Complex Event Recognition: Automata–based CER & Complex Event Forecasting

Alexander Artikis, Elias Alevizos

Institute of Informatics & Telecommunications,
NCSR Demokritos

alevizos.elias@iit.demokritos.gr
a.artikis@iit.demokritos.gr

http://cer.iit.demokritos.gr

Introduction

Why automata for Complex Event Recognition?

- Automata handle strings, i.e., sequences.
- Event streams may be viewed as sequences of events.
- Automata have already been used to do pattern matching.
- They seem a natural fit for pattern matching on streams, i.e., for CER
- ... but several extensions required

Motivating example

$$a^+(bb + c)^* a^+$$

- What is this? Regular expression.
- What does it express? Strings that begin and end with at least one a, having in the middle zero or more b's or c's, but b's always come in pairs.

An online automaton

- Assume we want to run a finite automaton $A(R)$ on a stream.
- E.g., $R = 01$ and the stream is $100\mathbf{01}110\mathbf{01}0...$
- What is the regular expression to be used?
  - If R is used, what would happen? Stuck/killed after the first 1.
  - We need to be able to skip events. How?
  - Use $\Sigma^* \cdot R = (0 + 1)^* \cdot (0 \cdot 1)$.

- Why can't we use Regular Expression matching?
- Presence of complex predicates.
  - $c.velocity > 30$
- Selection strategies. What is the selection strategy for classical automata? strict–contiguity.
- Efficiency and memory management with
  - Kleene closure. Need to store "unbounded" number of events to perform matching.
  - Output of complete matches. Need to store all events constituting a match.

Other automata (PDA)

- Push-down automata (PDA).
- They have a LIFO stack (memory).
- Equivalent to context-free grammars.
- But break equivalence between deterministic and non-deterministic PDA.
- Useful for? XML processing.
- Useful for CER? Memory is important for CER.

Other automata (Symbolic)

- Instead of symbols on transitions, use predicates.
- Each "symbol" can now be a tuple.
- A transition is activated if its predicate evaluates to true.
- How is this useful for CER?
- Events in an event stream are actually tuples.
- Easier to incorporate background knowledge.
- Symbolic (non-deterministic) automata can be more compact.
- See [DV17, AAP18a].

Automata for CER

- The literature on automata is huge.
- We will study automata suited for CER.
- These are specifically built for CER
  - Very efficient
  - ... but do not always have nice properties
- For details, see [GAA+20, CM12].

Complex Event Recognition with SASE

SASE

- A computational model for pattern matching on streams.
- Each event represents an occurrence of interest (timestamp plus other attributes).
- All input events merged into a single stream, ordered by the occurrence time.
- A pattern seeks a series of events that occur in the required temporal order and satisfy all additional constraints.
- Uses NFAs with buffers (a kind of "stack"). Why buffers?
  - We need memory for evaluating constraints and retrieval of matches.
- See [ADGI08, ZDI14].

SASE language

- *Sequencing* (SEQ) lists the required event types in temporal order, e.g., $SEQ(A, B, C)$.
- *Kleene closure* ($+$) collects a finite yet unbounded number of events of a particular type, e.g., $SEQ(A, B+, C)$.
- *Negation* ($\sim$ or !) verifies the absence of certain events in a sequence, e.g., $SEQ(A, \sim B, C)$.
- *Value predicates* further specify numerical constraints on the events addressed in SEQ (aggregate functions in Kleene closure).
- *Closure under union, negation and Kleene closure*, e.g., $SEQ(A, B) \cup SEQ(B, C)$.
- *Windowing* (WITHIN) restricts a pattern to a specific time period
- *Return* (return) constructs new events for output

SASE selection strategies

- *strict–contiguity*: selected events must be contiguous in the input.
- *partition–contiguity*: stream partitioned by a logical condition.
- *skip–till–next match*: ignores irrelevant events.
- *skip–till–any match*: non–deterministic actions.

SASE selection strategies: example

- $S = \{A, B, A, A, C, B\}$, $R := SEQ(A, B)$
- strict–contiguity:

SASE selection strategies: example

- $S = \{A, B, A, A, C, B\}$, $R := SEQ(A, B)$
- strict–contiguity: $M_1 = \{1, 2\}$
- skip–till–next:

SASE selection strategies: example

- $S = \{A, B, A, A, C, B\}$, $R := SEQ(A, B)$
- strict–contiguity: $M_1 = \{1, 2\}$
- skip–till–next: $M_1 = \{1, 2\}$, $M_2 = \{3, 6\}$, $M_2 = \{4, 6\}$
- skip–till–any:

SASE selection strategies: example

- $S = \{A, B, A, A, C, B\}$, $R := SEQ(A, B)$
- strict–contiguity: $M_1 = \{1, 2\}$
- skip–till–next: $M_1 = \{1, 2\}$, $M_2 = \{3, 6\}$, $M_2 = \{4, 6\}$
- skip–till–any: $M_1 = \{1, 2\}$, $M_3 = \{3, 6\}$, $M_2 = \{4, 6\}$ but also $M_4 = \{1, 6\}$

SASE evaluation model

- NFA^b, NFA with buffer and predicates.
- $A = (Q, E, \theta, q_1, F)$
  - Q set of states
  - E set of edges
  - θ set of formulas labelling the edges
  - q_1 start state
  - F final state (single)

Constructing NFA^b: example

PATTERN $SEQ(gapStart\ a,\ gapEnd\ b,\ speedChange\ c)$
WHERE partition-contiguity
AND [*vesselId*]
AND $c.velocity > 30$
WITHIN 3600

Constructing NFA^b example: states

- Use only the *PATTERN* clause.
- States: linear sequence of states.
  - Singleton states for non Kleene plus.
  - A pair of states for each Kleene plus.
  - A rightmost final state.

Constructing NFA^b example: edges

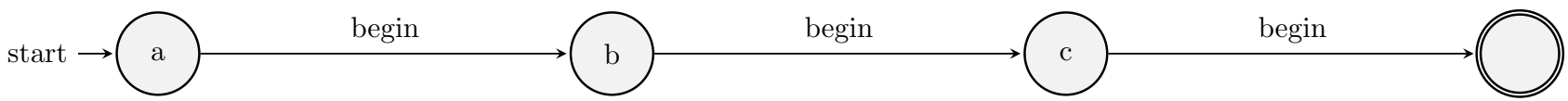
- Edges represent actions taken at the state.
- Edges described by a triplet.
  - a formula θ_{q_edge}: condition for taking the edge.
  - an operation on the stream (consume or not event).
  - operation on the buffer (write or not).

Constructing NFA^b example: edges

- Singleton states (or first states of Kleene plus): forward *begin* edge.
- Second states of Kleene plus: forward *proceed* edge + looping *take* edge.
- All states (except start and final) have a looping *ignore* edge (why?).
- *begin* and *take* edges consume events and write them to the buffer. *ignore* edges skip events in relaxed selection strategies. *proceed* edges are similar to ε transitions.

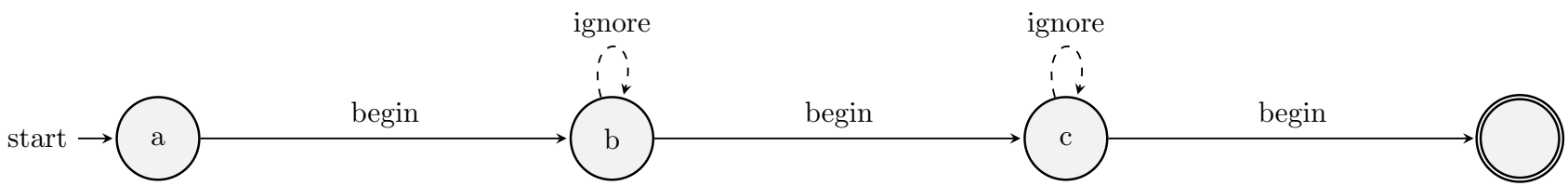
Constructing NFA^b example: *begin* edges

- Only singleton states.
- Add *begin* edges.



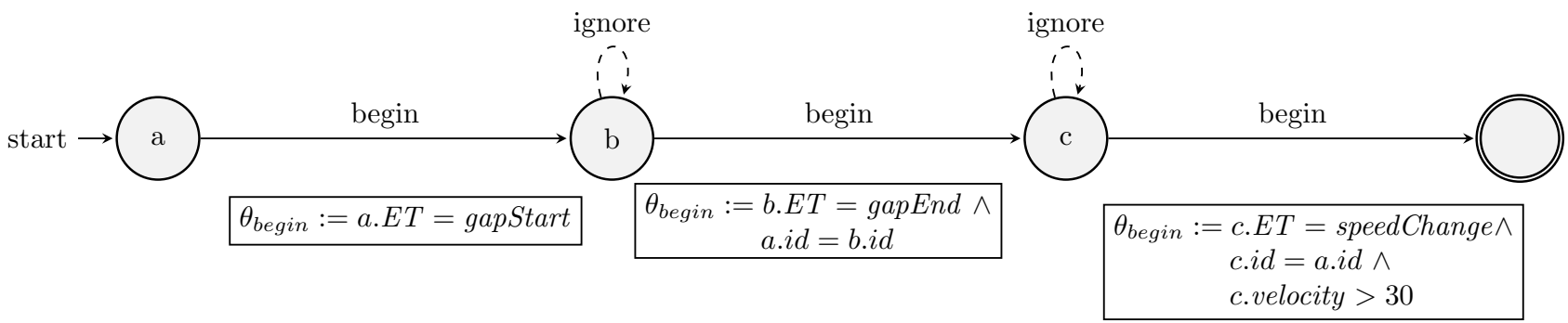
Constructing NFA^b example: *ignore* edges

- Only singleton states.
- Add *ignore* edges, except for start and final (dashed lines indicate that events are not written to buffer).



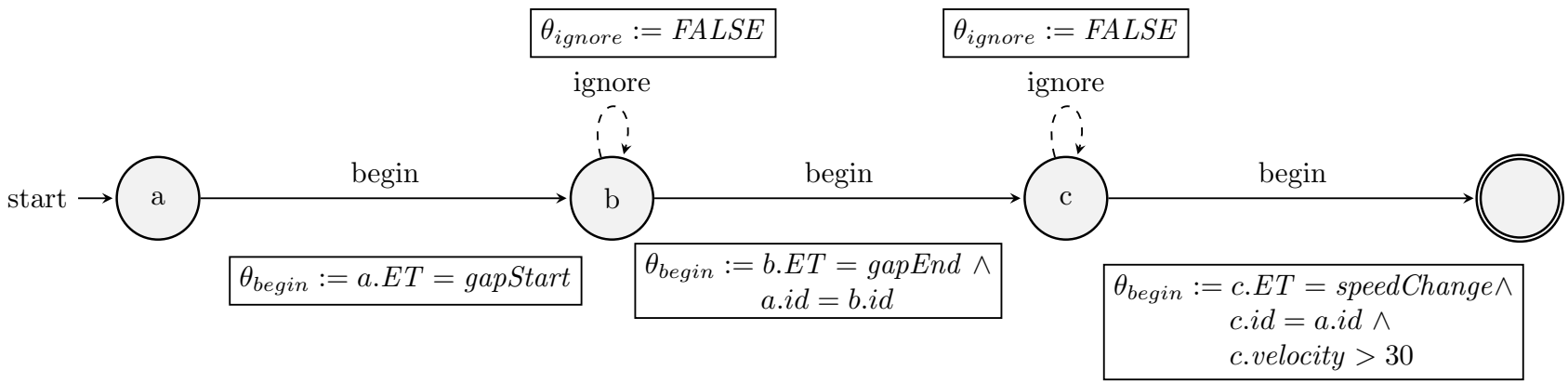
Constructing NFA^b example: placing predicates

- Gather all predicates/conjuncts ($a.ET = gapStart$, $b.ET = gapEnd$, $c.ET = speedChange$, $a.id = b.id$, $c.id = a.id$, $c.velocity > 30$, where ET stands for $EventType$).
- Determine *begin* edges to place predicates. Where? Last state where "identifier is instantiated".



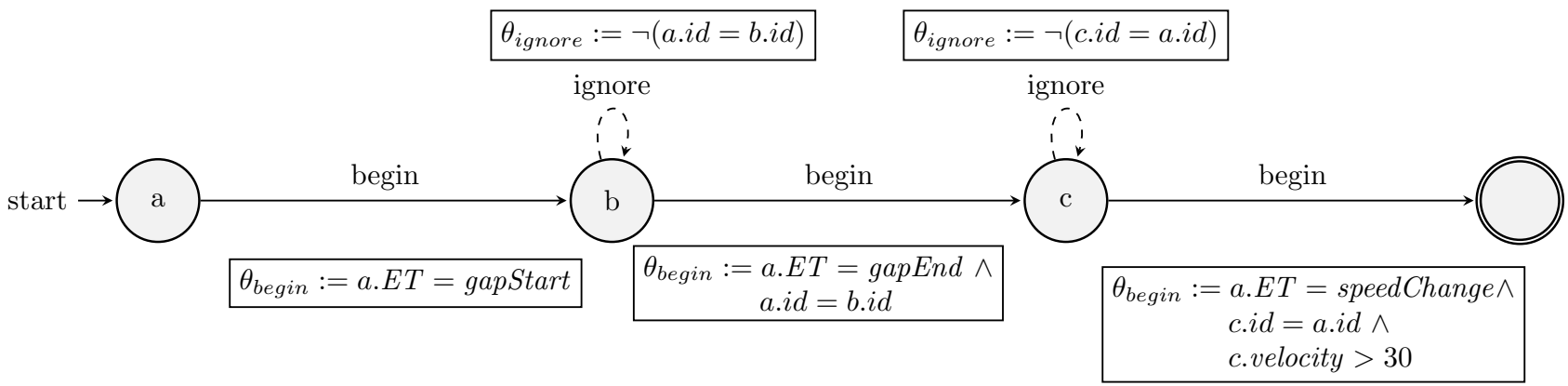
Constructing NFA^b example: strict–contiguity

- Predicates on *ignore* edges depend on selection strategy.
- Assume strict–contiguity. What predicate? $\theta_{ignore} := FALSE$



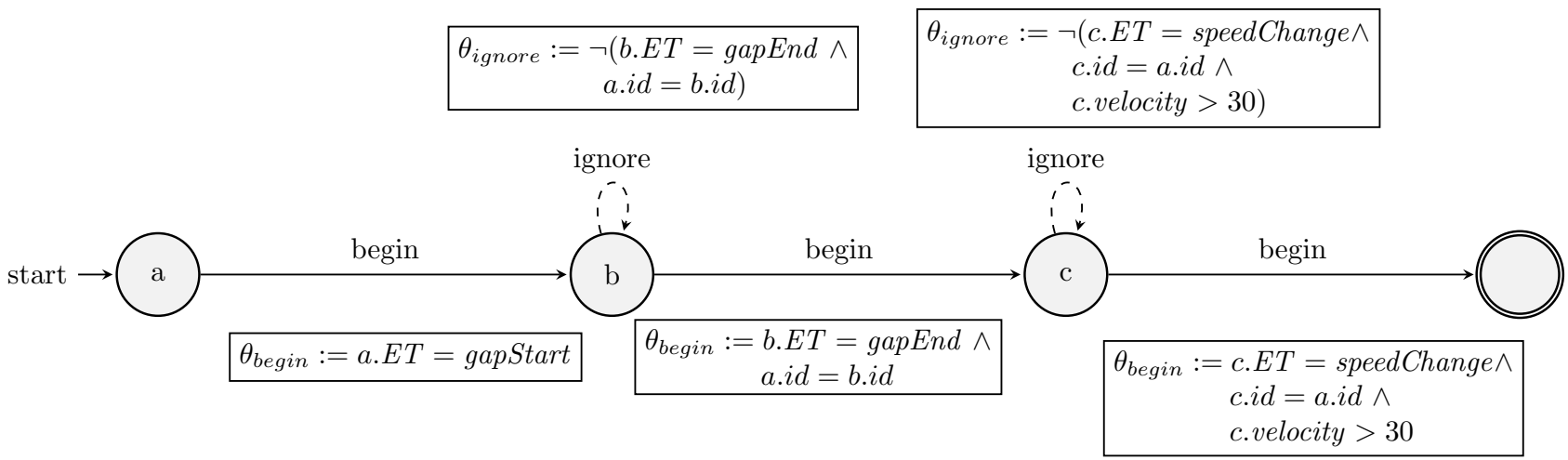
Constructing NFA^b example: partition–contiguity

- Predicates on *ignore* edges depend on selection strategy.
- Assume partition–contiguity. What predicate?
  $\theta_{ignore} := \neg(\textit{partition condition})$



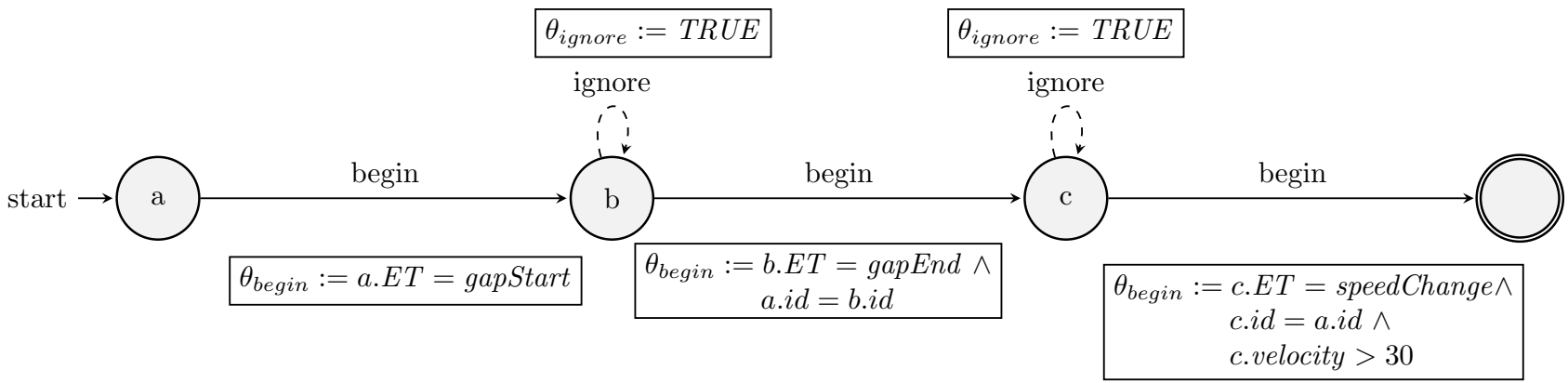
Constructing NFA^b example: skip–till–next

- Predicates on *ignore* edges depend on selection strategy.
- Assume skip–till–next. What predicate?
  $\theta_{ignore} := \neg(take\ or\ begin\ condition)$



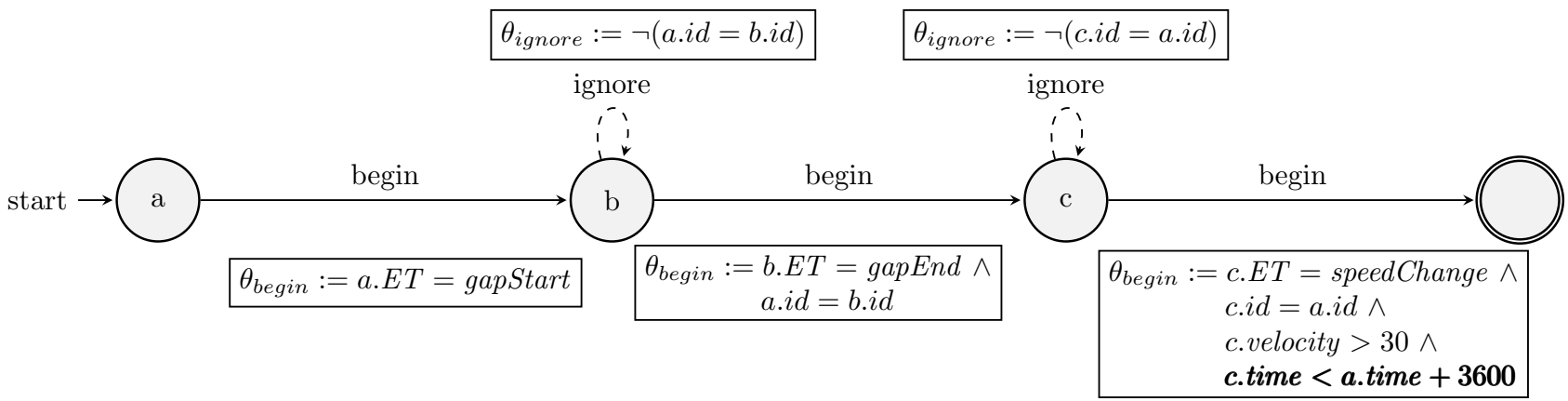
Constructing NFA^b example: skip–till–any

- Predicates on *ignore* edges depend on selection strategy.
- Assume skip–till–any. What predicate? $\theta_{ignore} := TRUE$



Constructing NFA^b example: window

- How do we treat windows?
- On the *begin* or *proceed* edge to the final state, add the within condition for the entire pattern.



Early optimizations

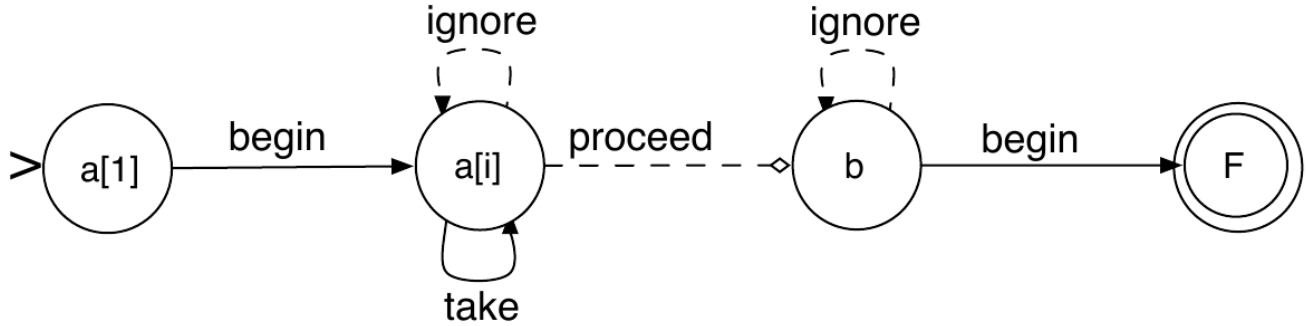
- Something better with windows?
  - Does it matter on which edge the within constraint is placed?
  - Can be copied on any edge, so move it early.
  - Invalid runs are not allowed to hang on.

When is a pattern detected?

- A NFA^b run is defined as :
  - the current state of NFA^b **AND**
  - the sequence of events selected into the buffer (the match) **AND**
  - the naming of the buffer elements, i.e., mapping of elements to states (why is this important?)
- A run is accepting if it has reached its final state.
- Each accepting run corresponds to a pattern match.

NFA^b example with matches

(a) NFA Structure



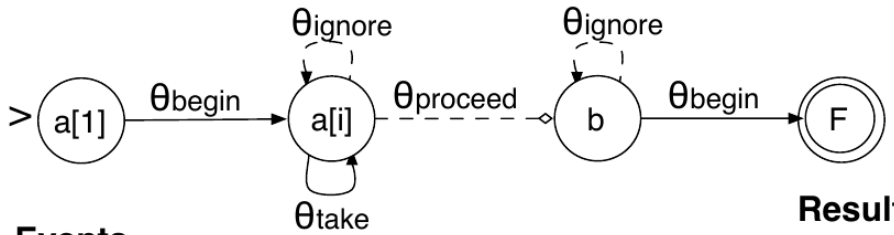
(b) Basic Formulas on Edges

θa[1]_begin = a[1].volume>1000	**θa[i]_take** = a[i].symbol=a[1].symbol ∧ a[i].price>avg(a[..i-1].price)	**θb_begin** = b.symbol=a[1].symbol ∧ b.volume<80%*a[a.LEN].volume ∧ b.time<a[1].time+1 hour
	θa[i]_proceed = True	
	θa[i]_ignore = ¬(a[i].symbol=a[1].symbol ∧ a[i].price>avg(a[..i-1].price)	**θb_ignore** = ¬(b.symbol=a[1].symbol ∧ b.volume<80%*a[a.LEN].volume)

Figure from [ADGI08]

Matches

θa[1]_begin =	**θa[i]_take** =	**θb_begin** =
a[1].volume>1000	a[i].symbol=a[1].symbol ∧ a[i].price>avg(a[..i-1].price)	b.symbol=a[1].symbol ∧ b.volume<80%*a[a.LEN].volume ∧ b.time<a[1].time+1 hour
	θa[i]_proceed = True	
	θa[i]_ignore = ¬(a[i].symbol=a[1].symbol ∧ a[i].price>avg(a[..i-1].price)	**θb_ignore** = ¬(b.symbol=a[1].symbol ∧ b.volume<80%*a[a.LEN].volume)



Events

	e1	**e2**	**e3**	**e4**	**e5**	**e6**	**e7**	**e8 ...**
price	100	120	120	121	120	125	120	120
volume	1010	990	1005	999	999	750	950	700

Results

	a[ ]	b
R1	[e1 e2 e3 e4 e5]	e6
R2	[e3 e4]	e6
R3	[e1 e2 e3 e4 e5 e6 e7]	e8
	...	...

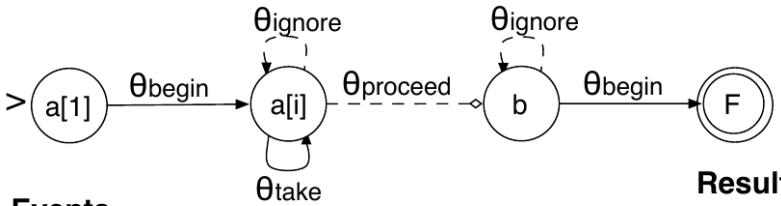
Figure from [ADGI08]

Runtime complexity

- Simultaneous runs (non–determinism)
  - *take–proceed*.
  - *ignore–proceed*.
  - *take–ignore*.
  - *begin–ignore*.
- Run duration
  - Large window values sustain runs longer.
  - Relaxed selection strategies allow more runs to survive until expiration.

Simultaneous runs: all selection strategies

θa[1]_begin =	**θa[i]_take** =	**θb_begin** =
a[1].volume>1000	a[i].symbol=a[1].symbol ∧ a[i].price>avg(a[..i-1].price)	b.symbol=a[1].symbol ∧ b.volume<80%*a[a.LEN].volume ∧ b.time<a[1].time+1 hour
	θa[i]_proceed = True	
	θa[i]_ignore = ¬(a[i].symbol=a[1].symbol ∧ a[i].price>avg(a[..i-1].price)	**θb_ignore** = ¬(b.symbol=a[1].symbol ∧ b.volume<80%*a[a.LEN].volume)



Events

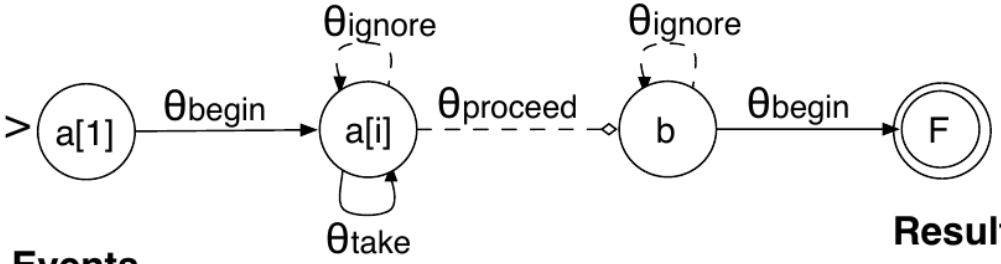
	e1	**e2**	**e3**	**e4**	**e5**	**e6**	**e7**	**e8 ...**
price	100	120	120	121	120	125	120	120
volume	1010	990	1005	999	999	750	950	700

Results

	a[ ]	b
R1	[e1 e2 e3 e4 e5]	e6
R2	[e3 e4]	e6
R3	[e1 e2 e3 e4 e5 e6 e7]	e8
	...	...

- What happens when *e*6 arrives?
- Both *take* and *proceed* edges triggered.
- Does selection strategy matter? No (simultaneous runs occur even with strict or partition–contiguity).

Buffer management (individual buffers)

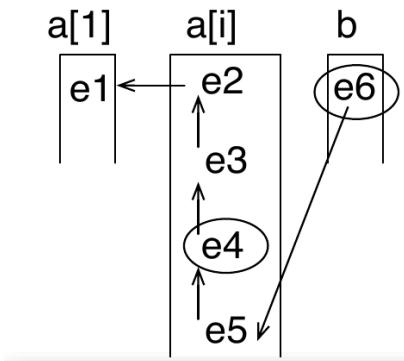


Events

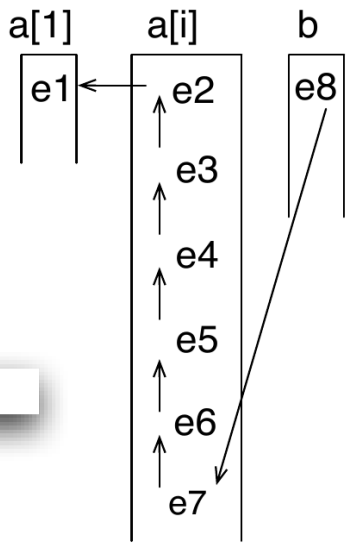
	e1	e2	e3	e4	e5	e6	e7	e8 ...
price	100	120	120	121	120	125	120	120
volume	1010	990	1005	999	999	750	950	700

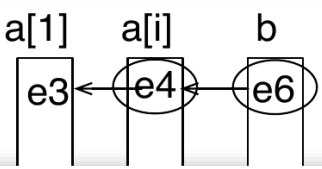
Results

	a[ ]	b
R1	[e1 e2 e3 e4 e5]	e6
R2	[e3 e4]	e6
R3	[e1 e2 e3 e4 e5 e6 e7]	e8
	...	...



(a) buffer for match R1





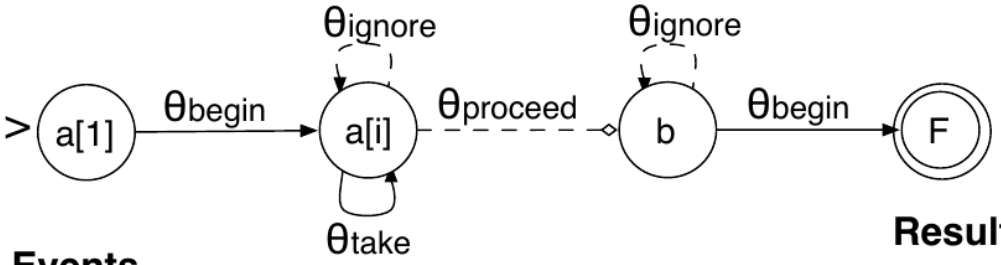
(b) buffer for match R2

(c) buffer for match R3

Buffer management (individual buffers)

- One stack for each state (except final).
- Stacks contain pointers to events that triggered *begin* or *take* moves.
- Each event has a pointer to the previously selected event (same or different stack).
- A match is retrieved by traversing the list starting from the last selected event.

Buffer sharing and versioning

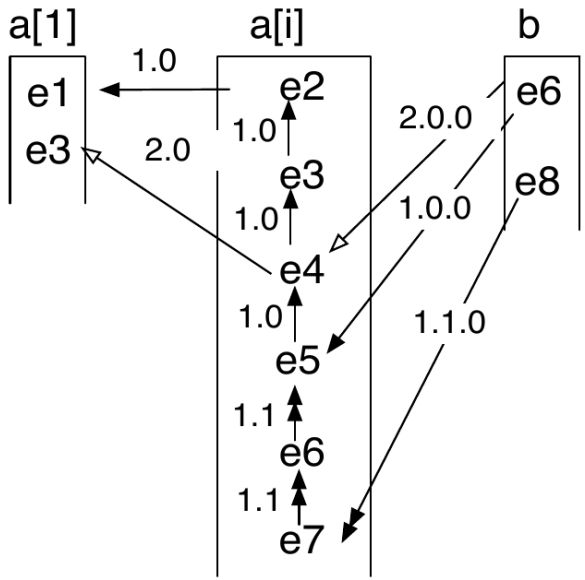


Events

	e1	e2	e3	e4	e5	e6	e7	e8 ...
price	100	120	120	121	120	125	120	120
volume	1010	990	1005	999	999	750	950	700

Results

	a[ ]	b
R1	[e1 e2 e3 e4 e5]	e6
R2	[e3 e4]	e6
R3	[e1 e2 e3 e4 e5 e6 e7]	e8
	...	...



(d) shared, versioned buffer for R1, R2, R3

Complex Event Recognition with FlinkCEP

FlinkCEP for Big Data processing

- Flink is a framework for real–time stream processing.
- FlinkCEP is a Flink library for Complex Event Processing.
- It is based on automata, very much in the spirit of SASE.
- An industrial solution with more features, flexibility and inherently scalable.

Why do we need new frameworks?

- 30.000 gigabytes of data generated every second.
- Extreme variety (server logs, clicks, sensor streams, scientific data).
- Traditional systems, like relational DBMSs, have failed to scale to Big Data (think of LAMP stack).

Lambda Architecture

- A general–purpose approach to implementing an *arbitrary* function on an *arbitrary* dataset and having the function return its results with *low latency*.

Lambda Architecture

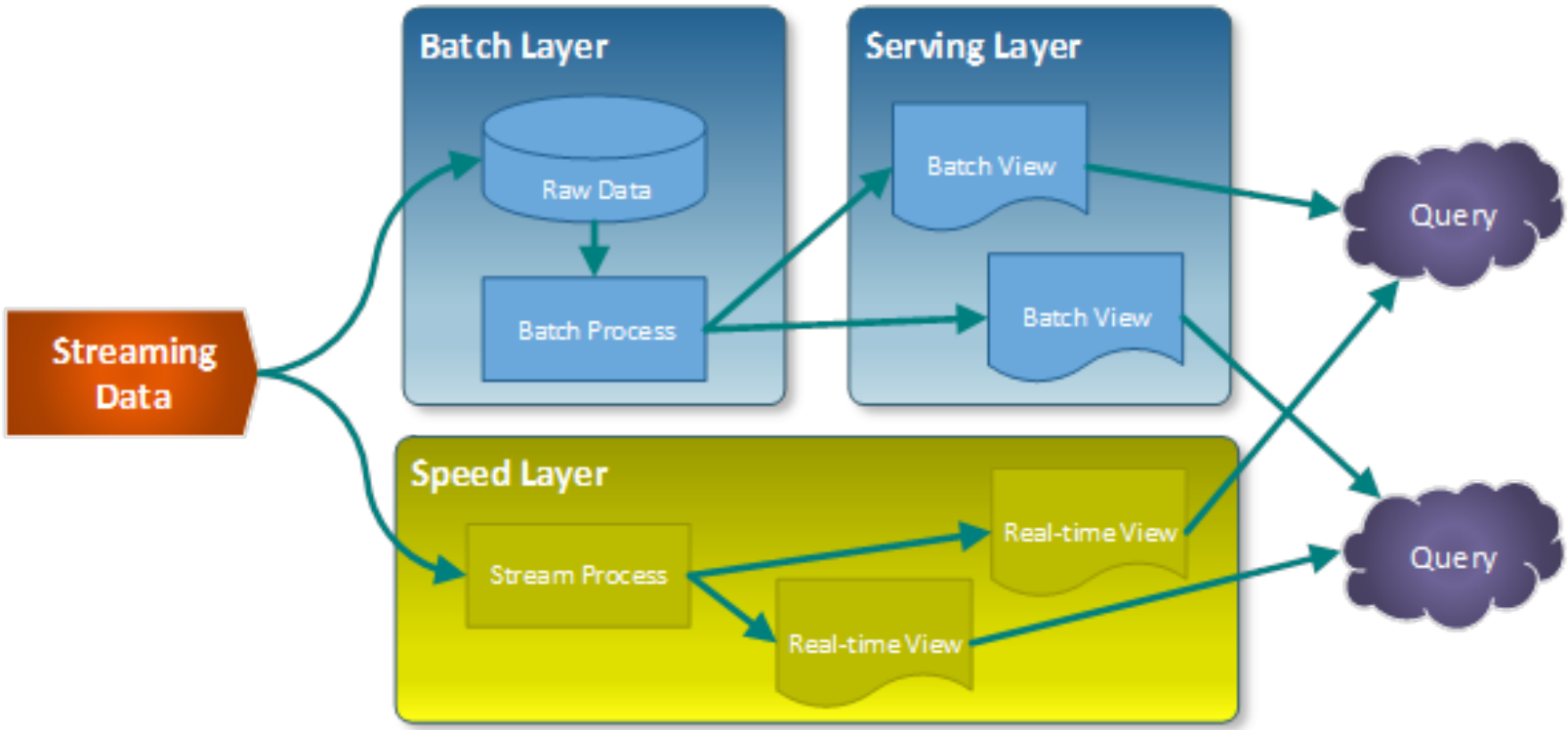
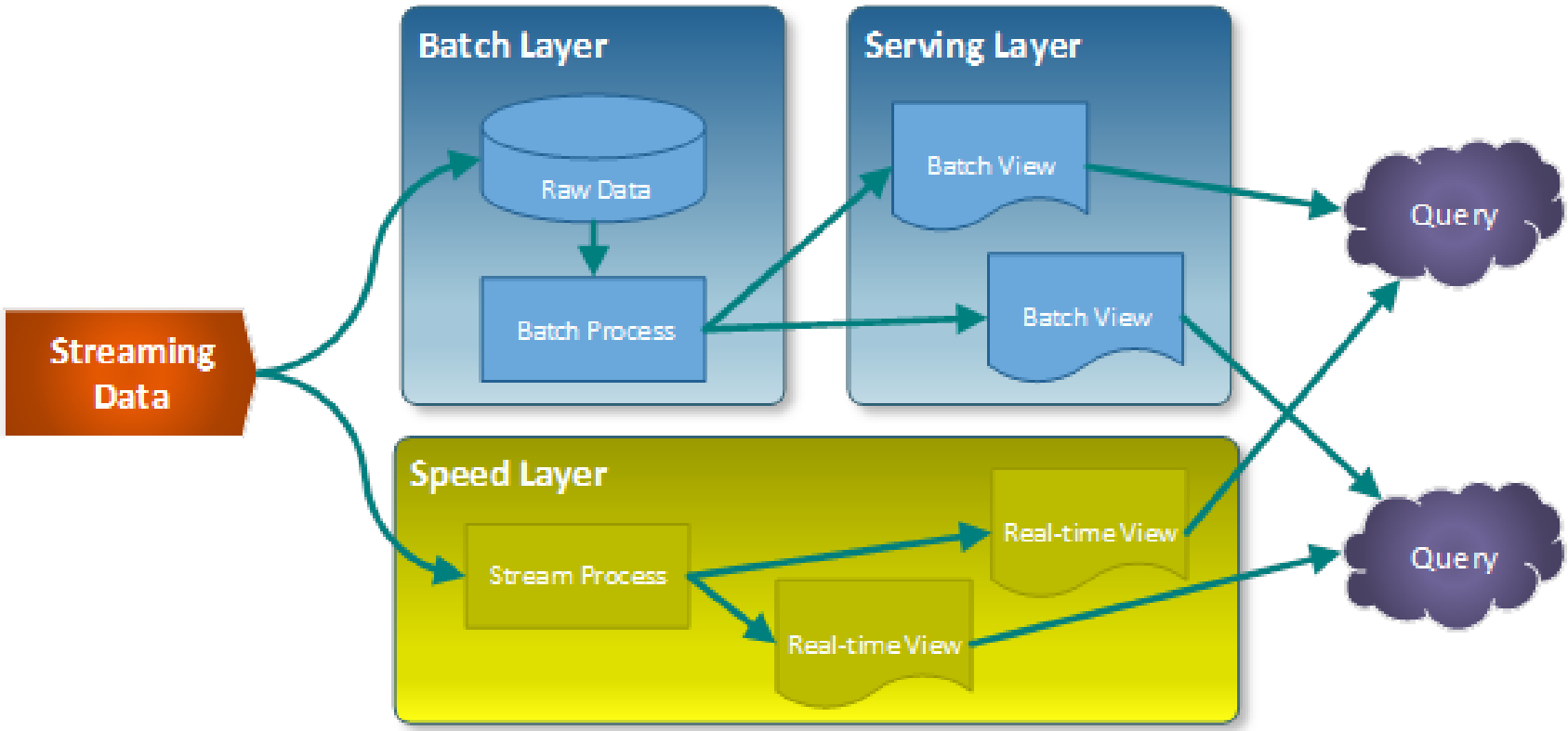


Figure from [Ver17]

Lambda Architecture

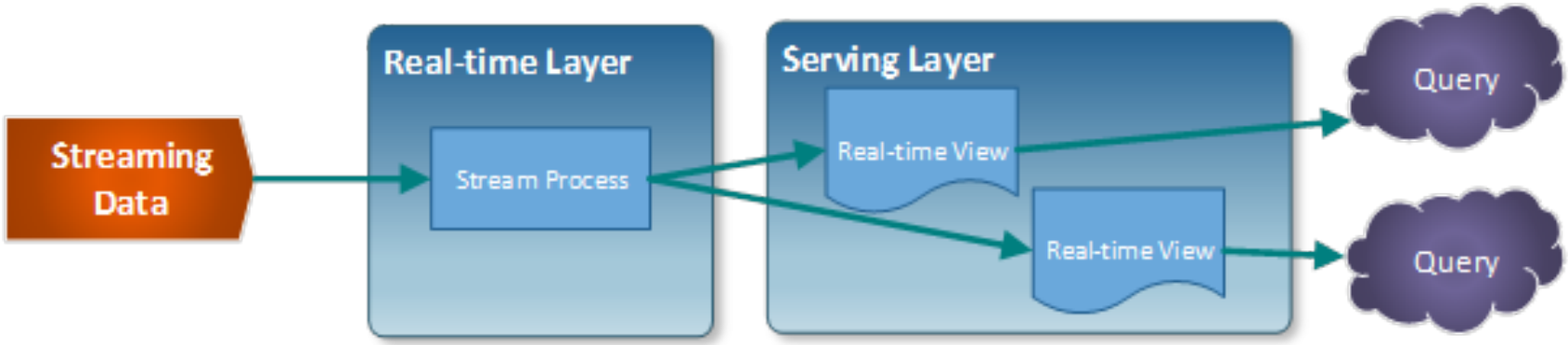
- *Complexity isolation* (data reaching the serving layer no longer needed in real–time views).
  - Why is this important?
  - Can discard all real–time views if anything goes wrong.
- Merge results from batch and real–time views.
- Exact results from batch, approximate from real–time.
- *Eventual accuracy* (batch layer repeatedly overrides the speed layer).

What is "wrong" with the lambda architecture?



- Logic duplication.
- Persistent storage.

Kappa Architecture



- A solution for streaming applications.
- No batch layer, stream processing.
- Advantages?
  - Avoids logic duplication (but needs to handle duplicate and out–of–order events).
  - Single point of contact for queries (avoids going through persistent storage).

Figure from [Ver17]

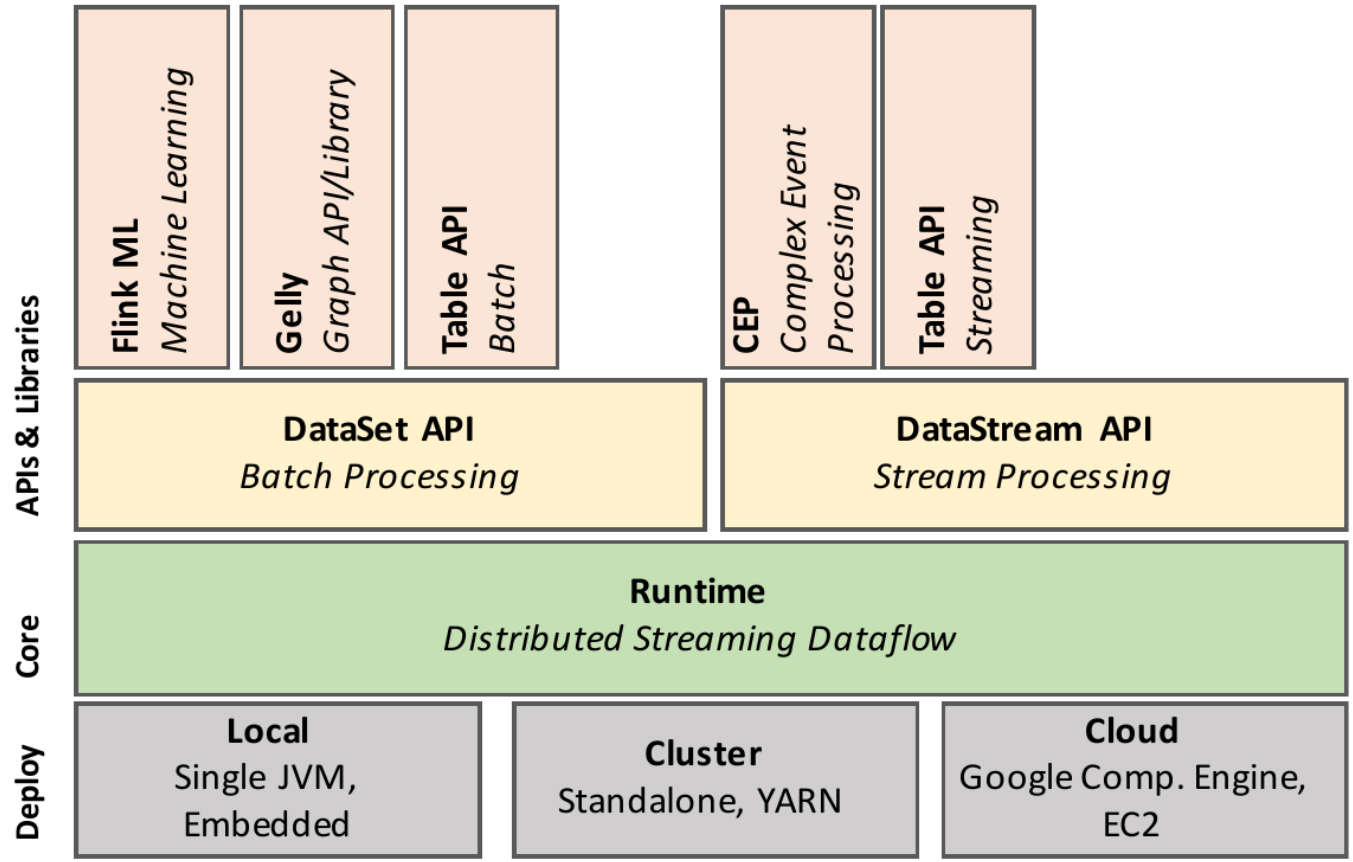
What is Flink?

- Open–source system for processing streaming and batch data.
- Can accommodate
  - Real–time analytics.
  - Historic data processing (batch).
  - Iterative algorithms (machine learning, graph analysis).
  - Continuous data pipelines.

Flink features

- True streaming
  - No micro–batches, e.g., Spark.
  - Windows are logical views, not actual data buffers.
  - Event historical data processing done by treating data as streams.
- High performance, low latency.
- Out–of–order events.
- Fault tolerance.
- Scalability.
- Exactly–once semantics for stateful computations (computations that need memory, e.g., calculating an average).
- Various transformations (map, filter, fold, window, aggregations, window join, etc.).

CEP in Flink stack



- Library on top of Flink.
- Uses the DataStream API.

Figure from [CKE+15].

FlinkCEP patterns

- Two types
  - Individual patterns.
  - Complex patterns (or pattern sequences), composed of individual patterns.

FlinkCEP individual patterns

- Individual stream elements with the same properties.
- Unique name.
- Conditions to accept element.
- Each individual pattern expects exactly one event by default
- ... but quantifiers (iteration with bounds) can also be used.

Individual patterns

- Singleton patterns: accept a single event.
- Looping patterns: accept one or more events.
- Example: $R = x \cdot y^+ \cdot z$.
  - Singleton? x and z.
  - Looping? y^+.

Looping patterns

```
// expecting 1 or more occurrences
somePattern.oneOrMore()

// expecting 0 or more occurrences
somePattern.oneOrMore().optional()

// expecting 4 occurrences
somePattern.times(4)

// expecting 0 or 4 occurrences
somePattern.times(4).optional()

// expecting 2, 3 or 4 occurrences
somePattern.times(2, 4)

// expecting 2 or more occurrences
somePattern.timesOrMore(2)
```

Differences to SASE

- Both support Kleene plus. What's missing in SASE?
- Kleene star. Expressed as? **.oneOrMore.optional**
- Exact counts, **.times(n)**.
- Bounds, **.times(n,k)**.

Simple conditions

```
// select all turns
turn.where(event => event.getName.equals("turn"))

// Define speedChange when you have either acceleration or deceleration
speedChange.where(event =>
     event.getName.equals("acceleration")).or(event =>
     event.getName.equals("deceleration"))

turn.where(event => event.getName.equals("turn")).where(event =>
     event.getTurnRate > 20)
```

- What Boolean operator does the last pattern express? **AND**.

Complex patterns

- A complex pattern (or sequence pattern) is composed of:
  - A sequence of individual patterns.
  - Selection strategy (contiguity condition in Flink terminology).
  - Time constraints.
  - Individual patterns connected through contiguity conditions.

FlinkCEP selection strategies

- $S = \{a, b, a, a, c, b\}$, $R = a \cdot b$
- Strict contiguity: $M_1 = \{1, 2\}$
- Relaxed contiguity: $M_1 = \{1, 2\}$, $M_2 = \{4, 6\}$
- Non-deterministic relaxed contiguity: $M_1 = \{1, 2\}$, $M_2 = \{4, 6\}$, but also $M_3 = \{3, 6\}$, $M_4 = \{1, 6\}$.
- Difference between SASE and FlinkCEP? Relaxed contiguity in FlinkCEP is stricter than skip–till-next in SASE (SASE would also detect $\{3, 6\}$).

Example pattern

```
// initial pattern to start the sequence
val start: Pattern[Event, _] = Pattern.begin("start")

// strict contiguity
val idle: Pattern[Event, _] = start.next("idle").where(event =>
    event.getSpeed < 0.1)

// relaxed contiguity
val idle: Pattern[Event, _] = start.followedBy("idle").where(event =>
    event.getSpeed < 0.1)

// non-deterministic relaxed contiguity
val idle: Pattern[Event, _] = start.followedByAny("idle").where(event
    => event.getSpeed < 0.1)

// strict contiguity, add another individual pattern
val abruptStart: Pattern[Event, _] =
    idle.next("abruptStart").where(event => event.getSpeed > 15)

// window
abruptStart.within(Time.seconds(10))
```

FlinkCEP features

- Simple conditions: filtering on single events.
- Iterative conditions: move forward according by taking into account previous events.
- Combine condition with logical **AND**, **OR**.
- Quantifiers.
- Windows (and timeouts).
- Hierarchies.
- Selection strategies.
- Out–of–order events.

Open issues

Nice features of automata

- Automata-based systems seem to satisfy many of the requirements for CER.
- Can naturally handle sequences of events.
- Industrial solutions, such as FlinkCEP, offer a significant number of extra features, thus providing substantial flexibility for defining patterns.
- Plus features for Big Data solutions, e.g., distribution, fault tolerance, out-of-order events.

A closer look

- Semantics provided only through the operational semantics of automata.
- How about the declarative semantics of the language? Missing.
- Example 1: can we use negation arbitrarily?
- But register automata are not closed under complement.
- Example 2: can we use Kleene-start arbitrarily?
- Does not seem to be the case. Loops allowed only on a single state.
- Can we re-write expressions (e.g., for optimization purposes)? Unclear.
- See [GRU19, GAA+20].

Further issues

- Background knowledge.
- Semantics of selection strategies may differ (e.g., SASE vs FlinkCEP).
- How to handle concurrency? Petri nets?
- Patterns without windows.
- Learning of sequential patterns.

References I


Elias Alevizos, Alexander Artikis, and George Paliouras, *Event forecasting with pattern markov chains*, DEBS, ACM, 2017, pp. 146–157.


_____, *Symbolic automata with memory: a computational model for complex event processing*, CoRR **abs/1804.09999** (2018).


_____, *Wayeb: a tool for complex event forecasting*, LPAR, EPiC Series in Computing, vol. 57, EasyChair, 2018, pp. 26–35.


Adnan Akbar, François Carrez, Klaus Moessner, and Ahmed Zoha, *Predicting complex events for pro-active iot applications*, WF-IoT, IEEE Computer Society, 2015, pp. 327–332.

References II


Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman, *Efficient pattern matching over event streams*, Proceedings of the 2008 ACM SIGMOD international conference on Management of data, ACM, 2008, pp. 147–160.


Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas, *Apache flink: Stream and batch processing in a single engine*, Bulletin of the IEEE Computer Society Technical Committee on Data Engineering **36** (2015), no. 4.


Gianpaolo Cugola and Alessandro Margara, *Processing flows of information: From data stream to complex event processing*, ACM Comput. Surv. **44** (2012), no. 3, 15:1–15:62.

References III


Loris D'Antoni and Margus Veanes, *The power of symbolic automata and transducers*, CAV (1), Lecture Notes in Computer Science, vol. 10426, Springer, 2017, pp. 47–67.


Nikos Giatrakos, Elias Alevizos, Alexander Artikis, Antonios Deligiannakis, and Minos N. Garofalakis, *Complex event recognition in the big data era: a survey*, VLDB J. **29** (2020), no. 1, 313–352.


Alejandro Grez, Cristian Riveros, and Martín Ugarte, *A formal framework for complex event processing*, ICDT, LIPIcs, vol. 127, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, pp. 5:1–5:18.


Vinod Muthusamy, Haifeng Liu, and Hans-Arno Jacobsen, *Predictive publish/subscribe matching*, DEBS, ACM, 2010, pp. 14–25.

References IV

- Suraj Pandey, Surya Nepal, and Shiping Chen, *A test-bed for the evaluation of business process prediction techniques*, CollaborateCom, ICST / IEEE, 2011, pp. 382–391.
- Wil M. P. van der Aalst, M. H. Schonenberg, and Minseok Song, *Time prediction based on process mining*, Inf. Syst. **36** (2011), no. 2, 450–475.
- Michael Verrilli, *From Lambda to Kappa: A Guide on Real-time Big Data Architectures*, 2017.
- Eugene Wu, Yanlei Diao, and Shariq Rizvi, *High-performance complex event processing over streams*, Proceedings of the 2006 ACM SIGMOD international conference on Management of data, ACM, 2006, pp. 407–418.

References V

 Haopeng Zhang, Yanlei Diao, and Neil Immerman, *On Complexity and Optimization of Expensive Queries in Complex Event Processing*, Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (New York, NY, USA), SIGMOD '14, ACM, 2014, pp. 217–228.