ORION: Optimizing OLAP Query Execution with Proactive Caching and Separate Operators

Zhixin Tong*
Shanghai Jiao Tong University
Shanghai, China
up2018@sjtu.edu.cn

Jiuchen Shi*
Shanghai Jiao Tong University
Shanghai, China
The Hong Kong Polytechnic University
Hong Kong, Hong Kong
shijiuchen@sjtu.edu.cn

Quan Chen†
Shanghai Jiao Tong University
Shanghai, China
chen-quan@cs.sjtu.edu.cn

Pu Pang
Shanghai Jiao Tong University
Shanghai, China
pangpu@cs.sjtu.edu.cn

Shixuan Sun
Shanghai Jiao Tong University
Shanghai, China
sunshixuan@sjtu.edu.cn

Jie Meng
Huawei Cloud
Chengdu, China
mengjie09@huawei.com

Jiang Liu
Huawei Cloud
Chengdu, China
liujiang56@huawei.com

En Shao
Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
shaoen@ict.ac.cn

Minyi Guo
Shanghai Jiao Tong University
Shanghai, China
guo-my@cs.sjtu.edu.cn

Abstract

Current work leverages data caching and operator execution accelerations to reduce the Online Analytical Processing (OLAP) query execution time on the disaggregated architecture with computation, cache, GPU, and storage clusters. However, their optimizations rely heavily on the OLAP engine, thus have defects of passive data fetching and integrated operator executions, leading to poor OLAP query execution performance. To resolve the above problems, we propose the **ORION** manager to take over the data and operator management capabilities from the OLAP engine for reducing OLAP query execution time. ORION consists of a *proactive cache manager* and a *fine-grained operator manager*. The cache manager reorders the data batch processing based on the cache availability and asynchronously fetches uncached data. The operator manager separates the integrated pushdown operators into executable functions, to enable properly scheduling on CPU or GPU nodes and allocating fine-grained resources. Experimental results show that ORION reduces the OLAP query execution time by 42.2% on average compared to state-of-the-art works.

CCS Concepts

• **Computer systems organization** → **Cloud computing**.

Keywords

OLAP, Disaggregated Architecture, Caching, Scheduling

ACM Reference Format:
Zhixin Tong, Jiuchen Shi, Quan Chen, Pu Pang, Shixuan Sun, Jie Meng, Jiang Liu, En Shao, and Minyi Guo. 2025. ORION: Optimizing OLAP Query Execution with Proactive Caching and Separate Operators. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3721145.3725752

*Zhixin Tong and Jiuchen Shi contributed equally to this work.
†Quan Chen is the corresponding author.

ICS '25, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3725752

1 Introduction

Online Analytical Processing (OLAP) refers to large-scale data analysis in areas like financial reporting, sales analysis, and customer behavior studies [8, 10]. Unlike Online Transaction Processing (OLTP) with frequent data writes and updates, OLAP emphasizes periodic data reading and analysis (e.g. daily) that often take minutes to hours [49, 60]. The OLAP queries are typically performed on the disaggregated architecture in which the hardware resource can be divided into four layers, including computation, cache with CPUs, GPU for acceleration, and storage [30, 38, 41, 62]. This architecture allows data to be stored remotely using cloud services like Amazon S3 [46] and Alibaba Cloud OSS [11]. OLAP engines such as Apache Spark [19], Presto [22], and Trino [23] divide large-scale data into batches and use CPUs and GPUs for parallel processing with the support of the disaggregated architecture.

Figure 1 shows the OLAP query execution workflow in the disaggregated architecture, including two main stages: data fetching and operator execution. For a query, the OLAP engine will first fetch a batch of required data from remote storage. Then, the data batch is processed with multi-staged operators that are packaged into a single process on the local servers. The above process is repeated until all data batches are processed. Several optimization techniques have been proposed to accelerate the OLAP query execution in both the data fetching and operator execution stages.

For the data fetching, since fetching a large amount of data from remote storage can slow down the OLAP query execution, caching methods like Alluxio deploy a cache cluster close to the OLAP engine to enhance data reuse and reduce data transmissions [25, 54]. For the operator execution, current OLAP engines [19, 22] allocate CPU cores for operators integrated in one process, and some of them (like Spark-RAPIDS [38]) offload all operators to GPUs to accelerate executions. Although speeding up the data fetching and operator executions through caching and GPUs, the above methods still have poor efficiency due to their strong dependence on the OLAP engine's management capabilities.

As for the data fetching aspect, some data batches available in the cache may be fetched repeatedly since the OLAP engine determines the data batch processing order without awareness of cache availability. Moreover, according to the OLAP engine's management, a data batch is only fetched after the previous data batch is processed, leading to synchronized data fetching and processing with long data waiting time. As for the operator execution aspect, our investigations show that some operators are IO-intensive while others are compute-intensive (Section 6.3.1). However, current OLAP engines integrate operators into a single process, and allocating CPU cores to them can result in some operators

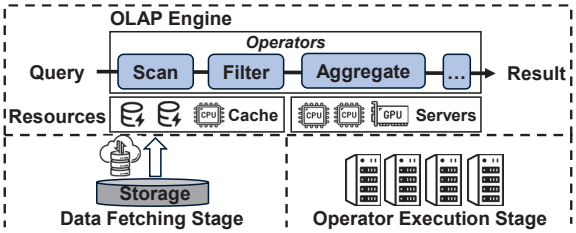


Figure 1: The execution workflow of the OLAP query.

obtaining insufficient resources. Moreover, when offloading all integrated operators to GPUs, the GPU memory can be overwhelmed due to IO-intensive operators loading a large amount of data into GPU memory while their performance on GPUs is similar to CPUs.

Even if combining all the above optimizations, it still cannot achieve optimal OLAP query execution performance. This is because it does not address the fact that the above optimizations are still reliant on the centralized management of the OLAP engine. Our investigations show that combining them still increases the query execution time by 46.5% on average compared to the Oracle execution time (i.e., eliminating data fetching time and using excessive GPUs for the operator to achieve the ideal execution performance).

To address above problems, we choose to separate the data and operator management responsibilities from the OLAP engine by introducing a dedicated manager. Since real-world big data analysis often requires multiple OLAP engines simultaneously to address the complexity of tasks [15, 57], this separation eliminates the invasive modifications when adapting to various engines. In this way, the OLAP engine remains in the computation cluster, but our manager, who determines data fetching and operator executions, is deployed in the cache layer to conveniently process hot data batches.

With the above design choices, more flexible data batch fetching and operator scheduling strategies can be utilized to speed up OLAP query executions. We summarize three key design guidelines for efficient OLAP query executions.

Firstly, strategies are required to reduce repeated data fetching and mitigate data processing blockage. Secondly, an execution mode is required to separate the integrated operators to allow the respective scheduling and resource allocation for each operator. Lastly, policies are required to schedule proper operators to GPUs and allocate just-enough resources for different operators, to avoid GPU memory overloading and improve OLAP query execution performance.

We therefore propose the **ORION** manager to optimize the OLAP query execution time[1]. ORION is positioned as an acceleration plugin for universal OLAP engines with little

[1]ORION is open-sourced via https://github.com/Tongzhixin/ORION.

intrusions. It comprises a *proactive cache manager* and a *fine-grained operator manager*. The cache manager reorders the data batch processing orders based on data availability in the cache, increasing the cache hit rate and reducing repeated data fetching. Moreover, it fetches the required data batches along with operator executions asynchronously to reduce the data waiting time. The operator manager has a function builder and a function scheduler. The function builder utilizes a novel execution mode to separate the integrated pushdown operators into executable functions. The function scheduler dispatches the functions to cache nodes with CPU cores or GPU nodes, considering the functions' diverse hardware resource demands. It also gradually searches just-enough resources for functions during the query execution process.

This paper makes three contributions.

- **Comprehensive analysis of inefficiencies in current OLAP query executions.** The analysis identifies the need for a dedicated manager to take over the data and operator management and corresponding optimization challenges.
- **Proactive reordering for data batch processing based on their availability in the cache.** This reordering strategy eliminates repeated data fetching and unnecessary data waiting time during the OLAP query execution.
- **Separating integrated operators into independent functions for flexible scheduling.** This enables the scheduling of proper operators to GPUs and allocates optimal resources for operators to speed up operator execution performance.

We have evaluated ORION on our testbed cluster with 4 servers and 2 NVIDIA A40 GPUs, and also deploy it for our real business on our production cluster with 600 servers and 100 GPUs. Experimental results show that ORION averagely reduces the query execution time by 42.2% and 60.8% compared to state-of-the-arts for testbed and production environments, respectively.

2 Related Work

Caching: Executing OLAP queries in a disaggregated storage architecture involves significant data movement [9]. To address this problem, Alluxio [5, 54] and JuiceFS [25] serve as cache components between the OLAP engine and remote storage, fetching the data required by the OLAP engine from remote storage to cache. Rubix [26] and Ceph [12] implemented similar functions but designed different optimizations on the cache. Moreover, FPDB [61, 62] utilizes the local cache for hot data batches to reduce data movement. However, these works processed data batches according to the OLAP engine's decision, leading to repeated data fetching and low hit rates with limited cache size.

Operator Executions: Current OLAP engines primarily execute operators using CPU cores within a single process, such as Executors in Spark [19]. Some other works like columnar execution [1] and SIMD-based optimizations [27] have improved the execution performance with CPU cores for OLAP queries. For example, DuckDB [43] leveraged an efficient in-memory, single-process design for faster query execution. However, these works allocated CPU cores for operators integrated in a single process, in which some operators may not be allocated enough resources, leading to low query execution performance.

GPUs are utilized to accelerate operator executions due to their parallel capabilities [24, 28, 35]. RRAPIDS [38] was developed on top of Spark to accelerate operator executions [19]. Ghive [32, 33] and OAs [64] offered acceleration libraries for Apache Hive [56] and Pandas [34], respectively. Some other works like Crystal and HeavyDB [6, 7, 31, 39, 42, 48] focused on enhancing data processing capabilities in DBMS but operate similar to RAPIDS. Moreover, some previous works [59, 65] observed that large-scale data computations could lead to insufficient GPU memory. Lee et al. [29] found that data analysis with IO-intensive operators is not suited for GPUs. The above works offload all operators integrated in a single process onto GPUs, which can result in GPU memory exhausted and poor OLAP query execution.

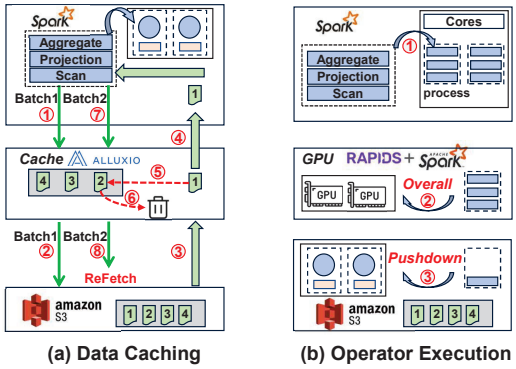
Computation Pushdown: It is a common idea for near-data processing [58]. PushdownDB [63] pushed down operators using S3 select SparkNDP [44] employed SQLite [13] to execute the SQL in remote storage. However, current remote storage [11, 46, 47] like S3 and OSS [11] only supported limited types of pushdown operators. These approaches relied on the computational capabilities of third-party remote storage, which only supported limited pushdown operators. Thus, the OLAP query execution performance was poor.

3 Background and Motivation

In this section, we first introduce the background of executing the OLAP query on the disaggregated architecture, and then investigate the inefficiency of current methods.

3.1 OLAP Query Execution on the Disaggregated Architecture

Current datacenters have evolved into disaggregated compute and storage clusters to enable flexible resource scaling [30, 38, 41, 50, 62]. To reduce the data fetching between compute and storage, the caching cluster (with large memory and CPU cores) is also deployed for caching data locally. Moreover, some works [7, 59, 65] integrated high-performance GPUs to accelerate the executions. Under this architecture, data caching and operator execution accelerations are utilized, but have poor query execution efficiency.



(a) Data Caching (b) Operator Execution

Figure 2: Optimizations for the OLAP query executions.

Firstly, current caching methods like Alluxio [5] passively fetch data from remote storage according to the OLAP engine, leading to data waiting and repeated data fetching. As an example in Figure 2(a), the data "Batch1" is only fetched when the Spark engine determines to process it, resulting in waiting for the data (①-④). Moreover, based on the batch processing order of Spark, the already cached "Batch2" may be eliminated to cache the new data "Batch1" for processing when the cache space is insufficient (⑤-⑥). It will be re-fetched when the Spark engine starts to execute it after (⑦-⑧), which enlarges the query execution time.

Secondly, as shown in Figure 2(b), current OLAP engines integrate the operators into a single process, which allocates computing resources as a whole without considering the operators' different resource demands (①). Moreover, some OLAP engines utilized GPU accelerated frameworks (e.g., Spark-RAPDIS) to speed up the OLAP operator executions [3, 38]. However, these frameworks offload all operators on GPUs, but the limited GPU memory can be overloaded due to IO-intensive operators loading a large amount of data. This results in reduced execution performance in turn (②).

Some other works [45, 46, 62, 63] also pushdown operators directly to the third-party remote storage for executions (③ in Figure 2(b)). This is implemented by using the specific modules that handle data transfer between compute and storage and determine pushdown operators, e.g., the Connector in Presto [21] and DataSource in Spark [20]. However, they only support limited formats (e.g., CSV) and pushdown operator types, as they rely highly on remote storage capabilities.

3.2 The Long Query Execution Time

In this section, we investigate the inefficiencies of current solutions for OLAP query executions.

The investigations are conducted by using the top-5 queries (marked as Q1, Q5, Q17, Q18, and Q21) with the typical operators and long execution time in TPC-H [14], which is popular in evaluating the performance of OLAP engine [16].

Table 1: Experiment specifications

	Specifications
Hardware	**Compute layer:** Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz, 32 CPU cores, 48GB Memory
	Cache layer: Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz, 8 CPU cores, 64GB Memory
	GPU layer: NVIDIA A40, CUDA Version: 12.2, PCI-e 4.0, 24GB Memory ×2
	Storage layer: Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz, 4 CPU cores, 8 GB Memory, 1Gbps network bandwidth
Software	Ubuntu 20.04.6 LTS with kernel 5.15.0-88-generic x86_64 spark-3.3.1, presto-server-0.287, rapids-4-spark_2.12-24.06.0 Alluxio-2.9.0, MinIO-RELEASE.2024-06-22T05-26-45Z

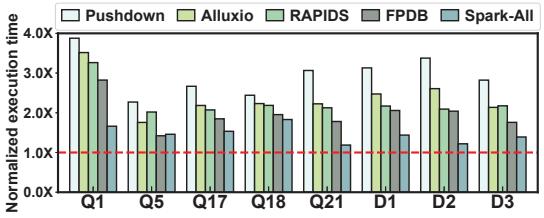


Figure 3: The execution time of OLAP queries with *Spark-Pushdown*, *Spark-Alluxio*, *Spark-RAPIDS*, *FPDB*, and *Spark-All* normalized to *Oracle*.

As a supplement, we evaluate three queries widely used in financial stock analysis in our production datacenters (marked as D1, D2, and D3), containing the distinct operators that are not present in TPC-H queries. We provide the unified TPC-H benchmark dataset with a scale factor of 100 for all queries fairly. The file remote storage format used is parquet [18], and each file size is 256MB. Table 1 summarizes the hardware and software configurations. We evaluate the query execution time on the OLAP engine Spark with state-of-the-arts.

Firstly, We deploy the S3-compatible object store MinIO [36] to form *Spark-Pushdown* with S3-Select. Secondly, we deploy the cache method Alluxio [5] with Spark to form *Spark-Alluxio*. Thirdly, we utilize one GPU card with the GPU accelerating framework RAPIDS [38] for Spark to form *Spark-RAPIDS*. Moreover, we deploy the state-of-the-art work FPDB, which leverages a local cache to store hot data batches and integrates pushdown computation to offload operators to S3-Select [61]. Since it is infeasible to pushdown operators to both remote storage and GPUs, there is no solution that combines caching, pushdown, and GPU acceleration at the same time. Thus, we enumerate the execution time of different operators in FPDB (executed on CPU cores) and RAPIDS (executed on GPUs), and sum up the minimum values for each operator to be the query execution time as *Spark-All*.

We further construct an *Oracle* by caching all data directly in memory and using 2 GPU cards for operator executions, completely eliminating data fetching time and achieving the most ideal execution performance. Figure 3 shows the query

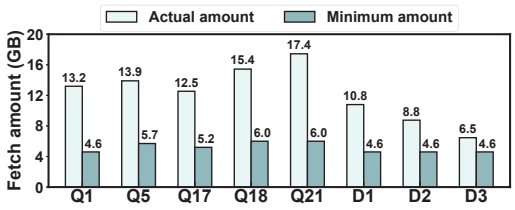


Figure 4: The data fetch amount from remote storage and the minimum amount for executing the queries.

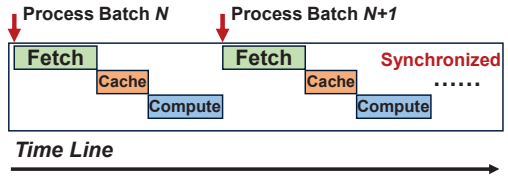


Figure 5: Synchronized data batch fetching problem.

execution time of different methods normalized to *Oracle*. We find that *Spark-All* has the best performance compared to others since it combines all the optimization methods for OLAP query executions. However, we also observe that all the query execution time of *Spark-All* is bigger than 1X, with the increase from 1.2X to 1.9X. The query execution inefficiency will be magnified on our real-world business queries with 400X larger data scale than TPC-H, which we will show their results in Section 8.8.

The OLAP query execution inefficiency of above works is caused by relying on the centric data and operator management capabilities of the OLAP engine.

3.3 Diving into the Underlying Reasons

Our investigation shows the long OLAP query execution time is caused by two major aspects: *passive data processing and fetching* and *integrated scheduling for operators*.

3.3.1 Passive Data Processing and Fetching. The first reason is the passive data fetching of current cache methods like Alluxio, based on the decision of the OLAP engine.

Figure 4 shows the actual data fetching amount from the remote storage and the minimum amount for executing the eight queries when the cache cluster is full with 80% of the required data of each query. The minimum amount of each query represents the least data amount required in execution. We can observe that the fetched data amount with the Alluxio method is higher than the minimum amount in all the queries, with an average increase of 2.3X. This is because the data batch processing order is determined by the OLAP engine, without flexible reordering based on the data availability in the cache. In this mode, cache miss occurs frequently and results in repeated fetching of the data available in the cache.

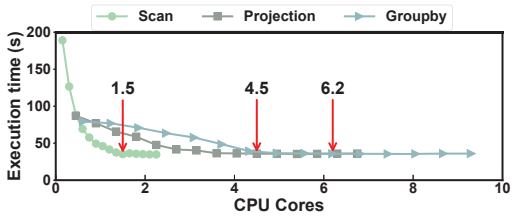


Figure 6: The execution time of three operators in Q1 under varying CPU core allocation.

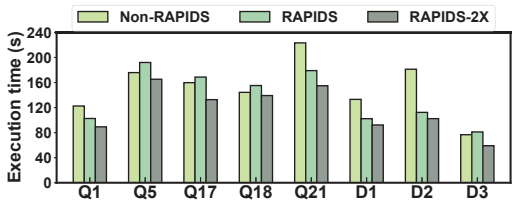


Figure 7: The execution time of all queries with Spark-Non-RAPIDS, Spark-RAPIDS, and Spark-RAPIDS-2X.

Moreover, as illustrated in Figure 5, for fetching the required data of a query from the remote storage, the cache cluster passively fetch the data batch required after the OLAP engine prepares to process it. In this mode, the data fetching and operator execution occur in a synchronized manner, which leads to an increase in the total query execution time.

3.3.2 Integrated Scheduling for Operators. In the operator execution stage, since some operators are compute-intensive while others are IO-intensive, allocating resources for all operators integrated into a single process can lead to OLAP query execution inefficiency.

On the one hand, current OLAP engines [19, 22] typically allocate CPU cores for integrated operators in a single process. Figure 6 shows the execution time of the Scan, Projection, and Groupby operators in Q1 when we gradually increase their CPU cores. We observe that execution time decreases as CPU cores increase, and no longer decreases after reaching certain thresholds, which vary as 1.5, 4.5, and 6.2 CPU cores, respectively. These results demonstrate that different operators have varying resource demands. When allocating resources for them with the same resources in a process, some operators may be over-provisioned while others are allocated insufficiently, resulting in decreased execution performance.

On the other hand, current GPU-accelerated methods like Spark-RAPIDS [38] schedule all integrated operators onto GPUs. Figure 7 compares the query execution time of Spark with RAPIDS and Spark without RAPIDS. RAPIDS performs worse for Q5, Q17, and Q18, while it offloads all operators onto GPUs. Looking into the GPU resource usage, the GPU utilization of RAPIDS is only 9.8% on average, while the GPU

memory is fully utilized. This is because some IO-intensive OLAP operators (e.g., select and filter), which do not demand many CUDA cores, exhaust GPU memory due to the large amount of data loaded. To further test our investigation, we enhance RAPIDS by utilizing twice the number of GPU cards as RAPIDS-2X, which increases the available GPU memory. As shown in Figure 7, Q5, Q17, and Q18 indeed achieve shorter query execution time compared to RAPIDS.

The above observations suggest that we should separate the integrated operators into functions and then decide their respective scheduling to GPUs or CPUs, as well as the allocated computing resources for each.

4 ORION Methodology

In this section, we first introduce the design choices and guidelines of ORION, and then explain the system overview of ORION.

4.1 Design Choices and Guidelines

The inefficiency of the OLAP query execution optimizations stems from the tight integration of data and operator management capabilities with the OLAP engine. Therefore, We introduce a well-designed manager to take over both the data fetching and operator execution management capabilities, allowing flexible batch processing and execution management. Based on Section 3.3, we summarize three key design guidelines for efficient OLAP query executions.

ORION should proactively re-order data batch processing based on data cache availability, and fetch data during query executions asynchronously. The passive data processing and fetching based on the OLAP engine can cause repeated data fetching and synchronous data waiting.

A fine-grained execution mode is required to separate operators. This is because the integrated execution mode for pushdown operators cannot enable fine-grained scheduling and resource allocation, leading to poor performance and resource inefficiency.

Efficient strategies are required to allocate just-enough CPU cores to different operators and schedule proper operators to GPUs. Allocating CPU cores to integrated operators may lead to some operators cannot obtain enough resources, and offloading all of them to GPUs may exhaust the limited GPU memory, which decreases OLAP query execution performance.

4.2 System Overview

Figure 8 shows the overview of ORION that comprises a *proactive cache manager* and a *fine-grained operator manager*. The cache manager reorders the data batch processing order based on availability in cache, to increase the cache hit and

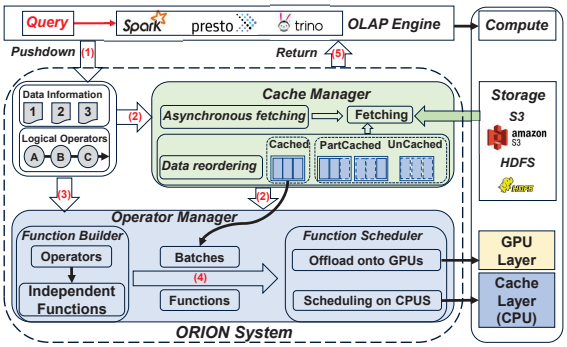


Figure 8: Design overview of ORION.

reduce repeated data fetching, and asynchronously fetches the required data batches.

The operator manager consists of two components: the function builder and the function scheduler. The function builder utilizes a fine-grained execution mode to package the pushdown operators into functions, which enables each operator to be fine-grained allocated resources. The function scheduler dispatches functions to CPU or GPU nodes and allocates just-enough resources for them.

For the cache manager, the key point is to proactively process and fetch data batches without relying on the decisions of the OLAP engine. We address this by integrating the data reordering and asynchronous fetching strategies. The reordering strategy reorders the data batch processing orders based on their cache availability, to avoid repeated fetching and enhance data reuse during query executions (Section 5.2). The fetching strategy fetches the not fully cached data batches along with operator executions in an asynchronous mode to reduce data waiting time (Section 5.3).

For the operator manager, since the pushdown operators of the OLAP engine are integrated, we propose a function builder to package each pushdown operator into an independent function. The packaged functions can flexibly adapt to various data formats (e.g., Parquet and CSV), support operators on both CPU and GPU hardware, and communicate efficiently with each other through pipes (Section 6.2). Moreover, the function scheduler assigns functions to either CPU or GPU nodes and gradually adjusts just-enough CPU cores for optimal function execution. The challenging part here is to offload proper functions to GPUs and efficiently adjust resources for different functions (Section 6.3).

Specifically, for executing an OLAP query, ORION works as follows. 1) ORION obtains the pushdown operators and required data batches from the daemon process, which is an extension of the OLAP engine's Connector. 2) The cache manager's data reordering strategy sorts the required data batches based on their cache availability and marks them as Cached, PartCached, and UnCached. If the cache cluster

is not full, the asynchronous fetching strategy fetches the PartCached and UnCached data batches from remote storage. 3) Meanwhile, the operator manager utilizes the function builder to package operators into independent functions. The function scheduler determines the scheduling locations (CPU or GPU) of these functions and allocates each function with initial computing resources. 4) The function scheduler executes the functions with data batches according to the order determined by the cache manager. It also continuously adjusts the just-enough resources for functions with different resource demands, and reschedules functions to CPU nodes when the limited GPU memory exhausts. 5) After each data batch is processed by all functions, the results are returned to the OLAP engine to conduct subsequent computations.

It is worth noting that ORION does not determine to push down which operators. This is the responsibility of the OLAP engine, and ORION focuses on efficiently managing the push-down operators on different hardware resources.

5 Proactive Cache Management

Based on our investigations in Section 3.3.1, we propose the proactive cache manager in this section. We first introduce the design choices of caching. Then, we propose data reordering and asynchronous fetching strategies to address the problems of passive data processing and fetching.

5.1 Caching Choices

Cached Data and How to Know It in Advance. The cache manager fetches the source data needed for the computation of the OLAP query, rather than the intermediate data during the query execution process. Moreover, along with the pushdown operators, the OLAP engine's Connector [20] also provides the data information. This information ensures that the cache manager knows where the data is and what format and partition it is.

Caching Format. For the data format of the OLAP query in the cache, most of the existing methods (e.g., Alluxio) [5, 25] fetch data from remote storage and store them in the cache with the granularity of files. In this mode, since the query may only need several columns of a file, redundant data will be fetched, which wastes the cache and occupies considerable network bandwidth. Therefore, we only fetch the required columns of the OLAP query and convert the different data formats (e.g., Parquet and CSV) into columnar format to cache. Since the number of rows in the file is not fixed, we further split each data column into batches, each with 1M rows. Each batch is uniquely identified by the file name, column name, and start and end rows.

Cache Replacement. For each data batch, the cache manager also marks a label to represent whether it has been processed by the function scheduler (Section 6.3). When the new data columns need to be fetched from the remote storage and the memory space of the cache layer is insufficient, we employ the Least Recently Used (LRU) policy to prioritize the eviction of processed data batches. If no batches can be eliminated, all cached batches are queued for processing, and the manager will stop fetching until some batches are processed. After the query execution is completed, all data batches in the cache will be marked back as unprocessed.

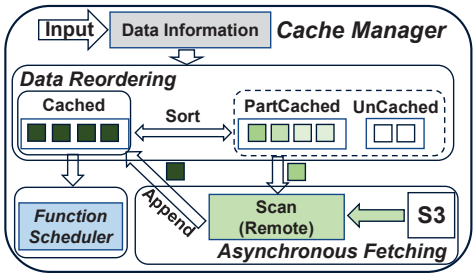


Figure 9: The design of the cache manager.

5.2 Data Batch Processing Reordering

To address the problem of passive data processing, the key idea of this strategy is to give priority to processing the batches that already exist entirely in the cache.

When the cache manager receives the data information with the file list and the column set used by this query, it first searches for all the existing columns in the cache. The cached columns that identify the same file, the same start row number, and the same end row number are assembled into data batches. These batches can be executed by operators in any order without incurring context-switching overhead, as they follow the same workflow and ensure correct results. Moreover, the cache manager does not account for the reordering of intermediate data since it stays localized within the operator pipeline.

As shown in Figure 9, the cache manager classifies the data batches into "Cached", "PartCached", and "UnCached" groups, indicating that all, some, or none of the required columns are in the cache, respectively. For each group, the data batches are also ordered according to the total cached column numbers. The data batches in the "Cached" groups are prioritized to process by the function scheduler (Section 6.3). In this way, when the space is insufficient to cache all data, we can increase the hit rate and reduce the repetitive data fetching, further enhancing the performance.

5.3 Asynchronous Fetching

When none or part of the required data is in the cache, we need to fetch the missing data batches from remote storage.

As shown in Figure 9, for "PartCached" and "UnCached" groups, we design to fetch the required data columns in

an asynchronous manner, independent of the batch execution order decided by the OLAP engine. During the data fetching, we design to fetch the missing columns first for the data batches in the "PartCached" group and then for the "UnCached". The data fetching order follows the data batch processing order within each group as determined by the data reordering strategy. These methods quickly convert more data batches to the "Cached" state, reducing data waiting during operator executions.

6 Fine-grained Operator Management

Based on the investigations presented in Sections 3.3.2, we propose a fine-grained operator manager in this section. We first introduce the brief overall workflow of the operator manager and then delve deeper into the internals of both the function builder and the function scheduler.

6.1 Overall Workflow

We first introduce the overall workflow of the OLAP query execution with the operator manager.

Input of the Operator Manager: The input of the operator manager is the logical pushdown operators from the OLAP engine at the computation layer. Specifically, upon a query's arrival, the OLAP engine parses and optimizes it, and pushdown operators are dispatched to the connector (e.g., DataSource v2 in Spark [20]).

Query Execution at Cache and GPU Clusters: The function builder first converts the logical pushdown operators into executable processing functions based on the operator characteristics. The function scheduler then schedules the functions onto different cache and GPU nodes and executes them with data batches according to the optimal order determined by the cache manager. During the query execution, it reschedules data batches back to cache nodes when GPU memory exceeds capacity, and adjusts the resource allocation dynamically.

Output of the Operator Manager: After a data batch is scheduled and executed by the operator manager, the result batch will be returned to the OLAP engine at the computation layer, which will continue the subsequent computations.

6.2 Function Builder

The key to addressing the integrated resource allocation for operators is to break them down into independent functions and execute them using a new computation mode.

As shown in Figure 10, to obtain independent functions, we first implement a function breakdown library named *FuncOpLib* to convert the operators into executable functions (①). The FuncOpLib is a set of implementations of logical operators. Each "FuncOp" is an internal implementation of all or a part of the operator as a more fine-grained execution step.

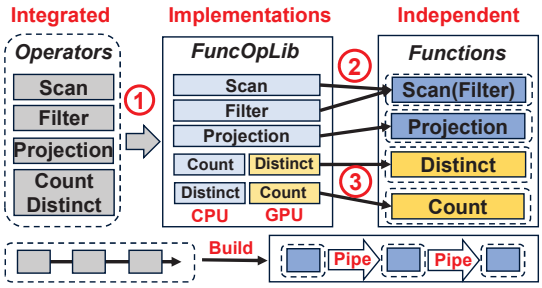


Figure 10: The design of the function builder.

The FuncOpLib also includes the different implementations on different hardware resources (i.e., CPU and GPU) to enable the functions to run on them.

When separating the operators, the Scan and Filter can be combined into a function to achieve better execution efficiency in some conditions (②). For example, when processing data in a columnar storage format, combining scan and filter takes advantage of the fast random read capability of columnar storage [2]. Moreover, if an operator includes multiple steps to run step by step in the execution process, we will break it down into more fine-grained functions, consisting of each step executed. For example, the CountDistinct operator is divided into two sequential functions, i.e., Distinct and Count, because only after knowing all deduplicated data can the correct result be counted (③).

After the above steps, the integrated operators that are executed in a process are built into independent executable functions with independent processes. Another issue is that these functions are now executed under the synchronized mode, i.e., a function waits for its downstream functions to finish execution. Therefore, we design a pipe between every two functions. The data completed by the previous function is placed in the pipe, and the current function polls the pipe to access the data and process it. In the producer-consumer model, each function execution is independent of the previous function. The above methods provide an architectural possibility for allocating fine-grained resources for different functions.

6.3 Function Scheduler

For the function scheduler, we first discuss about offloading appropriate functions to GPUs to speed up the OLAP query execution, and then introduce the efficient function scheduling on the cache nodes with CPU cores.

6.3.1 Function Offloading onto GPUs. In Section 3.3.2, we find that not all queries can be accelerated by GPUs in all cases. We conduct profiling for different functions (or operators) to determine which functions to offload to the GPU. We use Spark and RAPIDS as cases without GPU usage and

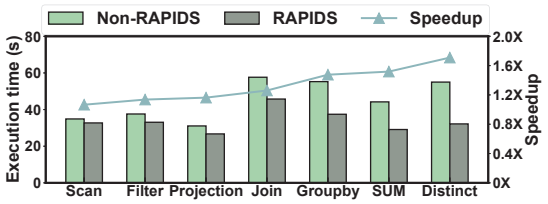


Figure 11: Function execution time on CPUs or GPUs.

with full GPU usage, and mark them as Non-RAPIDS and RAPIDS, respectively. We test a total of these operators, including Scan, Filter, Projection, Join, Groupby, and Aggregate (Sum and Distinct), which are common operators in TPC-H and real-world queries. The queries used here only contain one operation to ensure fairness. The configurations in the test are still the same as in Section 3.2.

Figure 11 shows the different execution time for each operator in Non-RAPIDS and RAPIDS, including the data transfer time via PCIe, and the speedup compared with the case of Non-GPU. We find that Aggregate (Sum and Distinct) and Groupby operators show substantial performance improvements when executed on GPUs. Compared with Non-RAPIDS, they have 71.7%, 52.1%, and 48.0% performance improvements, respectively. The Scan, Filter, and Projection operators, however, only increase the performance by 6.4%, 12.3%, and 16.2%, respectively.

This also aligns with our motivation in Section 3.3.2. In detail, the IO-intensive operators often execute early in queries with a large amount of data that can bring the GPU memory overhead, while compute-intensive operators are better suited for GPU execution. Based on these observations, the function scheduler is designed to selectively offload specific functions to GPUs, including Aggregate and Groupby, ensuring that these functions to benefit from GPU accelerations. The functions on GPUs will share the GPU resources.

For a function that requires GPUs, the intermediate data processed by its previous functions will be sent to the GPU node with the most remaining GPU memory for executions. The intermediate data between functions on the CPU and GPU is also passed through the pipe (Section 6.2). Moreover, if the GPU memory of all GPU nodes is insufficient, the subsequent to-be-processed batches on the GPU will be rescheduled for execution on cache nodes with CPU cores until the GPU nodes have sufficient memory again.

6.3.2 Function Scheduling on CPUs. We further consider the function scheduling on the cache nodes that utilize the CPU cores. According to Section 3.3.2, different functions require varying amounts of computing resources. By fully utilizing the separated mode described in Section 6.2, the goal is to adjust just-enough resources (CPU cores) for functions during the query execution.

Algorithm 1 Just-enough Resource Adjustment

```
1:  Tolerance ← 0.5, Rate ← 0.1
2:  function_list ← list of functions in execution order
3:  for function in function_list do
4:      pipe ← get_pipe(previous_function, function)
5:      num_batches ← pipe.size()
6:      function_concurrency ← function.concurrency()
7:      load_factor ← (num_batches / function_concurrency) -1
8:      if (load_factor + Tolerance) < 0 then
9:          function.reduce_resources(Rate)
10:     else if (load_factor - Tolerance) > 0 then
11:         function.increase_resources(Rate)
12:     else
13:         continue
14: Return isFinish
```

The initial resource allocation for functions is not always optimal, and adjusting just-enough resources gradually for functions requires obtaining the data pending amount and target concurrency. With the integrated function mode, the data generated by the previous function will be placed in the pipe, and the amount of batches serves as one input for resource adjustment. Another input is the user-defined target concurrency in the OLAP engine [40], which typically defaults to the allocated CPU number for use. For each function, the dynamic resource adjustment can be formulated as minimizing the difference between its data pending amount and target concurrency (i.e., size of the pipe).

We employ Algorithm 1 to achieve the optimization goal above. First, we initialize the adjustment $Tolerance$ and the adjustment $Rate$, and then retrieve the list of functions to be adjusted (lines 1-2). Similar to other works [51, 52], $Tolerance$ is proposed to prevent overly frequent resource adjustments, ensuring adjustments are only made within the tolerance range. For each function, we get the number of batches in the pipe and obtain the function's concurrency, then calculate the load factor as the ratio of the difference between them (lines 4-7). If the load factor plus $Tolerance$ is less than 0, we decrease the function's resources at the percent of $Rate$. If the load factor minus $Tolerance$ is greater than 0, we increase the function's resources. Otherwise, no adjustment is made. After all functions are processed, this round of resource adjustment is finished. This resource adjustment interval is set as 2 seconds in our experiments.

7 Implementation of ORION

In this section, we describe how ORION is implemented into the OLAP engine in our real production environment.

7.1 Data and Operator Acquisition

Current OLAP engines provide the connector API (e.g., DataSource v2 in Spark [20] and Connector in Presto [21]) for

connecting the computation and storage clusters. The OLAP engine offers registration of handlers for logical pushdown operators [21]. When the OLAP engine generates an execution plan, the registered operators are captured and handed over to the connector for processing.

By utilizing the capability of the connector API, ORION cleverly runs a daemon process for sending pushdown operators and data requirements from the OLAP connector to ORION as logical operators and data information. To enable ORION to separate from the daemon process on the OLAP side, we utilize the open-source Substrait [53], which provides a well-defined and cross-language specification for data computing operations. In this way, ORION can focus solely on parsing and processing the Substrait language, and support all common operators.

With the above implementations, ORION provides unified acceleration capabilities for OLAP engines without making intrusive modifications. Moreover, ORION can adapt to various types of storage clusters, e.g., AWS S3 and HDFS.

7.2 Cache and Function Execution

With the delegated management capabilities from the OLAP engine, ORION uses Ray's in-memory object store [37] as the infrastructure of the distributed cache layer, which is also responsible for synchronization among functions intra-server and inter-server. Moreover, taking full control of the data in ORION, data formats like CSV and Parquet are also converted to Apache Arrow format [17] and stored in a compressed column-based way, to retrieve data faster and save space. When functions start, the retrieved data is converted into batches using zero-copy techniques of Arrow and sent to the function scheduler.

For the function execution, ORION encapsulates the basic API of cuDF [3] to build functions that are easier to call on GPUs. By leveraging C++-compiled libraries from Apache Arrow, ORION inherits the performance advantages of compiled code and utilizes CPU SIMD instructions to achieve parallelism, enhancing performance compared to traditional JVM-based methods. Moreover, ORION utilizes Ray's Actor mechanism [55], resource management, and powerful scaling performance to implement function scheduling and resource adjustment across clusters. This ensures a consistent runtime environment across nodes in the cluster. ORION also leverages the NodeAffinityStrategy to effectively isolate the GPU cluster from the cache cluster, ensuring that only specific functions are scheduled to the GPUs.

8 Evaluation of ORION

In this section, we first evaluate ORION on query execution time and resource usage efficiency. Then, we investigate the effectiveness of each individual module and the scalability.

8.1 Evaluation Setup

Table 1 already shows the configurations of the experiment platform. We use the TPC-H benchmark with a scale factor of 100. The file storage format is parquet, and each file size is 256MB. The top five time-consuming queries with typical operators from the TPC-H are used: Q1, Q5, Q17, Q18, and Q21. As a supplement, we evaluate three queries widely-used in financial stock analysis in our production datacenters (marked as D1, D2, and D3) detailed in Section 3.2.

We compare ORION to several real-world solutions and academic works on two widely-used OLAP engines Spark [19] and Presto [22], respectively. For real-world solutions, we adopt the S3 Select pushdown method (marked as Pushdown) [44, 63], the caching method Alluxio [54], and the GPU-accelerated method RAPIDS [38] deployed with the Spark. For the academic work, we adopt the state-of-the-art system FPDB [62]. FPDB combines remote pushdown with local caching, and we implement it by integrating the Arrow storage format into the computation layer along with S3 Select. Furthermore, we follow the approach used to build *Spark-All* in Section 3.2, which combines FPDB and RAPIDS to construct *Presto-All*. It is equivalent to FPDB, since the Presto engine does not support RAPIDS.

We use the end-to-end execution time as the metric to measure the performance for the OLAP query. We also use the CPU core time, cache space, GPU utilization, and GPU memory as the resource efficiency comparison metrics.

8.2 Reducing Query Execution Time

In this subsection, we evaluate the query execution time of ORION and the baseline works on the OLAP engines Spark and Presto. For each query, we first test under no data available in the cache as the "Cold" case. Then, we test it again as the normal cached case while setting the capacity of the cache cluster to 80% of its required data.

Figure 12 shows the end-to-end query execution time with ORION and baseline works on Spark and Presto engines. Since both Pushdown and RAPIDS have no cache optimizations, the query execution time for the "Cold" and normal cases is the same, resulting in only one type of legend in Figure 12. We can observe that ORION achieves the shortest execution time in all the cases.

As statistics, ORION respectively reduces the execution time by 39.2%, 19.9%, 50.9%, 33.4% on average, compared to Pushdown, RAPIDS, Alluxio, and FPDB in the "Cold" cases, while 61.8%, 50.2%, 52.8%, and 42.2% on average in the normal cases for the Spark engine. Moreover, since the Presto does not support GPU-accelerated frameworks [4], we only compare ORION to the remaining baselines. ORION can reduce the query execution time by 43.4%, 48.6%, and 33.5% on average compared to Pushdown, Alluxio, and FPDB in

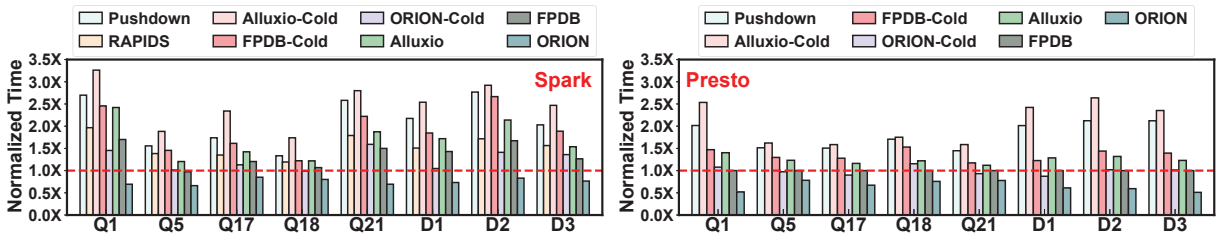


Figure 12: The query execution time with ORION and baselines normalized to Spark-All or Presto-All, respectively. The mark "-Cold" represents the special case of the first time OLAP query execution with no cached data at all.

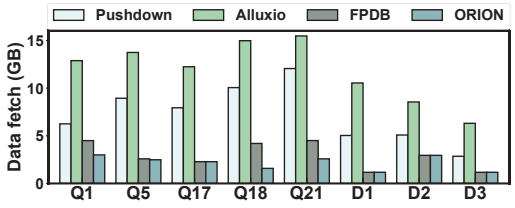


Figure 13: The data fetching amount from remote storage with Alluxio, FPDB, RAPIDS, and ORION on Spark.

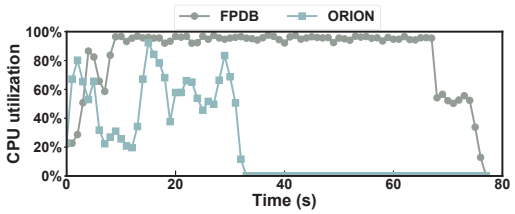


Figure 14: The CPU core utilization during the execution process of Q1 with FPDB and ORION.

the "Cold" cases, respectively, while 63.1%, 50.8%, and 40.3% in normal cases. ORION performs better in normal cached scenarios since its data ordering strategy plays roles in effectively increasing cache hit rate and reducing repeated data fetching for queries with cached data.

We can also find that ORION outperforms Spark-All that combines various optimizations, with an average execution time decrease of 25.8%. This is because Spark-All remains centered around the OLAP engine, which proves that simply combining different optimizations cannot achieve an efficiency similar to ORION. Presto-All is the same as FPDB, as it does not support GPU accelerations.

Observing the results in-depth, ORION can reduce the query execution time by 34.8% on average in the five queries of TPC-H, while obviously reducing that more by 49.1% for the three business queries on Spark, and Presto has similar results. This is because more Groupby and Aggregate operators are deployed in these real-world business queries, which can present more speedup when appropriately utilizing GPU resources by ORION. Conversely, TPC-H uses more Filter

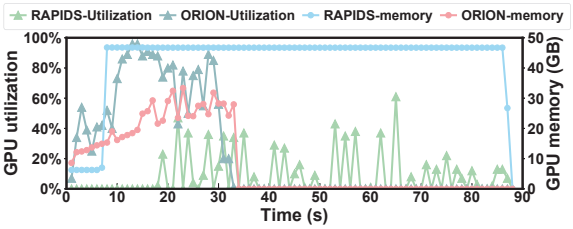


Figure 15: The GPU memory and utilization during execution process of Q1 under ORION and RAPIDS.

operators to filter data in advance, resulting in more reliance on table joins rather than aggregate.

8.3 Diving into Execution Time Reduction

We use Spark in the normal cached cases as examples to analyze why ORION has better performance than other works.

Figure 13 shows the total data fetching amount of all queries under ORION and baseline works. We can observe that ORION has the lowest data fetching amount, which is reduced by 67.3%, 81.4%, and 25.2% compared to Pushdown, Alluxio, FPDB, respectively. This is because ORION prioritizes processing the data batches that are available in the cache, rather than fetching data in the order requested by the OLAP engine like baselines. In this way, the repeated data fetching time is reduced. Moreover, when the query data is not cached at all, ORION utilizes an asynchronous data fetching strategy to make operator execution and data fetching overlap, which can also reduce waiting time.

Figure 14 shows the CPU utilization during the execution process of Q1 under the integrated resource allocation of FPDB and function-based mode of ORION, respectively. We observe that the function-based mode utilizes 39.2% fewer CPU cores on average with less query execution time. The average reduction of all the tested queries is 53.3%. This proves that the separated resource allocation results in higher resource utilization and better query execution performance.

Figure 15 shows the GPU memory usage and GPU utilization in Q1 for ORION and RAPIDS. We first observe that ORION completes the query execution earlier. Moreover,

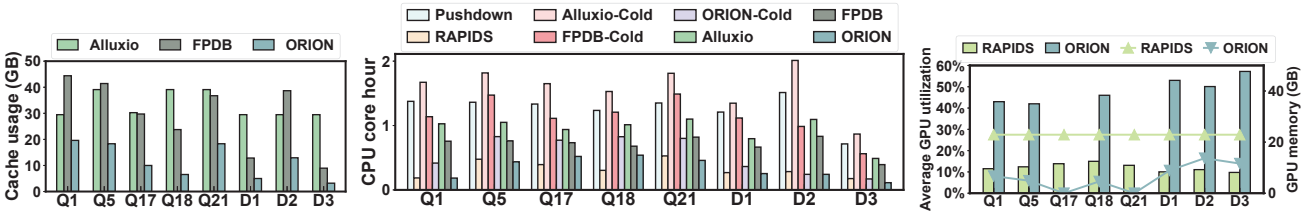


Figure 16: The cache space, CPU core hour, GPU utilization, and GPU memory usage with ORION and baselines.

ORION utilizes 48.6% on average of the GPU memory with higher GPU utilization while RAPDIS used up almost all GPU memory with significantly low GPU utilization of 11.4% on average during execution. The average GPU memory reduction of all the tested queries is 60.4%. ORION only offloads the compute-intensive operators (Groupby and Aggregate) to GPU, which improves the computation efficiency. By contrast, RAPIDS offloads all operators to GPUs that fill up the GPU memory, which reduces the data batch execution speed.

8.4 High Resource Efficiency

In this subsection, we evaluate ORION's efficiency in resource usage during the query execution. The resources include the CPU and cache space usage in the cache cluster, and the GPU utilization and memory usage in the GPU cluster. The results shown here are from the Spark engine, and the performance of the Presto is similar.

The left part of Figure 16 shows the cache space required for each query of ORION and the baseline works with cache optimizations. We observe that ORION can reduce the cache space by 64.6% and 59.7% on average than Allixio and FPDB, respectively. ORION uses the columnar format and only caches the set of columns used in the query, which achieves the most efficient cache usage. By contrast, Alluxio caches files one-to-one to the cache, and uses more cache space. FPDB uses arrow format without performing compression operations, thus leading to more cache usage.

The middle part of Figure 16 shows the CPU core time usage during the query execution under different cases with the Spark engine. We observe that ORION can respectively reduce the CPU core time by 56.2%, 65.2%, and 51.3% on average, compared to Pushdown, Alluxio, and FPDB in the uncached "Cold" cases, while that of 72.6%, 63.2%, and 51.3% in the normal cached cases. Moreover, ORION slightly utilizes more CPU cores than RAPIDS, with an average increase of 11.9%. However, RAPIDS has a longer query execution time than ORION and uses more expensive GPU memory.

The right part of Figure 16 shows the average GPU memory usage and GPU utilization for ORION and RAPIDS in normal cached cases. The results in "Cold" cases are similar. The bars and curves represent GPU utilization and GPU memory usage, respectively. We observe that ORION reduces

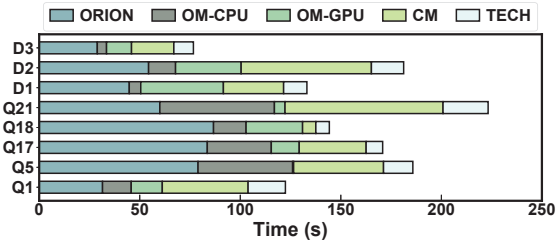


Figure 17: The contribution of each module to execution time difference between ORION and Non-ORION.

the GPU memory usage by 61.3% on average than RAPIDS. For the executions of all queries, RAPIDS will fill up the GPU memory (total 48GB), but the GPU utilization is as low as 12.1% on average. The GPUs are fully occupied but not efficiently utilized for operator executions, leading to performance decrease with low resource efficiency.

8.5 Effectiveness of Each Individual Module

In this subsection, we evaluate the contributions of OM-CPU, OM-GPU, CM, and TECH, which refer to the function scheduling on CPUs of operator manager, the function offloading onto GPUs, the cache manager, and technical implementations, respectively. The evaluation is conducted by continuously removing different modules from ORION until it degenerates into Non-ORION.

Figure 17 shows the optimization time of each module. On average, OM-CPU, OM-GPU, CM, and TECH can reduce the query execution time by 14.3%, 14.2%, 24.3%, and 8.9%, respectively, and a total of 61.8% compared to Non-ORION. In terms of contributions to the overall optimization effect, they contribute 23.2%, 23.0%, 39.4%, and 14.4%, respectively. We can observe that the cache manager and operator manager play the similar role (39.4% and 46.2%), while our technical implementations optimize for 14.4%. The contribution of TECH is brought by using in-memory column-based data format, which offers better caching and operator execution efficiency compared to traditional JVM-based methods. The reduction of data fetching amount and cache utilization are all contributed by the proactive cache manager. Moreover, ORION's operator manager reduces CPU core and GPU memory usage by separating and fine-grained scheduling operators.

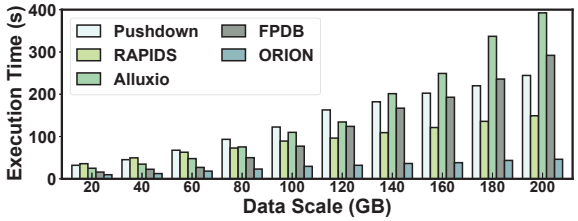


Figure 18: The query execution time with ORION and baseline works under different data scales.

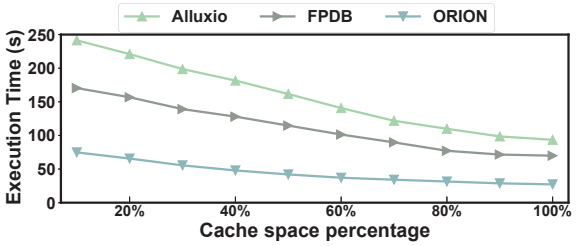


Figure 19: The execution time of Q1 with Alluxio, FPDB, and ORION for Spark under different cache space sizes.

Specially, OM-GPU plays a more important role in Q1, Q18, D1, and D2. This is because these queries involve a higher number of Groupby and Aggregate operations during executions and can be significantly accelerated by the GPU. Moreover, CM contributes more in Q21 and D2, as their data scales are larger, and CM enables a more efficient reduction of repeated data batch fetching and unnecessary data fetching.

8.6 The Impacts of Data Scale

To investigate the impact of the data scale, we modify the scale value of the TPC-H benchmark from 20 to 200 to generate different data scales. We then evaluate the query execution time of Q1 for ORION and different baseline works.

Figure 18 shows the query execution time of ORION and baseline works under the different data scales. The results show that ORION has the best performance under all data scales. As the data scale increases, ORION's advantage over other works also increases. For instance, the execution time of Alluxio is 2.6X compared to ORION with the data scale of 20, while that of 8.5X with the scale of 200.

The reasons originate from two aspects. Firstly, with the data scale increase, the cache manager of ORION can reduce more data waiting and repeated fetching time than baseline works with passive data management. Secondly, ORION can fine-grained manage the hardware resources, which brings better capabilities to process more data batches in parallel.

8.7 The Impacts of Cache Layer Size

In real-world scenarios, the cache cluster space may not be abundant to cache all the required data. To evaluate the impact of varying cache space sizes, we adjust the cache space ranges from 10% of the total required data of the queries to 100%, and then evaluate the execution time of Q1 for ORION and different baseline works.

Figure 19 shows the execution time of Q1 with ORION and baseline works on Spark under different cache space percentages. We can observe that ORION has the best performance under all cache space sizes. When the cache space is sufficient with caching 100% of the required data, ORION can reduce the query execution time by 70.9% and 61.1% compared to Alluxio and FPDB, respectively. When the cache space is insufficient at 10%, ORION can also reduce the query execution time by 69.2% and 56.1% correspondingly.

The results show that ORION is less affected by cache size than others. This is because ORION can process data batches based on data availability in the cache to reduce repeated data fetching when the cache space becomes smaller.

Table 2: Descriptions of our business queries

Query	Query Description	Main Operations
B1-B3	Evaluate certain characteristic data of stocks in different conditions.	Scan, Filter, Projection, Groupby, Aggregate
B4-B5	Filter by different conditions and remove duplicate transaction information.	Scan, Filter, Distinct
B6-B7	Filter transaction records by special characteristics, then aggregate, and sort.	Scan, Filter, Aggregate
B8	Retrieve transactions for certain period.	Scan, Filter
B9-B10	Retrieve special items of user information grouped by attributes and aggregate.	Scan, Filter, Aggregate, Join

8.8 Scalability Results in Production

ORION has also been deployed for our real-world business in our production datacenters. In real-world scenarios, systems often process data on a much larger scale with tens of billions of records (up to 40TB data). The production cluster also has a large scale with about 12,000 CPU cores, 3TB memory, and 100 GPU cards with each 8GB-16GB GPU memory. We select 10 of the most frequently used queries here, which have different functionalities and different data scales. For commercial confidentiality, we only describe the functionality of each query, as shown in Table 2.

Figure 20 shows the achieved speedup after deploying ORION compared to the performance before deployment in different queries on our interval OLAP engine. ORION achieves an average speedup of 2.6X on ten queries. The results of production deployment are better or equal than the query selected in the experiments in Section 8.2, because the query in production deployment has more operators suitable for GPU computation, while the data magnitude is also larger. The highest and lowest speedup is achieved on B2 and B8, respectively. This is because B2 uses the most computation operators and also uses the largest data scale,

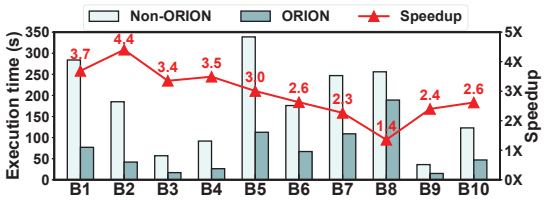


Figure 20: The execution time for business queries with ORION and Non-ORION on our interval OLAP engine.

which can leverage the most advantages when using ORION. Meanwhile, B8 has the least operators with a small data scale.

8.9 Overhead of ORION

In our evaluations, the average overhead of the cache manager is about 140ms for executing an OLAP query, which includes 125ms for the data batch reordering strategy and 15ms for creating processes to fetch data asynchronously. For the operator manager, the average overhead is 472ms, which includes 420ms for the function builder to transform operators into functions, and 52ms for function scheduling and resource adjustment during the query execution.

Therefore, the total overhead for executing an OLAP query is less than 650ms, which is negligible relative to the OLAP query execution time of tens to hundreds of seconds.

9 Conclusion

In this paper, we propose the ORION manager to take over the data and operator management capabilities from the OLAP engine to reduce OLAP query execution time, under the disaggregated computation, cache, GPU, and storage clusters. ORION's cache manager orchestrates data batch processing orders and asynchronously fetches required data, to avoid data waiting and repeated fetching during the query execution. ORION's operator manager includes a function builder and a function scheduler. The function builder separates the pushdown operators into independent functions. Then, the function scheduler determines the function scheduling to CPUs or GPUs, and adjusts just-enough resources for them, to enhance query execution performance. We have implemented ORION, and the results show that, compared to the state-of-the-art works, ORION can reduce the query execution time by 42.2% on average.

Acknowledgments

We sincerely thank our anonymous reviewers for their helpful comments and suggestions. This work is partially sponsored by the National Natural Science Foundation of China (62232011, 62302302).

References

[1] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, Samuel Madden, et al. 2013. The design and implementation of modern column-oriented database systems. *Foundations and Trends® in Databases* 5, 3 (2013), 197–280.

[2] Daniel J Abadi, Samuel R Madden, and Nabil Hachem. 2008. Column-stores vs. row-stores: how different are they really?. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 967–980.

[3] RAPIDS AI. 2024. *cuDF: GPU DataFrame Library.* https://github.com/rapidsai/cudf

[4] RAPIDS AI. 2024. *RAPIDS.* https://rapids.ai/

[5] Alluxio. 2024. *Alluxio.* https://www.alluxio.io/

[6] Nils Boeschen and Carsten Binnig. 2022. Gacco-a gpu-accelerated oltp dbms. In *Proceedings of the 2022 International Conference on Management of Data*. 1003–1016.

[7] Jiashen Cao, Rathijit Sen, Matteo Interlandi, Joy Arulraj, and Hyesoon Kim. 2023. GPU Database Systems Characterization and Optimization. *Proceedings of the VLDB Endowment* 17, 3 (2023), 441–454.

[8] Surajit Chaudhuri and Umeshwar Dayal. 1997. An overview of data warehousing and OLAP technology. *ACM Sigmod record* 26, 1 (1997), 65–74.

[9] Surajit Chaudhuri and Umeshwar Dayal. 1997. An overview of data warehousing and OLAP technology. *ACM Sigmod record* 26, 1 (1997), 65–74.

[10] Mohamed L Chouder, Stefano Rizzi, and Rachid Chalal. 2019. EXODuS: exploratory OLAP over document stores. *Information Systems* 79 (2019), 44–57.

[11] Alibaba Cloud. 2024. *Alibaba Cloud Object Storage Service.* https://www.alibabacloud.com/product/oss

[12] Ceph Community. 2024. *Ceph: A Scalable, High-Performance Distributed Storage System.* https://ceph.io/en/

[13] SQLite Consortium. 2024. *SQLite Home Page.* https://www.sqlite.org/

[14] Transaction Processing Performance Council. 2024. *TPC-H Benchmark.* http://www.tpc.org/tpch/

[15] Katerina Doka, Nikolaos Papailiou, Dimitrios Tsoumakos, Christos Mantas, and Nectarios Koziris. 2015. Ires: Intelligent, multi-engine resource scheduler for big data analytics workflows. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1451–1456.

[16] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. 2020. Quantifying TPC-H choke points and their optimizations. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1206–1220.

[17] Apache Software Foundation. 2024. *Apache Arrow.* https://arrow.apache.org/

[18] Apache Software Foundation. 2024. *Apache Parquet.* https://parquet.apache.org/ July 31, 2024.

[19] Apache Software Foundation. 2024. *Apache Spark Official Website.* https://spark.apache.org/

[20] Apache Software Foundation. 2024. *Spark SQL, DataFrames and Datasets Guide.* https://spark.apache.org/docs/latest/sql-data-sources.html

[21] Presto Foundation. 2024. *Connectors.* https://prestodb.io/docs/current/develop/connectors.html

[22] Presto Foundation. 2024. *Presto.* https://prestodb.io/

[23] Trino Foundation. 2024. *Trino.* https://trino.io/

[24] Kaihua Fu, Jiuchen Shi, Quan Chen, Ningxin Zheng, Wei Zhang, Deze Zeng, and Minyi Guo. 2022. QoS-aware irregular collaborative inference for improving throughput of DNN services. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.

[25] Juicedata Inc. 2024. *JuiceFS: A Distributed POSIX File System built on top of Redis and S3.* https://github.com/juicedata/juicefs

[26] Qubole Inc. 2017. *RubiX: A caching library that can be used to cache data from cloud stores such as Amazon S3, Azure Blob Storage, Google Cloud Storage, and others.* https://github.com/qubole/rubix

[27] Intel Corporation. 2020. Intel®AVX-512 Instructions. *Intel®Developer Zone* (2020). Available at https://www.intel.com/content/www/us/en/developer/articles/technical/intel-avx-512-instructions.html.

[28] Tomas Karnagel, René Müller, and Guy M Lohman. 2015. Optimizing GPU-accelerated Group-By and Aggregation. *ADMS@ VLDB* 8 (2015), 20.

[29] Suyeon Lee and Sungyong Park. 2021. Performance Analysis of Big Data ETL Process over CPU-GPU Heterogeneous Architectures. In *2021 IEEE 37th International Conference on Data Engineering Workshops (ICDEW)*. 42–47. https://doi.org/10.1109/ICDEW53142.2021.00015

[30] Guoliang Li, Haowen Dong, and Chao Zhang. 2022. Cloud databases: New techniques, challenges, and opportunities. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3758–3761.

[31] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2016. Hippogriffdb: Balancing i/o and gpu bandwidth in big data analytics. *Proceedings of the VLDB Endowment* 9, 14 (2016), 1647–1658.

[32] Haotian Liu, Bo Tang, Jiashu Zhang, Yangshen Deng, Xiao Yan, Xinying Zheng, Qiaomu Shen, Dan Zeng, Zunyao Mao, Chaozu Zhang, et al. 2022. Ghive: accelerating analytical query processing in apache hive via cpu-gpu heterogeneous computing. In *Proceedings of the 13th Symposium on Cloud Computing*. 158–172.

[33] Haotian Liu, Bo Tang, Jiashu Zhang, Yangshen Deng, Xinying Zheng, Qiaomu Shen, Xiao Yan, Dan Zeng, Zunyao Mao, Chaozu Zhang, et al. 2022. Ghive: A demonstration of gpu-accelerated query processing in apache hive. In *Proceedings of the 2022 International Conference on Management of Data*. 2417–2420.

[34] Wes McKinney et al. 2011. pandas: a foundational Python library for data analysis and statistics. *Python for high performance and scientific computing* 14, 9 (2011), 1–9.

[35] Jérôme Meinke. 2015. In-memory OLAP aggregation on GPUs using CUDA Dynamic Parallelism. (2015).

[36] Inc. MinIO. 2024. *MinIO.* https://min.io/

[37] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*. 561–577.

[38] NVIDIA. 2024. *Spark-RAPIDS.* https://nvidia.github.io/spark-rapids/

[39] OmniSci. 2020. *OmniSci Documentation: Overview.* https://docs.omnisci.com/v5.1.1/1_overview.html

[40] PrestoDB. 2024. *Presto Administration: Configuration Properties.* https://prestodb.io/docs/current/admin/properties.html

[41] Adam Prout, Szu-Po Wang, Joseph Victor, Zhou Sun, Yongzhu Li, Jack Chen, Evan Bergeron, Eric Hanson, Robert Walzer, Rodrigo Gomes, and Nikita Shamgunov. 2022. Cloud-Native Transactions and Analytics in SingleStore. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2340–2352. https://doi.org/10.1145/3514221.3526055

[42] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seungwon Min, Amna Masood, Jeongmin Park, Jinjun Xiong, Chris J Newburn, Dmitri Vainbrand, I-Hsin Chung, et al. 2023. GPU-initiated on-demand high-throughput storage access in the BaM system architecture. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 325–339.

[43] Mark Raasveldt and Hannes Mühleisen. 2019. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*. 1981–1984.

[44] Sri Pramodh Rachuri, Arun Gantasala, Prajeeth Emanuel, Anshul Gandhi, Robert Foley, Peter Puhov, Theodoros Gkountouvas, and Hui Lei. 2022. Optimizing Near-Data Processing for Spark. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. 636–646. https://doi.org/10.1109/ICDCS54860.2022.00067

[45] Amazon Web Services. 2024. *Amazon EMR: Use Presto with Amazon S3 Select.* https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-presto-s3select.html

[46] Amazon Web Services. 2024. *Amazon Simple Storage Service.* https://aws.amazon.com/s3/

[47] Amazon Web Services. 2024. *Getting Started with Amazon Redshift Spectrum.* https://docs.aws.amazon.com/redshift/latest/dg/c-getting-started-using-spectrum.html

[48] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A study of the fundamental performance characteristics of GPUs and CPUs for database analytics. In *Proceedings of the 2020 ACM SIGMOD international conference on Management of data*. 1617–1632.

[49] Jiuchen Shi, Kaihua Fu, Quan Chen, Changpeng Yang, Pengfei Huang, Mosong Zhou, Jieru Zhao, Chen Chen, and Minyi Guo. 2022. Characterizing and orchestrating VM reservation in geo-distributed clouds to improve the resource efficiency. In *Proceedings of the 13th Symposium on Cloud Computing*. 94–109.

[50] Jiuchen Shi, Kaihua Fu, Jiawen Wang, Quan Chen, Deze Zeng, and Minyi Guo. 2024. Adaptive QoS-aware microservice deployment with excessive loads via intra-and inter-datacenter scheduling. *IEEE Transactions on Parallel and Distributed Systems* (2024).

[51] Jiuchen Shi, Jiawen Wang, Kaihua Fu, Quan Chen, Deze Zeng, and Minyi Guo. 2022. Qos-awareness of microservices with excessive loads via inter-datacenter scheduling. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 324–334.

[52] Jiuchen Shi, Hang Zhang, Zhixin Tong, Quan Chen, Kaihua Fu, and Minyi Guo. 2023. Nodens: Enabling resource efficient and fast {QoS} recovery of dynamic microservice applications in datacenters. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 403–417.

[53] Substrait. 2024. *Substrait: Cross-Language Serialization for Relational Algebra.* https://substrait.io/

[54] Chunxu Tang, Bin Fan, Jing Zhao, Chen Liang, Yi Wang, Beinan Wang, Ziyue Qiu, Lu Qiu, Bowen Ding, Shouzhuo Sun, Saiguang Che, Jiaming Mai, Shouwei Chen, Yu Zhu, Jianjian Xie, Yutian (James) Sun, Yao Li, Yangjun Zhang, Ke Wang, and Mingmin Chen. 2024. Data Caching for Enterprise-Grade Petabyte-Scale OLAP. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 901–915. https://www.usenix.org/conference/atc24/presentation/tang

[55] Ray Team. 2024. *Ray: Actors.* https://docs.ray.io/en/latest/ray-core/actors.html

[56] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1626–1629.

[57] Dimitrios Tsoumakos and Christos Mantas. 2014. The case for multi-engine data analytics. In *Euro-Par 2013: Parallel Processing Workshops: BigDataCloud, DIHC, FedICI, HeteroPar, HiBB, LSDVE, MHPC, OMHI, PADABS, PROPER, Resilience, ROME, and UCHPC 2013, Aachen, Germany, August 26-27, 2013. Revised Selected Papers 19*. Springer, 406–415.

[58] Tobias Vinçon, Christian Knödler, Leonardo Solis-Vasquez, Arthur Bernhardt, Sajjad Tamimi, Lukas Weber, Florian Stock, Andreas Koch, and Ilia Petrov. 2022. Near-data processing in database systems on native computational storage under htap workloads. *Proceedings of the VLDB Endowment* 15, 10 (2022), 1991–2004.

[59] Jialun Wang, Wenhao Pang, Chuliang Weng, and Aoying Zhou. 2023. D-Cubicle: boosting data transfer dynamically for large-scale analytical queries in single-GPU systems. *Frontiers of Computer Science* 17, 4 (2023), 174610.

[60] Robert Wrembel and Christian Koncilia. 2007. *Data warehouses and OLAP: concepts, architectures, and solutions.* Igi Global.

[61] Yifei Yang, Matt Youill, Matthew Woicik, Yizhou Liu, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2021. Flexpushdowndb: Hybrid pushdown and caching in a cloud dbms. *Proceedings of the VLDB Endowment* 14, 11 (2021).

[62] Yifei Yang, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2024. FlexpushdownDB: rethinking computation pushdown for cloud OLAP DBMSs. *The VLDB Journal* (10 Jul 2024). https://doi.org/10.1007/s00778-024-00867-8

[63] Xiangyao Yu, Matt Youill, Matthew Woicik, Abdurrahman Ghanem, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2020. PushdownDB: Accelerating a DBMS using S3 computation. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1802–1805.

[64] Gina Yuan, Shoumik Palkar, Deepak Narayanan, and Matei Zaharia. 2020. Offload Annotations: Bringing Heterogeneous Computing to Existing Libraries and Workloads. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 293–306. https://www.usenix.org/conference/atc20/presentation/yuan

[65] Yansong Zhang, Yu Zhang, Jiaheng Lu, Shan Wang, Zhuan Liu, and Ruichen Han. 2020. One size does not fit all: accelerating OLAP workloads with GPUs. *Distributed and Parallel Databases* 38 (2020), 995–1037.