Scalable Online Interval Join on Modern Multicore Processors in OpenMLDB

Hao Zhang[#], Xianzhi Zeng[†‡], Shuhao Zhang[†‡], Xinyi Liu[#], Mian Lu[#], Zhao Zheng[#]

[#]*4Paradigm Inc.*
{zhanghao, liuxinyi, lumian, zhengzhao}
@4paradigm.com

[†‡]*Singapore University of Technology and Design*
1007420@mymail.sutd.edu.sg
shuhao_zhang@sutd.edu.sg

***Abstract*—OpenMLDB is an open-source machine learning database, that provides a feature platform computing consistent features for training and inference. The online interval join (`OIJ`), i.e., joining two input streams over relative time intervals, is becoming a core operation in OpenMLDB. Its costly nature and intrinsic parallelism opportunities have created significant interest in accelerating `OIJ` on modern multicore processors. In this work, we first present an in-depth empirical study on an existing parallel `OIJ` algorithm (`Key-OIJ`), which applies a key-partitioned parallelization strategy. `Key-OIJ` has been implemented in Apache Flink and used in real-world applications. However, our study points out the limitations of `Key-OIJ`, and reveals that `Key-OIJ` is not capable of fully exploiting modern multicore processors. Based on our analysis, we propose a new approach, the `Scale-OIJ` algorithm with a set of optimization techniques. Compared with `Key-OIJ`, `Scale-OIJ` is particularly efficient for handling workloads involving fewer keys, large time intervals, and large lateness configurations. The extensive experiments using real workloads have demonstrated the superior performance of `Scale-OIJ`. Furthermore, we have partially integrated and tested `Scale-OIJ` in the latest version of OpenMLDB, demonstrating its practicality in a machine learning database.**

I. Introduction

On-line decision augmentation (OLDA) powered by AI has become one of the fastest-growing and promising paradigms to enable timely decision making, and is forecast to take up 44% of total business value in AI [11]. The life cycle of an OLDA is depicted in Figure 1, where feature in-consistency caused by separate development teams/software stacks is a big challenge. OpenMLDB[1] is a machine learning database system at 4Paradigm that is natively designed for feature engineering (FE), aiming to provide a consistent and efficient FE platform for both offline training and online serving. OpenMLDB has been integrated into 4Paradigm's commercial product of machine learning platforms (the users include Industrial and Commercial Bank of China, Yum China and so on), as well as publicly available as open-source software (the users include Akulaku, Huawei, 37Games and so on). It is widely used in 100+ scenarios in production, such as network traffic forecast, anti-fraud, product recommendation [2], etc.

For a machine learning application, especially in the industry of OLDA scenarios, time-series features, e.g., the user

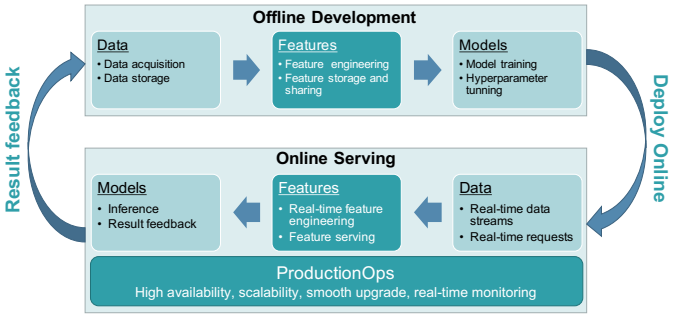


Fig. 1: Life Cycle of an OLDA Application

behaviour in the latest 10 minutes, are commonly used and significant to the model performance. Time-series features can be defined as window aggregations in SQL, where the online interval join (`OIJ`), i.e., joining two input streams over relative time intervals, is the core operation, as the window boundary is usually defined with respect to the primary table/stream[2], while window data is resident in a secondary table/stream. For example, in an online shopping platform, when a user is browsing or searching (recorded in the action table), we will recommend products based on the pre-defined features, which may require joining the tuples in the history orders within the last certain period (recorded in the order table). The window in `OIJ` is a relative time window to the time of the current tuple. It is different from the absolute time window in the general stream window join [17], [20], which is used to constrain the unbounded streams to a limited set of tuples. At OpenMLDB, we observed that, for many data-driven analytical tasks, users often have `OIJ` queries with a large time interval over high-rate data streams as features [1], [18].

The stringent response time requirements by online services (e.g., some bank users require 20 ms latency for the feature computing) make the accelerating `OIJ` a critical issue. Multicore processors that provide high processing capacity are ideal for executing costly `OIJ` queries with rich intrinsic parallelism opportunities. However, fully exploiting the potential of a multicore processor is often challenging due

[1]https://github.com/4paradigm/OpenMLDB

[2]We will use table or stream interchangeably in the paper without causing ambiguity.

to the complex processor microarchitecture. Furthermore, as data streams may arrive out of order, it complicates the parallel design of `OIJ` as the assumption of increasing time order is not valid. Unfortunately, there are few prior research works on parallelizing `OIJ`. There is a rich literature on parallelizing offline "interval join" [5]–[7], [13], which, however, have a different meaning. It is defined on interval relations where each tuple has `left` and `right` endpoints representing an interval, and "interval join" is basically a relational join with a predicate checking that tuple intervals overlap. In additional, they focus on static datasets stored in a database, which usually require full sorting to achieve a total time ordering. On the other hand, studies of parallel online/stream join, particularly parallel sliding window join [8], [12], [14], [15], [17], focus on a related but independent problem. The window boundaries are defined in terms of an absolute time interval, irrelevant of the relative timestamps of each tuple in the data streams. Thus the window can be naturally partitioned into independent sub-windows for high parallelism [12], [17], which is non-trivial for `OIJ`.

`Key-OIJ` is a key-partitioned based parallel `OIJ` algorithm, which is the only available `OIJ` algorithm, to the best of our knowledge. It has been adopted in Apache Flink [4] - a state-of-the-art stream processing system, and used in real-world applications [3]. To parallelize, `Key-OIJ` partitions and buffers each incoming tuple according to its key. In this way, input tuples may be concurrently joined with the buffer of the same key of the opposite stream. To check the intersection of the time interval, a thread may scan through the buffer to filter out relevant tuples. Ideally, tuples may be immediately removed from the buffer when they are not possible to form any matches. However, to handle potential out-of-order stream arrival, tuples can be only removed after a period of time. Lateness is usually specified to represent the maximum degree of disorder and can be used to indicate the expiration of tuples [9].

In this paper, we first present an in-depth empirical study of `Key-OIJ` using a set of real-world workloads and synthetic datasets from 4Paradigm. Our study points out the limitations of `Key-OIJ`, and reveals that `Key-OIJ` is not capable of fully exploiting modern multicore processors. First, `Key-OIJ` involves costly manipulation of out-of-order data, performing extremely poorly under large lateness. In particular, in Flink, a full data scan has to be conducted for each join operation. Second, `Key-OIJ` suffers from unbalanced workload scheduling due to its key-based partition strategy, especially when the number of keys is small. Third, `Key-OIJ` can not utilize the previously handled data among overlapping windows, which leads to significant redundant computation under a large window.

Based on our analysis, we propose a new approach, the `Scale-OIJ` algorithm with a set of optimization techniques:

1) *Single-Writer-Multiple-Reader Time-Travel Data Structure* to improve the efficiency of inserting, retrieving, and the expiration of data. Particularly, a concurrent two-layer skip-list is used to eliminate the unnecessary data retrieval of out-of-window data, making the lateness insignificant to the performance.
2) *Dynamic Balanced Schedule* to address the unbalanced key partition. In this way, the workload assignment is dynamically re-scheduled to balance each joiner and improve the overall performance.
3) *Incremental Window Aggregation* to address the window overlapping and reduce unnecessary repeat computation. Specifically, the *Subtract-on-Evict technique* [16] is adapted to share aggregation results among overlapping windows. Therefore, the computation efficiency is improved and high performance is achievable even when the window is large.

TABLE I: Notations

Notations	Description
$x = \{t, k, p\}$	An input tuple x with three attributes, i.e., timestamp, key and payload
S	base stream to join
R	probe stream to join
v	Arrival rate of a stream
u	Number of unique keys in a stream
PRE	preceding offset relative to the current time
FOL	following offset relative to the current time
$w_i = (t_i^s, t_i^e)$	A window with start timestamp t_i^s and end timestamp t_i^e
$\|w_i\|$	Window length for w_i
l	Lateness configuration

Compared with `Key-OIJ`, `Scale-OIJ` is particularly efficient for handling `OIJ` on workloads involving less keys, large time windows, and large lateness configurations. The evaluation results show that all of the optimizations are effective in improving the performance of `OIJ`. Putting them altogether achieves up to $24\times$ throughput improvement and up to 99% latency reduction on a recent multicore machine.

The remainder of this paper is organized as follows. Section II introduces preliminary and background of this study. Section III presents the methodology of this study including research goals and benchmark workloads, followed by the experimental study in Sections IV. Section V describes our efforts in addressing the performance issues of `OIJ`. We review the related work in Section VI and conclude in Section VII.

II. Preliminaries

We introduce the research background in this section. All notations used throughout the paper are summarized in Table I.

A. OpenMLDB

OpenMLDB is an united feature platform computing consistent features for machine learning training and inference. It mainly consists of a batch SQL engine (offline) and a realtime SQL engine (online), which share the same execution plan generator as shown in Figure 2. In this paper, we focus on the online engine, which is used to extract features in real time. The APIs provided by OpenMLDB is based on SQL, with an extension of `Window Union`[3], which achieves the

[3]https://openmldb.ai/docs/en/main/reference/sql/dql/WINDOW_CLAUSE.html

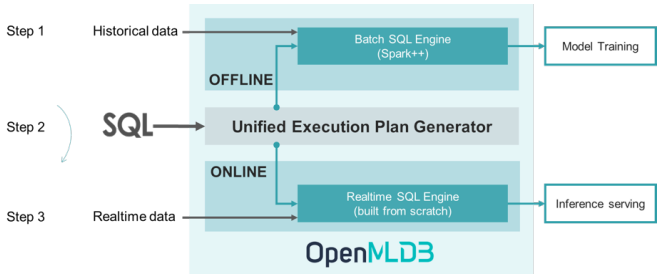


Fig. 2: Architecture of OpenMLDB

same semantics in the SQL way as interval join. To express the operation in Figure 3, the SQL in OpenMLDB can be written as follows:

```
SELECT sum(col2) over w1 FROM S
WINDOW w1 AS (
  UNION R
  PARTITION BY key
  ORDER BY timestamp
  ROWS_RANGE
  BETWEEN 1s PRECEDING AND 1s FOLLOWING);
```

Here we use `sum` as the aggregation function to demonstrate a complete SQL script. Such kind of features illustrated in the above SQL are common in most machine learning scenarios. For example, we may want some aggregation features in a product recommendation service, say, the recent 1 hour of user search history. This may involve an interval join of a `user` table and `search` table [4]. In addition, high throughput and low latency are commonly required in these cases. For instance, a 20 ms latency is strictly required by an online banking service of our customers.

However, OpenMLDB is more suitable for stable data processing, where a large portion of data is pre-loaded into the system before the online feature extraction. It currently has limited capability of stream processing. In particular, OpenMLDB is not good at handling workloads with high arrival rate as all the processing threads share the same data structure; thus insertion will become a potential performance bottleneck. Moreover, it cannot properly handle data out-of-order, which is common in streaming. Hence, in this work we equip OpenMLDB with the capability of stream processing.

B. Window-based Stream Processing

Stream processing performs queries on a large amount of continuously arriving data, with low latency and real-time requirements as the main characteristics. We define a *tuple* x as $x = \{t, k, p\}$, where t, k and p are the timestamp, key, and payload of the tuple, respectively. An *input stream* (denoted as R or S) is formed by a list of tuples chronologically arriving at the system. For a given data stream, we further define the number of tuples arriving in one second (i.e., tuples/s) as *arrival rate* (v), and use the *number of unique keys* (u) to reflect the total amount of different keys in the streams. To handle continuous input streams, at a moment in time, we usually operate on only a continuous bounded subset of the stream data (i.e., a window), which is defined below.

Definition 1: We define a **window** as $w_i = (t_i^s, t_i^e)$, where the timestamps t_i^s and t_i^e indicate the start and end time of window w_i, respectively. With regards to the data stream R, its tuples on w_i can be defined as $w_i^R = \{(t, k, p) \mid t_i^s \leq t < t_i^e\}$. We denote $|t_i^e - t_i^s|$ by $|w_i|$ as the **window length**.

A tuple x_0 with a smaller timestamp t_0 may arrive even later in the system than a tuple x_1 with a larger timestamp t_1. To handle such out-of-order arrival, `lateness` is usually used to allow late arrival of tuples. In particular, the `lateness` l specifies how much time tuples can be late in the system. As a result, the expiration of tuples has to be delayed for l in order to wait for the late-arrival tuples that may join with these expired tuples, thus guaranteeing the correctness of the final results, which has been proven in [9].

C. Parallel Online Interval Join

Definition 2: The `OIJ` joins elements of two streams S as the base stream and R as the probe stream, on the condition that S and R have a common key and tuples of stream R have timestamps that lie in the relative time window to the timestamp of stream S. The time window is denoted by (`PRE`, `FOL`), corresponding to the preceding and following offsets relative to the current timestamp. $S \bowtie_{OIJ} R$ are defined as all the interval join results $\{w_i^{\bowtie} \mid \forall S_i \in S\}$ between S and R.

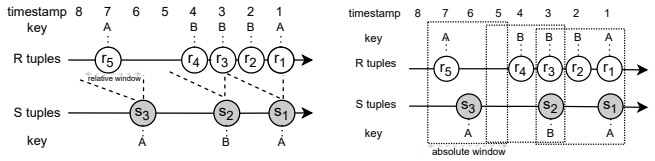
$w_i^{\bowtie} = \{< S_i, R_j > \mid w_i.start \leq R_j.timestamp \leq w_i.end$ and $R_j.key = S_i.key, \forall R_j \in R\}$

where $w_i = (S_i.timestamp-$`PRE`, $S_i.timestamp+$`FOL`). Finally the join results are aggregated, which will be used as features. The cardinality of aggregation results of $S \bowtie_{OIJ} R$ will be the same as that of the base stream S. It is important to note that the time window in `OIJ` is relative to the tuple in the base stream rather than an absolute time window in normal stream join [17].

Figure 3a illustrates an example of `OIJ` over two data streams with a time window of (-2s, 0), which means only the tuples arriving within 2 seconds relatively to the current tuple are concerned. The interval join results $S \bowtie_{OIJ} R$ are <s_1, r_1>, <s_2, r_3>, <s_2, r_4> and <s_3, r_5>, which are then aggregated per S_i to generate the final results <$s_1, agg(r_1)$>, <$s_2, agg(r_3, r_4)$> and <$s_3, agg(r_5)$>. By comparison, Figure 3b shows the common sliding window join with a window size of 3s and slide of 2s. In each slide window, every tuple from S is compared with every tuple from R on the equality condition of key without further checking on the timestamps. The slide window join results $S \bowtie_{slide} R$ (after removing duplicates) are <s_1, r_1>, <s_2, r_2>, <s_2, r_3>, <s_2, r_4> and <s_3, r_5>, where <s_2, r_2> is included as no relative window constraint is enforced for the sliding window.

The Current Solution: *Key-OIJ*. Parallelizing the streaming operations to improve performance by better exploiting modern hardware gained much traction recently [21]. To the best of our knowledge, `Key-OIJ` [4] is surprisingly the only available parallel algorithm of `OIJ`.

[4]In the context of stream processing, it would be `user` stream and `search` stream



(a) Interval Join (b) Sliding Window Join

Fig. 3: Interval Join and Sliding Window Join

For each new incoming tuple of either R or S stream, it is distributed to a specific working thread, which is called as `Joiner`, to conduct the *OIJ*. The distribution is based on the key's hash value, and each hash value is statically bonded to a certain Joiner. Simultaneously, the tuple is also stored in a buffer for future reference by the opposite stream. When a tuple is expired after a certain time (as determined by window length $|w_i|$ and lateness l), it will be cleaned from the buffer and is no longer possible to be joined with the opposite stream. As the key-based partition is the major component in achieving parallelization, we refer to this approach as *Key-OIJ* in the paper.

III. Methodology

In this section, we first present the research goals of this study. Next, we introduce our performance metrics and benchmark workloads, followed by the system specification.

A. Research Goals

This study aims to achieve scalable *OIJ* as a key extension to OpenMLDB. Specifically, we focus on reducing latency and improving the throughput of conducting *OIJ* on modern multicore servers. Two steps are involved to achieve the goal. First, we experimentally examine the existing solution – *Key-OIJ*. The common design space and critical pitfalls are revealed through a careful profiling study. Second, we investigate approaches to resolving the revealed bottlenecks and propose a novel approach, namely *Scale-OIJ*. The *Scale-OIJ* is further evaluated in comparison with existing solutions.

B. Performance Metrics

Throughout this study, we focus on two important performance metrics of streaming applications. First, *throughput* represents the overall processing efficiency. It is defined by the number of input tuples processed per second. Second, *latency* indicates the duration between the arriving of the tuple and the generation of its corresponding output. To rule out the contingency, we report the average value of throughput and cumulative distribution function (CDF) of latency.

C. Benchmark Workloads

We evaluate four real-world proprietary workloads in our experiments, of which there are two workloads (Workload A and D) in the logistics sector and two workloads (Workload B and C) in the retail sector. Table II summarizes our evaluated

TABLE II: Benchmark Workloads

Workload Name	Arrival rate v	Number of Keys u	Window length $\|w\|$ (s)	Lateness l (s)
Workload A	120 K/s	5	1	1
Workload B	200 K/s	111	150	10
Workload C	∞	45	8	100
Workload D	15 K/s	5	1	2

TABLE III: Server Specification

Component	Description
Processor	Intel(R) Xeon(R) Gold 6252 CPU (24 cores $\times$ 2 HyperThreading)
L3 cache size	35.75MB
Memory	384GB
OS & Compiler	CentOS Linux 7, compile with g++ 8.3.1

benchmark workloads. The datasets may not be fully sorted, hence we use lateness l to represent the degree of disorder of the dataset in order to achieve 100% accuracy [5].

- **Workload A.** This workload has medium window size and disorderliness (i.e., lateness). There are about 4000 matching elements in each time window and 400 elements in the range of lateness. There are 5 unique keys in this workload, meaning that it can be divided into at most 5 partitions according to the *Key-OIJ*.
- **Workload B.** The second workload has medium number of unique keys and disorderliness, yet its window size is large, with estimated 6000 matching elements in each time window.
- **Workload C.** The number of unique keys and window size are medium in this workload. For each time window, there are about 300 elements matched. However, it has large disorderliness. We has to store extra 10,000 elements on average to get accurate results.
- **Workload D.** The data distribution in this workload is similar to Workload A. However, it has relatively low arrival rate of 15 K/s.

D. System Specification

We implement *Key-OIJ* from scratch in C++. We have validated that our implementation significantly outperforms the original Java-based implementation in Flink. We conduct all our experiments on a recent multicore server with the Intel Xeon Gold 6252 processors, as specified in Table III.

IV. The Pitfalls of Existing Solutions

In this section, we present the performance evaluation results of applying *Key-OIJ* to different real-world use cases to reveal its shortcomings. We experimentally tune *Key-OIJ* to its best achievable performance in each case.

A. Overall Evaluation Results

Throughput. Figure 4 shows the throughput of *Key-OIJ* with varying numbers of Joiner threads under four workloads. We have the following key observations. First, the *Key-OIJ*

[5]In existing OpenMLDB applications, it is assumed that the aggregation must be exactly accurate.

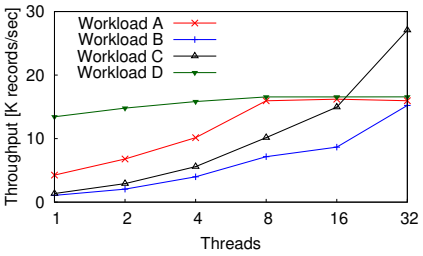


Fig. 4: Scalability under Four Real-world Cases

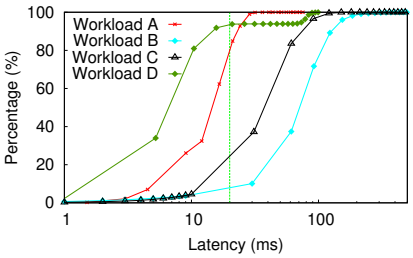


Fig. 5: Latency Distribution under Four Real-world Cases (green dashed line indicates the 20 ms latency required by a bank user of OpenMLDB)

fails to scale up with increasing core numbers when the number of keys is small (Workload A). Such ineffectiveness is mainly caused by the key-based partition method of `Key-OIJ` (Section II-C). Specifically, when there are only 5 keys in Workload A, at most 5 Joiner threads will be allocated. Second, the throughput of Workload B is much lower, as a larger window requires more time for data scanning and aggregation. Third, `Key-OIJ` scales relatively well on handling Workload C, but its throughput is far lower than that of Workload A when there are few cores. This is likely caused by the useless visit of data outside the window, which will be investigated further in Section IV-B. Fourth, when the input arrival rate is low (Workload D), `Key-OIJ` with a small number of cores can reach the maximum throughput, which is close to the arrival rate.

Latency. Figure 5 illustrates the cumulative density function (CDF) of `Key-OIJ` on handling various benchmark workloads under 16 join threads. `Key-OIJ` performs relatively well in Workload A and D, with 80%-90% below 20 ms, which is generally the requirement by our users (as shown in the green vertical line). However, it fails to deliver satisfying latency in workloads B and C, although their throughput scales well.

Time Breakdown. To further comprehend the ineffectiveness of `Key-OIJ`, we break down its processing time shown in Figure 6. We systematically categorize the time spent in running the `Key-OIJ` as the following components: 1) lookup time, which is the time spent on visiting all stored tuples to obtain those tuples in the time window; 2) match time, which is the time spent on doing aggregation with tuples in the time window; 3) other overhead, which is the time spent on other related processing such as writing the results, structure initialization, etc. From the breakdown, we can see that time is spent differently on "match" and "lookup"

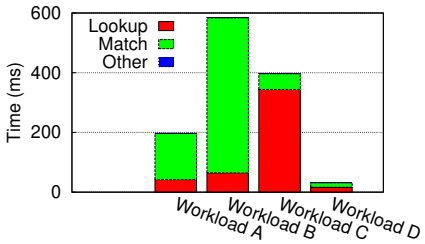


Fig. 6: Time Breakdown under Four Real-world Cases

TABLE IV: Default Workload

Parameter	Value
Key Number u	100
Window Size $\|w\|$	1000 us
Lateness l	100 us
Joiner Thread	16

among workloads. Specifically, when the window size is large (Workload B), match time becomes more dominant, as even more time is spent on doing window aggregation. On the contrary, when there is a medium window size but large lateness (Workload C), lookup time is much more than match time. This is caused by the more out-of-order tuple arrival, and `Key-OIJ` has to visit more tuples for each interval join.

Finding (1): `Key-OIJ` leads to poor throughput when 1) there are few keys and 2) the window is large. Furthermore, it leads to poor latency when the lateness is large.

B. Workload Sensitivity Study

We now use a synthetic dataset to further evaluate `Key-OIJ` by tuning its workload configurations. The default workload characteristics are summarized in Table IV.

Impact of Lateness l. We investigate the effect of tuning the lateness setting of query in Figure 7. The throughput of `Key-OIJ` drops rapidly with increasing lateness. This is because, with high lateness, `Key-OIJ` has to keep more tuples in the buffer in case we miss tuples that arrive too late. For every interval join operation, we have to visit all data stored in the buffer to filter out the tuples within the time window, as the data is not sorted in the `Key-OIJ` design. Hence, high lateness results in more tuples stored in memory and more time spent visiting all tuples.

We define *effectiveness* as the average ratio between the number of tuples within the time window and the total number of tuples visited as shown in Equation 1.

$$effectiveness = \frac{\sum_i^{S_i \in S} \frac{|w_i^R|}{|\bar{R}|}}{|S|} \quad (1)$$

where w_i^R is defined in Definition 2 and $\bar{R}$ denotes all the tuples in R that have to be visited to filter out the ones within the time window.

As shown in Figure 7, *effectiveness* decreases with increasing lateness, as extra tuples are buffered and more data has to be visited ($|\bar{R}|$ becomes larger). This correlates with the throughput trend, which reaffirms our analysis.

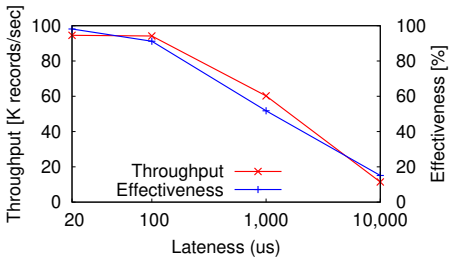
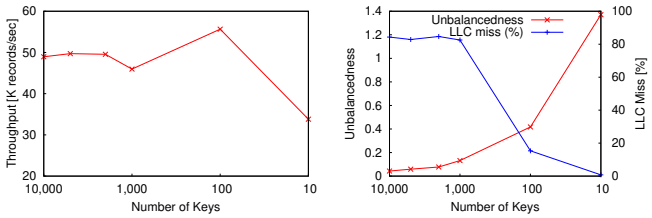


Fig. 7: Lateness Effect



(a) Throughput (b) Unbalancedness and LLC miss

Fig. 8: Key Number Effect

Impact of Number of Unique Keys u. Figure 8a shows the impact of varying number of unique keys u in workloads. There are two major observations.

First, as expected, throughput decreases fast when the number of unique keys decreases due to the unbalanced workloads distributed to the Joiner thread. We define the $unbalancedness$ as the standard deviation of workloads of all Joiner threads in Equation 2,

$$unbalancedness = \frac{1}{J \times \mu} \sum_{i=1}^{J} (W_i - \mu) \qquad (2)$$

where W_i is the workload of joiner J_i, μ is the average workload per joiner (i.e., $\mu = \frac{1}{J}\sum_{i=1}^{J} W_i$), and J is the number of joiners. As presented in the red line of Figure 8b, $unbalancedness$ increases significantly with less number of keys, leading to a decreasing of throughput.

Second, interestingly, there is a significant throughput increase when K decreases from 1000 to 100. We find that the root cause is the significant last-level cache (LLC) misses when there are many unique keys in workloads, due to its random access patterns when reading tuples from the buffers. LLC misses decrease when there are fewer keys as shown in the blue line of Figure 8b, which explains the throughput increase at these settings.

Impact of Window Size $|w|$. Lastly, we show the impact of tuning the window size in Figure 9. We can see that the throughput of `Key-OIJ` drops tremendously when enlarging the window. This is because there are more tuples that are within the window, taking more time for data reading and aggregation computing. Although as the window becomes larger, the overlapping area among neighbour windows becomes greater, `Key-OIJ` fails to take advantage of the overlapping results, thus causing significant redundant computations.

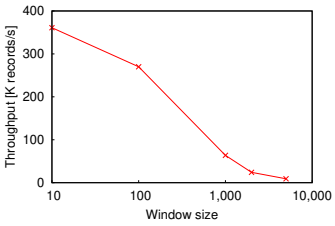


Fig. 9: Window Size Effect

Finding (2): The unsatisfying performance of `Key-OIJ` is caused by 1) its poor design in handling out-of-order data, 2) its inefficient workload distribution, and 3) its superfluous processing of overlapping windows.

V. Optimized Design of Interval Join

Based on the findings in Section IV, we propose an optimized design for `OIJ` in this section. Overall, we follow the key-based partition model of `Key-OIJ`; however, we allow to re-partition the data dynamically based on the workload distribution and shared processing of the same partition. By carefully designing the data structures and concurrency model, we propose a light-weight dynamic schedule algorithm to achieve high balancedness adaptively without data replication or migration. In particular, we design a time-travel data structure (Section V-A) to facilitate the efficient retrieval of the window data; Moreover, the time-travel data structure is designed to allow single-writer-multiple-readers (SWMR), which enables the multi-threaded shared processing for the same key. The unbalancedness observed in Section IV-B motivates the dynamic balanced workload schedule (Section V-B), which is able to distribute the workload evenly to the joiners based on the dynamic workload statistics. As also shown in Section IV, a larger window size, which is common in industry scenarios, indicates more data retrieval and computation, thus decreasing the throughput significantly. However, the large window also allows the possibility of window overlapping, which means two neighbour window aggregations may share a large proportion of work. Thus, it gives us an opportunity to optimize the performance by doing the aggregation incrementally, eliminating duplicate data retrieval and computation for the overlapping portion.

A. Time-Travel Data Structure

1) Ordered Indexing: To facilitate efficient window data retrieval, we carefully design an efficient time-travel data structure based on a double-layered skip-list, as shown in Figure 10. Basically, the first layer is a skip-list used to store the pairs of <key, second-layer skip-list>; and each component of the second layer is a skip-list used to store the pairs of <$timestamp$, $Tuple$>. To locate the tuple denoted as <key, $timestamp$>, we first search the first layer to get the second-layer skip-list containing all the tuples with the key key; then we search the second-layer skip-list to locate the tuple.

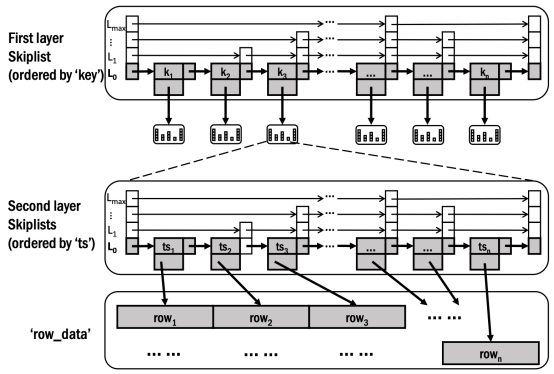


Fig. 10: SWMR Double-Layered Skip-List

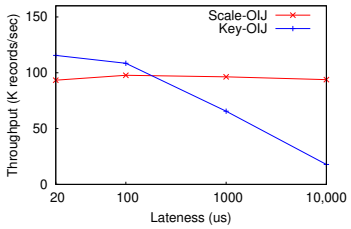


Fig. 11: Lateness

The time-travel data structure makes it possible to locate the window boundary in $O(logN_{key}) + O(logN_{ts})$ where N_{key} is the number of unique keys and N_{ts} is the number of timestamps per key. And only the effective window data will be visited. By contrast, the baseline method has to travel all the data to filter out the ones within the window boundary. Hence, when the degree of out-of-order is high, i.e., *lateness* is large, the baseline method performs much worse as we find in Section IV-B.

We repeat the *lateness* experiment with the addition of our optimization. As the *lateness* increases, the *effectiveness* of full data scan is decreasing, meaning that more data access and filtering are useless. For `Key-OIJ`, full data scan is a must as data is not ordered; while for `Scale-OIJ`, we can utilize the time-travel data structure to locate the window boundary directly and only access the necessary data within the window. As we can see from Figure 11, by enlarging the *lateness*, the performance of `Key-OIJ` decreases, while `Scale-OIJ` almost does not change.

2) SWMR Concurrency Property: In addition to the ordering property of the time-travel data structure, it is carefully designed to be lock-free in the multi-threaded environment. Specifically, it supports simultaneous writing by a single writer and reading by multiple readers without the use of lock. Algorithm 1 details the steps to *Search* a tuple of one layer of skip-list. This is a general process of searching in a skip-list, except that the reading of nodes follows the *Release-Acquire ordering* (**load_acquire**). Algorithm 2 demonstrates the steps to *Put* a new tuple. Line 1 to Line 11 are to find the position after which we insert the new tuple. Line 12 to Line 16 are to do the insertion atomically. The pre array (Line 6) will store all the previous nodes before the new node,

Algorithm 1: Search A Tuple

Input: <key, ts>
Output: the node containing the matched tuple

```
1  node ← HEAD
2  level ← HEIGHT
3  while true do
4      next = load_acquire (node[level].next)
5      if next == NULL || next.key > key then
6          if level <= 0 then
7              return node
8          level = level − 1
9      else if next.key == key then
10         return node
11     else
12         node = next
13 return node
```

Algorithm 2: Put A New Tuple

Input: <key, ts, x>
Output: N.A.

```
   // search the position to insert
1  node ← HEAD
2  level ← HEIGHT
3  while true do
4      next = node[level].next
5      if next == NULL || next.key >= key then
6          pre[level] = node
7          if level <= 0 then
8              break
9          level = level − 1
10     else
11         node = next
   // insert into the skiplist
12 new_node = NewNode(x, random_height)
13 for i ← 0 to height do
14     store_relaxed(new_node[i].next, pre[i].next)
15 for i ← 0 to height do
16     store_release(pre[i].next, new_node)
```

that is, the nodes in all the levels that are just not greater than the new node. We create a new node with random height, which should be less than the maximum height $HEIGHT$. First we'll update the $next$ pointers of the new_node to the $next$ pointers of the pre array with *Relaxed ordering* (Line 13 - Line 14), up to this point, there is no state change visible in the system, as there is no link from $HEAD$ to the new node. Afterwards, we will update the next pointers of pre array to the new node with *Release-Acquire ordering* (Line 15 - Line 16), until when the new node is atomically visible to the readers.

For the double-layered skip-list, the algorithm is the same for each layer, except that for the first layer, the key is the tuple key; while for the second layer, the key is the timestamp. This SWMR property is the core technique of enabling the shared interval join processing of the same key, which will be elaborated in Section V-B.

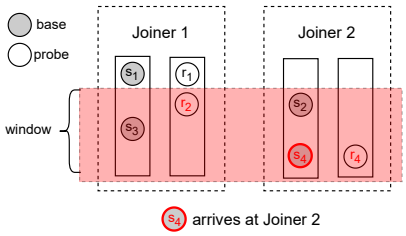


Fig. 12: An Example of Shared Processing

Finding (3): Time-travel data structure eliminates the unnecessary data retrieval of out-of-window data, making the *lateness* insignificant to the performance.

B. Dynamic Schedule

As we study in Section IV-B, when the number of unique keys is small, the performance of `Key-OIJ` is worse, due to the skewed key partition. That is, some Joiners may get more workload, while some get less, thus degrading the performance and hurting the scalability. Thus we design a dynamic schedule framework that allows partitioning the keys dynamically according to the workload. Moreover, it enables the shared processing of the same keys by multiple Joiners simultaneously.

1) Shared Processing: We design a shared processing framework to make it possible to process tuples of the same key by multiple joiners. The joiners sharing the workload of the same key form a *virtual team*. For the tuples with this key, it can be randomly distributed to any member of the *virtual team* without affecting the correctness. Joiners expose the *read* (**R**) accessibility to all the members within their *virtual teams*; while they keep the *write* (**W**) accessibility exclusively to themselves. The virtual teams are maintained via a mapping between the key (in particular, key hash range) and the partition set, which will be atomically replaced after a new schedule. Virtual team membership implicitly enforces the grant of **R/W** permission. For every tuple assigned to the joiner, it will insert (**W**) into its own index; while doing the interval join, it can read (**R**) the indices from all the members of its *virtual team*. Hence each joiner of a *virtual team* will share a portion of tuples under the same key. As elaborated in Section V-A, the SWMR property of our time-travel index guarantees the correctness and effectiveness of the shared processing framework.

We use Figure 12 to show the basic workflow of shared processing a base tuple. When the base tuple s_4 arrives at Joiner 2, it's going to traverse all the probe tuples within the window, i.e., probe tuples r_2 and r_4, which are feasible as Joiner 2 has the read (**R**) accessibility of both its local tuple r_4 and its team tuple r_2.

2) Dynamic Schedule: The shared processing framework allows dynamically re-partitioning the data without data migration. By collecting the data distribution statistics in runtime, we can derive the workload distribution of all the joiners, based on which we can re-schedule the partition assignments (i.e., key partition schedule) periodically. The objective of the re-schedule is to reduce the $unbalancedness$ (as defined in Equation 2) of workload schedule among joiners.

Specifically, we formalize the problem as follows:

a) Problem Definition: Given P partitions and J joiners, assign each P_i to J_j, where each P_i can be assigned to multiple joiners $J_j \dots J_k$.

b) Objective: $\arg\min_S unbalancedness(S)$, where S is a key partition schedule. As during the partition scheduling, we don't have the knowledge of actual tuple distribution in the time ahead, we use Equation 3 to estimate the workload (i.e., the estimated number of tuples processed) of each joiner.

$$W_i = \sum_{k \in J_i} \frac{|\{x \mid key(x) = k\}|}{|vt_k|} \tag{3}$$

where $\{x \mid key(x) = k\}$ is the set of tuples with key k processed by all joiners and $|vt_k|$ is the size of the virtual team of key k. For each joiner, we aggregate all the shared workload for all keys that are currently processed by this joiner.

It is obvious that the problem is NP-hard; thus we give a heuristic solution in Algorithm 3. To avoid the data migration overhead, we only allow sharing the ownership of a partition rather than transferring to another joiner. As a result, the joiner based on the old partition schedule is guaranteed to be in the virtual team of the key based on the new partition schedule. This naturally solves the correctness of tuples in transmission buffer between partitioners and joiners during the schedule change. The basic steps are summarized as follows:

1) calculate the workload for every joiner and select the joiners with maximum workload J_{max} and minimum workload J_{min} (Line 3-4)
2) try to replicate the partition with the largest workload in J_{max} to J_{min} (Line 5-7)
3) if the $unbalancedness$ decreases by a threshold, we break out (Line 8-9) and repeat Step 1) - 2)
4) it stops exploring if there is no change in the new schedule after an iteration (Line 11-12)
5) the statistics will be decayed at the end (Line 13)

With the help of the dynamic schedule, we can achieve good scalability even under a very small number of keys, as shown in Figure 13a, where `Key-OIJ` scales worse due to the unbalanced workload schedule.

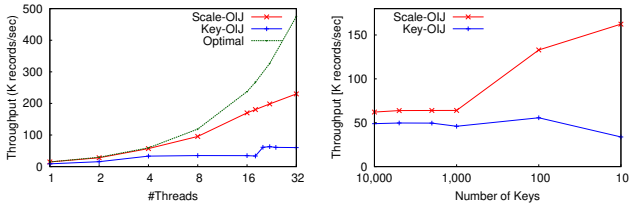
We further vary the number of unique keys using the synthetic dataset, as shown in Figure 13b. Our approach is able to adapt to all cases and achieve balanced processing even under the extreme case of a very small number of keys. However, the `Key-OIJ` that relies on the static key partition, fails to perform well if the number of unique keys is small. Interestingly, for our approach, under a small number of keys, the performance is even better than a large number of keys. We will investigate the cause later.

Figure 13c shows the $unbalancedness$ under a different number of unique keys. `Scale-OIJ` achieves very low $unbalancedness$ under all cases, while $unbalancedness$ is high under a small number of keys for `Key-OIJ`, which

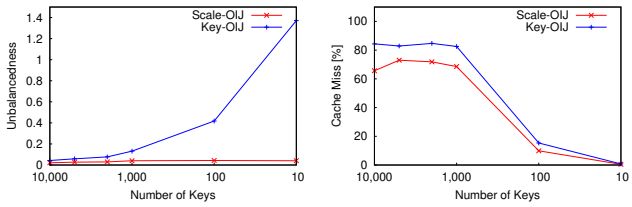
Algorithm 3: Dynamic Schedule

Input: Load distribution of all the keys, current key partition schedule S
Output: optimized key partition schedule

$S_{new} = S$
while *true* **do**
  calculate the workload W_i for every joiner J_i according to Equation 3
  select the maximum and minimum joiners: $J_{max} = \arg\max_i W_i$, $J_{min} = \arg\min_i W_i$
  add all key partitions $\forall p_j \in J_{max} \rightarrow$ priority queue $PQ_{J_{max}}$
  for $p_i \leftarrow PQ_{J_{max}}.top()$ **do**
    replicate p_i to J_{min} in the new schedule S_{new}
    if $last\_unbalancedness - unbalancedness > \delta$ **then**
      break
    $PQ_{J_{max}}.pop()$
  if S_{new} *does not change* **then**
    break
$\forall k \ |x_k| = \lambda \times |x_k|$
return the new schedule S_{new}



(a) Scalability under 10 Unique Keys (b) Throughput



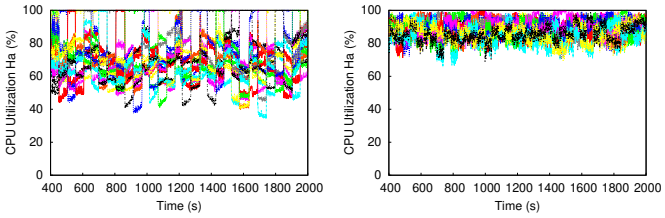
(c) *unbalancedness* (d) LLC Load Miss

Fig. 13: Performance under Different Number of Keys

explains why the performance is low under the case of 10 keys for `Key-OIJ` in Figure 13b.

Interestingly, as we can see from Figure 13b, there is a performance drop from 10 to 1000 for `Scale-OIJ` and 100 to 1000 for `Key-OIJ` (as we also observed in Figure 8a), which, however, is not consistent with the non-increasing trend of *unbalancedness* as depicted in Figure 13c. The reasons are two-fold:

- As the number of keys increases, the overhead associated with each unique key is increasing, such as individual data structure used to maintain each key, the work scheduling overhead, etc.
- The other cause is the increase of cache miss with the increase of the number of keys, as we have to access more data, which is estimated as $\#key \times window$. This is further verified in Figure 13d. The LLC miss surges greatly as the number of keys changes from 100 to 1000, which coincides with the performance drop illustrated in Figure 13b.



(a) `Key-OIJ` (b) `Scale-OIJ`

Fig. 14: CPU Utilization for Skewed Workload

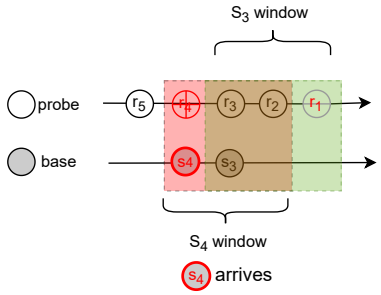


Fig. 15: Incremental Window Aggregation

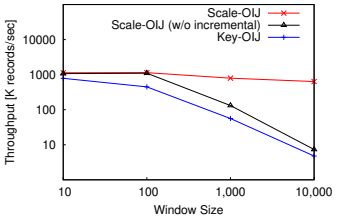


Fig. 16: Incremental Interval Join

Furthermore, we conduct an extra experiment on synthetic skewed streams where a random set of hot keys flow periodically with the same total number of records for every key. The number of unique keys is set to 10K, which is large enough to partition evenly even for `Key-OIJ` algorithm. Other parameters are set the same as Table IV. Figure 14 shows the variation of CPU utilization for the 16 joiners along time. It is obvious that with the dynamic schedule technique, `Scale-OIJ` can adapt promptly to the frequent workload change, revealed by the smoother CPU utilization variation than `Key-OIJ`.

Finding (4): By dynamically re-schedule the assignments, we can balance the workload of each joiner, thus improving the overall performance.

C. Incremental Online Interval Join

As we study in the real dataset, there is a high probability that neighbour windows would overlap during interval join, especially when the window is large, which incurs duplicate data access and computation. We adapt the existing *Subtract-*

on-Evict technique [16] to the interval join algorithm. In this paper we only focus on the invertible aggregation operators (e.g., sum, $count$, avg), which can be easily handled by the *Subtract-on-Evict* technique. For other non-invertible operators, some existing works [19] can also be adapted, which, however, is beyond the scope of the paper.

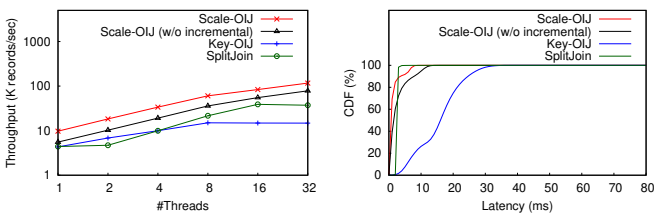
Basically, when a stale tuple is evicted from the window, we do a *Subtract* $\ominus$ on the running aggregate; when a new tuple is added into the window, we do a *Add* $\oplus$. For example, as shown in Figure 15, when we are handling the interval join aggregation for s_4, we can re-use the window aggregation Agg_{s_3} that is already calculated for s_3. The only extra work we have to do is to *Subtract* the value of r_1 which is out of window of s_4, and *Add* the value of r_4, which is not covered by the window of s_3. That is, $Agg_{s_4} = Agg_{s_3} \ominus r_1 \oplus r_4$. Hence we eliminate the data retrieval and computation of r_2 and r_3, which is significant if the overlap is large.

Figure 16 shows the throughputs under different window settings. With the help of the incremental interval join technique, our system can keep high throughput even with large windows.

Finding (5): By incremental online interval join, we are able to deliver high performance even when the window is large.

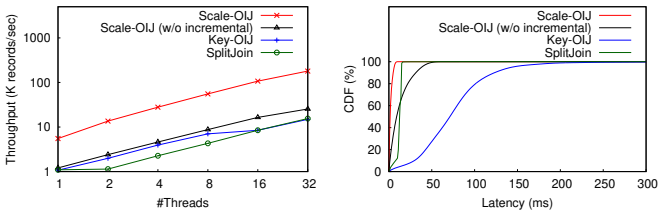
D. Experimental Results

We evaluate the performance of the four real-world workloads as described in Section IV by combining all the techniques we proposed in this section. We also adapt `SplitJoin` [12] to achieve the same semantics as `OIJ`. Basically, we follow their distribution and collection framework for parallelism, and add an extra predicate to filter out the tuples outside the relative window boundary for each join comparison. For Workload A, `Scale-OIJ` performs much better than `Key-OIJ` and `SplitJoin`, with latency less than 10 ms, as shown in Figure 17. `SplitJoin` also achieves a lower latency, which is even better than `Scale-OIJ` in terms of the maximum latency, as the round-robin workload distribution makes it more balanced. However, its throughput is far lower than `Scale-OIJ`, which is mainly caused by the heavy tuple broadcast traffic and the full data scan when doing the join operation. For Workload B, `Scale-OIJ` with incremental optimization improves the performance significantly, in terms of both throughput and latency as illustrated in Figure 18, as large window gives more opportunities of sharing computation results between neighbour windows. `SplitJoin` achieves even poorer throughput than `Key-OIJ` for small thread settings ($\leq$8), which is mainly because the balancedness advantage brought by `SplitJoin` is insignificant for low parallelism, while the overhead introduced by broadcast and "all-joiners-process-all-tuples" pattern dominates. We can see from Figure 19 that Workload C delivers a different message, that is, `Scale-OIJ` without incremental technique already boosts the performance greatly due to the elimination of unnecessary data access outside the window; while the incremental technique solely does not improve the performance much. `SplitJoin` achieves a similar throughput as `Key-OIJ` as they both suffer from the high cost of full table scan, especially when the out-of-window data is large. For Workload D, whose arrival rate is limited, `Scale-OIJ` can deliver lower latency than `Key-OIJ` and `SplitJoin`, with a similar throughput as



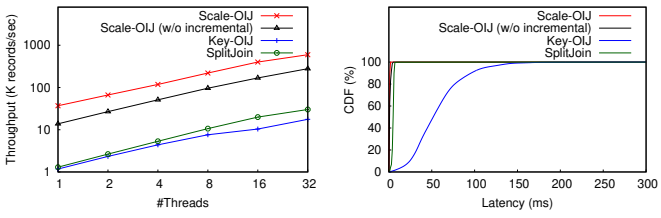
(a) Throughput (b) Latency

Fig. 17: Workload A



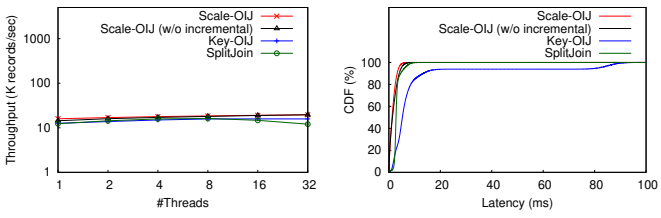
(a) Throughput (b) Latency

Fig. 18: Workload B



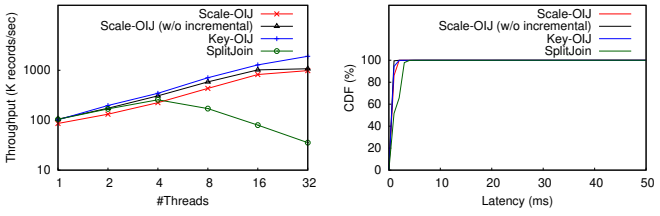
(a) Throughput (b) Latency

Fig. 19: Workload C



(a) Throughput (b) Latency

Fig. 20: Workload D



(a) Throughput (b) Latency

Fig. 21: Synthetic Workload

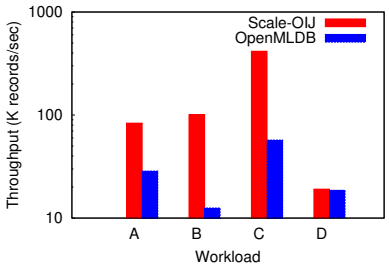
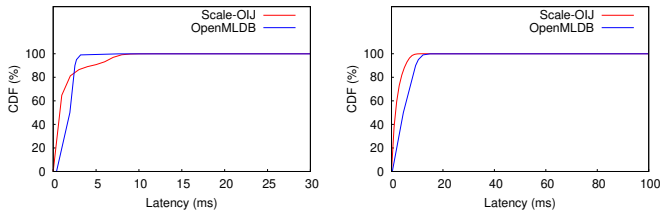


Fig. 22: Comparison with OpenMLDB (Throughput)



(a) Workload A (b) Workload B

(c) Workload C (d) Workload D

Fig. 23: Comparison with OpenMLDB (Latency)

shown in Figure 20.

We conduct another experiment with a synthetic workload to demonstrate the limitations of `Scale-OIJ`, as shown in Figure 21. The specification of this workload is shown in Table V, where the window w and lateness l is relatively small while the key number u is relatively large. In this workload, `Key-OIJ` performs the best as a large key number eliminates the skew in key distribution, resulting in balanced processing of all joiners. Small window partially negates the benefit from incremental processing as there is less overlapping between neighbour windows. Small lateness voids the time-travel data structure as most of the data is within the window and order does not speed up the processing. Interestingly, the performance of `SplitJoin` degrades after large threads, which is mainly because the overhead caused by tuple broadcasting over-kills the benefits brought by the balanced processing, and the small window setting deteriorates the problem as the computing is becoming even less than the data copying with large threads.

E. Comparison with OpenMLDB

In this section, we compare `Scale-OIJ` with OpenMLDB using the four real-world workloads. As OpenMLDB cannot handle the out-of-order cases, we remove the accuracy checking in OpenMLDB, thus eliminating the effect of lateness intentionally. Figure 22 and Figure 23 demonstrate

TABLE V: Specification of Synthetic Workload

Parameter	Value
Key Number u	1000
Window Size $\|w\|$	100 us
Lateness l	10 us

the throughput and latency results, respectively. For low-arrival-rate workloads (i.e., Workload D), OpenMLDB performs relatively well, even delivering lower latency than `Scale-OIJ`. However, OpenMLDB is not designed for streaming scenarios where the data arrival rate is high. In all other three workloads, `Scale-OIJ` is far better than OpenMLDB. Especially, it outperforms OpenMLDB by 8× and 7× when handling Workload B and C, respectively. For Workload B, the performance boost is mainly attributed to the incremental computing technique by `Scale-OIJ`, reducing the cost significantly of doing aggregation over the large window. For Workload C where the arrival rate is high, OpenMLDB cannot handle gracefully as the data insertions are often blocking. As we can see, OpenMLDB, which is even highly tuned for online feature computing, is still not able to handle streaming data at high arrival rate, as the assumption of read-intensive workload is no longer true for streaming data, and there is no effective mechanism to solve the out-of-order arrivals.

VI. Related Work

Works related to our approach can be broadly classified as follows: offline interval join and parallel online join.

Offline Interval Join. Interval-based join are popular in temporal databases, where each tuple has `left` and `right` endpoints representing an interval. Piatov et al. [13] proposed a gapless hash map to maintain the active candidate tuple set for cache efficiency and introduced lazy processing to reduce the repeated data scan. Dignös et al. [7] proposed *Overlap Interval Partitioning* to efficiently compute the interval join with a constant guarantee on the duration difference between tuple and its partition, and analytically derived the number of partitions k to balance the partition accesses and false hits. Chawda et al. [6] optimized the interval join in the MapReduce framework, and Bouros et al [5] proposed the forward scan method to conduct interval join, eliminating the costly maintenance of data structures like [13]. However, the "interval join" in their contexts is basically a relational join with a predicate checking that tuple intervals overlap, which has a different semantics from our `OIJ`. In addition, all their approaches require a total ordering by fully sorting, which cannot be achieved in the data streams where the tuples are continuously arriving and cleared.

Parallel Online Join. Online or stream join has received considerable attention in recent years due to its computational complexity and importance in various data management applications [20], [21]. To name a few, Gedik et al. [8] proposed the Cell join to exploit the computing power of the cell processor, which is the first work on parallelizing online join on modern hardware. Handshake join [17] and its low-latency improvement [14] propagate stream tuples along a

linear chain of cores in opposite directions to achieve scalable stream join. SplitJoin [12] achieves scalability in a different way. A top-down data flow model that splits the join operation into an independent `store` and `process` steps is introduced to reduce the dependency among processing units. Shahvarani et al. [15] addressed the challenge of index updating during window join by a partitioned in-memory merge-tree, which is shared among different threads for concurrent processing. However, unlike interval join, their window boundaries are 'absolute', irrelevant to the timestamp of each tuple in the input data streams. In contrast, the window boundaries of interval join are 'relative' to the timestamps of current tuples, making it non-trivial to partition the data based on the window.

VII. Conclusion And Future Works

This paper explores how to achieve scalable online interval join on modern multicores in OpenMLDB. We conduct detailed experimental studies with real-world and synthetic datasets on the only existing *OIJ* solution *Key-OIJ* [4]. Our results show that *Key-OIJ* leads to poor performance in three key aspects: 1) costly manipulation of out-of-order data; 2) unbalanced workload scheduling; 3) redundant computation of overlapping windows. We further propose a new approach called *Scale-OIJ* with 1) SWMR Time-Travel Data Structure, 2) Dynamic Balanced Schedule, and 3) Incremental Window Aggregation, to address each of the aforementioned poor designs. Our evaluation shows *Scale-OIJ* outperforms *Key-OIJ* by up to $24\times$ higher throughput and 99% lower latency, and improves the current version of OpenMLDB by up to $8\times$ higher throughput and 90% lower latency. It also performs better than the approach adapted from traditional stream joins (e.g., *SplitJoin* [12]) by up to $20\times$ in terms of throughput. *Scale-OIJ* has been partially integrated into OpenMLDB [6].

Streaming data at a high arrival rate is becoming common in online services [10], and hard real-time (e.g., 20 ms) of feature computing is often required in OLDA. However, traditional databases, even in-memory databases specifically optimized for speed like OpenMLDB, are not able to meet the requirements due to the streaming characteristics (e.g., high arrival rate, out-of-order arrival). Our proposal in this paper is an attempt towards making OpenMLDB able to handle high-rate data streams, focusing on the most common operator in machine learning databases - *OIJ*. There are many interesting open questions to further explore including but not limited to, NUMA-aware dynamic scheduling, tunable accuracy without prior knowledge (i.e., `lateness`), cache-conscious structures and scheduling strategies, prediction model for online data distribution, and incremental computing for non-invertible operators, which are not fully considered in this paper.

ACKNOWLEDGEMENT

Shuhao Zhang's work is partially supported by a SUTD Start-up Research Grant (SRT3IS21164).

[6]https://github.com/4paradigm/OpenMLDB/tree/stream

References

[1] A benchmark for real-time relational data feature extraction. https://github.com/decis-bench/febench. Last Accessed: 2023-01-03.

[2] Openmldb use cases. https://openmldb.ai/docs/en/main/use_case/index.html. Last Accessed: 2022-09-23.

[3] Real-time-experiment-analytics-at-pinterest-using-apache-flink, https://medium.com/pinterest-engineering/real-time-experiment-analytics-at-pinterest-using-apache-flink-841c8df98dc2. Last Accessed: 2020-11-23.

[4] Interval join in apache flink , https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/operators/joining.html, 2018. Last Accessed: 2020-09-17.

[5] P. Bouros and N. Mamoulis. A forward scan based plane sweep algorithm for parallel interval joins. *Proceedings of the VLDB Endowment*, 10(11):1346–1357, 2017.

[6] B. Chawda, H. Gupta, S. Negi, T. A. Faruquie, L. V. Subramaniam, and M. K. Mohania. Processing interval joins on map-reduce. In *EDBT*, pages 463–474, 2014.

[7] A. Dignös, M. H. Böhlen, and J. Gamper. Overlap interval partition join. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 1459–1470, 2014.

[8] B. Gedik, R. R. Bordawekar, and P. S. Yu. Celljoin: a parallel stream join operator for the cell processor. *The VLDB journal*, 18(2):501–519, 2009.

[9] Y. Ji, J. Sun, A. Nica, Z. Jerzak, G. Hackenbroich, and C. Fetzer. Quality-driven disorder handling for m-way sliding window stream joins. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 493–504, 2016.

[10] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl. Benchmarking distributed stream data processing systems. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1507–1518, 2018.

[11] S. J.-D. Lovelock, J. Hare, A. Woodward, and A. Priestley. Forecast: The business value of artificial intelligence, worldwide, 2017-2025. *Gartner.(ID G00348137)*, 2018.

[12] M. Najafi, M. Sadoghi, and H.-A. Jacobsen. Splitjoin: A scalable, low-latency stream join architecture with adjustable ordering precision. In *ATC*, pages 493–505, Denver, CO, June 2016. USENIX Association.

[13] D. Piatov, S. Helmer, and A. Dignös. An interval join optimized for modern hardware. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 1098–1109. IEEE, 2016.

[14] P. Roy, J. Teubner, and R. Gemulla. Low-latency handshake join. *Proceedings of the VLDB Endowment*, 7(9):709–720, 2014.

[15] A. Shahvarani and H.-A. Jacobsen. Parallel index-based stream join on a multicore cpu. In *Proc. SIGMOD*, SIGMOD '20, page 2523–2537, New York, NY, USA, 2020. Association for Computing Machinery.

[16] K. Tangwongsan, M. Hirzel, and S. Schneider. Low-latency sliding-window aggregation in worst-case constant time. In *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems*, DEBS '17, page 66–77, New York, NY, USA, 2017. Association for Computing Machinery.

[17] J. Teubner and R. Mueller. How soccer players would do stream joins. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 625–636, 2011.

[18] S. Wang, J. Li, M. Lu, Z. Zheng, Y. Chen, and B. He. A system for time series feature extraction in federated learning. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, CIKM '22, page 5024–5028, New York, NY, USA, 2022. Association for Computing Machinery.

[19] C. Zhang, R. Akbarinia, and F. Toumani. Efficient incremental computation of aggregations over sliding windows. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, KDD '21, page 2136–2144, New York, NY, USA, 2021. Association for Computing Machinery.

[20] S. Zhang, Y. Mao, J. He, P. M. Grulich, S. Zeuch, B. He, R. T. B. Ma, and V. Markl. Parallelizing intra-window join on multicores: An experimental study. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 2089–2101, New York, NY, USA, 2021. Association for Computing Machinery.

[21] S. Zhang, F. Zhang, Y. Wu, B. He, and P. Johns. Hardware-conscious stream processing: A survey. *SIGMOD Rec.*, 48(4):18–29, Feb. 2020.