Anser: Adaptive Information Sharing Framework of AnalyticDB

Liang Lin
Alibaba Group
Hangzhou, China
yibo.ll@alibaba-inc.com

Yuhan Li
Alibaba Group
Hangzhou, China
lyh200442@alibaba-inc.com

Bin Wu
Alibaba Group
Hangzhou, China
binwu.wb@alibaba-inc.com

Huijun Mai
Alibaba Group
Hangzhou, China
huijun.mhj@alibaba-inc.com

Renjie Lou
Alibaba Group
Hangzhou, China
json.lrj@alibaba-inc.com

Jian Tan
Alibaba Group
Hangzhou, China
j.tan@alibaba-inc.com

Feifei Li
Alibaba Group
Hangzhou, China
lifeifei@alibaba-inc.com

ABSTRACT

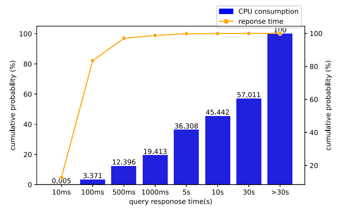
The surge in data analytics has fostered burgeoning demand for *AnalyticDB* on Alibaba Cloud, which has well served thousands of customers from various business sectors. The most notable feature is the diversity of the workloads it handles, including batch processing, real-time data analytics, and unstructured data analytics. To improve the overall performance for such diverse workloads, one of the major challenges is to optimize long-running complex queries without sacrificing the processing efficiency of short-running interactive queries. While existing methods attempt to utilize runtime dynamic statistics for adaptive query processing, they often focus on specific scenarios instead of providing a holistic solution.

To address this challenge, we propose a new framework called *Anser*, which enhances the design of traditional distributed data warehouses by embedding a new information sharing mechanism. This allows for the efficient management of the production and consumption of various dynamic information across the system. Building on top of *Anser*, we introduce a novel scheduling policy that optimizes both data and information exchanges within the physical plan, enabling the acceleration of complex analytical queries without sacrificing the performance of short-running interactive queries. We conduct comprehensive experiments over public and in-house workloads to demonstrate the effectiveness and efficiency of our proposed information sharing framework.

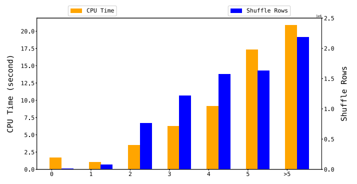
PVLDB Reference Format:
Liang Lin, Yuhan Li, Bin Wu, Huijun Mai, Renjie Lou, Jian Tan, and Feifei Li. Anser: Adaptive Information Sharing Framework of AnalyticDB. PVLDB, 16(12): 3636 - 3648, 2023.
doi:10.14778/3611540.3611553

1 INTRODUCTION

As modern organizations struggle with managing diverse workloads including batch processing, real-time data analytics, and unstructured data analytics, they face the challenge of maintaining optimal performance. To meet this challenge, there has been a trend towards system convergence, with many organizations transitioning towards uniform systems that can handle diverse workloads. One of the most widely adopted industry solutions is Spark [9, 49], a fast and flexible data processing engine that can handle diverse workloads. Similarly, Redshift [10, 26], a cloud-based data warehousing solution, offers automatic tuning capabilities to handle complex workloads more efficiently.



(a) The distribution of diverse workloads.



(b) Resource consumption of JOIN queries.

Figure 1: Statistics collected from *AnalyticDB* 's production workloads.

AnalyticDB [14, 46] is a high-performance data warehouse developed by Alibaba Cloud. It has been extensively adopted both internally for Alibaba Group's business operations and externally across a range of industries, such as e-commerce, finance, logistics, education, and entertainment. Within *AnalyticDB* , we have noticed a trend of increasing diversity in terms of query response times. As shown in Figure 1a, many simple and short queries, such as business-critical intelligence queries issued from dashboards, can be processed in milliseconds. These queries account for up to 80% of the customers' workloads in our production environments. To satisfy the quality of service requirements, it is essential to ensure that these interactive short queries have sufficient resources. Meanwhile, it is also common to have complex analytical queries that exceed hundreds of KB in size, involving aggregations, multi-way joins, and nested subqueries. Statistics show that long queries with response times (RT) more than 10 seconds account for over 10% of the

Proceedings of the VLDB Endowment, Vol. 16, No. 12 ISSN 2150-8097.
doi:10.14778/3611540.3611553

workloads, which yet consume more than 50% of the computation resources. To evaluate the resource consumption of the complex analytical queries, we collect related statistics of JOIN queries from *AnalyticDB* 's production workloads (as shown in Figure 1b), including the query CPU time and the number of shuffled rows. As the number of join operators increases, the required resources also grow dramatically.

As evidenced by the statistics above, optimization techniques for expensive batch computing tasks and ETL jobs play a vital role in improving the overall performance of modern data warehousing systems. As the number of concurrent queries increases, the competition for resources (*e.g.*, CPU, memory and network) between queries has become very serious. In some cases, long queries may exhaust resources in a database instance and subsequent short queries belonging to the same instance will not be processed. We summarize several key challenges that remain unsolved:

Challenge 1: Scenario customization for adaptive query processing. As workloads become increasingly diverse and statistics become less available, it has become clear that traditional "optimize-then-execute" strategies [24, 31, 41] are no longer sufficient. This realization has led to a broad range of studies in the field of adaptive query processing [20]. Many commercial databases have implemented various adaptive techniques, but the current approaches tend to build scenario-customized solutions for each technique, which can introduce unnecessary complexity into the system [7, 34, 44, 50]. For example, Spark's adaptive query execution [50] supports four features: mid-query re-optimization, dynamically coalescing shuffle partitions, dynamically switching join strategies, and dynamically optimizing skew joins. Each feature individually collects dynamic statistics and makes adjustments. A general-purpose information framework that can fit into these different scenarios would significantly reduce costs. By developing a framework that can share information across different adaptive techniques, it would be possible to eliminate redundant efforts and minimize the complexity of the system. Such a framework would enable data warehousing systems to optimize their performance without having to build scenario-customized solutions for each individual technique. Furthermore, the same statistics could be used by different cases (for example, all of the four features that Spark supports require shuffle file statistics), yet implementing each case individually deprives the opportunity for the same information to be used multiple times. In particular, when statistics collection is resource-consuming (such as with a bloom filter), sharing information among multiple cases could potentially significantly reduce costs.

Challenge 2: Effective and efficient management of dynamic statistics. To potentially identify and share common dynamic statistics, the information collection and utilization need to be decoupled from existing modules of query engine, and a holistic management of the information lifecycle is required to register, collect, store, disperse, and destroy the information. None of the previous studies have clearly defined the scope of the information that can be used in different adaptive techniques, nor have they framed a mechanism to manage the information lifecycle. In a production environment, information collection, transmission, and storage all lead to additional overhead. High-performance data warehouse requires such overhead to be diminished and separated from query execution process. Moreover, a carefully designed mechanism is necessary to limit the memory usage of the information storage that the dynamic statistics does not affect the overall system.

Challenge 3: Coordination with query scheduler. The statistics information can be holistically leveraged to optimize the adaptive adjustments. To this end, the scheduler naturally comes into the picture to orchestrate the execution orders of information consumer and producer. However, none of the previous studies have clearly defined a scheduler that is aware of the information dependencies. In batch processing systems, the transmission of adaptive statistics is mostly implemented as part of the execution process. Some approaches [12, 25, 32] add checkpoints in the execution plan that monitor statistics during execution and trigger re-optimization if necessary, while others rewrite the provider of information explicitly as a sub-expression in the query execution to provide adaptive statistics as part of the query execution process. Both approaches tie information transmission strictly with data processing, which means that the information consumer can only receive information from its upstream operators without considering the possibilities of receiving information sideways or discarding information with high production costs. Some real-time data analytics systems support passing information sideways, but mostly through tailored services. For example, Impala [5] implements a dynamic filter service to pass information sideways. Such services are customized for specific use cases and cannot be easily extended to others. Moreover, the scheduler is not aware of such information transmission. The consumer of the information either waits a static time period for statistics to arrive or only consumes available statistics before running. As execution plans become more complicated, useful statistics may not be consumed to provoke adaptive execution without cooperation with the scheduler. Therefore, a more sophisticated mechanism is required to manage the transmission and consumption of adaptive statistics, which takes into account the collaboration with the scheduler.

To this end, a novel information sharing framework, namely **a**daptive i**n**formation **s**haring fram**e**wo**r**k (*Anser*), is developed in *AnalyticDB* . Our major contributions are summarized as follows:

(1) The framework provides a uniform and effective interface for different modules to share various types and levels of information. At the operator level, *Anser* collects various types and levels of information, classifies according to their types and granularities, and passes to different modules across the query to tune for better performance during execution.
(2) The framework supports the automatic matching and transmission of the information between information producer and information consumer once the relationship is registered. It supports many-to-one and one-to-many information passing in a complex physical execution tree. The transmission is both low latency and efficient by the usage of information merging and push-based communication model.
(3) In conjunction with the framework, we design an information-aware scheduler, allowing for prioritization of scheduling sequences based on information dependencies. *Anser* improves query performance by sending information sideways based on pre-determined dependencies that the information can be

effectively consumed. The scheduler provides decoupled information transmission and data processing links, enabling efficient and optimized scheduling of complex queries.

(4) We conduct extensive experimental studies to demonstrate the effectiveness and efficiency of the proposed method over public and in-house datasets.

The rest of this paper is organized as follows. In § 2, we introduce the architecture of *AnalyticDB* and the lifecycle of a query in *AnalyticDB* . We show the design of *Anser* in § 3. In § 4, we introduce how the scheduler coordinates with *Anser* to improve the query processing efficiency. We conduct experimental studies in § 5 to demonstrate the effectiveness of *Anser*. In § 6, we discuss the related works, and we conclude in § 7.

2 BACKGROUND

In this section, we first present the overall architecture of *AnalyticDB* in Section 2.1. Then, we provide in Section 2.2 a description of the query execution process, including query planner/optimizer and query scheduling/execution, which is further illustrated using an end-to-end example in Section 2.3.

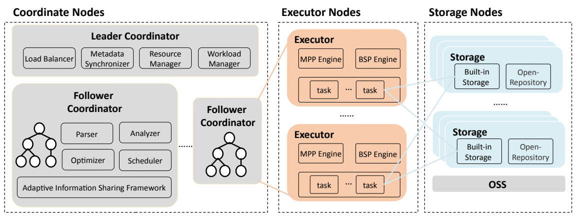


Figure 2: The architecture of *AnalyticDB*

2.1 Architecture of *AnalyticDB*

AnalyticDB adopts a massively-parallel and elastically-scalable query execution engine, which is decoupled from the storage engine. Figure 2 shows the overall three-tiered architecture. The three layers are all deployed on nodes, which are instances running on the Alibaba Cloud Elastic Compute Service [3].

The first layer is deployed on coordinator nodes and consists of a single leader and one or more followers. The leader serves as a centralized metadata synchronizer, collecting and notifying other followers of cluster-wide metadata through remote procedure calls (RPCs) when necessary. The followers act as frontends for query executions and manage one or more executors through RPCs. Upon receiving a query, a follower is responsible for compiling, planning, optimizing, and orchestrating its execution across the distributed executors.

The second layer consists of multiple symmetric executors deployed on executor nodes. The executor is responsible for query processing and supports both massively parallel processing (MPP) and bulk synchronous parallel (BSP), corresponding to an Interactive and a Batch compute mode, which is explicitly determined by users. The executor also supports an adaptive execution by interacting with the coordinator to adjust execution plans, resource allocations, degrees of parallelism, and access paths through adaptive information collected by *Anser*, which will be illustrated more in § 3. In *AnalyticDB* , the executor nodes could be easily and flexibly scaled out within seconds to meet the business requirements.

The third layer is deployed on the storage nodes. It includes a built-in storage and open-repositories with collections of external connectors (*e.g.*, OSS [6], HDFS [4], Kafka [2], etc.). The built-in storage system is a distributed storage system, and supports real-time data ingestion with strong consistency and high availability in compliance with the Raft consensus protocol. The storage system uses a data sharding partitioning strategy and a multi-raft architecture to support parallel processing. It supports a tiered storage that separates hot and cold data to reduce costs and a hybrid row-column storage layout with a fast and powerful index engine that supports predicate push-down to further accelerate query processing.

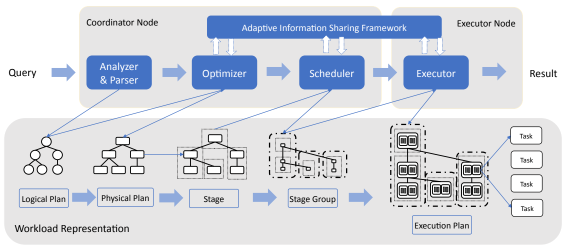


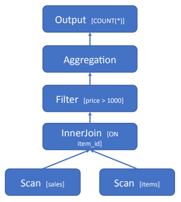
Figure 3: The query lifecycle management by *AnalyticDB* Runtime

2.2 Query execution of *AnalyticDB*

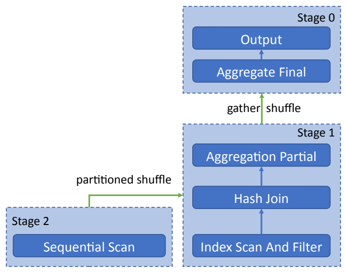
The *AnalyticDB* Runtime controls the entire lifecycle for each query from client submission to distributed execution. Figure 3 shows the overall pipeline.

Query Planner and Optimizer. After the SQL statement is submitted and assigned to one of the follower coordinators, the ANTLR-based *parser* first converts the statement into a syntax tree, then the *analyzer* resolves the relations and data types to build an initial logical plan in the form of a tree of *logical plan nodes*, where each node represents a logical operation (JOIN, FILTER, PROJECTION, *etc.*) and the parent-child relationship represents the data flow.

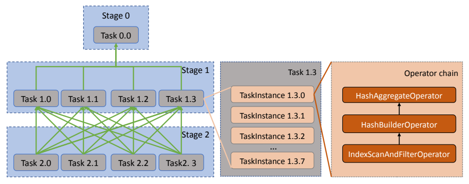
The *optimizer*, which supports both rule-based optimization (RBO) and the Cascades cost-based optimization (CBO) [23, 24, 36, 39, 41], further optimizes the basic logical plan. The RBO reads the initial logical plan tree as input, rewrites the plan tree based on a set of static rules without considering the data layouts and the physical characteristics of operations, and outputs a physical plan in the form of a tree of *physical plan nodes*. In this tree, each node stands for a physical *operator* that exactly maps to a physical implementation. Later, the CBO considers both the various costs associated with the operations as well as the data layout characteristics brought about by the distributed nature of executors and storage. Specifically, the optimizer collects 1) table statistics, including the number of rows and the average row length; 2) column statistics, including the number of distinct values (NDV) in a column, the minimum and maximum values in a column, data distributions (*e.g.*, histograms), and extended statistics to describe logical relationships among columns like a function dependency; 3) system



(a) The initial logical plan



(b) The optimized physical plan



(c) The distributed execution within a stage

Figure 4: An end-to-end example

statistics, including CPU and I/O performance and usage. These *optimizer statistics* are either collected from pre-analyzed offline metadata, or calculated by online sampling [8, 28, 35]. The CBO takes as input an initial physical plan tree, explores its equivalent implementations in a pruned space using techniques of a Cascades framework, analyzes and compares their costs using the collected optimizer statistics, and finally chooses an optimal physical plan.

During the CBO process, the distributed nature of executors and storage are considered. The distributed storage system stores data in partitions, and the executors deployed on different executor nodes process partitioned data in parallel. The *distributed property* in physical plan node describes how its processed data are partitioned. A subtree of physical plan nodes with the same distributed property (*i.e.*, process the data in the same partitions) forms a *stage*. Between two adjacent stages, in-memory or spillable data transfers, or *shuffles* are performed. The *blocking property* in physical plan node describes whether the physical operation needs to hold intermediate state to compute its output. Examples of operations with blocking property include hash-based sorting, aggregation or join operations [20]. The blocking property provides breakpoints in the execution process and opportunities for unscheduled physical plan nodes to be adjusted adaptively.

Query Scheduling and Execution. To execute the query plan, the *scheduler* first determines an order to schedule stages, then decides how and which nodes each stage is executed on the executors. The scheduler first divides stages into *stage groups*. A stage group holds one or more consecutive stages with at least one stage containing a physical plan node with blocking property, so that at least one breakpoint exists in the stage group to pause the execution. A Directed Acyclic Graph (DAG) is then formed with stage groups being nodes and shuffles between stage groups being edges. *AnalyticDB* Runtime supports two distinct scheduling policies: *all-at-once* policy and *group-phased* policy. The former dispatches all stage groups at once and benefits the latency-sensitive queries, whereas the latter schedules in a topological order according to DAG node dependencies.

AnalyticDB Runtime supports a two-level parallelism to accelerate the execution process. At the inter-node level, a stage is divided into *tasks* to run in parallel on one or more executor nodes. At the intra-node level, a task is further divided into *task instances* to run in parallel across threads. A task instance contains a chain of operators. Each operator corresponds to one physical plan node and performs corresponding computation on the input data. The task instances of the same task run on the same node so that execution states like hash tables, dictionaries and adaptive information can be shared among task instances. The degree of parallelism is determined by the scheduler to maintain stability of the overall cluster without sacrificing the query response time. The scheduler considers not only the current resource utilization but also the possible resources consumed by the subsequent scheduled tasks.

The scheduler interacts with the *resource manager* on the leader coordinator to allocate resources for tasks. For the all-at-once scheduling policy, the scheduler estimates and allocates resources at a query-level and only triggers execution if there are enough resources to run all query tasks. The query may throw an out-of-memory exception and fail if any task exhausts the cluster resource. For the group-phased scheduling policy, the scheduler estimates and allocates resources at a stage-group-level. In Batch mode, data is exchanged among stage groups in a spillable manner. Standard checkpoints are inserted throughout the data processing, and *AnalyticDB* is designed with a stage-group-level fault tolerance to automatically retry operations in the case of transient errors, which ensures the reliability of data processing.

2.3 An end-to-end example

Consider an example of SQL query:

```
        SELECT COUNT(*) from
        sales JOIN items on sales.item_id = items.item_id
        WHERE items.price > 1000;
```

It executes against two tables sales and items, partitioned by sales_id and item_id respectively. Figure 4a shows an initial logical plan. The RBO applies one logical optimization rule that pushes the filter on items into the connector to filter data as early as possible, also known as Predicate Pushdown [42]. Then, the optimized logical plan is assigned an initial physical plan, and the CBO evaluates the costs of its equivalent transformations and chooses the join order, the join and aggregation physical implementations, and the access path of table scans. Figure 4b shows a chosen optimal physical plan, with stages segmented by the distributed property of physical plan nodes and shuffles inserted between stages. Lastly, Figure 4c shows the stage and task scheduling process. The execution plan is scheduled in a bottom-up manner. For stage 1, four tasks are scheduled and by default eight task instances in each task are run in parallel simultaneously.

3 ADAPTIVE INFORMATION SHARING FRAMEWORK

In this section we present the design of *Anser* in details. We firstly define the adaptive information in §3.1, and introduce the design of *Anser* in § 3.2. Then, we present step-by-step implementation details in § 3.3 and discuss overhead introduced into the system and how we reduce the costs in § 3.4. Lastly, we show some applications of *Anser* developed in *AnalyticDB* in § 3.5.

3.1 Definition of adaptive information

The *adaptive information* is defined as all relevant statistics that could be collected or computed during execution to improve the query processing efficiency. It includes re-evaluated optimizer statistics as well as sideways information [30], that through equivalent relation inference, statistics from one relation could be applied to another. We characterize the adaptive information into two types: *primitive* information and *non-primitive* information. The *primitive* information is directly collected with trivial costs (*e.g.*, row count and MIN/MAX of columns), while *non-primitive* information is either directly detected from other primitive statistics (*e.g.*, NDV), or calculated with extra costs (*e.g.*, bloom filters [47, 48]). Using adaptive information, we could choose a better physical operator implementation (*e.g.*, replacing the original Sort-Merge JOIN operator with a Hash JOIN operator [22]), perform early pruning of unnecessary computations (*e.g.*, with the help of magic sets [41] or hash filters [16]), or adjust the degree of parallelism at either inter- or intra- node level.

With the distributed nature of our system, the information is associated with its granularity, in order to describe whether it is a partition or an aggregation statistic of the data set. During execution, data are divided into partitions and processed in parallel. As a result, a physical operator can only collect statistics on the partition it processes. If the information is passed to another operator with the same distributed property, the other operator could use partition-level statistics directly. Otherwise, for information consumed by operators with different distributed properties, or the optimizer/scheduler module which makes whole-picture adjustments, we need to integrate them into aggregated statistics.

The framework supports information sharing to avoid redundant computation. For information collected from different operators, if they describe the same data set, the information can either be reused if they are the same type of statistics, or be inferred if one can derive the other. For example, in Figure 5, table *t1* scans 28-billions rows of data and joins with three aggregated rows of table *t2.* Without adaptive information, normally, a full scan and a hash join will be executed. However, two adaptive strategies could be used to reduce the computation cost. In Case 1, the builder side of join could collect and pass a runtime filter to *t1* and performs a bloom join [36] before scan. In Case 2, a histogram could be used by the builder side of hash join to determine the output offsets to perform a radix partitioning [45]. The runtime bloom filter and the histogram are the two feasible adaptive information, where we only collect once from the hash table built by the aggregate operator. We could compute both the bloom filter and the histogram [48] respectively.

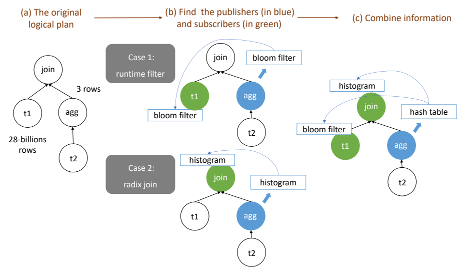


Figure 5: An example of derivable information.

The information used in adaptive applications such as bloom joins [47] requires strict currency control to ensure correctness. Furthermore, it is abundant in terms of its varying types and levels of granularity. Given the dynamic query workloads and the space-consuming characteristics of certain non-primitive information, we determine to merely collect and utilize the query-level data without persisting it for future use.

3.2 The architecture of *Anser*

Anser consists of three parts as shown in Figure 6: *publisher*, *subscriber* and *channel.* The publisher is implemented as a query operator that could be seamlessly ingested to a query plan, and it collects information and publishes the information to the channel. The channel works in a centralized mode. It contains a manager that matches publisher-subscriber relationship and manages the information lifecycle, and a service that receives, deduces, and passes the information to the subscriber. The subscriber is the consumer of the information which is also implemented as a query operator. It can interact with other operators, the optimizer, and the scheduler, to orchestrate the query execution.

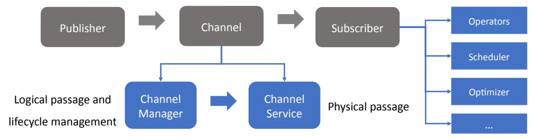


Figure 6: The architecture of the *Anser.*

Publisher. We utilize different semantics of operators to reduce the collection cost, without interrupting the pipeline of operators. All operators collect some primitive statistics like the number of rows and the data volume during execution, which can be directly used without introducing additional costs. Some operators derive non-primitive statistics with negligible cost. For example, the hash table is generally built by hash-based aggregation and join operators. We could use the hash table to obtain a hash set, a histogram, or to calculate the number of distinct values. The cost of collecting such derivable non-primitive statistics brings negligible costs. For the non-primitive statistics that cannot be derived, we define a *PubOperator* to compute the required information, and it is inserted into the original physical query plan to compute the non-primitive statistics during execution.

Subscriber. Based on different adaptive execution cases, the subscribers interact with operators, the optimizer or the scheduler to optimize the options of physical plan node, the execution plan, or the resource allocations of tasks. For example, for hash-based joins, we normally build the hash table over the relation of the smaller size. Due to the estimation error, the optimizer could be misled to build the hash table over the larger relation during the query planning phase. To re-optimize the join order, the optimizer needs to calibrate the real volume of the two input relations of the JOIN operator with a subscriber. Then the join reordering is triggered by a re-optimization step only if the adaptive information differs from the estimation. We define a *SubOperator* to consume the information and guide the adaptive execution. During execution, each subscriber has a *weak dependency* on corresponding publishers. Normally, the subscriber operator blocks the task instance and waits until the information is received and consumed. However, the information we pass is used to tune for better performance instead of calculating the final results. In abnormal circumstances that no information is received due to failures (*e.g.*, network), we cancel the blocking of the task instance after timeout. Such cancellable dependency is called a *weak dependency*.

Channel. The publisher and subscriber are linked via a channel, including a channel manager and a channel service. The channel manager builds the logical linkage between the publisher and the subscriber, and manages the lifecycle of the associated dynamic information. During the query planning phase, the manager matches each pair of publisher and subscriber and registers the corresponding channel. The matching relationship is built upon case-specific rules, and these rules will be explained in § 3.3. For reusable or derivable information, its publisher could be connected with more than one subscriber. During execution, once the information is invoked, the manager decides where to send it and when to destroy it. The manager stores two key data structures: 1) a graph consists of nodes representing publishers and subscribers and edges describing the relationship between them; 2) a hash map used to store information with its state as the value. The state represents the lifecycle and status of the information and contains instructions to clean and recycle memory allocated for the information once all relevant subscribers are destroyed. The key-value pairs in the hash map are accessible through information identifiers, and the states are updated based on the events triggered by the publishers and subscribers. The channel service establishes the physical connection between the publisher and the subscriber, determines the appropriate executor nodes to send the information, and manages information transfer. Additionally, it processes publishers as query operators in parallel across different executor nodes. Partition-level information from each operator/publisher is collected by the channel service, then transmitted to corresponding coordinator node using a client-server model. If global information is required, the information is aggregated then broadcasted to all subscribers.

3.3 Implementation

Information registration. To enable adaptive execution in each case, we design a greedy algorithm to declare the required information, specify the corresponding publishers and subscribers, and register them in the channel manager. This process involves several steps. Firstly, we define the necessary information for each adaptive execution case and identify the plan node in the plan node tree that produces or consumes that information. For instance, for bloom joins, the required information consists of bloom filters generated from all builder sides of hash joins, where builder nodes produce the information and probe nodes consume it. We create a global context to store information defined by a key-value hash map. The keys and values are symbols that represent the relations that produce or consume the information, respectively. The symbols are generated by the optimizer through algebraic equivalence that uniquely mark an attribute of the same sub-expression. Next, we traverse the plan node tree top-down and find the first-visited plan node whose output symbols match any of the keys in the global context. We only match one plan node for each information to avoid repetitive production of information. We add a plan node *PubNode* as the parent node of the matched plan node to collect and publish the information through processing output data from the matched plan node. At the same time, we search for all plan nodes whose input symbols match any of the values in the global context. For each matched plan node, we add a *SubNode* as a child node to subscribe to the information.

Secondly, we employ standard optimization rules such as those proposed in [17, 21] to push down the PubNodes and SubNodes. PubNodes are pushed as deep as possible in the query plan to ensure that information could be produced as early as possible. SubNodes are pushed down in a cost-based manner. For SubNode that prunes intermediate results and benefits downstream operators like bloom filters, we evaluate its data reduction rate and push it down if the rate is higher than its child plan nodes.

Lastly, we try to merge each PubNode/SubNode with their child plan nodes that provide derivable information (e.g., bloom filter as the publisher and AggregationNode as its child plan node) or have the same functionality (e.g., bloom filter as subscriber and FilterNode as its child plan node), which helps to avoid duplicate production and consumption of information. PubNodes/SubNodes that are not combined with existing plan nodes are translated into PubOperators/SubOperators. Following this, we set up channels for every pair of finalized publisher and subscriber and register them in the channel manager. Figure 7 depicts an example of the entire process.

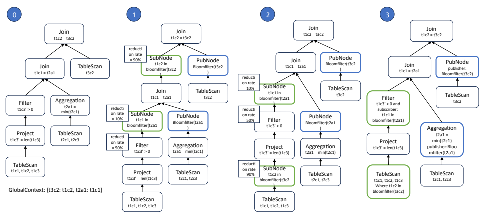

0
1
2
3
Join t1c2 = t3c2
Join t1c1 = t2a1
TableScan t3c2
Filter t1c3' > 0
Aggregation t2a1 = min(t2c1)
Project t1c3' = len(t1c3)
TableScan t2c1, t2c3
TableScan t1c1, t1c2, t1c3
GlobalContext: {t3c2: t1c2, t2a1: t1c1}
reduction rate = 90%
SubNode t1c2 in bloomfilter(t3c2)
PubNode Bloomfilter(t3c2)
reduction rate = 10%
SubNode t1c1 in bloomfilter(t2a1)
PubNode Bloomfilter(t2a1)
reduction rate = 50%
PubNode publisher: Bloomfilter(t3c2)
Filter t1c3' > 0 and subscriber: t1c1 in bloomfilter(t2a1)
Aggregation t2a1 = min(t2c1) publisher:Bloomfilter(t2a1)
TableScan t1c1, t1c2, t1c3 Where t1c2 in bloomfilter(t3c2)

Figure 7: An example of information registration.

Information collection and consumption. We implement PubOperator/SubOperator to collect and consume the information respectively. The PubOperator is simply a statistics collection operator. It outputs the original input data along with required statistics. We set a threshold for each PubOperator, which determines whether to cancel information passing during execution. The threshold is heuristically set to limit the memory and the CPU consumption that arise from collecting, transmitting, and consuming the information. The implementation of the SubOperator varies depending on specific use case. For cases where information subscription is resource-consuming, we evaluate the costs and benefits using standard cost models proposed in [41, 43], but with during-execution statistics as inputs, and set a threshold on the SubOperator. For example, in the case of bloom join, the SubOperator is a filter operator that prunes its input data streams with a bloom filter summary of information collected from the builder side. We estimate the data reduction rate of the published bloom filter, and cancel the subscription if the reduction rate is lower than the threshold. In other scenarios, the SubOperator may need to interact with the optimizer or the scheduler. The current implementation supports stage-group-level plan re-optimization and intra-node parallelism tuning. In the former case, the optimizer takes a hash map generated from the SubOperator as input, with keys as the plan nodes to be optimized and values as the collected statistics, and re-optimizes the physical plan based on during-execution statistics. In the latter case, the scheduler takes the upstream data size as input and heuristically estimates an appropriate number of task instances to be scheduled.

Information transmission. The channel service collects information from publishers, aggregates it if necessary, and shares the processed information with subscribers. The service is developed to fit the two-level parallelism of our system architecture, where each executor node deploys a local service, and the coordinator deploys a remote service. The local service collects information generated by PubOperators in task instances and aggregates them at the partition level once all PubOperators in the current task complete the information production. The local service then sends the partition-level information to the remote service through RPCs as soon as it is aggregated. The remote service collects information from the local services and aggregates it at the query level as soon as all tasks have finished running. Finally, the remote service sends the query-level information back to local services through RPCs for SubOperators to consume.

Several essential implementation details are in place to ensure the accuracy and timeliness of information transmission. Firstly, to handle network failures or operator cancellations, each information is labeled with a binary cancellation flag. This enables us to distinguish between actual empty information and failed or canceled information. Furthermore, we have implemented an ACK (acknowledgement) mechanism in conjunction with a retry policy allowing for a maximum of three attempts. If the RPC fails or the operator cancelled publication, we send an empty information with the cancellation flag set to true. This information is then directed to all subscribers, allowing for prompt cancellation of their subscriptions without waiting for other partitions. Secondly, we declare the expected partition number in the channel manager, and the remote service keeps track of the number of received partition-level information during transmission. Only when all partitions are collected successfully without cancellation, will we then construct the aggregated information and send it back to the local services. Thirdly, to optimize performance and reduce transmission delays, we have implemented a push-based model that allows information to be sent to subscribers as soon as it is published. If the information is ready before the subscriber/SubOperator is scheduled, it is cached locally and consumed asynchronously by the subscriber when scheduled.

3.4 Cost analysis

Adding publishers and subscribers during execution can lead to increased overhead due to the production, transmission, and consumption of information, potentially leading to a negative impact on response time. In this section we discuss how *Anser* diminishes such costs.

Firstly, we restrict the memory usage of *Anser* to reduce production and storage overheads. The information is only used at the query level and is destroyed as soon as the query is completed, with a memory limit of 1MB per information record and 200MB per channel service. The limits are set heuristically. We set a threshold (= 1MB/number of task instances) for each PubOperator to cancel production. Furthermore, we clean the oldest information cached in the service when the total size exceeds 200MB. This explicit resource restriction ensures that extremely large data is not cached, keeping CPU and network expenses insignificant. Secondly, we optimize data transmission and network costs by sending only necessary information. We support sharing the same information with multiple subscribers. In such cases, the information is sent to the coordinator node once and broadcasted to all associated executor nodes. For multiple subscribers on the same executor node, the information is sent only twice: once to the coordinator and once to the executor node, eliminating unnecessary network connections. We also merge multiple information records on the same executor node into a single RPC to reduce network requests. Additionally, in cases where partition-level information is required, the data is sent directly to subscribers on the same executor node through the local service, eliminating transmission through the network. Thirdly, to reduce information consumption costs, we set a case-specific threshold for each SubOperator and cancel consumption when its cost outweighs its benefits. Besides, the SubOperator blocks the execution of downstream operators until the information is received, which may lead to regressions on query response time. Section 4 will explain more on how the block time is adaptively adjusted to avoid such regressions.

3.5 Applications

We have built various adaptive applications on top of *Anser* . In this subsection, we discuss three of them implemented in the framework. We provide a general description of how each case improves query performance and how they are implemented in the framework.

Runtime filtering. Runtime filtering is supported in many open-source databases including Impala, Spark, Hive and Doris [5, 50]. It passes a dynamically built filter from one small sub-relation (the publisher in *Anser*) to a large sub-relation correlated by a join condition (the subscriber in *Anser*), and performs a bloom join on the subscriber with the runtime filter, in order to prune irrelevant results before we perform an expensive operation. Some systems such as Impala [5] limit subscribers and publishers to table scan operators directly joined by a condition. Spark's [50] dynamic partition pruning (DPP) additionally limits the table scans to be partitioned on join columns. In both implementations, the table scan waits a fixed amount of time for the filter to arrive and perform a bloom join. By early pruning of reading data, unnecessary I/O and network transmission could be avoided. Other implementations like AIP [30] deduce correlations during execution and set no limitations on subscribers and publishers, but a large table scan operation will only be avoided if a runtime filter is already built. *Anser* performs the greedy algorithm before execution to match as many publisher-subscriber pairs as possible, and multiple publishers that provide the same bloom filter will be merged.

Early stopping for joins with an empty subrelation. A join with an empty subrelation can be thought of as a runtime filter with no matching rows, which means the filter will prune all the rows from the other relation, resulting in an empty join output. Thus, all operators in the subscriber's operator chain could stop early. This technique is rarely implemented as an optimization rule since little optimizer statistics implies an empty relation for sure. Spark's adaptive query execution has a similar feature to dynamically detect and propagate empty relations [50]. In *Anser*, we do not need to collect extra information and could directly apply early stopping with information collected from runtime filtering.

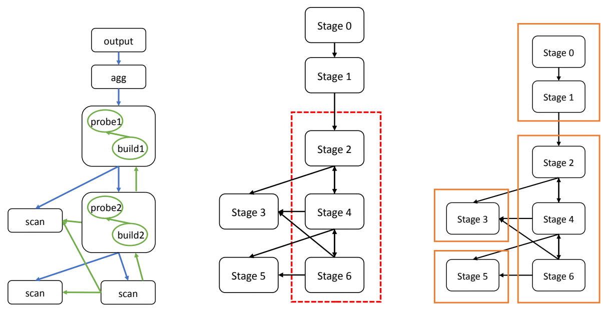
Adaptive partial aggregation. Partial aggregation push-down is a well-studied optimization technique [15, 27, 33] to reduce input data size by pushing group-by below certain operators. While it reduces computation and shuffle costs, it adds CPU and I/O costs due to the added aggregation operator. To evaluate the trade-off, the aggregation reduction ratio, which refers to the percentage of data reduced by pre-aggregation, is commonly used. Some systems, such as [37], estimate it using offline statistics, while others, such as [13, 38], use online sampling during execution. The former approach relies on accurate estimations while the latter introduces extra costs during execution. We improve upon the latter approach by integrating information collection into *Anser*, reducing sampling overhead. We initially register partial aggregation as the publisher and the subscriber, and push down the publisher node, trying to fuse with other operators who produce information that could derive the aggregation reduction ratio such as hash tables.

4 ANSER-BASED SCHEDULER

In this section, we present the *Anser*-based group-phased scheduler. It utilizes a novel scheduling policy that optimizes both data and information exchanges in the physical execution plan. The optimizer builds a physical plan and divides it into stages, which are then organized into a DAG of stage groups, as described in §4.1. Each stage group is assigned a priority, based on the scores defined in §4.2. The scheduler then dispatches stages in a step-by-step process, illustrated in §4.3. Finally, we discuss some implementation details in §4.4.

4.1 Stage group

The stage groups are created in a manner that ensures there are no cyclic dependencies among the data streams in the groups. Additionally, at least one source stage or one stage with blocking property is included in each group. This creates a DAG where the stage groups represent nodes, and shuffles between stages in different groups



(a) The dependencies among stages **(b) The strongly connected components** **(c) The generated stage groups**

Figure 8: Three steps to generate stage groups

represent edges. The skeleton graph of the DAG prescribes a fixed order for query evaluation. The stage groups are generated in three steps.

Step 1. We define a *strong dependency* between stages as follows: Stage A is strongly dependent on Stage B if there exists an operator in Stage A that requires data stream input from operators in Stage B. For instance, in-memory shuffles require the source operator to consume the data produced from the sink operator, or else the sink operator would be blocked. Another example is hash join where the probing operator is blocked by the builder side to build the hash table, meaning that all the probing stages are strongly dependent on all stages at the builder side to finish building. The presence of a strong dependency requires Stage B to be scheduled before Stage A. Figure 8a illustrates a sample graph where the directed edges represent strong dependencies. In this example, the query is in interactive mode with in-memory shuffles inserted between stages, and a hash join is selected as the join method. Blue lines connect sources to sinks, and green lines connect builders to probes, with both lines pointing from the stage that must be scheduled first to its dependent stage.

Step 2. We identify the strongly connected components in the graph generated in the previous step. Stages within a strongly connected component have cyclic dependencies and must be scheduled at the same time to avoid deadlocks. Figure 8b illustrates a strongly connected component, where all the stages within the red rectangle must be scheduled concurrently. Consider Stages 4 and 6 as an example. If Stage 6 is scheduled while Stage 4 is not, the output sink operator in Stage 6 would be blocked since the source operator in Stage 4 is not consuming data. Conversely, if Stage 4 is scheduled while Stage 6 is not, the probe operator in Stage 4 would be blocked since there is no input data from Stage 6 to build the hash table. As a result, stages with cyclic dependencies, such as Stages 4 and 6, must be scheduled concurrently.

Step 3. To generate the stage groups, we merge one or more strongly connected components into one stage group and require at least one source or blocking stage in each group. A stage is considered a blocking stage if it contains a physical plan node with a blocking property. This property provides natural breakpoints in the execution process, allowing for the re-evaluation of the query execution plan [20]. We use a depth-first search to traverse the graph from the output stage and treat strongly connected components as nodes. We cache visited components in a list until we encounter a component that contains either a source stage or a blocking stage. We then create a stage group consisting of all components in the cached list, clear the list, and continue traversing to form the next stage group. Figure 8c illustrates the resulting stage groups.

4.2 The priority of a stage group

Anser defines a *weak dependency* between operators such that a subscriber is weakly dependent on its corresponding publisher. Ideally, the publisher should run first and pass the required information before the subscriber starts running. Our scheduling algorithm takes this weak dependency into consideration. Unlike the strong dependency where the order of stages is required to avoid deadlocks, the order is preferred when there is a weak dependency. This is because the execution results are not affected by the order of weakly dependent stages.

We assign priorities of the generated stage groups to suggest a preferred order when multiple stage groups are to be scheduled in the DAG. The weak dependency is defined between operators, and one stage or one stage group may contain multiple publishers and subscribers. We assign an integer priority score $s \in [0, 5]$ to each stage group and schedule according to s in a descending order, where stage groups with $s = 0$ are scheduled first.

Firstly, we assign $s = 0$ to stage groups that are not strongly dependent on others or whose strong dependencies have already been scheduled, to make sure these stage groups are scheduled first. We then assign $s \in [1, 5]$ to stage groups falling under the following five categories respectively: (1) stage groups with subscribers whose corresponding publishers have all finished; (2) stage groups with publishers only; (3) stage groups without subscribers or publishers; (4) stage groups with subscribers whose publishers are running; (5) stage groups with subscribers whose publishers are pending. The priority scores are re-evaluated whenever multiple stage groups are ready to be scheduled.

4.3 The execution of Anser-based scheduler

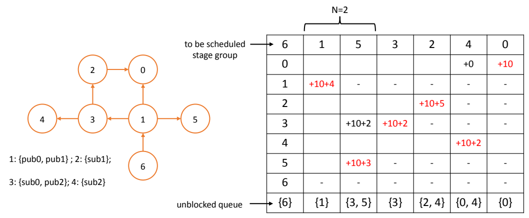
The *Anser*-based scheduler follows four steps to schedule queries. Figure 9 shows an example. Assuming in the example all stage groups take the same amount of time to execute, we demonstrate a static order to show how priorities are assigned and stage groups are chosen.

Step 1. We construct a DAG with stage groups as nodes and strong dependencies as directed edges. The DAG is generated by the process described in §4.1. The priority score of each node is calculated according to the rules described in §4.2.

Step 2. In brief, the execution order of all stage groups is determined by the topological sorting algorithm. In the implementation, we manage all stage groups with two priority queues. The unblocked queue consists of all stage groups with no strong dependencies on others, meaning they are ready to be dispatched. The blocked queue holds the remaining stage groups. Both queues are arranged as priority queues based on the priority scores of each stage group, which are dynamically updated by the stage group state listener. The state listener keeps track of the execution status of task instances, tasks, stages, and stage groups. After a stage group in the unblocked queue completes its execution, its dependents in the blocked queue are moved to the unblocked queue.

Step 3. To optimize resource utilization while still taking weak dependency into account, we dispatch N stage groups at a time based on the concurrency control factor N, which can be adjusted according to cluster resources usage. The N stage groups are prioritized in the unblocked queue first, and if there are only m ($m < N$) stage groups in the unblocked queue, we dequeue ($N - m$) stage groups from the blocked priority queue to execute.

Step 4. Within each dispatched stage groups, all stages are dispatched at the same time. Each stage is then broken into multiple tasks and runs in parallel on executor nodes.


N=2
to be scheduled stage group
2
0
4
3
1
5
6
1: {pub0, pub1} ; 2: {sub1};
3: {sub0, pub2}; 4: {sub2}
unblocked queue

6	1	5	3	2	4	0
0					+0	+10
1	+10+4	-	-	-	-	-
2				+10+5	-	-
3		+10+2	+10+2	-	-	-
4					+10+2	
5		+10+3	-	-	-	-
6	-	-	-	-	-	-
{6}	{1}	{3, 5}	{3}	{2, 4}	{0, 4}	{0}

Figure 9: An example of scheduling orders of stage groups

4.4 Implementation details and discussion

The scheduler of *AnalyticDB* , powered by a unified hybrid compute engine, efficiently handles workloads that consist of both low-latency high-concurrency interactive analytics and large-scale fault-tolerant offline jobs. To avoid maintaining multiple scheduling algorithms, we rely on a general-purpose scheduler that effectively allocates system resources.

To handle both short and simple as well as long and complex queries with no significant execution overhead, the scheduler of *AnalyticDB* needs to be versatile. For simple queries consisting of less than five stages, we apply the all-at-once scheduling policy without constructing stage groups to eliminate additional scheduling costs. However, for complex queries, scheduling all stages at once may hamper the downstream stages without input data as they cannot proceed until their upstream stages are completed. To address this issue, we use the group-phased scheduler, which schedules stages when their upstream stages start outputting data. It results in improved resource utilization and reduced idle-spinning of resources. In production, network connections are reduced by 50%-95%, peak memory utilization is reduced by 50%, and pending tasks are reduced by up to 99% for large and complex clusters. The average scheduling time cost of the group-phased scheduler is only 327 microseconds. Meanwhile, for batch workloads, typically involving large-scale data ETL from multiple tables, scheduling all table-scan tasks simultaneously as the bottom-most stages can cause concentrated hotspot distribution on I/O, resulting in degraded performance. To mitigate this issue, the group-phased scheduler restricts the number of concurrently scheduled stage groups, reducing hotspot distribution density.

5 EXPERIMENTAL STUDY

In this section, we conduct an experimental evaluation to validate the performance improvements of *Anser* discussed in §3 and §4. Firstly, we evaluate the overall performance on a benchmark workload. Then, we analyze three applications discussed in §3.5 on production workloads.

5.1 Experiments on benchmark

5.1.1 Experiment setup. We run benchmark tests on the TPC-DS workload with a scale factor of 1000. Our baseline system is Spark [50], which features the adaptive query execution (AQE) and the dynamic partition pruning (DPP). We compare Spark's DPP with *Anser* 's runtime filtering (RTF), which targets similar optimization scenarios. We use the built-in storage engine with a decoupled executor and storage design in *AnalyticDB* and the Hive [1] storage engine with a coupled executor and storage design in Spark. We set up three *AnalyticDB* clusters to simulate different system loads: idle, normal, and busy. All nodes have 16 CPU cores and 64GB main memory for both *AnalyticDB* and Spark. The idle/normal/busy *AnalyticDB* clusters respectively have 36/24/12 executor nodes and they all have 24 storage nodes. The Spark cluster has the same settings as the idle *AnalyticDB* cluster of 36 executor nodes.

5.1.2 Overall results. We run all 99 TPC-DS queries, and evaluate total response time with both AQE and DPP disabled in Spark, and *Anser* disabled in *AnalyticDB* . Then we turn on AQE and DPP/RTF separately, and finally we enable both features. The improvements of the query response time are shown in Figure 10. Compared with the baseline, for *AnalyticDB* , there is a 30% and 10% reduction on response time with RTF and AQE enabled respectively, and an overall of 61% improvement with both features enabled.

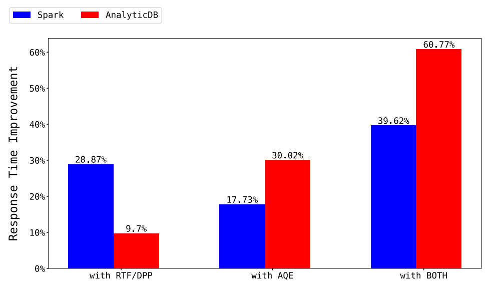


Figure 10: Overall performance comparison between *AnalyticDB* and *Spark*.

5.1.3 Ablation study. We conducted an ablation study using the RTF feature to demonstrate the effectiveness of our proposed methods. Enabling RTF results in a 57% improvement in response time (from 990.25s to 430.25s) for the TPC-DS 99 queries. Then, we select 22 relatively long-running queries (response time > 2s) out of 99 TPC-DS queries and analyze their response time. Figure 11 shows the response time of the 22 selected queries, with an average improvement of 81%. We then construct the following three experiments on the 22 selected queries to demonstrate the effectiveness of our designs.

Firstly, we evaluate the effectiveness of the greedy algorithm which declares required information and finds its publisher and subscriber by searching the entire plan node tree to maximize the number of subscribers through top-down traversal. Thresholds are set on PubOperator/SubOperator to cancel production or consumption if costs become significant, which are evaluated against during-execution statistics. In contrast, other approaches like Impala [5], limit the search space of options and only consider subscriber-publisher pairs as directly joined table scan operators, and use estimated statistics to decide whether to add these subscriber-publisher pairs, making it easier to miss optimization opportunities and effective subscribers. We compared our approach against a baseline algorithm following Impala's implementation. Before execution, a runtime filter is considered effective if its publisher and subscriber are correlated through a join criterion. With limitations on effective patterns and estimated cost upper bounds, the baseline algorithm generates 61 filters, whereas the greedy algorithm generates 170 filters. During execution, a runtime filter is considered effective if it filters more than 60% of input data. Out of the 170 filters generated, 104 of them are effective during execution. Therefore, with a greedy algorithm, more effective information can be extracted and utilized.

Secondly, we validate the benefits of constructing channels during the planning phase rather than execution. Pre-execution channels allow the scheduler to adjust priorities and block the subscriber until the filter is built and received. In contrast, during-execution search algorithms like AIP [30] fails to control the scheduler's priorities. For example, if the table scan-like subscribers may be processed before the arrival of the filter. We evaluate the total scanned data size with and without pre-execution channels. Without introducing the group-phased scheduler (to be evaluated in the third experiment), we apply an all-at-once scheduling policy and manually set the lowest priority on table scan subscribers, then compare both cases in the idle cluster. Figure 12 shows a comparison of scanned data size in gigabytes. Pre-execution channel registration reduces scanned data size by over 96%, resulting in significant reductions in network and I/O costs. Therefore, implementing a framework to manage information channels before execution is vital to make sure information is consumed as much as possible.

Thirdly, we evaluate the response time of two different scheduling policies, the all-at-once scheduling policy (SP1) and the *Anser*-aware-group-phased scheduling policy (SP2), under different system loads. We set the concurrency-control factor N to 4, 6, and 8 for busy, normal, and idling clusters, respectively. We set a fixed wait time on all subscribers for both policies, which is the maximum time allowed for subscribers to receive and consume the information. If the subscribers receive information before timeout, they start to consume as soon as the information is received. We vary the wait time from 0ms to 1200ms with a step size of 200ms, and add a wait time of 9999s as a baseline assuming all filters could be consumed. Figure 13 shows that the group-phased scheduler outperforms any wait time variation of the all-at-once scheduler. It also has similar performance with varying wait time, suggesting that the wait time is adaptively selected with stability and is relatively optimal due to priority control.

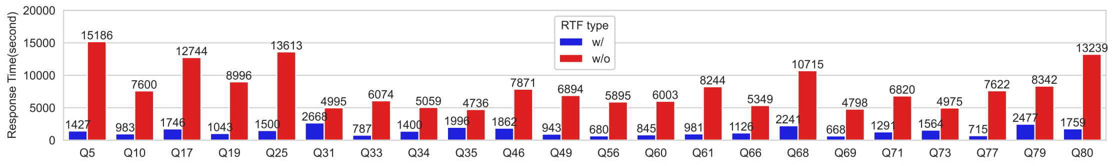


Figure 11: Response time test of RTF.

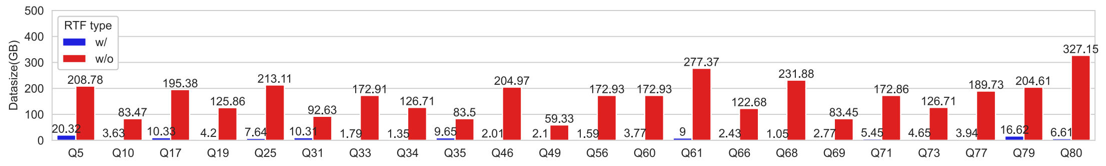


Figure 12: Scanned datasize test of RTF.

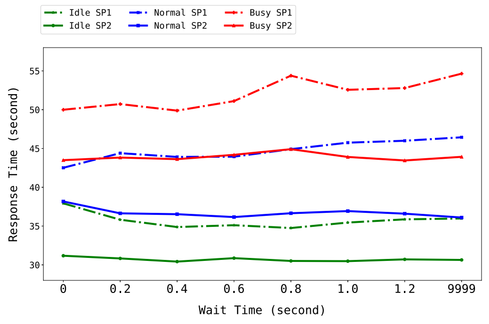


Figure 13: Comparison between group-phased-scheduler and all-at-once scheduler.

5.2 Performance evaluation in production environment

Anser along with the proposed scheduler has been deployed in more than 90% of *AnalyticDB* 's clusters in production environment. In this subsection, we present three representative cases on the performance of *Anser*'s applications on multiple production workloads. We also perform an analysis on the information management mechanism that evaluates whether certain information should be collected, using pre- versus during-execution statistics.

5.2.1 *Adaptive partial aggregation.*

There are more than 50% partial aggregation operators with an aggregation reduction ratio of less than 20%, while 5% operators with a reduction ratio of more than 99%. We evaluate the effectiveness of adaptively bypassing partial aggregation in Cluster A. The aggregation operator is the most CPU intensive operator in the cluster, and the cluster has a mixed distribution of aggregate reduction ratios. The cluster has 24 storage nodes and 24 executor nodes (16 CPU cores and 64GB main memory for each node). The partial aggregation operator takes up 34% CPU time of all aggregation operators. 92% of the partial aggregation operators have a reduction ratio of 0%, and more than 96% have a reduction ratio of less than 20%. Figure 14 shows the results of this test. We analyze four common patterns occurring more than 1000 times daily. These patterns involve sub-relations join and aggregation with distinct (A1 and A2), count (A3), or sum (A4) functions. We compare the average processed data size and wall time of the partial aggregation operator, as well as the query response time with adaptive partial aggregation ON versus OFF. We can see that, by skipping partial aggregation operators with low reduction ratio, CPU resources are saved and the end-to-end query response time also decreases.

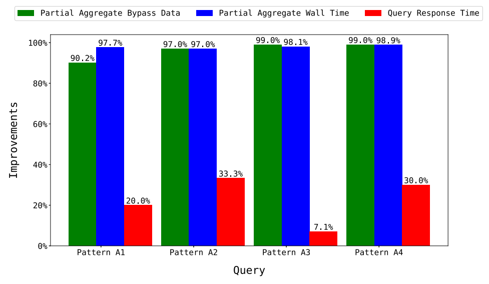


Figure 14: Evaluation of adaptive partial aggregation.

5.2.2 *Early stopping for joins.*

For all online patterns with at least one join operator, about 10% of them can benefit from empty join early stopping, defined by one side of join being empty while the other side processing more than 1MB data. We select two typical workloads to evaluate the effect. Cluster B has 6 executor nodes and 3 storage nodes (24 CPU cores and 96GB main memory for each executor node and 16 CPU cores and 64GB main memory for each storage node). Pattern B1 with more than 1000 daily occurrence has 7 joins and its execution plan is a left deep tree whose left-most source table is empty. By enabling early stopping on this pattern, all join operators as well as all operators at the builder sides are terminated. Cluster C has 3 executor nodes and 3 storage nodes (16 CPU cores and 64GB main memory for each node). 30% of its patterns with join operators can apply early stopping. Pattern C1 (or C2) has deep/bushy tree execution plans with 5 (or 3) joins, where the second-left-most/right-most source table is empty. Figure 15 shows the volume of data that are pruned and the reduction ratio of the query response time. These results demonstrate the significant performance benefits of early stopping for join over empty input relation.

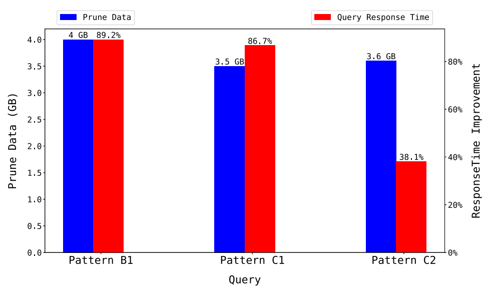


Figure 15: Evaluation of early stopping for joins.

5.2.3 *Runtime filtering.* RTF is effective in more than 85% online clusters. Two clusters are chosen to show its effectiveness. Cluster D has 8 executor nodes and 12 storage nodes (24 CPU cores and 96GB main memory for each executor node and 16 CPU cores and 64GB main memory for each storage node). Cluster E has 2 executor nodes and 3 storage nodes (16 CPU cores and 64GB main memory for each node). We compare the scanned data size and query response time of 3 query patterns from each cluster's workload, with and without RTF enabled. Patterns D1/D2/D3/E1 include 2 tables joins with one side having less than 1000 rows and the other having millions of rows. Pattern E2/E3 are 5/9 joins with multiple RTF generated and pushed to multiple subrelations including table-scans. Figure 16 shows the results. We can see that by pruning irrelevant data in sub-relations, RTF reduces the query response time and the scanned data size considerably, thus saves CPU, IO and network resources as a result.

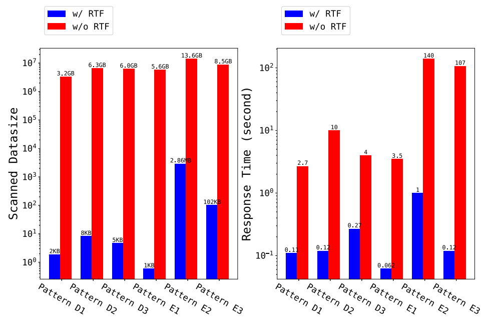


Figure 16: Evaluation of runtime filtering

5.2.4 *Evaluations on statistics.* We evaluate the accuracy of statistics collected before versus during execution. As mentioned in § 3.3, we have implemented thresholds on the publisher nodes to cancel production and transmission if the costs exceed a certain level. *Anser* uses during-execution statistics to evaluate such costs. We evaluate the effectiveness of the during-execution cost estimations based on aggregated analysis from all online clusters with *Anser* enabled. We take RTF as an example, whose threshold evaluates the filter_rate, which is an estimation of the data size that could be reduced by a runtime filter. We calculate act_filter_rate = $\frac{join\_act\_rows}{left\_act\_rows}$, pre_est_filter_rate = $\frac{join\_est\_rows}{left\_est\_rows}$, and adaptive_est_filter_rate = $\frac{right\_act\_rows}{left\_act\_rows}$, and compute a confusion matrix to compare the accuracy with 0.6 as the heuristic threshold. For starters, 8.11% of pre-estimated statistics are not a number (NaN). Table 1 shows the confusion matrices. The accuracy (the degree of closeness to true value, i.e., $TP + TN$) of pre-execution cost estimation is 62.78%, while the accuracy of during-execution adaptive cost estimation is 84.05%, which is an increase of more than 25%. Therefore, using a during-execution evaluation is more accurate both in terms of eliminating the useless information (TN) and keeping the useful information (TP).

Table 1: Confusion matrices for pre- versus adaptive- estimations

	TP	FN	FP	TN
pre-estimation	50.64%	16.14%	20.99%	12.23%
adaptive-estimation	60.97%	5.46%	10.49%	23.08%

6 RELATED WORKS

Adaptive query processing in literates is not only widely studied, but also extensively implemented in many commercial databases including Oracle [34], SQL Server [7], Spark [50], DB2 [44], *etc.* Most of these systems focus on a plan re-optimization with few alterations of the execution process, either with feedback statistics collected post-execution that improves the plan the next time it is generated [19, 44] or with dynamic optimization similar as our work that re-optimizes the plan during execution [50]. Other plan-based techniques require slight modifications on the execution process. A progressive parametric optimization such as [19, 29], generates parametric plans and caches the plans in an adaptive cursor sharing framework [50], which is progressively optimized based on execution statistics. Another array of techniques termed proactive reoptimization generates multiple sub-plans per pipeline in the query, and uses a choose-plan operator [25] or a switch operator [12] to choose between different sub-plans during execution in the pipeline based on run-time statistics. Also, there are row-routing techniques, originated from the Eddies framework [11, 18, 19, 40], that implement a run-time optimizer to route each row independently through a sequence of join operators. Though a routing-based technique reduces compile-time optimization overhead and provides high flexibility, it requires major changes in our push-style query processing engine and is not adopted in our system.

7 CONCLUSION

In this paper, we present a novel adaptive information sharing framework, called *Anser*, that has been implemented in the data warehousing service innovated by Alibaba Cloud, *AnalyticDB* . In this framework, we present the abstraction of dynamic information along with its applications. Such information could be generally used to improve the processing efficiency of query engines. The results of the experimental study also demonstrate the functionality of this framework.

REFERENCES

[1] [n. d.]. Apache Hive. https://hive.apache.org/. Last accessed 2023-03-01.

[2] [n. d.]. Apache Kafka. https://kafka.apache.org/. Last accessed 2023-03-01.

[3] [n. d.]. Elastic Compute Service. https://www.alibabacloud.com/product/ecs. Last accessed 2023-03-01.

[4] [n. d.]. HDFS Architecture Guide. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html. Last accessed 2023-03-01.

[5] [n. d.]. Impala Runtime Filtering. https://impala.apache.org/docs/build/html/topics/impala_runtime_filtering.html. Last accessed 2023-03-01.

[6] [n. d.]. Object Storage Service (OSS). https://www.alibabacloud.com/product/object-storage-service?spm=a3c0i.23458820.2359477120.2.26a77d3fagA3sE. Last accessed 2023-03-01.

[7] [n. d.]. Parameter Sensitive Plan optimization. https://learn.microsoft.com/en-us/sql/relational-databases/performance/parameter-sensitivity-plan-optimization?view=sql-server-ver16. Last accessed 2023-03-01.

[8] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European conference on computer systems*. 29–42.

[9] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 1383–1394.

[10] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J Green, Monish Gupta, Sebastian Hillig, et al. 2022. Amazon Redshift re-invented. In *Proceedings of the 2022 International Conference on Management of Data*. 2205–2217.

[11] Ron Avnur and Joseph M Hellerstein. 2000. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. 261–272.

[12] Shivnath Babu, Pedro Bizarro, and David DeWitt. 2005. Proactive re-optimization. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. 107–118.

[13] Srikanth Bellamkonda, Hua-Gang Li, Unmesh Jagtap, Yali Zhu, Vince Liang, and Thierry Cruanes. 2013. Adaptive and Big Data Scale Parallel Execution in Oracle. *Proc. VLDB Endow.* 6 (2013), 1102–1113.

[14] Chuangxian Wei Xiaoqiang Peng Liang Lin Sheng Wang Zhe Chen Feifei Li Yue Pan Fang Zheng Chengliang Chai Chaoqun Zhan, Maomeng Su. 2019. AnalyticDB: Realtime OLAP Database System at AlibabaCloud. In *Proceedings of the VLDB Endowment*, Vol. 12. 2059–2070.

[15] Surajit Chaudhuri and Kyuseok Shim. 1994. Including group-by in query optimization. In *VLDB*, Vol. 94. 12–15.

[16] Ming-Syan Chen, Hui-I Hsiao, and Philip S Yu. 1997. On applying hash filters to improving the execution of multi-join queries. *The VLDB journal* 6 (1997), 121–131.

[17] Ming-Syan Chen, Hui-I Hsiao, and Philip S Yu. 1997. On applying hash filters to improving the execution of multi-join queries. *The VLDB journal* 6 (1997), 121–131.

[18] Amol Deshpande. 2004. An initial study of overheads of eddies. *ACM SIGMOD Record* 33, 1 (2004), 44–49.

[19] Amol Deshpande, Joseph M Hellerstein, et al. 2004. Lifting the burden of history from adaptive query processing. In *VLDB*. Citeseer, 948–959.

[20] Amol Deshpande, Joseph M Hellerstein, and Vijayshankar Raman. 2006. Adaptive query processing: why, how, when, what next. (2006), 806–807.

[21] Jialin Ding, Umar Farooq Minhas, Badrish Chandramouli, Chi Wang, Yinan Li, Ying Li, Donald Kossmann, Johannes Gehrke, and Tim Kraska. 2021. Instance-optimized data layouts for cloud analytics workloads. In *Proceedings of the 2021 International Conference on Management of Data*. 418–431.

[22] David J. DeWitt Donovan A. Schneider. 1989. A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment. *1989 ACM SIGMOD international conference on Management of data)* (1989), 110–121.

[23] Mostafa Elhemali, César A Galindo-Legaria, Torsten Grabs, and Milind M Joshi. 2007. Execution strategies for SQL subqueries. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. 993–1004.

[24] Goetz Graefe. 1995. The cascades framework for query optimization. *IEEE Data Eng. Bull.* 18, 3 (1995), 19–29.

[25] Goetz Graefe and Karen Ward. 1989. Dynamic query evaluation plans. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*. 358–366.

[26] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon redshift and the case for simpler data warehouses. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 1917–1923.

[27] Ashish Gupta, Venky Harinarayan, and Dallan Quass. 1995. Aggregate-query processing in data warehousing environments. In *VLDB*, Vol. 95. Citeseer, 358–369.

[28] Joseph M Hellerstein, Peter J Haas, and Helen J Wang. 2007. 2007 Test-of-time Award "Online Aggregation". (2007), 1.

[29] Yannis E. Ioannidis, Raymond T. Ng, Kyuseok Shim, and Timos K. Sellis. 1997. Parametric query optimization. *The VLDB Journal* 6 (1997), 132–151.

[30] Zachary G. Ives and Nicholas E. Taylor. 2008. Sideways Information Passing for Push-Style Query Processing. *2008 IEEE 24th International Conference on Data Engineering* (2008), 774–783.

[31] Matthias Jarke and Jürgen Hartmut Koch. 1984. Query Optimization in Database Systems. *ACM Comput. Surv.* 16 (1984), 111–152.

[32] Navin Kabra and David J DeWitt. 1998. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*. 106–117.

[33] P-A Larson. 2002. Data reduction by partial preaggregation. In *Proceedings 18th International Conference on Data Engineering*. IEEE, 706–715.

[34] Allison W. Lee and Mohamed Zaït. 2008. Closing the query processing loop in Oracle 11g. *Proc. VLDB Endow.* 1 (2008), 1368–1378.

[35] Kaiyu Li and Guoliang Li. 2018. Approximate Query Processing: What is New and Where to Go? *Data Science and Engineering* 3 (2018), 379–397.

[36] Lothar F Mackert and Guy M Lohman. 1986. R* Optimizer Validation and Performance Evaluation. *Very Large Data Bases: Proceedings* 149 (1986), 149.

[37] Abhishek Modi, Kaushik Rajan, Srinivas Thimmaiah, Prakhar Jain, Swinky Mann, Ayushi Agarwal, Ajith Shetty, Shahid K I, Ashit Gosalia, and Partho Sarthi. 2021. New query optimization techniques in the Spark engine of Azure synapse. *Proceedings of the VLDB Endowment* 15, 4 (2021), 936–948.

[38] M. Oyamada. 2018. Accelerating Feature Engineering with Adaptive Partial Aggregation Tree. *2018 IEEE International Conference on Big Data (Big Data)* (2018), 5417–5419.

[39] Glenn Norman Paulley. 2001. *Exploiting functional dependence in query optimization.* University of Waterloo.

[40] Vijayshankar Raman, Amol Deshpande, and Joseph M Hellerstein. 2003. Using state modules for adaptive query processing. In *Proceedings 19th International Conference on Data Engineering (Cat. No. 03CH37405)*. IEEE, 353–364.

[41] Praveen Seshadri, Joseph M Hellerstein, Hamid Pirahesh, TY Cliff Leung, Raghu Ramakrishnan, Divesh Srivastava, Peter J Stuckey, and S Sudarshan. 1996. Cost-based optimization for magic: Algebra and implementation. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*. 435–446.

[42] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, et al. 2019. Presto: SQL on everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1802–1813.

[43] Tarique Siddiqui, Alekh Jindal, Shi Qiao, Hiren Patel, and Wangchao Le. 2020. Cost Models for Big Data Query Processing: Learning, Retrofitting, and Our Findings. *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (2020).

[44] Michael Stillger, Guy M Lohman, Volker Markl, and Mokhtar Kandil. 2001. LEO-DB2's learning optimizer. In *VLDB*, Vol. 1. 19–28.

[45] Michael Stonebraker. 1986. The case for shared nothing. *Database Engineering Bulletin)* (1986), 4–9.

[46] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. 2020. AnalyticDB-V: a hybrid analytical engine towards query fusion for structured and unstructured data. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3152–3165.

[47] Rongbiao Xie, Meng Li, Zheyu Miao, Rong Gu, He Huang, Haipeng Dai, and Guihai Chen. 2021. Hash Adaptive Bloom Filter. *2021 IEEE 37th International Conference on Data Engineering (ICDE)* (2021), 636–647.

[48] Yanjun Yao, Sisi Xiong, Hairong Qi, Yilu Liu, Leon M. Tolbert, and Qing Cao. 2015. Efficient Histogram Estimation for Smart Grid Data Processing With the Loglog-Bloom-Filter. *IEEE Transactions on Smart Grid* 6 (2015), 199–208.

[49] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 15–28.

[50] Yong Zhao and Rong Chen. 2021. Spark SQL Query Optimization Based on Runtime Statistics Collection. *2021 IEEE 6th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA)* (2021), 250–255.