finishing the Semantic Cache optimization step).
12. Driver continues with the optimization and execution of the query/job and sends back the results to user.

The figures below summarize this approach.

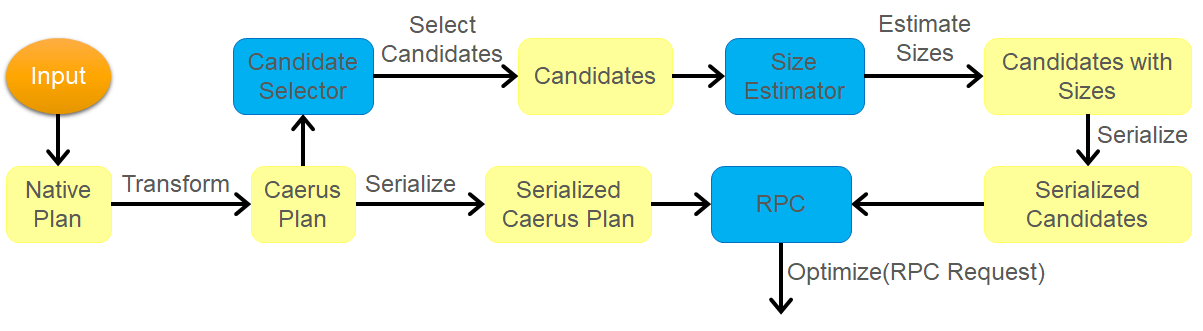


Figure 13: Semantic Cache Client protocol for mode 3 of Semantic Cache.

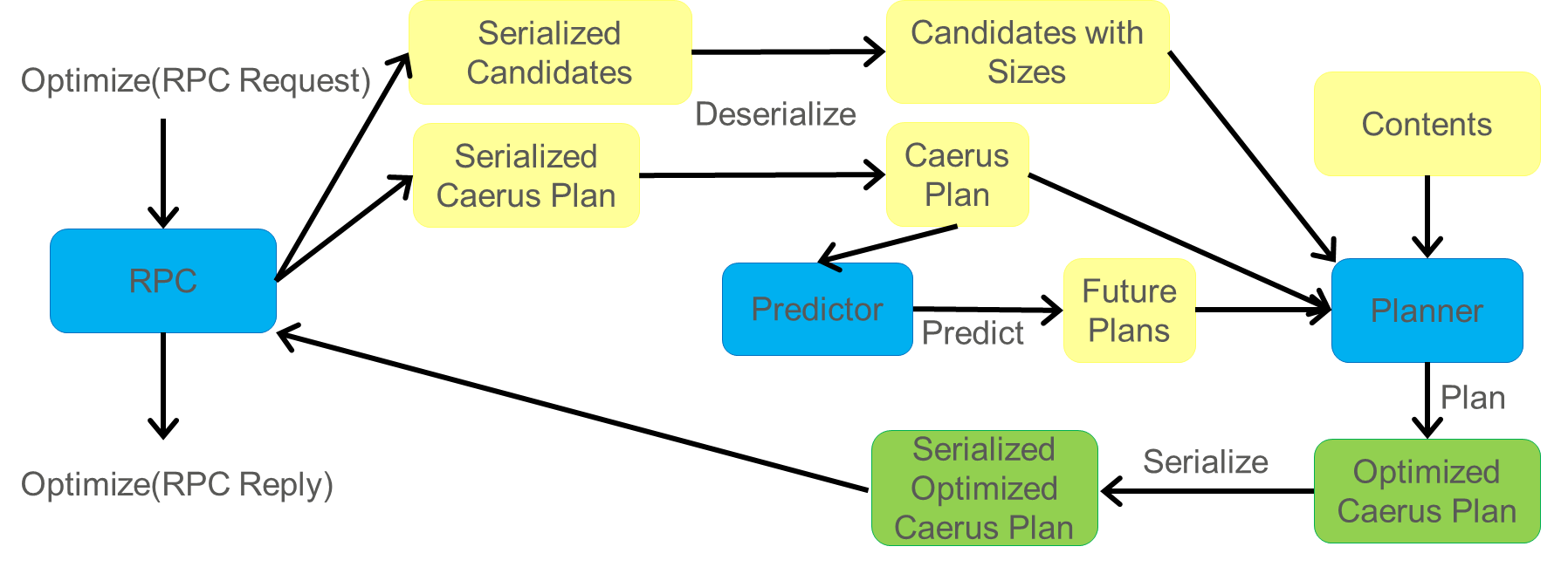


Figure 14: Semantic Cache Manager Protocol for mode 3 of Semantic Cache.

Mode 3 supports only one tier as of now (STORAGE\_DISK).

### Candidate Selector

There is only one implementation of Candidate Selector so far, Basic Candidate Selector. Basic Candidate Selector’s implementation is based on the below principles:

1. Propose candidates that are going to be used for optimizing the current plan.
2. Propose candidates that do not significantly increase cost for the current plan.

### Example for Basic Candidate Selector

Plan:

Candidates (output) for plan:

,

),

Notice that the repartitioning is not included in the candidates because it violates the second principle (adds extra shuffle for writing to the cache). It is clear that all these candidates, if they were already in the cache, would optimize the current plan (maintain the first principle).

### Size Estimator

The Size Estimator is used for the following functions:

* Predict write size of write candidates.
  + Reserve appropriate space in cache. Useful in all modes.
  + Determine the correct tier to use. Useful for future implementations of planner.
  + Approximate cost for creating the candidate. Useful for potential planner(s).
* Predict read size if and when candidate is utilized.
  + Approximates cost of reading when this candidate is utilized. Useful for potential planner(s).

As of now, Semantic Cache Manager only has a dummy Basic Size Estimator which makes sure to always reserve more space than needed (overestimate write size of candidates) to ensure correctness and bases the read predictions on completely arbitrary numbers based on the initial data size that these candidates are based on. Thus, Basic Size Estimator is correct but inefficient.

### Predictor

Predictor is responsible for making a prediction for the future. Every predictor can predict all the plans up to a specific customized window size. Semantic Cache Manager supports two types of predictors. An Oracle one which is configured with all the plans in the future (with the same sequence that they are executed) and with a Reverse Order one which reverses all the past plans that Semantic Cache Manager has encountered and returns them as future predictions.

### Planner

Semantic Cache Manager supports three different metrics for planners (Reference Count, Reference Distance, Storage I/O) as of now. Planner follows the below three phases when a new plan arrives:

1. Select a top candidate from the candidates of this plan.
   1. Find the original score for all plans (provided by Predictor) value with existing contents in Semantic Cache.
   2. For each candidate, find the score needed with existing contents plus the candidate for all plans (provided by Predictor). Subtract original score from candidate score to find the incremental candidate score.
   3. Eliminate all candidates with negative incremental score.
   4. If all of them are eliminated go to step 4.
   5. Pick the candidate with the best incremental score per cache byte (write size estimation) for inclusion in the cache.
   6. Add top candidate to contents.
2. Evict content if necessary. Repeat this process until contents fit in the cache.
   1. Find the original score for all plans (provided by Predictor) value with existing contents in Semantic Cache.
   2. For each content, find the score needed with existing contents minus the content for all plans (provided by Predictor).
   3. Subtract the content score from original score to find the incremental content score.
   4. Pick the content with the least incremental score per cache byte (write size estimation) for eviction from the cache.
   5. Remove content from contents.
   6. If content happens to be the top candidate picked in the previous phase, then cache keeps original contents. Go to step 4.
3. Planner schedules the necessary addition/write to the cache and any evictions that are needed.
4. Optimize plan using the same optimization process as in mode 1.

For scores, Semantic Cache supports the following:

1. Reference Count, which is the number of plans (provided by Predictor) that utilize specific candidate/content.
2. Reference Distance (inverse), which is the minimum index out of the plans that utilize specific candidate/content.
3. Storage I/O size (opposite), which is the total amount of data that is loaded/written from/to storage for a specific cache configuration.

An example for Storage I/O Planner is shown below.

Plan:

Contents:

Candidates:

,

),

Future Plans:

Benefits (I/O bytes) per byte written in the cache (for future plans):

,

New Contents:

,

)

If they do not fit in the cache () evict:

Benefits (I/O bytes) per byte written in the cache (for future plans):

Semantic Cache Manager in this example transforms the plan to write caching candidate and evict repartitioning content.

## Termination of the Semantic Cache Client

When Semantic Cache Client is terminated along with the Driver (end of application/session), a final message is sent to the Semantic Cache Manager. The purpose is two-fold:

1. In case of Semantic Cache Admin session, Semantic Cache Manager cancels all withstanding reservations.
2. In case of User session, Semantic Cache Manager, makes sure to remove any references to content (to decide if the content is safe for deletion).

## Heartbeats

This functionality addresses (eliminates) assumption number 2 (driver cannot fail). When a Semantic Cache Client connects to Semantic Cache Manager, a heartbeat is required in frequent intervals. If Semantic Cache Client misses three consecutive heartbeats, the Semantic Cache Client is considered terminated, and all subsequent actions take place (see Termination of the Semantic Cache Client). Any subsequent requests made by the specific client are denied by the Semantic Cache Manager (a new initialization would be required).

## Update/Invalidate Cache Mechanism

This functionality addresses (weakens) assumption number 3 (source data cannot be modified to source data cannot be overwritten). The Semantic Cache Client before sending an optimize request (see Semantic Cache Read) to the server should be able to find all the files involved along with their boundaries. There are three different cases:

1. If the Semantic Cache Manager finds exact contents related to the source files (including their boundaries) then this means that there was no update in the source content and the optimization can be fully integrated.
2. If the cached contents are related to a strict subset of the source files provided by the Semantic Cache Client (additional files and/or previous files were extended), then this means that the cached contents can be utilized to partially optimize the corresponding plan.
3. If the cached contents intersect with the source files but they are not a subset of the source contents (potential deletion of a file), then the optimization cannot apply. The above mechanism ensures correctness in case the source data cannot be overwritten.

The current implementation does make the optimization only in case of perfect match (1st and 3rd case work as intended).

Invalidation should naturally occur. Invalidated content would no longer be used for optimization and would be a primary candidate for eviction from Semantic Cache Admin (or later by the eviction policy of the Semantic Cache Manager itself).

An update interface is provided to the Semantic Cache Admin to update the cached content according to recent changes (or later automatically update content when deemed appropriate by the Semantic Cache Manager). This might not require invalidation and re-write of the content since this is going to be an expensive procedure. As of now, this functionality is not implemented.

## Backup Plan

This functionality addresses (weakens) assumption number 4 (from the cached content in Cache Storage is always available and cannot be corrupted or modified without Semantic Cache Manager approving the modification to the cached content cannot be modified without Semantic Cache Manager approving the modification). When a plan is optimized, if the job fails to finish, the Driver always uses a backup plan specified by Semantic Cache Manager. This creates a chain of plans where the first is popped from the chain is executed and if it fails, it repeats the procedure with the second one, etc. The last backup plan in the chain should always be the original plan. Thus, if cached content is not available or corrupted then the Driver would have the choice to fall back to the next backup plan in the chain. However, this mechanism comes with a significant cost. If the issue (failure) is caused by the source data and the cause is propagated through the cached content, then the Driver is going to execute all the plans in the chain (including the original one) as opposed to executing only the original one (without the Semantic Cache).

The optimization process shown at **Figure 12**,additionally to the main optimized plan, outputs the backup plan chain as well. As of now, this functionality is not implemented.

## Logging

This functionality addresses (weakens) assumption number 5 (Semantic Cache Manager, although it is a centralized service is always available and cannot fail to Semantic Cache Manager is always available). Semantic Cache Manager creates a persistent log where it logs all the hard state changes related to the cached contents. If it fails, a new Semantic Cache Manager reads the persistent log, creates a new compressed one (describing only its initial state), begins the initialization process and continues writing to the log according to the new changes. Notice that there is going to be an interval where Semantic Cache Manager is unavailable, between the failure of the first Semantic Cache Manager and the end of the initialization process of the second one. This mechanism might be used to update the configuration as well (add/remove tiers, change available size of a baseline cache, change port for Semantic Cache Manager service, etc.).

As of now, this functionality is not implemented.

## Replication

This functionality addresses (eliminates) assumption number 5 (Semantic Cache Manager, although it is a centralized service is always available and cannot fail) along with Logging. A replication scheme might be used to replicate the functionality of Semantic Cache Manager. This should increase the optimization step cost significantly but since this is a tiny fraction of the overall query/job execution time, it should not be noticeable to the users.

As of now, this functionality is not implemented.

## Security

Security mechanisms can be utilized to further improve this project.

Authentication tokens (or capabilities) can be used along with access control policies in Semantic Cache Manager to ensure that modifications happen only when approved by Semantic Cache Manager. This functionality addresses (eliminates) assumption number 4 (from the cached content in Cache Storage is always available and cannot be corrupted or modified without Semantic Cache Manager approving the modification to the cached content cannot be modified without Semantic Cache Manager approving the modification) along with Backup Plan. As of now this functionality is not implemented.

A second mechanism is to ensure that the access control policies for readers are not very restrictive. During the optimization step, a Semantic Cache Client should check whether the user has access to the initial files. If it has, then it continues with the optimization as planned. Otherwise, Semantic Cache Client returns the original plan as is without applying any optimizations.

# Clients

## Spark (org.openinfralabs.caerus.cache.client.spark)

A Semantic Cache Client has been implemented for Spark (Spark Client). The client is initialized using a SparkSession object, which attaches the client to the specific session and application, and the network address of the corresponding SemanticCacheManager. The Spark Client, during its setup, it connects to the SemanticCacheManager and then, registers the optimization as part of the regular spark optimization process (using spark.experimental.extraOptimizations variable). The optimization is positioned at the end of the logical optimization process in Spark making it a preferred position for applying the Semantic Cache optimization (filters, projections are pushed down in the plan, redundancies have been eliminated, etc.).