ICE: Managing Cold State for Big Data Applications

Badrish Chandramouli [1], Justin Levandoski [2], Eli Cortez [3]

Microsoft

Redmond, WA, USA

[1]badrishc@microsoft.com, [2]justin.levandoski@microsoft.com, [3]elicorte@microsoft.com

***Abstract*—The use of big data in a business revolves around a monitor-mine-manage (M3) loop: data is monitored in real-time, while mined insights are used to manage the business and derive value. While mining has traditionally been performed offline, recent years have seen an increasing need to perform all phases of M3 in real-time. A stream processing engine (SPE) enables such a seamless M3 loop for applications such as targeted advertising, recommender systems, risk analysis, and call-center analytics. However, these M3 applications require the SPE to maintain massive amounts of state in memory, leading to resource usage skew: memory is scarce and over-utilized, whereas CPU and I/O are under-utilized. In this paper, we propose a novel solution to scaling SPEs for memory-bound M3 applications that leverages natural access skew in data-parallel subqueries, where a small fraction of the state is hot (frequently accessed) and most state is cold (infrequently accessed). We present ICE (incremental cold-state engine), a framework that allows an SPE to seamlessly migrate cold state to secondary storage (disk or flash). ICE uses a novel architecture that exploits the semantics of individual stream operators to efficiently manage cold state in an SPE using an incremental log-structured store. We implemented ICE inside an SPE. Experiments using real and synthetic datasets show that ICE can reduce memory usage significantly without sacrificing performance, and can sometimes even improve performance.**

I. Introduction

The use of big data in a business can be summarized by the *monitor-mine-manage* (*M3*) loop [1], [2]. Briefly, users monitor or archive incoming data in real time, and mine the archived "big data" to derive insights that feed into the manage phase, where these insights are used to improve or derive more value for the business. Two popular M3 applications are introduced below (Section III describes them in greater detail).

1) Recommender Systems [3]: Platforms such as NetFlix, LinkedIn, Google News, and Microsoft XBox need to recommend movies, news items, and blog posts to customers. They monitor user ratings of items (e.g., movie ratings or news "likes"). They mine the massive collected data, for example, by building similarity models (between users and items, or pairs of items). During the manage phase, they provide either on-demand or push-based recommendations to users by applying the model to that users ratings and preference.

2) Behavior Targeting Advertising [2]: Advertising platforms such as Microsoft AdCenter and Yahoo! Ads show targeted ads to users based on their historical behavior. They monitor user behavior in the form of ad impression and ad click logs, search history, and URLs visited. They mine the large quantities of user data to eliminate activity by automated bots, remove spurious clicks, reduce dimensionality, and build user models. During the manage phase, they track recent per-user behaviour in real time. Given an opportunity to show an ad, they score the user and display the most relevant ad.

There is an increasing need to close the M3 loop, performing all phases in real time [2]. For instance, a real-time M3 loop allows a recommender system to suggest news articles or blog posts in time for them to be relevant. An advertising platform can use recent and rapidly changing trends (e.g., flash sales or unexpected events) to target user appropriately. Thus, we are witnessing a growing trend towards real-time in-memory analytics, where a stream processing engine (SPE) is used for processing large quantities of data in all the three phases of M3. For instance, in the mine phase, actions such as recommendation similarity model generation and maintenance for pairs of items, bot elimination, and dimensionality reduction are easily expressed using temporal queries. The same queries are used for one-pass analytics over offline data [2], e.g., to tune parameters by back-testing queries on historical data.

A. Challenges

M3 applications have two main characteristics that have deep implications on traditional SPE architectures:

1) Partitioned Computation: M3 queries typically operate on large quantities of data (e.g., millions of users and ads). At the same time, every sub-query is usually naturally partitionable by some key. For instance, item-item model generation in a recommender system is partitionable by items (e.g., movies), while generating item-item pairs for building the model is partitionable by user. Bot elimination in targeted advertising can be partitioned by user, whereas dimensionality reduction can be done on a per-ad basis. Streaming systems incorporate a *grouped sub-query* (GSQ) operation to express such partitionable queries. This operator takes a sub-query and a partition key, and logically executes the sub-query for every distinct partition key.

2) Large Windows and State: SPEs traditionally operate on a limited history of data (in the form of windows). This allows query state to be retained in memory for processing events efficiently. However, M3 applications need large (or infinite) windows, with a large amount of state in memory. In a recommender, we may retain prior ratings of every user for several months or years in order to build robust similarity models [3]. The models themselves may be retained and updated in memory, for all items in the system. In behavior targeted advertising, real-time scoring requires us to retain user behavior for weeks in order to correlate with new behavior and use the model to determine the category the user would be classified into for ad selection. Bot elimination requires us to maintain several counters and other auxiliary state on a per-user basis, with a large sliding window (e.g., six hours) over which bots are detected.

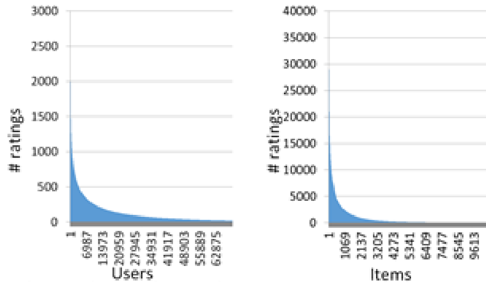


Fig. 1: Popularity of users and items in the MovieLens dataset.

SPEs are traditionally designed for main-memory-resident operations and are unable to efficiently handle the above characteristics of M3 applications. Using a traditional database is out of question given the need to handle *continuous queries* (*CQs*) in real time with temporal processing semantics. A simple solution is to scale out processing on multiple machines, by partitioning data on a per-subquery basis. Unfortunately, this can result is a significant waste of resources as queries are memory bound, leading to low CPU utilization per machine. For instance, we found CPU utilization to be less than 20% when using *Microsoft StreamInsight* (a commercial SPE) for a behavior-targeted advertising application. Scale-out to gain memory is even less desirable in pay-per-hour cloud environments, where we wish to make full use all of the machines resource (both disk and memory). Traditional load-shedding schemes [4], [5] drop tuples to handle load; however, choosing tuples to drop that will reduce state significantly while introducing low result error is hard for arbitrarily complex M3 queries, and is not an option for applications where query correctness is required.

B. Proposal: Use Storage before Scaling Out

We make two key observations that hold for our target applications. First, although a large amount of state is "in play" in the engine, only a small fraction of the state is actively accessed by the query at any given point in time. The remaining state, which we call *cold state*, is necessary but not actively accessed at any given point during processing. Second, the granularity of state classification for any sub-query in the application matches exactly with the partitioning key used for that sub-query. To make this point concrete, Figure 1 provides the distribution of user and item popularity in the MovieLens dataset [6] of 10M movie ratings; both are highly skewed. Many users are either inactive (not rating items or receiving recommendations) or have stopped using the service altogether. Further, there exists a long tail of unpopular items that are rarely updated or accessed during the recommendation process. Sub-queries in the recommender application are partitioned by the same grouping keys (UserId and Item). Likewise, in targeted advertising, although we track the behavior of millions of users over a long period, only a small fraction of users are active at any given point in time.

Based on these observations, we propose leveraging secondary storage to offload cold state on-the-fly in an SPE at a *per-sub-query-per-group* granularity. To achieve this, we need to efficiently detect and offload cold state during runtime. Further, we efficiently reload cold state on-the-fly into main memory when necessary. This facility allows the SPE to handle applications with large state at low latency, without sacrificing performance or query correctness, while using fewer machines or main memory.

We considered traditional page caching for managing cold state in the SPE, but found this approach to be inadequate. Caching is a well-known method used in systems that assumes most lives on storage. It does not work well in our setting due to two reasons. First, SPEs are commonly built assuming data is memory-resident and do not use page-level indirection for efficiency reasons. Second, page-level caching cannot reason about and exploit knowledge inherent in the query semantics such as group-level access skews (we discuss further caching drawbacks in Section VII). Previous research has also investigated interfacing SPEs with secondary storage (or an RDBMS) to query historical data [7], [8]. Our scenario is different; we study scaling SPEs by transparently and efficiently offloading (or reloading) cold *query internal state* on-the-fly to (or from) secondary storage, with a goal of keeping latency low while using significantly lesser memory. Finally, we note that our problem is significantly different from that of efficient hash table maintenance because SPEs operate at a higher level of abstraction - an ensemble of incrementally updatable and windowed grouped states organized as a graph – which provides us with many more challenges and opportunities for efficient state management.

C. Contributions

We design and build a system called ICE (for incremental cold-state engine) that adds native support for secondary storage by using a new *physical-operator-aware* approach to cold-state management. ICE exploits query semantics and the state organization of individual operators to efficiently handle cold state in an SPE. ICE has the following features:

(1) ICE leverages the grouped sub-query (GSQ) operation for fine-grained cold state management. This is based on our finding that M3 applications are usually data parallel, where a computation is repeated for every partitioning key. GSQ, which takes a sub-query and a partitioning key and executes the sub-plan for every key, corresponds to this data-parallel paradigm. ICE uses a cold state manager (CSM) that tracks groups with low overhead and decides which groups should migrate to secondary storage.

(2) The CSM is decoupled from the stream engine internals. It only knows about logical group ids and sends migration commands for making specific groups cold as events to the relevant GSQ and its sub-plan. The GSQ is in charge of migrating the group to/from storage. This decoupled architecture allows us flexibility in how we implement the CSM, for instance, to easily swap cold data classification techniques without modifying engine internals.

(3) Individual physical operators within a GSQ react to migration command events by deciding how and what state moves between secondary storage and main memory. This *physical-operator-aware* approach works well in practice because every physical operator knows its own data access patterns and is in a good position to judge how it manages its own state.

(4) ICE manages cold state efficiently using a *log-structured store* (*LSS*) that stores cold parts of data structures

used by various operators (e.g., join and aggregation). We introduce the concept of *automatic garbage collection* within our LSS that uses event expirations to simplify the (usually tedious and expensive) log-structured garbage collection process. We also leverage other stream characteristics to improve LSS performance, such as clean/dirty read tracking and delta updating.

(5) We implemented ICE to work with a test version of Microsoft StreamInsight. Extensive experiments with real and synthetic workloads demonstrate the feasibility of our techniques and their ability to allow the SPE to scale to large workloads in M3 applications by efficiently managing cold data using secondary storage.

This paper is organized as follows. Section II covers preliminaries, while Section III details two M3 applications. Section IV introduces the ICE architecture, with Section V focusing on the cold state manager. Section VI covers migration-aware stream operators. Section VII contains discussions, while Section VIII reports experimental results. Section IX covers related work, and we conclude in Section X.

II. Preliminaries

Streams and Queries A stream is a sequence of events that each consist of a payload (e.g., user movie rating), an occurrence time LE (for *left endpoint*), and a window expiration time RE (for *right endpoint*). The interval [LE, RE) indicates the window or period over which the event influences output. For "instantaneous" events with no lifetime, called *point events*, RE is set to LE+δ where δ is the smallest possible time-unit. Sometimes, we may not know the end time of an event when it is created. Thus, an event may appear as a *start-edge* event with lifetime [LE, ∞) to indicate that the event does not expire. This event may later be followed by an *end-edge* event that updates the window expiration to an earlier time (RE). Start and end edges correspond to I-streams and D-streams in STREAM [9], positive and negative tuples in NILE [10], and insertions and revisions in Borealis [7].

A continuous query (CQ) submitted by the user is converted into a query plan that consists of a tree of operators, each of which performs some transformation on its input streams and produces an output stream. A key operation in an SPE is the ability to run per-group computations, which is described next.

Grouped Sub-Query (GSQ). The GSQ operation allows the user to specify a grouping key (e.g., user id) and a query sub-plan to execute on each sub-stream corresponding to events with the same grouping key. In most SPEs, the GSQ operation is implemented using the following sub-operations: (1) A *group* operator reads each event from its input stream, computes its grouping key, and augments the event with this key to create a grouped stream as output. (2) The *sub-query*, consisting of a DAG of streaming operators, receives and produces grouped streams. (3) A final *ungroup* operator removes the grouping key from events in the stream, ending the grouped computation. Todays SPEs such as STREAM, Borealis, Storm, Spark Streaming, IBM System S, Microsoft StreamInsight, and Naiad include a GSQ operation, either as native query constructs (e.g., group-and-apply in Microsoft StreamInsight) or by explicitly offering a streaming $\langle$key, value$\rangle$ abstraction, where key serves as the grouping key for the data.

Operators All streaming operators are usually designed to accept and output grouped streams. For example, an aggregate operator, that receives events with $\langle$group-key, payload$\rangle$ pairs, maintains per-group-key state, and outputs a stream of data-batches with per-group aggregates $\langle$group-key, aggregate$\rangle$. The state inside such operators is usually organized by group, with a hash table. When a new event arrives at the operator, it looks up and updates the corresponding operator state in the hash table.

Consider the streaming equi-join operation (*StreamJoin*), which outputs the streaming join between its left and right input events. The output lifetime is the intersection of the joining event lifetimes. StreamJoin can be implemented with a streaming *grouped cross-product* (*GCP*) operator that operates over grouped streams; equi-join is identical to a cross-product over two input streams that are each grouped by the equi-join key. The implementation of GCP is identical to the well-known symmetric hash join, where the active events for each input are stored in a right or left internal join synopses (implemented as hash tables on the grouping or join key). A common application of StreamJoin is when the left input consists of point events in this case, StreamJoin effectively filters out events on the left input that do not intersect any matching event lifetime in the right synopsis. Section III provides examples of this pattern found in real applications. Other grouped stateful operators include set difference, aggregates (sum, top-k, etc.), and user-defined operators (see Section VI).

As discussed earlier, ICE tracks and classifies cold state by group, since it is a natural granularity that fits M3 applications well. In the next section, we present two example end-to-end applications that will help better understand our approach.

III. Example Applications

A. Real-Time Recommender System

A recommender system ingests tuples signifying a user action in a system (e.g., a Netflix user rating a movie, a Facebook using "liking" a post), builds a recommendation model based on these actions, and uses the model to generate recommendations for users. Systems usually perform the model-building step offline in batch; for instance, after ingesting K new user ratings. However, a recommender system can be built with an SPE to bring the entire recommendation process "online" by immediately updating the model as the system ingests new events. We refer to this as a real-time recommender system [3].

Figure 2 depicts the stream query plan for a real-time recommender using item-based collaborative filtering. There are two types of input: (1) update events that represent user ratings for items and (2) recommend events that represent requests for recommendations for a target user. All input events are point events and use a common schema (Timestamp, StreamId, UserId, ItemId, Rating), with StreamId values of 0 and 1 denoting an update and recommend event respectively (in the latter case, ItemId and Rating are null). Recommend events can either be start-edges, which denote the registration of a push-based recommendation request, or end-edges, which denote de-

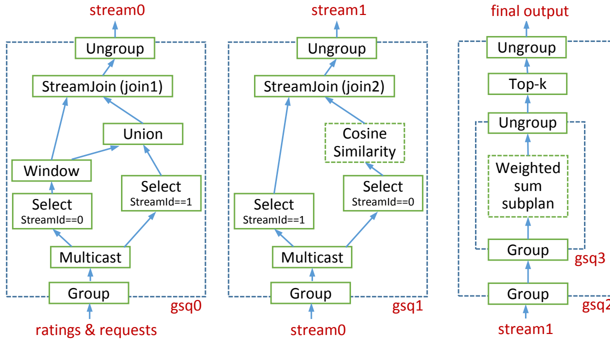


Fig. 2: Continuous queries for a recommender system.

registration. One-time requests use a point recommend event. We describe the approach as two separate phases:

1) **Model Building**. We first perform a GSQ (gsq0) by UserId to compute pairs of item ratings by the same user. For each user, we window their rating events to limit historical ratings from contributing to the current recommender model. For each rating for item Item1 by user User1, we perform a self-join using a StreamJoin (join1) to produce events (UserId, Item1, Rating1, Item2, Rating2) for every pair of items rated by User1.
   These events from join1 are fed into the next GSQ operator (gsq1) with grouping key Item1, where we compute the cosine similarity vector for Item1 (shown as a dotted box inside the sub-query of gsq1). This vector contains, for each other Item2 that pairs with Item1, an aggregate cosine similarity score across all users for that pair of items. It produces a stream of (Item1, SimScoreVector) events. Each similarity score is computed in an incremental manner using a combination of sum and project operations. These events serve as input to a StreamJoin join2 still within gsq1 that effectively holds the model in memory (in the right synopsis) for future scoring during recommendation generation, which is discussed next.
2) **Recommendation Generation**. When a new recommendation request event arrives for user User1, we first send the event to join1 inside the GSQ gsq0, in order look up the previously rated items (Item1) for User1. The results are grouped by Item1 in GSQ gsq1 where, for each item Item1, we look up similarity scores of related items using join2, and output, for every item Item1 rated by User1, the similarity vector of Item1 with every other item (say Item2) that is similar to Item1.
   These events are re-grouped by UserId using GSQ gsq2. For each user, we group by the second item (Item2) using GSQ gsq3, and an aggregate (weighted sum) is used to compute the predicted recommendation score of Item2 for User1. Finally, we use a Top-k aggregation operator to generate the final recommendation for User1.

Any change to either items rated by the user or to the model itself causes the query to produce an update to the top-k result.

Discussion. Several points are worth noting from this example. First, all operations are data-parallel and partitioned using GSQs. Second, the internal state of sub-queries within each GSQ can be huge. For example, if we never expire old ratings, the system needs to hold all historical rating pairs in memory. Third, the GSQ group keys are natural candidates for the granularity of cold state detection as there is natural skew in access characteristics across the group keys (e.g., some items and users are hot or lukewarm, while the rest are cold). Fourth, even within a group, there are specific access characteristics for each operator. For instance, recommend events access the opposite join synopsis of join2 in a read-only fashion. Further, point-recommend events do not update the corresponding join synopsis, whereas edge events update it in a write-only fashion (they do not need to read prior recommend events). Later sections will show how ICE generalizes and exploits each of these features to reduce memory footprint without sacrificing high performance.

B. *Behavioral Targeting Advertising*

In prior work we demonstrated how behavior targeted advertising applications can be expressed using streaming queries [1]. Briefly, ad impressions, ad clicks, and search data is used to eliminate data related to spurious users in a bot elimination phase that uses a GSQ by UserId. The output is used to generate user behavior vectors (by UserId) that are joined to click information to serve as input to a dimensionality reduction and learning sub-plan that groups by ad id. The resulting models are used to score individual users when needed, by joining their user behavior vector to the model and generating an ad click probability for each active ad in the system. The points noted previously for the recommender system hold for this application as well, as they do for many other streaming applications we have encountered in practice.

IV. ICE ARCHITECTURE

Figure 3 depicts the ICE architecture. ICE consists of two main components, highlighted in light gray. (1) A cold state manager (CSM) located outside the core stream engine that tracks group access statistics and classifies cold state that can be pushed to secondary storage; (2) migration-aware stateful operators capable of migrating groups to and from secondary storage. These two components represent a clean logical/physical split. The CSM deals only with logical group ids, and knows nothing about their physical implementation. Meanwhile, operators within a GSQ deal with a group's physical implementation. Operators are responsible for migrating a group to and from secondary storage. The CSM and stream engine communicate using control events. The stream engine sends the CSM a "hit" event to register an access for a group. The CSM sends the stream engine a "migrate" event to notifying a particular GSQ that a group should migrate to secondary storage.

Cold State Manager (CSM). The cold state manager (CSM) is responsible for identifying cold state (i.e., groups) and notifying the system about which cold groups can migrate to secondary storage once the system nears its memory limit. The CSM consists of three sub-components: (1) Access statistics storage maintains lightweight statistics for each group in the system; (2) A cold state classifier is occasionally invoked that uses the access statistics to identify cold groups in the system; (3) A set of event queues (one per GSQ) allow the CSM to receive group access events and send migration events to the GSQ branches.

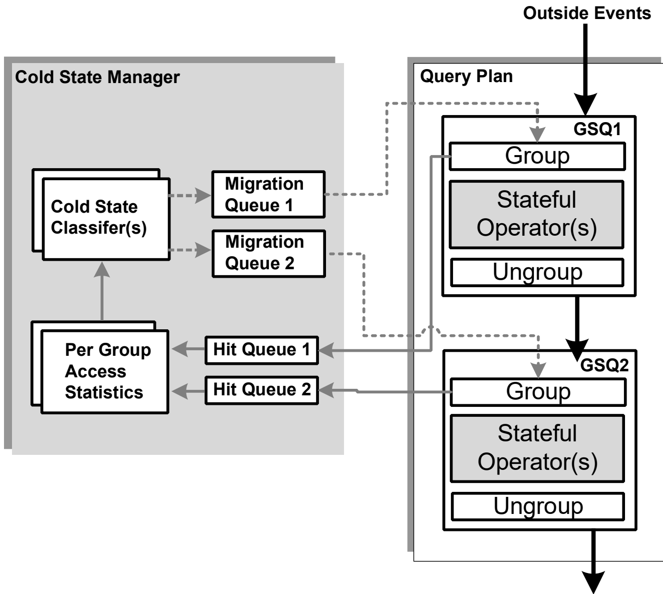


Fig. 3: ICE architecture

The CSM is a separate component located outside the core engine. We chose this design for several reasons. First, it removes the CSM logic from the systems critical path (the query plan). Second, we can offload CSM duties to a separate core, socket, or machine if necessary. Third, it allows for a flexible implementation; pieces of the CSM, such as the cold state classifier, can be swapped out without modifying the stream engine internals. We detail the CSM in Section V.

Migration-aware stateful operators. ICE uses a decentralized architecture, where each stateful operator within a GSQ takes responsibility for migrating state associated with a group to and from secondary storage. This design choice is crucial, as individual operators can best leverage their specific semantics to optimize migration. Logically, upon receiving a "migrate" control event, an operator must serialize group state and write it to secondary storage. Likewise, if the operator must access group state when it is on secondary storage, it must logically read the state back into memory.

Our approach to stream operators managing state on secondary storage is different from operators in an RDBMS. RDBMS operators optimize for the case where most data is disk-resident. Conversely, stream operators optimize for memory-based data. We assume that once an operator spills cold groups to secondary storage, the group will rarely be accessed again (or at least for a long period of time). If and when the group does becomes hot again, it is brought back into memory and will likely remain there for an extended period.

The operators interface with a log-structured storage (LSS) layer that treats storage as an append-only log. Our LSS design diverges from traditional log-structured storage [11] in two ways. (1) Delta updating. We write delta updates (instead of re-writing full state) in order to reduce write traffic. (2) Self-cleaning. Since most stream events naturally expire (i.e., have a finite RE), our LSS automatically garbage collects old state written prior in the log by simply moving the log head forward past all expired state. This is much more efficient than traditional log-structured garbage collection that must reclaim space by copying active state forward in the log. We discuss the details of migration-aware operators, the LSS, and several operator optimizations in Section VI.

V. Cold State Manager

The purpose of the cold state manager (CSM) is to identify cold state and notify the appropriate streaming operators that this state should migrate to secondary storage. The CSM is completely separate from the SPE and runs on a separate thread(s). It knows nothing about physical layout of groups in the system, it only knows about logical group ids. The CSM communicates with the stream engine through control events.

The CSM receives notification of a group access by receiving a hit event on a queue from the GSQ performing the access. It uses hit events to maintain statistics for each group in the system (across all GSQs). Once the SPE nears its memory limit, the CSM performs classification using the access statistics to identify cold groups. For each cold group identified, the CSM enqueues a migrate event on an outgoing queue to inform the appropriate GSQ that it should migrate the group to secondary storage.

As mentioned in Section IV, the primary CSM components are event queues, access statistics storage, and a cold state classifier. In our current design, the CSM maintains separate queues, statistics, and classifiers for each GSQ in the system. Our decision to partition the CSM components per GSQ is primarily due to architectural and threading flexibility. This is because an SPE can parallelize a query plan by scheduling each GSQ on a separate thread. We therefore need a way to scale the CSM to the number of GSQs. For instance, we can use a single CSM instance on a single thread that services all GSQs. This configuration is useful if there are a few very active GSQs in the system. However, we can easily use a dedicated CSM thread for every GSQ in a plan. This configuration is useful for large, complex query plans with multiple parallel GSQs. A hybrid configuration is also possible. The rest of this section covers the details of each main component in the CSM.

A. *Event Queues*

The CSM maintains two event queues for every GSQ. (1) An incoming hit queue, where the GSQ enqueues a control event containing the logical group-id that signifies an access to that group. (2) An outgoing migration queue, where he CSM enqueues a control event for the GSQ containing the cold group-id that should migrate to secondary storage. For performance reasons, we batch control events in both directions.

Maintaining separate events queues for each GSQ in the system has two advantages: (1) *No synchronization*. Each GSQ may run in parallel. Maintaining a single queue for receiving messages would require a single-reader, multiple-writer model that requires synchronization if a CSM services multiple GSQs. Using separate queues allows us to use a single-reader, single-writer model with synchronization-free queues; this approach achieves great performance in a memory-bound system. (2) *Direct messaging*. Maintaining a single outgoing queue would require a migration event to flow through the entire plan until it reached the correct GSQ, whereas separate

queues allows the CSM to directly communicate with the appropriate GSQ.

B. Access Statistics Storage

The CSM maintains group access statistics for all GSQs in the query plan. We currently maintain a separate statistics dictionary for each GSQ containing (group-id, statistics) pairs. The statistics entry is determined by the cold state classification technique implemented in the CSM. We do not propose a new classification technique in this work. Instead, we implemented and tested three well-known classification schemes from the caching literature: clock (LRU), least-frequently used (LFU), and exponential smoothing. Since the CSM is decoupled from the SPE engine, adding a new classification technique to ICE is simple.

Each GSQ reports group accesses using a "hit" control event, as described in the previous section. Upon receiving the hit event from a GSQ, the CSM finds the statistics dictionary for that GSQ and updates the appropriate statistic(s). For example, if the classifier for GSQ uses a clock technique, the statistics update requires (re)setting a single reference bit. Meanwhile, a classifier using exponential smoothing must update the access time and current exponential smoothing score for a group. For all classification techniques we tested, the statistics required were lightweight, so we do not believe statistical maintenance or storage will be a significant overhead in practice.

C. Cold State Classification

When system memory reaches a given threshold, ICE invokes the cold state classifier to identify cold groups. ICE maintains a separate classifier for each GSQ, and the classification technique used is interchangeable. This means we can employ different techniques for each GSQ in the query plan. As mentioned above, we do not propose a new classification technique. Our framework is flexible enough to use several well-known, practical techniques from the caching literature.

The CSM monitors system memory, and invokes a classification step once memory nears a predefined system threshold. The goal of this process is to identify enough cold groups such that the system reduces its memory usage by a predefined threshold M%. Given a set of GSQ operators in a query plan G_1, G_2, ..., G_n, we require that each operator migrate some percentage (of the total M%) of its state to secondary storage. Our current implementation uses a straightforward but fair weighted policy, whereby each GSQ must migrate an amount of state proportional to the amount of total memory it consumes.

The classification process works in three steps. (1) The CSM begins by determining the target amount of memory T_i each operator GSQ_i must migrate. If there are multiple CSMs servicing a query plan, this step requires a short communication round to determine these values. (2) Starting with GSQ_1, the CSM classifies a small batch of the k coldest groups for that operator. The CSM then sends GSQ_1 a batch of migration events containing these cold group ids. The CSM continually polls memory usage while GSQ_1 migrates groups. If the last k groups sent is not enough to reach GSQ_1s target T_1, the CSM repeats this step, notifying GSQ_1 that it should migrate k more groups. (3) Once GSQ_1 has migrated enough groups to reach its target T_1, the CSM repeats step 2 for GSQ_2. This process repeats until the CSM has notified all GSQs. While relatively simple, we found this weighted method to be quite effective. We plan to explore more advanced techniques as future work. The classification decision is "local" to each GSQ. That is, we perform classification for a GSQ one at a time. We do not consider "cross-GSQ" correlation when classifying cold state.

VI. Migration-Aware Operators

This section describes role and implementation of migration-aware stream operators within ICE. The operators are responsible for the physical aspects (e.g., storage, migration) of cold data.

When an operator receives a migration event from the CSM telling it which group is "cold", it must: (1) locate the group in its in-memory data structure; (2) prepare the group state (e.g., stream events) for migration, e.g., consolidating and compressing the state if possible; (3) write the state to secondary storage. All operators use the log-structured storage layer to persist state on secondary storage (e.g., disk or flash). The operator must also remember a marker for each groups state on cold storage; this is a 8-byte word representing the offset of the state on the LSS. The operator uses the marker to seek and read back the state if it needs to be brought back into memory.

The rest of this section describes the details of the migration-aware operators. We begin by describing the log-structured storage layer used to persist state on secondary storage. We then detail the implementation of the streaming join, and discuss how to implement other migration-aware stateful operators.

A. Log-Structured Storage

Each operator interfaces with our log structured storage (LSS) layer for reading and writing cold state. This section describes our LSS and two optimizations we implement that make it efficient in a stream processing scenario: (1) *delta updates*: appending only incremental delta records to previously written state and (2) *automatic garbage collection*: taking advantage of event expirations to automatically recycle the old section of the log. To our knowledge, exploiting event expirations for automatic garbage collection is a technique unique to our streaming context.

1) Log Structuring Basics: Log structuring treats storage as an append log [11]. All writes are performed at the tail of the log. There are no updates in place, meaning when a new storage block is written (e.g., page, record), its previous version (back in the log) becomes inactive, or "garbage". For this reason, traditional log-structuring requires garbage collection to reclaim inactive state. This allows the log to "wrap around" and recycle file space. As we will see, due to delta updates and natural event expiration, our LSS can avoid much of this expensive garbage collection step.

The main advantage of log structuring is write performance since all writes are sequential. It is also common to use large write buffers to ensure full use of I/O bandwidth. This is especially important in our scenario since cold group migrations

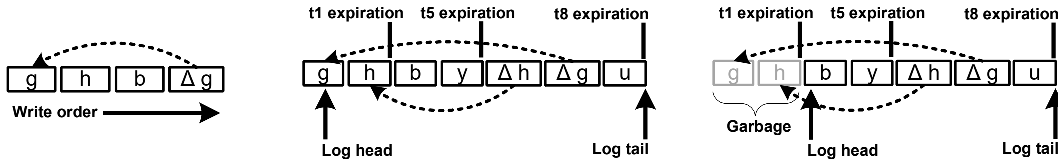


Fig. 4: Log structured garbage collection: delta updates (left), expiration markers (middle), automatic garbage collection (right).

(writes) appear on the critical path within the stream operators. Large in-memory buffers imply that an operator is unlikely to block during migration, e.g., waiting for a free buffer, since buffers drain fast due to sequential I/O.

Log structuring is especially beneficial for flash SSD storage [12]. Flash does not allow updates in place to physical pages. It requires a flash translation layer (FTL) to translate logical page addresses (given by the client) to an internal physical page address. Thus data on a physical page may relocate many times. This is an expensive process; however, log structuring obviates the need for such a mechanism.

2) Delta Updates: One straightforward way to implement log structuring is to always write full operator state back to secondary storage (e.g., a complete group synopsis in the case of join). This approach leads to wastefulness since very little state may have actually changed, which inflates log usage, increases write amplification, and also necessitates garbage collection to clean up old versions earlier in the log.

Instead, our LSS writes deltas that describe incremental updates to group state already on the log. Writing deltas requires the operator to pass the LSS two parameters: (1) serialized bytes containing one or more delta record that describe an update to state already on the LSS, or, if this is the first write, the entire serialized group state; (2) the existing state's current offset on LSS (null if we are migrating a group for the first time). The LSS writes the bytes along with a back pointer to the state already on the log (the provided offset). The LSS returns the offset of the newly written delta bytes that the operator uses to remember the cold state position in its marker. This new offset represents the "root" of the state on the LSS.

Figure 4 (left side) provides an example of the LSS after writing a delta for a group g already on the LSS, where the offset of Δg is the states "root" on the LSS. Since deltas cause a group's state to be chained back on log, reading state into memory may require traversal of the chain. This traversal causes random seeks. While this may be a problem on systems with hard disks, it is less of a problem on flash for which there is less penalty for random reads [12], [13], [14].

3) Automatic Garbage Collection: A benefit of using log structured storage in an SPE is that we can take advantage of event expiration to implement automatic garbage collection. The basic idea is to mark state on the LSS with a monotonically increasing expiration timestamp. Ensuring monotonicity means all state before a marker t is guaranteed to have an expiration time less than or equal to t. Once time advances in the SPE, the head of the log moves forward past all markers that have expired, effectively recycling the space once used by the expired state.

The advantage of this technique is that our LSS does not need to implement a heavyweight garbage collection mechanism to perform space reclamation. This a very big win, since garbage collection is an oft-cited drawback of log structured storage, which requires finding "active" state far back in the log and copying it forward to the tail.

We note that if events never expire (i.e., they have infinite right endpoints) then automatic garbage collection is not possible and we must use traditional log structured garbage collection. However, many applications (such as those outlined in Section III) consist of events with large lifetimes (windows) with finite right endpoints, so automatic garbage collection is possible in these scenarios.

Implementation details. When an operator migrates group state (i.e., stored events) to the LSS, it marks the state with an expiration time t_e that is the larger value between (1) the largest RE of all events in the migrating group, or (2) the directly previous expiration value written to the LSS. Selecting t_e this way ensures expiration times are monotonically increasing on the LSS (a requirement). Expiration markers may be written at short intervals (e.g., for every group) or larger intervals (e.g., on every buffer write boundary). Writing at shorter intervals implies garbage collection advances in smaller steps more often, and vice versa for larger intervals. Figure 4 (middle) provides an example of our LSS with three expiration markers, t_1, t_5, and t_8.

Garbage collection occurs automatically once time advances within the stream engine (e.g., due to a punctuation [9], [15]). In ICE, once the stream engine generates (or receives) a punctuation, it notifies the LSS of the current time tn. The LSS will then move the head of the log forward to the furthest expiration marker te such that $t_n \leq t_e$. Figure 4 (right) provides an example of automatic GC for or running example, where time advances past t_1. In this case, the log head moves forward to marker t_1 recycling space used by groups g and h.

With automatic GC, back pointers on delta records may point past the log head into recycled (garbage) territory. This is alright, since it is no longer needed in memory since all recycled state is guaranteed to be expired and no longer contributes to the query answers. When an operator must read back state from the LSS, it will only read the valid delta state in the active section of the log. In essence, one can view our GC mechanism as automatically retracting expired events from the LSS. This also has the advantage of avoiding transfer of useless (i.e., expired) state from the LSS back to memory.

B. Migration-Aware Streaming Join

As a concrete example, this section describes the implementation details for a migration-aware streaming join operator. A join is one of the most state-heavy operators in an SPE. As described in Section II, when a streaming join is inside a GSQ, it naturally partitions its left and right join synopses by the grouping key. It performs the join by applying the hash function to the group key (stored on the incoming event),

which in turn points to the state within the synopsis necessary to perform the join. Our recommender system application provides an example of a streaming join (c.f. Section III). The join within the first GSQ operator gsq0 partitions its state by user id (the grouping key). The state stored for each user is a set of update events that correspond to user rating history (e.g., movies previously rated). Update events are stored in the right and left synopses to generate all pairs of ratings, and join with all incoming recommend events from the right. Since recommend events are point events, the right join synopsis does not hold these events. Likewise, GSQ operator gsq1 partitions by Item1, and stores for every Item1, the model (similarity vector) associated with that item. The left synopsis for join2 is empty as it receives only recommend events, whereas the right synopsis contains these per-item model vectors.

When a join receives a migration control event containing a group id, it is logically required to migrate all state associated with that group id from the left and right join synopses. In our example, the group id for gsq0 represents a user, thus the join would migrate update events for all infrequent users (i.e., ratings history) from the join synopses. Recommend events are not migrated as they are not present in either join synopsis. In case of gsq1, parts of the model (for infrequent items) get migrated to cold store by ICE. We next discuss implementation details, followed by optimizations that improve performance significantly by making use of operator-specific semantics.

1) Implementation: Each group hash bucket within a join maps to a dynamic array that stores references to events that belong to that group. Migrating a group state involves three main steps. First, we serialize the events from our dynamic array into a compact byte format. Second, we append the bytes to an in-memory LSS buffer that is flushed to secondary storage once full. Finally, we "mark" the entry in the hash table by replacing the event array with an eight-byte entry. We use the high order bit of the marker as a flag to notify that the state is on secondary storage. We use the rest of the bits to store the location of the bytes on the LSS.

If an access comes for a group that is on secondary storage, we request its state from the LSS at the offset stored in the marker. We then read the compressed bytes from secondary storage and reconstruct the event array in memory. The entry remains in memory until it becomes cold again.

2) Operator-Specific Optimizations: There are several optimizations we implement in order to reduce I/O traffic from the operators to the LSS and improve performance. We discuss these optimizations in the context of the streaming join operator. In general, similar optimizations are possible for other stream operators as well.

Writing deltas instead of whole state. As discussed earlier, an operator writes only deltas back to the LSS that represent incremental changes to a group's state. Using our example, say we received a recommend event that caused a group g to be read from the LSS. Later, two rating events r_1 and r_2 are added to g. Now, if g subsequently migrates, the operator will only serialize and write r_1 and r_2 (instead of g's entire state) back to the LSS. This optimization works in concert with our LSS delta updating functionality. To implement this optimization, the operator cannot discard its marker containing the group's location on the LSS. The operator must pass this offset to the LSS so that it can "link" the delta back to the previous state on the log.

Migrating dirty state only. This classic optimization avoids writing state back to storage that is "clean" (i.e., not updated). This optimization is especially useful for joins that store state that is "read-mostly". This is exactly the case in our recommender query example for gsq0: a recommend point event probes for a group g containing user rating history. This is a read-only access and the point event is not stored in the opposite synopsis. If all subsequent accesses to g are read only, we can avoid writing g back to the LSS if it had been written before. Note that we make the dirty state determination on a per-synopsis basis, which provides maximum flexibility in cases where only one of the two synopses is read-mostly (e.g., contains slow-changing reference data). In case of gsq1, the right synopsis of join2 contains model information that may be read-mostly, depending upon how often the model elements are refreshed.

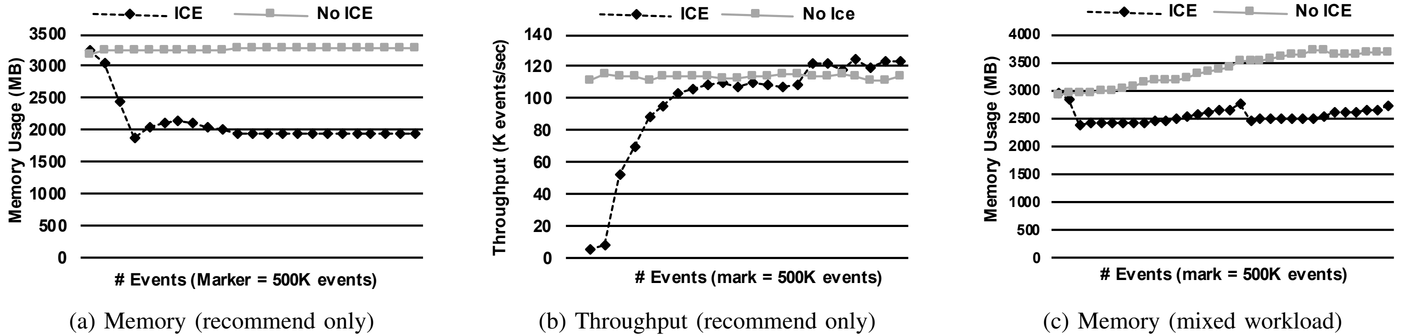
Add-only. Another optimization we can exploit is knowledge that a group access is "add-only". In this case, we can avoid reading the group's state if it is currently on the LSS (and not in memory). In this case, we simply add the event as a delta marker to the in-memory hash bucket. This is especially relevant to the streaming join since the non-probe side of the synopsis is add-only.

C. Implementing Other Operators

1) Streaming Set Difference: Similar to streaming join, the set difference operator takes two streams as input. It produces an output stream that consists of the left input stream, limited to time intervals where no matching event exists on the right stream. Set difference behaves similarly to a join, with left and right synopses that are symmetrically probed. Thus, all the optimizations discussed earlier apply.

2) Top-k: Top-k maintains a list of active events sorted by the ordering attribute for each group. Each incoming event either adds to or removes from this ordered list. The ordered list is usually implemented as a red-black-tree. Note that all active events need to be maintained as any event could later enter the top-k due to other events expiring (each event may have a different right endpoint). We use the LSS to store the red-black-tree associated with individual groups. The delta write optimization is very useful as the number of active events in a group may be very large.

Another optimization relevant to top-k is partial state reconstruction. The basic idea is that when retrieving state from the LSS, we can push additional predicates that indicate how much state is sufficient to be brought back into memory to compute the new top-k result. Assume for simplicity that ordering attribute values are unique (dropping this assumption is straightforward, but is omitted for brevity). We track the value of the current k^{th} ranked active element. If we receive m new end-edges with the same RE, we only need to bring back the current top k+m' elements, where m' is the number of new input events with ordering attribute less than that of the current k^{th} ranked element. Further, if we receive m new start-edges with the same LE, we only need to retrieve the top k-m' elements from LSS.



(a) Memory (recommend only) (b) Throughput (recommend only) (c) Memory (mixed workload)

Fig. 5: Memory and throughput experiments.

3) User-Defined Operators: The user-defined operator APIs usually support serialization and deserialization in order to achieve high availability. This support can be leveraged to handle cold state for such operators. In addition, we have optimized delta-serialization APIs with dirty state indication that, if implemented by query writers, can support our LSS optimizations discussed earlier.

VII. Discussion

The previous sections discussed ICE and its novel mechanisms for identifying and migrating cold data to secondary storage. Page caching [16] is a well-known technique for managing memory and disk-resident data. This section discusses why caching is not appropriate for our environment and why ICE is necessary.

Space overhead. SPEs are built assuming all data is memory resident. Systems that use page caching (e.g., an RDBMS) are built assuming data may be paged in or out at any point in time. Page-based indirection through buffer pools requires significant space overhead. Pages require extra metadata (e.g., offset arrays) and may contain unused space leading to fragmentation. Page organization is especially problematic for SPEs due to event expiration. Expiring (deleted) events consistently create free space holes on pages, leading to unused space and requiring constant defragmentation to reclaim this space. For this reason SPEs use small event-based granularities. ICE allows the stream engine to continue using event-base storage organization since the operators are responsible for physically organizing and transferring events to and from cold storage.

CPU Overhead. Paging requires a significant amount of overhead on the systems critical path to perform page space management, e.g., updating cache statistics, storage (re)allocations, and defragmentation. ICE removes cold state classification from the systems critical path by performing statistical updates and classification lazily in a separate CSM. This processing can take place on a separate thread or another machine, if necessary. ICE adds CPU overhead to the critical path in two dimensions. (1) Operators enqueueing hit events for the CSM. While lightweight (i.e., no synchronization), ICE can easily reduce this overhead further by enqueuing only a sample of events. (2) Operators receiving a migration event must write cold state to the LSS. Since the LSS uses large in-memory buffers this write will have small to modest overhead. This overhead can be further decreased by making migration events low priority, for example, by processing N data events (i.e., events contributing to the query result) for every migration event. In our experience migration events tend to be bursty (sent in bulk when the system reaches a memory threshold). Making migrations low priority will amortize write overhead over the period between “bursts”.

Classification granularity. Page-based organization manages hot and cold data at the granularity of a page. If used in a stream engine, events would likely be placed randomly on free pages as they arrive. In this case, page-based organization would not be as precise as group organization in classifying cold state. ICE, on the other hand, classifies cold state at the granularity of a group. As we have seen, groups naturally exhibit real-world access skew (e.g., users or items in a recommender system). Exploiting group access patterns allows ICE to be precise in the state it classifies and migrates to secondary storage. This allows the SPE to maintain good performance while shedding memory overhead.

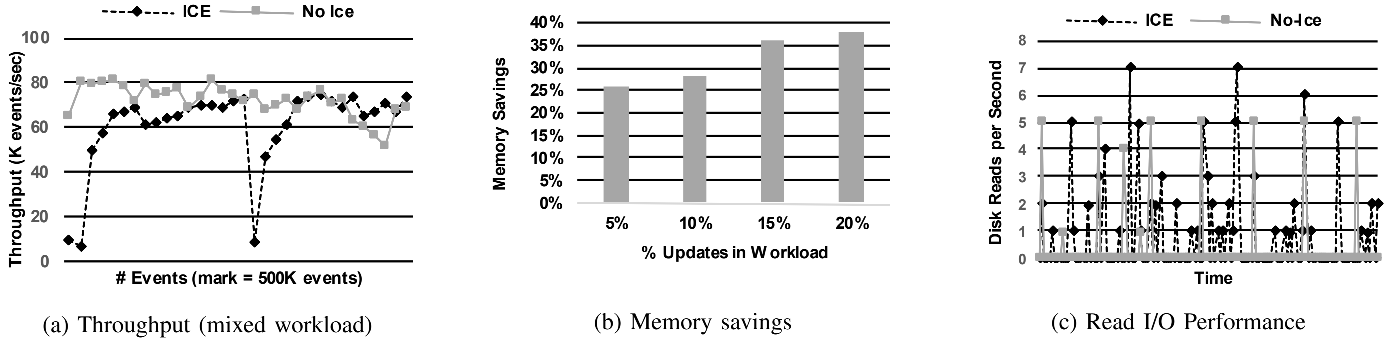
VIII. Experiments

A. Experiment Setup

We implemented ICE to work with a test version of Microsoft StreamInsight. We ran all experiments on an Intel Core2 8400 at 3GHz with 16GB or RAM. We use a Seagate Barracuda 1TB hard disk (7200 RPM over SATA 3 Gb/s interface). Since our experiments use a hard disk, our numbers represent a lower bound on ICE performance (e.g., paying random read penalties that we would not pay when using Flash SSD).

We make use of two data sets in our experiments. (1) Synthetic. This data consists of 20M events for 300K groups; we generate the group id for an access using the Zipf distribution. (2) MovieLens. The rest of our experiments use real data taken from the MovieLens recommendation system. This data contains 10M ratings from 70K users for 10,600 movies. Each rating contains a timestamp, which we use to submit update events to the system in timestamp order. We generate recommend events using the following method: 85% of the events come from 15% of the users (the hot set) while the remaining 15% come from the 85% of the users (the cold set). We use the recommender query described in Section III-A to evaluate system performance on the MovieLens data.

ICE uses the well-known clock replacement policy for identifying cold groups. We chose clock due to its benefits



(a) Throughput (mixed workload)

(b) Memory savings

(c) Read I/O Performance

Fig. 6: Throughput for mixed workload, memory savings, and read I/O experiments.

in space and memory overhead, as well as relative accuracy (though this piece pluggable, as mentioned in Section V).

B. Recommender Workload

In this section, we experimentally evaluate ICE using the real MovieLens data run on the streaming recommender query described in Section III-A. We compare our streaming system with ICE and without ICE (abbr. No ICE). Our primary metrics are throughput and memory usage for both implementations. For both workloads, we first initialize the system by feeding 7M update events (out of 10M) to build an initial recommender model. We then start measuring performance for a given workload, which is a mix of both update events and recommend events.

1) Recommend-Only Workload: We begin by evaluating a recommend-only workload where we only send recommend events to the system. We set the memory limit to 2.6GB. This is sufficient to store the hot set of groups (recall from Section 3.1 that a major overhead is storing per-user rating history in GSQ gsq0). Figure 5a reports the memory usage of both approaches. Over time, the No ICE method memory usage remains static, this is expected since new state (i.e., ratings) does not enter the system. ICE initially migrates roughly 1.5GB of cold state to secondary storage (left-hand downward slope). After this migration, memory usage is stable, meaning ICE is capable of enforcing the memory limit imposed on it. This is significant, since the gap between ICE and No ICE represents a 41% drop in memory usage.

Figure 5b reports the throughput for this experiment. Since ICE requires an initial bulk migration, the throughput is initially well below No ICE. After the initial migration is complete, however, ICE equals and eventually surpasses the throughput performance of No ICE. We believe this performance improvement is due to better use of the memory hierarchy leading to better cache locality within the (grouped) streaming join.

2) Mixed Workload: This experiment evaluates ICE using a mixed workload of both update events and recommend events. The update to recommend ratio is 1:9. Figure 5c reports the memory usage for both approaches. Memory usage for the No ICE approach increases during query runtime as new rating state enters the system. Meanwhile, ICE initially migrates cold state to secondary storage to meet its memory limit (left-hand side of graph). During runtime memory usage increases and nears the limit. The jump in the middle of the graph represents ICE performing cold state migration again to maintain the limit. On average, ICE reduces memory usage by 28% for this workload.

Figure 6a reports the throughput for both approaches. The throughput for the No ICE approach slowly decreases during query runtime. This performance decrease is due to the addition of update events to the workload. As more ratings are added to the system, the operators must accommodate this state by increasing capacity of their internal data structures, leading to poorer performance. ICE shows an initial drop in throughput due to its initial migration. However, its throughput eventually matches that of No ICE. The drop in throughput during the middle runtime is due to the second migration. We note that this drop represents the worst-case degradation in throughput since our prototype currently processes all migrations within each operator before processing new events; this can be greatly improved by alternating migration and data events in within the operator. ICE throughput quickly recovers from its migration and its performance eventually surpasses the No ICE approach. Like the recommend-only experiment, this improvement is again to better use of the memory hierarchy.

3) Effect of Varying Rating Updates: The previous experiments show that ICE is capable of maintining performance commensurate to that of a No ICE approach while using less memory. We now further explore memory savings as the state update rate of a workload increases. This experiment increases the percentage of update events in the recommender workload and reports the memory usage difference at the end of the run (workloads are run for the same amount of time). Figure 6b reports the results. As the number of rating events increases, a stream system without ICE must maintain more state during the workload runtime. The benefit of ICE is magnified as the update rate increases, as it is able to migrate cold state to secondary storage, while the No ICE approach requires holding new state entirely in memory. For each run, the throughput of ICE remains on par with (and sometimes surpasses) No ICE.

4) I/O and CPU Usage: This section studies the I/O and CPU overhead caused by ICE. These experiments employ a performance monitor during the run of the mixed workload that samples disk reads and CPU usage.

Read I/O. Figure 6c plots the disk reads per second for both approaches. Again, the No ICE process does not perform actual reads, so its plot represents the ambient read traffic in

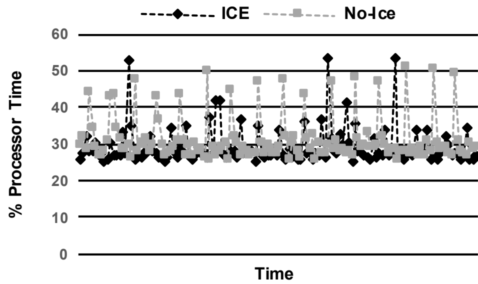


Fig. 7: CPU usage.

	Average Disk Writes/Sec
ICE with Read-Only Synopses	30.156
ICE without Read-Only Synopses	84.464

TABLE I: Effect of read-only join synopses.

the system. ICE performs occasional disk reads to perform lookups and updates on groups that had previously migrated, however its read rate is not much larger than No ICE. On average ICE performs 0.734 reads/second over the lifetime of the workload. This is acceptable compared to No ICE with that performs 0.311 reads/second on average. This low read-rate also implies ICE correctly identifies cold groups, since very few cold groups must be brought back into memory.

CPU. Figure 13 plots the CPU usage for both approaches. There is no large divergence between the two approaches. No ICE exhibits an average CPU usage of 29.42% over the entire run, while ICE exhibits usage of 31.26% on average. The CPU overhead for ICE involves cold data identification logic (i.e., managing the clock), as well as extra overhead to serialize groups for I/O. We consider this CPU overhead acceptable.

C. Optimizations

1) Read-only Join Synopses: To further test the overhead of the read-only optimization (Section VI), we ran the mixed recommender workload on ICE with the read-only join synopsis optimization both enabled and disabled. Table I reports the average write I/Os per second observed over the lifetime of the workload. This optimization is able to reduce disk write I/O by 64%. We consider this a very good improvement and a cornerstone of the good overall performance of ICE.

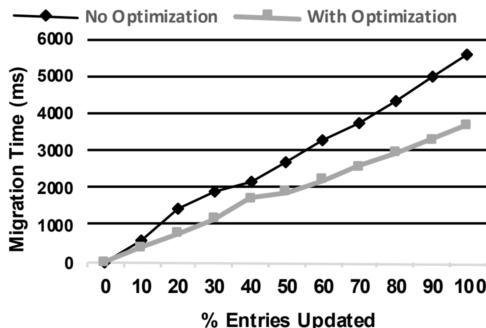


Fig. 8: Effect of delta updates.

2) Effect of Delta Updating: This experiment tests the efficiency gains of delta updating our log-structured store instead of writing whole state on every migration. To isolate the effects of delta updating, we perform a micro-benchmark that fills a join synopses with 5M ratings spread across 70K users, where each user corresponds to a group partition. It then forces all groups to migrate to secondary storage and subsequently reads them all back into memory using a read-only mode. It then updates Y% of the groups (y-axis) and measures the time to migrate (write) all groups back to storage (x- axis). Figure 8 provides reports this migration performance for ICE with and without the delta write optimization. At 0% updates, there is no write I/O overhead due to the read-only optimization. As the update percentage, the effect of writing deltas instead of whole pages is apparent in the performance divergence between the two approaches. On average, the delta write optimization leads to a 30% performance improvement over the baseline.

IX. Related Work

Page Caching. Page caching [16], [17] is a well-known method for managing cold data. However, as discussed in Section VII, page caching is not a good solution for our setting for the following reasons reasons. (1) *Space overhead*: stream systems are main-memory optimized and cannot tolerate page-based indirection for performance reasons; (2) *CPU overhead*: adding page-based (re)allocation and defragmentation overhead to a stream systems critical path leads to unpredictable performance; (3) *Sub-optimal organization*: page-based organization assigns data randomly to pages as it enters the system, which leads to sub-optimal separation of cold and hot data. ICE avoids all of these pitfalls.

Delta Updating. Delta updating on log-structured storage has been explored in the context of access methods [13], [14]. We use delta updates in the context of managing events on our LSS. Writing deltas to LSS allows us to implement a novel automatic garbage collection scheme that avoids the pitfalls of traditional log-structured garbage collection.

Non-page caching techniques. Web caching [18] classifies and caches data at various (non-page) granularities (e.g. web page, arbitrary objects). This work creates various novel caching techniques and policies that incorporate both document size and network latency. There has also been recent work on managing cold data in main-memory OLTP engines [19], [20], [21]. This work takes advantage of OLTP access patterns in order to partition cold and hot data at record granularity. Our work differs dramatically by studying cold state and scalability in SPEs that have very different query semantics than web caches or OLTP engines. Our work also classifies and migrate state at a group granularity that is unique to SPEs.

Log-structured storage. Log-structured storage [11] was originally applied to file systems to avoid random writes and maximize disk write bandwidth. The ICE storage layer deviates from traditional log structuring in two novel ways. (1) *Logical delta updates*: Instead of (re)writing whole pages to the log tail, ICE writes only logical delta updates signifying group state changes since the last write. This optimization greatly reduces write traffic and increases the number of updates that fit into a single write buffer. (2) *Automatic garbage*

collection: ICE takes advantage event expiration in SPEs (i.e., finite REs) in order to naturally allow events expire on the log and thereby reclaim log space. Thus in most cases, ICE avoids the expensive garbage collection process common in log-structuring.

Scaling Stream Processing Systems. ICE addresses the fundamental problem of scaling an SPE in the face of memory pressure. Existing classes of techniques to address this problem include: (1) *Multi-machine scale out* [22], [23]. While scale out solutions for SPEs are important, we view this work as orthogonal to ours; we provide the benefit of higher scalability on a single node by better balancing CPU, memory, and I/O resources on the machine, and exploiting secondary storage and the access characteristics common in our target applications. (2) *Leveraging distributed storage*. Systems such as MillWheel [24] allow users to offload stream state to a distributed key-value store. In contrast, we focus on efficiently leveraging local storage (such as SSD), and provide a higher level of abstraction, with physical operators that manage cold state automatically using a cold state manager. (3) *Load shedding* [25], [26]. This method drops stream events to manage load, thereby producing approximate results. For complex big data applications, it can be difficult to estimate the effect of dropped tuples on accuracy. Further, users may wish to get exact answers in many cases. ICE migrates infrequently accessed state (groups) to cheaper secondary storage, allowing exact result generation. ICE is particularly useful in a cloud setting, where economical and balanced use of hardware is desirable. In case applications can tolerate dropped tuples, existing load shedding techniques can be applied alongside ICE for better scalability.

Stream queries accessing historical data. Most SPEs interface with secondary storage (or an RDBMS) in order to query historical data [7], [27]. In this scenario, ad-hoc queries can access (expensive) historical data at any time. Thus, techniques such as data reduction [8] have been proposed to improve access to historical data. Our scenario differs significantly. ICE helps SPEs to scale by offloading cold query internal state to secondary storage, thereby trading off memory for I/O.

X. Conclusion

A stream processing engine (SPE) enables efficient execution of big data applications such as targeted advertising, recommender systems, risk analysis, and call-center analytics. Such applications require the SPE to maintain and process massive amounts of in-memory state. This paper proposes ICE, a novel framework that allows an SPE to scale to the needs these memory-bound applications. ICE exploits the natural access skew exhibited by many of these applications by dynamically offloading state to secondary storage. We implemented ICE in a commercial SPE. Experiments using both real and synthetic workloads reveal ICE reduces memory footprint without sacrificing performance, making it a viable solution for many data-intensive application scenarios.

References

[1] B. Chandramouli *et al.*, "Data Stream Management Systems for Computational Finance," *IEEE Computer*, vol. 43, no. 12, pp. 45–52, 2010.

[2] B. Chandramouli, J. Goldstein, and S. Duan, "Temporal Analytics on Big Data for Web Advertising," in *ICDE*, 2012, pp. 90–101.

[3] B. Chandramouli, J. Levandoski, A. Eldawy, and M. Mokbel, "StreamRec: A Real-Time Recommender System," in *SIGMOD*, 2011, pp. 1243–1246.

[4] B. Gedik, K. Wu, P. S. Yu, and L. Liu, "Adaptive Load Shedding for Windowed Stream Joins," in *CIKM*, 2005, pp. 171–178.

[5] N. Tatbul, U. Çetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker, "Load Shedding in a Data Stream Manager," in *VLDB*, 2003, pp. 309–320.

[6] "MovieLens Datasets. `http://grouplens.org/datasets/movielens/`."

[7] D. J. Abadi *et al.*, "The Design of the Borealis Stream Processing Engine," in *CIDR*, 2005, pp. 277–289.

[8] S. Chandrasekaran and M. J. Franklin, "Remembrance of Streams Past: Overload-Sensitive Management of Archived Streams," in *VLDB*, 2004, pp. 348–359.

[9] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and Issues in Data Stream Systems," in *PODS*, 2002, pp. 1–16.

[10] M. A. Hammad *et al.*, "Nile: A Query Processing Engine for Data Streams," in *ICDE*, 2004, p. 851.

[11] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," in *SOSP*, 1991, pp. 1–15.

[12] B. K. Debnath, S. Sengupta, and J. Li, "SkimpyStash: RAM Space Skimpy Key-Value Store on Flash-based Storage," in *SIGMOD*, 2011, pp. 25–36.

[13] J. Levandoski, D. B. Lomet, and S. Sengupta, "The Bw-Tree: A B-Tree for New Hardware Platforms," in *ICDE*, 2013, pp. 302–313.

[14] ——, "LLAMA: A Cache/Storage Subsystem for Modern Hardware," *PVLDB*, vol. 6, no. 10, pp. 877–888, 2013.

[15] D. Maier, J. Li, P. A. Tucker, K. Tufte, and V. Papadimos, "Semantics of data streams and operators," in *ICDT*, 2005, pp. 37–52.

[16] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The LRU-K page replacement algorithm for database disk buffering," in *SIGMOD*, 1993, pp. 297–306.

[17] T. Johnson and D. Shasha, "2q: A low overhead high performance buffer management replacement algorithm," in *VLDB*, 1994, pp. 439–450.

[18] M. Rabinovich and O. Spatscheck, *Web Caching and Replication*. Addison-Wesley, 2001.

[19] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. B. Zdonik, "Anti-Caching: A New Approach to Database Management System Architecture," *PVLDB*, vol. 6, no. 14, pp. 1942–1953, 2013.

[20] A. Eldawy, J. J. Levandoski, and P. Larson, "Trekking Through Siberia: Managing Cold Data in a Memory-Optimized Database," *PVLDB*, vol. 7, no. 11, pp. 931–942, 2014.

[21] F. Funke, A. Kemper, and T. Neumann, "Compacting Transactional Data in Hybrid OLTP & OLAP Databases," *PVLDB*, vol. 5, no. 11, pp. 1424–1435, 2012.

[22] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. B. Zdonik, "Scalable Distributed Stream Processing," in *CIDR*, 2003.

[23] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. J. Shenoy, "A Platform for Scalable One-Pass Analytics using MapReduce," in *SIGMOD*, 2011, pp. 985–996.

[24] 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," in *Very Large Data Bases*, 2013, pp. 734–746.

[25] B. Gedik, K. Wu, P. S. Yu, and L. Liu, "Adaptive Load Shedding for Windowed Stream Joins," in *CIKM*, 2005, pp. 171–178.

[26] N. Tatbul, U. Çetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker, "Load Shedding in a Data Stream Manager," in *VLDB*, 2003, pp. 309–320.

[27] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, F. Reiss, and M. A. Shah, "TelegraphCQ: Continuous Dataflow Processing," in *SIGMOD*, 2003, p. 668.