Trading Continuous Queries

Jin Cheng, Ningning Ding, John C.S. Lui, *Fellow, IEEE*, and Jianwei Huang, *Fellow, IEEE*

***Abstract*—In the Big Data era, data trading significantly enhances data-driven decision-making by facilitating data sharing. Streaming data from sources such as mobile devices and social media platforms creates new opportunities and challenges for data trading. Traditional data trading methods, designed for one-time queries over static database snapshots, neglect the growing need for trading continuous queries over streaming data. If applied directly to continuous queries, existing methods often result in repeated and imprecise charges that reduce the seller's profit, as they do not consider computation sharing during continuous query execution. To address these challenges, we propose CQTrade, the first mechanism for continuous query-based data trading, which incorporates computation sharing in query execution and integrates seamlessly with existing trading mechanisms. Our contributions are threefold: (1) we provide a theoretical analysis of prevalent computation-sharing techniques, including cost modeling and closed-form computation-sharing strategy derivation; (2) we formulate a general optimization problem to maximize the seller's profit, adaptable to various computation-sharing techniques; (3) we identify that our optimization problem merges vector bin packing and multidimensional knapsack challenges, and we tackle this complexity with a tailored branch-and-price algorithm that decomposes the problem into a masterproblem and multiple sub-problems, achieving a globally optimal solution. Evaluation shows CQTrade improves trading success rate by 12.8% and increases seller profit by 28.7% compared to traditional methods.**

***Index Terms*—Data trading, continuous query, online analytics, computation sharing, branch-and-price algorithm.**

Received 11 March 2025; revised 31 July 2025; accepted 14 October 2025. Date of publication 24 October 2025; date of current version 4 February 2026. This work was supported by the National Natural Science Foundation of China under Grant 62502412 and Grant 62271434, in part by the Shenzhen Stability Science Program 2023, in part by the Shenzhen Institute of Artificial Intelligence and Robotics for Society, and in part by the Longgang District Shenzhen's "Ten Action Plan" for Supporting Innovation Projects under Grant LGKCSDPT2024002. The work of John C.S. Lui is supported in part by the GRF-14202923. Recommended for acceptance by J. Ren. *(Corresponding Author: Jianwei Huang.)*

Jin Cheng is with the School of Science and Engineering and Shenzhen Institute of Artificial Intelligence and Robotics for Society, The Chinese University of Hong Kong, Shenzhen 518172, China, and also with the The Chinese University of Hong Kong, Shenzhen 518172, China (e-mail: jincheng2@link.cuhk.edu.cn).

Ningning Ding is with the Data Science and Analytics Thrust and the Internet of Things Thrust, Information Hub, Hong Kong University of Science and Technology (Guangzhou), Guangzhou 510530, China (e-mail: ningningding@hkust-gz.edu.cn).

John C.S. Lui is with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shenzhen 518172, China (e-mail: cslui@cse.cuhk.edu.hk).

Jianwei Huang is with the School of Science and Engineering, Shenzhen Institute of Artificial Intelligence and Robotics for Society, Shenzhen Key Laboratory of Crowd Intelligence Empowered Low-Carbon Energy Network, and CSIJRI Joint Research Centre on Smart Energy Storage, The Chinese University of Hong Kong, Shenzhen 518172, China, and also with Shenzhen Loop Area Institute, Shenzhen 518000, China (e-mail: jianweihuang@cuhk.edu.cn).

Digital Object Identifier 10.1109/TMC.2025.3625547

I. Introduction

A. Background and Motivation

DATA has become a key asset in data-driven decision-making, particularly in areas such as machine learning [2]. This shift has accelerated the development of data trading markets [3]. Mobile devices, including smartphones, IoT sensors, and wearables, continuously generate large volumes of streaming data that offer valuable insights for real-time analytics [4]. To enable the exchange of such valuable data, platforms like AWS Data Exchange [5] and Xignite [6] have emerged to support efficient data sharing. These platforms help strengthen machine learning models by enabling access to diverse training data. In 2022, the global data trading market was valued at $968 million and is projected to grow at an annual rate of 25% from 2023 to 2030 [7].

The rise of online analytics and machine learning has shifted data trading from one-time queries to continuous query (CQ)-based approaches. Modern applications increasingly rely on continuous queries (CQs) for streaming data, particularly from mobile devices, rather than traditional one-time queries over static database snapshots [8], [9]. Unlike traditional queries over static snapshots, CQs operate over streaming data and provide ongoing updates. A continuous query aggregates data over a moving time window defined by a *range*, and produces results periodically based on a fixed *slide* interval [8], [9]. For example, in stock trading, a CQ can monitor the highest price in a 30-minute window and update results every 5 minutes. In IoT systems, CQs can track average vehicle speed every 3 minutes over a 15-minute range to support real-time traffic control. As shown in Fig. 1, CQs also support online machine learning [10], [11], [12], [13] by continuously feeding models with fresh data to improve prediction accuracy and responsiveness.[1]

However, existing data trading methods are not directly applicable to data from continuous queries, as they are primarily designed for one-time queries over static database snapshots. A key limitation is the lack of *computation sharing* across time windows and between queries, an essential feature for efficient CQ execution [14]. Without this sharing, the system must repeatedly process overlapping data, leading to unnecessary charges, lower seller profits, and inefficient data usage.

To demonstrate traditional data trading limitations, we present an example where a stock monitoring system periodically

[1] Although our motivating examples focus on mobile-generated data streams, the mechanism is broadly applicable to continuous queries over any streaming data environment, such as industrial sensor networks, financial market data, and social media platforms.

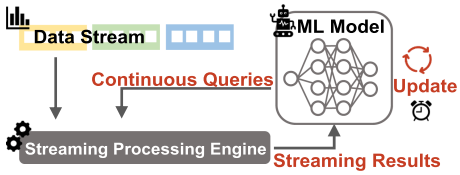


Fig. 1. Online machine learning.

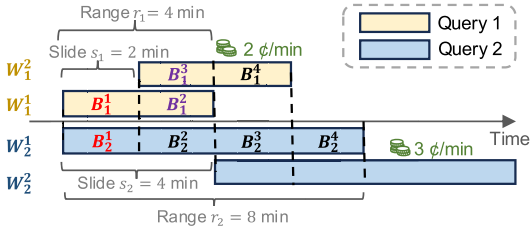


Fig. 2. Examples of continuous queries.

reports the latest highest stock price to two customers through distinct queries, each with unique temporal requirements and payments. We distinguish these queries with subscripts 1, 2 for clarity. The customers pay 2¢/min and 3¢/min for their respective queries. In a one-time query-based trading mechanism[2], achieving periodically updated results requires repeated one-time queries over the database. With execution costs of 5¢ and 14¢ for Query 1 and Query 2, the costs per unit time amount to 2.5¢/min (5¢/2 min) and 3.5¢/min (14¢/4 min), respectively. Since these execution costs exceed the customer payments, the seller cannot profitably execute these queries, resulting in zero profit.[3]

To overcome these limitations, we introduce *computation sharing* for continuous queries. As illustrated in Fig. 2, continuous queries can be broken into smaller blocks based on their time window edges. Many of these blocks process the same data, leading to significant overlap in computation. This overlap enables two forms of *computation sharing*, which can greatly improve execution efficiency:

- *Self-sharing* across time windows: This type of sharing occurs when consecutive windows overlap, as illustrated by Blocks B_1^2 and B_1^3 in Query 1. Specifically, while Window W_1^1 combines the highest prices from blocks B_1^1 and B_1^2, Window W_1^2 uses blocks B_1^3 and B_1^4. Since B_1^2 and B_1^3 cover identical time periods, the system can effectively reuse B_1^2's stored information for Window W_1^2.
- *Mutual-sharing* among different queries: This type of sharing emerges when multiple queries process the same time period, as illustrated by Blocks B_1^1 and B_2^1. Specifically, these blocks share the same results when computing their respective windows, avoiding redundant calculations.

[2]Traditional one-time query-based mechanisms apply static prices based on data granularity, assuming isolated executions. In continuous query settings, platforms may treat each window independently and charge repeatedly. While buyers may accept this, sellers incur redundant costs across overlapping windows. Without computation sharing, total costs are more likely to exceed payments, leading to reduced or negative profit.

[3]Although adaptive per-execution pricing for one-time queries is theoretically feasible, it relies on dynamic cost estimation and per-query negotiation, which are not supported by existing one-time query systems. This adds overhead and may cause high latency or cost for time-sensitive buyers, limiting practical use.

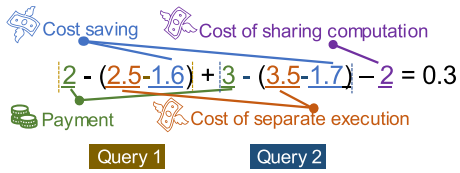


Fig. 3. Profit of the example under computation sharing.

Computation sharing enables profitable data trading by reducing execution costs. Although sharing computation among multiple queries introduces additional overhead (e.g., 2¢/min for calculation and storage), the system can achieve substantial cost savings. Specifically, the execution costs of Query 1 and Query 2 are reduced by 1.6¢/min and 1.7¢/min, respectively. The final execution costs are obtained by subtracting the reduced execution cost from the original separate execution costs (i.e., 2.5¢/min − 1.6¢/min and 3.5¢/min − 1.7¢/min). Given a total payment of 5¢/min from the buyers and a shared computation cost of 2¢/min, the seller obtains a net profit of 0.3¢/min, as shown in Fig. 3. This analysis illustrates how sharing intermediate computations lowers marginal costs and transforms otherwise unprofitable or marginal queries into economically viable ones.

We present a mechanism for continuous query trading that leverages *computation sharing*. The mechanism treats CQs as marketable assets, integrating technical execution with economic optimization. Within this mechanism, customers specify their CQs, while the seller designs and executes query plans to maximize profit based on the trading strategy. Each plan specifies which CQs to execute and how to share computation among queries within the same plan. Different query plan combinations lead to diverse computation patterns, which directly affect both execution cost and seller profit. Therefore, it is crucial to design a trading strategy that balances these patterns to achieve maximum profitability in continuous query-based data trading.

However, this mechanism presents several challenges. The first challenge lies in modeling costs of continuous queries, particularly due to computation sharing across different time windows (self-sharing) and among queries (mutual-sharing), as illustrated in Fig. 2. Current research primarily focuses on practical implementation, lacking detailed theoretical cost modeling, which is essential for effective trading strategy design. This brings us to our first key question:

Key Question 1. How can we model the costs of continuous queries, considering computation sharing across different windows and among different queries?

Second, as depicted in Figs. 2 and 3, computation sharing introduces additional operational costs stemming from the extra steps required for computation sharing, including costs for computing common blocks and storing shared results. Although it offers potential benefits, computation sharing does not always guarantee positive profit. This uncertainty necessitates a strategic trade-off between implementing sharing and non-sharing for

both self-sharing and mutual-sharing types. Therefore, the seller must carefully design the trading strategy to achieve a balance, leading to our second key question:

Key Question 2. What is the optimal trading strategy that balances the trade-offs of computation sharing, for both self-sharing and mutual-sharing, to maximize the seller's profit?

Furthermore, multiple computation-sharing techniques exist in practice, each with its own execution process [14]. These differences result in distinct cost models and require trading strategies tailored to each technique. This diversity requires a general trading mechanism adaptable to varying cost structures, prompting our third key question:

Key Question 3. How can sellers formulate a general trading mechanism that adapts to different computation-sharing techniques?

B. Contributions

Our main contributions are summarized as follows:

- *Mechanism for continuous query-based data trading:* To the best of our knowledge, this is the first work on continuous query-based data trading. We incorporate computation sharing, a key component of continuous query execution, to enhance seller profitability. Moreover, our mechanism integrates seamlessly with existing trading mechanisms while preserving their desired properties, as its optimization goals are orthogonal to those of other trading systems.
- *Theoretical analysis of multiple computation-sharing techniques:*: We address the gap in theoretical analysis for multiple computation-sharing techniques by providing cost modeling and deriving closed-form optimal sharing strategies for both self-sharing and mutual-sharing types. This analysis provides valuable insights for the design of the trading mechanism. Despite each technique having distinct cost formulations, we identify a common cost structure and formulate a general optimization problem for profit maximization, adaptable across these techniques.
- *Algorithm design for the trading strategy:* Our general optimization problem combines elements from vector bin packing and multidimensional knapsack problems, resulting in high computational complexity. To manage this complexity, we decompose the problem into a master problem and multiple pricing subproblems, developing a tailored branch-and-price algorithm to find a globally optimal solution.
- *Evaluation of the proposed mechanism:* Both theoretical insights and experimental results demonstrate that our trading mechanism can profit from continuous queries that traditional one-time query-based mechanisms would reject. Our evaluation indicates that the proposed mechanism improves the success rate of data trading by 12.8% and boosts seller profit by an average of 28.7%, compared to the one-time query-based data trading methods in the current data market.

The rest of this paper is organized as follows: Section II reviews related works. Section III presents the system model. Section IV systematically analyzes the computation sharing. Section V details the optimal trading strategy. Section VI presents the experimental evaluation of the trading mechanism. Finally, Section VII concludes this paper.

II. Related Work

We organize related work into two key areas: query-based data trading methods and data stream management systems.

A. Query-Based Data Trading Methods

Existing research on data trading has primarily focused on developing mechanisms for static databases, emphasizing properties such as privacy preservation [15], [16], [17], arbitrage-freeness [3], [18], truthfulness [19], [20], and security [20], [21], [22]. Although these approaches provide robust solutions for specific data trading challenges, they are generally restricted to one-time queries executed over static database snapshots. However, the growing demand for online analytics and real-time machine learning across domains—from stock markets [23] to IoT systems [24]—calls for continuous query-based data trading. Traditional methods, when applied to continuous queries, often lead to inefficiencies and imprecise charges due to the lack of computation sharing [25], as illustrated in Fig. 2. This oversight can significantly undermine the effectiveness of data trading in dynamic environments.

Our work addresses this gap by proposing a trading mechanism that incorporates computation sharing, a key feature of continuous queries, to enhance seller profits through optimized resource utilization. Since optimizing computation sharing is orthogonal to the objectives of existing one-time query-based data trading methods, as demonstrated in Section III-A1, our mechanism seamlessly integrates with current trading methods by enhancing their functionality without altering their fundamental properties, such as arbitrage-freeness and truthfulness.

B. Data Stream Management Systems

Existing research on continuous queries has primarily concentrated on the design of data stream management systems (DSMS), which are crucial for continuous query execution in real-world applications [26]. These efforts include developing continuous query languages [27], creating advanced processing engines [28], [29], and implementing distributed systems [30], [31], [32]. A critical technique within these systems is computation sharing, which enhances execution efficiency by reusing results from common computational blocks [25].

To provide a more precise positioning, our work is related to three distinct strands of research. First, parallel execution techniques have been widely explored to accelerate large-scale analytical queries, including massively parallel join optimization [33] and GPU-based query planning over disaggregated architectures [34]. Second, our problem formulation is closely connected to multi-query optimization (MQO), where prior works focus on sub-expression reuse [35], materialized view selection [36], and cost-model learning for query grouping [37]. Third, in the context of stream or sensor data processing, computation sharing has been studied through shared window

aggregation [38] and demand-based grouping for query-efficient execution [39]. While these approaches typically aim at performance optimization under centralized execution, CQTrade uniquely integrates profit-driven pricing and resource budgeting into the shared execution paradigm, enabling strategic trading decisions across buyers and brokers.

While conventional continuous query processing [25], [26], [27], [28], [29], [30], [31], [32] focuses on timely and resource-efficient execution, it often overlooks the economic implications of such operations in data trading. In contrast, CQTrade treats queries as marketable commodities and enables sellers to strategically select and execute them to maximize profit through computation sharing. This bridges the gap between technical execution and profit-driven decision-making in continuous query markets.

To address this gap, we propose a continuous query-based trading mechanism that leverages computation sharing to capitalize on computational resource optimization and maximize economic benefits. While practical advancements exist in implementation techniques, there is a lack of theoretical analysis on cost modeling and optimal sharing strategies. We tackle this by analyzing computation-sharing techniques, identifying a common cost structure, and developing a flexible trading mechanism to improve economic outcomes.

III. System Model

This section presents a comprehensive mechanism for trading continuous queries. We begin in Section III-A by detailing our trading system design, encompassing three key components: continuous query modeling, query plan modeling, and system cost modeling. Section III-B analyzes three widely adopted computation-sharing techniques for continuous query execution, along with their corresponding cost modeling. Building upon these, Section III-C formulates a general optimization problem that adapts to various computation-sharing techniques through their common cost structure. We summarize key notations in Table I for clarity.

A. Trading System for Continuous Queries

Our proposed trading mechanism, CQTrade, builds upon a data stream management system architecture [26], as illustrated in Fig. 4, with the corresponding interaction flow shown in Fig. 5. In this mechanism, customers first submit continuous queries (CQs) to a centralized query repository based on their analytical requirements. These queries form a set $\mathcal{Q} = \{q_1, \ldots, q_N\}$, each associated with specific window specifications [27]. Next, the platform determines the optimal trading strategy by jointly considering query compatibility, resource budgets, and expected profitability. Finally, customers make payments, and the platform executes the selected query plans by coordinating with the stream processor and storage system to process data and deliver results.

The execution of these queries incurs two primary types of costs: processing costs and memory costs, as detailed in Section III-A3. Our analysis adopts a sell-side market perspective, where we consider the data as already available for trading.

TABLE I
List of Key Notations

Notation	Description
$\mathcal{Q}$	Set of continuous queries (CQs): $\mathcal{Q} = \{q_1, q_2, \ldots\}$
$\mathcal{P}$	Set of query plans: $\mathcal{P} = \{p_1, p_2, \ldots\}$
$\mathcal{T}$	Set of sharing types for the plans: $\mathcal{T} = \{T_{sm}, T_{sn}, T_{nn}\}$
q_j	j-th continuous query, defined as $q_j = (s_j, r_j, \pi_j) \subseteq \mathcal{Q}$
p_i	i-th query plan, where $p_i \subseteq \mathcal{P}$
N	Number of continuous queries
K	Number of query plans
M	Number of plans of type T_{sm}
π_j	Payment for CQ q_j
θ_p	Processing cost coefficient
θ_m	Memory cost coefficient
$s_j(s_i)$	Slide of CQ q_j (query plan p_i)
$r_j(r_i)$	Range of CQ q_j (query plan p_i)
$\gamma_j(\gamma_i)$	Overlap factor of CQ q_j (query plan p_i)
$E(p_i)$	Number of partials generated by Plan p_i per unit time
$\Phi(p_i)$	Number of final aggregations of plan p_i on each partial
$\Gamma(p_i)$	Number of intermediate results of plan p_i to be stored
x_{ij}	Binary variable: 1 if query j is assigned to plan i
y_i	Binary variable: 1 if plan i is activated

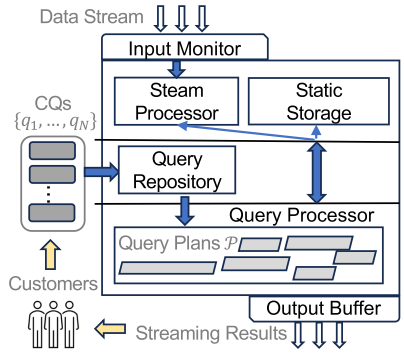


Fig. 4. Overview of CQTrade.

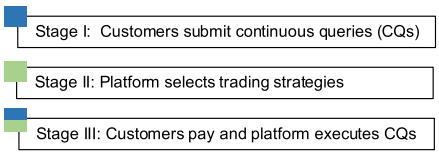


Fig. 5. Interaction flow of CQTrade.

This assumption treats initial data acquisition and preprocessing costs as sunk costs, enabling us to focus on the core market operations. This approach not only simplifies our analysis but also reflects the practical constraints faced by data sellers in real-world markets. In the following subsections, we present the detailed modeling for three fundamental aspects: continuous queries, query plans, and system costs.

1) Continuous Query Modeling: We focus on aggregate continuous queries [40], which serve as key tools in online analytics applications due to their critical role in real-time information aggregation. The formal definition of a continuous query is as follows:

Definition 1 (Continuous Query). A continuous query $q = (s, r, \pi)$ performs data aggregation over streaming data using time windows. Each window is characterized by:

- A 'range' (r): specifying the time span of the window;

- A 'slide' (s): determining the interval for result updates;
- A 'payment' (π): representing the per-unit-time compensation to the seller.

Our mechanism primarily addresses distributive aggregate functions, which are formally defined as follows:

Definition 2 (Distributive Function). A function $F(\cdot)$ is distributive if there exists a function $G(\cdot)$ such that for any set $X_{1\cdots n} = \{x_1, \ldots, x_n\}$ divided into two subsets $X_{1\cdots i} = \{x_1, \ldots, x_i\}$ and $X_{i+1\cdots n} = \{x_{i+1}, \ldots, x_n\}$:

$$F(X_{1\cdots n}) = G(F(X_{1\cdots i}), F(X_{i+1\cdots n}))$$

Common examples of distributive functions include max, sum, and $count$. Our focus on distributive functions is strategic rather than limiting. Besides distributive functions, aggregation functions can be further categorized into algebraic and holistic types. Algebraic functions, such as $average$, are computed by applying additional operations to the outputs of distributive functions. For example, $average$ is derived by dividing sum by $count$, both of which are distributive. Integrating algebraic functions into our trading mechanism involves incorporating the cost of these additional operations into the query plan, as detailed in the cost modeling section. In contrast, holistic functions, such as $median$, require access to the entire dataset and do not support computation sharing. These functions typically demand specialized algorithms that are highly context-dependent and not amenable to generalization [14]. Therefore, this work does not address holistic aggregation functions.

The seller receives multiple CQs from customers, each with different time window specifications (i.e., ranges and slides) under the same pre-aggregation filters (i.e., aggregate functions and group-by attributes). *For handling queries with different pre-aggregation filters, we can directly integrate existing diverse mechanisms designed while retaining their desired properties [15], [19], as computation sharing optimization is orthogonal to pre-aggregation filter optimization.* We assume that CQs submitted to the trading system satisfy: $s \in \{2^i | i \in \mathbb{N}\}$ and r is divisible by s, which largely covers practical CQ patterns [25]. Furthermore, in line with [25], [41], we assume data arrives with rate λ in order.

2) Query Plan Modeling: The seller constructs multiple query plans [42] based on the trading strategy outlined in Section III-C, specifying how to execute the CQs in set $\mathcal{Q}$. We begin with the formal definition of query plan:

Definition 3 (Query Plan). A query plan ($p \subseteq \mathcal{Q}$) represents a set of continuous queries executed in one container, with a computation sharing type $t \in \mathcal{T}$ indicating how to share computation.

Only CQs within the same plan can share computation. The types of computation sharing ($\mathcal{T} = \{T_{sm}, T_{sn}, T_{nn}\}$) are categorized as follows:[4]

- T_{sm}: Plans that support both self-sharing and mutual-sharing computation.

[4]We do not consider plan types that support only mutual-sharing computation, as mutual-sharing builds upon the principles of self-sharing, as discussed in Section IV-B. Without self-sharing, different windows in the same CQ will couple with each other, preventing other CQs from reusing results produced by non-self-sharing CQs.

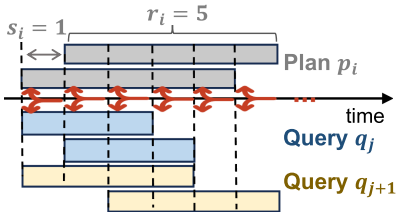


Fig. 6. The plan slide and range.

- T_{sn}: Plans that support only self-sharing computation.
- T_{nn}: Plans that do not support any computation sharing.

The seller predefines a set of empty plans $\mathcal{P} = \{p_1, p_2, \ldots\}$ based on system capacity [43].[5] Each plan $p_i \in \mathcal{P}$ is allocated a processing budget (B_i^p) and a memory budget (B_i^m).

For plans $p_i \in \mathcal{P}$ of type T_{sm}, CQs within the same plan may have different slide and range values. To enable computation sharing, the system assigns each plan a predefined *plan slide* (s_i) and *plan range* (r_i), as shown in Fig. 6. Only CQs whose slide is no smaller than s_i and range no larger than r_i can be grouped under the same plan. This design allows eligible CQs, despite differing in window parameters, to share a common set of partial results. Final aggregation is then performed separately for each CQ based on these shared partials. For clarity, we use subscript i for plans and j for queries, as summarized in Table I.

For plans with sharing types T_{sn} and T_{nn}, CQs are processed separately, removing the need for multiple plan configurations and allowing a single predefined plan for each type, thereby optimizing resource integration. Besides, the mechanism includes a plan for CQs that are not executed. To simplify notation, we define $K = M + 3$ as the total number of predefined plans, where M denotes the number of plans with type T_{sm}, and the "3" accounts for one plan each for type T_{sn}, type T_{nn}, and the non-execution plan, ensuring that all plan types are represented in the predefined plan set.

3) System Cost Modeling: Drawing from industry practices in Amazon Web Services (AWS) [44] and Google Cloud [45], we decompose the execution costs into processing cost $c_p(\cdot)$ and memory cost $c_m(\cdot)$.[6] The processing cost primarily reflects CPU computations and is proportional to the number of aggregated data points per time unit, scaled by the coefficient θ_p [44], [45]. The memory cost corresponds to the storage of computational results and scales linearly with the number of stored results, determined by the coefficient θ_m [44], [45]. Communication costs are excluded from this model as they are not coupled across

[5]The predefined set of empty plans reflects practical system capacity constraints, as many stream processing systems pre-allocate task slots or containers based on available resources. This design reduces scheduling latency and operational overhead, transforming the trading strategy into a query-to-plan allocation problem under fixed system budgets.

[6]These components represent the primary drivers of resource consumption in practical systems. They are explicitly instantiated under linear, tree-based, and queue-based strategies (see Section III-B), and are integrated into the unified profit model through equations (1) and (2). This modular and extensible design ensures that the mechanism accommodates diverse execution settings and aligns with pricing schemes commonly adopted in real-world cloud-based streaming services.

windows and queries and can be directly deducted from the payments.

The system's total cost comprises the costs of all query plans, which vary with their sharing types. In the following, we analyze the cost of each plan under different sharing types.

- For plans $p_i \in \mathcal{P}$ of type T_{sm}, which involve both self-sharing and mutual-sharing computations, the processing cost includes the *partial aggregation cost* for aggregating shared partial results and the *final aggregation cost* for CQ to aggregate the final results. Partial aggregation occurs every s_i time units with λs_i data points, resulting in a cost of $\theta_p \lambda$ (i.e., $\theta_p \lambda s_i / s_i$). The final aggregation cost is the sum of the costs for aggregating each CQ's final results based on the partials, which is θ_p multiplied by the number of times all partial results are combined into the final aggregation per unit time. Similar to [46], [47], we define the final aggregation cost for plan p_i as $\theta_p E(p_i)\Psi(p_i)$, where $E(p_i)$ is the number of partials generated per unit time, and $\Psi(p_i)$ is the number of final aggregations performed on each partial. Therefore, the processing cost of plan p_i with type T_{sm} is:

$$C_p(p_i) = \theta_p \left(\lambda + E(p_i)\Psi(p_i)\right), \tag{1}$$

  which is consistent with the analysis in Section IV. The memory cost of plan p_i with type T_{sm} depends on the number of intermediate results stored for sharing, denoted as $\Gamma(p_i)$, and is expressed as:

$$C_m(p_i) = \theta_m \Gamma(p_i). \tag{2}$$

- For plans $p_i \in \mathcal{P}$ of types T_{sn} and T_{nn}, since no mutual-sharing computations occur, the plan cost is the sum of the costs of executing each CQ separately. For type T_{sn}, the separate execution of a CQ is a special case of plan execution, where the cost of separately executing q_j is equal to the cost of a plan with $s_i = s_j$ and $r_i = r_j$ that exclusively executes q_j. For type T_{nn}, without any computation sharing, the separate CQ cost for each technique is the sum of c_p^n and c_m^n as detailed in equations (14) and (15) in Section IV-A. For the non-execution plan, both cost and profit are zero.

However, different computation-sharing techniques yield different formulations for $\Psi(\cdot)$ and $\Gamma(\cdot)$. Despite these differences, all techniques share a common expression for the partial aggregation rate: $E(p_i) = 1/s_i$, meaning that one partial result is generated every s_i time units. In the following, we introduce three widely used computation-sharing techniques and describe their respective cost models.

B. Computation Sharing Techniques

This section introduces three widely adopted computation-sharing techniques used in practical applications [14]: linear, tree-based, and queue-based approaches, as illustrated in Fig. 7. We first detail their respective aggregation processes in Section III-B1 and then provide a comprehensive analysis of their cost modeling in Section III-B2.

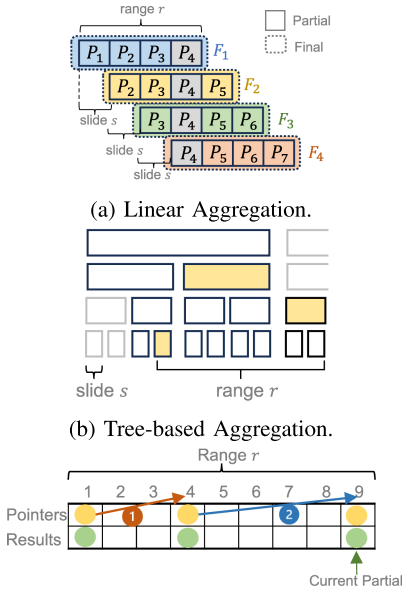


Fig. 7. Computation-sharing techniques for CQ execution.

1) Aggregation Process in Different Techniques: The execution of CQs involves two fundamental operations: *partial aggregation* and *final aggregation*. As shown in Fig. 7(a), the system divides the data stream into smaller segments, referred to as *partials*, based on time window boundaries. It computes an intermediate result for each partial, which is stored for later use. During final aggregation, only the partials relevant to each time window are combined to compute the full result. Each incoming data point affects only the partials it belongs to. In practice, the choice of an appropriate technique primarily depends on system capabilities and query characteristics.

The three techniques implement aggregations as follows:

- Linear aggregation directly reuses each partial multiple times across different final aggregations. For example, in Fig. 7(a), partial P_4 contributes to four final aggregations (i.e., F_1–F_4).
- Tree-based aggregation organizes partials into a binary tree, where the interval length doubles at each level, starting with one partial at the bottom. For final aggregation, the system selects the minimal set of intervals needed to cover the CQ range, as illustrated in Fig. 7(b) with marked intervals.
- Queue-based aggregation leverages two circular queues: one for storing partial results and another for pointers that guide calculation skips. As shown in Fig. 7(c), the system processes a CQ spanning positions 1 to 9 by following the pointers and aggregating the relevant partials.

While these computation-sharing techniques are well-established in practical implementations, their theoretical foundations—particularly in cost modeling and optimal sharing strategies—remain underexplored. This gap is particularly important for continuous query-based data trading, where economic efficiency and execution performance must be jointly

optimized. To bridge this gap, our work makes the following contributions: (1) a comprehensive cost modeling analysis and the formulation of a general optimization problem, adaptable to various computation-sharing techniques, as presented in Section III-C; (2) the development of closed-form optimal sharing strategies in Section IV; and (3) an efficient solution to the general optimization problem in Section V.

2) Cost Modeling in Different Techniques: This section derives the components $\Psi(p_i)$ and $\Gamma(p_i)$ from equations (1) and (2) for different computation-sharing techniques, enabling overall system cost computation.[7] Note that $E(p_i) = s_i$ remains constant across all techniques, as discussed in Section III-A2.

a) Linear Aggregation ($\mathfrak{L}$): For a plan p_i utilizing linear aggregation technique $\mathfrak{L}$, each CQ $q_j \in p_i$ performs $\gamma_j = r_j/s_j$ final aggregations on each partial. Consequently:

$$\Psi(p_i)^{\mathfrak{L}} = \sum_{q_j \in p_i} \gamma_j, \tag{3}$$

The storage requirement corresponds to $\gamma_i = r_i/s_i$ partials:

$$\Gamma(p_i)^{\mathfrak{L}} = \gamma_i. \tag{4}$$

b) Tree-based Aggregation ($\mathfrak{T}$): For a plan p_i using the tree-based technique $\mathfrak{T}$, the number of levels in a binary tree is $\log_2(\gamma_i) + 1$, where $\gamma_i = r_i/s_i$ represents the number of leaf nodes. For each CQ q_j covering $\gamma_j = r_j/s_j$ partials, the final aggregation involves a minimal set of internal tree nodes. Based on the properties of binary trees, each query q_j contributes $\lfloor \log_2(\gamma_j - 1) \rfloor + \frac{\gamma_j - 1}{2^{\lceil \log_2(\gamma_j) \rceil}}$ operations to the final aggregation. Thus, we have:

$$\Psi(p_i)^{\mathfrak{T}} = \sum_{q_j \in p_i} \lfloor \log_2(\gamma_j - 1) \rfloor + \frac{\gamma_j - 1}{2^{\lceil \log_2(\gamma_j) \rceil}}. \tag{5}$$

For quick approximations, $\Psi(p_i)^{\mathfrak{T}}$ can be estimated as $\sum_{q_j \in p_i} (\log_2(\gamma_j - 1) + 1)$. The system also needs to store the results for all tree nodes. According to the binary tree property, there are $2\gamma_i - 1$ nodes in total, leading to:

$$\Gamma(p_i)^{\mathfrak{T}} = 2\gamma_i - 1. \tag{6}$$

c) Queue-based Aggregation ($\mathfrak{Q}$): For a plan p_i executed with the queue-based technique $\mathfrak{Q}$, the uncertainty caused by pointer skipping makes direct calculation of factors challenging. However, simulation results [48] suggest that each CQ performs approximately three final aggregations per partial. Thus, we have:

$$\Psi(p_i)^{\mathfrak{Q}} = 3|p_i|, \tag{7}$$

where $|p_i|$ denotes the number of CQs in plan p_i. The system requires two queues to store the γ_i partials and their corresponding pointer values, resulting in:

$$\Gamma(p_i)^{\mathfrak{Q}} = 2\gamma_i. \tag{8}$$

[7]We model three representative aggregation strategies, namely linear, tree-based, and queue-based aggregation, each with distinct processing and memory cost formulations defined in equations (3) through (8). These models capture common execution patterns in stream query engines and can be integrated into the general cost expressions in equations (1) and (2).

Different aggregation techniques lead to different cost formulations, making it challenging to develop a unified trading strategy. To address this, we identify a *common cost structure* that underlies all three techniques. Based on this insight, we formulate a general optimization problem that adapts to each technique and aims to maximize the seller's profit. This problem will be introduced in detail in the next section.

C. Formulation of the General Problem

We formulate a general optimization problem that is adaptable to different techniques, aiming to determine the optimal trading strategy and maximize the seller's profit. The profit is calculated as the payments for executed CQs minus the plan costs. Despite the diverse cost structures introduced by different aggregation techniques, we identify a *common cost structure* that facilitates a unified optimization approach.

As discussed in Section III-A3, the plan processing cost consists of two components: (1) the partial aggregation cost, $\theta_p \lambda$, for the newly arrived data points, which is shared among all CQs in the same plan, and (2) the final aggregation cost, $\theta_p(E(p_i)\Psi(p_i))$, which varies based on the final aggregation required for different CQs.

This cost structure reveals a consistent pattern across different computation-sharing techniques. The total cost for each technique can be decomposed into two distinct elements: a plan-level *start-up cost* and *query-specific costs*. The plan-level start-up cost includes both partial aggregation expenses and memory overhead, shared across the entire plan. In contrast, query-specific costs reflect the individual processing requirements of each CQ within the plan, varying according to their specific computational demands.

1) A General Optimization Problem for Different Techniques: Based on this common structure, we formulate a general mixed-integer programming problem.[8] applicable across different computation sharing techniques, introducing two key binary decision variables:

- $\boldsymbol{x} = [x_{ij}]$: a binary matrix of size $K \times N$, where each element $x_{ij} \in \{0, 1\}$ indicates whether continuous query (CQ) q_j is assigned to plan p_i.
- $\boldsymbol{y} = [y_i]$: a binary vector of size $K \times 1$, where each element $y_i \in \{0, 1\}$ indicates whether plan p_i is activated (i.e., $y_i = 1$ if any query is assigned to plan p_i).

The profit maximization problem ($\mathbb{P}_1$) can be expressed as:

$$\begin{aligned} &\max_{\boldsymbol{x},\boldsymbol{y}} \sum_{i=1}^{K} \sum_{j=1}^{N} x_{ij} a_{ij} - \sum_{i=1}^{K} C_i y_i \\ &s.t. \sum_{j=1}^{N} x_{ij} \boldsymbol{w}_{ij} \leq \boldsymbol{W}_i y_i, \text{ for } i \in \{1, \ldots, K\}, \\ &\sum_{i=1}^{K} x_{ij} = 1, \text{ for } j \in \{1, \ldots, N\}, \end{aligned} \tag{$\mathbb{P}_1$}$$

[8]The optimization formulation captures core characteristics of continuous query markets by integrating pricing decisions, plan constraints, and computation sharing benefits. These elements jointly enable strategic query admission under budget and resource limits, while maintaining a modular structure.

$$x_{ij} \in \{0,1\}, \text{ for } i \in \{1,\ldots,K\}, j \in \{1,\ldots,N\},$$
$$y_i \in \{0,1\}, \text{ for } i \in \{1,\ldots,K\},$$

where a_{ij} is the CQ net profit from allocating CQ q_j to plan p_i, C_i is the start-up cost for plan p_i, $\boldsymbol{w}_{ij}$ is query-specific cost of allocating CQ q_j to plan p_i, and $\boldsymbol{W}_i$ is the capacity of plan p_i. Note that $a_{ij}s$, C_is, $\boldsymbol{w}_{ij}s$, and $\boldsymbol{W}_{ij}s$ vary according to the specific computation-sharing technique employed.

2) Problem Formulation Process: We illustrate the detailed problem formulation process using linear aggregation as an example, while the formulations for tree-based and queue-based aggregation techniques are provided in Appendix A due to space constraints. For a plan p_i of types T_{ms} and T_{sn}, the start-up cost is defined as:

$$C_i = \theta_p\lambda + \theta_m\gamma_i, \tag{9}$$

which consists of the partial aggregation cost and the memory cost for storing partial results. Let $\boldsymbol{W}_i = [B_i^p - \theta_p\lambda, B_i^m - \theta_m\gamma_i]$ represent the cost budget remaining for query plan p_i, including the processing and memory budgets available for the CQs after deducting the corresponding start-up costs. For each CQ $q_j \in p_i$, there is a final aggregation cost (i.e., query-specific cost) $\theta_p\gamma_j/s_i$, consistent with equation(3), and a corresponding payment π_j. Thus, the CQ net profit a_{ij} of allocating CQ q_j to plan p_i is defined as:

$$a_{ij} = \begin{cases} \pi_j - \theta_p\gamma_j/s_i, & \text{if } s_i \le s_j \text{ and } r_i \ge r_j, \\ 0, & \text{otherwise}, \end{cases} \tag{10}$$

and the corresponding continuous query-specific cost vectors (with the first dimension representing processing cost and the second representing memory) are defined as follows:

$$\boldsymbol{w}_{ij} = \begin{cases} [\theta_p\gamma_j/s_i, 0], & \text{if } s_i \le s_j \text{ and } r_i \ge r_j, \\ [L, L], & \text{otherwise}, \end{cases} \tag{11}$$

where L is a large constant penalty term that discourages the selection of CQs with small slides or large ranges, as discussed in Section III-A. Note that the query-specific memory cost is zero since memory costs for partial results are shared among CQs within a plan and included in the start-up cost.

For a plan p_i of type T_{nn}, where no computation sharing occurs, the start-up cost is $C_i = 0$. The query-specific cost corresponds to the CQ cost without computation sharing, as detailed in equations (14) and (15). Consequently, for a CQ $q_j \in p_i$, the net profit is given by:

$$a_{ij} = \pi_j - \theta_p\gamma_j - \theta_m. \tag{12}$$

The cost budget for query plan p_i is defined as $\boldsymbol{W}_i = [B_i^p, B_i^m]$, and the query-specific cost is expressed as:

$$\boldsymbol{w}_{ij} = [\theta_p\gamma_j, \theta_m]. \tag{13}$$

For the null plan p_i (where $i = M + 3$), both the start-up cost C_i and the query-specific cost are zero. Consequently, for each CQ q_j, the net profit of assigning CQ q_j to the null plan p_i is given as $a_{ij} = 0$. The cost budget for query plan p_i is defined as $\boldsymbol{W}_i = [L, L]$, permitting the plan to accommodate any number of CQs. The query-specific cost for any CQ q_j is expressed as $\boldsymbol{w}_{ij} = [0, 0]$.

In this section, we have presented the system model, encompassing the trading system design, computation-sharing techniques, and problem formulation. The derivation of an optimal trading strategy under computation sharing is challenging due to the coupling effects across different time windows (i.e., self-sharing) and different queries (i.e., mutual-sharing). In the following section, we present a theoretical analysis of computation sharing to establish foundations and provide insights for deriving the optimal trading strategy.

IV. Computation Sharing Analysis

Building upon the preliminaries established in Section III, we derive the optimal self-sharing computation strategy in Section IV-A and the optimal mutual-sharing computation strategy in Section IV-B, using linear aggregation techniques. These analytical results provide crucial insights that guide the design of our general trading mechanism. We then extend this analysis to encompass tree-based and queue-based aggregation techniques in Section V, addressing the general optimization problem across different computation-sharing techniques.[9]

A. Analysis of Self-Sharing

For self-sharing computation, we analyze the cost modeling and derive the optimal sharing strategy to maximize profit, focusing on a single CQ $q = (s, r, \pi)$. Three trading strategies are considered for CQ execution: non-sharing trading strategy $\mathcal{P}_{ns}$, sharing trading strategy $\mathcal{P}_s$, and null trading strategy $\mathcal{P}_{null}$ (i.e., no execution).

a) Cost Modeling: For query execution, the seller can either implement computation sharing or not. In the non-sharing scenario, the system performs periodic aggregation of a single result from λr data points within a time window of duration r every s time units. The processing cost for non-sharing execution is:

$$c_p^n(q) = \theta_p \cdot \lambda\gamma, \tag{14}$$

where $\gamma = r/s$ represents the overlap factor. The corresponding memory cost is:

$$c_m^n(q) = \theta_m \cdot 1. \tag{15}$$

reflecting the storage requirement for one computational result per time window

b) Strategy Comparison: Next, we analyze the costs and profits for each strategy.

- For $\mathcal{P}_{ns}$, the profit of the strategy is:
$$\pi - c_p^n(q) - c_m^n(q), \tag{16}$$
utilizing the non-sharing costs defined in (14) and (15).
- For $\mathcal{P}_s$, the processing cost comes from both partial and final aggregations. The system aggregates $s\lambda$ data points

[9]While semantic caching [49], [50], [51] has been studied for query optimization in distributed and ad hoc settings, its applicability to streaming continuous queries is limited by the need for fresh computation over evolving sliding windows. In contrast, our mechanism employs computation block sharing to reuse partial aggregations during execution and incorporates cost-aware plan selection under resource constraints.

for partial aggregation and $\gamma = r/s$ partial results for final aggregation every s time interval. The processing cost is:

$$c_p^s(q) = \theta_p(s\lambda + \gamma)/s, \tag{17}$$

With γ partial results stored, the memory cost becomes:

$$c_m^s(q) = \theta_m \gamma. \tag{18}$$

yielding a profit of:

$$\pi - c_p^s(q) - c_m^s(q). \tag{19}$$

- For $\mathcal{P}_{null}$, it yields zero profit and incurs no cost.

For analytical clarity, we normalize the data arrival rate to $\lambda = 1$.[10] By comparing the profits of these trading strategies, we derive the optimal self-sharing strategy as follows.

Lemma 1 (Optimal Strategy for Self-sharing Computation). For self-sharing computation, the optimal strategy for profit maximization is one of the following: (1) the non-sharing strategy $\mathcal{P}_{ns}$, (2) the sharing strategy $\mathcal{P}_s$, or (3) the null strategy $\mathcal{P}_n$. Different system parameters lead to different optimal strategies $\mathcal{P}^*$, determined as follows:

- If $\pi \leq \bar{\pi}$, then:

$$\mathcal{P}^* = \begin{cases} \mathcal{P}_{ns}, & \text{if } r \leq \check{r}, \\ \mathcal{P}_n, & \text{otherwise;} \end{cases} \tag{20}$$

- If $\pi > \bar{\pi}$, then:

$$\mathcal{P}^* = \begin{cases} \mathcal{P}_s, & \text{if } \theta_p > \theta_m,\ s > \bar{s}, \text{ and } \bar{r} < r \leq \hat{r}; \\ \mathcal{P}_{ns}, & \text{if } \theta_p > \theta_m,\ s > \bar{s}, \text{ and } r \leq \bar{r}, \\ & \text{or} \theta_p \leq \theta_m; \\ \mathcal{P}_n, & \text{otherwise;} \end{cases}$$

where $\bar{r} = \frac{s^2(\theta_m - \theta_p)}{(\theta_m - \theta_p)s + \theta_p}$, $\check{r} = \frac{s(\pi - \theta_m)}{\theta_p}$, $\hat{r} = \frac{s^2(\pi - \theta_p)}{\theta_p + \theta_m s}$, $\bar{s} = \frac{\theta_p}{\theta_p - \theta_m}$, and $\bar{\pi} = \frac{\theta_p\theta_m + \theta_m^2 s - \theta_p^2 s}{\theta_p + \theta_m s - \theta_p s}$.

The detailed proof is provided in Appendix B, from which we derive two significant insights:

Observation 1 (Limited Profit of Self-sharing). The self-sharing strategy P_s does not universally guarantee profit maximization, particularly in low-payment scenarios ($\pi \leq \bar{\pi}$).

This occurs because, with smaller payments, the maximum range allowing optimal strategy implementation becomes too narrow to effectively leverage computation-sharing benefits.

Observation 2 (Parameter Effect). Strategy efficiency exhibits a complex relationship with the slide parameter s. For P_{ns}, effectiveness first decreases as s increases, particularly when $s < 2\bar{s}$ and $\partial r/\partial s < 0$, before eventually increasing.

This relationship highlights the importance of careful trading strategy design that considers appropriate parameter trade-offs.

B. Analysis of Mutual-Sharing

We now analyze the cost modeling and optimal mutual-sharing strategy in a two-CQ case, where $q_1 = (r_1, s_1, \pi_1)$ and $q_2 = (r_2, s_2, \pi_2)$. Notably, mutual-sharing builds upon self-sharing[11] in computation-sharing techniques, as shown in Fig. 8. Hence, we examine cases where all queries inherently involve self-sharing. Without loss of generality, we assume $s_1 \leq s_2$. Accordingly, we consider five trading strategies:

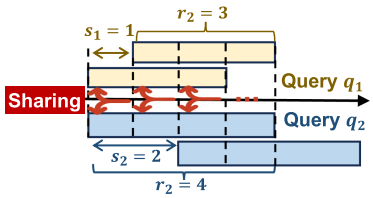


Fig. 8. The mutual-sharing strategy: $\mathcal{P}_{ms}$.

- Mutual-sharing strategy $\mathcal{P}_{ms}$=$\{\{q_1, q_2\}\}$ executes both CQs with mutual-sharing computation. The shortest slide (i.e., s_1) determines the common partial length, as shown in Fig. 8. The processing cost comes from partial aggregations of λs_1 data points every s_1 time unit and final aggregations for both queries. Each partial is used $\gamma_1 = r_1/s_1$ and $\gamma_2 = r_2/s_2$ times for q_1 and q_2 in the final aggregations, leading to γ_1 and γ_2 partial results aggregated every s_1 time units for q_1 and q_2. The memory cost depends on the number of partial results stored and used in final aggregations for the two CQs, which is the maximum range divided by s_1. Hence, if $r_1 \leq r_2$, the profit is calculated as follows:

$$R_{ms} = \pi_1 + \pi_2 - \theta_p(\lambda s_1 + (\gamma_1 + \gamma_2))/s_1 - \theta_m(r_2/s_1). \tag{21}$$

- Double self-sharing strategy $\mathcal{P}_{ss}$=$\{\{q_1\}, \{q_2\}\}$ executes both CQs separately, where the cost is the sum of the separate self-sharing cost defined in (17) and (18), leading to the profit:

$$R_{ss} = \sum_{i=1,2} \left(\pi_i - c_p^s(q_i) - c_m^s(q_i)\right). \tag{22}$$

- Single self-sharing strategy I $\mathcal{P}_{s1} = \{\{q_1\}\}$ only executes the first CQ (q_1), with the profit $R_{s1} = \pi_1 - c_p^s(q_1) - c_m^s(q_1)$.
- Single self-sharing strategy II $\mathcal{P}_{s2} = \{\{q_2\}\}$ only executes the second CQ (q_2), with the profit being $R_{s2} = \pi_2 - c_p^s(q_2) - c_m^s(q_2)$.
- Null strategy $\mathcal{P}_n = \emptyset$ executes no queries, with zero profit.

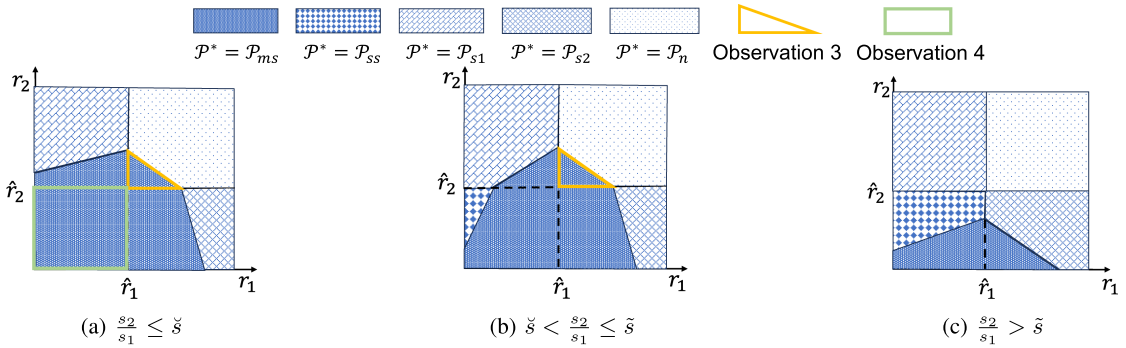
To determine the optimal strategy for mutual-sharing computation, we first compare P_{ms} and P_{ss}. When $R_{ms} \leq R_{ss}$, comparing (21) and (22) yields the condition for r_2:

$$r_2 \leq \grave{r}_2 = \frac{\theta_p + \theta_m\gamma_1}{\frac{\theta_p}{s_1 s_2} - \frac{\theta_p}{s_2^2} + \frac{\theta_m}{s_1} - \frac{\theta_m}{s_2}}, \tag{23}$$

which indicates that R_{ms} yields higher profit when r_2 is small.

[10] This normalization corresponds to setting the unit of system time to $1/v$ seconds when the actual arrival rate is v data points per second. In this way, $\lambda = 1$ denotes one data point per normalized time unit. This normalization does not affect the generality of the results, as processing costs scale linearly with λ, and the optimal plan selection logic remains invariant.

[11] Without self-sharing, different windows in the same CQ will couple with each other, and other queries cannot reuse the results from the non-self-sharing queries. In Fig. 8, both sharing types emerge.



(a) $\frac{s_2}{s_1} \leq \breve{s}$ (b) $\breve{s} < \frac{s_2}{s_1} \leq \tilde{s}$ (c) $\frac{s_2}{s_1} > \tilde{s}$

Fig. 9. The optimal strategy for mutual-sharing computation in the case $(r_1 \leq r_2)$, where $\breve{s} = \min(\frac{\pi_2}{\pi_2-\theta_p}, \frac{\theta_p^2+\theta_m s_1 \pi_1}{(\pi_2-\theta_p)(\theta_p+\theta_m s_1)} + 1)$ and $\tilde{s} = \max(\frac{\pi_2}{\pi_2-\theta_p}, \frac{\theta_p^2+\theta_m s_1 \pi_1}{(\pi_2-\theta_p)(\theta_p+\theta_m s_1)} + 1)$.

For strategies involving separate execution with self-sharing computation (P_{ss}, P_{s1}, P_{s2}, and P_{ss}), each CQ executes only if its payments cover the separate self-sharing costs defined in (17) and (18). Specifically, CQ q_i executes when $\pi \geq c_p^s(q_i) + c_m^s(q_i)$, which is equivalent to the following condition:

$$r_i \leq \hat{r}_i = \frac{s_i^2(\pi_i - \theta_p)}{\theta_p + \theta_m s_1}, \tag{24}$$

indicating that CQ q_i will be executed when r_i is small.

Based on our analysis, we present Lemma 2, which characterizes the optimal mutual-sharing strategy. With a similar induction, we present the optimal strategy for the case where $r_1 > r_2$ in Appendix C.

Lemma 2 (Optimal Strategy for Mutual-Sharing Computation). For mutual-sharing computation, there are five potentially optimal trading strategies for profit maximization in the case where $r_1 \leq r_2$: (1) Mutual-sharing strategy $\mathcal{P}_{ms}$, (2) Double self-sharing strategy $\mathcal{P}_{ss}$, (3) Single self-sharing strategy I $\mathcal{P}_{s1}$, (4) Single self-sharing strategy II $\mathcal{P}_{s2}$, and (5) Null strategy $\mathcal{P}_n$. These optimal strategies are mutually exclusive. Based on the system parameters, the optimal trading strategy $\mathcal{P}^*$ is as follows:

$$\mathcal{P}^* = \begin{cases} \mathcal{P}_{ss}, & \text{if } r_1 \leq \hat{r}_1 \text{ and } \grave{r}_2 < r_2 \leq \hat{r}_2, \\ \mathcal{P}_{s2}, & \text{if } r_1 > \hat{r}_1 \text{ and } \acute{r}_2 < r_2 \leq \hat{r}_2, \\ \mathcal{P}_{s1}, & \text{if } r_1 \leq \hat{r}_1 \text{ and } r_2 > \max(\tilde{r}_2, \hat{r}_2), \\ \mathcal{P}_{n}, & \text{if } r_1 > \hat{r}_1 \text{ and } r_2 > \max(\breve{r}_2, \hat{r}_2), \\ \mathcal{P}_{ms}, & \text{otherwise}, \end{cases} \tag{25}$$

where $\grave{r}_2 = \frac{\theta_p+\theta_m\gamma_1}{(\frac{1}{s_1}-\frac{1}{s_2})(\frac{\theta_p}{s_2}+\theta_m)}$, $\acute{r}_2 = \frac{\pi_1-\theta_p\frac{\gamma_1}{s_1}}{(\frac{1}{s_1}-\frac{1}{s_2})(\frac{\theta_p}{s_2}+\theta_m)}$, $\tilde{r}_2 = \frac{\pi_2+\theta_m\gamma_1}{\frac{1}{s_1}(\frac{\theta_p}{s_2}+\theta_m)}$, and $\breve{r}_2 = \frac{\pi_1+\pi_2-\theta_p-\theta_p\frac{\gamma_1}{s_1}}{\frac{1}{s_1}(\frac{\theta_p}{s_2}+\theta_m)}$.

To clarify Lemma 2, we visualize the optimal strategy region in Fig. 9 by considering r_1 and r_2 as variables and categorizing cases by the s_2/s_1 ratio. We get the following counter-intuitive observations:

Observation 3 (Synergistic Effects). Queries that are not profitable when executed separately may become profitable when executed together using mutual-sharing computation.

For CQ q_i, if $r_i > \hat{r}_i$, then the payment π_i cannot cover the cost of separate execution as illustrated by (24). However, in Fig. 9(a) and (b), the region in the yellow triangle (i.e., $r_1 > \hat{r}_1$ and $\hat{r}_2 < r_2 \leq \breve{r}_2$) shows that even if the payment of each CQ cannot cover its separate cost (i.e., $r_1 > \hat{r}_1$ and $r_2 > \hat{r}_2$), the optimal strategy can make profits by executing all of them. This indicates that the mutual-sharing computation boosts both the success rate and profit of data trading, which is also validated in our experiment part in Section VI.

Observation 4 (Dominance of Mutual-sharing). With small slide ratio s_2/s_1, even if the payment of each CQ can cover its separate cost, the double self-sharing strategy $\mathcal{P}_{ss}$ will never be the optimal strategy.

The double self-sharing strategy $\mathcal{P}_{ss}$ has the potential to be the optimal plan only when each CQ can cover its separate cost (i.e., $r_1 \leq \hat{r}_1$ and $r_2 \leq \hat{r}_2$). However, as shown in the region in the green box in Fig. 9(a), mutual-sharing plan $\mathcal{P}_{ms}$ is always better than $\mathcal{P}_{ss}$ when $r_1 \leq \hat{r}_1$ and $r_2 \leq \hat{r}_2$. This is because the mutual-sharing plan $\mathcal{P}_{ms}$ benefits from small s_2/s_1. With a smaller s_2/s_1, the final aggregations of q_2 in strategy $\mathcal{P}_{ms}$ cover fewer partials, which can benefit largely from mutual-sharing computation. We put the detailed proof and more observations in Appendix C.

Takeaway: (1) Self-sharing computation cannot always bring positive profit, especially with low payment. (2) Mutual-sharing computation cannot always bring positive profit either, particularly when the slide ratio is high.

This section has presented a theoretical analysis of computation sharing, covering self- and mutual-sharing types, and has derived optimal strategies and key insights. This analysis forms the foundation for continuous query-based data trading. Based on these cases, we proceed to solve the general trading optimization problem $\mathbb{P}_1$ in the following section.

V. Optimal Trading Strategy

This section develops a tailored solution to problem $\mathbb{P}_1$ for obtaining the optimal trading strategy through three steps:

1) **Problem Reformulation** (Section V-A): We apply Dantzig-Wolfe decomposition [52] to transform $\mathbb{P}_1$ into an equivalent problem $\mathbb{P}_2$. This reformulation decomposes

the large-scale problem into more tractable subproblems, improving computational efficiency

2) **Relaxation Solution** (Section V-B): We employ column generation to solve the linear programming (LP) relaxation of $\mathbb{P}_2$. This intermediate step provides both lower bounds for the branch-and-bound process and high-quality initial solutions for the integer program.
3) **Integer Solution** (Section V-C): We develop a customized branch-and-price framework that combines branch-and-bound with dynamic column generation to enforce integer constraints while maintaining computational efficiency.

A. Problem Reformulation: Dantzig-Wolfe Decomposition

In this subsection, we employ Dantzig-Wolfe Decomposition to reformulate the original problem into a tighter formulation—one where the relaxed solution frequently satisfies integer constraints naturally, thus accelerating the branching process convergence [52]. This decomposition divides the complex problem into more tractable subproblems, improving both efficiency and solution quality. First, we define the feasible allocation pattern for a plan as follows:

Definition 4 (Feasible Allocation Pattern). A feasible allocation pattern $f \subseteq \mathcal{Q}$ for a plan is a set of CQs from $\mathcal{Q}$ that can be allocated into the plan while satisfying all constraints.

Let $\mathcal{F}^i$ be the set containing all feasible allocation patterns for plan p_i, and $f_l^i \in \mathcal{F}^i$ be the l-th feasible allocation pattern for $p_i \in \mathcal{P}$. Let $\mathcal{F} = \cup_{q_i \in \mathcal{P}} \mathcal{F}^i$ be the set containing all pattern sets. A feasible allocation pattern f_l^i is denoted by a vector $\boldsymbol{b}_{f_l^i}$ of indicator functions $b_{f_l^i}^j$ such that $b_{f_l^i}^j = 1$ if CQ q_j is executed in the pattern f_l^i. The profit of pattern f_l^i is then the difference between the total profit of CQs in that allocation pattern, subtracted by the start-up cost of the plan p_i:

$$h_{f_l^i} = \sum_{q_j \in \mathcal{Q}} a_{ij} b_{f_l^i}^j - C_i, \tag{26}$$

where a_{ij} is the CQ net profit and C_i is the plan start-up cost, as defined in Section III-A3. This leads to the following problem reformulation using Dantzig-Wolfe Decomposition:

Lemma 3 (Reformulation). Problem $\mathbb{P}_1$ can be reformulated as the following Problem $\mathbb{P}_2$:

$$\begin{aligned}
&\max_{\boldsymbol{\lambda}} \sum_{p_i \in \mathcal{P}} \sum_{f_l^i \in \mathcal{F}_i} h_{f_l^i} \lambda_{f_l^i} \\
&s.t. \sum_{p_i \in \mathcal{P}} \sum_{f_l^i \in \mathcal{F}_i} b_{f_l^i}^j \lambda_{f_l^i} = 1, \text{ for } j \in \{1, \ldots, N\}, \qquad (\mathbb{P}_2) \\
&\sum_{f_l^i \in \mathcal{F}_i} \lambda_{f_l^i} \leq 1, \text{ for } i \in \{1, \ldots, K\}, \\
&\lambda_{f_l^i} \in \{0, 1\}, \text{ for } f_l^i \in \mathcal{F}_i, i \in \{1, \ldots, K\}.
\end{aligned}$$

Therefore, based on the reformulation, the problem can be decomposed into the following problems: (1) a master problem $\mathbb{P}_2$, and (2) multiple subproblems to generate all the feasible allocation patterns.

While solving $\mathbb{P}_2$ with the complete set of feasible allocations $\mathcal{F}$ would yield the optimal solution, the exponential number of feasible patterns makes the direct solution impractical. To address this computational challenge, we propose a two-phase approach. First, we employ column generation to solve the LP relaxation of $\mathbb{P}_2$, dynamically generating patterns through iterative solutions (Section V-B). Then, we incorporate branch-and-bound to enforce integer constraints in our branch-and-price algorithm (Section V-C).

B. Relaxation Solution: Column Generation

We address Problem $\mathbb{P}_2$ through Linear Programming (LP) relaxation using column generation, which serves as the foundation for our branch-and-price framework. The solution procedure follows an iterative process that starts with an initial set of patterns. In each iteration, the subproblems generate only those allocation patterns with the potential to improve the current solution. A new pattern, if one exists, must yield a positive marginal profit among all possible patterns. If no such pattern exists, the procedure terminates, having found the optimal solution to the relaxed master problem. For each plan p_i, we define the marginal profit of a given pattern $f_l^i \in \mathcal{F}_i$ as:

$$\begin{aligned}
r_{f_l^i} &= \sum_{j=1}^{N} b_{f_l^i}^j u_j + h_{f_l^i} - \epsilon_i \\
&= \sum_{j=1}^{N} b_{f_l^i}^j u_j + \sum_{j=1}^{N} b_{f_l^i}^j a_{ij} - C_i - \epsilon_i,
\end{aligned} \tag{27}$$

where u_j represents the dual variable associated with CQ q_j in $\mathbb{P}_2$, indicating the marginal value of q_j to the solution, while ϵ_i is the dual variable for Plan p_i, capturing the allocation cost.

The restricted master problem contains significantly fewer variables than the original problem, enabling efficient solutions through linear programming algorithms such as the simplex method. This iterative column generation approach ensures optimal solution convergence [53]. We next detail the construction of the restricted master problem (Section V-B1) and the pricing subproblems (Section V-B2).

1) Restricted Master Problem: Problem $P_{res-master}$ operates on $\mathcal{F}' \subseteq \mathcal{F}$, where $\mathcal{F}_i' \subseteq \mathcal{F}_i$ contains the currently generated patterns for plan p_i along with initial patterns, updated iteratively. Therefore, the restricted master problem $\mathbb{P}_{res-master}$ is as follows:

$$\begin{aligned}
&\max_{\boldsymbol{\lambda}} \sum_{p_i \in \mathcal{P}} \sum_{f_l^i \in \mathcal{F}_i'} h_{f_l^i} \lambda_{f_l^i} \\
&s.t. \sum_{p_i \in \mathcal{P}} \sum_{f_l^i \in \mathcal{F}_i'} b_{f_l^i}^j \lambda_{f_l^i} = 1, \text{ for } j \in \{1, \ldots, N\}, \\
&\sum_{f_l^i \in \mathcal{F}_i} \lambda_{f_l^i} \leq 1, \text{ for } i \in \{1, \ldots, K\}, \\
&0 \leq \lambda_{f_l^i} \leq 1, \text{ for } f_l^i \in \mathcal{F}_i, i \in \{1, \ldots, K\}
\end{aligned}$$

$$(\mathbb{P}_{res-master}).$$

The key distinction between Problem $\mathbb{P}_{res-master}$ and the LP relaxation of Problem $\mathbb{P}_2$ lies in that $\mathbb{P}_{res-master}$ considers only $\mathcal{F}'$, a subset of feasible allocations, which substantially reduces the computational complexity.

2) Pricing Subproblems: For each plan $p_i \in \mathcal{P}$, Problem $P_{subproblem-i}$ generates feasible allocation patterns with the potential to increase profit. In each iteration, based on the dual variables in the restricted master problem, we need to find the patterns with maximum increase profit defined in (27) to join $\mathcal{F}'_i$ and enhance the current solution. Accordingly, for each plan $p_i \in \mathcal{P}$, we formulate the corresponding pricing subproblem $\mathbb{P}_{subproblem-i}$ as follows:

$$
\begin{aligned}
Z_i = \max_{\boldsymbol{x}} \quad & \sum_{q_j \in \mathcal{Q}} (a_{ij} + u_j) z_j \\
s.t. \quad & \sum_{q_j \in \mathcal{Q}} \boldsymbol{w}_{ij} z_j \leq \boldsymbol{W}_i, \\
& z_j \in \{0, 1\}, \text{ for } j \in \{1, \ldots, N\} \\
& \qquad\qquad\qquad\qquad (\mathbb{P}_{subproblem-i})
\end{aligned}
$$

where z_j is a binary variable that indicates whether allocates CQ q_j into the plan or not. Therefore, the potential increased profit of the optimal pattern of plan p_i is $Z_i - C_i - \epsilon_i$, where C_i is the start-up cost for the plan p_i as shown in Section III-A3 and ϵ_i is the dual variable of p_i as introduced in (27).

Since the subproblem for each plan is independent, we can deal with the subproblems parallelly. Only patterns yielding positive marginal profits are candidates for inclusion in the optimal solution. If any pattern with positive marginal profit exists, we select the one maximizing the profit to augment the set $\mathcal{F}'$ and incorporate it into the restricted master problem. Otherwise, the absence of profitable patterns indicates that the current solution cannot be further improved, signaling the algorithm's termination.

This column generation approach iteratively solves Problem $\mathbb{P}_2$ with LP relaxation. To satisfy integer requirements for variables $\lambda_{f_{il}}$, we implement the branch-and-price framework, detailed in the subsequent section.

C. Integer Solution: Branch-and-Price Framework

The branch-and-price framework integrates column generation with branch-and-bound techniques to satisfy integer constraints. This section details the algorithm design and implementation for solving Problem $\mathbb{P}_1$.

We first introduce the whole procedure of the proposed algorithm based on a binary search tree. We start with a heuristic initial solution and an initial pattern set $\mathcal{F}'$ at the root node, and each branch node inherits the feasible patterns from its parent. Each node first solves a mixed integer programming problem $\mathbb{P}_{res-master}$, which is additionally constrained by the branching rules of the current node. Based on the dual variable in $\mathbb{P}_{res-master}$, the current node then solves a set of subproblems $\mathbb{P}_{subproblem-i}$ to find the patterns that have the potential to improve the solution as we introduce in Section V-B. If patterns with positive increased profit are found, we add them to the current pattern set $\mathcal{F}'$ and improve the solution iteratively. Otherwise, we find the optimal solution under the rules of the current node. If it is integral and achieves higher profit than the lower bound, we update the lower bound. Otherwise, if the solution is lower than the lower bound, we stop branching in the current node, and if the solution is fractional, we branch the node according to the branching rule.

The effectiveness of the branch-and-price algorithm relies heavily on an appropriate branching rule. The rule should be tailored to specific problems to ensure it does not destroy the problem structure. Besides, the rule can be used in pricing subproblems to generate patterns while respecting branching rules effectively. According to [54], if the solution is fractional, then we can always find a CQ pair j and j' which satisfies:

$$
0 < \sum_{b^j_{f^i_l}=1, b^{j'}_{f^i_l}=1} \lambda_{f^i_l} < 1. \tag{28}
$$

This leads to two branching rules:

$$
\sum_{\forall i,l: b^j_{f^i_l}=1, b^{j'}_{f^i_l}=1} \lambda_{f^i_l} = 1, \tag{29}
$$

and

$$
\sum_{\forall i,l: b^j_{f^i_l}=1, b^{j'}_{f^i_l}=1} \lambda_{f^i_l} = 0. \tag{30}
$$

We create the left child node using rule (29) and the right child node using rule (30). By integrating the column generation and branching procedure, we derive Algorithm 1 to obtain the globally optimal trading strategy for Problem $\mathbb{P}_1$ that maximizes seller profits.

The branch-and-price algorithm guarantees global optimality for problem $\mathbb{P}_1$ by combining branch-and-bound with column generation to solve LP relaxations at each node. Convergence is ensured due to the finiteness of the solution space and termination of column generation when no improving columns remain. Although the worst-case complexity is exponential, empirical studies [55], [56] show that branch-and-price performs well in practice, particularly when exploiting structural sparsity. In our case, the sparse query-to-plan matrix and modular subproblem structure significantly reduce the effective search space.

In this section, we have introduced a branch-and-price framework tailored for Problem $\mathbb{P}_1$ to obtain the globally optimal trading strategy. The next section presents the experimental evaluation of the mechanism's performance.

VI. Experimental Evaluation

This section presents a comprehensive evaluation of the proposed trading mechanism through extensive numerical experiments under diverse scenarios. We begin in Section VI-A by outlining the experimental setting, followed by Section VI-B, which provides a detailed analysis and discussion of the experimental results.

Algorithm 1: The Branch-and-Price Algorithm.

```
Initialize a node queue Q_node and a lower bound;
Create the root node using a heuristic algorithm to
  generate an initial feasible allocation set;
Enqueue the root node into Q_node;
while Q_node is not empty do
    Dequeue a node n from Q_node and initialize F' with it parent patterns;
    Construct the Restricted Master Problem and the Pricing Subproblems for node n;
    repeat
        Solve the Restricted Master Problem to select allocation patterns from F';
        Solve the Pricing Subproblems to generate patterns for F';
    until No allocation with positive increased profit is found;
    if The solution achieves higher profit than the lower bound then
        if The solution is integral then
            Update the lower bound and record the solution;
        else
            Branch the node, adding left and right child node to Q_node;
Return the lower bound and corresponding solution;
```

A. Experimental Setting

Due to the lack of publicly available benchmark datasets for continuous query generation with economic parameters, we follow common practice in the CQ literature [57], [58], [59], [60]] and use synthetic query workloads to evaluate performance. To ensure practical relevance, our payment and cost models are calibrated using resource pricing data from commercial platforms such as Amazon's pricing model [44].

1) Continuous Query Generator: Following continuous query research [57], [58], [59], [60], our evaluation employs a synthetic continuous query dataset generated by a specialized generator, allowing precise parameter control while encompassing realistic operational scenarios. The generator systematically varies the experimental parameters to explore different settings, enabling straightforward adaptation of our methodology to specific contexts through parameter modifications. To emulate different scales of data stream management systems, we set the default configuration to 1000 continuous queries ($N = 1000$) and 10 sharing plans ($M = 10$), with these parameters adjusted in subsequent analyses (Sections VI-B1 and VI-B2). CQ slides s follow a uniform distribution within $[1, S_{max}]$, rounded to the nearest power of two, with $S_{max} = 1024$ as the default maximum, striking a balance between granularity and computational feasibility. CQ ranges r are determined by their respective slides, where $r_i = \gamma_i \cdot s_i$, with overlap factor γ_i uniformly distributed in $[1, \Gamma_{max}]$. The parameter Γ_{max} is set to 10000 by default.

2) Payment and Cost Model: The customer payment model reflects data utility through two temporal dimensions: freshness and coverage. Prior studies [61] have established the slide interval s as a measure of the age of information between consecutive updates. Building upon this concept, we characterize data freshness as $1/s$ to capture the inverse relationship between update frequency and information aging. The range parameter r quantifies temporal coverage, with larger ranges providing enhanced statistical robustness. Thus, for a customer with utility preference $\phi \sim U[0, \Phi_{\max}]$, we define the payment as $\pi(\phi, r, s) = \phi r/s$. Drawing from Amazon's pricing model [44], we normalize processing and memory cost coefficients to $\theta_p = 1$ and $\theta_m = 0.36$, respectively.

3) Baselines: Given the current absence of continuous query-based data trading models in the market, our study evaluates our proposed mechanism against the repeated one-time query-based method (RO) [2], [3], which is widely utilized in current data markets but does not incorporate computation sharing. This comparison demonstrates the advantages of continuous query-based data trading methods, particularly through three computation-sharing techniques: the linear technique (CQT-L) for basic sequential processing, the tree-based technique (CQT-T) for hierarchical optimization, and the queue-based technique (CQT-Q) for efficient parallel execution.

B. Results

This section presents and analyzes the experimental results of our proposed mechanism's performance across diverse parameter configurations. Through systematic parameter variation and subsequent analysis of their individual effects, we assess the mechanism's sensitivity to parametric changes and its operational robustness. Our experimental observations revealed several notable phenomena, which are emphasized in *italic* font throughout this section.

1) Number of Continuous Queries: We assessed the scalability of our mechanism by varying the number of continuous queries (CQs) from 50 to 3000, with results shown in Fig. 10(a) and (b). The success rate—defined as the ratio of accepted to total CQs—demonstrates that our methods (CQT-L, CQT-T, and CQT-Q) significantly outperform the RO baseline. *The observed trend—initial decline, followed by recovery, and then a drop—reflects the mechanism's adaptive behavior under increasing workloads.* Initially, as more individual CQs are added, system capacity becomes strained, causing a decline in success rates. The system then recovers as shared plans are activated, leveraging computation reuse to support additional queries. Eventually, success rates fall again as system capacity reaches its limits, indicating a scalability ceiling. Profit, in contrast, shows a clear upward trend with more CQs, illustrating the mechanism's ability to capitalize on increased demand while maintaining profitability.

2) Number of Query Plans: To further evaluate scalability, we varied the number of available query plans from 5 to 100, as depicted in Fig. 10(c) and (d). The CQT-L method consistently achieves higher profit than RO, with a peak improvement of 28.7%. The profit increases sharply between 10 and 30 plans as

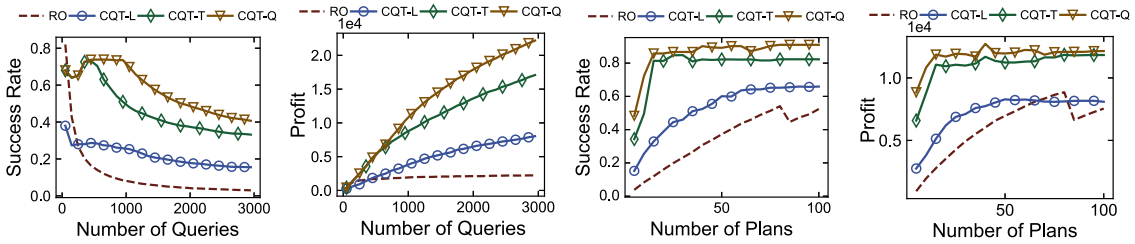


Fig. 10. Success rate and profit with an increasing number of CQs and query plans.

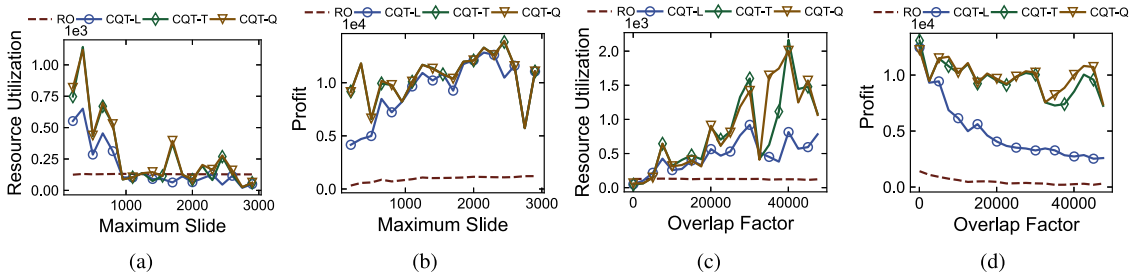


Fig. 11. Resource utilization and profit with increasing max slides and ranges.

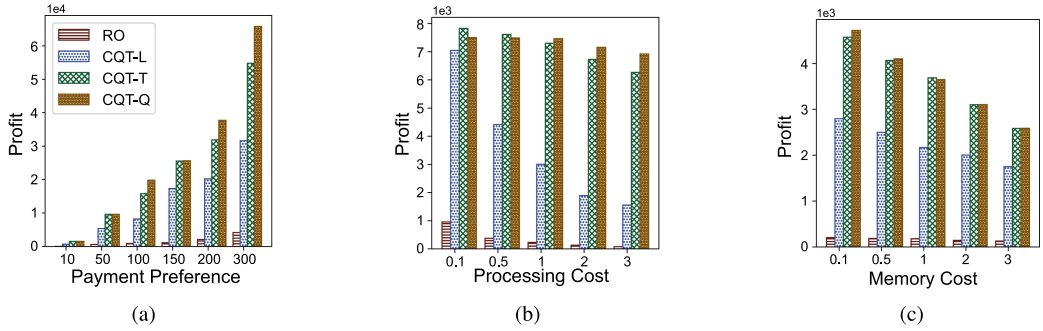


Fig. 12. Profit with different payment preferences and cost coefficients.

the demand for CQ execution intensifies. *As the number of plans continues to grow, CQT-Q and CQT-L show more fluctuation*, benefiting from increased residual capacity via computation sharing. Meanwhile, CQT-L and RO demonstrate more stable profit growth, roughly proportional to the expanded planning options.

3) Maximum Slide: We examined the effects of varying the maximum slide parameter (S_{max}) across a range of 300 to 3000, as shown in Fig. 11(a) and (b). Fig. 11(a) presents resource utilization, defined as the ratio of consumed to available resources. *In continuous query trading, utilization shows periodic changes with increasing slide values, whereas RO remains relatively flat.* This stability in RO arises from its reliance on a fixed overlap factor—i.e., the ratio of window range to slide—used in its cost estimation. The observed variations in our methods stem from dynamic plan selection responding to sliding window characteristics, as further confirmed by profit trends in Fig. 11(b).

4) Overlap Factor for the Range: We analyzed the effect of varying the maximum overlap factor (Γ_{max}) by increasing the window range from 500 to 50000, as shown in Fig. 11(c) and (d). The results reveal periodic fluctuations in resource utilization similar to those seen in the sliding window experiment. *Importantly, Fig. 11(d) shows a consistent decline in profit across all methods as* Γ_{max} *increases.* This is due to the direct relationship between range size and processing overhead, which limits the profit margins in high-overlap scenarios.

5) Payment Preference and Cost: We varied parameters including payment preference ($\Phi_{\max}$), processing cost (θ_p), and memory cost (θ_m) to simulate different pricing models based on AWS [44] and Google Cloud [45]. Results in Fig. 12(a)–(c) show that CQ-based methods consistently outperform RO. *As the payment preference increases, the profit gap among different CQ trading strategies widens*, showcasing the efficiency of computation sharing in handling higher query volumes. Moreover, performance differences among CQ methods depend on cost parameters. For example, under lower processing costs ($\theta_p = 0.1$), CQT-T achieves higher profits than CQT-Q, thanks to its more efficient memory use Fig. 12(c). These findings

underscore the importance of cost-aware method selection to maximize profitability in varying deployment scenarios.

VII. Conclusion

We presented CQTrade, the first mechanism for continuous query-based data trading, aiming to enhance data trading for online data analytics. Our contributions are threefold: (1) we conducted a theoretical analysis of multiple computation-sharing techniques in continuous query execution, an area largely unexplored in existing literature yet crucial to data trading; (2) building on this analysis, we formulated a general optimization problem that adapts to various continuous query execution techniques, with the objective of maximizing the seller's profit; (3) we developed a branch-and-price algorithm to address the inherent complexity of the optimization problem, which incorporates elements of vector bin packing and multidimensional multiple knapsack problems. Our empirical evaluation demonstrated that CQTrade improved data trading success rates by 12.8% and increased seller profit by 28.7% on average, compared to conventional one-time query-based trading methods.

While our current mechanism focuses on the sell-side market to optimize seller profits under computation sharing, a key future direction is to extend CQTrade to bilateral settings that incorporate buyer-side decision-making. This includes utility-driven objectives, joint welfare maximization, or equilibrium-based formulations that account for both buyer utility and seller cost. Incorporating buyer bidding, auction-based pricing, and QoS differentiation would support more dynamic and realistic trading, enabling incentive-compatible mechanisms under resource constraints.

Furthermore, while CQTrade supports centralized execution and colocated queries in distributed settings, extending it to finer-grained deployments may require additional coordination to maintain computation sharing. Exploring such extensions is a promising direction for future work. Another important future direction is to validate CQTrade in real-world deployment scenarios, including integration with live data streams and end-to-end system evaluation under practical constraints.

References

[1] J. Cheng, N. Ding, J. C. Lui, and J. Huang, "Continuous query-based data trading," in *Proc. ACM SIGMETRICS conf.*, 2024, pp. 73–74.

[2] Z. Cong, X. Luo, J. Pei, F. Zhu, and Y. Zhang, "Data pricing in machine learning pipelines," *Knowl. Inf. Syst.*, vol. 64, no. 6, pp. 1417–1455, 2022.

[3] M. Zhang, F. Beltrán, and J. Liu, "A survey of data pricing for data marketplaces," *IEEE Trans. Big Data*, vol. 9, no. 4, pp. 1038–1056, Aug. 2023.

[4] O. T. Modupe et al., "Reviewing the transformational impact of edge computing on real-time data processing and analytics," *Comput. Sci. IT Res. J.*, vol. 5, no. 3, pp. 603–702, 2024.

[5] AWS Data Exchange, 2023. [Online]. Available: https://aws.amazon.com/cn/data-exchange

[6] Xignite, 2023. [Online]. Available: https://www.xignite.com

[7] Grand View Research, "Data marketplace market report," 2023. [Online]. Available: https://www.grandviewresearch.com

[8] M. Orabi, Z. Al Aghbari, I. Kamel, and D. Mouheb, "Keeping an eye on moving objects: Processing continuous spatial-keyword range queries," *GeoInformatica*, vol. 28, no. 1, pp. 117–143, 2024.

[9] R. Kesavan, D. Gay, D. Thevessen, J. Shah, and C. Mohan, "Firestore: The NoSQL serverless database for the application developer," in *Proc. 2023 Proc. Int. Conf. Data Eng.*, 2023, pp. 3376–3388.

[10] X. Dai, Z. Wang, J. Xie, X. Liu, and J. C. Lui, "Conversational recommendation with online learning and clustering on misspecified users," *IEEE Trans. Knowl. Data Eng.*, vol. 36, no. 12, pp. 7825–7838, Dec. 2024.

[11] R. Zhou, X. Zhang, J. C. Lui, and Z. Li, "Dynamic pricing and placing for distributed machine learning jobs: An online learning approach," *IEEE J. Sel. Areas Commun.*, vol. 41, no. 4, pp. 1135–1150, Apr. 2023.

[12] J. Ye, D. Lin, K. Cai, C. Zhou, J. He, and J. C. Lui, "Data-driven rate control for RDMA networks: A lightweight online learning approach," in *Proc. IEEE Int. Conf. Distrib. Comput. Syst.*, 2023, pp. 1–11.

[13] X. Wang, J. Ye, and J. C. Lui, "Online learning aided decentralized multi-user task offloading for mobile edge computing," *IEEE Trans. Mobile Comput.*, vol. 23, no. 4, pp. 3328–3342, Apr. 2024.

[14] J. Verwiebe, P. M. Grulich, J. Traub, and V. Markl, "Survey of window types for aggregation in stream processing systems," *VLDB J.*, vol. 32, pp. 985–1011, 2023.

[15] M. Xiao, M. Li, and J. J. Zhang, "Locally differentially private personal data markets using contextual dynamic pricing mechanism," *IEEE Trans. Dependable Secure Comput.*, vol. 20, no. 6, pp. 5043–5055, Nov./Dec. 2023.

[16] Z. Cai, X. Zheng, J. Wang, and Z. He, "Private data trading towards range counting queries in Internet of Things," *IEEE Trans. Mobile Comput.*, vol. 22, no. 8, pp. 4881–4897, Aug. 2023.

[17] Z. He and Z. Cai, "Trading aggregate statistics over private Internet of Things data," *IEEE Trans. Comput.*, vol. 73, no. 2, pp. 394–407, Feb. 2024.

[18] C. Chen, Y. Yuan, Z. Wen, Y.-P. Wang, and G. Wang, "GShop: Towards flexible pricing for graph statistics," in *Proc. Int. Conf. Data Eng.*, 2024, pp. 2612–2624.

[19] S. R. Pandey, P. Pinson, and P. Popovski, "Strategic coalition for data pricing in IoT data markets," *IEEE Internet Things J.*, vol. 11, no. 4, pp. 6454–6468, Feb. 2024.

[20] J. Li, J. Li, X. Wang, R. Qin, Y. Yuan, and F.-Y. Wang, "Multi-blockchain based data trading markets with novel pricing mechanisms," *IEEE/CAA J. Automatica Sinica*, vol. 10, no. 12, pp. 2222–2232, 2023.

[21] L. Xue, J. Ni, D. Liu, X. Lin, and X. Shen, "Blockchain-based fair and fine-grained data trading with privacy preservation," *IEEE Trans. Comput.*, vol. 72, no. 9, pp. 2440–2453, Sep. 2023.

[22] H. Xu, S. Qi, Y. Qi, W. Wei, and N. Xiong, "Secure and lightweight blockchain-based truthful data trading for real-time vehicular crowdsensing," *ACM Trans. Embedded Comput. Syst.*, vol. 23, no. 1, pp. 1–31, 2024.

[23] T. Singh, R. Kalra, S. Mishra, Satakshi, and M. Kumar, "An efficient real-time stock prediction exploiting incremental learning and deep learning," *Evolving Syst.*, vol. 14, no. 6, pp. 919–937, 2023.

[24] L. Yang and A. Shami, "A lightweight concept drift detection and adaptation framework for IoT data streams," *IEEE Internet Things Mag.*, vol. 4, no. 2, pp. 96–101, Jun. 2021.

[25] W. Yue, L. Benson, and T. Rabl, "Desis: Efficient window aggregation in decentralized networks," in *Proc. Joint Eur. Conf. Mach. Learn. Knowl. Discov. Databases*, 2023, pp. 1501–1514.

[26] M. Fragkoulis, P. Carbone, V. Kalavri, and A. Katsifodimos, "A survey on the evolution of stream processing systems," *VLDB J.*, vol. 33, no. 2, pp. 507–541, 2024.

[27] L. Golab and M. T. Ozsu, *Data Stream Management*. Berlin, Germany: Springer, 2022.

[28] M. Budiu, F. McSherry, L. Ryzhyk, and V. Tannen, "DBSP: Automatic incremental view maintenance for rich query languages," 2022, *arXiv:2203.16684*.

[29] A. Jayarajan, W. Zhao, Y. Sun, and G. Pekhimenko, "TiLT: A time-centric approach for stream query optimization and parallelization," in 2023*Proc. Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2023, pp. 818–832.

[30] K. Peddireddy, "Kafka-based architecture in building data lakes for real-time data streams," *Int. J. Comput. Appl.*, vol. 185, no. 9, pp. 1–3, 2023.

[31] O.-C. Marcu and P. Bouvry, "Big data stream processing," Ph.D. dissertation, Univ. Luxembourg, Esch-Belval Esch-sur-Alzette, Luxembourg, 2024.

[32] A. Bonifati and R. Tommasini, "An overview of continuous querying in (modern) data systems," in *Proc. Companion 2024 Int. Conf. Manage. Data*, 2024, pp. 605–612.

[33] R. Mancini, S. Karthik, B. Chandra, V. Mageirakos, and A. Ailamaki, "Efficient massively parallel join optimization for large queries," in *Proc. 2022 Int. Conf. Manage. Data*, 2022, pp. 122–135.

[34] A. Mhedhbi, C. Kankanamge, and S. Salihoglu, "Optimizing one-time and continuous subgraph queries using worst-case optimal joins," *ACM Trans. Database Syst.*, vol. 46, no. 2, pp. 1–45, 2021.

[35] B. Gurumurthy, V. R. Bidarkar, D. Broneske, T. Pionteck, and G. Saake, "Exploiting shared sub-expression and materialized view reuse for multi-query optimization," *Inf. Syst. Front.*, pp. 1–16, 2024.
[36] S. Zinchenko and D. Ponomaryov, "The selection problem in multi-query optimization: A comprehensive survey," 2024, *arXiv:2412.11828*.
[37] P. Sioulas and A. Ailamaki, "Scalable multi-query execution using reinforcement learning," in *Proc. 2021 Int. Conf. Manage. Data*, 2021, pp. 1651–1663.
[38] G. Theodorakis, A. Koliousis, P. Pietzuch, and H. Pirk, "Lightsaber: Efficient window aggregation on multi-core processors," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2020, pp. 2505–2521.
[39] J. Hülsmann, J. Traub, and V. Markl, "Demand-based sensor data gathering with multi-query optimization," in *Proc. VLDB Endowment*, vol. 13, no. 12, pp. 2801–2804, 2020.
[40] S. Guirguis, M. A. Sharaf, P. K. Chrysanthis, and A. Labrinidis, "Three-level processing of multiple aggregate continuous queries," in *Proc. Int. Conf. Data Eng.*, 2012, pp. 929–940.
[41] C. Chung, S. Guirguis, and A. Kurdia, "Competitive cost-savings in data stream management systems," in *2014 COCOON*. Berlin, Germany: Springer, 2014.
[42] J. Karimov, T. Rabl, and V. Markl, "Astream: Ad-hoc shared stream processing," in *Proc. Int. Conf. Manage. Data*, 2019, pp. 607–622.
[43] L. Doshi et al., "Kepler: Robust learning for parametric query optimization," in *Proc. ACM Manage. Data*, vol. 1, no. 1, pp. 1–25, 2023.
[44] Amazon, "Amazon web services," 2024. [Online]. Available: https://aws.amazon.com/
[45] Google, "Google cloud," 2024. [Online]. Available: https://cloud.google.com/
[46] S. Guirguis, M. A. Sharaf, P. K. Chrysanthis, and A. Labrinidis, "Optimized processing of multiple aggregate continuous queries," in *Proc. Conf. Inf. Knowl. Manage.*, 2021, pp. 1515–1524.
[47] A. U. Shein and P. K. Chrysanthis, "Multi-query optimization of incrementally evaluated sliding-window aggregations," *IEEE Trans. Knowl. Data Eng.*, vol. 34, no. 8, pp. 3899–3911, Aug. 2020.
[48] A. U. Shein, P. K. Chrysanthis, and A. Labrinidis, "FlatFIT: Accelerated incremental sliding-window aggregation for real-time analytics," in *Proc. 29th Int. Conf. Sci. Statist. Database Manage.*, 2017, Art. no. 5.
[49] Q. Ren, M. H. Dunham, and V. Kumar, "Semantic caching and query processing," *IEEE Trans. Knowl. data Eng.*, vol. 15, no. 1, pp. 192–210, Jan./Feb. 2003.
[50] R. Mohandoss, "Context-based semantic caching for LLM applications," in *Proc. 2024 IEEE Conf. Artif. Intell.*, 2024, pp. 371–376.
[51] S. Dasgupta et al., "waLLMartcaChe: A distributed, multi-tenant and enhanced semantic caching system for LLMs," in *Proc. Int. Conf. Pattern Recognit.*, 2024, pp. 232–248.
[52] G. B. Dantzig and P. Wolfe, "Decomposition principle for linear programs," *Operations Res.*, vol. 8, no. 1, pp. 101–111, 1960.
[53] A. Schrijver, *Theory of Linear and Integer Programming*. Hoboken, NJ, USA: Wiley, 1998.
[54] D. M. Ryan and B. A. Foster, "An integer programming approach to scheduling," *Comput. Scheduling Public Transport Urban Passenger Veh. Crew Scheduling*, pp. 269–280, 1981.
[55] G. Gamrath, T. Koch, S. J. Maher, D. Rehfeldt, and Y. Shinano, "SCIP-Jack—a solver for STP and variants with parallelization extensions," *Math. Program. Computation*, vol. 9, no. 2, pp. 231–296, 2017.
[56] J. W. Mamer and R. D. McBride, "A decomposition-based pricing procedure for large-scale linear programs: An application to the linear multicommodity flow problem," *Manage. Sci.*, vol. 46, no. 5, pp. 693–709, 2000.
[57] S. Mell, F. Bastani, S. Zdancewic, and O. Bastani, "Synthesizing trajectory queries from examples," in *Computer Aided Verification*. Berlin, Germany: Springer, 2023.
[58] A. Chaudhary, S. Zeuch, V. Markl, and J. Karimov, "Incremental stream query merging," in *Proc. Joint Eur. Conf. Mach. Learn. Knowl. Discov. Databases*, 2023, pp. 604–617.
[59] R. Zhu, Y. Jia, X. Yang, B. Zheng, B. Wang, and C. Zong, "Multiple continuous top-K queries over data stream," in *Proc. Int. Conf. Data Eng.*, 2024, pp. 1575–1588.
[60] J. Feng, Z. Li, and Q. Chen, "Towards exploratory query optimization for template-based SQL workloads," in *Proc. Int. Conf. Data Eng.*, 2024, pp. 151–164.
[61] Z. Liu, "Data freshness in information-update systems: Modeling, scheduling, and tradeoffs," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 51, no. 3, pp. 46–49, 2024.

Jin Cheng received the BS degree from Xiamen University, and the MS degree from the University of Chinese Academy of Sciences. She is currently working toward the PhD degree in the joint doctoral program between The Chinese University of Hong Kong, Shenzhen, and The Chinese University of Hong Kong. Her research interests include data trading, network economics, online optimization, and database systems.

Ningning Ding received the BS degree in information science and engineering from Southeast University, in 2018, and the PhD degree in information engineering from the Chinese University of Hong Kong, in 2022. She is a Tenure-Track assistant professor with the Data Science and Analytics Thrust and the Internet of Things Thrust within the Information Hub at the Hong Kong University of Science and Technology (Guangzhou). Before that, she was a postdoctoral scholar with the Department of Electrical and Computer Engineering, Northwestern University, USA. Her research interests include focuses on interdisciplinary areas of artificial intelligence, network systems, and network economics, with a current emphasis on trustworthy learning and efficient coordination in distributed environments.

John C.S. Lui (Fellow, IEEE) received the PhD degree in computer science from UCLA. He is currently the Choh-Ming Li chair Professor with the Department of Computer Science & Engineering (CSE), Chinese University of Hong Kong (CUHK). After his graduation, he joined the IBM Laboratory. He later joined the CSE Department, CUHK. His current research interests include quantum networks, online learning algorithms and applications (e.g., multi-armed bandits, reinforcement learning), machine learning on network sciences and networking systems, large-scale data analytics, network economics, large-scale storage systems, and performance evaluation theory. He has served with the IEEE fellow Review Committees. He is an elected member of the IFIP WG 7.3, fellow of ACM, senior research fellow of the Croucher Foundation, fellow of the Hong Kong Academy of Engineering Sciences (HKAES).

Jianwei Huang (Fellow, IEEE) is a presidential chair professor and associate vice president with the Chinese University of Hong Kong, Shenzhen. He is also the associate director with the Shenzhen Institute of Artificial Intelligence and Robotics for Society. He has served as the editor-in-chief of *IEEE Transactions on Network Science and Engineering* and the associate editor-in-chief of the *IEEE Open Journal of the Communications Society*. He is an IEEE ComSoc Distinguished Lecturer, a Clarivate Web of Science Highly Cited Researcher, and an Elsevier Most Cited Chinese Researcher.