AJOSC: Adaptive Join Order Selection for Continuous Queries

XINYI YE, Peking University, China
XIANGYANG GOU*, The University of New South Wales, Australia
LEI ZOU, Peking University, China
WENJIE ZHANG, The University of New South Wales, Australia

Multi-way join, which refers to the join operation among multiple tables, is widely used in database systems. With the development of the Internet and social networks, a new variant of the multi-way join query has emerged, requiring continuous monitoring of the query results as the database is updated. This variant is called continuous multi-way join. The join order of continuous multi-way join significantly impacts the operation's cost. However, existing methods for continuous multi-way join order selection are heuristic, which may fail to select the most efficient orders. On the other hand, the high-cost order computation will become a system bottleneck if we directly transfer join order selection algorithms for static multi-way join to the dynamic setting. In this paper, we propose a new *A*daptive *J*oin *O*rder *S*election algorithm for the *C*ontinuous multi-way join queries named AJOSC. It uses dynamic programming to find the optimal join order with a new cost model specifically designed for continuous multi-way join. We further propose a lower-bound-based incremental re-optimization algorithm to restrict the search space and recompute the join order with low cost when data distribution changes. Experimental results show that AJOSC is up to two orders of magnitude faster than the state-of-the-art methods.

CCS Concepts: • **Information systems → Query optimization**.

Additional Key Words and Phrases: Continuous Queries; Join Order Selection; Adaptive Query Optimization; Performance Optimization

ACM Reference Format:
Xinyi Ye, Xiangyang Gou, Lei Zou, and Wenjie Zhang. 2025. AJOSC: Adaptive Join Order Selection for Continuous Queries. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 126 (June 2025), 27 pages. https://doi.org/10.1145/3725263

1 Introduction

Multi-way join combines tuples in multiple tables based on the join predicates. It is widely used in database systems [9, 30, 33]. With the development of the Internet and social networks, a new variant of the multi-way join query has emerged, called continuous multi-way join. It requires continuous monitoring of the query results as the database is updated. Continuous multi-way join queries are widely used in applications such as sales management [44] and fraud detection [20, 33]. For example, the newly added results of some cyclic joins signal recent fraudulent activities in e-commerce databases [33].

*Corresponding author

Authors' Contact Information: Xinyi Ye, Peking University, Beijing, China, yexinyi@pku.edu.cn; Xiangyang Gou, The University of New South Wales, Sydney, Australia, xiangyang.gou@unsw.edu.au; Lei Zou, Peking University, Beijing, China, zoulei@pku.edu.cn; Wenjie Zhang, The University of New South Wales, Sydney, Australia, wenjie.zhang@unsw.edu.au.

ACM 2836-6573/2025/6-ART126
https://doi.org/10.1145/3725263

The update stream in a continuous multi-way join is composed of a sequence of tuple insertions and deletions. A query Q is previously registered in the database system, and for each insertion (or deletion), we need to find and report the added (or deleted) results in the result set of Q. These results must contain the updated tuple t. Thus, these results are derived from joining the updated table containing only tuple t with other tables. For example, in Figure 1, the query Q joins table T_R, T_S, T_W, T_X, T_Y and T_Z in database D. When tuple t_S^0 is added to table T_S, $\Delta Q(D)$ is the join results of table $T_R, \Delta T_S, T_W, T_X, T_Y$ and T_Z where table ΔT_S only contains tuple t_S^0.

Join order is the order in which tables are joined during the execution of a multi-way join. Join order influences the number of intermediate results generated and is therefore important to query performance. In continuous multi-way join, the join always starts at the updated table, since this table contains only the updated tuple and all updated results contain this tuple. But the other tables still need to be carefully ordered to optimize query efficiency. Thus, for every table, we select and store the join order starting from it. Whenever a table is modified by tuple insertion or deletion, we follow the stored join order that starts from it to compute the updated results.

Multiple join order selection algorithms in static settings have been proposed, including greedy [22] and dynamic programming methods [37]. However, there are few previous works in the dynamic settings. The existing algorithms for join order selection of continuous multi-way join queries are based on heuristics [28], which may select suboptimal orders. On the other hand, if we transform the dynamic programming algorithms in static settings [37] to the dynamic settings, we can select optimal join orders, but high selection costs will be introduced.

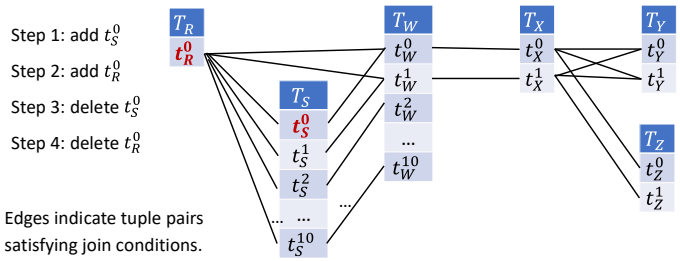

Fig. 1. An example of a database D where a query Q that joins table T_R, T_S, T_W, T_X, T_Y and T_Z is executed.

The dynamic programming algorithm in static settings (abbr. StaticDP) chooses the optimal order $T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n$ of a query Q in descending order of i. When deciding T_i, the tables $T_n, T_{n-1}, ..., T_{i+1}$ have been chosen. The remaining unordered tables and the join conditions among them form a subquery P. T_i is chosen to minimize the estimated computation cost of P, where P's result is computed by first joining $(P - \{T\})$ using the optimal order, and then join $(P - \{T\})$'s result with table T.

However, two challenges will arise if we apply this method to tackle continuous multi-way join queries in the dynamic settings, which limits its efficiency.

Challenge 1: Its cost model brings large order selection overhead in the dynamic settings, since multiple costs need to be estimated for each subquery. Note that we need to estimate the computation costs of a series of subqueries in the order selection process of StaticDP. In the dynamic settings, when the updated table changes, the computation cost of a subquery P also changes. This occurs for two reasons. First, the query execution must start from the updated table in the dynamic settings, thus the updated table influences the join order. Second, only the updated tuple is included in the updated table when executing the query, thus the updated table influences

the cardinality distribution. Hence, multiple computation costs must be calculated for the same subquery, increasing the order selection overhead.

Example 1.1. Consider step 2 and 3 depicted in Figure 1. In this figure, an edge between a pair of tuples indicates that the pair of tuples satisfies the join condition between the tables. For the subquery P that joins table T_R and T_S, when the updated table is T_R (step 2), the join order is $T_R \rightarrow T_S$. Since the tuple t_R^0 is added, the table $\Delta T_R = \{t_R^0\}$. In this case, we join $\{t_R^0\}$ with all 11 tuples in T_S. In contrast, when the updated table is T_S (step 3), the order of this subquery is $T_S \rightarrow T_R$. Since the tuple t_S^0 is deleted, the table $\Delta T_S = \{t_S^0\}$. In this case, we join $\{t_S^0\}$ with 1 tuple t_R^0 in T_R. This shows that the computation cost of the subquery P when the updated table is T_S (step 3) is much lower than when the updated table is T_R (step 2). Thus, two computation costs are needed for the subquery P in the dynamic settings.

Challenge 2: dynamic change of data distribution necessitates order recomputation. With tuples inserted or deleted, the data distribution might change, so the previously optimal join order might become suboptimal, causing performance degradation. As a result, we need to recompute the optimal join orders when the data distribution changes, which significantly increases the order selection overhead.

In this paper, we propose a novel adaptive join order selection algorithm for continuous multi-way joins (**AJOSC**), to identify high-quality join orders with low overhead in the dynamic settings. We propose the following techniques that address the aforementioned challenges and distinguish AJOSC from prior arts.

Firstly, to tackle challenge 1, we introduce a new cost model, the ***Look-Ahead Cost*** (abbr. LA cost) which enables cost estimations to be shared across updated tables for the same subquery. The LA cost of a subquery P is defined as the cost to use a single instance of P to compute the corresponding results of Q that contains this instance, where an instance is one row in P's results. Different from the computation cost of P used in StaticDP, the LA cost is not influenced by the updated table. Therefore, each subquery only needs one LA cost. Furthermore, we propose an order selection algorithm that is based on LA cost and searches the optimal join orders with the aid of a cost dependency graph (abbr. CDG). We can prove that the time complexity of our order selection algorithm is n times smaller than StaticDP, where n is the number of tables in the multi-way join.

Secondly, to address challenge 2, we propose an incremental reordering algorithm to update the join orders when data distribution changes significantly. We keep monitoring statistics about data distribution in the database. When a significant change is observed, our incremental algorithm only carries out recomputation in a small area of the CDG which is influenced by the changes. This reduces recomputation costs. Moreover, we propose a lower-bound-based method. It delays the update of node values in CDG when they will not affect the optimal order, and further reduces the recomputation cost.

Thirdly, we design a set of mechanisms to decide when to trigger the join reordering. A naive approach is to trigger reordering when a statistic changes beyond a predefined threshold. But this approach can lead to unnecessary reorderings when encountering outliers in statistics, which brings a large overhead. The reasons are as follows. We monitor statistics by maintaining a weighted average of collected values in query execution, where recent values are assigned higher weights to capture the most recent trend. This may cause fluctuations in the statistics if outliers are collected recently, which triggers unnecessary reorderings. To fix this, we propose the reordering delay mechanism that delays reordering unless changes in statistics are sustained, thus mitigating the effects of outliers.

To sum up, our main contributions are as follows:

- We propose a new cost model named LA cost suitable for join order selection in continuous multi-way join. Based on the LA cost, we propose a cost dependency graph-based dynamic programming algorithm to select optimal join orders with low cost.
- We propose an incremental order recomputation method which decreases the cost of order recomputation when the data distribution changes.
- We propose a reordering delay mechanism to decide the timing of updating the join orders which avoids unnecessary reorderings.
- We conduct extensive experiments to evaluate the performance of AJOSC. AJOSC accelerates the queries by up to two orders of magnitude compared to existing algorithms.

2 Preliminaries

*Definition 2.1 (**Multi-way Join Query**).* A multi-way join query Q can be formulated as $\{V, E\}$. V is a multiset where its unique elements form a subset of the tables in the database D. The set E consists of join conditions among these tables, in the form of $T_1.a_1 = T_2.a_2$, where a_1 and a_2 denote attributes in the table. The result set of a query Q in the database D, denoted $Q(D)$, is a set of tuple concatenations. Each concatenation in $Q(D)$ contains exactly one tuple from each table in V, and the tuples satisfy all the join conditions in E. We call such a concatenation an ***instance*** of the query Q, denoted as $inst(Q)$.

Note that V is a multiset since a query can include multiple copies of the same table. For example, an instance of the query with $V = \{T_R, T_{S_1}, T_{S_2}\}$ and $E = \{T_R.a_1 = T_{S_1}.a_1, T_R.a_2 = T_{S_2}.a_1\}$ may appear in the form (t_R^0, t_S^0, t_S^1). Here, T_{S_1} and T_{S_2} are copies of table T_S, while t_S^0 and t_S^1 are tuples in table T_S.

In this paper, we represent a query as $\{T_1, T_2, ..., T_n\}$ where $T_1, T_2, ..., T_n$ are all the tables in V. And we represent an instance as $(t_1, t_2, ..., t_n)$ where $t_i \in T_i$.

Example 2.2. This is a multi-way join query where we join the tables $T_R, T_S, T_W, T_X, T_Y, T_Z$ together:

```
SELECT * FROM T_R, T_S, T_W, T_X, T_Y, T_Z WHERE
T_R.s_id = T_S.id AND T_W.r_id = T_R.id
AND T_S.w_id = T_W.id AND T_W.year = T_X.year
AND T_Y.x_id = T_X.id AND T_Z.id = T_X.z_id
```

We represent this query as $\{T_R, T_S, T_W, T_X, T_Y, T_Z\}$. For the database D shown in Figure 1, the concatenation $(t_R^0, t_S^0, t_W^0, t_X^0, t_Y^0, t_Z^0)$ is an instance of the query $\{T_R, T_S, T_W, T_X, T_Y, T_Z\}$.

In this paper, we focus on queries with equi-join conditions, but our algorithm can also be extended to other join conditions. As we focus on join order selection, we do not consider other operations such as select, project, and aggregate in this paper.

*Definition 2.3 (**Update Stream**).* An update stream is a constant flow of update operations that describes the update process of a database. An update operation is a triple (T, t, op) where $op = +(or -)$, which means adding (or deleting) a tuple t from a table T.

*Definition 2.4 (**Continuous Multi-way Join**).* Given a database and an update stream, a continuous multi-way join is a previously registered and continuously monitored multi-way join query. For each update d in the update stream, we need to compute $\Delta Q(D, d) = Q(D + d.t) - Q(D)$ (if $d.op = +$) or $Q(D) - Q(D - d.t)$ (if $d.op = -$), i.e., to compute the change of the result set induced by the update.

Note that $\Delta Q(D, (T, t, op))) = Q(D')$, where D' is the same as D except that the updated table T is replaced by $\Delta T = \{t\}$.

Example 2.5. For the query in Example 2.2, when the update operation $d = (T_R, t_R^0, +)$ is performed on the database, $\Delta Q(D, d)$ are the results of the query

```
SELECT * FROM ΔT_R, T_S, T_W, T_X, T_Y, T_Z WHERE
ΔT_R.s_id = T_S.id AND T_W.r_id = ΔT_R.id
AND T_S.w_id = T_W.id AND T_W.year = T_X.year
AND T_Y.x_id = T_X.id AND T_Z.id = T_X.z_id
```

where the table ΔT_R contains only tuple t_R^0.

*Definition 2.6 (**Join Graph**).* A join graph of a query Q is a labeled, undirected graph. Each vertex in the join graph uniquely represents a table in $Q.V$, while each edge in the join graph uniquely represents a join condition in $Q.E$. The endpoints of each edge represent the two tables involved in the join condition.

For example, the join graph in Figure 2 represents the SQL query in Example 2.2.

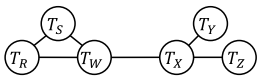

Fig. 2. An example join graph

*Definition 2.7 (**Subquery**).* A subquery P of a multi-way join query $Q = \{V, E\}$ can be formulated as another multi-way join query $\{V', E'\}$. Here, $V' \subseteq V$, and E' consists of the join conditions in E that involve only tables in V'. We use the notation $inst(P) \subseteq inst(Q)$ to indicate that for all $T_i \in V'$, $inst(P).t_i = inst(Q).t_i$.

Example 2.8. Here is the subquery $\{T_R, T_S, T_W, T_X\}$ of the query $\{T_R, T_S, T_W, T_X, T_Y, T_Z\}$ in Example 2.2.

```
SELECT * FROM T_R, T_S, T_W, T_X WHERE
T_R.s_id = T_S.id AND T_W.r_id = T_R.id
AND T_S.w_id = T_W.id AND T_W.year = T_X.year
```

The instance $(t_R^0, t_S^0, t_W^0, t_X^0) \subseteq (t_R^0, t_S^0, t_W^0, t_X^0, t_Y^0, t_Z^0)$ because tuples $t_R^0, t_S^0, t_W^0, t_X^0$ are the same in the two instances.

*Definition 2.9 (**Join Order**).* The **join order** in the context of a continuous multi-way join query under an update (T, t, op) refers to the sequence of subqueries, denoted $P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_n$, which plays a crucial role in the execution of the query. This sequence satisfies the following conditions:

(1) $P_1 = \{\Delta T\}$, where T is the updated table and $\Delta T = \{t\}$.
(2) $P_n = Q$.
(3) For any i, P_{i+1} contains one more table compared to P_i.

The computation of a continuous multi-way join query under update (T, t, op) involves executing multiple subqueries in a join order $P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_n$.

Since P_{i+1} contains one more table compared to P_i, the join order can also be represented as a sequence of tables $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n$, where:

(1) $T_1 = \Delta T$ denotes the updated table;
(2) $T_i = P_i.V - P_{i-1}.V$ denotes the i-th table to join.

In subsequent sections, these two representations of join order will be used interchangeably.

Note that the join graph of every P_i must be connected. If a subquery's join graph is disconnected, its results become a Cartesian product of the results of its connected components, leading to numerous subquery results and high computation costs. Therefore, we define a subquery as ***valid*** if its join graph is connected. Any mention of a subquery hereafter will imply a valid subquery by default.

Problem Statement. *Join order selection for continuous multi-way join queries* determines the join orders that optimize the execution efficiency of these queries.

*Definition 2.10 (**Partial Join Order**).* The **partial join order** starting at a subquery P refers to a segment of a join order sequence that starts from P and ends with the final query Q, denoted $P_j \rightarrow P_{j+1} \rightarrow \ldots \rightarrow P_n$, where $P_j = P$ and $P_n = Q$. The partial join order can also be represented as a sequence of tables $T_{j+1} \rightarrow T_{j+2} \rightarrow \ldots \rightarrow T_n$, where $T_i = P_i.V - P_{i-1}.V$ $(j + 1 \leqslant i \leqslant n)$. In subsequent sections, these two representations of the partial join order will be used interchangeably.

We define 2 notations about the partial join order here.

(1) ϕ_P denotes a partial join order starting at P.
(2) $N(P)$ denotes table T_{j+1} in the *optimal* partial order $T_{j+1} \rightarrow T_{j+2} \rightarrow \ldots \rightarrow T_n$, i.e. the next table to join after P in the optimal partial order.

Table 1. The notation table

Notation	Description
t_T	a tuple in table T
$Match(T, t_{T'})$	Set of tuples t_T that satisfy the join conditions between T' and T with a given tuple $t_{T'}$
$inst(Q)$ / $inst(P)$	Instance of the query Q/ subquery P
$inst(P) \subseteq inst(Q)$	$inst(P)$ has consistent tuple with $inst(Q)$ for each table in P
ϕ_P	a partial join order starting at the subquery P
$CandN(P)$	Candidate tables for the next table to join after P, where tables inside P have been ordered
$CandL(P)$	Candidate tables for the last table to join in P, where tables outside P have been ordered
$N(P)$	The next table to join after P in the optimal partial order

*Definition 2.11 (**Candidate Table**).* When selecting a join order $T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n$, the set of candidate tables for T_i includes all tables that can potentially be ordered in the i-th position $(i \geq 2)$ without inducing invalid subqueries.

In this paper, we select the join order either with a back-to-front manner (in StaticDP) or with a front-to-back manner (in AJOSC). In the former case, when we select T_i, tables $T_{i+1} \sim T_n$ have been chosen. The unordered tables form a subquery P, and T_i is the last table to join in P. We use $CandL(P)$ to denote the candidate tables of T_i. A table is selected into $CandL(P)$ only if the remaining tables in P are connected after removing it. In the latter case, when we select T_i, tables $T_1 \sim T_{i-1}$ have been chosen. The ordered tables form a subquery P, and T_i is the next table to join after P. We use $CandN(P)$ to denote the candidate tables of T_i. A table is selected into $CandN(P)$ only if it is connected to at least one table in P.

Table 1 presents the notations in this paper.

3 StaticDP and Its Limitations

In this section, we introduce the dynamic programming algorithm in static settings (abbr. StaticDP) [37] and then discuss its limitation when transferred to the dynamic setting. It explains our motivation to design the new techniques in the following sections.

The dynamic programming algorithm in static settings computes the optimal order by recursively deciding T_i with a decreasing order of i, i.e. from T_n to T_1. To decide which table should be T_i, the algorithm computes the estimated costs for each candidate table and then selects the one with the lowest estimated cost. Specifically, T_i should be decided by

$$T_i = Argmin_{T \in CandL(P)} \left(SCost(P - \{T\}) + SJC(P - \{T\}, T)\right)$$

where

(1) $P = \{T_1, T_2, ..., T_i\}$;
(2) $SCost(\cdot)$ is the cost of computing results of a subquery using the optimal order;
(3) $SJC(\cdot)$ is the cost to join the result of a subquery with a table.

The join cost $SJC(\cdot)$ is estimated by cardinality estimation, and $SCost(P)$ is recursively computed by

$$\begin{cases} SCost(P) = \min\limits_{T \in CandL(P)} \left(SCost(P - \{T\}) + SJC(P - \{T\}, T)\right) \\ SCost(\{T\}) = 0, \forall T \in Q \end{cases}$$

Note that during the process of selecting an order, $SCost(P)$ of every valid subquery P needs to be calculated.

When transferred to continuous multi-way join, there are 2 limitations that decrease the performance of StaticDP.

Firstly, its cost model brings large order selection overhead in the dynamic setting. In continuous multi-way join, we need to start query execution from the updated table, thus different updated tables result in different optimal join orders for the same subquery. Moreover, only one updated tuple is included in the updated table in query execution, thus different updated tables also result in different cardinality distributions. As a result, the same subquery P will get different $SCost(P)$ when the updated table is different. Thus, we need to estimate multiple costs for each subquery, which brings large order selection overhead.

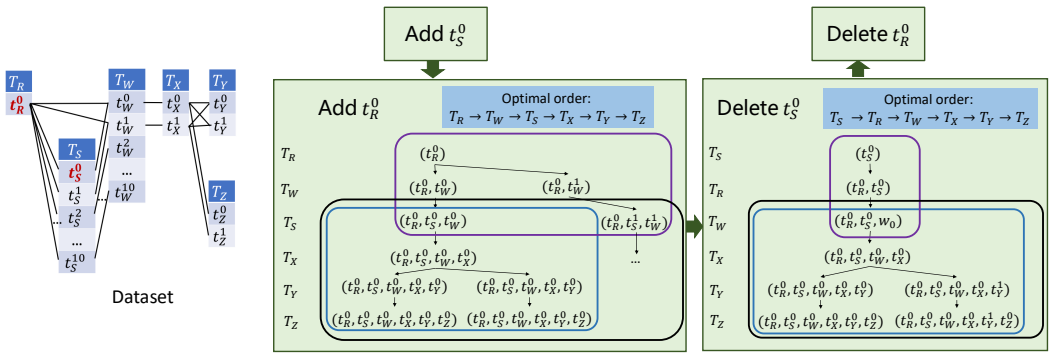

Fig. 3. The search process when the updated table is T_R and T_S respectively.

Example 3.1. Figure 3 shows a database and the search process following the optimal order when the updated tables are T_R and T_S, respectively. The process of computing the change of the subquery $\{T_R, T_S, T_W\}$ is shown in purple rounded rectangles. For the subquery $\{T_R, T_S, T_W\}$, 5 instances appear in the search process starting from table T_R, while only 3 instances appear in the search process starting from table T_S. Hence, $SCost(\{\Delta T_R, T_S, T_W\})$ and $SCost(\{T_R, \Delta T_S, T_W\})$ are different. Similarly, $SCost(\{T_R, T_S, \Delta T_W\})$ is also a different cost. Therefore, we need to compute 3 costs for subquery $\{T_R, T_S, T_W\}$ in the dynamic settings, but only 1 in static settings. Thus, the overhead is larger in the dynamic settings.

This leads to a higher time complexity of StaticDP in the dynamic settings than static ones.

Theorem 3.2. *The time complexity of StaticDP is $O(n2^n)$ in static settings, but $O(n^2 2^n)$ in the dynamic settings ($n = |Q.V|$).*

Proof. In StaticDP in static settings, the order selection time is dominated by the time to compute $SCost(P)$ for all subqueries P. For a query with $|Q.V| = n$, there are $O(2^n)$ possible subqueries. To compute $SCost(P)$ for any subquery P, we need to evaluate $SJC(P - \{T\}, T)$ for each table T in $CandL(P)$. This process has an overhead of $O(n)$ for a single subquery. Therefore, when we consider both the number of subqueries and the overhead to compute their costs, the resulting time complexity is $O(n2^n)$.

In the dynamic settings, the overhead to compute one cost is n and there are $O(2^n)$ possible subqueries, as is the case in static settings. However, in the dynamic settings, for every subquery P, we need to estimate $|P.V|$ costs, and $|P.V| = O(n)$. Thus, the time complexity in the dynamic settings is $O(n^2 2^n)$.

□

Secondly, StaticDP lacks an efficient order recomputation method. When the data distribution changes, we need to update the stored join orders to keep them optimal. However, as there is no incremental order recomputation method in StaticDP, we have to compute all the orders from scratch, which further increases the overhead.

To address the above two limitations, we propose three key techniques in AJOSC: First, we propose a new cost model, LA cost, specialized for continuous multi-way joins, which will be introduced in Section 4. Second, we propose an incremental order recomputation method in Section 5. Third, we propose a set of mechanisms to decide when to trigger the order recomputation in Section 6.

4 Order Computation with LA Cost

4.1 LA Cost

In this subsection, we introduce the LA cost, which facilitates sharing cost estimation across different updated tables for the same subquery. Because a single LA cost suffices for each subquery, the overhead of AJOSC is much smaller than that of StaticDP in the dynamic settings.

4.1.1 *Definition of LA Cost*

For every subquery P, we compute its LA cost $LA(P)$. $LA(P)$ is the expected cost to use an instance of P to compute the results of Q that contains this instance following the optimal order. There are 2 differences between $LA(P)$ and the $SCost(P)$ of StaticDP algorithm:

- $LA(P)$ is the cost to compute the results of Q using results of P, while $SCost(P)$ is the cost to compute results of P according to the updated tuple. As stated above, the execution of a continuous multi-way join query starts from the updated table. Because all subqueries P that we explore contain the updated table, the cost of joining P will be influenced by the

updated table. On the other hand, as the updated table is not involved in the process of using P's result to compute Q's results, the optimal order and the computation cost of this process will not be influenced by it. For example, in Figure 3, the search processes in the two blue rounded rectangles are the same, but those in the purple rounded rectangles are different.

- $LA(P)$ is defined for only an instance of P, while $SCost(P)$ is defined for all instances of P related to the updated tuple. When the updated tables differ, the number of calculated instances of P might differ, but $LA(P)$ is irrelevant to it. For example, in Figure 3, there are two instances of $\{T_R, T_S, T_W\}$ when the updated table is T_R and one instance when the updated table is T_S. But $LA(\{T_R, T_S, T_W\})$ is only related to the search process in the blue rounded triangle, thus the same.

4.1.2 *Computation of LA Cost*

In this subsection, we discuss how to compute the LA cost. The LA cost $LA(P)$ is the minimum expected cost to use one instance $inst(P)$ to compute $\{inst(Q) \mid inst(P) \subseteq inst(Q)\}$ among all possible partial join orders. $LA(P)$ can be computed as

$$LA(P) = min_{\phi_P} cost(\phi_P)$$

where $cost(\phi_P)$ is the expected cost to compute $\{inst(Q) \mid inst(P) \subseteq inst(Q)\}$ given an instance $inst(P)$ using the partial join order ϕ_P.

The $cost(\phi_P)$ can be divided into **two parts**. **The first part** is the expected cost to join $inst(P)$ with table T, where T is the first table in ϕ_P, i.e. the next table to join. Specifically, this is the expected cost to compute the result set $\{inst(P^+) \mid inst(P) \subseteq inst(P^+)\}$ given $inst(P)$, where $P^+ = P \cup \{T\}$. To compute the result set, we need to find all tuples $t_T \in T$ that meet this criterion:

Criterion 4.1. *For all table $T' \in P$ where $(T', T) \in E$, $t_{T'}$ and t_T satisfy the join conditions between T' and T, where $t_{T'}$ is the tuple from table T' in $inst(P)$.*

To identify these tuples, we examine every table $T' \in P$ where $(T', T) \in E$. For every such table T', we compute the candidate tuple set $Match(T, t_{T'})$ which contains all tuples $t_T \in T$ that satisfy the join conditions between T' and T with $t_{T'}$. By computing the intersection $\cap_{T' \in P, (T', T) \in E} Match(T, t_{T'})$, we get the tuples satisfying criterion 4.1. Therefore, by summing the expected sizes of $Match(T, t_{T'})$ for all $T' \in P$ such that $(T', T) \in E$, we can efficiently approximate the cost to join $inst(P)$ with table T. To closely estimate the expected size of $Match(T, t_{T'})$, we employ the statistic $deg(T', T)$ to represent it and iteratively refine its value during the execution of the query. Specifically, we initiate $deg(T', T)$ by 0. Every time we get a set $Match(T, t_{T'})$ in query execution, $deg(T', T)$ is updated by

$$deg(T', T) \leftarrow deg(T', T) * (1 - \theta) + |Match(T, t_{T'})| * \theta \tag{1}$$

where θ is a predefined parameter. In this way, we obtain a weighted average for $deg(T', T)$, where the weight of the more recent data is larger. Thus, we can catch the distribution of the recent data and compute the suitable join order for the current data distribution. Note that this statistical maintenance method is independent of the key contributions in this paper and can be replaced by other statistical maintenance methods. For example, ADWIN [8] and SDDM [34] can detect distribution change of a stream. The average of all $|Match(T, t_{T'})|$ after the last distribution change can represent the current distribution and can be used as $deg(T', T)$.

In summary, the first part of $cost(\phi_P)$ can be calculated by $\sum_{T' \in P, (T', T) \in E} deg(T', T)$.

The second part is the expected cost to compute the set $\{inst(Q)\}$ from the set $\{inst(P^+)\}$, where $P^+ = P \cup \{T\}$ and the instances in the two sets satisfy $inst(P) \subseteq inst(P^+)$ and $inst(P) \subseteq inst(Q)$. This cost can be calculated by $|\{inst(P^+)\}| * cost(\phi_{P^+})$, since $cost(\phi_{P^+})$ is the expected cost for each instance $inst(P^+)$ to compute the final result further along the order ϕ_{P^+}. To estimate $|\{inst(P^+)\}|$

which is the size of the intersection $\cap_{T' \in P,(T',T)\in E}|Match(T, t_{T'})|$, we use $|\{inst(P^+)\}| = w(P,T) * \sum_{T' \in P,(T',T)\in E} deg(T', T)$, where $w(P,T)$ is the expected value of $\frac{|\cap_{T' \in P,(T',T)\in E} Match(T,t_{T'})|}{\sum_{T' \in P,(T',T)\in E}|Match(T,t_{T'})|}$. Similarly to $deg(\cdot)$, $w(\cdot)$ is updated by

$$w(P,T) \leftarrow w(P,T) * (1-\delta) + \frac{|\cap_{T' \in P,(T',T)\in E} Match(T, t_{T'})|}{\sum_{T' \in P,(T',T)\in E}|Match(T, t_{T'})|} * \delta \tag{2}$$

where δ is a predefined parameter.

Note that we need to update the statistics $w(\cdot)$ and $deg(\cdot)$ during query execution. However, if we always use the optimal orders to execute the query, some tables T may never be selected as the next table to join for some subqueries P, and the corresponding statistics $w(P,T)$ will not be updated. This makes some LA costs inaccurate, and the selected orders may get stuck in local optima. To solve this problem, we use the ϵ-greedy method. We use the optimal orders to execute the query with a probability of $1-\epsilon$ and use random orders with a probability of ϵ, where ϵ is a predefined parameter. With this strategy, every statistic value has a chance to be updated.

To summarize, the second part of $cost(\phi_P)$ can be calculated by $cost(\phi_{P^+}) * w(P,T) * \sum_{T' \in P,(T',T)\in E} deg(T', T)$.

By **summing up the two parts**, $cost(\phi_P)$ can be computed recursively using

$$cost(\phi_P) = (1 + w(P,T) * cost(\phi_{P^+})) * \sum_{T' \in P,(T',T)\in E} deg(T', T).$$

The recursion terminates with $cost(\phi_Q) = 0$.

Thus, $LA(P)$ can be recursively computed by

$$\begin{aligned} LA(P) &= min_{\phi_P} cost(\phi_P) \\ &= min_{T,\phi_{P^+}} \left((1 + w(P,T) * cost(\phi_{P^+})) * \sum_{T' \in P,(T',T)\in E} deg(T', T) \right) \\ &= min_{T \in CandN(P)} \\ &\quad \left((1 + w(P,T) * LA(P \cup \{T\})) * \sum_{T' \in P,(T',T)\in E} deg(T', T) \right) \end{aligned}$$

The recursion ends with $LA(Q) = 0$. To enhance readability, we add a term $LA(P|T)$ to represent the smallest $cost(\phi_P)$ among all possible ϕ_P whose first table is T:

$$LA(P|T) = (1 + w(P,T) * LA(P \cup \{T\})) * \sum_{T' \in P,(T',T)\in E} deg(T', T). \tag{3}$$

Thus,

$$LA(P) = min_{T \in CandN(P)} LA(P|T). \tag{4}$$

Recall that the first table in the optimal partial order starting at P is denoted by $N(P)$. It can be computed by

$$N(P) = argmin_{T \in CandN(P)} LA(P|T). \tag{5}$$

4.2 Order Computation

4.2.1 *The timing of order computation*

When a continuous multiway join query is registered in the database, the query is first executed using random orders. During query execution, we gather statistics about the data distribution in the database. After handling a predefined number of updates and collecting enough information,

we have collected enough statistics. At this time, we compute the optimal orders that start from each table from scratch. After this, we continue to execute the query and keep monitoring the statistics, and use an incremental algorithm to update the join orders when the distribution changes significantly.

4.2.2 *The method of order computation*

In this section, we introduce how to compute the optimal orders from scratch. The method to incrementally update these orders will be introduced in Section 5. In order to maximize the computation sharing of LA costs and support the order selection, we propose a cost dependency graph (abbr. CDG). It indicates the dependency relationships among the LA costs of all subqueries. And we can find optimal orders by traversing the CDG.

The CDG is a directed acyclic graph. A node in the CDG represents a subquery P. We call $LA(P)$ and $N(P)$ the values of the node P, and store them in the node. An edge illustrates the dependency between the values of two subqueries, indicating that the values of the source node depend on that of the destination node. According to Equation 3, 4 and 5, an edge in the CDG must start at a node P and end at a node $P \cup \{T\}$. On each edge from P to $P \cup \{T\}$, we annotate $LA(P|T)$, the related statistic $w(P, T)$, and the related statistics $deg(T', T)$ for all $T' \in P$ such that $(T', T) \in E$. The statistics and the LA cost of the destination node help calculate $LA(P|T)$, and $LA(P|T)$ help calculate the values of the source node.

We define the following concepts in CDG:

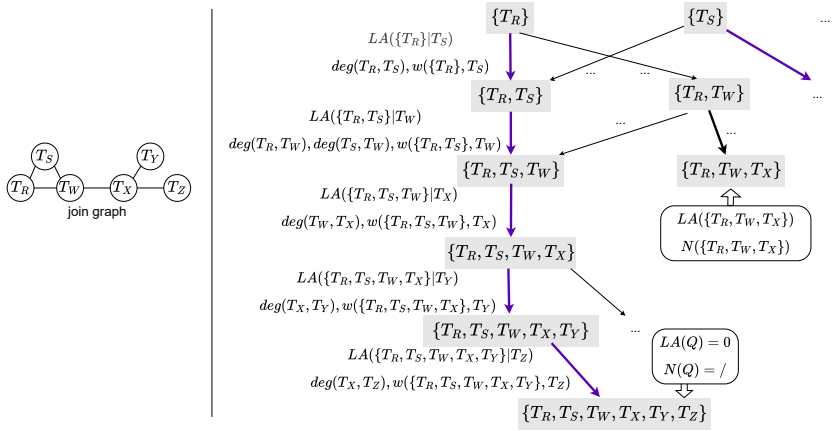

Fig. 4. The CDG structure.

(1) A **leaf node** is a node without children. According to Equation 3, 4 and 5, Q is the only leaf node in a CDG. Note that $LA(Q) = 0$ and $N(Q)$ has no meaning.
(2) A **root node** is a node without parents. For every table $T \in Q$, $\{T\}$ is a root node, thus there are $|Q.V|$ root nodes in a CDG.
(3) An **order path** is a path from a root node to the leaf node. This path should resemble $\{T_1\} \rightarrow \{T_1, T_2\} \rightarrow \ldots \rightarrow \{T_1, T_2, \ldots, T_n\}$, illustrating the join order.
(4) An **optimal order path** is an order path representing an optimal order. The optimal order $\{T_1\} \rightarrow \{T_1, T_2\} \rightarrow \ldots \rightarrow \{T_1, T_2, \ldots, T_n\}$ satisfies $T_{i+1} = N(\{T_1, T_2, ..., T_i\})$ for all i.

Example 4.1. The example of a CDG and its corresponding join graph is shown in Figure 4. The edge from $\{T_R\}$ to $\{T_R, T_S\}$ in Figure 4 indicates that the values of $\{T_R\}$ depend on $LA(\{T_R, T_S\})$.

We use a bold edge from P to $P \cup \{T\}$ to show $N(P) = T$. We use edges in purple to indicate the optimal order paths: the leftmost purple path represents the optimal join order $T_R \rightarrow T_S \rightarrow T_W \rightarrow T_X \rightarrow T_Y \rightarrow T_Z$. Note that all purple edges are bold, indicating that optimal orders are calculated by computing $N(\cdot)$.

We can perform a bottom-up breadth-first search (BFS) on the CDG to compute the values of all nodes, where $LA(P)$ is computed with Equation 3 and 4, and $N(P)$ with Equation 5. By employing BFS, we ensure that before calculating the values of a node, the values of all its children are already computed. This allows for the direct computation of the node's values via Equation 3, 4 and 5.

For each updated table $T \in Q$, we need to compute an optimal order $T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n$ that starts from it, where $T_1 = \Delta T$. We can obtain this order by a walk in the CDG guided by $N(\cdot)$. Specifically, we start from the root node $P_0 = \{T\}$, and end at the leaf node Q. For each node P_i we visit, the next node is $P_{i+1} = P_i \cup \{N(P_i)\}$. The nodes we visit form the optimal join order.

4.3 Complexity Analysis

Theorem 4.2. *The time complexity of using AJOSC to get optimal orders from scratch is $O(n2^n)$ ($n = |Q.V|$).*

Proof. The order selection time of using AJOSC to compute optimal orders from scratch is dominated by the time to compute the LA costs for all subqueries P. To compute the LA cost for a subquery P, we need to evaluate $LA(P|T)$ for each T in $CandN(P)$, which has an overhead of $O(n)$. For a query with $|Q.V| = n$, there are $O(2^n)$ possible subqueries. Thus, the time complexity is $O(n2^n)$. □

Theorem 4.2 shows that the complexity of using AJOSC to get optimal orders from scratch is a lot smaller than that of StaticDP ($O(n^2 2^n)$) in the dynamic settings.

5 Incremental Order Recomputation

In Section 4.2, we discussed how to compute the optimal order from scratch. In this section, we discuss how to recompute the optimal orders when data distribution changes significantly, where the criteria for a significant change will be discussed in Section 6. One naive approach would be to recompute the optimal orders from scratch. This requires visiting and updating all nodes in the CDG. Since there are $O(2^n)$ nodes and $O(n2^n)$ edges in the CDG ($n = |Q.V|$), visiting all of them will bring an unacceptable overhead when n is large. To address this issue, we propose an incremental method to recompute the order. This method updates only the nodes affected by notable changes in data distribution. Moreover, we propose the LBR method, a **L**ower **B**ound based computation **R**eduction technique to further reduce the recomputation cost. For a simpler presentation, we first introduce the basic incremental recomputation algorithm in Section 5.1, and then discuss the LBR method in Section 5.2.

5.1 Basic Version

In this section, we introduce how to recompute the order incrementally when the data distribution changes. The pseudocode is shown in Algorithm 1.

When a statistic s changes significantly, we perform bottom-up breadth-first searchs (BFS) to update the values of all affected nodes (line 1-20 in Algorithm 1). Specifically, the BFSs start at the nodes whose values are calculated **directly** from the statistic s. These nodes are:

(1) The node P if $s = w(P, T)$.
(2) The nodes P that satisfy $T' \in P$ and $T \in CandN(P)$ if $s = deg(T', T)$.

Line 1-6 in Algorithm 1 shows the process to compute these nodes. For every node P_{start} whose values are calculated directly from s, we perform an individual BFS starting from P_{start}. During the bottom-up BFS (line 7-20 in Algorithm 1), we visit P_{start}'s ancestors, as their values are based on $LA(P_{start})$ and thus are calculated implicitly based on the changed statistic. We update the values of each node encountered during the BFS, thus all affected node values are updated according to Equation 3, 4 and 5. We prune a node from the BFS when its values remain unchanged after the update (line 17 of Algorithm 1).

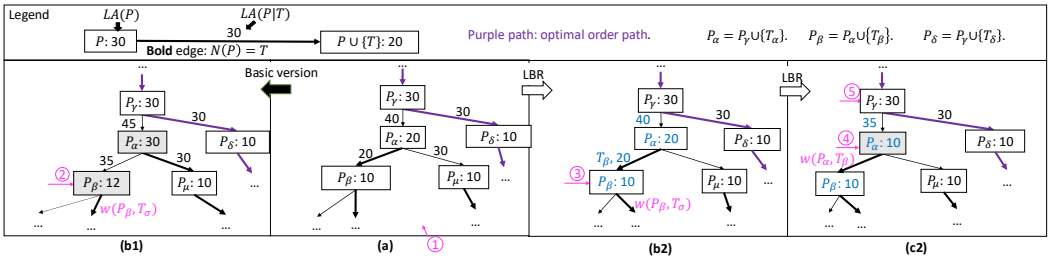

Fig. 5. An example of the basic version and the LBR method. Subfigure (a) show a CDG. Subfigure (b1) shows the CDG after executing the basic version, while (b2) and (c2) show the CDG after executing the LBR method. Note that Subfigure (b1) is on the left side of (a). Black values are accurate and blue values are *LOWERBOUND*. Changed nodes are highlighted in grey.

Example 5.1. In Figure 5(a), we illustrate a part of a CDG, where the LA costs are annotated on every node (see ①). In Figure 5(b1) which is on the left side of 5(a), we demonstrate the behavior of the basic version when $w(P_\beta, T_\sigma)$ changes from 10 to 15. A BFS is initiated from the node P_β (see ②). We then recalculate $LA(P_\beta)$ and $N(P_\beta)$, discovering that $N(P_\beta)$ remains unchanged but $LA(P_\beta)$ updates from 10 to 12. Then, we visit P_β's parent, P_α. Similarly, we recompute P_α's values and move to its parent, P_γ. Then, since P_γ's values do not change after the update, we prune it and the BFS stops.

After all affected values are updated by the BFS, we derive the optimal orders in the same way as discussed in Section 4.2 (line 21-29 of Algorithm 1).

This incremental algorithm only updates affected nodes, thus avoids traversing the entire CDG. We can further reduce the value recomputation by allowing the node values to be inaccurate when they will not affect the optimal order. See the following subsection for details.

5.2 The LBR Method

In this subsection, we discuss the LBR method that further reduces recomputation cost. The LBR method is based on the following observation:

THEOREM 5.2. *If a node P is not on any optimal order path in the CDG, the optimal orders will not change if $LA(P)$ increases.*

PROOF. When P is not on any optimal order path, every order path containing P corresponds to an order $\phi_{\{T\}}$ with $\phi^*_{\{T\}} \neq \phi_{\{T\}}$, where $\phi^*_{\{T\}}$ is the optimal order starting at T. Since $\phi^*_{\{T\}}$ is optimal, we have $cost(\phi_{\{T\}}) \geq cost(\phi^*_{\{T\}})$. Since $LA(P)$ increases, $cost(\phi_{\{T\}})$ increases, and $cost(\phi) \geq cost(\phi^*_{\{T\}})$ still holds. Thus, the optimal order $\phi^*_{\{T\}}$ is not affected. □

Example 5.3. In Figure 5(b1), when $w(P_\beta, T_\sigma)$ increases from 10 to 15, if we do not update nodes P_β and P_α, the optimal order path will continue to be ... $\rightarrow P_\gamma \rightarrow P_\delta \rightarrow$ ... and is correct.

Algorithm 1: *IncrementalRecomputation*

Data: the significantly changed statistic s
Result: the join order after the recomputation

$D \leftarrow$ an empty set /* which contains all nodes whose values are calculated directly from s. */
/* update the CDG */
if $s = w(P, T)$ **then**
  $D \leftarrow D \cup \{P\}$
else
  /* $s = deg(T', T)$ */
  foreach P *that satisfy* $T' \in P$ *and* $T \in CandN(P)$ **do**
    $D \leftarrow D \cup \{P\}$
foreach $P_{start} \in D$ **do**
  $q \leftarrow$ an empty queue
  $q.push(P_{start})$
  while $!q.empty$ **do**
    $P \leftarrow q.pop()$
    $LA(P)_{old} \leftarrow LA(P)$
    foreach $T \in CandN(P)$ **do**
      $LA(P|T) \leftarrow (1 + w(P, T) * LA(P \cup \{T\})) * \sum_{T' \in P, (T', T) \in E} deg(T', T)$
    $LA(P) \leftarrow min_{T \in CandN(P)} LA(P|T)$
    $N(P) \leftarrow argmin_{T \in CandN(P)} LA(P|T)$
    if $LA(P) \neq LA(P)_{old}$ **then**
      if P *is not a root node* **then**
        foreach P*'s parent node* P_{parent} **do**
          $q.push(P_{parent})$
/* Compute optimal orders according to $N(P)$ */
$orders \leftarrow$ an empty dictionary
foreach $T \in Q$ **do**
  $P \leftarrow \{T\}$
  $order\_sequence \leftarrow T$
  while $P \neq Q$ **do**
    $order\_sequence \leftarrow (order\_sequence \rightarrow N(P))$
    $P \leftarrow P \cup \{N(P)\}$
  $orders[T] \leftarrow order\_sequence$
return $orders$

Therefore, if the LA cost of a node increases and the node is not on any optimal order path, we do not need to compute the node's accurate values, thus reducing recomputation.

In the LBR method, we add a tag to every node P in the CDG to record whether the node values are accurate or not. The tag is either $EXACT$ or $LOWERBOUND$. $EXACT$ indicates that the values are accurate. $LOWERBOUND$ indicates that the current $LA(P)$ and $N(P)$ are not accurate, and

that the accurate $LA(P)$ is no less than the currently stored $LA(P)$. In other words, the current $LA(P)$ is the lower bound of the actual $LA(P)$. When we determine that the change of a node's values will not affect the optimal orders, we can simply set its tag as $LOWERBOUND$ and do not update its values.

Next, we explain how to maintain the CDG and compute the optimal orders in the LBR method.

Initialization: when the continuous multiway join query begins, we compute the CDG from scratch using the method in Section 4.2, and set all tags as $EXACT$.

When a statistic changes significantly, similarly to the basic version, the update of the CDG and the join orders can be divided into two phases. The first phase is to perform bottom-up BFSs in the CDG to maintain the values and tags of the affected nodes. The second phase is to compute the optimal orders.

First Phase: CDG Maintenance. In the first phase, we use different strategies in the BFSs depending on whether the change is an increase or decrease.

Statistic increment: when a statistic s increases, we set the tag of every node visited in the bottom-up BFSs as $LOWERBOUND$, and do not update their values. As stated above, the increment of an LA cost may not affect the optimal orders. Thus, we delay the value updates to the order computation phase, where we determine whether these updates will affect the optimal orders, and apply updates only when necessary. We prune a node P from the BFS when either of the following two cases happens:

(1) When P's tag is already $LOWERBOUND$ before we visit it.
(2) When we travel to P from $(P \cup \{T\})$ and find that P's tag is $EXACT$ and $N(P) \neq T$.

In the second case, since $N(P) \neq T$, we know that $(P \cup \{T\})$ is not in the optimal partial order ϕ_P^* before the update. As the statistics increment must result in the increment of $LA(P \cup \{T\})$, $(P \cup \{T\})$ stays out of ϕ_P^* after the update. Thus ϕ_P^* does not change. Recall that $LA(P) = cost(\phi_P^*)$, and $N(P)$ is the first table in ϕ_P^*. Thus, $LA(P)$ and $N(P)$ is not changed. Hence, the node P can be pruned from the BFS.

Example 5.4. In Figure 5(b2), we illustrate the behavior of the LBR method when $w(P_\beta, T_\sigma)$ increases from 10 to 15. BFS starts at P_β (see ③), and $P_\beta.tag$ is set as $LOWERBOUND$ (values with tag $LOWERBOUND$ are marked in blue). Then, we visit its parent P_α. Since $N(P_\alpha) = T_\beta$, we do not prune P_α and the BFS continues. We assign $P_\alpha.tag$ to $LOWERBOUND$ and proceed to its parent P_γ. As $P_\gamma.tag = EXACT$ and $N(P_\gamma) \neq T_\alpha$, according to the second rule listed above, we can prune this BFS branch. Note that we avoid recomputing the values of P_γ and P_β using the LBR method, thus reducing the overhead.

Statistic decrement: when a statistic decreases, the BFSs to maintain the node values are the same as the basic version, except for tag maintenance. When we visit a node P during BFSs, we first compute $N(P)$ and $LA(P)$ according to the node values of its children. Then we set $P.tag$ as the tag of its child $P \cup \{N(P)\}$. Because $LA(P)$ are computed by $min_{T \in CandN(P)} LA(P|T)$, it relies on the value of $LA(P|N(P))$, which further relies on $LA(P \cup \{N(P)\})$. If the tag of $P \cup \{N(P)\}$ is $LOWERBOUND$, the node values of P are also inaccurate. We prune a node if its current values and tag do not change.

Example 5.5. In Figure 5(c2), we demonstrate the behavior when $w(P_\alpha, T_\beta)$ decreases from 20 to 10. A BFS is initiated from the node P_α (see ④). At the node P_α, $LA(P_\alpha)$ updates from 20 to 10, $N(P_\alpha)$ remains unchanged, and $P_\alpha.tag$ remains the same as $P_\beta.tag$, namely $LOWERBOUND$. We then visit P_α's parent, P_γ. Since P_γ's values and tag do not change, we prune this BFS branch and the BFS stops.

Second Phase: Order Computation. When we complete the BFSs in the first phase, we start computing the optimal orders. Similarly to the basic version, we compute the optimal order starting at an updated table T by a top-down traverse from root node $\{T\}$ following $N(\cdot)$. However, as we delay a part of the node value updates during the above BFSs, the accurate value of $N(\cdot)$ of the descendants of T need to be computed in this process.

We use a recursive procedure $GetActualValues(P)$ (Algorithm 2) to calculate the accurate $LA(P)$ and $N(P)$ for each node P that we visit in the top-down search. In the procedure, we initially assign $LA(P)$ to infinity (line 1). Subsequently, we visit P's children $P \cup \{T\}$ in ascending order of $LA(P|T)$ to refine $LA(P)$ and $N(P)$ until a termination condition is met (line 2-19).

Specifically, when visiting a child $P \cup \{T\}$, we first compute its accurate values (line 6-7). If the child's tag is $LOWERBOUND$, we call $GetActualValues(P \cup \{T\})$ to get its accurate values. Otherwise, its values are already accurate. Next, we compute $LA(P|T)$ from $LA(P \cup \{T\})$ (line 8).

Finally, we update $LA(P)$ as the minimum of all seen $LA(P|T)$, i.e. update $LA(P)$ as $min(LA(P), LA(P|T))$. $N(P)$ is updated accordingly (line 9-11).

Note that we do not need to visit all the children of P. We can stop the visits when we meet one child $P \cup \{\hat{T}\}$ where $LA(P|\hat{T}) \geq LA(P)$ (line 15-19). Because we visit the children of P in ascending order of $LA(P|T)$, the following unvisited children must have larger $LA(P|T)$ values, and will not influence $LA(P)$. Although these $LA(P|T)$ may not be accurate, since they are lower bounds, the accurate values are even higher. Therefore, we can safely prune the visits to them. When this happens, we set $P.tag = EXACT$, and return the procedure (line 18-19).

Algorithm 2: *GetActualValues*

Data: Node P
Result: Accurate $LA(P)$ and $N(P)$. (Return value: $NULL$)

1 $LA(P) = INF$
2 $child\_vec \leftarrow$ sort P's children $P \cup \{T\}$ in ascending order of $LA(P|T)$.
3 $child\_num \leftarrow child\_vec.size()$
4 **foreach** i *in* $0, 1, ..., child\_num - 1$ **do**
5 $T \leftarrow child\_vec[i] - P$ // the child node is $P \cup \{T\}$.
6 **if** $(P \cup \{T\}).tag = LOWERBOUND$ **then**
7 $GetActualValues(P \cup \{T\})$
8 $LA(P|T) \leftarrow (1 + w(P, T) * LA(P \cup \{T\})) * \sum_{T' \in P, (T', T) \in E} deg(T', T)$
9 **if** $LA(P) > LA(P|T)$ **then**
10 $LA(P) \leftarrow LA(P|T)$
11 $N(P) \leftarrow T$
12 **if** $i = child\_num - 1$ **then**
13 $P.tag \leftarrow EXACT$
14 **return**
15 $P_{next\_child} \leftarrow child\_vec[i + 1]$
16 $\hat{T} \leftarrow P_{next\_child} - P$
17 **if** $LA(P) \leq LA(P|\hat{T})$ **then**
18 $P.tag \leftarrow EXACT$
19 **return**

Example 5.6. We discuss how to execute $GetActualValues(P_\gamma)$ **after** updating the CDG when $w(P_\alpha, T_\beta)$ decreases from 20 to 10. The updated CDG is shown in Figure5(c2). P_γ (see ⑤) has two children, P_α and P_δ, and their corresponding $LA(P_\gamma|\cdot)$ values are 35 and 30 respectively. Thus, we visit the node with the smaller $LA(P_\gamma|\cdot)$ value first, which is P_δ. Since $P_\delta.tag = EXACT$, its LA cost is accurate. We set $LA(P_\gamma)$ to 30 and $N(P_\gamma)$ to T_δ. Then we visit P_α and find that $LA(P_\gamma|T_\alpha) = 35$, which is larger than $LA(P_\gamma)$. Thus, we set $P_\gamma.tag$ as $EXACT$ and return the procedure. Note that we do not need to compute the actual values of P_α and its descendants.

In this way, when encountering a node P whose tag is $LOWERBOUND$ during the optimal order computation, we call $GetActualValues(P)$ to calculate its actual values.

6 Reordering Delay Mechanism

In Section 5, we covered the method for recomputing the join order when there is a significant change in a statistic. In this section, we clarify the criteria for a "significant change". Specifically, we determine when a change in a statistic will trigger join reordering.

One naive approach would be to trigger reordering when a statistic changes beyond a predefined ratio. Specifically, for each statistic, which is either of the form $w(\cdot)$ or of the form $deg(\cdot)$, we store two values. The first is a current value, which is used to compute the $LA(P|T)$ values. The second is an actual value, which is updated by Equation 1 or 2 during query execution. When the ratio of the current value to the actual value (or its reciprocal) is beyond a predefined threshold $thre$, we set the current value as the actual value and trigger the reordering algorithm. Here, the current value also serves as a historical record of the statistic value.

However, this approach will trigger unnecessary reordering when encountering outliers of $|Match(T, t_{T'})|$ values which significantly deviate from its expected values. Recall that the statistic $deg(T', T)$ are weighted averages of the $|Match(T, t_{T'})|$ values encountered during query execution, with more weight given to recent data. A recent outlier of $|Match(T, t_{T'})|$ will drastically change the actual value of $deg(T', T)$ and trigger the order recomputation. However, since such outliers are rare, their impact on the data distribution is small and the statistics will quickly return to the expected values as new updates are handled. Thus, the recomputations triggered by outliers do not improve the join orders under the current data distribution and are not worth the overhead.

We propose a reordering delay mechanism to solve this problem. We add a counter for every $deg(T', T)$. Every time we update $deg(T', T)$ during the execution of the query, we check whether the ratio of the current value to the actual value (or its reciprocal) is beyond $thre$. If so, the counter is added by 1. When the counter reaches a predefined counter threshold c, the reordering algorithm is triggered and the current value is set as the actual value. In this way, the current value will be updated only when the actual value persists for a certain period, which indicates that the expected value of $|Match(T, t_{T'})|$ has really changed. Thus, we will trigger reordering only when the data distribution has really changed, and the reordering overhead will be greatly reduced. Note that we do not use the reordering delay mechanism for the statistics $w(\cdot)$, because according to our experiments, outliers of $\frac{|\cap_{T' \in P_\cdot (T', T) \in E} Match(T, t_{T'})|}{\sum_{T' \in P_\cdot (T', T) \in E} |Match(T, t_{T'})|}$ values are very rare and will not influence the overall performance.

7 Experimental Evaluation

In this section, we discuss the experimental evaluation. In Section 7.1, we introduce the experiment setup. In Section 7.2, we compare AJOSC with state-of-the-art algorithms in terms of speed. In Section 7.3, we conduct ablation studies of the LBR method in Section 5.2 and the reordering delay mechanism in Section 6.

7.1 Experiment Setup

In the experiments, all competing algorithms are implemented in c++. The codes are open sourced anonymously [2]. We conduct experiments on a Linux server with 1TB DRAM and two Intel Xeon 2.30GHz 16-core CPU. We use $\theta = 0.04, \gamma = 0.05, t = 1.5, c = 200$ by default in the implementation. If a query takes longer than 4 hours to complete, we terminate the query. The speedups are calculated using 4 hours as the execution time for queries exceeding 4 hours.

Datasets and workloads. We conduct experiments on three benchmarks, the join order benchmark (JOB)[29], LDBC-SNB [17], and JCC-H[10].

In the experiments, the datasets in these benchmarks are used to create the initial databases and generate the update streams. Specifically, the tables in these benchmarks are divided into 2 categories: dimensional tables and fact tables. *The dimensional tables* are static and are used to create the initial databases. The tuples inside these tables are in the initial database and never deleted. *The fact tables* are used to generate the update streams. We use the sliding window model [13] to transform the fact tables into update streams. To be specific, we assign a timestamp to each tuple in the fact tables. The tuple insertions are generated sequentially according to the assigned timestamps of the tuples. To generate tuple deletions, we define a sliding window size of N and keep track of the maximum timestamp T among the inserted tuples. When $T - N$ becomes larger than the timestamp of a tuple, the deletion of the tuple is generated.

As for the query workloads, we remove the select, project and aggregate operators in the queries of these benchmarks. The compared algorithms differentiate with each other only in the execution of join operators. The performance of the other operators is the same. Thus, we remove the other operators to focus on the join performance.

Table 2 shows the statistics of the three datasets.

Table 2. Properties of the benchmarks

	# tuples		raw dataset size	sliding window size
	dimensional	fact		
JOB	1.8M	72.4M	3.6GB	10M tuples
LDBC	24K	175M	6.2GB	3 days
JCC	10K	8.7M	1.1GB	1M tuples

JOB uses the IMDB dataset which includes real-world data about movies and related information about actors, directors, production companies, etc. It consists of 21 tables. We assign 12 tables with less than 1M tuples as dimensional tables and 9 other tables as fact tables. We randomly shuffle all tuples in the fact tables and assign a monotonically non-decreasing timestamp for each tuple with a fixed rate, except tuples in the "movie keyword" table. We set the sliding window size to 10M tuples. The insertions and deletions of tuples in the "movie keyword" table are manipulated to examine the performance of AJOSC under conditions of sudden changes in data distribution, a scenario frequently encountered in real-world applications. To achieve this, we periodically insert a part of the tuples in the "movie keyword" table consecutively. These tuples are deleted consecutively shortly thereafter. Note that the update stream is generated using a combination of the sliding window model and explicit deletions: the deletions of the tuples in the "movie keyword" table do not follow the sliding window model. JOB contains 33 query structures. $|Q.V|$ in each query structure ranges from 4 to 17, and $|Q.E|$ ranges from 4 to 28. Each query structure has 2-6 variants with different select conditions. Since we omit the select conditions, we obtain 33 queries with equi-join conditions only.

LDBC-SNB uses a synthetic graph dataset simulating social networks. We transform the dataset with a scale factor of 10 from the graph model to the relational model following [24] to obtain a relational dataset. It includes 13 tables, where the LDBC SNB documentation [1] categorizes 3 tables as dimensional tables and 10 tables as fact tables. The update stream is generated with the sliding window model. We set the sliding window size to 3 days and assign the timestamp to every tuple using the creation dates given in the original dataset. These creation dates are generated to simulate real-world social networks [1], thus its data distribution change reflects the pattern in real-world applications. LDBC-SNB has 21 graph queries. We transform the queries to multiway join queries following [24]. After the transformation, some queries contain no more than 2 tables, and thus are not multi-way join queries. We omit those queries to get 6 cyclic and 6 acyclic multi-way join queries. $|Q.V|$ of each query ranges from 3 to 6, and $|Q.E|$ ranges from 2 to 8.

JCC-H represents a modified form of TPC-H, which is a benchmark designed for decision support. The dataset models the retail industry market, and it features highly skewed data distributions. It consists of 8 tables, where we assign 3 tables with less than 0.1M tuples as dimensional tables, and 5 other tables as fact tables. In the experiments, we randomly shuffle all tuples and assign monotonically non-decreasing timestamps to them with a fixed rate. We set the sliding window size to 1M tuples. As for query workloads, JCC-H consists of 22 queries, which contain select, project, and aggregate operations. After deleting these operations, only 4 different queries remain. To expand the query set, we randomly select 30 additional queries from the JCC-H dataset. We transform the dataset into a graph and sample 30 subgraphs of sizes 4, 6, and 8. Then we transform these subgraphs back to multi-way join queries. Since most of these queries take more than 4 hours to execute, we only report queries that can be completed within 4 hours in Section 7.2. $|Q.V|$ in each reported query ranges from 3 to 6, and $|Q.E|$ ranges from 3 to 7.

Comparative Algorithms. We compare AJOSC with 5 algorithms. There are four state-of-the-art join order selection methods for continuous multi-way joins, named Two-Step+ [28], Golab's algorithm [18], Lottery scheduling [6] and AGreedy [7] respectively. All algorithms use heuristics. Specifically, TwoStep+, Golab's and AGreedy use greedy approaches, where the table that is expected to yield the smallest intermediate results is ordered first. Two-Step+ recomputes the join order every time an update appears. Golab's algorithm recomputes the join orders only when the ratio of the recent stream rates to previous rates (or their reciprocal) exceed a predefined threshold which is set as 1.2 in our experiments. AGreedy uses a structure called matrix view to maintain statistics and recompute the order when the statistics show that an invariant is violated. The other heuristic, Lottery Scheduling, decides the next table to join using a probabilistic approach, where the tables expected to yield the smallest intermediate results have more chance to be selected. In addition to the 4 heuristics, we also compare AJOSC with the dynamic programming algorithm (StaticDP) described in Section 3. We use StaticDP to recompute the join orders from scratch in a batch manner to adapt to changes in data distribution. The batch size is set to 1000 updates.

These methods are all the state-of-the-art join order selection methods for continuous multi-way joins we find.

Data ingestion method. In our experiments, we process data updates in the update stream consecutively in the order of their timestamps. When an algorithm finishes processing an update operation, it will immediately get the next one in the update stream. In this way, each algorithm is evaluated with its utmost processing power, i.e. with the maximum data ingestion rate it can handle. This processing method is widely use in previous works, e.g. [28].

7.2 Speed Comparison

We compare the execution time and the tail latency of the queries.

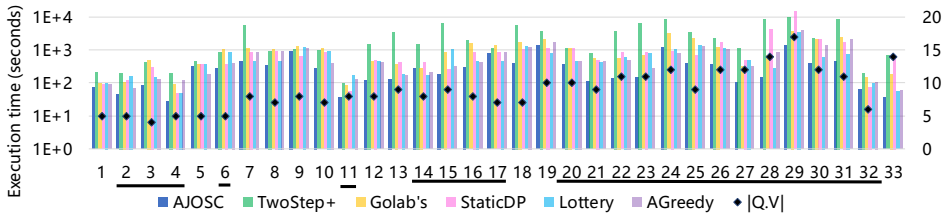

Fig. 6. Comparison of the execution time on JOB. The underlined queries involve the "movie keyword" table and are influenced by abrupt fluctuations in data distribution. The execution times exceeding 4 hours are marked as 4 hours.

Table 3. Average execution time (seconds)

	AJOSC	TwoStep+	Golab's	StaticDP	Lottery	AGreedy
JOB	**359**	2857	1048	1225	638	698
LDBC	**404**	1604	1302	618	806	881
JCC	**2935**	7950	7240	4014	4747	5919

Table 4. Average 99.9% tail latency (microseconds)

	AJOSC	TwoStep+	Golab's	StaticDP	Lottery	AGreedy
JOB	**62**	1386	130	419	107	100

The average execution time is shown in Table 3. Due to space limitation, we only show the execution time of every query in JOB in Figure 6. AJOSC is faster than all the comparative algorithms and the speed-up is up to 2 orders of magnitude. Averaging across all three benchmarks, AJOSC is on average 4.0x faster than Two-Step+, 2.6x faster than Golab's algorithm, 1.8x faster than StaticDP, 1.7x faster than Lottery Scheduling, and 2.0x faster than AGreedy. As stated in Section 7.1, the execution time reflects the maximum data ingestion rate that an algorithm can handle, which can be computed by dividing the total update number by the execution time.

Recall that the JOB dataset simulates sudden changes in data distribution by continuously inserting and deleting tuples from the "movie keyword" table. In Figure 6(a), queries involving this table are underlined. AJOSC is 4.48x faster than StaticDP for these queries in terms of average query time. This is significantly higher compared to the speedups for the queries with stable data distribution. This demonstrates the effectiveness of AJOSC in timely response to distribution change. In contrast, StaticDP with the batch manner update cannot adapt quickly to the change and suffers from sub-optimal orders. On the other hand, when the distribution is stable, AJOSC is still faster than StaticDP because of its lower overhead of order computation.

In queries with larger $|Q.V|$, the speedup of AJOSC over StaticDP is greater, which supports the theoretical analysis presented in Section 4.3 that the time complexity of StaticDP is $O(|Q.V|)$ times larger than that of AJOSC. For example, the $|Q.V|$ of query 1 and 28 are respectively 5 and 14, and the speedups are 1.22x and 14x respectively. The speedup of the larger query 28 is much larger than that of the smaller query 1.

The 99.9% tail latencies are shown in Table 4. Compared to AJOSC, Two-Step+'s average latency is 22x higher, Golab's is 2.1x higher, StaticDP's is 8x higher, Lottery Scheduling's is 1.7x higher and AGreedy's is 1.6x higher. These findings indicate a significant performance disparity in favor of AJOSC with respect to latency management.

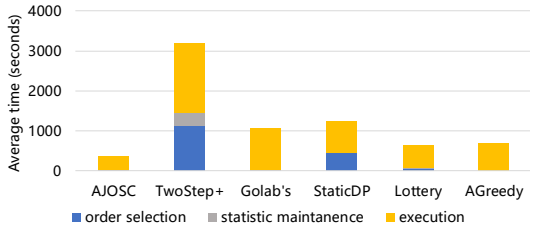

Fig. 7. Running Time Breakdown

7.3 Ablation Studies

In this subsection, we prove that the LBR method and the reordering delay mechanism are effective, by comparing AJOSC with degraded versions without those techniques using the JOB benchmark. The experiment results are shown in Table 5.

Table 5. Ablation study: Average execution time (seconds).

	AJOSC	basic incremental	naïve trigger reordering
JOB	359	485	535

The effect of the LBR method. We compare AJOSC with a degraded version using the basic version of the incremental reordering algorithm in Section 5.1. The full AJOSC is 1.35x faster than the degraded version on average, which proves the effectiveness of the LBR method.

The effect of the reordering delay mechanism. We compare AJOSC with a degraded variant that utilizes a naive triggering reordering algorithm, which triggers order recomputation whenever statistics change beyond a predefined threshold. The full AJOSC is 1.49x faster than the degraded version on average, which proves the effectiveness of the reordering delay mechanism.

7.4 Running Time Breakdown

The process of running a continuous multiway join query includes maintaining statistics, selecting join orders, and executing the query. Figure 7 shows the average running time breakdown on JOB. As shown in the figure, the overhead of AJOSC is remarkably small. The running time is almost all on query execution. The overhead of Golab's and AGreedy is also very small, but their selected orders are worse than AJOSC's, since they are both greedy heuristics. The overhead of Lottery scheduling, StaticDP and TwoStep+ is much larger than AJOSC, but they fail to select better orders.

7.5 Scalability Evaluation

In this subsection, we vary the sliding window size to test the scalability of AJOSC. The results are shown in Figure 8. We show the average time of all queries in JOB, and show the results of queries 5, 19 and 29 which cover small, medium, and large query sizes. As shown in Figure 8, the execution time is longer with larger sliding window sizes. This is because when the window size increases, the number of active tuples in the fact tables increases, where active tuples are the tuples inserted and not deleted yet. Having a larger number of active tuples leads to higher join costs among these tuples.

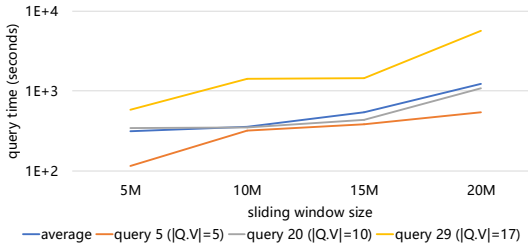

Fig. 8. Performance with different sliding window sizes

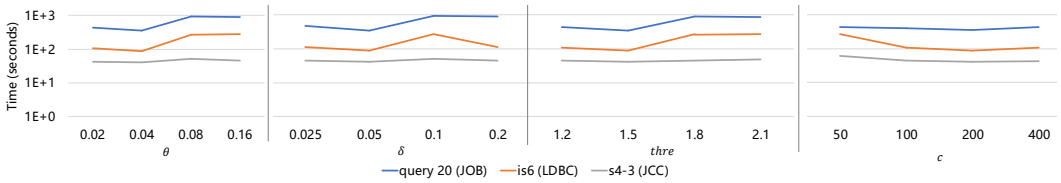

Fig. 9. Sensitivity

7.6 Sensitivity Evaluation

In this subsection, we vary the parameters θ, δ, $thre$ and c to test the sensitivity of AJOSC to the parameters on all 3 benchmarks. We choose query 20 from JOB, is6 from LDBC, s4-3 from JCC as the representative queries. These queries cover all benchmarks, and show the sensitivity of AJOSC for different data distributions. The results are shown in Figure 9.

Recall that θ determines the weights of new values when updating the $deg(\cdot)$ statistics (Equation 1), and δ determines the weights of new values when updating the $w(\cdot)$ statistics (Equation 2). For these 2 parameters, increasing their values tends to create greater jitter in statistics. This will trigger unnecessary reorderings that bring greater overhead. Furthermore, when encountering outliers, larger parameters will cause the statistics to increase greatly. This leads to inaccurate statistics, which cause AJOSC to select suboptimal orders. In contrast, smaller parameters make reordering less frequent, which can lead to outdated orders. As shown in Figure 9, both larger and smaller θ and δ result in longer execution time.

$thre$ is the threshold for a "significant" change in statistics (Section 6). When $thre$ is small, AJOC frequently triggers reordering, leading to high overhead. When $thre$ is large, the algorithm rarely reorders, which results in outdated orders. As shown in Figure 9, both larger and smaller $thre$ result in longer execution time.

c is the count threshold in the reordering delay mechanism. When c is small, outliers trigger more unnecessary order recomputations, which enlarges overhead and causes suboptimal orders because outliers make statistics inaccurate. When c is large, when data distribution changes, order recomputation will be delayed, which results in suboptimal orders when orders have not been recomputed. As shown in Figure 9, both larger and smaller c results in longer execution time.

Generally, the performance of AJOSC is stable when θ ranges from 0.02 to 0.04, when δ ranges from 0.025 to 0.05, and when c ranges from 100 to 400. On JOB and LDBC, the performance of AJOSC is stable when $thre$ ranges from 1.2 to 1.5. On JCC, the performance of AJOSC is stable when $thre$ ranges from 1.2 to 2.1. Note that JCC has a lower sensitivity to parameters. Because the data distribution is stable on JCC, delay in order recomputation will not lead to suboptimal orders.

7.7 Assumption Verification

Recall that when a tuple is inserted into (or deleted from) the database, we assumpt that the join order needs to start at the updated table. In this subsection, we conduct experiments to verify this assumption. We conduct this experiment on query 1 of JOB. We enumerate all the 36 possible join orders that can keep the join graph connected and avoid Cartesian products in the joining process. For each join order, we use it in the whole executing process of a query. Since the updated table is constantly changing, most continuous queries are executed with a join order that does not start from the updated table. It turns out that none of these 36 experiments finishes in 4 hours. In contrast, it only takes 72 seconds to run query 1 of JOB when the join orders always start at the updated table. This shows that if the updated table is not the first in the join order, the query execution will be really slow.

8 Related Work

8.1 Continuous Queries

Continuous queries, also known as standing queries, continuously monitor query results as the database is updated. The CQL continuous query language [5] is a SQL-based declarative language to register continuous queries. AJOSC can be used in CQL engines to accelerate continuous multi-way join.

8.2 Join Order Selection

8.2.1 Static setting. The join order selection problem, i.e. how to select a good join order for a multi-way join query, has been richly explored in the static settings [11, 14, 31, 36, 37, 41, 42, 47–49]. One classical solution is StaticDP [37], which we discuss in Section 3. With the development of machine learning, recently several works use deep reinforcement learning to learn a good join order [11, 31, 48, 49].

Static setting + adaptive query processing. In static settings, cardinality estimation is important for join order selection, but estimating cardinalities before query execution both accurately and quickly is very hard. To get proper join orders without using too much time estimating cardinalities before execution, some works begin query execution with suboptimal orders, but gather additional information during query execution. This additional information helps refine the cardinality estimates, and the query processing system can switch to better join orders using the refined accurate estimates. This technique is called adaptive query processing. The works on adaptive query processing in static settings include [41, 42, 47]. They use machine learning techniques to use new information to compute new query plans.

8.2.2 Dynamic setting. Join order selection for the dynamic settings has also been explored in previous works [6, 7, 18, 28]. Since in the dynamic settings, data distribution may change dramatically over time, the idea of adaptive query processing is also used in the dynamic settings. Adaptive query processing techniques in the dynamic settings collect information during query execution and compute new join orders suitable for the current data distribution. The lottery scheduling method [6] decides the next table to join using the current join selectivity estimates. However, it spends a lot of time exploring suboptimal orders [32], which makes it slower than AJOSC. This is also shown in our experiments in Section 7.4. Two-Step+ [18] and Golab's algorithm [28] use greedy approaches to choose join orders using the current cost estimates. The Stanford STREAM system [4] proposes AGreedy [7], a greedy approach to adaptively change the filtering order of pipelined stream filters, which can be transformed to compute the join order. Comparing to [7, 18, 28], AJOSC can select optimal join orders if the statistics are accurate, while these 3 greedy approaches cannot.

Thus, AJOSC spends less time executing the queries, which is shown in our experiments in Section 7.4.

8.3 Other Techniques for Continuous Join Queries

To accelerate the execution of continuous multiway join, numerous works maintain intermediate results of the join queries and join the incoming tuple with the intermediate results to obtain the results directly. This topic is called incremental view maintenance (abbr. IVM), and extensive research has been conducted on this topic [3, 21, 25, 40, 43]. These works [19, 23, 32, 46] apply adaptive query processing methods to accelerate incremental view maintenance. Continuous subgraph matching (abbr. CSM) algorithms [12, 27, 35, 38, 39], which are analogous to continuous multi-way join algorithms in the context of graph databases, operate similarly to IVM approaches. AJOSC differs from these works: IVM and CSM algorithms maintain views to accelerate the queries, which occupy large amounts of memory, making these works inapplicable to scenarios with a large amount of data. For example, [25] processes 125M tuples (totaling 5GB), but requires 32GB of memory, which is 6.4 times the data size. In contrast, AJOSC does not use views and consumes much less memory, making it applicable in a broader range of scenarios. Moreover, the join order selection algorithm of the IVM/CSM methods cannot work without maintaining views, since the join orders of these methods depend on the views.

There are also prior works studying window-based multi-joins, e.g. [50]. In these works, tuples are deleted in the order of their timestamps. In contrast, AJOSC is capable of managing not only window-based multi-joins, but also arbitrary deletions (which are also called explicit deletions). In the explicit deletion scenario, tuples are deleted according to received instructions, which do not follow temporal order.

There are also studies on sharing the computation of multiple continuous multi-way join queries [16, 33] and reaching high scalability in the distributed setting [15, 26, 45]. In this paper, we focus on accelerating a single query on a single computation node.

9 Conclusion

In this paper, we propose AJOSC, an adaptive join order selection algorithm for continuous multi-way join queries. We propose a new cost model named LA cost that can select high-quality join orders with low overhead, an incremental reordering algorithm that can recompute the join orders with low overhead when data distribution changes, and a reordering delay mechanism to decide the timing of updating the join orders which avoids unnecessary reorderings. Experimental results show that AJOSC accelerates the queries by up to two orders of magnitude compared to prior arts.

Acknowledgments

This work was supported by The National Key Research and Development Program of China under grant 2023YFB4502303, ARC DP230101445 and ARC FT210100303. The corresponding author is Xiangyang Gou.

References

[1] 2024. ldbc-snb-specification. https://ldbcouncil.org/ldbc_snb_docs/ldbc-snb-specification.pdf

[2] 2024. *Source code of AJOSC*. https://anonymous.4open.science/r/AJOSC/

[3] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. 2012. DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views. (2012).

[4] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. 2016. STREAM: The Stanford Data Stream Management System. In *Data Stream Management: Processing High-Speed Data Streams*, Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi (Eds.). Springer, Berlin, Heidelberg, 317–336. doi:10.1007/978-3-540-28608-0_16

[5] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal* 15 (2006), 121–142.

[6] Ron Avnur and Joseph M. Hellerstein. 2000. Eddies: Continuously Adaptive Query Processing. *ACM SIGMOD Record* 29, 2 (June 2000), 261–272. doi:10.1145/335191.335420

[7] Shivnath Babu, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa, and Jennifer Widom. 2004. Adaptive Ordering of Pipelined Stream Filters. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD '04)*. Association for Computing Machinery, New York, NY, USA, 407–418. doi:10.1145/1007568.1007615

[8] Albert Bifet and Ricard Gavalda. 2007. Learning from time-changing data with adaptive windowing. In *Proceedings of the 2007 SIAM international conference on data mining*. SIAM, 443–448.

[9] Arezo Bodaghi and Babak Teimourpour. 2018. Automobile insurance fraud detection using social network analysis. *Applications of Data Management and Analysis: Case Studies in Social Networks and Beyond* (2018), 11–16.

[10] Peter Boncz, Angelos-Christos Anatiotis, and Steffen Kläbe. 2018. JCC-H: adding join crossing correlations with skew to TPC-H. In *Performance Evaluation and Benchmarking for the Analytics Era: 9th TPC Technology Conference, TPCTC 2017, Munich, Germany, August 28, 2017, Revised Selected Papers 9*. Springer, 103–119.

[11] Jin Chen, Guanyu Ye, Yan Zhao, Shuncheng Liu, Liwei Deng, Xu Chen, Rui Zhou, and Kai Zheng. 2022. Efficient Join Order Selection Learning with Graph-based Representation. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '22)*. Association for Computing Machinery, New York, NY, USA, 97–107. doi:10.1145/3534678.3539303

[12] Sutanay Choudhury, Lawrence Holder, George Chin, Khushbu Agarwal, and John Feo. 2015. A Selectivity Based Approach to Continuous Pattern Detection in Streaming Graphs. arXiv:1503.00849 [cs]

[13] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 2002. Maintaining Stream Statistics over Sliding Windows. *Siam Journal on Computing* 31, 6 (2002), 1794–1813.

[14] David DeHaan and Frank Wm Tompa. [n. d.]. Optimal Top-Down Join Enumeration (Extended Version). ([n. d.]).

[15] Manuel Dossinger and Sebastian Michel. 2019. Scaling Out Multi-Way Stream Joins Using Optimized, Iterative Probing. In *2019 IEEE International Conference on Big Data (Big Data)*. 449–456. doi:10.1109/BigData47090.2019.9005973

[16] Manuel Dossinger and Sebastian Michel. 2021. Optimizing Multiple Multi-Way Stream Joins. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, Chania, Greece, 1985–1990. doi:10.1109/ICDE51399.2021.00188

[17] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 619–630. doi:10.1145/2723372.2742786

[18] Lukasz Golab and M. Tamer Özsu. 2003. - Processing Sliding Window Multi-Joins in Continuous Queries over Data Streams*. In *Proceedings 2003 VLDB Conference*, Johann-Christoph Freytag, Peter Lockemann, Serge Abiteboul, Michael Carey, Patricia Selinger, and Andreas Heuer (Eds.). Morgan Kaufmann, San Francisco, 500–511. doi:10.1016/B978-012722442-8/50051-3

[19] Joseph Gomes and Hyeong-Ah Choi. 2008. Adaptive Optimization of Join Trees for Multi-Join Queries over Sensor Streams. *Information Fusion* 9, 3 (July 2008), 412–424. doi:10.1016/j.inffus.2007.06.001

[20] Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. 2020. General dynamic Yannakakis: conjunctive queries with theta joins under updates. *The VLDB Journal* 29, 2 (2020), 619–653.

[21] Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. 2020. General Dynamic Yannakakis: Conjunctive Queries with Theta Joins under Updates. *The VLDB Journal* 29, 2-3 (May 2020), 619–653. doi:10.1007/s00778-019-00590-9

[22] Balakrishna R Iyer and Arun N Swami. 1994. Method for optimizing processing of join queries by determining optimal processing order and assigning optimal join methods to each of the join operations. US Patent 5,345,585.

[23] Albert Jonathan, Abhishek Chandra, and Jon Weissman. 2020. WASP: Wide-area Adaptive Stream Processing. In *Proceedings of the 21st International Middleware Conference*. ACM, Delft Netherlands, 221–235. doi:10.1145/3423211.3425668

[24] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An Active Graph Database. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, Chicago Illinois USA, 1695–1698. doi:10.1145/3035918.3056445

[25] Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. 2023. F-IVM: Analytics over Relational Databases under Updates. doi:10.48550/arXiv.2303.08583 arXiv:2303.08583 [cs]

[26] Jeyhun Karimov, Tilmann Rabl, and Volker Markl. 2019. AJoin: Ad-Hoc Stream Joins at Scale. *Proceedings of the VLDB Endowment* 13, 4 (Dec. 2019), 435–448. doi:10.14778/3372716.3372718

[27] Kyoungmin Kim, In Seo, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. 2018. TurboFlux: A Fast Continuous Subgraph Matching System for Streaming Graph Data. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, Houston TX USA, 411–426. doi:10.1145/3183713.3196917

[28] Danh Le-Phuoc. 2018. Adaptive Optimisation For Continuous Multi-Way Joins Over RDF Streams. In *Companion of the The Web Conference 2018 on The Web Conference 2018 - WWW '18*. ACM Press, Lyon, France, 1857–1865. doi:10.1145/3184558.3191653

[29] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proceedings of the VLDB Endowment* 9, 3 (Nov. 2015), 204–215. doi:10.14778/2850583.2850594

[30] Feifei Li. 2019. Cloud-native database systems at Alibaba: Opportunities and challenges. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2263–2272.

[31] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *Proceedings of the VLDB Endowment* 12, 11 (July 2019), 1705–1718. doi:10.14778/3342263.3342644 arXiv:1904.03711 [cs]

[32] Mengmeng Liu, Zachary G. Ives, and Boon Thau Loo. 2016. Enabling Incremental Query Re-Optimization. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, San Francisco California USA, 1705–1720. doi:10.1145/2882903.2915212

[33] Amine Mhedhbi, Chathura Kankanamge, and Semih Salihoglu. 2021. Optimizing One-time and Continuous Subgraph Queries Using Worst-case Optimal Joins. *ACM Transactions on Database Systems* 46, 2 (May 2021), 6:1–6:45. doi:10.1145/3446980

[34] Simona Micevska, Ahmed Awad, and Sherif Sakr. 2021. SDDM: an interpretable statistical concept drift detection method for data streams. *Journal of intelligent information systems* 56, 3 (2021), 459–484.

[35] Seunghwan Min, Sung Gwan Park, Kunsoo Park, Dora Giammarresi, Giuseppe F. Italiano, and Wook-Shin Han. 2021. Symmetric Continuous Subgraph Matching with Bidirectional Dynamic Programming. *Proceedings of the VLDB Endowment* 14, 8 (April 2021), 1298–1310. doi:10.14778/3457390.3457395

[36] Guido Moerkotte and Thomas Neumann. 2006. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *Proceedings of the 32nd International Conference on Very Large Data Bases* (Seoul, Korea) *(VLDB '06)*. VLDB Endowment, 930–941.

[37] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data* (Boston, Massachusetts) *(SIGMOD '79)*. Association for Computing Machinery, New York, NY, USA, 23–34. doi:10.1145/582095.582099

[38] Shixuan Sun, Xibo Sun, and Bingsheng He. 2022. RapidFlow: An Efficient Approach to Continuous Subgraph Matching. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2415–2427.

[39] Xibo Sun, Shixuan Sun, Qiong Luo, and Bingsheng He. 2022. An In-Depth Study of Continuous Subgraph Matching. *Proceedings of the VLDB Endowment* 15, 7 (March 2022), 1403–1416. doi:10.14778/3523210.3523218

[40] Christoforos Svingos, Andre Hernich, Hinnerk Gildhoff, Yannis Papakonstantinou, and Yannis Ioannidis. 2023. Foreign Keys Open the Door for Faster Incremental View Maintenance. *Proceedings of the ACM on Management of Data* 1, 1 (May 2023), 40:1–40:25. doi:10.1145/3588720

[41] Kostas Tzoumas, Timos Sellis, and Christian S Jensen. 2008. A Reinforcement Learning Approach for Adaptive Query Processing. (2008).

[42] Junxiong Wang, Immanuel Trummer, Ahmet Kara, and Dan Olteanu. 2023. ADOPT: Adaptively Optimizing Attribute Orders for Worst-Case Optimal Join Algorithms via Reinforcement Learning. *Proceedings of the VLDB Endowment* 16, 11 (July 2023), 2805–2817. doi:10.14778/3611479.3611489

[43] Qichen Wang and Ke Yi. 2020. Maintaining Acyclic Foreign-Key Joins under Updates. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. ACM, Portland OR USA, 1225–1239. doi:10.1145/3318464.3380586

[44] Qichen Wang, Chaoqi Zhang, Danish Alsayed, Ke Yi, Bin Wu, Feifei Li, and Chaoqun Zhan. 2021. Cquirrel: Continuous Query Processing over Acyclic Relational Schemas. *Proceedings of the VLDB Endowment* 14, 12 (July 2021), 2667–2670. doi:10.14778/3476311.3476315

[45] Qihang Wang, Decheng Zuo, Zhan Zhang, Siyuan Chen, and Tianming Liu. 2023. An adaptive non-migrating load-balanced distributed stream window join system. *The Journal of Supercomputing* 79, 8 (2023), 8236–8264.

[46] Zuozhi Wang, Kai Zeng, Botong Huang, Wei Chen, Xiaozong Cui, Bo Wang, Ji Liu, Liya Fan, Dachuan Qu, Zhenyu Hou, Tao Guan, Chen Li, and Jingren Zhou. 2023. Tempura: A General Cost-Based Optimizer Framework for Incremental Data Processing (Journal Version). *The VLDB Journal* 32, 6 (Nov. 2023), 1315–1342. doi:10.1007/s00778-023-00785-1

[47] Ziyun Wei and Immanuel Trummer. 2022. SkinnerMT: Parallelizing for Efficiency and Robustness in Adaptive Query Processing on Multicore Platforms. *Proceedings of the VLDB Endowment* 16, 4 (Dec. 2022), 905–917. doi:10.14778/3574245.3574272

[48] Zhengtong Yan, Valter Uotila, and Jiaheng Lu. 2023. Join Order Selection with Deep Reinforcement Learning: Fundamentals, Techniques, and Challenges. *Proceedings of the VLDB Endowment* 16, 12 (Sept. 2023), 3882–3885. doi:10.14778/3611540.3611576

[49] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. 2020. Reinforcement Learning with Tree-LSTM for Join Order Selection. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1297–1308. doi:10.1109/ICDE48307.2020.00116

[50] Dongdong Zhang, Jianzhong Li, Kimutai Kimeli, and Weiping Wang. 2006. Slidingwindow based multi-join algorithms over distributed data streams. In *22nd International Conference on Data Engineering (ICDE'06)*. IEEE, 139–139.

Received October 2024; revised January 2025; accepted February 2025