Raising the Level of Abstraction for Time-State Analytics With the Timeline Framework

Henry Milner[1], Yihua Cheng[1,2], Jibin Zhan[1], Hui Zhang[1,3], Vyas Sekar[1,3], Junchen Jiang[1,2], Ion Stoica[1,4]
{hmilner,ycheng,jibin,hzhang,vsekar,jjiang,istoica}@conviva.com
[1]Conviva, Inc, [2]University of Chicago, [3]Carnegie Mellon University, [4]UC Berkeley

ABSTRACT

Across many domains, we observe a growing need for more complex *time-state analytics*, which entails context-sensitive stateful computations over continuously-evolving systems and user/machine states. For instance, in video distribution, we want to analyze the total time video sessions spend in a buffering state. We argue that modern data processing systems entail (a) high development time and complexity and (b) poor cost-performance tradeoffs, for such workloads. We make a case for a Timeline abstraction for serving this class of workloads. By raising the level of abstraction using Timelines, we can reduce development complexity *and* improve cost-performance tradeoffs. We demonstrate the early promise in a production-scale video analytics deployment. We posit that the Timeline abstraction is more generally applicable across domains and enables new opportunities for further research.

1 INTRODUCTION

Across many domains (e.g., content delivery, ad analytics, finance, datacenters, telecom, security), we observe the emergence of a class of data analysis requirements, which we refer to as *time-state analytics*, which requires modeling *context-sensitive metrics computed over continuously-evolving state of a system of interest.* More concretely, consider a video distribution scenario (Figure 1), where we want to check whether video sessions using a particular content delivery network (CDN) are having quality issues (e.g., long buffering stalls) [14]. Operators use such analysis to identify and mitigate user experience issues. As seen in Figure 1, computing metrics such as connection-induced rebuffering requires us to model the state of the player and the user to ignore buffering during initialization and after user seeks; i.e., model stateful behavior in context. Similar needs arise in financial (e.g., [11]) or security monitoring (e.g., [29]) applications where analyzing a sequence of transactions made over time can help identify fraud or compromised credentials.

Unfortunately, existing data processing systems, including streaming systems (e.g., [1, 2]), timeseries databases (e.g., [6]), SQL extensions (e.g., [5]), and batch systems (e.g., [12, 34]), are ill-equipped to address time-state analytics. Such approaches do not provide good abstractions for modeling processes evolving continuously over time. Consequently, while in theory such systems can tackle time-state analytics, in practice they entail poor cost-performance tradeoffs and involve significant development complexity. This leads to inefficient workarounds and complex code with semantic bugs, creating a technical barrier for analysts, as we will see in §2.

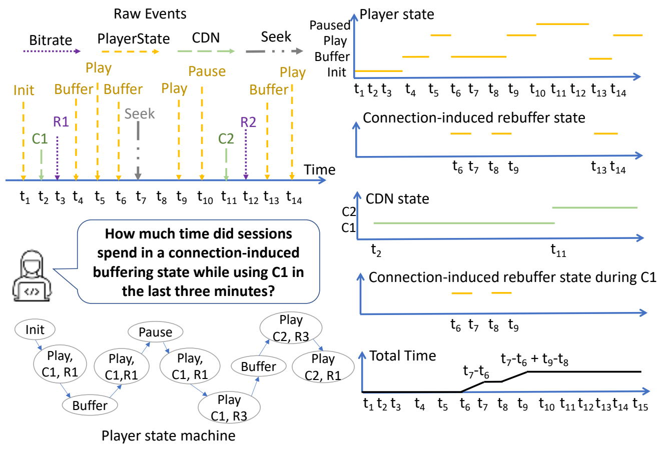


Figure 1: An illustrative example of the challenges of Time-State Analytics in video content distribution. The right-hand side shows how a Timeline mental model can help express this intent more naturally.

We argue the need to *raise the level of abstraction* to support time-state analytics workloads. Indeed, powerful abstractions for handling states and processes evolving continuously over time have been developed in other communities, including functional reactive programming (e.g., [27]), signal processing (e.g., [25]), and formal methods (e.g, [3]). We make an analogous observation here and propose a similar abstraction for data processing in the area of time-state analytics, which we call *Timelines*. The right-hand side of Figure 1 shows an example. At a high level, Timelines allow analysts to more directly model dynamic processes and express the requirements of time-state analytics, in contrast to conventional approaches such as streaming systems (e.g., [1]), timeseries databases (e.g., [6, 26]), and temporal extensions to SQL (e.g., [21, 24]). By raising the level of abstraction and offering a more intuitive mental model, we can enable data analysts to write *simple, succinct, and efficient* queries.

We define a Timeline Algebra defining operations over three basic TimelineTypes: state transitions, discrete events, and numerical values. We show how existing workflows (e.g., SQL, Spark) can be extended to support these operations [1, 5, 8, 35]. As a proof of concept, we have implemented a Timeline workflow and run it in production at a large video analytics company. In comparison with the company's earlier system, pilot deployments show a threefold reduction in cost, an order-of-magnitude decrease in development time, and near-zero semantic bugs. More importantly, it has enabled *evolvability* to support new use cases.

Looking forward, Timeline-based processing opens new opportunities and research questions. First, while prototyping Timeline as an overlay on existing platforms is already performant, pushing

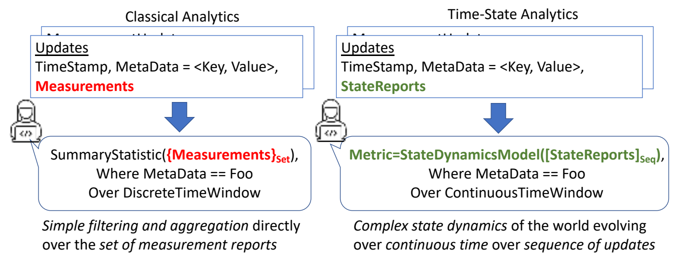

Classical Analytics
Updates
TimeStamp, MetaData = <Key, Value>,
Measurements
SummaryStatistic({Measurements}Set),
Where MetaData == Foo
Over DiscreteTimeWindow
Simple filtering and aggregation directly over the set of measurement reports
Time-State Analytics
Updates
TimeStamp, MetaData = <Key, Value>,
StateReports
Metric=StateDynamicsModel([StateReports]Seq),
Where MetaData == Foo
Over ContinuousTimeWindow
Complex state dynamics of the world evolving over continuous time over sequence of updates

Figure 2: Contrasting classical analytics requirements and emerging time-state analytics

Timeline support deeper into the infrastructure can unleash further benefits. Second, the structure exposed by Timelines can enable cross-query optimizations that would be difficult, if not impossible, to realize using prior solutions. Third, the benefits of the Timeline abstraction can be generalized to many domains (e.g., finance, e-commerce, security). Finally, we can use Timelines to build user interfaces that lower the barrier of entry for data analysis.

2 MOTIVATION

In this section, we start with examples of Time-State Analytics that arise in diverse domains. Then, we delve deeper into a concrete scenario to highlight pain points with existing solutions.

Time-State Analytics Setting: We consider a general data processing setting, where we have a stream of raw *updates* associated with a *timestamp*, some *metadata* of interest (e.g., the user's geolocation, device type), and some *measurements*. The analyst wants to *query* the data over some *time window*, and requests some summary of a *stateful metric*, computed over updates and measurements.

To make this concrete, we consider a few examples:

- *Video distribution (e.g., [14]):* What is the total rebuffering time of a session when using CDN1, ignoring buffering occurring within 5 seconds after a user seek?[1] What is the session play time when using CDN1 and cellular networks?
- *IoT monitoring (e.g., [22]):* How long did a device spend in a "risky" state; i.e., continuously high battery temperature ($\geq 60°$C) and high memory usage ($\geq$ 4GB) for $\geq$ 5 minutes?
- *Finance (e.g., [11]):* Did a credit card user make purchases at locations $\geq$ 20 miles apart within 10 mins of each other?
- *Cybersecurity (e.g., [29]):* Did an Android user send a lot of ($\geq$ 100) DNS requests after visiting xyz.com in the last hour?
- *Mobile app monitoring (e.g., [9]):* Did an iPhone user quit advancing in a game when the ad took $\geq$ 5 seconds to load?

Figure 2 contrasts these types of queries, which we refer to as *time-state analytics*, from classical analytics over data streams. Traditional queries entail simple filtering and aggregation operations over *direct measurements*; e.g., average salary or average temperature over a sequence of events filtered by some subgroup of interest. In contrast, the above queries entail more complex analysis to model the state dynamics of the physical world to compute complex metrics as they evolve over continuous time, using the raw state updates. For most classical queries, the sequence and continuous time semantics are not critical; i.e., we can view the computations as operations over a *set* of updates. In contrast, time-state analytics needs to carefully model the *continuous sequential* evolution of the world states to compute context-sensitive metrics of interest.

To see this more clearly, let us delve deeper into Figure 1. We have a sequence of updates for a given video session: we see connection issues causing buffering, users initiating player seeks, and the player switching CDNs, among other updates. From these raw updates, we need to compute business-relevant metrics [14] such as the connection-induced rebuffering (CIR). These metrics are not directly expressed in the updates but need to be carefully computed by modeling the physical world state at each point in time. This requires *stateful computations* tracking and combining multiple states over points in time; e.g., track the state of the player (i.e., playing vs. buffering) and user actions over time, eliminate buffering induced by user events (e.g., player seek, init), and measure the duration when these conditions occur.

There is one additional aspect of time-state analytics workloads worth noting. Given the nature of operations in which such queries are used, they typically arise in an *ad hoc* manner during some analysis; e.g., CDN1 has low performance or some observation of anomalous user behaviors. These are not pre-written queries because the nature of anomalies or incidents cannot be predicted. Since queries are written on demand, the time to develop and debug the queries is also a critical consideration.

Existing frameworks and limitations: Taken together, the above characteristics of time-state analytics have fundamental implications for (a) development time and complexity and (b) cost vs. performance tradeoffs. To understand why, consider the simpler case (for ease of explanation) where the analyst is tasked with writing queries in a *batch mode* in an existing SQL- or Spark-like system. In reality, production systems for time-state analytics may need to run in a streaming model over updates [2]. Streaming will entail added complexity (e.g., semantics of event vs. processing time, windowing, view materialization) [2]. For ease of explanation, we focus on batch-like queries in this paper, noting that there are known approaches to bridge batch and streaming modes.

A canonical workflow here would be to store raw updates in a tabular format with the columns: $\langle$T, P, A, C$\rangle$ denoting respectively the event time, the player state, the user action, and the new CDN event corresponding to Figure 1. Figure 3 shows the typical mental model we have observed data analysts using for this query. (Note that the tabular mental model applies for both SQL and Spark.) Here, one starts with a table of discrete events. Discretely-evolving state is handled by window functions over this table, ordered by timestamp. The analyst starts with the raw event table and then selects the events of interest that occurred before the desired time. Then, the analyst proceeds to add some temporary data structures for tracking the state, and then has to express a duration computation, and finally tracks this duration metric per CDN.

Figure 4 shows a snippet of how this mental model translates into SQL and Spark (using the standard DataFrame API in Python). Just visually, this is quite messy; dealing with the state evolving continuously over time is hard. These complex code snippets are not merely hypothetical or contrived examples; these are adapted from actual code that data analysts used to write in production. Even modern extensions such as timeseries databases cannot support

[1]Such buffering needs to be ignored when diagnosing connection problems.

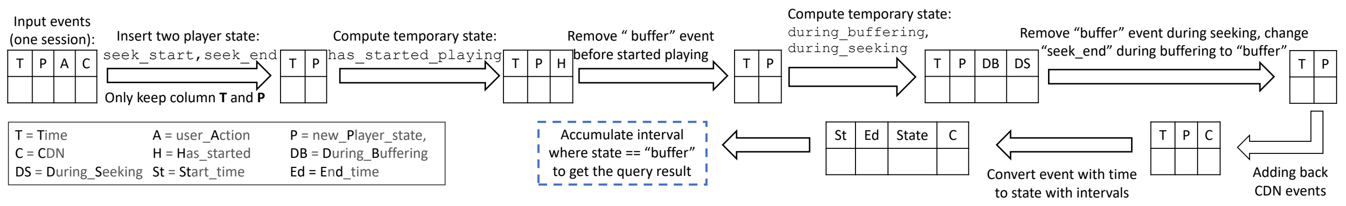

Input events (one session):
T P A C
Insert two player state: seek_start, seek_end
Only keep column T and P
T P
Compute temporary state: has_started_playing
T P H
Remove " buffer" event before started playing
T P
Compute temporary state: during_buffering, during_seeking
T P DB DS
Remove "buffer" event during seeking, change "seek_end" during buffering to "buffer"
T P
Adding back CDN events
T P C
Convert event with time to state with intervals
St Ed State C
Accumulate interval where state == "buffer" to get the query result
T = Time
C = CDN
DS = During_Seeking
A = user_Action
H = Has_started
St = Start_time
P = new_Player_state,
DB = During_Buffering
Ed = End_time

Figure 3: A typical contrived mental model that analysts need to use with traditional tabular data models (e.g., Spark, SQL)

```sql
1   WITH SeekAsPlayerState(T, P) as (
2     SELECT T, P FROM heartbeats WHERE P IS NOT NULL
3     UNION SELECT T, "Seek_st" FROM heartbeats WHERE A IS NOT NULL
4     UNION SELECT T + 5, "Seek_ed" FROM heartbeats WHERE A IS NOT NULL ),
5   IgnoreBufBeforePlay(T, P) as (
6     SELECT T, P FROM (
7       SELECT T, P, Max(If(P == 'play', 1, 0)) OVER (PARTITION BY 1 ORDER BY T)
            ↪ as H
8       FROM SeekAsPlayerState) WHERE H == True ),
9   DuringBufferTable(T, P, DB) as (
10    SELECT T, P, LAST(tmp1) IGNORE NULLS OVER (PARTITION BY 1 ORDER BY T)
11    FROM (
12      SELECT T, P,
13            CASE P WHEN 'buffer' THEN True WHEN 'Seek_st' THEN NULL WHEN 'Seek_ed'
                ↪ THEN NULL ELSE FALSE END as tmp1
14      From IgnoreBufBeforePlay ) ),
15  DuringSeekTable(T, P, DB, DS) as (
16    SELECT T, P, DB,
17           (T - Max(If(P == 'Seek_st', T, 0)) OVER (PARTITION BY 1 ORDER BY T)
               ↪ ) < 5 as tmp2
18    FROM DuringBufferTable ),
19  IgnoreBufInSeek(T, P) as (
20    SELECT T, P FROM (
21      SELECT T, DS, IF(P == 'Seek_ed' and DB, 'buffer', P) as P
22     FROM DuringSeekTable ) WHERE NOT (P == 'buffer' AND DS) ),
23  WithCDNAndQuery(T, P, C) as (
24   SELECT T, P, NULL FROM IgnoreBufInSeek
25   UNION SELECT T, NULL, C FROM heartbeats where C IS NOT NULL
26   UNION SELECT 2022-07-21 10:05, NULL, NULL s),
27  Intervals(Ed, St, State, CDN) as (
28    SELECT T, LEAD(T, 1) OVER (PARTITION BY 1 ORDER BY T), P, C
29    FROM (
30      SELECT T,
31            LAST(P) IGNORE NULLS OVER (PARTITION BY 1 ORDER BY T) as P,
32            LAST(C) IGNORE NULLS OVER (PARTITION BY 1 ORDER BY T) as C
33      FROM WithCDNAndQuery ) )
34  SELECT SUM(St - Ed) as result FROM Intervals
35  WHERE Ed < 2022-07-21 10:05 AND State == 'buffer' AND CDN == 'CDN1'
```

(a) SQL code

```python
1   w = Window.partitionBy().orderBy("T")
2   seekAsPlayerState = heartbeats.filter("P is not null")
3       .union(heartbeats.filter("A is not null")
4           .withColumn("P", F.lit("Seek_st")))
5       .union(heartbeats.filter("A is not null")
6           .withColumn("T", col("T") + 5)
7           .withColumn("P", F.lit("Seek_ed")))
8       .select("T", "P")
9   ignoreBufBeforePlay = seekAsPlayerState.withColumn("H",
10          F.max(when(col("P") == 'play', 1).otherwise(0)).over(w))
11      .filter("H == True")
12  duringBuffer = ignoreBufBeforePlay.withColumn("DB",
13          when(col("P") == "buffer", True)
14          .when((F.col("newplayerState").contains("seek")), None)
15          .otherwise(False))
16      .withColumn("DB", last("DB", ignorenulls=True).over(w))
17  duringSeek = duringBuffer.withColumn("DS",
18      (col("T") - F.max(when(col("P")== "Seek_st", col("T")).otherwise(0)).over
            ↪ (w)) < 5)
19  ignoreBufInSeek = duringSeek.withColumn("P",
20          when((col("P") == "Seek_ed") & (col("DB") == True), "buffer")
21          .otherwise(col("P")))
22      .filter((col("P") != "buffer") | (~col("DS")))
23      .select("T", "P")
24  CDN = heartbeats.select("T", "C").filter("C is not null")
25  queryPoints = spark.createDataFrame(["2022-07-22 10:05"], DateType()).toDF("
        ↪ T")
26  withCDNAndQuery = ignoreBufInSeek.unionByName(CDN, allowMissingColumns=
        ↪ True)
27      .unionByName(queryPoints, allowMissingColumns=True)
28  intervals = withCDNAndQuery.withColumn("P", last("P", ignorenulls=True).
        ↪ over(w))
29      .withColumn("C", last("C", ignorenulls=True).over(w))
30      .withColumn("next_change_T", lead("T", 1).over(w))
31      .withColumn("duration", col("next_change_T") - col("T"))
32  result = intervals.filter(col("C") == "CDN1")
33      .filter("T < 2022-07-21 10:05")
34      .filter(col("P") == "buffer")
35      .agg(sum("duration").alias("cirDuration"))
```

(b) PySpark code

Figure 4: Traditional SQL and Spark code for computing per-CDN buffering time from video data, adapted by actual examples data analysts wrote in production.

such stateful operations either; they primarily focus on efficient storage and simple numerical aggregations over measurements (e.g., [6, 26]). One might think the low-level stateful processing APIs provided by data processing systems such as Flink [8] and Dataflow [1] could help. Unfortunately, in our experience, these tend to make things worse for expressing such more complex and stateful query intents.[2]

Why existing frameworks fall short: The issue here is not that existing systems (e.g., SQL, Spark) are unable to express the logic for time-state analytics. Rather, they offer a *low level of abstraction* based on a tabular mental model which makes it hard, if not impossible, to write succinct, well-reasoned, and performant code for time-state analytics. We illustrate a few examples below:

- *Tracking state evolution:* Time-State Analytics require tracking and modeling stateful dynamics of the physical world. For instance, we need to check if a state "play" happened in the past. However, there is no natural support in existing frameworks, and this is expressed with cryptic clauses with `max` and `when` in line 10 in Figure 4b. Similarly, expressing state transitions is tedious; e.g., from line 12 to line 16 in Figure 4 we have a complex case clause to determine if the session is in a "buffering" state.
- *Computing time spent in states:* Another key requirement is to compute the *duration spent in some state*. With no natural operation, the Spark implementation uses overly complicated code like lines 28-32 and 34-35 in Figure 4b.
- *Evaluating a metric at a given point in time:* The tabular representation naturally tracks only those points in time that happen

[2]For brevity, we omit code samples for streaming queries.

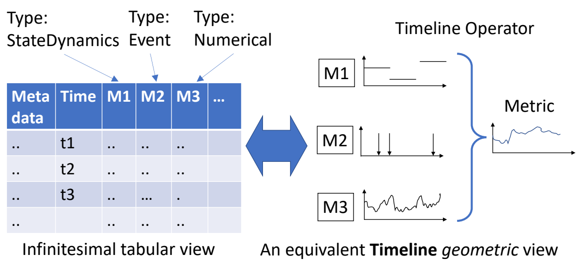


Figure 5: Equivalence between a (hypothetical) table with infinitesimal timestamps and the intuitive Timeline view.

to have updates, which usually do not coincide with the desired query points. To find the duration in the connection-induced rebuffering state up to a given time, the Spark code in lines 25, 27, and 33 in Figure 4b add an unnatural querying row at that point in time.

This complexity also has performance implications. For instance, executing a query requires multiple scans of the tabular data, adding intermediate columns and rows. This can get aggravated if we need finer time-resolution analysis as the number of rows increases linearly with the resolution using a tabular representation. Furthermore, supporting *ad hoc* queries is challenging as debugging such logic for correctness and performance before deployment is difficult. Finally, these problems are worsened (not shown) when we move to streaming realizations; i.e., the analysis will often want to calculate the modeled state at specific points in time, e.g. once per minute or once per hour. Since the table only captures the event time, not query time, it requires more hacks (e.g., adding dummy events to track query time, or discretizing time).

3 THE TIMELINE FRAMEWORK

The above discussion highlights that existing data processing frameworks lack the right abstractions for supporting the requirements of time-state analytics. Thus, we argue the need for identifying a better data model and data processing abstraction.

In this context, we draw inspiration from an observation in a classic paper by Fred Brooks: *"Geometric abstractions are powerful tools [7]."* Brooks argues that physical world systems work because designers can visualize layouts and identify problems. In contrast, the software is hard to visualize, and this hinders design [7]. While Brooks' pessimism may be true for software systems in general, there are specific domains where such abstractions do exist and if have the potential to dramatically reduce system complexity.

Our contribution in this work lies in identifying a *geometric abstraction* specific to Time-State Analytics! Indeed, we already saw this in Figure 1 — human mental models for state machines and metrics over time have an *intuitive temporal* representation, which allows us to more naturally reason about state dynamics over continuous time. Figure 5 offers an alternative view, showing the equivalence between a hypothetical tabular view with infinitesimally small timestamps and an equivalent Timeline view. Each measurement column (M1-M3) exhibits one of three types of possible temporal behaviors: events appearing at discrete points in time, states changing over time in a step function like manner, or continuously-varying numerical values.

Given the Timeline mental model, writing queries to compute context-sensitive stateful metrics becomes intuitive geometric manipulations over Timeline types. Figure 6 shows a Timeline-based mental model for expressing the same query intent from §2. Logically, Timeline-based processing retains the DAG or pipeline abstraction from prior data processing systems (e.g., [1]) as seen in Figure 6. The key difference is a set of high-level Operators and TimelineTypes, where the data being transferred and modified between DAG nodes are Timelines rather than low-level data frames or tables. For instance, instead of the complexity of translating events into states and modeling the "duration-spent-in-state" semantics, the Timeline mental model is cleaner, compact, and naturally lends itself to more succinct code (see Figure 7b).

3.1 TimelineTypes and Operations

TimelineTypes: There are three natural ways for dynamic processes to vary over time, which we call the basic TimelineTypes:[3]

- `StateDynamics`: This captures a state having a value at each point in continuous time but changing at discrete points; e.g., player state and CDN state in our example.
- `Numerical`: This captures values varying continuously over time that often arise as intermediate representations; e.g., the time spent in the connection-induced rebuffering state.
- `Event`: This captures a sequence of discrete events, e.g., user seek events, player state *updates*, and CDN *updates*. Each point in time has a collection of one or more event values.

Timeline Operators: Timeline operators enable analysts to *declaratively* specify computation over the Timeline objects. Table 1 describes a subset of these operators and their semantics. Logically, we can divide these operators into two classes: (1) generalizations of classical operators that operate pointwise on Timeline data but have natural analogs in a tabular/relational model; and (2) *Timeline-unique operators* that are challenging to express and/or implement in a tabular model. Each Timeline Operator also has a natural *geometric visualization* that is easy for a data analyst to understand and explain. The rightmost column shows examples of the geometric interpretation for each of these Operators. Using these operators, we can more naturally express the processing logic required; e.g., in our example query using the TL_DurationWhere and TL_HasExistedWithin operations can simplify the query complexity of logical operations that were exceedingly complex to express in SQL and Spark.

3.2 Implementation and Integration

To use the Timeline data model in existing data-processing systems, we need to: (1) translate the events stored in the system's original format to one or more Timelines; (2) apply user-specified Timeline Operators (such as those in Table 1)[4]; and (3) translate the resulting Timelines back into the original data format. In some

[3]This set of three basic time-varying patterns is similar to prior work on functional reactive programming [10].
[4]The TL prefix is just for visual ease of differentiating classical APIs from ours.

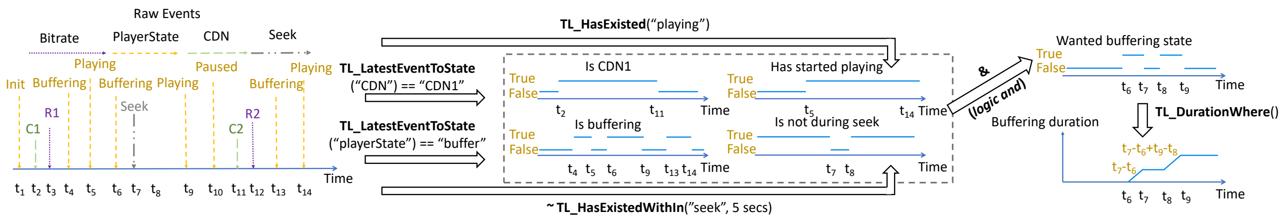


Figure 6: An intuitive geometric mental model using Timelines for the same query

```
SELECT TL_DurationWhere(
   TL_LatestEventToState(playerStateChange) = 'buffer' AND
   TL_HasExisted(playerStateChange = 'play') AND
   NOT TL_HasExistedWithin(userAction = 'seek', 5s) AND
   TL_LatestEventToState(cdnChange) = 'CDN1'
) AS result
FROM heartbeats
TIMELINE WITH EVENT TIME t
EVALUATE AT EVENT TIME 2022-07-21 10:05:00
```

(a) Refactored SQL

```
result = heartbeats.toTimeline(eventTime = col("t"))
   .select(TL_DurationWhere(
         (TL_LatestEventToState(col("playerStateChange")) == "buffer") &
         TL_HasExisted(col("playerStateChange") == "play") &
         ( ~ TL_HasExistedWithin(col("userAction") == "seek", 5s)) &
         (TL_LatestEventToState(col("cdnChange")) == "CDN1")
      ).as("cirDuration"))
   .TL_EvaluateAt("2022-07-21 10:05:00")
```

(b) Refactored PySpark

Figure 7: Reformulating queries using Timeline Operators

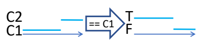
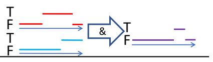
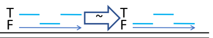
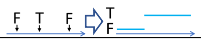
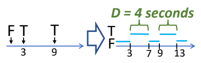
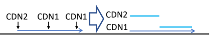
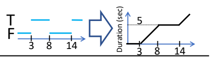
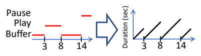
Timeline generalizations of classical Operators		
==, <, > [constant]	Compare each update or state with a fixed value, producing True or False	
&, | [timeline]	Combine 2 timelines by applying a logical operation at each point in time	
~	Logically invert each update or state	
Timeline-specific Operators		
TL_HasExisted	A StateDynamics timeline of the *cumulative OR*	
TL_HasExistedWithin	As TL_HasExisted, but resets to False after a specified duration *D* without True values	
TL_LatestEventToState	A StateDynamics Timeline of the latest update	
TL_DurationWhere	A Numerical Timeline of the cumulative duration where the state was True	
TL_DurationInCurState	A Numerical Timeline of the duration since the last state change	

Table 1: Some examples of basic Timeline Operators.

sense, Timeline can be viewed as a *domain-specific extension* to support the complex requirements of handling Time-State Analytics atop existing data processing frameworks, rather than a complete replacement for such frameworks. For instance, in the context of the *what-where-when-how* modeling of streaming systems [2], our contribution is a high-level abstraction for the "What" component to describe the nature of operations on the input stream.

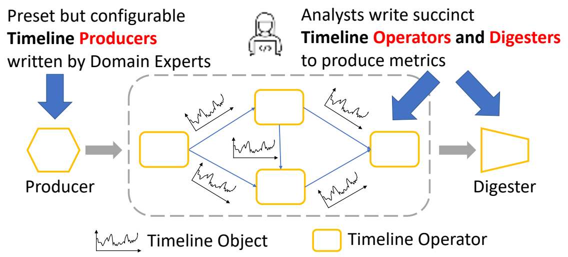


Figure 8: DAG of Producer, Operator, and Digester.

Figure 8 shows an abstract Timeline-based processing DAG. We refer to steps (1) and (3) as Timeline Producers and Digesters respectively. Abstractly, a Timeline Producer's job is to understand the encoding of the input and identify the timestamp or time range associated with each piece of incoming data. A Timeline Digester calculates the final outputs (e.g., stateful metrics like the buffering duration) by evaluating the final Timeline data at specified timestamps or time ranges and encodes the result in a format for downstream consumers (e.g., tables, summary statistics). Data analysts use the Timeline library of pre-defined operators. In the future, we expect to support user-defined functions as well expressed using more basic operations.

In our production system, we have implemented the Timeline data model and operations directly as an in-memory Scala library that operates only on Timelines. To expand the potential applicability, we sketch potential integrations with other analytics toolkits:

Timeline + Spark: A DataFrame can be converted to a Event Timeline, operated upon by Timeline Operators, and then converted back to a DataFrame. Since a Timeline can be viewed as a table with exactly one row per point in time (Figure 5), row-wise operations on DataFrames extend naturally to Timelines, and the API methods for Timelines look familiar to users of the DataFrame API. Figure 7b shows how we can significantly simplify the Spark-like code with new Timeline APIs for stateful and duration operators that were the bane of the native Spark code.

Timeline + SQL: Since Spark's DataFrame is logically a table, our Timeline extension for SQL (syntactic details not presented due to space considerations) is semantically similar to that for Spark. Special keywords convert a table to a Timeline, SQL-like syntax applies Timeline Operators, and then a special digest keyword converts these points back to a table.[5]

[5]We do acknowledge that embedding Timeline operations in SQL violates a purist relational view. That said, modern SQL has itself pragmatically moved beyond a pure relational view and thus this departure is worth the practical benefits.

3.3 Potential Benefits

Development time and complexity: In many cases, analysts have some high-level intents, but the process of expressing them in low-level data analysis toolkits often leads to semantic or logical bugs. With Timeline, analysts have a much cleaner geometric/visual mental model for expressing intents. Thus, the time for them to reason about the logic of query intents can be significantly reduced, and thus reduces the potential for semantic bugs. Visually contrasting the code in Figure 7b with Figure 4 makes it plain that the code is much simpler and cleaner to reason about, fitting the geometric mental model of the query intent.

Performance: Using Timelines lends itself to *structure- and semantic-aware* optimizations in two related ways: in the *memory footprint* to store individual Timelines and the *compute footprint* required for Timeline operations. In a tabular representation, representing a time-varying phenomenon entails a row per timestamp or window [6]. Depending on the time resolution the size will vary. In contrast, since we know the temporal and type structures of the TimelineTypes we can use more natural encodings. `Event` objects naturally entail *sparse* encodings; i.e., instead of tracking whether events occurred at each timestamp/window, we only need to store *when it occurs*. For `StateDynamics` and `Numerical` objects, we define a *span* as an event time interval associated with either the value over that interval (for `StateDynamics`) or an encoding of its evolving numerical values (for `Numerical`). Then, `StateDynamics` and `Numerical` objects can be represented as a compact list of *span* elements rather than enumerating each timestamp/window.

Given such compact structures, we can implement semantic-aware operations efficiently over these encodings. We illustrate two (non-exhaustive) examples below:

- *Overlap-aware short-circuiting:* Consider the expression x & y. Evaluating this naively involves merge-sorting the spans of x and y, then evaluating `And` on each interval between changes in x or y. Once we have evaluated a `False` span covering another span, a cleverer implementation can skip the evaluation and merge-sort of the latter span.
- *State-aware short-circuiting:* Consider TL_HasExisted (x), which checks if a `True` value has occurred in Timeline x. This stateful operation has a "sink" state: once we see a `True` in x, the result is a constant. Instead of naively evaluating the operation over all the elements in x, we can avoid computation after reaching the sink, producing a single span covering *all time* after that point.

4 EARLY PROMISE

We have implemented and run pilot Timeline systems in production: a batch system using Apache Spark and a streaming system based on Akka. The system receives raw event data from video players and uses Timelines to compute per-session metrics. Downstream, these metrics are stored in Apache Druid for efficient and flexible aggregation.[6] Each day, the system processes ≈ 50 billion events from 1 billion video sessions across 180 million devices.

We discuss early evidence of the benefits:

Dataset	Description	Size
Video	Synthetic events of a video session, including player state change, user action, CDN, and network type.	2M state-change events. 30M state measurements at a time granularity of 100ms.
IoT	A public device ops dataset [33], including battery, CPU, and memory states of IoT devices.	10M state measurements at a time granularity of 30 seconds.

Table 2: Description of datasets used for benchmarking

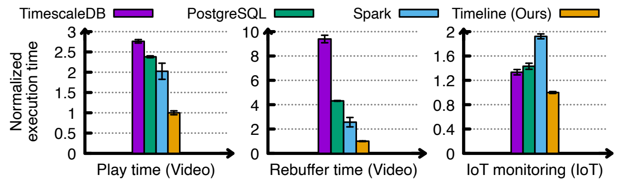


Figure 9: Comparing the normalized execution time on three queries of Timeline against legacy processing systems.

- *Development time and complexity:* Before using Timelines, we computed the same video session metrics in Apache Spark using procedural, event-driven code. Training developers to write new queries would take 6 months, and it would take weeks for a trained developer to write new intents *reliably*. Using Timelines, the time to train new hires dropped from months to weeks, and the time to support new queries dropped from weeks to days. Semantic bugs and inconsistencies across time granularities were common in the previous system, but dropped by >80%.
- *Cloud cost:* Our system needs to produce metrics at multiple time granularities to support diverse use cases that trade off points latency and efficiency. Supporting this with canonical systems required running separate instances of the same Spark code for each time granularity. Once created, Timelines make it simple to run queries at multiple granularities. Thus, it eliminates duplicated computation, reducing the cost by about 75%.

Performance benchmarks: For proprietary reasons, we are unable to show the performance of Timeline in a production workload. As a preliminary benchmark, we run the *Video distribution* and *IoT monitoring* queries defined in §2. The queries are written from scratch and using the datasets in Table 2. We execute the query on a linux machine with 16GB RAM and limit all the baselines to use one CPU core. Figure 9 compares the normalized execution time of Timeline against Timescale [32], PostgreSQL, and Spark [5].

For the play time query, Timeline is 2.8×, 2.4× and 2.0× faster than Timescale, PostgreSQL and Spark respectively. Compared to Timescale, all of PostgreSQL, Spark, and Timeline can operate on event-based inputs instead of a huge amount of state measurements per 100ms. Additionally, the efficient encodings and Timeline-specific optimizations make our prototype better than PostgreSQL and Spark. The performance gain becomes larger on the more complicated rebuffering time query: Timeline is 9.4×, 4.3×, and 2.6× faster than all the baselines respectively. For IoT monitoring query, since all of the methods are having the same input, Timescale becomes better than PostgreSQL and Spark. However, Timeline is still 33% faster than Timescale, because it has fewer operations by skipping the inputs if the current device is not in the "risky" state.

[6]The design of this aggregation layer to support spatio-temporal and subpopulation queries is outside the scope of this paper.

5 RELATED WORK

Our work is related to a number of prior efforts in the area of database systems, Big-Data processing, functional reactive programming, complex event processing, signal databases, and temporal modeling in formal methods. We discuss these how Timelines can build upon advances in these areas and how it complements these existing efforts.

Batch processing systems: Many modern Big-Data batch processing systems draw their lineage from Map-Reduce [12]. However, these systems largely inherit from the tabular mental model and do not offer good abstractions for time-state analytics. We envision Timeline based domain-specific extensions that sit atop these existing frameworks.

Stream processing systems: To tackle low latency and streaming, several proposals exist (e.g., [8, 16, 17, 35]). We refer readers to the book by Akidau et al., for an excellent overview [2]. Others add SQL-like support for streaming queries (e.g., [4]). While these offer some primitive stateful processing, they are relatively low-level abstractions that do not support the continuously evolving state models necessary for handling time-state analytics.

Temporal extensions to SQL: There have been prior attempts to add time semantics into SQL [30]. These proposals (e.g., TSQL2) still present a *tabular* model. In contrast, a Timeline represents each continuous time, simplifying stateful computations. Unlike TSQL2's operations, the Timeline operations are explicitly not intended to be relational, but use Producers and Digesters as a translation layer. We believe this tradeoff is worth making for the benefits of this abstraction. In any case, given the growing importance of time-state analytics across many domains of data processing, we hope our work rekindles this discussion.

Time series databases: Timeseries Databases (e.g., [26, 32]) are mostly concerned with statistical aggregates and trend analysis over measurements rather than stateful computations. That said, many of their storage and encoding optimizations (e.g., compression) may be applicable to the design of Timelines-based backends.

Complex event processing: Our work falls within the broad class of prior work on Complex Event Processing (e.g., [13, 15, 19]). These efforts extended classical SQL to support flexible filters and patterns over event streams and applications to specific domains (e.g., RFID data). Our proposed SQL extensions can build on language design lessons from this literature. However, we believe that recasting event stream processing with a geometric Timeline abstraction offers a more intuitive mental model for data analysis that can enable us to express more complex context-sensitive tasks.

Signal-oriented databases: Our work is also related to prior work on the integration of sensor signal processing and database systems (e.g., [18, 25]). The primary focus of these efforts was in creating unified frameworks for merging signal processing oriented functions (e.g., Fourier Analysis) and classical relational queries. We share the observation and motivation that in many domains of data processing purely relational models are not ideal. However, the kinds of context-sensitive stateful analytics we envision in our target workloads is different from signal processing functions specific to sensor processing.

Functional reactive programming: In terms of high-level abstractions, our work is also related to work in the programming language community on Functional Reactive Programming (FRP) (e.g., [10, 23, 27, 28]). These systems were largely driven by applications in the computer graphics, user interface design, and robotics domains. The concept of Timeline is conceptually similar to the notion of a "Signal" mapping time-to-values and our Operators are analogous to Signal Functions in the FRP literature. There are some recent efforts to apply these in the context of Big-Data processing as well (e.g., [31]). Our work can both inform and build on abstractions and API design lessons in this literature.

Temporal Logic and Formal Methods: Our work on stateful and temporal processing is also related to efforts in the formal methods literature (e.g., [3, 24]). An interesting direction of future work is using some of these efforts for formal verification and analysis of query intents expressed using Timeline Operators.

Optimizations: Prior efforts have identified optimizations for stream processing (e.g., [20] and functional reactive programming (e.g., [10, 28]). An exciting direction of future work is to explore these opportunities on top of the Timeline framework.

6 FUTURE OUTLOOK

Generality: Our early wins have been in video analytics. We posit that the Timeline abstraction has broader applications. An interesting direction is exploring how the Timeline framework can be extended to support future domains (§2); e.g., do we need more TimelineTypes, Operators, and so on.

Performance optimizations: Our prototype is built atop existing streaming and batch systems. Looking forward, we envision further improvements by adopting a *native* support of Timeline spanning the language and the compute/storage layer.

Streaming semantics: Seen in the streaming context, our primary focus here is on the API for expressing "what" to compute [2]. That said, we speculate that the cleaner abstraction can simplify other correctness and semantic requirements in a distributed and streaming setting; e.g., better support for windowing, watermarking, and correctness in distributed settings. We leave this for future work.

User interfaces: We envision a natural visual or graphical query interface to users as seen in Figure 6. This would enable "no-code" approaches that lower the barrier of entry to express complex analysis and make the benefits of Timeline available to non-programmers.

ACKNOWLEDGMENTS

We thank Akara Sucharitakul, Haijie Wu, Oleg White, Florin Dobrian, and other members of the Conviva infrastructure team for their valuable inputs.

REFERENCES

[1] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. 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. *Proc. VLDB Endow.*, 8:1792–1803, 2015.

[2] T. Akidau, S. Chernyak, and R. Lax. *Streaming Systems: The What, Where, When, and How of Large-Scale Data Processing.* O'Reilly Media, Inc., 2018.

[3] R. Alur and T. A. Henzinger. Logics and models of real time: A survey. In *Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems)*, pages 74–106. Springer, 1991.
[4] A. Arasu, S. Babu, and J. Widom. Cql: A language for continuous queries over streams and relations. In *DBPL*, volume 2921 of *LNCS*, pages 1–19. Springer, 2003.
[5] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational Data Processing in Spark. In *Proc. ACM SIGMOD*, 2015.
[6] A. Bader, O. Kopp, and M. Falkenthal. Survey and comparison of open source time series databases. *Datenbanksysteme für Business, Technologie und Web (BTW 2017)-Workshopband*, 2017.
[7] F. P. Brooks. No Silver Bullet—Essence and Accident in Software Engineering. In *Proc. World Computing Conference*, 1986.
[8] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society TCDE*, 36(4), 2015.
[9] Q. A. Chen, H. Luo, S. Rosen, Z. M. Mao, K. Iyer, J. Hui, K. Sontineni, and K. Lau. Qoe doctor: Diagnosing mobile app qoe with automated ui control and cross-layer analysis. In *Proc. ACM SIGCOMM IMC*, 2014.
[10] G. Chupin and H. Nilsson. Functional reactive programming, restated. In *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming*, pages 1–14, 2019.
[11] A. Dal Pozzolo, G. Boracchi, O. Caelen, C. Alippi, and G. Bontempi. Credit card fraud detection: a realistic modeling and a novel learning strategy. *IEEE TNNLS*, 2018.
[12] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04*, pages 137–150, San Francisco, CA, 2004.
[13] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. M. White. Cayuga: A general purpose event monitoring system. In *Conference on Innovative Data Systems Research*, 2007.
[14] F. Dobrian, V. Sekar, A. Awan, I. Stoica, D. Joseph, A. Ganjam, J. Zhan, and H. Zhang. Understanding the impact of video quality on user engagement. In *Proc. ACM SIGCOMM 2011 Conf.*, 2011.
[15] O. Etzion, M. Chandy, R. v. Ammon, and R. Schulte. Event-driven architectures and complex event processing. In *IEEE SCC'06*, 2006.
[16] R. Evans. Apache storm, a hands on tutorial. In *IC2E*. IEEE CS, 2015.
[17] C. Gencer, M. Topolnik, V. Ďurina, E. Demirci, E. B. Kahveci, A. Gürbüz, O. Lukáš, J. Bartók, G. Gierlach, F. Hartman, U. Yılmaz, M. Doğan, M. Mandouh, M. Fragkoulis, and A. Katsifodimos. Hazelcast jet: Low-latency stream processing at the 99.99th percentile, 2021.
[18] L. Girod, Y. Mei, R. Newton, S. Rost, A. Thiagarajan, H. Balakrishnan, and S. Madden. The Case for a Signal-Oriented Data Stream Management System . In *Proc. CIDR*, 2007.
[19] D. Gyllstrom, E. Wu, H.-J. Chae, Y. Diao, P. Stahlberg, and G. Anderson. Sase: Complex event processing over streams (demo). In *Conference on Innovative Data Systems Research*, 2006.
[20] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. A catalog of stream processing optimizations. *ACM Comput. Surv.*, 46(4), mar 2014.
[21] N. Jain, S. Mishra, A. Srinivasan, J. Gehrke, J. Widom, H. Balakrishnan, U. Çetintemel, M. Cherniack, R. Tibbetts, and S. Zdonik. Towards a streaming sql standard. *Proc. VLDB Endow.*, 1(2), 2008.
[22] D. Jamieson. The Growing Power of IoT in Preventative Maintenance. https://www.particle.io/blog/the-growing-power-of-iot-in-preventative-maintenance/, 2021.
[23] G. Mainland, G. Morrisett, and M. Welsh. Flask: Staged functional programming for sensor networks. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, page 335–346, New York, NY, USA, 2008. Association for Computing Machinery.
[24] O. Maler and D. Nickovic. Monitoring temporal properties of continuous signals. In *FORMATS'04*, pages 152–166. Springer, 2004.
[25] M. Nikolic, B. Chandramouli, and J. Goldstein. Enabling signal processing over data streams. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 95–108, 2017.
[26] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. *Proc. VLDB Endow.*, 8(12):1816–1827, 2015.
[27] I. Perez, M. Bärenz, and H. Nilsson. Functional reactive programming, refactored. In *Proceedings of the 9th International Symposium on Haskell*, Haskell 2016, page 33–44, New York, NY, USA, 2016. Association for Computing Machinery.
[28] N. Sculthorpe and H. Nilsson. Optimisation of dynamic, hybrid signal function networks. In *Symposium on Trends in Functional Programming*, 2009.
[29] S. Shin, Z. Xu, and G. Gu. EFFORT: Efficient and Effective Bot Malware Detection. In *Proc. INFOCOM*, 2012.
[30] R. T. Snodgrass. *The TSQL2 temporal query language*. Springer, 1995.
[31] R. Terrell. Real-Time Stream Analysis in Functional Reactive Programming . https://www.infoq.com/presentations/stream-analysis-fp/.
[32] Timescale. TimescaleDB API Reference. https://docs.timescale.com/api/latest/, 2022.
[33] Timescaledb sample datasets. https://docs.timescale.com/timescaledb/latest/tutorials/sample-datasets/#sample-datasets, 2022.
[34] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *NSDI*, 2012.
[35] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *Proc. SOSP*, 2013.