Timestamped State Sharing for Stream Analytics

Yunjian Zhao, Zhi Liu, Yidi Wu, Guanxian Jiang, James Cheng*, Kunlong Liu, Xiao Yan
The Chinese University of Hong Kong
{yjzhao, zliu, ydwu, gxjiang, jcheng, klliu, xyan}@cse.cuhk.edu.hk

Abstract—State access in existing distributed stream processing systems is restricted locally within each operator. However, in advanced stream analytics such as online learning and dynamic graph analytics, enabling state sharing across different operators makes application development easier and stream processing more efficient. In addition, when stream records are timestamped, proper time semantics should be defined for both state updates and fetches. We propose a new state abstraction to address the limitations of existing systems and develop a distributed stream processing system, Nova, with native support for timestamped state sharing. We validate the expressiveness and efficiency of Nova with extensive experiments.

Index Terms—State Sharing, Distributed Stream Processing, Online Learning, Dynamic Graph Analytics

1 Introduction

DISTRIBUTED stream processing systems (**DSPSs**) are gaining more attention due to the popularity of online applications that continuously generate data and the demands for real-time data analytics to obtain insights for business decisions. Many DSPSs, e.g., Spark Streaming [1], Structured Streaming [2], Flink [3], Storm [4], Samza [5], Heron [6], MillWheel [7], Timely Dataflow [8], [9] and Differential Dataflow [10] are developed and used in industry.

This paper studies *state sharing in DSPSs*. Most existing DSPSs [1], [2], [3], [4], [9] describe streaming applications in dataflow graphs and support stateful operators. For scalability, an operator instantiates multiple instances, so that the entries of a state are partitioned among the operator instances and processed in parallel. This approach assumes that the processing of a stream record only needs to access the state entries in a local partition. Consider, for example, a *word count* application that counts the occurrences of each word, where the *state* (i.e., *(word, count)* pairs) is partitioned by the words (i.e., keys), and each operator instance updates the counts only for the words in its own partition. For such applications, state management is simple and existing DSPSs can achieve low latency and high throughput.

However, state sharing is more complicated for some advanced stream analytics such as online learning and dynamic graph analytics. Typical applications of online learning include online recommendation [11], click-through rate prediction [12] and contextual decision making [13], while dynamic graph analytics such as continuous RDF query [14], cycle detection [15] and community detection [16] have also been widely used in industry. For processing these analytics workloads, existing DSPSs suffer from various performance problems due to the following difficulties in state sharing.

First, different operators and their instances need to access multiple partitions of a state or states, instead of accessing only the local partition of a state. For examples, trainers (i.e., instances of an operator) in online learning need to access multiple parts of the model (i.e., the state), or different queries (i.e., operators) in graph analytics need to access the neighbors of multiple vertices. However, *existing DSPSs lack of an efficient mechanism to allow states to be concurrently shared across different operators and their instances*.

Second, states in these workloads are inherently associated with time information, which should be considered in both state access and storage. For example, different trainers access model parameters at different iterations (as logical timestamps), or different queries look for the neighbors of vertices within specific time periods. When different operators or instances access the same state, they may access the values with different timestamps. Moreover, access to the state entries may not follow the order of their timestamps. However, *existing DSPSs also lack of clear time semantics to guarantee the consistency and correctness of state access*.

There are alternative solutions, e.g., using an external storage to keep the global state, but we will show in §2 that these solutions are inadequate and address the state sharing problem in existing DSPSs by a new **system-wide state** abstraction (§3). Our contributions are as follows:

- We identify critical limitations of existing DSPSs for state sharing in processing an important class of stream analytics applications, e.g., online learning, dynamic graph analytics (§2).
- We propose a new state abstraction with flexible time semantics for state access (§3).
- We develop a DSPS called **Nova** that integrates our state abstraction with the popular dataflow model [17], [18] adopted in existing DSPSs, and propose system designs and optimizations for efficient state access and fault tolerance (§4).
- We evaluate Nova with extensive experiments on a variety of workloads including stream analytics benchmark, online learning, and real-time cycle detection to show the benefits and effectiveness of our design. Nova significantly outperforms alternative solutions for these applications (§5).

* *The corresponding author.*

2 Background and Motivation

In this section, we give the background of state management in existing DSPSs and discuss alternative solutions for state sharing. We also discuss related work.

2.1 State Management in existing DSPSs

Most existing DSPSs [1], [2], [3], [4], [5], [6], [7], [8], [10] adopt a dataflow model [18] and users construct dataflow graphs with various operators (e.g., *Map*, *GroupBy*) to process stream records.

Users inherit a class that has a state as its member variable or write a function whose parameters include a state, and then take the class or the function as the argument of an operator. Such an API ties a *state* to an operator, and we call such a state as **in-operator state**. Note that *Keyed State* in Flink, or states attached with the *mapWithState* and *updateStateByKey* operators in Spark Streaming, are also in-operator states. At runtime, an in-operator state is partitioned among the instances of its operator and an operator instance can only access its local partition of the state. State accesses within an operator instance are processed sequentially, and thus state management in existing DSPSs do not need to handle data races or consistency issues.

2.2 State Sharing and Possible Solutions

Many modern streaming data analytics workloads require *state sharing*. In online learning, model parameters (i.e., the state) need to be shared among the instances of an operator that runs the machine learning algorithm, where an instance may read or update any part of the model. For pattern matching in a dynamic graph, multiple query operators match different patterns and need to access different parts of the graph at the same time, while the graph itself is being updated continuously with an edge stream. In this case, the state (i.e., the graph) needs to be shared among both the operators (each of them is handling the matching of one specific pattern) and the instances (which are parallelizing the matching of a pattern) of an operator.

As a state can be concurrently accessed by multiple operators, state access consistency becomes an important issue. In particular, many streaming applications also operate on records with timestamps, which can be either physical (e.g., the time an edge is created in a dynamic graph) or logical (e.g., the iteration number in online learning), and thus *state access with time semantics* needs to be imposed. For example, in real-time cycle detection, a typical operation is to query the neighbors of a vertex within a time window W, in which case the *state access semantics* should ensure that a query should always return the correct result, i.e., not to miss any neighbor within W nor to return those outside of W, regardless of the time order by which the neighbors are updated (i.e., the updates may arrive out of order).

Read-shared in-operator state. Some DSPSs also provide a special type of in-operator state, e.g., *Queryable State* in Flink [3] and *TridentState* in Trident [19], which can be read by other operators. Such a state is partitioned by key among the instances of a "writer" operator, where a local partition is updated only by the corresponding "writer" instance. Other operators can perform *read-only* operations

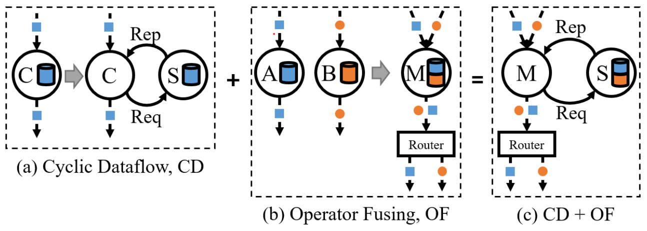


Fig. 1: Workaround solutions

to read the *latest* snapshot of the state, which is useful for serving ad-hoc queries to obtain immediate insights. However, such a state design fails to ensure consistent results as readers only read the latest updated state, but updates with timestamp earlier than the reads may actually arrive late (e.g., due to network packet jitter).

Workaround solutions. It is possible to implement some workarounds for state sharing in existing DSPSs, as shown in Figure 1 and explained below.

Cyclic Dataflow (CD) To share a keyed state among the instances of a single operator C, we can create a new *serving operator* S to manage the state, where C sends update/read requests to S, S processes the requests and sends the replies to C. The request ($C \rightarrow S$) and reply ($S \rightarrow C$) form a cylic dataflow and the requests/replies are delivered by shuffle operations. For example, FlinkPS [20] uses this workaround for machine learning workloads on Flink.

Operator Fusing (OF) To share a keyed state among two operators A and B, we can fuse A and B into a single operator M, where the states of A and B are stored in M, and their input records are co-partitioned and streamed to M. In this case, the processing of records from A and B that share the same key can write and read the local partition (with the key) of the state in M.

CD+OF While CD/OF is restricted to one specific state access pattern, we can combine them (i.e., **CD+OF**) to share keyed state among the instances of multiple operators: multiple operators are fused (by OF) into a single operator C in CD, while the state is managed by S in CD.

The above workarounds have the following drawbacks. For **OF**, it destroys the independence of operators and thus complicates the execution logic, since it requires the operators that access the shared state to be fused into a monolithic dataflow structure. The case becomes even more complicated when operators process different stream record types and have different processing logic. The fusion of operators also makes the implementation of applications more complicated and error-prone. For **CD**, it adds a great burden on users to handle the correct request delivery and state access semantics for different applications. For example, FlinkPS takes about 400 effective lines of code to implement the necessary semantics for online learning based on Flink and there is also performance problem as reported in §5.2. As for **CD+OF**, it combines CD and OF to allow state sharing among instances of multiple operators, but also inherits the drawbacks of both CD and OF (§5.3).

We list various state sharing solutions in Table 1. Although some of these solutions can support state sharing among multiple operators and operator instances, they all suffer from various drawbacks as we have discussed above. In addition, none of these solutions support well-defined

TABLE 1: State management with existing DSPSs

	State sharing among		State access semantics
	instances	operators	
In-operator state	-	-	-
Queryable state	✓	✓	-
TridentState	✓	✓	-
CD	✓	-	-
OF	-	✓	-
CD + OF	✓	✓	-
Our proposal	✓	✓	✓

state access with time semantics. As we will show in §3.4, such state access semantics is essential to guarantee the correctness of timestamped stream applications. We also remark that [14], [15] commented that it is beyond the expressiveness of today's popular DSPSs to handle stream analytics workloads such as those we consider in this paper. This motivates us to design a new state abstraction (§3) to support more expressive state sharing with time semantics, while preserving the popular dataflow model [17], [18].

2.3 State Management by External Systems

We may consider to use specialized external systems to manage states, while computation is performed in a DSPS. For example, we can use a DSPS with parameter server for online learning, and use Esper with Apache Jena for C-SPARQL workloads. This, however, leads to non-negligible overheads of cross-system communication and data transformation for state access [5], [14], [21]. As we will show in §5.2, such overheads cannot be ignored for stream applications as they usually require high throughput and low latency. Besides, a composite solution that uses two different types of systems adds extra burdens to users to implement the application logics and ensure state consistency and fault tolerance [22]. Depending on the fault tolerance mechanism of the DSPS and the external system, users need to either implement write-ahead logging inside the DSPS or modify the external system intrusively.

For more general applications, we can consider key-value stores or databases, e.g., Redis (an in-memory data store widely used as database, cache and message broker in industry), VoltDB (an in-memory database for processing multiple data streams) and InfluxDB (a time-series database with optimizations on storing and processing timestamped data). However, they suffer from the same problems as specialized systems. But unlike specialized systems, general systems lack semantics for state access, i.e., updates and queries are independent from each other and there is no guarantee that updates are applied before queries even if the updates happen before the queries. Thus, they rely on the DSPS to guarantee the correctness of state access. As we will show in §5.1, the lack of semantics limits the efficiency of state access.

2.4 Related Work

State sharing. SDG [23] is a computation framework that translates an imperative java program into distributed dataflow execution. Users can use special annotations to indicate two state access patterns, *Partitioned State* and *Partial State*. *Partitioned State* is designed for partition-able states but restricts state access to be inside a local partition. *Partial State* is designed for states that are not partition-able and require global aggregation. These two access patterns are not sufficient for advanced stream analytics workloads. For example, in dynamic graph analytics, a vertex typical needs to query its k-hop neighbors and thus needs to *actively access multiple partitions*. Such access patterns cannot be expressed by *Partitioned State* and *Partial State*. S-Store [24] extends H-Store, a main-memory OLTP system, to support OLTP on streaming data with ACID transactional semantics. States are stored in time-varying tables to be shared among transactions. Their design targets for transaction processing with ACID, while our target applications demand more for high throughput and low latency, for which a less strict state access semantics instead of ACID often provides a better guarantee on the throughput/latency requirements. Recently, shared arrangement [25] was proposed to support multi-versioned state sharing for streaming applications, which shares indexed state across operators with the semantics that is identical to maintaining individual copies of the state for each operator. However, it requires co-partitioning a state with stream records that access it, and is not general to the applications that require accessing multiple partitions of a state (e.g., online learning).

Stream processing systems (SPSs). The execution model of existing SPSs can be divided into two categories: one-record-at-a-time and micro-batch. Flink [3], Storm [4], Samza [5], MillWheel [7], Noria [26], Naiad [8], Timely Dataflow [9] and Differential Dataflow [10] all adopt the one-record-at-a-time model, where stateful operators process stream records from upstream, update their internal state and then emit results to downstream. Spark Streaming [1], Structured Streaming [2] and Trident [19] group streaming records into mini-batches. The computation of a mini-batch is structured as a set of deterministic, stateless partitioned tasks. The state is represented as a distributed immutable dataset and updated in a batch fashion. These SPSs lack an expressive state sharing mechanism with consistency guarantee to support the advanced stream analytics workloads discussed in §1 and rely on either workaround solutions (§2.2) or external storage to share states (§2.3).

2.5 Timestamps

The dataflow model [17] summarizes two time domains that are widely adopted in existing DSPSs, i.e., **Event Time** (the time when an event actually occurred) and **Processing Time** (the time when the event was observed by the DSPS during processing). When using **Event Time**, data records are assumed to carry event timestamps when entering the DSPS. The DSPS directly uses the carried event timestamps and the processing would yield consistent and deterministic results. As for **Processing Time**, it does not provide determinism for many reasons, e.g., the time differences among the machines, the speed for processing an event in the DSPS. In this paper, the applications we target at have strong requirements on the consistency and correctness of state access, which can be hardly satisfied using **Processing Time**. Thus, we mainly use **Event Time** as timestamps.

2.6 Watermark Management

Watermark [7], [17] is important in stream processing. As we use watermark to guarantee the correctness of the state access semantics in our system, we discuss watermark management as follows.

Stream records come into a DSPS by the source operators. A source operator forwards the records it receives to downstream operators and then emits a watermark w to its downstreams, which indicates that no more records with timestamp $t < w$ will be emitted to downstreams in the future. Any future record with timestamp $t < w$ will be dropped by the source operator, which is to ensure the timeliness of stream processing. Except the source operators, if an operator has multiple upstream operators, the operator maintains the latest watermark of all its upstreams and takes the minimum upstream watermark. The operator then emits a watermark as the minimum value between the minimum upstream watermark and the minimum timestamp of the records that are still being processed by the operator. Thus, in our discussion, *a watermark w emitted by an operator guarantees that records with timestamp smaller (i.e., earlier) than w have all been sent to the downstream operators, except those delayed records that are dropped by the source operators.*

3 Programming Model

We present our new state abstraction, its programming interface and time semantics in this section.

3.1 State Abstraction

We call the *state* in existing DSPSs as **in-operator state** (§2.1). We introduce a new state abstraction, called **system-wide state**, which is designed to work alongside with in-operator state. For simplicity, we use **State** to refer to **system-wide state** in this paper. Each entry in a State consists of a *key* and a *valueSet*, where *key* is a unique ID and *valueSet* is a set of (*timestamp*, *value*) pairs. Users may create multiple States to manage logically different sets of State entries. For each State, multiple instances can be created according to user-specified parallelism and the State is partitioned by the keys of its entries among the State instances.

3.2 Update and Fetch Operators

To manage States, we introduce two State access operators: (1) **Update**, which issues requests to update or add values of State entries at specific timestamps; (2) **Fetch**, which issues requests to fetch the values of State entries. We first define the two operators in this subsection and then give some examples of using them in §3.5.

The two operators process timestamped stream records and watermarks from their upstream operators. To process a stream record, a Fetch/Update operator may send *fetch/update requests* to and receive *replies* from the State. A Fetch/Update operation takes three inputs:

1) The *State* to be accessed;
2) A user-specified *request function* to construct a fetch/update request for a stream record;
3) A user-specified *fetch/update rule* for the State to process a fetch/update request.

Update Operator issues update requests to be sent to the State. An update request consists of a key K, a timestamp T and a new value V. Upon receiving an update request, the State processes it with the update rule and then sends a *reply* acknowledgment to the Update operator. If K does not exist, the State creates a new State entry with K as the key and $\{(T, V)\}$ as the *valueSet*. If K exists, a new (T, V) pair is inserted into the *valueSet* of the entry with key K (if a pair with timestamp T already exists, then it is replaced by the new pair).

Users may specify different update rules for their applications. For example, in online learning, gradients (i.e., new values) are carried by update requests to update the model parameters at each iteration, where the update rule is "adding the gradients to the parameters". For a dynamic graph, an edge addition (or deletion) $a \rightarrow b$ at time T appends a vertex addition *+b* (or a deletion *-b*) to the neighbor list of vertex *a* at T, where *+b* or *-b* is the new value.

Fetch Operator issues fetch requests to a State. A fetch request carries a key K and a reply timestamp T, where K is used by the State to locate the entry and T decides when the fetch request can be processed by the State according to the time semantics to be defined in §3.4.

The State processes a fetch request using the user-specified fetch rule. For example, for cycle detection in a dynamic graph, to fetch the neighbors of a vertex that match certain labels L within a time window $[T_1, T_2]$, we specify the vertex id as the key K and set the reply timestamp T to T_2. Then, we write a fetch rule to select neighbors that fall in $[T_1, T_2]$ and match the labels in L. The fetch rule is then executed at time $T = T_2$, i.e., all edges with timestamps before T_2 have been collected. In online learning, a fetch request fetches the model parameters at a specific iteration, where the fetch rule is used to adopt a computation model (e.g., BSP, SSP, ASP).

Comparison. The Update and Fetch operators are different from the state access APIs in existing DSPSs, e.g., `mapWithState` and `updateStateByKey` in Spark Streaming, which still belong to *in-operator states* and do not support state sharing. In contrast, our State abstraction supports multiple States to be shared among instances of multiple operators. Our design provides users the flexibility to control how different operators jointly share and maintain the shared States, as opposed to the alternative solutions presented §2.2-2.3. Compared with *update/get* operations in key-value stores, Update and Fetch are dataflow-level APIs used together with normal dataflow operators (e.g., *Map*, *Join*, *GroupBy*) of a DSPS. As dataflow-level APIs, they provide efficient and user-transparent system supports for asynchronous request processing, watermark management and fault tolerance.

3.3 Progress Operator

The Fetch and Update operations alone are not sufficient for State management for the following two reasons.

First, fetch/update requests may arrive out of time order because the stream records that generate these requests may arrive out of order and network packet jitter may also disorder these requests. Fetch/update requests generated by different operators may be interdependent. It is common that an update request, which updates a value at time T_1, may arrive later than a fetch request, which looks for a value at time T_2, where $T_1 < T_2$ (i.e., update at T_1 should happen

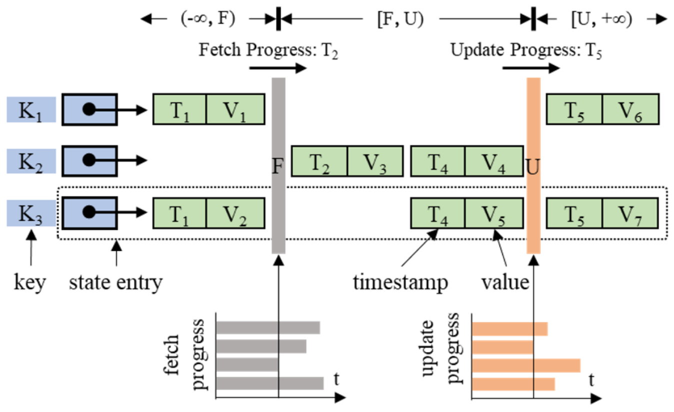


Fig. 2: Time semantics of State access

before T_2). We need to ensure that the update request at T_1 is applied before the fetch request at time T_2 is replied. To this end, we introduce **update progress** U, which indicates the progress of updates to a State, so that the State can tell whether it has applied all the update requests sent from all its Update operators before U. Thus, U denotes the boundary where the values of any State entry with timestamp less than U are final.

Second, as time goes by, update requests keep adding new values to existing State entries and creating new State entries. For stream applications, more attention is put on recent values and old values become less important. It is helpful to store old values compactly to improve fetch speed and reduce memory usage, e.g., a sequence of edge insertion(+) and deletion(−) $[+e_1, +e_2, +e_3, -e_2, -e_3]$ can be compacted into $[+e_1]$. To this end, we introduce **fetch progress** F, which indicates the progress of fetches from a State. We use F to trigger the compaction of values that have timestamp less than F (i.e., before F). Users can define different *compaction rules* for their applications. An example is given in §3.5.

To update U and F, we introduce the **Progress operator**. A Progress operation takes three inputs: the *State*, a *flag* specifying whether it is a fetch or update progress, and the current *progress*. Monotonically increasing progresses are to be sent to the State. The current progress is often just the latest watermark that the Progress operator has received from its upstream operators, as the watermark (say, w) serves to ensure that all records before w have sent their fetch/update requests and received all the replies. There could be multiple Progress operators, each reporting a different fetch/update progress f/u to the State. The State maintains the minimum value from the latest reported fetch/update progresses of the Progress operators as the **global fetch/update progress** F/U. Some examples of using the Progress operator are given in §3.5.

Discussion. The Progress mechanism follows the philosophy of watermark [7], [17] in existing DSPSs, where watermark helps trigger certain actions when all records before a logical timestamp have arrived. In our case, a State manages two types of independent "*watermark*"s simultaneously, i.e., update progress and fetch progress.

3.4 Time Semantics of State Access

Based on the global fetch/update progress F/U, we define the time semantics of State access requests. As shown in Figure 2, U and F logically divide the *valueSet* of each State entry into three ranges:

1) Within $[U, +\infty)$, values may still be updated.
2) Within $[F, U)$, values are final and no update request with timestamp $T < U$ will be received.
3) Within $(-\infty, F)$, values can be compacted.

With the above time ranges and their implications defined, we can check the validity of a fetch request, i.e., whether the request can be replied. Given a fetch request with reply timestamp T, the request can be replied as long as $T < U$. Users can customize their fetch requests according to the requirements of their applications. We show two typical types of fetch requests below.

Read-committed fetch is to obtain values within a time range $[T_1, T_2]$ (both endpoints can be exclusive or inclusive), where $-\infty \leq T_1 \leq T_2 < U$. The reply timestamp of the request is set to T_2, and the State replies the fetch request only when $T_2 < U$. For example, in cycle detection that finds newly generated cycles with maximum length c within the latest time window $W = [T - span, T)$, for each newly added edge (u, v), we fetch neighbors within c hops of the vertices u and v using read-committed fetch requests with time range $[T - span, T)$.

Read-uncommitted fetch is to obtain values within $[T_1, T_2]$ but the reply timestamp is set to T_1. This type of fetch allows users to fetch values that may be still undergoing updates, which is useful in applications that can tolerate relaxed consistency and are more interested in immediate answers. The condition $T_1 < U$ must be satisfied before the fetch can be replied, while T_2 may be smaller or larger than U. In case of $T_2 > U$, it means values within $[T_1, U)$ are final while those in $[U, T_2]$ are not.

Read-uncommitted fetch is particularly common in iterative optimization algorithms in machine learning. Consider mini-batch-based online training by Stale Synchronous Parallel (SSP) with staleness s. The parameters (i.e., the shared State) fetched by a worker for the i-th mini-batch should have been updated by the $(i - s)$-th mini-batch of all workers. Each worker sends its latest batch number to the State as its update progress and fetch progress, and issues read-uncommitted fetch requests with time range $[i - s, +\infty)$ to obtain the latest-updated parameters within the bound of staleness.

Discussion. Time semantics can also be added to an external storage system that manages states for a DSPS, but it requires a substantial amount of modification in order to organize the timestamped data, track update progresses to answer fetch requests correctly, and track fetch progresses for compaction of old values. There is also non-negligible cross-system communication overhead that is critical to latency-sensitive stream applications.

3.5 Advertising Campaign Stream Analysis

We illustrate the use of State using an *advertising campaign stream analysis* (*AdCamp*) application from a large

```
1  dym_table = make_state<Key, Time, Val>(); // Key is advertisement id, Val is campaign id.
2
3  dym_table.compaction_rule = (Entry e, Time F) { // `F` is the global fetch progress
4    timestamped_camps = e.values_within((-inf, F));
5    // remove all campaigns except the lastest one with timestamp before F
6    if (not timestamped_camps.empty()) { e.erase_until(timestamped_camps.last().time); }
7  };
8
9  update_stream.update(dym_table,
10     /* request func */ (Update u) { return UpdateRequest(/*key*/ u.ad, /*time*/ u.time, /*args*/ u.camp); },
11     /*  update rule */ (Entry e, UpdateRequest r) { return Pair(/*time*/ r.time, /*new value*/ r.camp); })
12   .progress(dym_table, /*flag*/ UPDATE_PROGRESS,
13     /*progress func*/ (T upstream_watermark) { return upstream_watermark; });
14
15 event_stream.fetch(dym_table,
16     /*request func */ (Event e) { return CommittedFetchRequest(/*key*/ e.ad, /*time*/ e.time); },
17     /*  fetch rule */ (Entry e, CommittedFetchRequest r) { return e.values_within((-inf, r.time)).last().camp; })
18   .progress(dym_table, /*flag*/ FETCH_PROGRESS,
19     /*progress func*/ (T upstream_watermark) { return upstream_watermark; })
20   .map((Pair<Event, Campaign> e_with_camp) { /* compute and return the effectiveness of `camp` */ });
```

Listing 1: Code snippet for the advertising campaign stream anslysis application

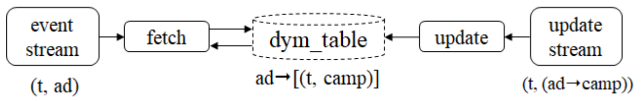


Fig. 3: AdCamp description

online shopping platform. The application consists of an "*event*" stream that emits "user viewed advertisement *ad* at time t_a" records, and an "*update*" stream that emits "*ad* belongs to promotion campaign *camp* starting at time t_c" records. The "*update*" stream models a dynamic mapping, which records that an *ad* at a specific time belongs to which campaign *camp*. At any specific time, an advertisement is only assigned to exactly one campaign. The purpose of AdCamp is to analyze how effective a promotion campaign is.

Figure 3 describes the AdCamp application and Listing 1 gives the code snippet. We create a State, called *dym_table*, to keep the *dynamic mapping* from *ad* to *camp* at time *t* (line 1). The "*update*" stream uses the Update operator to update the entries in *dym_table*, which adds a pair (*t*, *camp*) to the entry with key *ad* (lines 9-11). There is also a Progress operator (hidden in Figure 3 for simplicity) that reports the upstream watermark (i.e., the watermark of the applied updates) as update progress to the State (lines 12-13).

For each "*event*" record with timestamp t, the Fetch operator constructs a read-committed fetch request with time range $(-\infty, t]$, where *ad* is the key and the fetch rule returns the latest value within $(-\infty, t]$ (lines 15-17). That is, the fetch returns the campaign *camp* that *ad* belonged to at time t. Similar to update progress, another Progress operator is used to report the watermark of answered events as the fetch progress to the State (lines 18-19).

We also use the compaction rule, which retains the latest value before F for each State entry and all earlier values are deleted (lines 3-7).

Here, the compaction only needs to retain the latest value before F because for any incoming "*event*" record "user viewed *ad* at time t", the definition of F implies $t \geq F$. Thus, the mapping (*ad*, *camp*) must have a timestamp t_1 such that either (1) $F \leq t_1 \leq t$, or (2) $t_1 < F \leq t$. For Case (1), the value is not affected by the compaction. For Case (2), the fetch rule returns the latest value before t, which is just t_1. We also remark that the values within the range $[F, U)$ should not be compacted because all values (not just the latest one) within this time range may still be fetched by an incoming stream record.

Compared with the workaround solutions in §2.2, our State abstraction and operators provide a general programming framework for timestamped state sharing, where the details of State operations, correctness and optimizations are transparent to users.

4 System Design & Implementation

Overview. We developed a DSPS, called Nova, to implement the State abstraction and integrate it with the popular dataflow model [17], [18]. Nova follows the master-worker architecture. The master distributes workloads to workers, monitors the liveness of each worker, and initiates system checkpoint. Workers run multiple executors concurrently. As Figure 4 illustrates, each executor either (1) processes stream records on a subgraph of a user-defined dataflow graph, or (2) manages State entries and handles State access requests both from local and remote executors.

We implemented Nova using the C++ Actor Framework (CAF) [27], which provides an actor model layer, where actors are a set of concurrent entities that communicate via asynchronous message passing. Each executor in Nova is an actor in CAF. We translated dataflow graphs to actor execution and implemented various system components for distributed computation with fault tolerance and exactly-once processing guarantee.

Users construct a dataflow graph using the State access operators (i.e., *Fetch*, *Update* and *Progress*) and common dataflow operators (e.g., *Map*, *Join*, *Shuffle*, etc.). Nova splits a dataflow graph into subgraphs, where two connected operators are placed in different subgraphs if and only if the data movement between them results in *wide dependency* [28], i.e., a many-to-many mapping from the instances of an upstream operator to the instances of a downstream operator. Wide dependency is common for operations such as *Shuffle* and *Broadcast*. *Source operators* have no upstream and fetch data from data sources (e.g., Kafka) and then emit records to its downstream. *Sink operators* have no downstream and send records to other systems for durability or further processing.

To start the execution, a worker instantiates a subgraph on multiple executors according to user-defined parallelism,

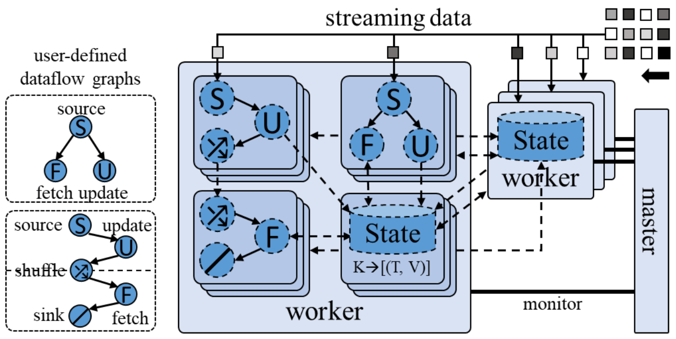


Fig. 4: On the left are two dataflows. On the right it shows their execution at runtime, where each of the rounded rectangles inside a worker represents an executor. A solid (dash) circle represents an operator (instance).

and the executors process different partitions of records in a data parallel manner. State instances also run on separated executors. The entries of a State are partitioned among the State instances. Each State instance manages one partition of the State and processes update/fetch requests and progress messages from State operator instances.

Although Nova itself is a DSPS, the focus of this paper is not on stream processing and thus in the following sections we only discuss the system components and techniques used for efficient State operations, message delivery and fault tolerance for State access.

4.1 State Operations

To process a stream record, an Update/Fetch operator may send one or more update/fetch requests to the State. The Update/Fetch operator waits for *all* the replies of a stream record before forwarding the record (and the fetch replies) to its downstream operators. The *out-of-order processing* paradigm [17], [29] is adopted here, which forwards a record as soon as all of its replies are received, without waiting for any preceding record. This enables efficient *asynchronous* processing on the stream records.

Upon receiving a fetch request, the State first checks whether the request can be replied according to the time semantics described in §3.4, i.e., whether the reply timestamp T of the request is less than the global update progress U. If yes, the request is processed with the fetch rule and the result is sent back to the requesting Fetch operator immediately. Otherwise, the State buffers the request in a priority queue, called the *pending queue*, where requests are sorted by their reply timestamps. When the update progress U moves forward, the State checks and processes the requests starting from the head of the pending queue until meeting a request that cannot be replied.

Upon receiving an update request, the State finds the entry according to the key specified in the request and then processes the update using the update rule. After that, the State sends an update acknowledgment back to the requesting Update operator to indicate that the update has been applied.

We next discuss optimizations for State operations.

Prioritizing replies. An Update or Fetch operator needs to process both stream records (from upstream operators) and replies (from the State). One critical issue is *delayed replies* due to head-of-line blocking, i.e., the replies are queued after a long list of new stream records while the downstream operators are waiting for the replies, especially when records are flowed in small batches. The blocking destabilizes the end-to-end latency and the situation can get worsen in cycles if the blocking is not handled. Thus, we prioritize replies over stream records, so that (1) the delivery of replies have higher priority than new stream records, and (2) an Update or Fetch operator always processes replies (if any) before processing new stream records and generating new requests.

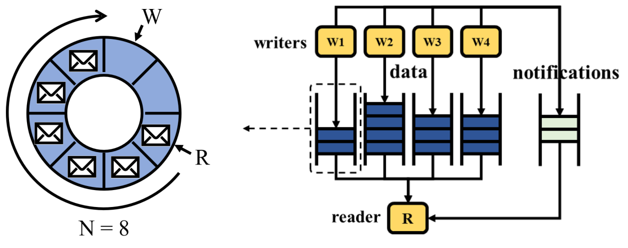
Update progress acceleration. In most cases, an update Progress operator is *a direct downstream* of an Update operator. Let `P` be the Progress operator and `U` be the Update operator. When `U` emits a watermark w to `P`, it means that `U` has received (from the State) all the replies for its update requests that have timestamps smaller than w. `P` then reports w as the update progress to the State. Let t_1 be the time when `U` has sent (to the State) all its update requests that have timestamps smaller than w, and t_2 be the time when `U` emits w. There is a gap between t_1 and t_2. As w is used to update the global update progress, the larger $(t_2 - t_1)$, the greater is the latency of processing fetch requests since a fetch request needs to wait for longer time before the global update progress pasts its reply timestamp. In the case when `P` is *not a direct downstream* of `U`, `U` should emit w at t_2 in order to guarantee the correctness of State access. However, when `P` is *a direct downstream* of `U`, we can actually push t_2 to t_1, i.e., `U` can emit w at t_1, as we analyze below.

Since the two operators `U` and `P` are in the same dataflow subgraph, the progress message m will be put into the same message queue as the update requests, in the order of the message creation time. The State consumes messages in the queue in FIFO order. Thus, when the State sees m, all update requests that were sent to the State before m must have been processed. This acceleration allows an update Progress operator to send timely and correct progress of the update operations to the State.

Compaction of old values. With stream records coming in continuously, a State keeps accumulating more entries and each entry may accumulate more values with different timestamps. This will lead to unbounded memory usage and slow down State access. Compaction addresses this problem.

We may *actively* call compaction when the fetch progress F moves forward. However, in case of a big increment on F, a large number of values will have to be compacted, which results in a halt on request processing and thus a sudden, sharp increase in the latency, even if compaction is processed in parallel by many State instances. Also, if F advances frequently, this approach wastes time as each compaction goes through all the State entries but only a small fraction of the entries may be compacted. Thus, the active approach may create latency spikes. For example, active compaction for the AdCamp workload (tested with an event rate of 1M/s and update rate over 50K/s) leads to highly unstable 99% latency (from hundreds of msec to tens of sec).

We consider a hybrid approach. We first take a *lazy* approach, which piggybacks the compaction of the old values of a State entry when we actually access the entry. For the case that some State entries may need to wait for a long time for compaction because they are not frequently accessed, we schedule an active compaction procedure periodically with a configurable period. For each scheduled compaction, we sweep over a fixed number of entries (starting from where the previous compaction stops) such that normal State processing is much less affected. Compared with active compaction, this approach has stable 50% and 99% latency at 2ms and 60ms for the *AdCamp* workload.



(a) Message queue (b) Multiple writers

Fig. 5: Conflict-free message queues

4.2 State Message Delivery

State messages refer to Fetch/Update requests and replies, and progress messages. The delivery of State messages are all based on messaging among operators. When stream records come at a fast rate, a large number of requests and replies are generated and delivered among many State operator instances and State instances, both within the same machine and across the network. That raises a challenge to the design of the message delivery queue to meet the low-latency requirement.

One standard design [27], [30] is a single-reader-many-writer queue, where only one reader consumes the queue and many writers can push messages to the queue using atomic compare-and-swap (CAS) operation. However, reads may conflict with writes and writes may also conflict with each other when writing to the same queue. The conflicts lead to retries of reads and writes. The more conflicts we have, the more retries we will get, which leads to higher end-to-end latency. The case is especially common when many State operator instances send requests at a high rate to the same State instance, which leads to many conflicts in reading/writing from/to the message queue of the State instance (for local delivery) or the network communication module (for remote delivery).

Conflict-free message delivery. To address the above-mentioned problem, we first adopt a circular message queue design for the delivery from one writer to one reader, as shown in Figure 5a. The message queue is based on a circular array of fixed length N, with a **write offset** W and a **read offset** R. Each slot of the array stores one message. The offset W (or R) indicate the latest position that the writer (or reader) can write (or read). When a new message is written, W is incremented by 1. W is reset to 0 if $W = N$, meaning that the writer should write from the beginning next time. The offset R is incremented in a similar way when reading messages in the queue.

To handle message delivery from multiple writers to one reader, we create one message queue for the connection from each writer to the reader, e.g., in Figure 5b we have four queues (in blue) from each writer connected to the reader. This design eliminates (1) the conflicts among writers because each writer writes to a separated queue, and (2) the conflicts between a reader and a writer because they operate on different positions of the queue and update different offsets, unless the queue is full. Here, a reader or a writer is physically an executor in Nova. The number of executors in one machine is usually not larger than the number of CPU cores. Thus, even for a full connection among all the executors, the number of queues that an executor needs is still reasonably small, and even a full scan on these message queues has little overhead.

Timely message processing. While the message queue design is conflict-free, it is challenging to ensure that a reader reacts timely to new messages. One simple solution is busy-waiting for new messages, i.e., the reader keeps checking if there is any new message in all the queues. However, this approach results in CPU cores being occupied by writer and reader threads, which wastes CPU resource and makes other threads (e.g., threads for network I/O, disk I/O or timing service) more difficult to obtain CPU resource for their processing.

To avoid busy-waiting, we use a **notification** mechanism as follows. We associate a **notification flag** F_i with each conflict-free queue Q_i. Initially $F_i = 0$. When a new message is written to Q_i, we have two cases. (1) $F_i = 0$, indicating that the new message is not noticed by the reader. In this case, the writer of Q_i sets $F_i = 1$ and sends a notification to the reader. When the reader receives a notification, the reader first sets $F_i = 0$ and then reads all the messages it sees in Q_i. (2) $F_i = 1$, meaning that a notification has been sent by a prior message, though the reader has not received it yet. In this case, we simply wait for the reader to receive the notification and read Q_i. Thus, the notification mechanism ensures that new messages pushed into Q_i will be read by the reader when the reader receives a notification from Q_i.

The notifications are written/read to/from a queue using CAS operation, which brings back some conflict overhead. However, for this notification queue, we can effectively lower its conflict overhead as follows. We reduce the need for notification by keeping the reader actively reading from conflict-free queues that are receiving new messages. Upon receiving a notification sent by some queue Q_i, the reader marks Q_i as "*toRead*" and triggers a periodical consumption procedure (if it was not already triggered by a prior notification of some Q_j). This procedure runs repeatedly in rounds as long as there is a "*toRead*" queue. In each round, the reader reads the messages in each "*toRead*" queue. If the reader fails to get any message from a "*toRead*" queue Q_i (i.e., no new message is pushed into Q_i) in the last K rounds ($K = 100$ by default), it un-marks Q_i, sets $F_i = 0$, and reads Q_i once again to consume messages that may have arrived during the time gap between the last read and the setting of F_i.

Using the above mechanism, when the stream speed is high, the reader will get new messages from the conflict-

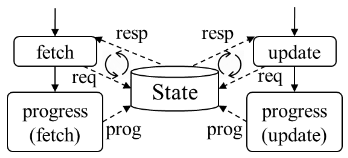


Fig. 6: Communication between State and operators

free queues continuously and writers need not send notifications frequently to the reader, where the writer/reader can achieve high write/read speed without conflict. When a queue is empty, the reader needs not waste time checking the queue but it can react to new messages with the help of notifications.

4.3 Fault Tolerance on State Access

Asynchronous barrier snapshotting (*ABS*) [22], [31] proposed by Apache Flink is a state-of-art technique for fault tolerance and *exactly-once* processing, which avoids global halts and supports cyclic dataflow graphs. Nova adopts *ABS* as the underlying fault tolerance mechanism to ensure the exactly-once message delivery for all the physical communications. In this section, we discuss how we ensure fault tolerance on the communication between State and State operators based on *ABS*, and optimize ABS specifically for State access.

We first attempt to apply ABS directly for the communication between State and State operators. Consider the communication pattern shown in Figure 6. ABS periodically generates a *checkpoint barrier*. When a Fetch/Update operator instance receives a barrier, it first persists its *state*[1] to durable storage, then sends the barrier to its downstream Progress operator instance and all the State instances. Similarly, a Progress operator instance also persists its state and sends the barrier to the State instances. When a State instance receives the barriers from all the associated Fetch/Update/Progress operator instances, it persists its state (including the State entries, the yet-to-process fetch requests, and the latest reported progresses of the Progress operator instances) and sends the barriers through the "resp" edges back to the Fetch/Update operator instances. The Fetch/Update operator instances need to log all the replies that they receive through the "resp" edge until they receive a barrier. Persisted states along with logged replies form a complete checkpoint. In case of failure, each operator instance loads its persisted state from the latest *globally-completed checkpoint* and resumes the execution from there.

While this solution promises exactly-once processing, it is not efficient as a State instance has to wait for checkpoint barriers from many operator instances. As shown in [22], waiting for more barriers incurs higher alignment time (i.e., the time gap from the receipt of the first barrier to the receipt of the last barrier), which in turn adds higher end-to-end latency to record processing.

To address the above problem, we make the following change: *(1) Fetch operator instances and update Progress instances do not send checkpoint barriers to the State instances; (2) State instances do not persist yet-to-process fetch requests*. The above optimization helps reduce the alignment time as Fetches are more frequent than Updates. However, it is not straightforward to see why this change still ensures the correctness of fault tolerance, which we explain below.

To show the correctness, we only need to consider **(A)** *whether requests and progress messages are delivered correctly*, and **(B)** *whether requests and progress messages are processed by a State instance in a correct order*.

For **(A)**, the exactly-once delivery of update requests and fetch progress messages are automatically covered by ABS. For fetch requests, since un-responded fetch requests are checkpointed by the Fetch operator instances that process them, the exactly-once delivery of fetch requests can be guaranteed by simply re-sending these requests during recovery. As for update progresses, missing some of them due to failure does not matter, because new update progresses will be generated.

For **(B)**, we need to ensure that (i) an update/fetch progress p is seen by the State only after all update/fetch requests before p have all been processed, and (ii) a fetch request with reply timestamp t is replied only when all update requests before t have all been processed. Point (i) is guaranteed because after the recovery, an update/fetch progress is also generated only when all the update/fetch requests before the progress are replied. As in the case when *update progress acceleration* is applied, (i) is also ensured as the analysis in §4.1 still holds. With (i) guaranteed, the update progress U and the fetch progress F are thus correctly maintained, which guarantees (ii) by the time semantics discussed in §3.4.

5 Experimental Evaluation

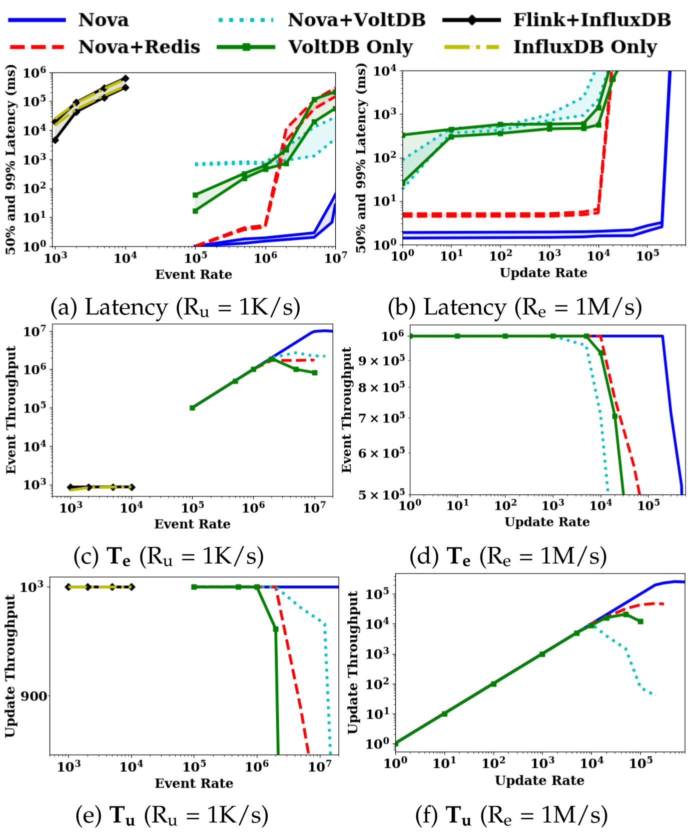
We evaluated Nova using three applications and compared with alternative solutions (§2.2-2.3) to show that a general solution for state sharing is needed. We used a cluster of 10 machines, each with 16 Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz and 32GB RAM, connected with 10Gbps Ethernet (average latency $\approx$ 80us). For all the systems tested, we carefully tuned their parameters to give the best performance as we could.

5.1 Advertising Campaign Stream Analysis

We used the AdCamp workload in §3.5 as a benchmark to measure the performance of State access. We compared Nova with three **composite solutions**: Nova with state managed by Redis (5.0.0), Nova with state managed by VoltDB (10.1.1), and Flink (1.12.0) with state managed by InfluxDB (2.0.3). *By comparing Nova with Nova+Redis and Nova+VoltDB, we want to show that the performance differences come only from the differences in state management, i.e., using Nova, Redis or VoltDB.*

In addition, we also compared Nova with **DB-only solutions**: VoltDB and InfluxDB, by directly implementing the AdCamp workload on them without using any DSPS, where state is managed by VoltDB or InfluxDB while requests and replies of updates and events are handled by threads using VoltDB or InfluxDB client library.

1. The state here refers to the (local) state of an operator instance, which is not the same as the entries stored in a State, i.e., a system-wide state. Note that although the data and the results of stream processing are stored in a State, each operator in Nova still keeps a minimal amount of essential information (e.g., upstream watermark) for its processing.

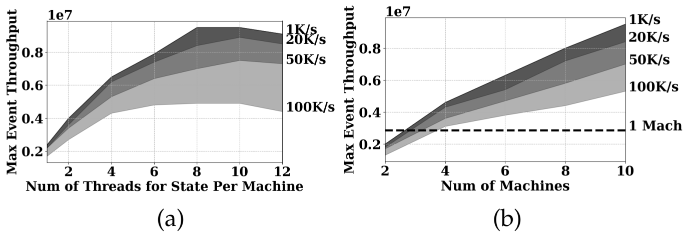


(a) Latency (R_u = 1K/s) (b) Latency (R_e = 1M/s)

(c) $\mathbf{T_e}$ (R_u = 1K/s) (d) $\mathbf{T_e}$ (R_e = 1M/s)

(e) $\mathbf{T_u}$ (R_u = 1K/s) (f) $\mathbf{T_u}$ (R_e = 1M/s)

Fig. 7: Performance on the AdCamp workload



(a) (b)

Fig. 8: Scalability performance on the AdCamp workload

The storage systems, i.e., Redis, VoltDB and InfluxDB, do not support time semantics for state access, i.e., they cannot tell whether updates up to a given timestamp are all applied. Thus, before we issue a fetch request on time range $(-\infty, T_1]$, we need to ensure that every update with timestamp T_2, where $T_2 \in (-\infty, T_1]$, has been applied in the storage system. We maintain a progress P for all replied updates, i.e., P guarantees that all updates with timestamp $T_2 < P$ have been applied in the storage system. For composite solutions, P is delivered from the update stream to the event stream to indicate that queries in the time range $(-\infty, T_1]$, where $T_1 < P$, can now be sent. For DB-only solutions, P is updated to the DB system and the event stream learns the changes of P by actively querying the DB system. Asynchronous operation is used for both update and query operations for the communication between the external storage and the DSPS.

We report the latency for different *update rate* $\mathbf{R_u}$ and *event rate* $\mathbf{R_e}$, i.e., the number of updates or events generated per second by the sources. We also report the actual *update throughput* $\mathbf{T_u}$ and *event throughput* $\mathbf{T_e}$, i.e., the number of updates and events processed per second.

In Fig 7a and Fig 7b, each pair of curves plot the 99% and 50% latency of a method. Among all the methods, Nova has significantly lower and stable latency. For R_u=1K/s, the 99% latency of Nova remains less than 60ms when R_e=10M/s. For R_e=1M/s, Nova can handle R_u up to 200K/s with latency around 3ms. We explain the main difference between Nova and the other methods as follows.

For Nova+Redis, Nova+VoltDB, and VoltDB, a fetch request r needs to wait for the progress P of the replied updates to pass r's time range before r can be sent out, but the advance of P can be delayed by some update requests that are not timely processed. When the update rate is higher, the advance of P needs to wait for more update requests to be replied and this leads to higher latency. In contrast, a fetch request in Nova can be sent to the State once it is created, without the need to check the progress of update replies. The time semantics of State (§3.4) ensures that fetch and update requests will be processed correctly. Compared to the progress P above, the update progress U in Nova is advanced more timely with **the acceleration technique** (§4.1), which helps the fetch requests to be replied timely.

We report T_e and T_u in Fig 7c-7f. Nova achieves significantly higher T_e and T_u than the other methods. We notice that T_u of Nova+VoltDB and VoltDB in Fig 7f drops rapidly when R_u goes beyond 10K/s and 50K/s, while T_u of Nova and Nova+Redis eventually becomes flat and stable. With higher R_u, more keys are created and each key carries more timestamped values. For VoltDB, each query needs to scan more entries in the table and each update suffers more delay. But for Nova and Nova+Redis, a query/update only needs to look up a key from a hash table and searches/inserts the value in a sorted structure, which is much more efficient and stable even under a large amount of keys and values.

Flink+InfluxDB and InfluxDB have similar performance. In both the composite solution and DB-only solution, we observed that InfluxDB used all the CPU cores to process updates and queries, but the processing throughput was still low and the latency was still high. Thus, we were only able to obtain the results for Flink+InfluxDB and InfluxDB when $\mathbf{R_u}$ and $\mathbf{R_e}$ are as low as 1K/s.

We also examined the scalability of our State. Fig 8 reports the maximum T_e that Nova can handle (within 200ms latency). Each curve in Fig 8 is for a different R_u, from 1K/s to 100K/s. Fig 8a shows that the maximum T_e increases when more threads are added for State. T_e stops increasing when the number of threads is around 8 to 10 where all the CPU cores are fully utilized by the State and its access operators. If more threads are added, State starts to compete with other operators (e.g., update) for computing resources, which degrades the performance. In Fig 8b, the maximum T_e scales linearly when we increase the number of machines from 2 to 10. Note that when 1 machine is used, T_e is stable around 2.9M/s for $R_u < 100K/s$, which is higher than T_e when 2 machines are used because of the network communication overhead. But as the number of machines increases to 3 or 4, T_e is already higher than 2.9M/s.

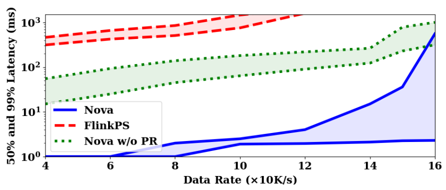


Fig. 9: Online inference latency

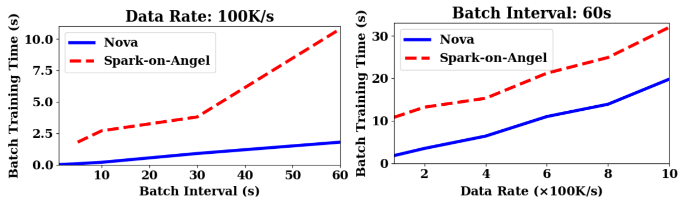


Fig. 10: Online training time

5.2 Online Learning

We evaluated the use of State in online learning applications. We tested both (1) ***inference*** with low-latency requirement, where samples were processed one after another, and (2) ***training*** with high-throughput demand, where samples are grouped by a window operation on mini-batches and processed in a BSP manner. We used FTRL [32], [33], a standard online learning algorithm used in industry, on the kdd12 dataset (with 54M features) on 10 machines. For inference, we compared with FlinkPS [20], which is a parameter-server implementation based on Flink using the CD workaround (§2.2). For training, we compared with a composite solution, Spark-on-Angel (2.0.0) [34], [35], which integrates Apache Spark and Angel (a parameter server) for online learning in industry (it outperforms native Spark in machine learning workloads under batch processing).

In Fig 9, each pair of curves report the 50% and 99% latency for inference. Nova achieves very low 50% latency in all cases. In contrast, the latency of FlinkPS is hundreds of times larger than Nova mainly due to inefficient parameter pulling operation. The *pull replies* in FlinkPS are processed as normal stream records and Flink does not have a mechanism to raise the priority of pull replies. Thus, the pull replies suffer from the head-of-line blocking (as discussed in §4.1), which becomes worse at a higher data rate since pull replies and new stream samples can easily fill up the message queue of a worker in FlinkPS. This further triggers back pressure and causes cyclic deadlock. FlinkPS provides a user-configurable option to set a limit on the number of unanswered *pull requests* of each worker. But as we tested, the maximum limit we can set is only 1.8K, which gives the lowest latency without deadlock. In contrast, using **prioritizing reply (PR)** (§4.1), the *pull replies* (i.e., fetch replies) in Nova have higher priority than normal records in terms of delivery and processing. To validate this, we disabled PR in Nova and Fig 9 shows that its latency worsens significantly. The result is still better than FlinkPS because Nova has other designs such as conflict-free message delivery (§4.2) that allows it to process requests and replies more efficiently under higher data rates (§5.4).

Fig 10 reports the average training time of a mini-batch under (1) different batch intervals and fixed data rate at 100K/s and (2) different data rates and fixed batch interval at 60s. Nova records competitive training time compared with Spark-on-Angel, showing that the State design can handle online training with high throughput. To analyze the performance, Table 2 reports the percentage of time spent on different steps of the training: pull request generation (*PullGen*) and delivery (*Pull*), gradient calculation (*Cal*), push request generation (*PushGen*) and delivery (*Push*). The percentages of each step for different settings do not vary significantly and we report the average. We found that Spark-on-Angel spent most of the time on *Pull* and *Push*, while Nova spent more time on *PullGen* and *Cal*. Although *PullGen* and *Cal* are most time consuming in Nova, their actual time is still relatively small. For *Pull* and *Push* that are communication intensive, Nova's percentages are significantly lower than Spark-on-Angel, which shows that Nova is more efficient on the delivery of pull/push (i.e., fetch/update) requests/replies.

TABLE 2: Mini-batch training time decomposition

	PullGen	Pull	Cal	PushGen	Push
Nova	30%	9%	51%	4%	6%
Spark-on-Angel	7%	37%	18%		38%

Based on the above analysis, we further studied the communication between Spark and Angel. Here we mainly discuss *Pull* since the delivery for both *Pull* and *Push* are similar. A Spark executor puts pull requests to *Dispatcher* threads via lock-based blocking queues and then waits until all requests are replied. Each *Dispatcher* uses *Requester* threads to deliver pull requests to one server in Angel. Another type of *Dispatcher* threads use *Responser* threads to process pull replies from servers in Angel, where the *Responsers* use lock to avoid conflicts when writing results to the same executor. The request processing is asynchronous, but the frequent uses of locks incur non-negligible overheads on the *Pull* (and *Push*) operation in Spark-on-Angel. In contrast, the message delivery in Nova is also asynchronous but the **conflict-free message queue** design and **notification mechanism** (§4.2) enable efficient message delivery and processing, which we will further examine in §5.4.

5.3 Real-time Cycle Detection (RTCD)

In this experiment we evaluated the performance of Nova for RTCD. Following [15], we created three queries to detect cycles of length 3, 4 and 5 that are formed in the recent 48, 24 and 12 hours. When an edge update with timestamp t arrives, we detect new cycles in the updated graph within time window $[t - s, t]$, where s is 48, 24 or 12 hours. We generated a graph stream from a public social graph, Twitter [36], as follows: first, we extracted a subgraph from Twitter as the starting graph and assigned timestamps to its edges, where the timestamps are evenly distributed over a period of 48 hours; then, we randomly sent the remaining edges as an edge stream.

We compared Nova with Timely Dataflow (0.10.0) [9], denoted as **Timely**. Nova and Timely use the same RTCD algorithm and the difference is in the state management. We use the CD+OF workaround (§2.2) to manage state in Timely as follows. We implemented two types of operators: a *state* operator for storing the dynamic graph as partitioned in-operator state and three *query* operators for processing the three queries. Each time a new batch of edge updates is sent

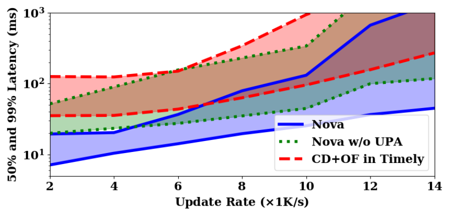


Fig. 11: Query time of RTCD

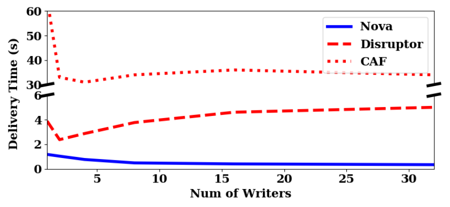
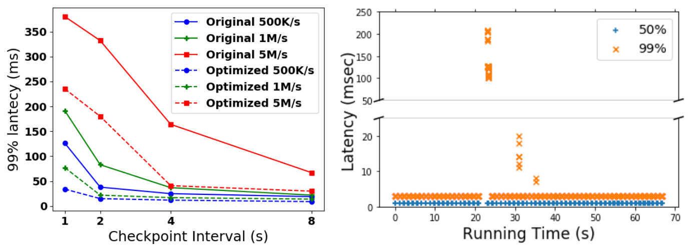


Fig. 12: Delivery time for 100M messages



(a) Checkpointing (b) Recovery

Fig. 13: Fault tolerance results (best viewed in color)

to the state operator to update the graph, and to the query operators to trigger query execution. The query operators and the state operator form a *cyclic dataflow* for request/reply delivery of graph data, where the query operator fuses request processing and cycle detection.

We ran Nova and Timely on 10 machines, and carefully tested Timely with different number of worker threads to get the best results. We started with a graph with 3.5M edges and the systems processed a batch of new edge updates in each second, where the update rate varied from 2K/s to 14K/s. In Fig 11, each pair of curves report the 50% and 99% latency for processing a query, which shows that Nova achieves significantly better performance especially at higher data rates.

We analyzed the CD+OF workaround and found that its performance problem is due to its large amount of progress traffic. To compact stale graph data, the state operator needs to know the progresses of the query operators, in order to get a correct time boundary T such that no more update with timestamp $t < T$ is still being processed inside the dataflow cycle. However, to get the correct T in the CD+OF workaround, we need to maintain different progresses at different iterations for the requests and replies that are flowing inside the dataflow cycle, which generates many progresses to be delivered. In addition, since the state operator is inside the cycle, it also has to report progresses to the query operators. In contrast, our State design decouples State progress from State access operations, where a State needs not send progress to other operators. We use a Fetch operator in a dataflow cycle to fetch the neighbors of a vertex iteratively, but we report fetch progresses by a Fetch Progress operator *after* the dataflow cycle so that the Progress operator can report to the State the output watermark of the dataflow cycle, without iteration information.

We also evaluated the effectiveness of **Update Progress Acceleration (UPA)** (§4.1). We disabled UPA in Nova and Fig 11 shows that UPA improves the 99% query processing time, as it helps advance the update progress more timely.

5.4 Performance on Message Delivery

We evaluated the performance of the message delivery mechanism in §4.2 with an $X : 1$ communication scenario, which measures the competition among X writers communicating with the same reader. Each writer and the reader run in separate threads. The total number of messages sent by the writers was 100M and each message was a 8-byte integer to mimic a message pointer. The length of each message queue was set to $2^{16} = 65{,}536$. We varied X from 1 to 32 (as each machine has 32 cores). We compared Nova with Disruptor [30] and the default delivery mechanism in CAF (as Nova is built on CAF). Disruptor is a high performance inter-thread messaging library, where the message queue is also based on a cyclic array design but different writers push messages to the same queue via CAS operation. As writers in Disruptor share the same queue, the length of its queue was set to 65,536X.

Fig 12 reports the delivery time for 100M messages. When the number of writers increases, the delivery time of Disruptor and CAF first decreases because messages are written concurrently into the queue, but then increases because the conflicts among the writers on the CAS operations become severe and the overhead of conflicts outplays the benefit of concurrent writes. In contrast, Nova's delivery time decreases continuously. With 32 writers, Nova has a throughput of 298M messages/sec. This is because each writer in Nova writes to its own queue and writers have no conflict with each other, and thus more workers effectively improve the overall message delivery throughput.

5.5 Performance on Fault Tolerance

We examined Nova's performance on checkpointing and recovery using the AdCamp workload on 10 machines. For checkpointing, we fixed the update rate to 1K/s and tested different event rates and checkpoint intervals. We compared our optimized ABS in §4.3 with the original ABS [22], [31]. Figure 13a shows that our optimization effectively reduces the 99% latency, especially for larger event rates and smaller checkpoint intervals. To evaluate the performance of recovery, we killed one of the workers after the application ran stably for a period of time and replaced it with a backup worker. We set the checkpoint interval to 5s. Figure 13b shows that the worker failure happened at around 20s and the 50% latency remained stable at 2ms, while the 90% latency went up to around 200ms but soon dropped and stabilized at 4ms.

6 Conclusions

We presented a new system-wide state abstraction with well-defined time semantics, filling in a missing piece in existing stream processing systems, i.e., timestamped state sharing. We built Nova, a prototype system, to validate our State design with efficient implementation and optimizations. Experiments show that Nova effectively improves performance on a wide range of workloads such as advertising campaign stream analysis, dynamic graph analytics, and online learning. The results also show that designs such as conflict-free message delivery, prioritizing replies, update progress acceleration and optimized ABS are effective.

Acknowledgments. We thank the reviewers for their constructive comments that help significantly improve the quality of the paper. This work was supported by GRF 14208318 from the RGC of HKSAR.

References

[1] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the twenty-fourth ACM symposium on operating systems principles*. ACM, 2013, pp. 423–438.

[2] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia, "Structured streaming: A declarative API for real-time applications in apache spark," in *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, 2018, pp. 601–613.

[3] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink^{TM}: Stream and batch processing in a single engine," *IEEE Data Eng. Bull.*, vol. 38, no. 4, pp. 28–38, 2015. [Online]. Available: http://sites.computer.org/debull/A15dec/p28.pdf

[4] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. V. Ryaboy, "Storm@twitter," in *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, 2014, pp. 147–156. [Online]. Available: https://doi.org/10.1145/2588555.2595641

[5] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, "Samza: stateful scalable stream processing at linkedin," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1634–1645, 2017.

[6] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, T. K. Sellis, S. B. Davidson, and Z. G. Ives, Eds. ACM, 2015, pp. 239–250. [Online]. Available: https://doi.org/10.1145/2723372.2742788

[7] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "Millwheel: Fault-tolerant stream processing at internet scale," *PVLDB*, vol. 6, no. 11, pp. 1033–1044, 2013.

[8] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: a timely dataflow system," in *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, M. Kaminsky and M. Dahlin, Eds. ACM, 2013, pp. 439–455. [Online]. Available: https://doi.org/10.1145/2517349.2522738

[9] Timely dataflow. "https://github.com/TimelyDataflow/timely-dataflow".

[10] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard, "Differential dataflow," in *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org, 2013. [Online]. Available: http://cidrdb.org/cidr2013/Papers/CIDR13_Paper111.pdf

[11] D. H. Stern, R. Herbrich, and T. Graepel, "Matchbox: large scale online bayesian recommendations," in *Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20-24, 2009*, J. Quemada, G. León, Y. S. Maarek, and W. Nejdl, Eds. ACM, 2009, pp. 111–120. [Online]. Available: https://doi.org/10.1145/1526709.1526725

[12] C. Li, Y. Lu, Q. Mei, D. Wang, and S. Pandey, "Click-through prediction for advertising in twitter timeline," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Sydney, NSW, Australia, August 10-13, 2015*, L. Cao, C. Zhang, T. Joachims, G. I. Webb, D. D. Margineantu, and G. Williams, Eds. ACM, 2015, pp. 1959–1968. [Online]. Available: https://doi.org/10.1145/2783258.2788582

[13] A. Agarwal, S. Bird, M. Cozowicz, L. Hoang, J. Langford, S. Lee, J. Li, D. Melamed, G. Oshri, and O. Ribas, "Making contextual decisions with low technical debt," *arXiv preprint arXiv:1606.03966*, 2016.

[14] Y. Zhang, R. Chen, and H. Chen, "Sub-millisecond stateful stream querying over fast-evolving linked data," in *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 2017, pp. 614–630. [Online]. Available: https://doi.org/10.1145/3132747.3132777

[15] X. Qiu, W. Cen, Z. Qian, Y. Peng, Y. Zhang, X. Lin, and J. Zhou, "Real-time constrained cycle detection in large dynamic graphs," *PVLDB*, vol. 11, no. 12, pp. 1876–1888, 2018. [Online]. Available: http://www.vldb.org/pvldb/vol11/p1876-qiu.pdf

[16] A. Anagnostopoulos, J. Lacki, S. Lattanzi, S. Leonardi, and M. Mahdian, "Community detection on evolving graphs," in *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, D. D. Lee, M. Sugiyama, U. von Luxburg, I. Guyon, and R. Garnett, Eds., 2016, pp. 3522–3530. [Online]. Available: http://papers.nips.cc/paper/6173-community-detection-on-evolving-graphs

[17] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *PVLDB*, vol. 8, no. 12, pp. 1792–1803, 2015. [Online]. Available: http://www.vldb.org/pvldb/vol8/p1792-Akidau.pdf

[18] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey, "Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language," in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, 2008, pp. 1–14. [Online]. Available: http://www.usenix.org/events/osdi08/tech/full_papers/yu_y/yu_y.pdf

[19] Trident. "http://storm.apache.org/releases/current/Trident-tutorial.html".

[20] Flink-parameter-server. "https://github.com/FlinkML/flink-parameter-server".

[21] T. Rabl, M. Sadoghi, H. Jacobsen, S. Gómez-Villamor, V. Muntés-Mulero, and S. Mankowskii, "Solving big data challenges for enterprise application performance management," *PVLDB*, vol. 5, no. 12, pp. 1724–1735, 2012. [Online]. Available: http://vldb.org/pvldb/vol5/p1724_tilmannrabl_vldb2012.pdf

[22] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas, "State management in apache flink: Consistent stateful distributed stream processing," *PVLDB*, vol. 10, no. 12, pp. 1718–1729, 2017. [Online]. Available: http://www.vldb.org/pvldb/vol10/p1718-carbone.pdf

[23] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. R. Pietzuch, "Making state explicit for imperative big data processing," in *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014.*, 2014, pp. 49–60. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/castro-fernandez

[24] J. Meehan, N. Tatbul, S. Zdonik, C. Aslantas, U. Çetintemel, J. Du, T. Kraska, S. Madden, D. Maier, A. Pavlo, M. Stonebraker, K. Tufte, and H. Wang, "S-store: Streaming meets transaction processing," *PVLDB*, vol. 8, no. 13, pp. 2134–2145, 2015. [Online]. Available: http://www.vldb.org/pvldb/vol8/p2134-meehan.pdf

[25] F. McSherry, A. Lattuada, M. Schwarzkopf, and T. Roscoe, "Shared arrangements: practical inter-query sharing for streaming dataflows," *Proc. VLDB Endow.*, vol. 13, no. 10, pp. 1793–1806, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol13/p1793-mcsherry.pdf

[26] J. Gjengset, M. Schwarzkopf, J. Behrens, L. T. Araújo, M. Ek, E. Kohler, M. F. Kaashoek, and R. T. Morris, "Noria: dynamic, partially-stateful data-flow for high-performance web applications," in *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA,*

October 8-10, 2018., 2018, pp. 213–231. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/gjengset

[27] D. Charousset, R. Hiesgen, and T. C. Schmidt, "Revisiting actor programming in C++," *Comput. Lang. Syst. Struct.*, vol. 45, pp. 105–131, 2016. [Online]. Available: https://doi.org/10.1016/j.cl.2016.01.002

[28] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, 2012, pp. 15–28.

[29] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier, "Out-of-order processing: a new architecture for high-performance stream systems," *PVLDB*, vol. 1, no. 1, pp. 274–288, 2008. [Online]. Available: http://www.vldb.org/pvldb/vol1/1453890.pdf

[30] Lmax disruptor. "https://github.com/LMAX-Exchange/disruptor".

[31] P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas, "Lightweight asynchronous snapshots for distributed dataflows," *CoRR*, vol. abs/1506.08603, 2015. [Online]. Available: http://arxiv.org/abs/1506.08603

[32] H. B. McMahan, "Follow-the-regularized-leader and mirror descent: Equivalence theorems and L1 regularization," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2011, Fort Lauderdale, USA, April 11-13, 2011*, 2011, pp. 525–533. [Online]. Available: http://proceedings.mlr.press/v15/mcmahan11b/mcmahan11b.pdf

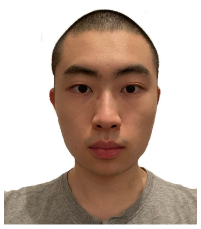
[33] H. B. McMahan, G. Holt, D. Sculley, M. Young, D. Ebner, J. Grady, L. Nie, T. Phillips, E. Davydov, D. Golovin, S. Chikkerur, D. Liu, M. Wattenberg, A. M. Hrafnkelsson, T. Boulos, and J. Kubica, "Ad click prediction: a view from the trenches," in *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA, August 11-14, 2013*, 2013, pp. 1222–1230. [Online]. Available: https://doi.org/10.1145/2487575.2488200

[34] J. Jiang, L. Yu, J. Jiang, Y. Liu, and B. Cui, "Angel: a new large-scale machine learning system," *National Science Review*, vol. 5, no. 2, pp. 216–236, 2017.

[35] J. Jiang, B. Cui, C. Zhang, and L. Yu, "Heterogeneity-aware distributed parameter servers," in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 463–478.

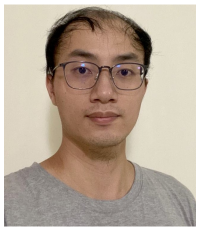
[36] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. Manhattan, USA: ACM Press, 2004, pp. 595–601.

Yunjian Zhao is currently a PhD candidate in the Department of Computer Science and Engineering at the Chinese University of Hong Kong. His research interests include graph databases, online pattern matching and stream processing systems.



Zhi Liu is currently a PhD student in the Department of Computer Science and Engineering at the Chinese University of Hong Kong. His research interests include graph databases, stream processing systems and cluster resource management.

Yidi Wu is currently a PhD candidate in the Department of Computer Science and Engineering at the Chinese University of Hong Kong. His research interests include machine learning systems, distributed data processing systems and cluster schedulers.



Guanxian Jiang is currently a PhD student in the Department of Computer Science and Engineering at the Chinese University of Hong Kong. His research interests include graph databases and distributed computing systems.

James Cheng is currently an associate professor in the Department of Computer Science and Engineering at the Chinese University of Hong Kong. His research interests include distributed systems, graph databases, machine learning systems, and cluster resource management.

Kunlong Liu was a research assistant in the Department of Computer Science and Engineering at the Chinese University of Hong Kong when he worked on this project.

Xiao Yan was a PhD candidate in the Department of Computer Science and Engineering at the Chinese University of Hong Kong when he worked on this project. His research interests include large-scale similarity search, distributed machine learning, and graph neural networks.