Twisted Twin: A Collaborative and Competitive Memory Management Approach in HTAP Systems

Jiani Yang
Zhejiang University & Zhejiang Key Laboratory of Big Data Intelligent Computing
jianiyang_cs@zju.edu.cn

Sai Wu
Zhejiang University & Zhejiang Key Laboratory of Big Data Intelligent Computing
wusai@zju.edu.cn

Yong Wang
Huawei
wangyong308@huawei.com

Dongxiang Zhang
Zhejiang University
zhangdongxiang@zju.edu.cn

Yifei Liu
Zhejiang University
liuyifei0@zju.edu.cn

Xiu Tang
Zhejiang University
tangxiu@zju.edu.cn

Gang Chen
Zhejiang University
cg@zju.edu.cn

ABSTRACT

Many GaussDB customers, particularly small and medium-sized enterprises (SMEs), require high transaction throughput with occasional analytical queries. HTAP systems that deploy both OLTP and OLAP engines on a single server to manage hybrid workloads have become increasingly popular among customers for achieving high cost-efficiency and data freshness. However, co-locating these systems can lead to resource contention, particularly for memory, potentially degrading overall system performance and causing Service-Level Agreements (SLA) violations. To address this issue, we propose T^2 (**T**wisted **T**win), an adaptive memory management approach that dynamically allocates memory between OLTP and OLAP components. This approach ensures OLTP meets SLA while optimizing the efficiency of OLAP query processing. However, this is non-trivial, as memory allocation triggers a cascade of effects, including in-memory column selection and data synchronization, both critical in HTAP systems. To overcome these challenges, we introduce a Bayesian optimization framework tailored for fluctuating workloads that adjusts memory allocation responsively. Experiments conducted on the real-world HTAP system, GaussDB-HTAP, demonstrate the effectiveness and efficiency of T^2.

PVLDB Reference Format:
Jiani Yang, Sai Wu, Yong Wang, Dongxiang Zhang, Yifei Liu, Xiu Tang, and Gang Chen. Twisted Twin: A Collaborative and Competitive Memory Management Approach in HTAP Systems. PVLDB, 18(10): 3312 - 3325, 2025.
doi:10.14778/3748191.3748197

1 INTRODUCTION

Popular HTAP (Hybrid Transactional/Analytical Processing) systems [9, 21, 29, 33, 48] employ two specialized engines with dedicated data stores to handle transactional and analytical workloads, with periodic data synchronization between the two. However, the underlying storage architectures and synchronization mechanisms vary across these systems. After several years of marketing HTAP systems, we have observed that many GaussDB [22] customers, particularly small and medium-sized enterprises (SMEs), primarily require high transaction throughput with occasional analytical queries. Simultaneously, these customers want to avoid the risks associated with outdated data impacting decision-making.

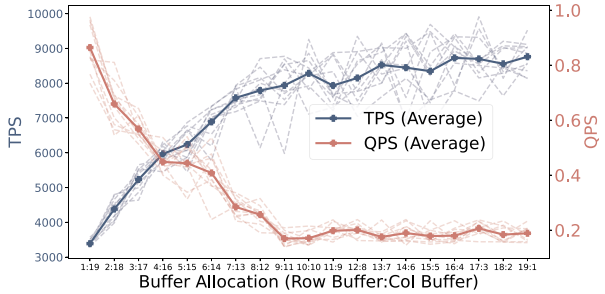

Figure 1: The Impact of Buffer Allocation Between Row Store and Column Store (HyBench)

To meet these needs, GaussDB has co-located OLTP and OLAP systems within the same machine, similar to other widely used HTAP commercial systems (eg. Oracle [27], SQL Server [29], PolarDB-IMCI [48]). It employs a row-native architecture that persists row-oriented data while utilizing in-memory column stores as secondary storage. Updates are written to an in-memory delta table, ensuring complete data freshness. However, this architecture suffers from poor performance isolation [30, 39, 41], raising concerns among customers about a severe drop in OLTP throughput, which may critically disrupt business operations and service responsiveness. Therefore, we aim to identify an approach that ❶ ensures OLTP performance meets the Service-Level Agreement (SLA), and ❷ optimizes OLAP queries as much as possible.

Memory allocation remains a critical factor influencing the performance of HTAP systems. As shown in Figure 1, the x-axis denotes the memory partition ratio between OLTP and OLAP components. TPS (transactions per second) measures transactional throughput, while QPS (queries per second) reflects analytical query performance. Each configuration was executed repeatedly to confirm the robustness of the observed trends. The solid lines in the figure represent the averaged results, and the dashed lines indicate individual runs. The results show that increasing memory allocation enhances

Proceedings of the VLDB Endowment, Vol. 18, No. 10 ISSN 2150-8097.
doi:10.14778/3748191.3748197

the performance of both components, albeit in a nonlinear manner. For OLTP workloads, additional memory helps retain hot records in memory, thereby reducing I/O overhead. For OLAP workloads, allocating more memory to the column store increases the likelihood of columnar data being cached, reducing the need to fall back to the row store during analytical query execution.

This trade-off highlights the memory competition between the OLTP and OLAP components in the HTAP system, which behave like **twisted twins**, each competing for memory to optimize its performance. However, in real-world applications the peak periods of OLTP and OLAP typically do not coincide. This creates a collaborative opportunity to improve memory utilization by adjusting memory allocation dynamically to achieve goals ❶ and ❷.

There exists extensive research on memory allocation in traditional database systems [35, 42, 44, 52]. However, no established methods specifically address memory allocation for the two subsystems of HTAP. In fact, it is non-trivial task:

1) For the OLTP system, it is difficult to manually determine how much memory should be allocated to meet SLA requirements. For the OLAP system, both the amount of memory allocated and the selection of which columns to load into memory under this memory constraint directly impact OLAP performance.

2) The interference between OLTP and OLAP makes the situation even more complex. Insufficient memory allocation for OLAP may prevent analytical queries from accessing data in the in-memory column store, resulting in increased disk I/O, which in turn affects OLTP performance. Besides, data updates between OLTP and OLAP components must be periodically synchronized, which also affects both systems. Frequent synchronizations can reduce transactional query throughput, while infrequent ones force analytical queries to read from delta tables, thereby degrading OLAP's performance.

3) Given that OLTP and OLAP workloads typically exhibit staggered peak resource demands, it is essential to reallocate memory when workload shifts, which expands the problem's search space. Moreover, loading new columns from disk into column store incurs additional costs that must be considered during reallocation, further complicating the problem in dynamic scenarios.

In this paper, we introduce T^2 (**T**wisted **T**win), an adaptive memory management approach for HTAP systems to address the above challenges. Given the complexity of the system and the numerous influencing factors, we design a comprehensive optimization framework. Specifically, we identify three critical modules in HTAP systems that require optimization: column selection, data synchronization, and memory allocation. To reduce the problem's search space, we propose a two-layer solution. First, we optimize memory management for static workloads. Then, for dynamic workloads, we decompose them into discrete static workloads and apply the static approach at appropriate times. In summary, this paper makes the following key contributions:

- We adopt a Bayesian optimization framework to efficiently allocate memory while optimizing column selection and data synchronization strategies within the loop. This framework introduces a lightweight yet effective algorithm to provide a holistic solution for static workloads.
- We developed an adaptive memory management strategy for dynamic hybrid workloads, using a time-series prediction model to forecast query patterns and a heuristic algorithm to optimize reorganization timing, improving memory utilization.
- We built T^2 on GaussDB and evaluated its performance using HTAP benchmarks. Our results demonstrate that T^2 outperforms existing solutions by dynamically adjusting key components.

The paper is organized as follows: Section 2 presents the design of T^2 and defines the memory management problem. Section 3 discusses static workload handling, while Section 4 covers optimization for dynamic workloads. Section 5 evaluates our system with HTAP benchmarks, Section 6 reviews related work, and Section 7 concludes the paper. To improve readability, we provide a list of symbols in the technical report[1].

2 OVERVIEW

In this section, we first provide an overview of the GaussDB-HTAP system, clearly defining our problem and overall objective. We then present the system design of T^2 to achieve the defined objective.

2.1 Design of GaussDB-HTAP

As mentioned, our target users mainly require OLTP processing with occasional lightweight OLAP queries. To address this, we use GaussDB as a case study and integrate OLAP components into the original GaussDB to enhance its HTAP capabilities. The resulting system is referred to as GaussDB-HTAP.

The architecture of GaussDB-HTAP, shown in Figure 2, features a router that directs queries to either the OLAP or OLTP modules based on their processing requirements and characteristics. Specifically, transactional queries are routed to the Row Execution Engine, while analytical queries are sent to the Vector Execution Engine. To support these modules, the buffer pool is divided into two components: a row store buffer that caches all data, and a column store buffer that stores a user-specified subset of columns in columnar format for efficient OLAP processing. For transactional queries, the row engine retrieves data from either the row store buffer or disk as needed. For analytical queries, vector engine initially attempts to fetch data from the column store buffer. If the required data is not available in the column store, the query falls back to retrieving data from the row store buffer or disk. The data in the column store buffer is not persisted to disk and is rebuilt after each GaussDB-HTAP restart. Like many other HTAP systems (e.g., [27, 29, 40]), we use delta tables to record incremental changes in the row store, such as inserts, updates, and deletions, which are periodically synchronized with the column store (see 3.2 for details).

Although T^2 was developed based on GaussDB's system architecture and data synchronization techniques, the core ideas and methodologies we propose are equally applicable to other systems [27, 29, 48] that utilize row store as primary storage, supplemented by an in-memory column store to optimize OLAP performance. In our technical report, we provide a more detailed discussion on how T^2 can be applied to other HTAP systems.

2.2 Problem Definition

Our approach prioritizes meeting the service level agreement (SLA) of the OLTP workload while enhancing the efficiency of the OLAP

[1]Technical Report:https://yplusone.github.io/files/TwistedTwin_tech_report.pdf

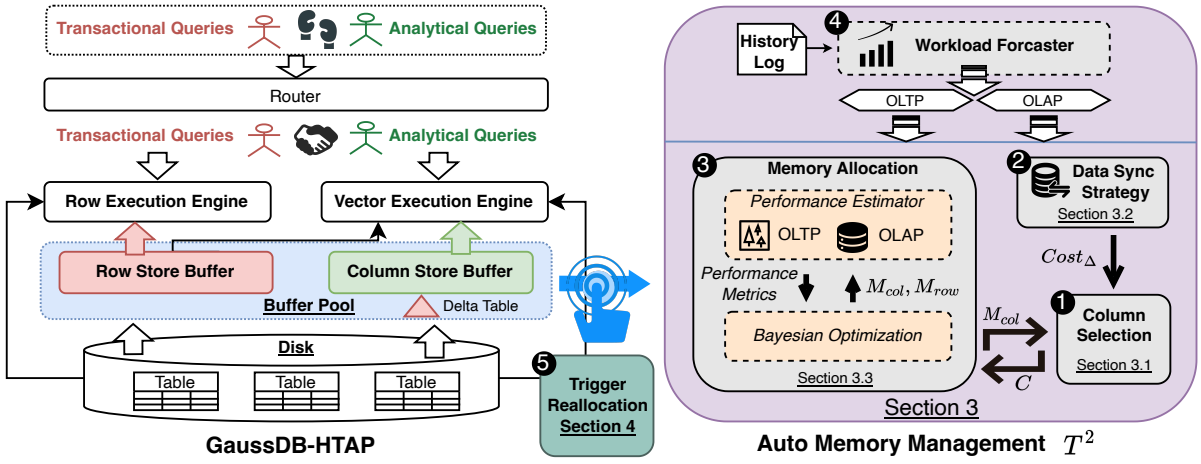

Figure 2: Architecture of GaussDB-HTAP System and Auto Memory Management Algorithm T^2

workload. In this paper, we focus on dynamically adjusting memory allocation between the two buffers to achieve our goal. Let $\mathcal{M}_{\text{total}}$ represent the total memory available for the whole buffer pool. The memory allocation must satisfy the following constraint:

$$\mathcal{M}_{\text{row}} + \mathcal{M}_{\text{col}} \leq \mathcal{M}_{\text{total}},$$

where $\mathcal{M}_{\text{row}}$ and $\mathcal{M}_{\text{col}}$ denote the memory allocated to the row and column store buffers, respectively. Let $W(C)$ denote the size of selected column set C in the column store. Naturally, we have:

$$W(C) \leq \mathcal{M}_{\text{col}}.$$

A strong dependency is observed between the transactional throughput of the system and $\mathcal{M}_{\text{row}}$, which we model through the function $\mathcal{P}_{\text{OLTP}}(\mathcal{M}_{\text{row}})$. A comprehensive discussion of other contributing parameters is provided in Section 3.3.1.

2.2.1 Static Workload. In the static scenario, the SLA threshold of the OLTP workload is fixed as θ. The objective is to guarantee service responsiveness by ensuring that the system's TPS, represented by $\mathcal{P}_{\text{OLTP}}(\mathcal{M}_{\text{row}})$, remains greater than or equal to θ. With this constraint, the focus shifts to optimizing OLAP workload $\mathcal{W}_{\text{ap}}$ by efficiently utilizing the remaining memory resources. The cost of executing the analytical workload is denoted by $\text{Cost}_{\mathcal{W}_{\text{ap}}}(C)$, and the cost of reading and synchronizing delta tables is given by $\text{Cost}_\Delta(C, \mathcal{K})$. Here, $\mathcal{K}$ represents the data synchronization strategy.

The static scenario yields the following optimization problem:

$$\begin{aligned} \text{Minimize} \quad & \text{Cost}_{\mathcal{W}_{\text{ap}}}(C) + \text{Cost}_\Delta(C, \mathcal{K}) \\ \text{Subject to} \quad & \mathcal{P}_{OLTP}(\mathcal{M}_{\text{row}}) \geq \theta, \\ & \mathcal{M}_{\text{row}} + \mathcal{M}_{\text{col}} \leq \mathcal{M}_{\text{total}}, \\ & W(C) \leq \mathcal{M}_{\text{col}}. \end{aligned} \tag{1}$$

The solution is to develop an algorithm that determines the optimal memory allocations $\mathcal{M}_{\text{row}}$ and $\mathcal{M}_{\text{col}}$, selects the appropriate set of columns C, and chooses the data synchronization strategy $\mathcal{K}$ to minimize the total cost while satisfying the constraints on OLTP.

2.2.2 Dynamic Workload. In real-world scenarios, TPS requirements and OLAP workloads often fluctuate over time (e.g., by hour or day), making static memory allocation insufficient. When OLTP load is low, more memory can be allocated to OLAP; when OLTP demand is high, the system should prioritize OLTP throughput.

To capture such dynamics, we divide time into equal-sized windows $t_0, t_1, ..., t_n$ and adapt memory allocation in each window accordingly. However, triggering dynamic memory reallocation incurs a switching cost, as new columns must be loaded from disk into the column store buffer. We denote this switching cost as $\text{Cost}^{t_i}_{\text{switch}}$. Frequent modifications to the column store content can lead to substantial overhead. Therefore, the main objective is to minimize the overall cost, as described below:

$$\text{Min.} \quad \sum_{t_i} \text{Cost}_{\mathcal{W}'_{\text{ap}}(t_i)}(C_{t_i}) + \text{Cost}_\Delta(C_{t_i}, \mathcal{K}) + \text{F}(t_i) \cdot \text{Cost}^{t_i}_{\text{switch}} \tag{2}$$

This dynamic scenario introduces two key components beyond the static case: (1) a workload predictor that estimates future OLTP and OLAP demands, denoted as θ'_{t_i} and $\mathcal{W}'_{\text{ap}}(t_i)$, respectively. The system must ensure that the OLTP SLA is satisfied in each interval by meeting the constraint $\mathcal{P}_{OLTP}(\mathcal{M}^{t_i}_{\text{row}}) \geq \theta'_{t_i}$. (2) a reallocation strategy $F(t_i)$ that determines whether memory should be reallocated at the beginning of each time window.

We designed the algorithm for OLTP-prioritized HTAP systems, which is the most common usage scenario. However, the framework is flexible: it can be adapted to OLAP-prioritized settings by modifying constraints (e.g., bounding OLAP latency while optimizing TPS). These extensions are further discussed in the technical report.

2.3 Design of T^2

Solving the optimization problem is challenging due to the nonlinear relationships between variables, constraints, and objectives. To approach an optimal solution efficiently, we design a workflow with the following components: To solve the optimization problem in a static scenario, we design three modules. The first, ❶ **Column Selection**, identifies the optimal column set C within the memory constraint $\mathcal{M}_{\text{col}}$. The second, ❷ **Data Synchronization Strategy**, uses a cost-based model to determine the optimal timing for data synchronization and estimates the additional cost, Cost_Δ. This cost is then integrated into the Column Selection module to account for the impact of data updates. Finally, ❸ **Memory Allocation**

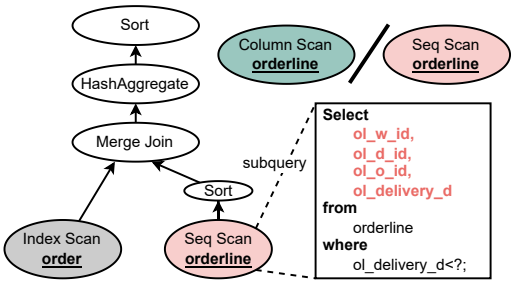

Figure 3: Example Plan Tree for Q12 in CH-benCHmark [11], and Extracting Scanning Queries from Leaf Nodes

allocates memory between $\mathcal{M}_{\text{row}}$ and $\mathcal{M}_{\text{col}}$, based on Bayesian optimization. This module also includes a performance estimator to predict the performance of both OLTP and OLAP workloads, guiding the optimization process and achieving goals of T^2.

To solve the optimization problem in a dynamic scenario, we first need the ❹ **Workload Forecaster** to predict the characteristics of OLTP and OLAP workloads using a time-series prediction network. The predicted characteristics will be used by components ①, ②, and ③. Additionally, they will guide the ❺ **Trigger Reallocation** module in determining the optimal time for memory reallocation. This triggers the algorithm for the static scenario problem, which reallocates memory and updates the selected columns.

3 T^2 FOR STATIC WORKLOADS

In this section, we address the static workload scenario from Section 2.2.1. Under a static workload, memory allocated to the OLAP component is fixed, and our objective is to select the most valuable columns for the column store. This selection process must account for two critical factors: 1) the impact of selected columns on query performance, and 2) the additional costs of updating these columns.

We can model the column selection challenge in HTAP systems as analogous to the knapsack problem, where each column represents an item, the total memory capacity is the knapsack's limit, and the impact of a column on query performance reflects the item's value. However, our scenario introduces complexities beyond the classical knapsack problem, as columns are interdependent. These interactions substantially enlarge the search space and complicate both modeling and solution. Additionally, accounting for the cost of updating columns further complicates the problem, making it even more difficult to model and resolve. We begin by addressing the first challenge.

3.1 Column Selection with No Updates

In GaussDB-HTAP, users explicitly load specific columns into the column store buffer using statements such as "ALTER TABLE orderline COLVIEW(ol_w_id, ol_d_id)". This manual process requires users to decide which columns to load, which can be labor-intensive. Moreover, column choices based on intuition or experience may not lead to optimal performance improvements.

A key challenge in column selection is that the performance benefit depends on the presence of a complete set of columns required by a query. Column selection typically follows an all-or-nothing principle: a query can only benefit from a "Column Scan" when all required columns are present in the column store buffer. As illustrated in Figure 3, a "Seq Scan" (i.e., a full table scan on the row store) can be replaced by a more efficient "Column Scan" only if all four referenced columns are loaded into column store. Otherwise, the query must fall back to accessing data from the row store.

We have developed an integer programming model specifically designed to address the column selection problem in HTAP systems. The primary goal of this model is to enhance the execution efficiency of OLAP queries by selecting the most appropriate columns. Thus, the model aims to minimize the total execution time for all queries within a given OLAP workload. The workload $\mathcal{W}_{\text{ap}}$ consists of a set of queries, represented as $\mathcal{W}_{\text{ap}} = Q_1, Q_2, \ldots, Q_s$. The smallest logical caching unit in the column store buffer is a column, and the full set of M columns is denoted as $Columns = C_1, C_2, \ldots, C_M$.

Since changes in the leaf nodes of a query plan do not impact non-leaf nodes, we can focus exclusively on the "seq scan" node in the execution plan. Therefore, the queries in the workload $\mathcal{W}_{\text{ap}}$ can be decomposed into K subqueries, each corresponding to a leaf node, retrieving data from a single table. We define a quintuple for each subquery q_l as $(q_l, G_l, f_l, \text{cost}_{\text{row}}^{q_l}, \text{cost}_{\text{col}}^{q_l})$, where $0 < l \leq K$. In this context, q_l represents the subquery responsible for data scanning and filtering. For example, a sequential scan on the orderline table can be extracted as a subquery. The variable f_l denotes the execution frequency of subquery q_l. The columns involved in subquery q_l form a group G_l. cost_{row} and cost_{col} represent the costs of retrieving data through row store sequential scan and column scan, respectively. This benefit can either be estimated by the database or measured through actual query execution.

We define x_m as the decision variable indicating whether column C_m is selected. Subquery q_l can perform a column scan if and only if all columns in G_l are selected. Therefore, the objective can be formulated as:

$$\text{Minimize} \sum_{l=1}^{K} \left(\text{cost}_{\text{col}}^{q_l}(\Pi_{m \in G_l} x_m) + \text{cost}_{\text{row}}^{q_l}(1 - \Pi_{m \in G_l} x_m) \right) \cdot f_l$$

Additionally, the total memory size of the selected columns must not exceed the limit $\mathcal{M}_{\text{col}}$:

$$\sum_{m=1}^{M} (w_m x_m) \leq \mathcal{M}_{\text{col}}, \quad \text{where } x_m \in \{0, 1\} \text{ for each } m.$$

Here, w_m represents the memory usage of column C_m. The above objective function is non-linear, which complicates the problem-solving process. In fact, the column selection problem is NP-hard, as we show in the following proof:

Proof. The Maximum Diversity Problem (MDP) involves finding a clique with maximum edge weight in a graph, constrained by a maximum number of nodes, with non-negative edge weights. The MDP is known to be NP-hard [18]. We reduce the MDP to the column selection problem to demonstrate the latter's complexity. Consider a special case of the column selection problem where each query involves exactly two columns. We can represent this scenario as a complete graph $\mathcal{G} = (V, E)$ where:

- Each column corresponds to a node in V, with weight of 1.
- Suppose each subquery only involves two columns. An edge $(i, j) \in E$ exists between two nodes C_i and C_j if they are involved in the same query, and the edge weight $\text{cost}_{\text{row}}^{q} - \text{cost}_{\text{col}}^{q}$ represents the performance gain of storing both columns C_i and C_j in column storage. For columns not involved in the same query, the edge weight is 0.

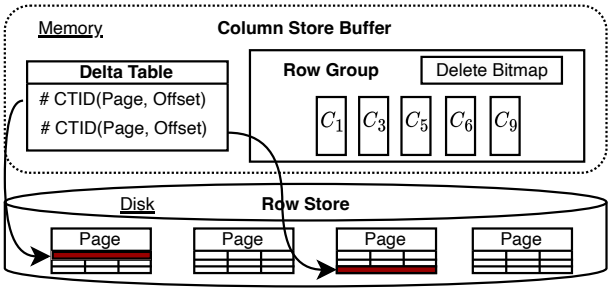

Figure 4: Delta Table Working Mechanism

The column selection problem is to select a subset of columns such that the total performance gain (sum of edge weights) is maximized, subject to the memory constraint $\mathcal{M}_{\text{col}}$. This is equivalent to imposing a constraint on the number of nodes in the selected clique. This special case of the column selection problem is equivalent to the MDP. Since MDP is NP-hard, the column selection problem is also NP-hard by reduction. Hence, the problem is NP-hard. □

Since the column selection problem can be reduced to the MDP, it inherits the same level of complexity, making dynamic programming solutions equally impractical. In this paper, following the approach of Glover and Wolsey [19], we can linearize this nonlinear problem by introducing new decision variables. Specifically, we replace the product of variables in the objective function with a new set of decision variables, z_l, to indicate whether query q_l is scanning from column storage. When $z_l = 1$, all columns in G_l must be present in column storage for the query to benefit. The objective can be simplified as:

$$\text{Minimize} \quad \sum_{l=1}^{K} \left(\text{cost}_{\text{col}}^{q_l} z_l + \text{cost}_{\text{row}}^{q_l}(1 - z_l)\right) \cdot f_l \tag{3}$$

This transformation needs additional linear constraints to ensure that the new variables $z_l = \Pi_{m \in G_l} x_m$:

$$\sum_{m \in G_l} x_m \geq |G_l| \cdot z_l, \quad 0 < l \leq K$$

Specifically, $z_l = 1$ only if all x_m for $m \in G_l$ are equal to 1, thus satisfying the constraint $\sum_{m \in G_l} x_m \geq |G_l| \cdot z_l$. As a result, we have transformed the nonlinear model into a linear one. Although the decision space has expanded, this approach enables more effective optimization while maintaining feasibility.

3.2 Adaptive Data Synchronization Strategy

Next, we address the second challenge: accounting for the additional costs incurred by updates to the selected columns. To tackle this, we introduce our data synchronization system.

Most existing HTAP systems use delta tables to minimize the impact of updates on the in-memory column store. While delta tables help reduce the overhead of updating the column store, they also introduce additional costs associated with reading the data.

Figure 4 illustrates the mechanism of the delta table and column storage. Data is loaded into memory by organizing it into row groups, with columns stored contiguously in an array format within each group, forming a Column Unit (CU). Each row group has an associated delta table that tracks changes to the data, including updated, inserted, or deleted tuples (UID operations). To track these changes, the delta table uses a CTID (Tuple Identifier), which consists of a block number and tuple offset. The CTID uniquely identifies a tuple's physical location in row storage.

During a column scan, the delta table is accessed, and the CTID helps locate the tuple's position, enabling quick data retrieval and merging with the CU's column data. The complete column vector is then passed to the next operator. In this design of delta table (e.g., [27, 29, 40]), the cost of reading the delta table $\text{Cost}_{\text{read}}$ and synchronization $\text{Cost}_{\text{sync}}$, are proportional to the amount of data in the delta table, N , i.e., $\text{Cost}_{\text{read}} \propto N$, $\text{Cost}_{\text{sync}} \propto N$.

One of the key decisions in HTAP systems is determining the optimal timing for data synchronization. In T^2, we ensure snapshot consistency for OLAP queries. Thus, for a query at timestamp t_i, we load data from the delta table up to t_i. Since the cost is proportional to the size of the delta table, it is crucial to prevent excessive data accumulation, which can significantly degrade the performance of in-memory column reads. Regular synchronization and merging of delta table data into columnar storage are essential to maintain performance. However, selecting the right synchronization frequency presents a major challenge. Synchronizing too frequently can negatively impact transactional performance, while infrequent synchronization can degrade the performance of column reads.

To address this challenge, we have established a synchronization threshold. When the data in the delta table exceeds a threshold α, a synchronization process is triggered. The key problem is determining the optimal value for α. In this paper, we conduct cost estimations for the delta table and aim to identify the optimal α.

3.2.1 Cost Model for Delta Table. For both reading and synchronizing the delta table, the operations are proportional to the number of records in the table, with a complexity of $O(n)$. Therefore, we use a linear model to estimate the costs for both:

$$\text{Cost}_{\text{read}}(t) = w_0 N(t) + w_1$$
$$\text{Cost}_{\text{sync}}(t) = w_2 N(t) + w_3$$

Here $N(t)$ represents the number of records in delta table at time t. By measuring the time taken to perform reading and synchronizing operations on delta tables with different row counts, we can obtain some samples to derive the coefficients by fitting these samples with linear regression to get specific values of w_0,w_1,w_2,w_3.

3.2.2 Pick the Synchronization Threshold. We model the problem by selecting a time period $[0, T]$ and assume that data updates are uniformly distributed within this period. The number of records in the delta table grows linearly at a constant rate $\frac{b}{T}$, where b is the total number of UID operations or update records in delta table during $[0, T]$. A synchronization occurs whenever the number of records reaches the threshold α, resulting in $\frac{b}{\alpha}$ synchronization events during $[0, T]$.

The cost of each synchronization is $w_2\alpha + w_3$, leading to a total $\frac{b}{\alpha}$ times synchronization cost:

$$\text{Cost}_{\text{sync}} = \frac{b}{\alpha} \times (w_2\alpha + w_3)$$

We assume that read events are uniformly distributed over the period, with a total of ν delta table reads during $[0, T]$. Since the expected number of records in the delta table is $\frac{\alpha}{2}$ (for further reasoning, refer to the technical report), the expected cost of one

time reading during each synchronization cycle is:

$$E[\text{Cost}_{\text{read}}(t)] = \frac{w_0\alpha}{2} + w_1$$

Thus, the total cost of reading delta table during $[0, T]$ is:

$$\text{Cost}_{\text{read}} = v \times \left(\frac{w_0\alpha}{2} + w_1\right)$$

The overall cost is the sum of synchronization and read costs:

$$\text{Cost}_\Delta = \frac{b}{\alpha}(w_2\alpha + w_3) + v\left(\frac{w_0\alpha}{2} + w_1\right) \tag{4}$$

We derive the optimal α by setting the derivative of Cost_Δ to zero:

$$\frac{d\text{Cost}_\Delta}{d\alpha} = -\frac{bw_3}{\alpha^2} + \frac{vw_0}{2} = 0$$

Solving the equation, we get the synchronization threshold α:

$$\alpha = \sqrt{\frac{2bw_3}{vw_0}}$$

This threshold α minimizes the total cost introduced by delta table, which can be adjusted based on data update and read rates to optimize the synchronization process. The time period $[0, T]$ can be determined in real scenarios, provided that the frequency of UID operations and reading requests remains relatively stable.

While our model assumes uniform distributions of updates and reads, we also analyze its robustness under skewed access patterns. In the worst case where reads always access the full α records, the total cost increases moderately, and the deviation remains bounded. Detailed derivations and empirical evidence under skewed workloads are provided in the technical report.

3.2.3 Data update effect. To account for the effect of data updates in column selection, we introduce a penalty term into the objective function of Equation 3. After determining the optimal synchronization threshold α, the total costs associated with the delta table, denoted as Cost_Δ, can be computed. These costs, incurred from both reading and synchronizing the delta table, act as a penalty for not selecting frequently updated columns.

Substitute the $\alpha = \sqrt{\frac{2bw_3}{vw_0}}$ into the Equation 4, we can get the update cost for table j:

$$\text{Cost}_\Delta^j = b_j w_2 + \sqrt{2b_j w_3 v_j w_0} + v_j w_1, \quad 0 < j \le J \tag{5}$$

Let J denote the total number of tables, where b_j represents the number of UID operations on table j, and v_j denotes the number of accesses to delta table j. If any column from a table is selected, a delta table must be prepared for that table to handle UID operations, which incurs a cost Cost_Δ. To account for this, we introduce an additional decision variable, u_j, to indicate whether the columns of table j are loaded into memory. To ensure that the penalty is only applied when columns from a table are selected, we add a corresponding constraint. The overall optimization model is as follows:

$$\begin{aligned}
&\text{Minimize } \sum_{l=1}^{K}\left(\text{cost}_{\text{col}}^{q_l} z_l + \text{cost}_{\text{row}}^{q_l}(1 - z_l)\right) \cdot fl + \sum_{j=1}^{J} \text{Cost}_\Delta^j \cdot u_j \\
&\text{Subject to } \sum_{m=1}^{M} w_m x_m \le \mathcal{M}_{\text{col}}, \\
&\quad \sum_{m \in G_l} x_m \ge |G_l| \cdot z_l, \quad 0 < l \le K, \\
&\quad u_j \ge x_m, \quad \text{for all } m \in S_j, \text{ and } 0 < j \le J \\
&\quad x_m \in \{0, 1\}, 0 < m \le M.
\end{aligned} \tag{6}$$

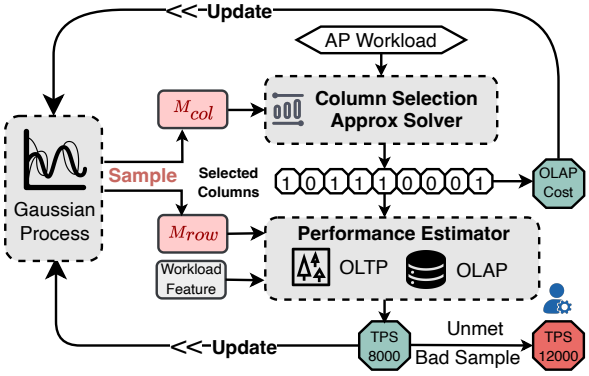

Figure 5: T^2 Workflow for Static Workload Optimization

where S_j denotes the set of columns in table j. This constraint ensures that u_j is activated (set to 1) if any column m from table j is selected, incurring the additional delta table cost.

The model defined in Equation 6 is an integer programming model, which can be solved using exact methods such as branch-and-bound. These methods, when applied via solvers like CBC [15], are effective for small workloads (e.g., CH-benCHmark [11], HyBench [51]), often delivering exact solutions within seconds. However, as the problem's dimensions, represented by M and K, grow, the computational and memory requirements for finding exact solutions can increase exponentially. In large-scale scenarios, heuristic and approximation algorithms [26, 31, 43] are typically employed to find near-optimal solutions within a reasonable timeframe. We also present an approximate solver tailored to the column selection problem in Section 3.3.2.

However, the parameter $\mathcal{M}_{\text{col}}$ is user-defined in the current model. Our goal is to develop a method that allows the model to automatically determine the optimal value of $\mathcal{M}_{\text{col}}$ for the HTAP system. We discuss our proposed solution below.

3.3 Bayesian Optimization for Memory Allocation

The key of our memory allocation algorithm is to achieve the goal defined in Section 2.2.1 by determining the values of $\mathcal{M}_{\text{col}}$ and $\mathcal{M}_{\text{row}}$ within the constraint of limited memory. However, the relationship between memory allocation and system performance in OLTP and OLAP workloads is complex, with non-linear and counterintuitive effects. This makes it difficult to optimize memory allocation without getting stuck in local optima. Additionally, evaluating performance is computationally expensive due to the need for invoking the column selector.

To address this, we adopt a Bayesian optimization framework [16], which efficiently explores the memory allocation space by balancing exploitation and exploration, leading to a near-optimal solution without exhaustive evaluations. Figure 5 illustrates the workflow of the T^2 algorithm for optimizing static workloads:

Step 1: Collect initial memory allocation samples $\mathcal{M}_{\text{col}}$ and evaluate their objective function values $\mathcal{U}(\mathcal{M}_{\text{col}})$.

Step 2: Use the initial observations to build a Gaussian Process model as a surrogate model that approximates $\mathcal{U}(\mathcal{M}_{\text{col}})$.

Step 3: Use an acquisition function, such as Expected Improvement (EI) [23], to select the next memory allocation $\mathcal{M}'_{\text{col}}$ for evaluation, balancing exploration and exploitation.
Step 4: Evaluate $\mathcal{U}(\mathcal{M}'_{\text{col}})$ and update the surrogate model.
Step 5: Repeat Steps 2-4 until convergence, ultimately finding the optimal $\mathcal{M}^*_{\text{col}}$ that minimizes the total cost of OLAP while satisfying the TPS constraint.

Given that the total memory allocated for the buffer pool is fixed, $\mathcal{M}_{\text{col}}$ is treated as the variable, while $\mathcal{M}_{\text{row}}$ is derived as $\mathcal{M}_{\text{total}} - \mathcal{M}_{\text{col}}$. Allocating more memory does not adversely affect the performance of either row store or column store buffers. Within the framework of Bayesian optimization, a major challenge remains: how to define and efficiently determine the influence of $\mathcal{M}_{\text{col}}$ and $\mathcal{M}_{\text{row}}$ on OLTP and OLAP systems in order to achieve the objective outlined in Section 2.2.

3.3.1 Performance Estimator. Firstly, we define the objective function of $\mathcal{U}(\mathcal{M}_{\text{col}})$:

$$\mathcal{U} = \begin{cases} \text{Cost}_{\mathcal{W}_{\text{ap}}}(C) + \text{Cost}_{\Delta}(C, \mathcal{K}), & \text{if } \mathcal{P}(\mathcal{W}_{\text{ap}}, C, \mathcal{M}_{\text{row}}) \geq \theta, \\ \lambda \cdot \Big(\theta - \mathcal{P}(\mathcal{W}_{\text{ap}}, C, \mathcal{M}_{\text{row}})\Big), & \text{if } \mathcal{P}(\mathcal{W}_{\text{ap}}, C, \mathcal{M}_{\text{row}}) < \theta. \end{cases}$$

Here, $\text{Cost}_{\mathcal{W}_{\text{ap}}}(C) + \text{Cost}_{\Delta}(C, \mathcal{K})$ represents the total OLAP cost, including both the query execution cost and the overhead of handling delta tables. The function $\mathcal{P}_{OLTP}(\mathcal{W}_{\text{ap}}, C, \mathcal{M}_{\text{row}})$ extends the simplified model by incorporating additional influencing factors, such as the concurrent OLAP workload and selected columns, to estimate OLTP throughput more accurately.

This objective aligns with the system goal defined in Section 2.2: to satisfy the user's target OLTP TPS while minimizing OLAP query cost. When resource constraints make it infeasible to fully meet the TPS target θ, the optimization seeks to approximate it as closely as possible. To enforce this behavior, we introduce λ as a sufficiently large penalty coefficient, which ensures that configurations leading to large TPS shortfalls are strongly discouraged.

As discussed in earlier sections, the cost of OLAP queries can be estimated either by the database optimizer or through actual query execution. The cost associated with Cost_{Δ} can be estimated using the cost model we proposed in Section 3.2. However, estimating the TPS for OLTP remains a challenge.

The traditional approach [46] uses logarithmic regression to model the relationship between $\mathcal{M}_{\text{row}}$ and TPS. However, this is insufficient in HTAP scenarios, where the execution of OLAP queries can also significantly affect TPS. If analytical queries need to retrieve data from the row store, they may cause transactional queries to experience longer wait times due to I/O requests.

To better capture such interference, we construct a richer feature space to train our TPS prediction model. Each sample is represented as a tuple $<\mathcal{M}_{\text{row}}, \mathcal{W}_{\text{ap}}, C>$, where $\mathcal{M}_{\text{row}}$ and C denote the memory allocated to the row store and the columns loaded into the column store. From $\mathcal{W}_{\text{ap}}$, we extract two feature vectors: $\mathbf{scan}_{\text{seq}}$, with $\mathbf{scan}_{\text{seq}}[j]$ indicating the number of sequential scans on table j, and $\mathbf{scan}_{\text{index}}$, with $\mathbf{scan}_{\text{index}}[k]$ indicating the number of index scans on index k. These features quantify the impact of OLAP queries on OLTP performance.

We explored several regression models that are well-suited for capturing nonlinear relationships. Experimental results (Table 6) show that tree-based models achieve significantly higher accuracy under the same feature set, and GBT offers the best trade-off between prediction accuracy and inference efficiency. Therefore, we adopt GBT as the TPS prediction model.

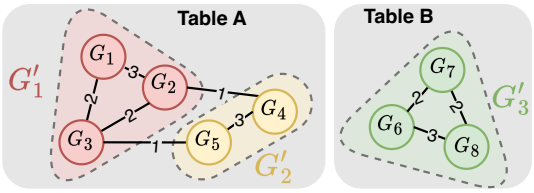

Figure 6: Graph Construction for Column Combination Similarity and Community Detection

Our approach assumes a stable runtime environment, such as a standalone deployment or a resource-isolated container, where external resource contention is minimal.

The model $\mathcal{P}$ is trained using historical execution data, with additional samples generated via simulations during idle periods, enabling accurate predictions of TPS under the current system state. However, during the Bayesian optimization process, we must evaluate $\mathcal{U}(\mathcal{M}_{\text{col}})$ multiple times, with each evaluation requiring the selected column set C, which necessitates solving Equation 6, a process that is inherently time-consuming. Therefore, it is crucial to find a more efficient method to obtain the selected column set.

3.3.2 Approximate Solver for Column Selection. In column selection problems, the number of columns M and tables J is fixed and relatively small, while the number of subqueries, K, can be large. Consequently, assigning a decision variable to each column combination G_l rapidly increases problem size, as each introduces an additional constraint, complicating computational complexity. Therefore, it is essential to reduce the dimensionality of K.

Observations of column selection solutions indicate that similar column combinations are often selected concurrently. This insight allows for a reduction in the solution space by grouping similar column combinations before their input into the solver. As illustrated in Figure 6, we construct a graph to encapsulate the similarity between column combinations. The vertex set V consists of all unique column combinations, where each vertex $v_l \in V$ represents a column combination G_l. The similarity between v_l and $v_{l'}$ is quantified by the number of shared columns $|G_l \cap G_{l'}|$. Vertices are interconnected if the similarity is positive, with edges weighted by this similarity. Given that column combinations from different tables do not share columns, vertices from different tables are not linked.

To simplify the dimensionality of K, we group similar column combinations to form a union, representing all grouped column combinations. Spectral clustering [47] is applied to determine the community partitions of the graph, identifying K' communities. In the figure, vertex colors differentiate the communities. From another perspective, this approach groups similar subqueries into the same community. Consequently, in the optimization problem, a single decision variable per community suffices to determine whether all queries within that community will perform a column scan, effectively reducing the dimension of K to K'.

By limiting the number of communities K', we can control the problem size within a certain range, thereby keeping the solving

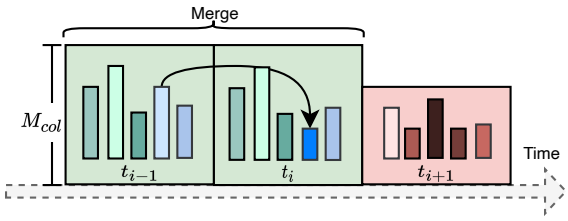

Figure 7: Illustration of Time Interval Merging

time within limits. Additionally, during the Bayesian optimization process, since the column selection approx solver needs to be called repeatedly, we can pre-determine the community partitions. This way, the community detection algorithm only needs to be invoked once during the entire Bayesian optimization process, further reducing the solver overhead.

4 T^2 FOR DYNAMIC WORKLOADS

4.1 Approach Overview

Real-world application workloads fluctuate over time, rendering static models ineffective as they may provide outdated results. While one solution is to periodically invoke the static model, this approach is both passive and inefficient.

In this section, we present our solution for managing dynamic workloads. To address these varying demands, we employ an advanced time series prediction model, PatchTST [32], to forecast future workloads. For OLTP queries, we predict the total number of requests, without distinguishing between individual queries. However, for OLAP queries, we forecast the request rates for each query template Q_s, as the performance characteristics of different templates can vary significantly. Let $R_h^{(Q_s)}$ denote the request rate for template Q_s at time interval h. The model input consists of a historical sequence spanning the past h time intervals.

$$\mathbf{R_h} = \{R_1^{(Q_s)}, R_2^{(Q_s)}, \ldots, R_h^{(Q_s)}\}_{s=1}^{L}$$

where L denotes the total number of query templates. Our goal is to predict the query request rates for the upcoming time periods $h+\gamma$: $\hat{R}_{h+1}^{(Q_s)}, \hat{R}_{h+2}^{(Q_s)}, \ldots, \hat{R}_{h+\gamma}^{(Q_s)}, 0 < s \leq L$. PatchTST enables simultaneous forecasting of multiple time series for $\{Q_1, ..., Q_L\}$, allowing us to leverage the relationships between different query templates, which improves the accuracy of predictions for all OLAP query templates.

With the predictions, we can obtain all necessary parameters for Equation 6, including query frequencies f, UID operation rates b, and delta table read rates v. A straightforward approach is to invoke the static model at the start of each new time window to perform the necessary adjustments. However, this basic solution overlooks the I/O overheads introduced by these adjustments, which can result in suboptimal performance.

We illustrate this issue in Figure 7. Each rectangle represents a time interval, with its height indicating the value of $\mathcal{M}_{\text{col}}$ (the size of the memory buffer for the column store), which fluctuates as OLTP requests change over time. In each time window, T^2 selects the optimal columns for the predicted OLAP workloads using the static model. However, this can lead to suboptimal results when considering multiple time intervals.

For example, if column C_m is selected by our model at time t_i, then discarded and replaced by another column $\bar{C_m}$ at time t_{i+1}, the I/O costs of discarding and loading C_m and $\bar{C_m}$ introduce unnecessary overhead. Worse, if at time t_{i+2}, column C_m is again deemed beneficial to keep in the column store, we end up in an inefficient cycle of repeatedly loading and discarding column C_m.

One possible solution is to merge multiple time intervals and optimize for the entire period. We propose a dynamic reallocation trigger algorithm to adaptively merge time intervals. This algorithm adapts memory allocation more efficiently by continuously monitoring workload changes and reallocating memory resources only when significant shifts are detected, minimizing both I/O overhead and the risk of suboptimal memory usage.

4.2 Dynamic Reallocation Trigger Algorithm

The dynamic problem can be reduced to the multi-period knapsack problem [14], which is known to be strongly NP-hard. Due to space limitations, we omit the formal proof. Furthermore, this problem becomes even more complex because each time the re-optimization process is applied, there is a switching cost, the cost incurred by loading columns that were not present in the previous interval, denoted as $\text{Cost}_{\text{switch}}$:

$$\text{Cost}_{\text{switch}}^{t_i} = \sum\nolimits_{T_j \in Tables(\Delta C_{t_i})} \text{Cost}_{\text{row}}(T_j)$$

where $\Delta C_{t_i} = C_{t_i} \setminus C_{t_{i-1}}$ represent the set of columns present in interval t_i but not in interval t_{i-1}, and $Tables(\Delta C_{t_i})$ be a function that returns a set of unique tables containing columns in ΔC_{t_i}. $\text{Cost}_{\text{row}}(T_j)$ denotes the cost for performing full table scan on table T_j from row store, which is the main cost for loading columns of table T_j. The challenge is to determine when it is worth incurring the switching cost to reorganize. This decision is encapsulated by the function $\text{F}(t_i)$ introduced in Equation 2.

To address this challenge, we extend the heuristic algorithm from [49] and propose a greedy algorithm that incrementally merges adjacent intervals based on potential cost savings. The key idea is to gradually merge intervals to identify local optimal solutions, continuing this process until intervals with significantly different query patterns are encountered.

We define the function $\mathcal{S}(t_i)$ to calculate the total cost of OLAP and the switch cost during the time period t_i (or across a group of time intervals):

$$\mathcal{S}(t_i) = \mathcal{U}(\mathcal{M}_{\text{col}}^{t_i}) + \text{Cost}_{\text{switch}}^{t_i}$$

Algorithm 1 presents our pseudocode. The variable ϕ denotes groups of time intervals. Initially, each interval is its own group. We evaluate potential savings by merging adjacent intervals. As depicted in Figure 7, we group interval t_i with t_{i+1}, calculate the new cost $\mathcal{S}(t_i \cup t_{i+1})$, and compare it to the cost when the intervals are separate. This difference is considered as a cost saving. If merging the intervals results in positive savings, the pair is added to a max-heap as a candidate for merging, as executed in the function $\text{CalculateAndPushSavings}(\phi, i, H)$.

The algorithm iteratively merges the pair of intervals with the highest potential savings, updates the group list, recalculates savings for adjacent intervals, and adjusts the heap accordingly. This process continues until no further profitable merges are possible, at which point the final set of merged intervals is returned. The output ϕ represents the grouped time intervals. Since the boundaries of these groups dictate when memory reorganizations should occur,

Benchmark	SF	Size	J	M	#AP	#Gen
HyBench	100x	129G	8	68	13	1000
CH-benCHmark	500x	57G	12	108	22	1000

Table 1: Benchmark Specifications, detailing scale factor (SF), total size, number of tables (J), columns (M), analytical query templates (#AP) and generated queries (#Gen).

we use the function $F(t_i)$ from Equation 2 to determine whether a reorganization is necessary at the start of each interval t_i.

Each invocation of CalculateAndPushSavings(ϕ, i, H) triggers the static algorithm described in Section 3 to compute $\mathcal{U}(\mathcal{M}_{\text{col}}^{t_i})$. The resulting memory allocation for $\mathcal{M}_{\text{col}}$ and $\mathcal{M}_{\text{row}}$, along with the selected column set C for interval t_i, can then be cached. When triggers reorganizations, we can retrieve solution from the cache. Then these changes can be applied to the database using the relevant tuning tools provided by the database.

In this way, T^2 offers an approximate solution to Equation 2 for dynamic workloads, optimizing memory allocation between OLTP and OLAP tasks. Although this approach may not always yield a globally optimal solution due to the complexity of dynamic workloads, it still outperforms static optimization.

Algorithm 1 Greedy Merge Intervals with Max-Heap

```
Initialize interval groups φ = [[1], [2], ..., [n]]
Initialize an empty max-heap H, to store merge savings
procedure CalculateAndPushSavings(φ, i, H)
    saving = S(φ[i]) + S(φ[i+1]) − S(φ[i] ∪ φ[i+1])
    if saving > 0 then
        Push (saving, i, i+1) into H
    end ifreturn H
end procedure
for i = 1 to n − 1 do
    H = CalculateAndPushSavings(φ, i, H)
end for
while not H.isEmpty() do
    (saving, i, j) = H.popMax()
    Merge groups: φ[i] = φ[i] ∪ φ[j]
    Remove φ[j] from the list φ
    for j ∈ {i − 1, i + 1} and 1 ≤ j ≤ n do
        H = CalculateAndPushSavings(φ, j, H)
    end for
end while
return φ
```

5 EXPERIMENT

In this section, we evaluate the performance of T^2, built on the GaussDB-HTAP. Our experiments underscore the importance of integrating a memory management algorithm to optimize performance in HTAP systems. We first assess the overall performance of T^2 through a one-day workload simulation. Following this, we conduct a detailed analysis of the effectiveness of its key modules: performance estimator, dynamic algorithm, column selection and memory allocation.

5.1 Experimental Setup

5.1.1 Configurations. The experiments were conducted on a server equipped with an Intel(R) Xeon(R) Gold 6161 CPU operating at 2.20GHz, 503GB memory, 7.3T SSD. We integrated our tool into centralized GaussDB-HTAP, running on EulerOS.

5.1.2 Benchmark. We construct our benchmarks by combining real-world query arrival rate patterns with synthetic workloads. Specifically, we extract OLTP and OLAP query rates from real anonymized database logs. Due to privacy concerns, we cannot use the actual queries or data directly. Instead, we simulate realistic hybrid workloads by issuing synthetic queries according to the real-world arrival rate distributions.

Table 1 summarizes the two synthetic benchmarks used: HyBench [51] and CH-benCHmark [11]. **HyBench** emulates banking workloads in HTAP settings and includes 18 transactions and 13 analytical queries. We use a scale factor of 100x and focus on the transactional and analytical components. **CH-benCHmark** combines the transactional workload of TPC-C with analytical queries adapted from TPC-H. We use a scale factor of 500x.

To evaluate the effectiveness of T^2, we not only used workloads generated by standard benchmarks but also incorporated synthetic queries to simulate fluctuating query patterns over time. We adopted a typical aggregation query template:

select agg_func(t.a), t.b, ..., from t where t.c > $1 group by t.b;

Here, the tables and columns are randomly selected from the benchmark schemas. We generate 1,000 such queries for each benchmark, resulting in CH-benCHmark-Gen and HyBench-Gen workloads.

During the experiments, transactional queries are issued at a fixed Requests Per Second (RPS), following patterns derived from the real workload. For example, when the RPS was set to 5,000, the system dynamically adjusted the sleep intervals between query dispatches to maintain a steady rate of 5,000 queries per second. Similarly, the timing and volume of analytical queries also follow the real-world OLAP query rate curve.

5.1.3 Metric. To assess the system's ability to handle OLTP demands under varying load conditions, we introduce the **Fulfillment Ratio (FR)** metric. FR quantifies how effectively the system satisfies incoming query requests and is defined as:

$$FR = TPS/RPS$$

where TPS is the number of transactional queries completed per second, and RPS is the incoming transactional demand. FR reflects system performance: FR = 1 means full demand is met, while FR < 1 indicates the system cannot handle all incoming requests.

For OLAP workloads, we define the metric **Impr** to quantify the improvement in analytical query execution time, comparing the performance of the HTAP-enabled system (GaussDB-HTAP) with the original system (GaussDB), which lacks HTAP capabilities:

$$Impr = (T_{GaussDB} - T_{GaussDB-HTAP})/T_{GaussDB} \times 100\%$$

The original GaussDB uses only row-format storage with a single row store buffer and does not support column store or column selection/eviction. In contrast, GaussDB-HTAP introduces a separate column store buffer alongside the row store buffer, enabling manual column loading and efficient OLAP execution.

5.1.4 Comparison Approaches. As no prior work fully manages memory contents, we evaluate several column selection and memory allocation methods. For column selection, we consider:

HyBench($\mathcal{M}_{\text{total}} = 80G$)												
Method	Static-7:1		Static-1:1		Static-1:7		STMM		T^2-static		T^2-dynamic	
	FR	Impr	FR	Impr	FR	Impr	FR	Impr	FR	Impr	FR	Impr
HAMCS	1.00	13.56%	1.00	22.20%	0.912	23.11%	1.00	27.95%	1.00	28.06%	1.00	35.88%
IPNC	1.00	15.74%	1.00	32.12%	0.936	39.41%	1.00	34.98%	1.00	36.45%	1.00	48.70%
GACC	1.00	12.03%	1.00	40.62%	0.940	44.29%	1.00	39.38%	1.00	42.49%	1.00	50.28%
T^2-CS-Approx	1.00	16.39%	1.00	49.33%	0.953	60.03%	1.00	46.97%	1.00	49.92%	1.00	65.90%
T^2-CS	1.00	18.13%	1.00	51.36%	0.956	60.69%	1.00	51.17%	1.00	54.44%	1.00	**67.07%**
CH-benCHmark($\mathcal{M}_{\text{total}} = 30G$)												
Method	Static-7:1		Static-1:1		Static-1:7		STMM		T^2-static		T^2-dynamic	
	FR	Impr	FR	Impr	FR	Impr	FR	Impr	FR	Impr	FR	Impr
HAMCS	1.00	15.38%	1.00	23.72%	0.872	34.62%	1.00	25.39%	1.00	27.42%	1.00	48.08%
IPNC	1.00	5.77%	1.00	33.97%	0.865	52.56%	1.00	34.84%	1.00	35.52%	1.00	47.44%
GACC	1.00	18.59%	1.00	33.33%	0.841	54.49%	1.00	35.27%	1.00	37.24%	1.00	44.62%
T^2-CS-Approx	1.00	20.14%	1.00	36.84%	0.887	62.93%	1.00	37.21%	1.00	41.78%	1.00	71.29%
T^2-CS	1.00	22.35%	1.00	39.26%	0.891	63.75%	1.00	38.25%	1.00	45.24%	1.00	**74.36%**

Table 2: Overall Performance Evaluation of Combined Memory Allocation and Column Selection Methods

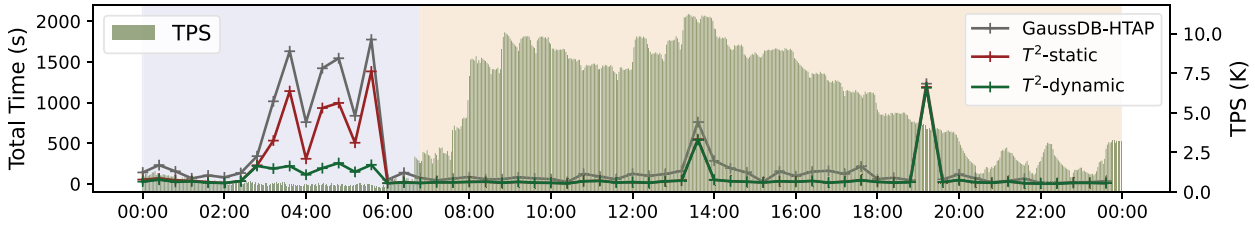

Figure 8: Performance of OLTP and OLAP Workloads During a One-Day Simulation (HyBench $\mathcal{M}_{\text{total}}$ = 80G)

1) **HAMCS.** Heatmap Automated Memory Column Selection, used in commercial HTAP systems like Oracle DUAL [33] and SQL Server [29], dynamically adjusts in-memory columns based on access frequency, retaining hot columns and evicting cold ones.

2) **IPNC.** The Integer Programming without Correlation (IPNC) method [3] transforms the column selection problem into a 0/1 knapsack problem. It based on the assumption of independence and does not account for correlations between multiple columns.

3) **GACC.** We develop the Greedy Algorithm for Column Combinations (GACC), a heuristic that selects columns with the highest benefit-to-memory ratio under memory constraints (see technical report for pseudocode).

4) **T^2-CS.** This algorithm, detailed in Section 3.1, utilizes CBC [15] to solve the optimization model. **T^2-CS-Approx.** is the approximation version of T^2 proposed in Section 3.3.2.

For memory allocation, we examine the following methods:

1) **Static-$\mathcal{M}_{\text{row}}$:$\mathcal{M}_{\text{col}}$.** This approach employs a fixed memory allocation. In our experiments, we evaluate allocation ratios of 7:1, 1:1, and 1:7 for the memory of row and column store buffers.

2) **STMM.** The Self-Tuning Memory Manager (STMM) [42] used in IBM DB2 [36], adaptively optimizes memory usage across DBMS caches. While it does not target columnar caches, we adapt its principles to enable dynamic memory allocation between row and column buffers. To our knowledge, no existing method addresses this specific objective.

3) **T^2-static.** This approach indicates the memory allocation described in Section 3.3, where Bayesian optimization is used to determine the optimal values for $\mathcal{M}_{\text{row}}$ and $\mathcal{M}_{\text{col}}$.

4) **T^2-dynamic.** This approach refers to the dynamic memory allocation method outlined in Section 4, which allows for the dynamic adjustment of both memory allocation and selected columns.

GaussDB-HTAP offers tuning tools to adjust memory for row and column buffers, load and discard columns, and trigger data synchronization. We implemented these comparative methods and used the tuning tools to interact with GaussDB-HTAP.

5.2 Overall Performance

In this experiment, we simulated a day-long hybrid workload scenario, dividing the day into 60 intervals, each representing a real-time duration of 24 minutes, to evaluate the effectiveness of our proposed approach. We extracted realistic hybrid workload patterns, such as RPS for OLTP and OLAP query request rates, from GaussDB logs, and performed data anonymization before conducting the experiment to ensure privacy protection. Both OLTP and OLAP workloads exhibit tidal-like patterns, with fluctuations between day and night, which allowed us to adjust memory allocation between the two components accordingly.

For the OLTP tests in HyBench and CH-benCHmark, we used the original benchmark queries. The OLAP queries were generated from query templates, consisting of both the original benchmark templates and our synthetic templates, with a 22:78 ratio for CH-benCHmark and 13:87 for HyBench. Random queries were uniformly generated from these templates and continuously fed to the database for processing.

Metric	GaussDB	Static-7:1	Static-4:4	Static-1:7	STMM	T^2
$\mathcal{M}_{\text{row}}$ (G)	80.00	70.00	40.00	10.00	40.07	22.73
$\mathcal{M}_{\text{col}}$ (G)	0.00	10.00	40.00	70.00	39.93	57.27
Hit Ratio (%)	98.22	98.32	97.68	91.38	97.67	95.33
I/O Impr (%)	0.00	14.20	35.22	77.35	35.34	72.97

Table 3: Memory Footprint of experiments on HyBench

$\mathcal{M}_{\text{total}}$		5G	15G	30G	60G
STMM	$\mathcal{M}_{\text{col}}$	2.55	7.53	15.01	29.95
	FR	0.59	0.70	0.98	1.00
	Impr	15.98%	20.08%	18.18%	36.78%
T^2-static	$\mathcal{M}_{\text{col}}$	0.02	0.11	9.12	32.06
	FR	0.85	0.93	1.00	1.00
	Impr	1.13%	1.64%	16.49%	39.33%
T^2-dynamic	FR	0.86	0.95	1.00	1.00
	Impr	7.78%	11.89%	22.54%	53.69%

Table 4: Evaluation with Varying $\mathcal{M}_{\text{total}}$(HyBench)

Table 2 presents the results of a day-long simulation, combining various memory allocation methods with different column selection strategies. The horizontal axis represents memory allocation techniques, and the vertical axis lists column selection methods. For both T^2-static and T^2-dynamic, when combined with different column selection methods, the column selection algorithm replaces the column selection solver in the Bayesian optimization process. Once the memory allocation is determined, the selected column method is also used to choose the columns. The table records the FR and Impr metrics to evaluate performance for both OLTP and OLAP workloads. The results show that all approaches achieve improvements over the original GaussDB, as indicated by the positive Impr values. However, when the memory allocation ratio is set to 1:7, the methods fail to meet the OLTP SLA (with $FR < 1$), highlighting the importance of choosing appropriate memory allocation strategies to ensure OLTP requirements are met. By comparing different memory allocation methods with the same column selection strategy, the T^2-static method outperforms all other static approaches, and T^2-dynamic further amplifies the advantages of the T^2 approach in both the HyBench and CH-benCHmark scenarios. In the column selection method comparison, the optimal version, T^2-CS, and its approximate version, T^2-CS-Approx, outperform all other methods.

Table 3 presents the memory allocation and key performance metrics on HyBench. Both STMM and T^2 support fine-grained memory control, which is critical. Because small differences in column-store allocation can determine column loading feasibility and affect many queries. The table also includes cache hit rates and I/O improvement metrics, which reflect OLAP query I/O savings. As expected, the hit ratio correlates with $\mathcal{M}_{\text{row}}$, while the improvement in I/O increases with $\mathcal{M}_{\text{col}}$.

Figure 8 illustrates the behavior of OLTP and OLAP workloads during the experiment. The bars represent TPS, while the line graph shows the total query latency for OLAP queries in each interval. Different background colors distinguish between the nighttime and daytime workloads. Notably, by dynamically adjusting the memory configuration, we can significantly accelerate OLAP workloads during the night, when there are fewer OLTP requests, allowing more memory to be allocated to the column store. During the day, most of the memory is allocated to the OLTP module, and while the dynamic strategy still outperforms the static approach, the margin of improvement is smaller.

Workload	HAMCS	IPNC	GACC	T^2-CS-Approx	T^2-CS
HyBench	102.35	155.75	103.21	168.4	230.18
CH-benCHmark	81.13	137.36	104.68	143.15	193.82

Table 5: Execution Time (s) Comparison of Different Methods with T^2-dynamic Memory Allocation Strategy

Model	Log Reg.	RF	SVR	MLP	GBT
Training(ms)	47.67	674.64	13.31	1223.75	245.13
Inference(ms)	0.01	0.27	0.10	0.04	0.04
MAE(TPS)	1276.87	563.04	1889.10	2008.90	**556.16**
MARE	13.72%	5.34%	23.38%	19.33%	**5.23%**

Table 6: Comparing Algorithms for TPS Prediction

We evaluate the performance of T^2 under reduced $\mathcal{M}_{\text{total}}$, as shown in Table 4. When memory is tight, the model prioritizes OLTP throughput, allocating most memory to the row store and maintaining high FR. Compared to STMM, T^2 achieves higher FR in low-memory settings, demonstrating its OLTP-aware optimization.

Table 5 presents the running times of different column selection methods combined with T^2-dynamic. While T^2-CS-Approx and T^2-CS are not the most efficient methods, they are still sufficiently fast for practical use, as the algorithm can determine memory management actions for the upcoming 24 hours.

5.3 Evaluation of Different Components

5.3.1 Effect of Performance Estimator. We compare GBT [17] with several regression models, including Logarithmic Regression [46], Random Forest (RF) [4], Support Vector Regression (SVR) [13], and Multi-layer Perceptron (MLP) [37]. All models are trained on the same dataset with identical features. The evaluation metrics are Mean Absolute Error (MAE) and Mean Absolute Relative Error (MARE). As shown in Table 6, tree-based models (RF and GBT) outperform others in accuracy. GBT achieves the lowest MAE and MARE, along with fast training and inference time, making it suitable for real-time use. This evaluation is conducted under a stable runtime environment, as assumed by our system design. We refer readers to our technical report for detailed robustness experiments under disturbed conditions.

5.3.2 Effect of Dynamic Algorithm. In this experiment, we validate the importance of Algorithm 1. As shown in Figure 9, we compare our dynamic reallocation approach with the static approach, as well as with approaches that trigger reallocation every 1, 6, and 12 intervals (each representing 24 minutes). Following the experiment setting in Section 5.2, we test the average query processing time and column loading time for each interval. The percentages in the figure represent the percentage reduction in total time relative to the static algorithm. Static or infrequently updated methods reduce I/O for column loading but can't adapt to changing query patterns, leading to suboptimal OLAP performance. Conversely, frequent adjustments incur high I/O overhead. Our dynamic approach balances these extremes, optimizing both performance and I/O costs.

5.3.3 Effect of Column Selection. In this experiment, we evaluate the performance of our proposed column selection algorithm. As

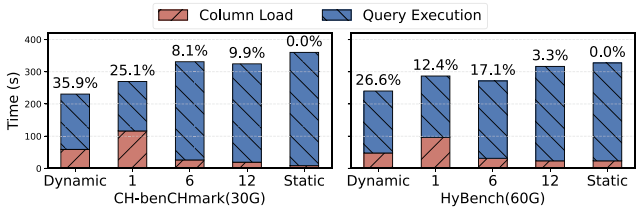

Figure 9: Effect of Reallocation Trigger Algorithm

Method	Hybench		CH-benCHmark	
	Max loss	Max Time	Max loss	Max Time
HAMCS	-41.29%	0.003s	-25.99%	0.00s
IPNC	-8.42%	0.08s	-19.49%	0.17s
GACC	-7.62%	18.17s	-7.17%	8.69s
T^2-CS-Approx	**-2.55%**	10.10s	**-3.49%**	9.06s
T^2-CS	-	45.58s	-	65.93s

Table 7: Degradation and Execution Time Analysis of Different Column Selection Algorithms

shown in Table 7, since T^2 provides the optimal solution, we use it as the baseline for comparison. "Max loss" denotes the largest reduction in time savings relative to the optimal method T^2, measured as a percentage of the query time on row store. "Max time" indicates the maximum execution time under different memory sizes for $\mathcal{M}_{\text{col}}$. Although some competing methods occasionally achieve performance similar to that of T^2, they are prone to getting trapped in local optima, leading to a higher max loss. To ensure a more controllable execution time for solving the integer programming formulation in our column selection, we propose T^2-CS-Approx. As the results show, T^2-CS-Approx delivers the least performance degradation while maintaining a controlled execution time.

5.3.4 Effect of Memory Competition. To evaluate the impact of memory competition between row and column buffers, we compared T^2 with fixed memory allocation ratios under varying RPS loads of 9,000 and 13,000. The tested ratios were $1:7$, $3:5$, $5:3$, and $7:1$, representing different memory distributions between the row and column buffers. The total memory available was set to 40GB and 80GB. Performance was measured by the average execution time of OLAP workloads and the Fulfillment Ratio (FR), with OLAP queries as described in Section 5.2. As shown in Figure 10, when RPS reaches 9,000 or 13,000 and $\mathcal{M}_{\text{row}}$ is insufficient, the FR drops below 1, potentially violating SLA requirements. Under these high-load conditions, T^2 optimally adjusts memory allocation ratios to meet OLTP requirements while minimizing OLAP execution times, for both $\mathcal{M}_{\text{total}}$ = 40GB and 80GB.

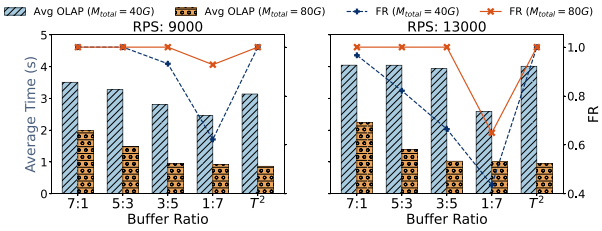

Figure 10: Comparative Analysis of T^2 Versus Fixed Ratio Memory Allocation (HyBench)

6 RELATED WORK

HTAP Architectures. HTAP systems can be categorized based on their storage architectures [41]. Row-native systems, such as SQL Server, Oracle, and PolarDB-IMCI [27, 29, 48], primarily store data in a row format, supplemented by a secondary in-memory column store. Column-native systems, like SAP HANA and MemSQL [10, 40], store data in columns with in-memory delta stores for updates. Row-native architectures, including GaussDB-HTAP, prioritize OLTP workloads but face challenges related to column selection and memory allocation when adapting to OLAP queries. This paper emphasizes the need for adaptive memory management in such systems to improve OLAP handling and overall performance. Although different HTAP systems have design differences, T^2 can be ported across them with minor modifications.

Column Selection and Index Recommendation. Systems like SQL Server, Oracle, and PolarDB-IMCI [27, 29, 48] construct in-memory column stores for HTAP applications. Oracle Dual [33] dynamically manages columnar data based on access frequency, keeping frequently accessed columns and evicting less-used ones. Approaches like [3] treat column selection as a knapsack problem, but neglect the interdependencies between columns. Similarly, research on index recommendation shares common ground with our scenario. Some studies frame the problem as a knapsack problem [7, 12], but unlike column selection, it lacks the 'all-or-none' feature. Learning-based methods [25, 28, 34, 50] are often costly to train and challenging to transfer across different datasets.

Automatic Database Tuning. Automatic database tuning has been extensively studied in previous works, such as tuning systems [2, 6, 45], index advisor [8, 24, 38], and automatic view advisor [1, 5, 20]. However, there is no previous work on HTAP system tuning, particularly with regard to memory allocation between columnar and row-based storage. Our work fills this critical gap by introducing the first automated tool for optimizing HTAP systems.

7 CONCLUSION

In this paper, we introduce T^2, an adaptive memory management approach that better aligns with user-side requirements in HTAP systems, prioritizing OLTP queries while optimizing OLAP queries. T^2 adjusts memory allocation between OLTP and OLAP to efficiently balance two workloads, featuring a column selection strategy and a data synchronization module. The column selection strategy identifies the most valuable columns for OLAP queries to be maintained in the in-memory column store, while the synchronization module optimizes the timing for merging updated data from the OLTP system with the in-memory column store. By dynamically allocating memory, optimizing column selection, and synchronizing data, T^2 ensures OLTP SLA compliance while enhancing OLAP performance. Our experiments show that T^2 outperforms both traditional OLTP systems and static memory allocation methods, demonstrating its effectiveness in real-world HTAP scenarios.

ACKNOWLEDGMENTS

This work has been supported by Zhejiang Province "Jianbing" Key R&D Project of China (No.2025C01010), CCF-Huawei Populus Grove Fund (No.CCF-HuaweiDB2022007).

REFERENCES

[1] Rafi Ahmed, Randall G. Bello, Andrew Witkowski, and Praveen Kumar. 2020. Automated Generation of Materialized Views in Oracle. *Proc. VLDB Endow.* 13, 12 (2020), 3046–3058. https://doi.org/10.14778/3415478.3415533

[2] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 1009–1024. https://doi.org/10.1145/3035918.3064029

[3] Martin Boissier, Rainer Schlosser, and Matthias Uflacker. 2018. Hybrid Data Layouts for Tiered HTAP Databases with Pareto-Optimal Data Placements. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 209–220. https://doi.org/10.1109/ICDE.2018.00028

[4] Leo Breiman. 2001. Random Forests. *Mach. Learn.* 45, 1 (2001), 5–32. https://doi.org/10.1023/A:1010933404324

[5] Mihai Budiu, Tej Chajed, Frank McSherry, Leonid Ryzhyk, and Val Tannen. 2023. DBSP: Automatic Incremental View Maintenance for Rich Query Languages. *Proc. VLDB Endow.* 16, 7 (2023), 1601–1614. https://doi.org/10.14778/3587136.3587137

[6] Baoqing Cai, Yu Liu, Ce Zhang, Guangyu Zhang, Ke Zhou, Li Liu, Chunhua Li, Bin Cheng, Jie Yang, and Jiashu Xing. 2022. HUNTER: An Online Cloud Database Hybrid Tuning System for Personalized Requirements. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 646–659. https://doi.org/10.1145/3514221.3517882

[7] Alberto Caprara, Matteo Fischetti, and Dario Maio. 1995. Exact and Approximate Algorithms for the Index Selection Problem in Physical Database Design. *IEEE Trans. Knowl. Data Eng.* 7, 6 (1995), 955–967. https://doi.org/10.1109/69.476501

[8] Surajit Chaudhuri and Vivek R. Narasayya. 1997. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld (Eds.). Morgan Kaufmann, 146–155. http://www.vldb.org/conf/1997/P146.PDF

[9] Jianjun Chen, Yonghua Ding, Ye Liu, Fangshi Li, Li Zhang, Mingyi Zhang, Kui Wei, Lixun Cao, Dan Zou, Yang Liu, Lei Zhang, Rui Shi, Wei Ding, Kai Wu, Shangyu Luo, Jason Sun, and Yuming Liang. 2022. ByteHTAP: ByteDance's HTAP System with High Data Freshness and Strong Data Consistency. *Proc. VLDB Endow.* 15, 12 (2022), 3411–3424. https://doi.org/10.14778/3554821.3554832

[10] Jack Chen, Samir Jindel, Robert Walzer, Rajkumar Sen, Nika Jimsheleishvilli, and Michael Andrews. 2016. The MemSQL Query Optimizer: A modern optimizer for real-time analytics in a distributed database. *Proc. VLDB Endow.* 9, 13 (2016), 1401–1412. https://doi.org/10.14778/3007263.3007277

[11] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi Kuno, Raghunath Nambiar, Thomas Neumann, Meikel Poess, Kai-Uwe Sattler, Michael Seibold, Eric Simon, and Florian Waas. 2011. The mixed workload CH-benCHmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems* (Athens, Greece) *(DBTest '11)*. Association for Computing Machinery, New York, NY, USA, Article 8, 6 pages. https://doi.org/10.1145/1988842.1988850

[12] Debabrata Dash, Neoklis Polyzotis, and Anastasia Ailamaki. 2011. CoPhy: A Scalable, Portable, and Interactive Index Advisor for Large Workloads. *Proc. VLDB Endow.* 4, 6 (2011), 362–372. https://doi.org/10.14778/1978665.1978668

[13] Harris Drucker, Christopher J. C. Burges, Linda Kaufman, Alexander J. Smola, and Vladimir Vapnik. 1996. Support Vector Regression Machines. In *Advances in Neural Information Processing Systems 9, NIPS, Denver, CO, USA, December 2-5, 1996*, Michael Mozer, Michael I. Jordan, and Thomas Petsche (Eds.). MIT Press, 155–161. http://papers.nips.cc/paper/1238-support-vector-regression-machines

[14] Bruce H. Faaland. 1981. Technical Note - The Multiperiod Knapsack Problem. *Oper. Res.* 29, 3 (1981), 612–616. https://doi.org/10.1287/OPRE.29.3.612

[15] J. Forrest and the CBC team. [n. d.]. CBC (COIN-OR Branch-and-Cut). https://doi.org/10.5281/zenodo.1334726

[16] Peter I. Frazier. 2018. A Tutorial on Bayesian Optimization. *CoRR* abs/1807.02811 (2018). arXiv:1807.02811 http://arxiv.org/abs/1807.02811

[17] Jerome H. Friedman. 2001. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics* 29, 5 (2001), 1189 – 1232. https://doi.org/10.1214/aos/1013203451

[18] Jay B. Ghosh. 1996. Computational aspects of the maximum diversity problem. *Oper. Res. Lett.* 19, 4 (1996), 175–181. https://doi.org/10.1016/0167-6377(96)00025-9

[19] Fred W. Glover and Eugene Woolsey. 1974. Technical Note - Converting the 0-1 Polynomial Programming Problem to a 0-1 Linear Program. *Oper. Res.* 22, 1 (1974), 180–182. https://doi.org/10.1287/OPRE.22.1.180

[20] Yue Han, Guoliang Li, Haitao Yuan, and Ji Sun. 2023. AutoView: An Autonomous Materialized View Management System With Encoder-Reducer. *IEEE Trans. Knowl. Data Eng.* 35, 6 (2023), 5626–5639. https://doi.org/10.1109/TKDE.2022.3163195

[21] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. 2020. TiDB: A Raft-based HTAP Database. *Proc. VLDB Endow.* 13, 12 (2020), 3072–3084. https://doi.org/10.14778/3415478.3415535

[22] Huawei. 2024. GaussDB: Next-Generation Distributed Database. https://www.huaweicloud.com/intl/en-us/product/gaussdb.html Accessed: June 16, 2024.

[23] Donald R. Jones, Matthias Schonlau, and William J. Welch. 1998. Efficient Global Optimization of Expensive Black-Box Functions. *J. Glob. Optim.* 13, 4 (1998), 455–492. https://doi.org/10.1023/A:1008306431147

[24] Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. 2020. Magic mirror in my hand, which is the best in the land? An Experimental Evaluation of Index Selection Algorithms. *Proc. VLDB Endow.* 13, 11 (2020), 2382–2395. http://www.vldb.org/pvldb/vol13/p2382-kossmann.pdf

[25] Jan Kossmann, Alexander Kastius, and Rainer Schlosser. 2022. SWIRL: Selection of Workload-aware Indexes using Reinforcement Learning. In *Proceedings of the 25th International Conference on Extending Database Technology, EDBT 2022, Edinburgh, UK, March 29 - April 1, 2022*, Julia Stoyanovich, Jens Teubner, Paolo Guagliardo, Milos Nikolic, Andreas Pieris, Jan Mühlig, Fatma Özcan, Sebastian Schelter, H. V. Jagadish, and Meihui Zhang (Eds.). OpenProceedings.org, 2:155–2:168. https://doi.org/10.48786/EDBT.2022.06

[26] Sebon Ku and Bogju Lee. 2001. A set-oriented genetic algorithm and the knapsack problem. In *Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No.01TH8546)*, Vol. 1. 650–654 vol. 1. https://doi.org/10.1109/CEC.2001.934453

[27] Tirthankar Lahiri, Shasank Chavan, Maria Colgan, Dinesh Das, Amit Ganesh, Mike Gleeson, Sanket Hase, Allison Holloway, Jesse Kamp, Teck-Hua Lee, Juan Loaiza, Neil MacNaughton, Vineet Marwah, Niloy Mukherjee, Atrayee Mullick, Sujatha Muthulingam, Vivekanandhan Raja, Marty Roth, Ekrem Soylemez, and Mohamed Zaït. 2015. Oracle Database In-Memory: A dual format in-memory database. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, Johannes Gehrke, Wolfgang Lehner, Kyuseok Shim, Sang Kyun Cha, and Guy M. Lohman (Eds.). IEEE Computer Society, 1253–1258. https://doi.org/10.1109/ICDE.2015.7113373

[28] Hai Lan, Zhifeng Bao, and Yuwei Peng. 2020. An Index Advisor Using Deep Reinforcement Learning. In *CIKM '20: The 29th ACM International Conference on Information and Knowledge Management, Virtual Event, Ireland, October 19-23, 2020*, Mathieu d'Aquin, Stefan Dietze, Claudia Hauff, Edward Curry, and Philippe Cudré-Mauroux (Eds.). ACM, 2105–2108. https://doi.org/10.1145/3340531.3412106

[29] Per-Åke Larson, Adrian Birka, Eric N. Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. 2015. Real-Time Analytical Processing with SQL Server. *Proc. VLDB Endow.* 8, 12 (2015), 1740–1751. https://doi.org/10.14778/2824032.2824071

[30] Elena Milkai, Yannis Chronis, Kevin P. Gaffney, Zhihan Guo, Jignesh M. Patel, and Xiangyao Yu. 2022. How Good is My HTAP System?. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1810–1824. https://doi.org/10.1145/3514221.3526148

[31] Nima Moradi, Vahid Kayvanfar, and Majid Rafiee. 2022. An efficient population-based simulated annealing algorithm for 0-1 knapsack problem. *Eng. Comput.* 38, 3 (2022), 2771–2790. https://doi.org/10.1007/S00366-020-01240-3

[32] Yuqi Nie, Nam H. Nguyen, Phanwadee Sinthong, and Jayant Kalagnanam. 2023. A Time Series is Worth 64 Words: Long-term Forecasting with Transformers. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/forum?id=Jbdc0vTOcol

[33] Oracle. 2021. *Oracle. Database In-Memory Guide. Public Documentation 21c.* https://docs.oracle.com/en/database/oracle/oracle-database/21/dbimr/index.html 71–88.

[34] R. Malinga Perera, Bastian Oetomo, Benjamin I. P. Rubinstein, and Renata Borovica-Gajic. 2021. DBA bandits: Self-driving index tuning under ad-hoc, analytical workloads with safety guarantees. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 600–611. https://doi.org/10.1109/ICDE51399.2021.00058

[35] Wendy Powley, Patrick Martin, Nailah Ogeer, and Wenhu Tian. 2005. Autonomic buffer pool configuration in PostgreSQL. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, Waikoloa, Hawaii, USA, October 10-12, 2005*. IEEE, 53–58. https://doi.org/10.1109/ICSMC.2005.1571121

[36] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm, and Liping Zhang. 2013. DB2 with BLU acceleration: so much more than just a column store. *Proc. VLDB Endow.* 6, 11 (Aug. 2013), 1080–1091. https://doi.org/10.14778/2536222.2536233

[37] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. 1986. Learning representations by back-propagating errors. *Nature* 323 (1986), 533–536. https:

//api.semanticscholar.org/CorpusID:205001834

[38] Rainer Schlosser, Jan Kossmann, and Martin Boissier. 2019. Efficient Scalable Multi-attribute Index Selection Using Recursive Strategies. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE, 1238–1249. https://doi.org/10.1109/ICDE.2019.00113

[39] Sijie Shen, Rong Chen, Haibo Chen, and Binyu Zang. 2021. Retrofitting High Availability Mechanism to Tame Hybrid Transaction/Analytical Processing. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, Angela Demke Brown and Jay R. Lorch (Eds.). USENIX Association, 219–238. https://www.usenix.org/conference/osdi21/presentation/shen

[40] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. 2012. Efficient transaction processing in SAP HANA database: the end of a column store myth. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman (Eds.). ACM, 731–742. https://doi.org/10.1145/2213836.2213946

[41] Haoze Song, Wenchao Zhou, Heming Cui, Xiang Peng, and Feifei Li. 2024. A survey on hybrid transactional and analytical processing. *VLDB J.* 33, 5 (2024), 1485–1515. https://doi.org/10.1007/S00778-024-00858-9

[42] Adam J. Storm, Christian Garcia-Arellano, Sam Lightstone, Yixin Diao, and Maheswaran Surendra. 2006. Adaptive Self-tuning Memory in DB2. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim (Eds.). ACM, 1081–1092. http://dl.acm.org/citation.cfm?id=1164220

[43] Javid Taheri, Shaghayegh Sharif, Xing Penju, and Albert Y. Zomaya. 2012. Paralleled Genetic Algorithm for Solving the Knapsack Problem in the Cloud. In *2012 Seventh International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, 3PGCIC 2012, Victoria, BC, Canada, November 12-14, 2012*, Fatos Xhafa, Leonard Barolli, and Kin Fun Li (Eds.). IEEE, 303–308. https://doi.org/10.1109/3PGCIC.2012.54

[44] Wenhu Tian, Patrick Martin, and Wendy Powley. 2003. Techniques for automatically sizing multiple buffer pools in DB2. In *Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative Research, October 6-9, 2003, Toronto, Ontario, Canada*, Darlene A. Stewart (Ed.). IBM, 294–302. https://dl.acm.org/citation.cfm?id=961367

[45] Immanuel Trummer. 2022. DB-BERT: A Database Tuning Tool that "Reads the Manual". In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 190–203. https://doi.org/10.1145/3514221.3517843

[46] Thin-Fong Tsuei, Allan N. Packer, and Keng-Tai Ko. 1997. Database buffer size investigation for OLTP workloads. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data* (Tucson, Arizona, USA) *(SIGMOD '97)*. Association for Computing Machinery, New York, NY, USA, 112–122. https://doi.org/10.1145/253260.253279

[47] Ulrike von Luxburg. 2007. A tutorial on spectral clustering. *Stat. Comput.* 17, 4 (2007), 395–416. https://doi.org/10.1007/S11222-007-9033-Z

[48] Jianying Wang, Tongliang Li, Haoze Song, Xinjun Yang, Wenchao Zhou, Feifei Li, Baoyue Yan, Qianqian Wu, Yukun Liang, ChengJun Ying, Yujie Wang, Baokai Chen, Chang Cai, Yubin Ruan, Xiaoyi Weng, Shibin Chen, Liang Yin, Chengzhong Yang, Xin Cai, Hongyan Xing, Nanlong Yu, Xiaofei Chen, Dapeng Huang, and Jianling Sun. 2023. PolarDB-IMCI: A Cloud-Native HTAP Database System at Alibaba. *Proc. ACM Manag. Data* 1, 2, Article 199 (June 2023), 25 pages. https://doi.org/10.1145/3589785

[49] Christophe Wilbaut, Said Hanafi, and Said Salhi. 2008. A survey of effective heuristics and their application to a variety of knapsack problems. *IMA journal of management Mathematics* 19, 3 (2008), 227–244.

[50] Wentao Wu, Chi Wang, Tarique Siddiqui, Junxiong Wang, Vivek R. Narasayya, Surajit Chaudhuri, and Philip A. Bernstein. 2022. Budget-aware Index Tuning with Reinforcement Learning. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 1528–1541. https://doi.org/10.1145/3514221.3526128

[51] Chao Zhang, Guoliang Li, and Tao Lv. 2024. HyBench: A New Benchmark for HTAP Databases. *Proc. VLDB Endow.* 17, 5 (2024), 939–951. https://www.vldb.org/pvldb/vol17/p939-zhang.pdf

[52] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. 2019. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 415–432. https://doi.org/10.1145/3299869.3300085