

01

## Information Flow Processing

## ■ Author

- ◆ João Moura Pires ([jmp@fct.unl.pt](mailto:jmp@fct.unl.pt))
- ◆ José Júlio Alferes ([jja@fct.unl.pt](mailto:jja@fct.unl.pt))

- This material can be freely used for personal or academic purposes without any previous authorization from the author, provided that this notice is maintained/kept.
- For commercial purposes the use of any part of this material requires the previous authorization from the author(s).

# Bibliography

- Many examples are extracted and adapted from:

- ◆ **Processing Flows of Information: From Data Stream to Complex Event Processing**, GIANPAOLO CUGOLA and ALESSANDRO MARGARA, Politecnico di

Milano, ACM Computing Surveys, Vol. 44, No. 3, Article 15, Publication date: June 2012

- ◆ **Event Processing in Action**, OPHER ETZION PETER NIBLETT, 2011, Manning

Publications Co

# Table of Contents

## ■ Introduction and Motivation

# Table of Contents

## ■ Introduction and Motivation

## ■ Background

# Table of Contents

## ■ Introduction and Motivation

## ■ Background

## ■ Active Database Systems

# Table of Contents

- **Introduction and Motivation**
- **Background**
- **Active Database Systems**
- **Data Stream Management Systems**

# Table of Contents

- **Introduction and Motivation**
- **Background**
- **Active Database Systems**
- **Data Stream Management Systems**
- **Complex Event Processing Systems**

# Table of Contents

- **Introduction and Motivation**
- **Background**
- **Active Database Systems**
- **Data Stream Management Systems**
- **Complex Event Processing Systems**
- **Framework for IFP Systems**

## Introduction and Motivation

# Application Domains

- Applications requires processing **continuously flowing data from geographically distributed sources at unpredictable rates to obtain timely responses to complex queries**

# Application Domains

- Applications requires processing **continuously flowing data from geographically distributed sources at unpredictable rates to obtain timely responses to complex queries**
  - ◆ Wireless sensor networks

# Application Domains

- Applications requires processing **continuously flowing data from geographically distributed sources at unpredictable rates to obtain timely responses to complex queries**
  - ◆ Wireless sensor networks
  - ◆ Financial tickers

# Application Domains

- Applications requires processing **continuously flowing data from geographically distributed sources at unpredictable rates to obtain timely responses to complex queries**
  - ◆ Wireless sensor networks
  - ◆ Financial tickers
  - ◆ Fraud detection

# Application Domains

■ Applications requires processing **continuously flowing data from geographically distributed sources at unpredictable rates to obtain timely responses to complex queries**

- ◆ Wireless sensor networks
- ◆ Financial tickers
- ◆ Fraud detection
- ◆ Traffic management

# Application Domains

■ Applications requires processing **continuously flowing data from geographically distributed sources at unpredictable rates to obtain timely responses to complex queries**

- ◆ Wireless sensor networks
- ◆ Financial tickers
- ◆ Fraud detection
- ◆ Traffic management
- ◆ Delivery systems, etc...

# Application Domains

- Applications requires processing **continuously flowing data from geographically distributed sources at unpredictable rates to obtain timely responses to complex queries**
  - ◆ Wireless sensor networks
  - ◆ Financial tickers
  - ◆ Fraud detection
  - ◆ Traffic management
  - ◆ Delivery systems, etc...
- Information Flow Processing (IFP) domains

# Application Domains

- Applications requires processing **continuously flowing data from geographically distributed sources at unpredictable rates to obtain timely responses to complex queries**
  - ◆ Wireless sensor networks
  - ◆ Financial tickers
  - ◆ Fraud detection
  - ◆ Traffic management
  - ◆ Delivery systems, etc...
- Information Flow Processing (IFP) domains
- Information Flow Processing (IFP) engine

# Application Domains

- The concepts of **timeliness** and **flow processing** are crucial for justifying the need for a **new class of systems**

# Application Domains

- The concepts of **timeliness** and **flow processing** are crucial for justifying the need for a **new class of systems**
- In traditional DBMSs
  - ◆ Require **data to be (persistently) stored and indexed** before it could be processed
  - ◆ Process data **only when explicitly asked by the users**, i.e., asynchronously with respect to its arrival

# Application Domains

- The concepts of **timeliness** and **flow processing** are crucial for justifying the need for a **new class of systems**
- In traditional DBMSs
  - ◆ Require **data to be (persistently) stored and indexed** before it could be processed
  - ◆ Process data **only when explicitly asked by the users**, i.e., asynchronously with respect to its arrival
- Example: **Detecting fire in a building by using temperature and smoke sensors**
  - ◆ A fire alert has to be notified as soon as the relevant data becomes available

# Application Domains

- The concepts of **timeliness** and **flow processing** are crucial for justifying the need for a **new class of systems**
- In traditional DBMSs
  - ◆ Require **data to be (persistently) stored and indexed** before it could be processed
  - ◆ Process data **only when explicitly asked by the users**, i.e., asynchronously with respect to its arrival
- Example: **Detecting fire in a building by using temperature and smoke sensors**
  - ◆ A fire alert has to be notified as soon as the relevant data becomes available
  - ◆ There is no need to store sensor readings if they are not relevant to fire

# Application Domains

- The concepts of **timeliness** and **flow processing** are crucial for justifying the need for a **new class of systems**
- In traditional DBMSs
  - ◆ Require **data to be (persistently) stored and indexed** before it could be processed
  - ◆ Process data **only when explicitly asked by the users**, i.e., asynchronously with respect to its arrival
- Example: **Detecting fire in a building by using temperature and smoke sensors**
  - ◆ A fire alert has to be notified as soon as the relevant data becomes available
  - ◆ There is no need to store sensor readings if they are not relevant to fire
  - ◆ The relevant data can be discarded as soon as the fire is detected, since all the information they carry

# Application Domains

- The concepts of **timeliness** and **flow processing** are crucial for justifying the need for a **new class of systems**
- In traditional DBMSs
  - ◆ Require **data to be (persistently) stored and indexed** before it could be processed
  - ◆ Process data **only when explicitly asked by the users**, i.e., asynchronously with respect to its arrival
- Example: **Detecting fire in a building by using temperature and smoke sensors**
  - ◆ A fire alert has to be notified as soon as the relevant data becomes available
  - ◆ There is no need to store sensor readings if they are not relevant to fire
  - ◆ The relevant data can be discarded as soon as the fire is detected, since all the information they carry

Very Dynamic Environment

# Dynamic Data Example

## ■ Using a sensor network measuring temperature and smoke, for fire alerts

- ◆ We want data to be processed continuously for detecting the fire prone conditions, and not only when users query
- ◆ We don't want to store all the measures, especially those that have nothing to do with fire conditions.

# Dynamic Data Example

## ■ Using a sensor network measuring temperature and smoke, for fire alerts

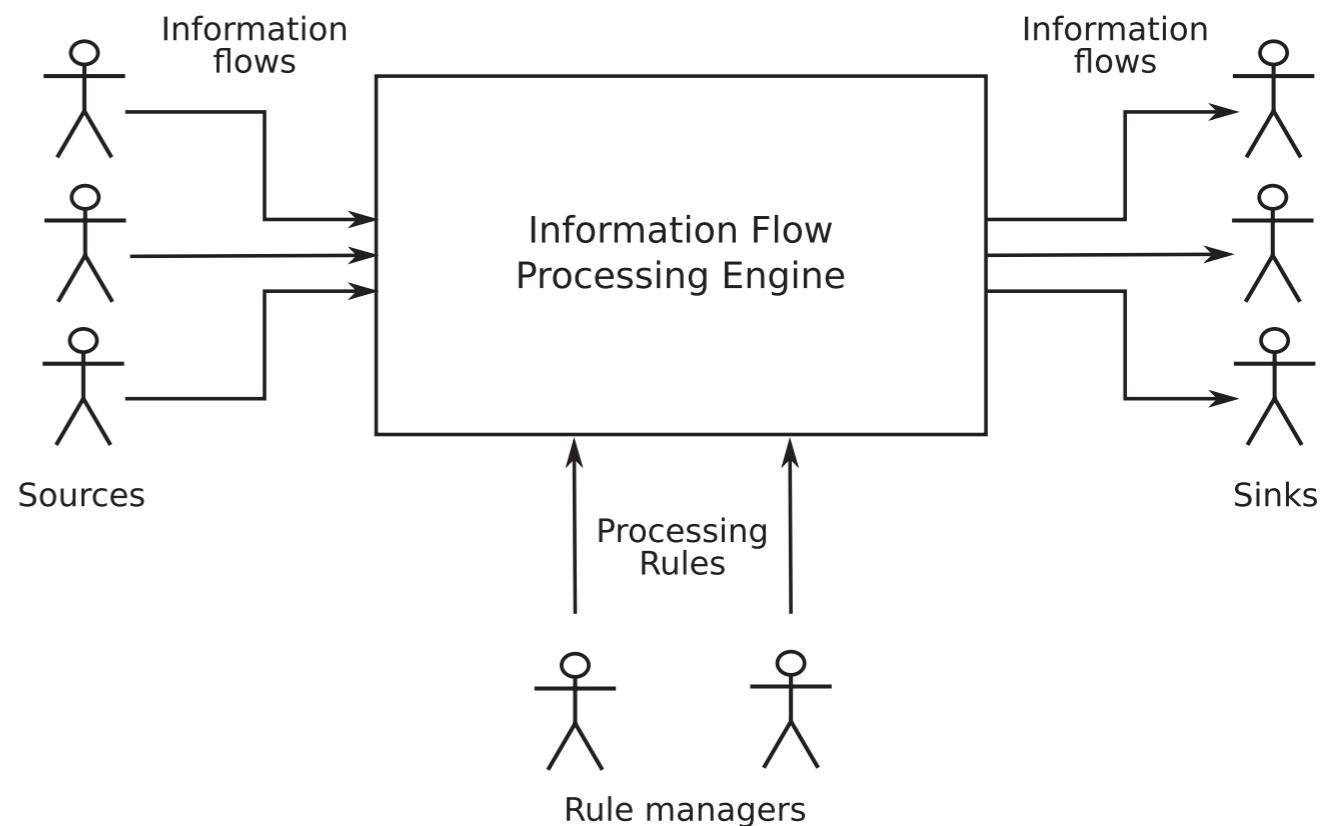
- ◆ We want data to be processed continuously for detecting the fire prone conditions, and not only when users query
- ◆ We don't want to store all the measures, especially those that have nothing to do with fire conditions.
  - Even those that alert for fire, are only needed until the fire alert is emitted; after that we may discard them

# Dynamic Data Example

- **Using a sensor network measuring temperature and smoke, for fire alerts**
  - ◆ We want data to be processed continuously for detecting the fire prone conditions, and not only when users query
  - ◆ We don't want to store all the measures, especially those that have nothing to do with fire conditions.
    - Even those that alert for fire, are only needed until the fire alert is emitted; after that we may discard them
- **Implementing this in a system designed for static data (e.g. a DBMS) is not adequate**

# Approaches to IFPs

- Systems specifically designed to process information as a flow (or a set of flows) according to a set of pre-deployed processing rules.
- These systems differ in a wide range of aspects, including
  - ◆ Architecture
  - ◆ Data models,
  - ◆ Rule languages
  - ◆ Processing mechanisms



# Approaches to IFPs

## ■ Two main models

- ◆ The **data stream processing model** [Babcock et al. 2002]
  - Processing streams of data coming from different sources to produce new data streams as output
  - **Evolution of traditional data processing, as supported by DBMSs** to become the **Data Stream Management Systems (DSBMs)**

# Approaches to IFPs

## ■ Two main models

- ◆ The **data stream processing model** [Babcock et al. 2002]
  - Processing streams of data coming from different sources to produce new data streams as output
  - **Evolution of traditional data processing, as supported by DBMSs** to become the **Data Stream Management Systems (DSBMs)**
- ◆ The **complex event processing model** [Luckham 2001].
  - flowing **information items** as **notifications of events happening in the external world**, which have to be **filtered and combined** to understand what is happening in terms of **higher-level events**

# Approaches to IFPs: **data stream** processing model

- Evolution of traditional data processing, as supported by DBMSs to become the **Data Stream Management Systems (DSBMs)**

# Approaches to IFPs: **data stream** processing model

- Evolution of traditional data processing, as supported by DBMSs to become the **Data Stream Management Systems (DSBMs)**
  - Traditional DBMSs are designed to work on **persistent data** where **updates** are relatively **infrequent**

# Approaches to IFPs: **data stream** processing model

- Evolution of traditional data processing, as supported by DBMSs to become the **Data Stream Management Systems (DSBMs)**
  - Traditional DBMSs are designed to work on **persistent data** where **updates** are relatively **infrequent**
  - DSMSs are specialized in dealing with **transient data** that is **continuously updated**

# Approaches to IFPs: **data stream** processing model

- Evolution of traditional data processing, as supported by DBMSs to become the **Data Stream Management Systems (DSBMs)**
  - Traditional **DBMSs** are designed to work on **persistent data** where **updates** are relatively **infrequent**
  - **DSMSs** are specialized in dealing with **transient data** that is **continuously updated**
  - **DBMSs run queries just once** to return a complete answer,

# Approaches to IFPs: **data stream** processing model

## ■ Evolution of traditional data processing, as supported by DBMSs to become the **Data Stream Management Systems (DSBMs)**

- Traditional **DBMSs** are designed to work on **persistent data** where **updates** are relatively **infrequent**
- **DSMSs** are specialized in dealing with **transient data** that is **continuously updated**
- **DBMSs run queries just once** to return a complete answer,
- **DSMSs execute standing queries** which **run continuously** and **provide updated answers as new data arrives**

# Approaches to IFPs: **data stream** processing model

## ■ Evolution of traditional data processing, as supported by DBMSs to become the **Data Stream Management Systems (DSBs)**

- Traditional **DBMSs** are designed to work on **persistent data** where **updates** are relatively **infrequent**
- **DSMSs** are specialized in dealing with **transient data** that is **continuously updated**
- **DBMSs run queries just once** to return a complete answer,
- **DSMSs execute standing queries** which **run continuously** and **provide updated answers as new data arrives**
- **DSMSs process incoming data through a sequence of transformations based on common SQL operators**, like selections, aggregates, joins, and all the operators defined in general by relational algebra

# Approaches to IFPs: **complex event** processing model

- Flowing **information items** as notifications of events happening in the external world, which have to be filtered and combined to **understand what is happening in terms of higher-level events**

# Approaches to IFPs: **complex event** processing model

- Flowing **information items** as notifications of events happening in the external world, which have to be filtered and combined to **understand what is happening in terms of higher-level events**
  - focus of this model is on **detecting occurrences of particular patterns** of (low-level) events that represent the higher-level events

# Approaches to IFPs: **complex event** processing model

- Flowing **information items** as notifications of events happening in the external world, which have to be filtered and combined to **understand what is happening in terms of higher-level events**
  - focus of this model is on **detecting occurrences of particular patterns** of (low-level) events that represent the higher-level events
  - The contributions to this model come from different communities:
    - distributed information systems,

# Approaches to IFPs: **complex event** processing model

- Flowing **information items** as notifications of events happening in the external world, which have to be filtered and combined to **understand what is happening in terms of higher-level events**
  - focus of this model is on **detecting occurrences of particular patterns** of (low-level) events that represent the higher-level events
  - The contributions to this model come from different communities:
    - distributed information systems,
    - business process automation,

# Approaches to IFPs: **complex event** processing model

- Flowing **information items** as notifications of events happening in the external world, which have to be filtered and combined to **understand what is happening in terms of higher-level events**
  - focus of this model is on **detecting occurrences of particular patterns** of (low-level) events that represent the higher-level events
  - The contributions to this model come from different communities:
    - distributed information systems,
    - business process automation,
    - control systems,

# Approaches to IFPs: **complex event** processing model

- Flowing **information items** as notifications of events happening in the external world, which have to be filtered and combined to **understand what is happening in terms of higher-level events**
  - focus of this model is on **detecting occurrences of particular patterns** of (low-level) events that represent the higher-level events
  - The contributions to this model come from different communities:
    - distributed information systems,
    - business process automation,
    - control systems,
    - network monitoring, sensor networks,

# Approaches to IFPs: **complex event** processing model

- Flowing **information items** as notifications of events happening in the external world, which have to be filtered and combined to **understand what is happening in terms of higher-level events**
  - focus of this model is on **detecting occurrences of particular patterns** of (low-level) events that represent the higher-level events
  - The contributions to this model come from different communities:
    - distributed information systems,
    - business process automation,
    - control systems,
    - network monitoring, sensor networks,
    - and middleware, in general

## Background

# Information Flow Processing (IFP) Domain

## ■ IFP application domain

- ◆ users need to **collect information produced by multiple, distributed sources** for **processing it in a timely way** in order to **extract new knowledge as soon as the relevant information is collected.**

# Information Flow Processing (IFP) Domain

## ■ IFP application domain

- ◆ users need to **collect information produced by multiple, distributed sources** for **processing it in a timely way** in order to **extract new knowledge as soon as the relevant information is collected.**

## ■ Examples

- ◆ Environmental monitoring, users need to process data coming from sensors deployed in the field to acquire information about the observed world, **detect anomalies**, or **predict disasters as soon as possible.**

# Information Flow Processing (IFP) Domain

## ■ IFP application domain

- ◆ users need to **collect information produced by multiple, distributed sources** for **processing it in a timely way** in order to **extract new knowledge as soon as the relevant information is collected.**

## ■ Examples

- ◆ Environmental monitoring, users need to process data coming from sensors deployed in the field to acquire information about the observed world, **detect anomalies**, or **predict disasters as soon as possible.**
- ◆ Financial applications require a continuous analysis of stocks to **identify trends.**

# Information Flow Processing (IFP) Domain

## ■ IFP application domain

- ◆ users need to **collect information produced by multiple, distributed sources** for **processing it in a timely way** in order to **extract new knowledge as soon as the relevant information is collected.**

## ■ Examples

- ◆ Environmental monitoring, users need to process data coming from sensors deployed in the field to acquire information about the observed world, **detect anomalies**, or **predict disasters as soon as possible.**
- ◆ Financial applications require a continuous analysis of stocks to **identify trends**.
- ◆ Fraud detection continuous streams of credit card transactions to be observed and inspected to **prevent frauds.**

# Information Flow Processing (IFP) Domain

## ■ IFP application domain

- ◆ users need to **collect information produced by multiple, distributed sources** for **processing it in a timely way** in order to **extract new knowledge as soon as the relevant information is collected.**

## ■ Examples

- ◆ Network intrusion detection have to analyze network traffic in real-time, generating alerts when something unexpected happens.

# Information Flow Processing (IFP) Domain

## ■ IFP application domain

- ◆ users need to **collect information produced by multiple, distributed sources** for **processing it in a timely way** in order to **extract new knowledge as soon as the relevant information is collected.**

## ■ Examples

- ◆ Network intrusion detection have to analyze network traffic in real-time, generating alerts when something unexpected happens.
- ◆ RFID-based inventory management performs continuous analysis of RFID readings to track valid paths of shipments and to capture irregularities

# Information Flow Processing (IFP) Domain

## ■ IFP application domain

- ◆ users need to **collect information produced by multiple, distributed sources** for **processing it in a timely way** in order to **extract new knowledge as soon as the relevant information is collected.**

## ■ Examples

- ◆ Network intrusion detection have to analyze network traffic in real-time, generating alerts when something unexpected happens.
- ◆ RFID-based inventory management performs continuous analysis of RFID readings to track valid paths of shipments and to capture irregularities
- ◆ Manufacturing control systems often require anomalies to be detected and signaled by looking at the information that describe how the system behaves

# Information Flow Processing (IFP) Domain

## ■ IFP application domain

- ◆ users need to **collect information produced by multiple, distributed sources** for **processing it in a timely way** in order to **extract new knowledge as soon as the relevant information is collected.**

## ■ Common to these examples

# Information Flow Processing (IFP) Domain

## ■ IFP application domain

- ◆ users need to **collect information produced by multiple, distributed sources** for **processing it in a timely way** in order to **extract new knowledge as soon as the relevant information is collected.**

## ■ Common to these examples

- ◆ the need for **processing information as it flows** from the periphery to the center of the system **without requiring**, at least in principle, the **information to be persistently stored.**

# Information Flow Processing (IFP) Domain

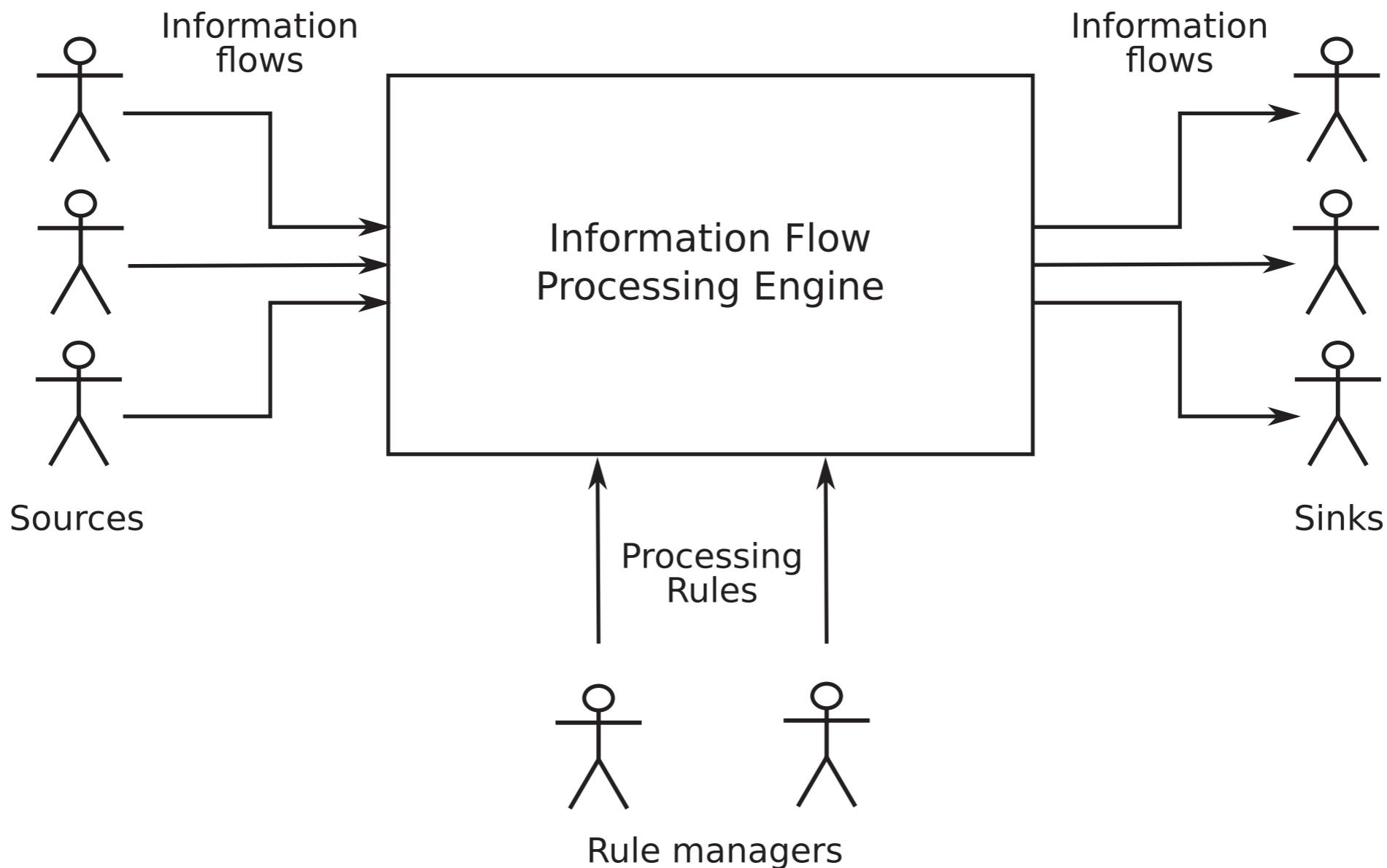
## ■ IFP application domain

- ◆ users need to **collect information produced by multiple, distributed sources** for **processing it in a timely way** in order to **extract new knowledge as soon as the relevant information is collected.**

## ■ Common to these examples

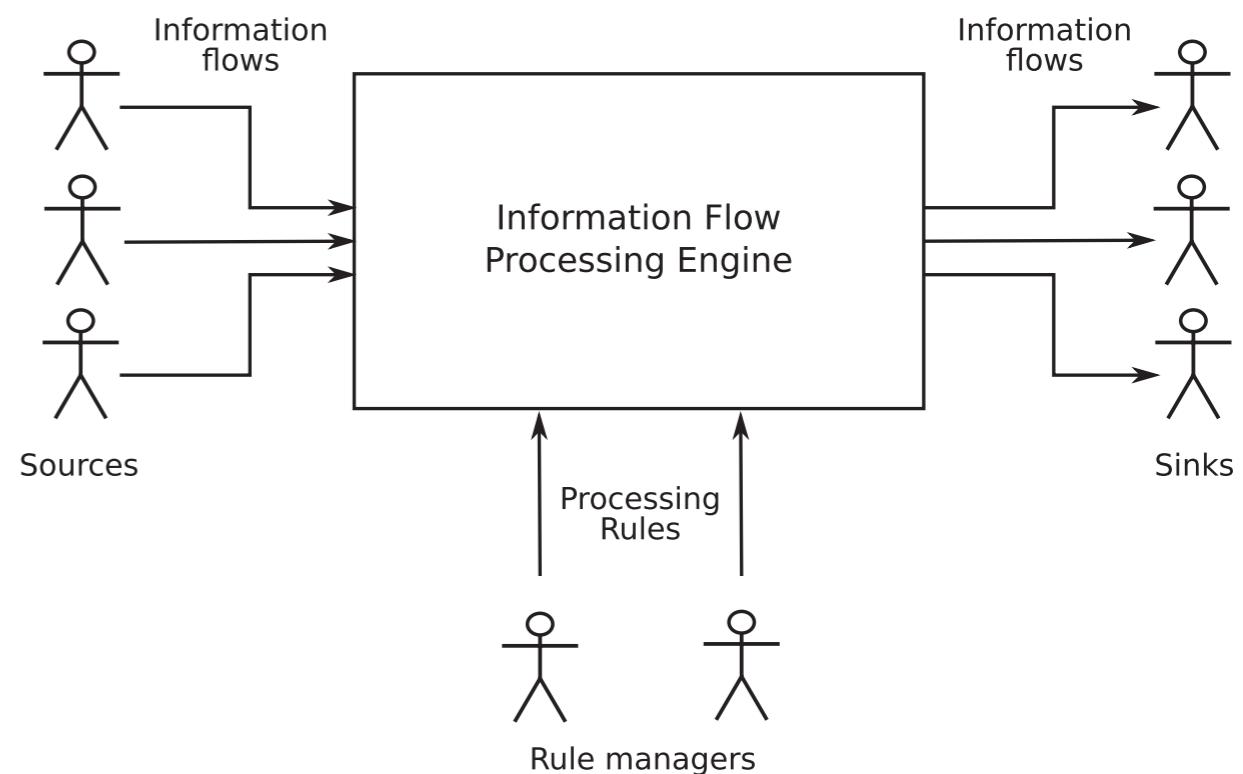
- ◆ the need for **processing information as it flows** from the periphery to the center of the system **without requiring**, at least in principle, the **information to be persistently stored.**
- ◆ Once the flowing data has been processed, thereby producing new information, it can be discarded while the **newly produced information leaves the system as output.**

# Information Flow Processing (IFP) tool



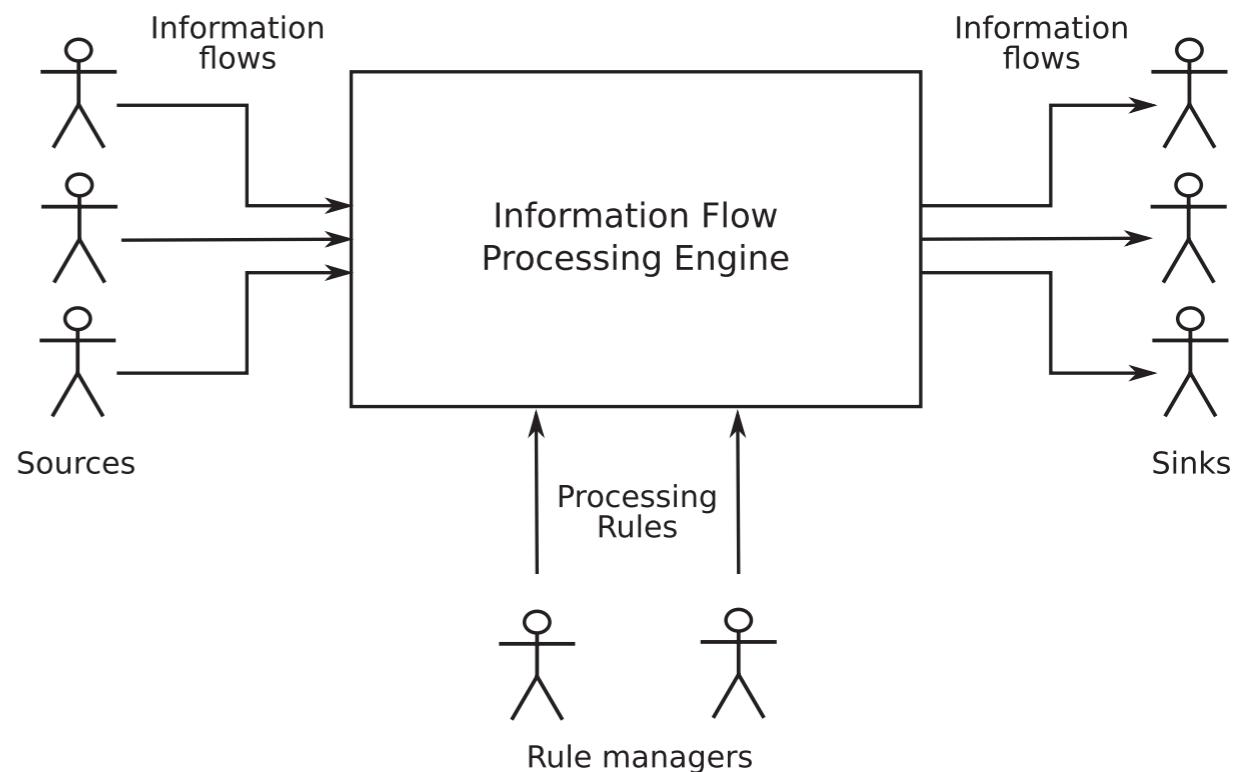
# Information Flow Processing (IFP) tool

- The IFP engine is a tool that **operates according to a set of processing rules** which **describe how incoming flows of information have to be processed to timely produce new flows as outputs**



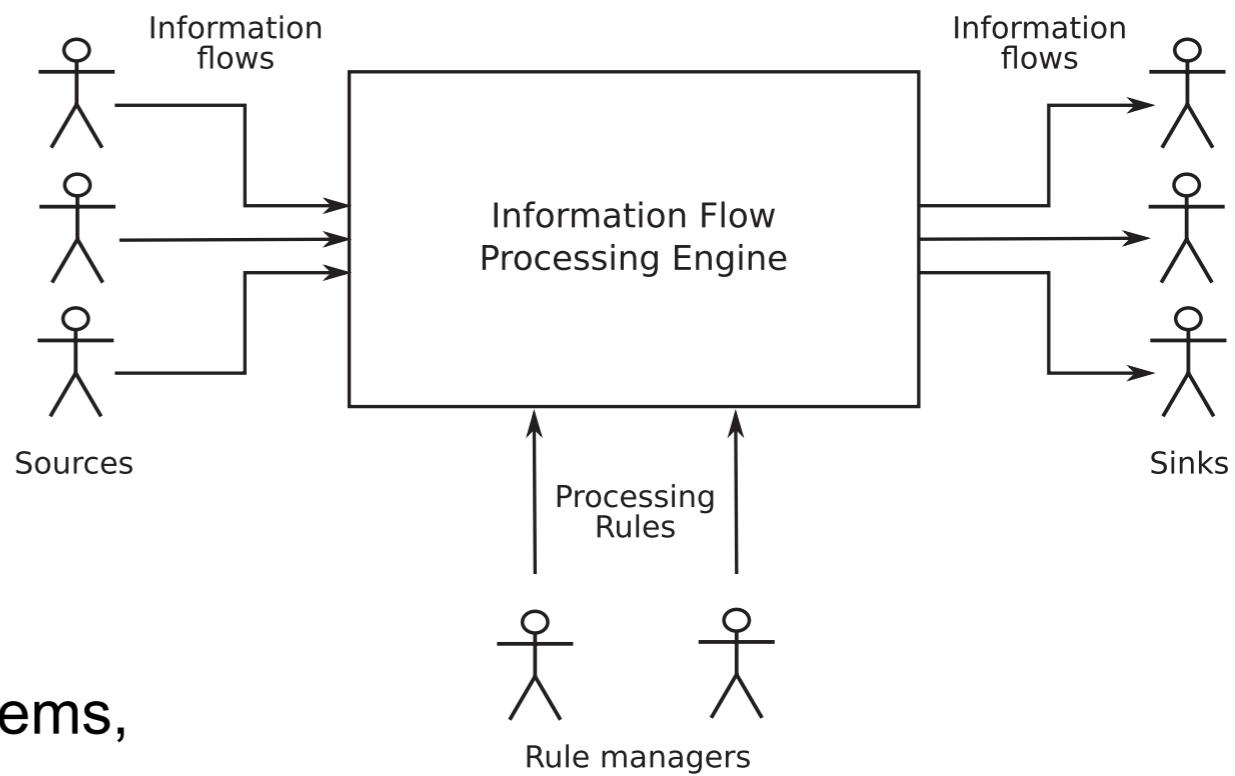
# Information Flow Processing (IFP) tool

- The IFP engine is a tool that **operates according to a set of processing rules** which **describe how incoming flows of information have to be processed to timely produce new flows as outputs**
- Information sources*
- Information items* that are part of the same flow are **neither necessarily ordered nor of the same kind**



# Information Flow Processing (IFP) tool

- The IFP engine is a tool that **operates according to a set of processing rules** which **describe how incoming flows of information have to be processed to timely produce new flows as outputs**
- Information sources*
- Information items* that are part of the same flow are **neither necessarily ordered nor of the same kind**
- The IFP engine processes the information items, **as soon as they are available**, according to a set of processing rules which specify how to **filter, combine, and aggregate** different flows of information item by item to **generate new flows**, which represent the output of the engine



# Information Flow Processing (IFP) tool: concerns

# Information Flow Processing (IFP) tool: concerns

- The IFP tool needs to **perform real-time or quasi real-time processing of incoming information** to produce new knowledge (i.e., outgoing information)

# Information Flow Processing (IFP) tool: concerns

- The IFP tool needs to **perform real-time or quasi real-time processing of incoming information** to produce new knowledge (i.e., outgoing information)
- The IFP tool needs an **expressive language** to describe how incoming information has to be processed, with the ability of **specifying complex relationships among the information items** that flow into the engine and are relevant to sinks

# Information Flow Processing (IFP) tool: concerns

- The IFP tool needs to **perform real-time or quasi real-time processing of incoming information** to produce new knowledge (i.e., outgoing information)
- The IFP tool needs an **expressive language** to describe how incoming information has to be processed, with the ability of **specifying complex relationships among the information items** that flow into the engine and are relevant to sinks
- The IFP tool needs **scalability** to effectively cope with situations in which a **very large number of geographically distributed information sources and sinks** have to **cooperate**.

## Active Database Systems

# IFPs: Active Database Systems

## ■ Traditional DBMS are passive

- ◆ HADP - Human-Active Database-Passive
- ◆ Everything that happens in a database must be explicitly asked by a user/program
- ◆ It is not possible to ask a database to send a notification when specific situations are detected
- ◆ This makes them highly inadequate for IFP

# IFPs: Active Database Systems

- **Traditional DBMS are passive**
  - ◆ HADP - Human-Active Database-Passive
  - ◆ Everything that happens in a database must be explicitly asked by a user/program
  - ◆ It is not possible to ask a database to send a notification when specific situations are detected
  - ◆ This makes them highly inadequate for IFP
- ◆ **Active Database Systems** have been developed to overcome this limitation: they can be seen as an **extension of classical DBMSs**, where the reactive behavior can be moved (totally or in part) from the application layer into the DBMS.

# IFPs: Active Database Systems

- The **knowledge model** - active rules as composed of three parts. (**ECA** rules)

# IFPs: Active Database Systems

- The **knowledge model** - active rules as composed of three parts. (**ECA** rules)
  - ◆ **Event.** The event part defines which sources can be considered as event generators: **some systems only consider internal operators** (like a tuple insertion or update), while **others also allow external events**, like those raised by clocks or external sensors.

# IFPs: Active Database Systems

- The **knowledge model** - active rules as composed of three parts. (**ECA** rules)
  - ◆ **Event.** The event part defines which sources can be considered as event generators: **some systems only consider internal operators** (like a tuple insertion or update), while **others also allow external events**, like those raised by clocks or external sensors.
  - ◆ **Condition.** The condition part specifies when an event must be taken into account; for example, we can be interested in some data only if it exceeds a predefined limit.

# IFPs: Active Database Systems

- The **knowledge model** - active rules as composed of three parts. (**ECA** rules)
  - ◆ **Event.** The event part defines which sources can be considered as event generators: **some systems only consider internal operators** (like a tuple insertion or update), while **others also allow external events**, like those raised by clocks or external sensors.
  - ◆ **Condition.** The condition part specifies when an event must be taken into account; for example, we can be interested in some data only if it exceeds a predefined limit.
  - ◆ **Action.** The action part identifies the set of tasks that should be executed as a response to an event detection: **some systems only allow the modification of the internal database**, while others **allow the application to be notified about the identified situation.**

# IFPs: Active Database Systems

- The **execution model** defines how rules are processed at runtime. Five phases have been identified.

# IFPs: Active Database Systems

- The **execution model** defines how rules are processed at runtime. Five phases have been identified.
  - ◆ **Signaling.** Detection of an event.

# IFPs: Active Database Systems

- The **execution model** defines how rules are processed at runtime. Five phases have been identified.
  - ◆ **Signaling.** Detection of an event.
  - ◆ **Triggering.** Association of an event with the set of rules defined for it.

# IFPs: Active Database Systems

- The **execution model** defines how rules are processed at runtime. Five phases have been identified.
  - ◆ **Signaling.** Detection of an event.
  - ◆ **Triggering.** Association of an event with the set of rules defined for it.
  - ◆ **Evaluation.** Evaluation of the conditional part for each triggered rule.

# IFPs: Active Database Systems

- The **execution model** defines how rules are processed at runtime. Five phases have been identified.
  - ◆ **Signaling.** Detection of an event.
  - ◆ **Triggering.** Association of an event with the set of rules defined for it.
  - ◆ **Evaluation.** Evaluation of the conditional part for each triggered rule.
  - ◆ **Scheduling.** Definition of an execution order between selected rules.

# IFPs: Active Database Systems

- The **execution model** defines how rules are processed at runtime. Five phases have been identified.
  - ◆ **Signaling.** Detection of an event.
  - ◆ **Triggering.** Association of an event with the set of rules defined for it.
  - ◆ **Evaluation.** Evaluation of the conditional part for each triggered rule.
  - ◆ **Scheduling.** Definition of an execution order between selected rules.
  - ◆ **Execution.** Execution of all the actions associated to selected rules.

# IFPs: Active Database Systems

- Active database systems are used in three contexts:
  - ◆ As a **database extension**: active rules refer **only to the internal state of the database**, for example, to implement an automatic reaction to constraint violations.
  - ◆ In **closed database applications**: active rules can support the semantics of the application, but **external sources of events are not allowed**.
  - ◆ In **open database applications**: **events may come both from inside the database and from external sources**. This is the domain that is closer to IFP.
- Triggers:
  - ◆ They can be used solely to impose some (tricky) integrity conditions
  - ◆ But, with them (some) reactive behaviour may be moved to the DBMS, rather than staying at the level of the user/program

# More (optional) information about Triggers

- Modified example from "Database System Concepts", Silberschatz, Korth and Sudarshan

# Exemplo de Trigger

- Imagine uma situação em que o banco aceita que haja saldos negativos e, nesses casos:
  - coloca o saldo a 0
  - cria um empréstimo com o valor em dívida
  - Atribui a este empréstimo um número idêntico ao da conta de depósito
- O trigger deve ser executado sempre que há uma actualização na relação *account* que faz com que o saldo passe a negativo.

# Codificação do Exemplo em SQL:1999

```
create trigger overdraft_trigger after update on account
referencing new row as nrow
for each row
when nrow.balance < 0
begin atomic
    insert into borrower
        (select customer_name, account_number
         from depositor
         where nrow.account_number =
               depositor.account_number);
    insert into loan values
        (nrow.account_number, nrow.branch_name,
         - nrow.balance);
    update account set balance = 0
        where account.account_number = nrow.account_number
end
```

# Eventos e Acções de Triggers em SQL

- Os eventos que podem fazer disparar um trigger são **insert**, **delete** ou **update**
- No Oracle, também podem disparar triggers eventos de **servererror**, **logon**, **logoff**, **startup** e **shutdown**.
- Triggers sobre **update** podem-se restringir só a alguns atributos
  - E.g. **create trigger overdraft\_trigger after update of balance on account**
- Pode-se referenciar o valor dos atributos antes e depois da modificação
  - **referencing old row as** : para **deletes** e **updates**
  - **referencing new row as** : para **inserts** e **updates**
- Pode-se fazer disparar um trigger antes do evento, para codificar restrições.  
E.g. converter espaços em **null**.  
  
**create trigger setnull\_trigger before update on r  
referencing new row as nrow  
for each row  
when nrow.phone\_number = ''  
set nrow.phone\_number = null**
- Para além do **before** e do **after** no Oracle existe também o **instead of**.

# Sintaxe de Triggers em Oracle

```
create [or replace] trigger <nome_trigger>
{before | after | instead of} <evento>
[referencing old as <nome_anter>]
[referencing new as <nome_depois>]
for each row
when <condição>
begin
<Sequencia de comandos, terminados por ;>
end;
/
```

- *Evento* pode ser:

```
delete on <tabela ou view>
insert on <tabela ou view>
update on <tabela ou view>
update of <atributos separados por ,> on <tabela ou view>
servererror, logon, logoff, startup ou shutdown
```

- Os comandos são PL/SQL o que inclui os comandos SQL, mais WHILEs, IFs, etc (ver manuais)
- Dentro da condição os *nome\_anter* e *nome\_depois* podem ser usados sem mais. Mas nos comandos têm que ter o símbolo ‘:’ antes!!!

# Statement Triggers

- São executados após (antes, ou em vez de) uma instrução completa vs. os anteriores que são executadas após alterações em cada linha

- Sintaxe:

```
create [or replace] trigger <nome_trigger>
{before | after | instead of} <evento>
begin
    <Sequencia de comandos, terminados por ;>
end;
```

- Para ser usado quando as condições são para testar globalmente e não linha a linha.

# Uso de triggers

- Podem usar-se para implementar assertions, fazendo `raise_application_error` quando as condições não se verificam.
- Não usar triggers:
  - Quando as restrições podem ser impostas doutra forma (com a excepção das asserções)!!
    - Os triggers são mais difíceis de manter e são menos eficientes.
  - Quando se querem manter sumários
    - Para tal usem-se views e se eficiência for importante usem-se materialized views
- Os triggers permitem uma grande generalidade na imposição de restrições e, também por isso mesmo, devem ser usados com grande cuidado.

# Exemplo

```
create table alunos( num_aluno number(6) not null, ...,
                      cod_curso number(3) not null,
primary key (num_aluno),
unique (num_aluno, cod_curso)
foreign key cod_curso references curso);
```

```
create table curso_cadeira(
                      cod_curso number(3) not null,
                      cod_cadeira number(3) not null, ...,
primary key (cod_curso, cod_cadeira), ...);
```

```
create table inscricoes( num_aluno number(6) not null,
                           cod_curso number(3) not null,
                           cod_cadeira number(5) not null,
                           data_inscricao date not null, ...,
primary key (num_aluno, cod_curso, cod_cadeira, data_inscricao),
foreign key (num_aluno, cod_curso)
                           references alunos(num_aluno, cod_curso),
foreign key (cod_curso, cod_cadeira) references curso_cadeira);
```

# Triggers para actualização de vistas

- Podemos utilizar triggers para efectuar modificações através de vistas.
- Para tal, criamos triggers para todas as operações permitidas, como por exemplo:
  - para a inserção (do tipo instead of insert on),
  - para a remoção (do tipo instead of delete on)
  - para a actualização (do tipo instead of update on).
- Consideremos a vista:
- **`create view info_empréstimos as  
select loan_number, customer_name, amount  
from borrower natural inner join loan`**

# Triggers para actualização de vistas

- Se quisermos permitir a remoção de empréstimos através da vista, criamos o trigger:

```
create trigger remove_empréstimos instead of delete
  on info_empréstimos referencing old row as orow
  for each row
    begin
      delete from loan where loan_number = orow.loan_number ;
      delete from borrower
        where loan_number = orow.loan_number ;
    end
```

# Triggers para actualização de vistas

- Se quisermos permitir a inserção de empréstimos através da vista, criamos o trigger:

```
create trigger insere_empréstimos
instead of insert on info_empréstimos
referencing new row as nrow
for each row
begin
    insert into loan
        values      (nrow.loan_number, NULL, amount);
    insert into borrower
        values      (nrow.customer_name, nrow.loan_number)
end
```

# Triggers para actualização de vistas

- Se quisermos permitir a actualização do valor do empréstimo através da vista, criamos o trigger:

```
create trigger actualiza_empréstimos
instead of update of amount on info_empréstimos
referencing new row as nrow
referencing old row as orow
for each row
begin
    update loan
    set amount = nrow.amount
        where loan_number = orow.loan_number;
end
```

# Triggers para inserção de chaves

- Se quisermos preencher automaticamente a chave de um tuplo, aquando da sua inserção, recorrendo a uma sequência:

```
create trigger chave_aluno  
before insert on alunos  
for each row  
declare  
    aluno_id number;  
begin  
    select seq_aluno.nextval  
    into aluno_id  
    from dual;  
    :new.num_aluno := aluno_id;  
end
```

# Acções Externas

- Por vezes podemos querer que um dado evento faça disparar uma acção para o exterior.
  - Por exemplo, numa base de dados de uma armazém, sempre que a quantidade de um produto desce abaixo (devido a um **update**) de um determinado valor podemos querer encomendar esse produto, ou disparar algum alarme.
- Os triggers não podem ser usados para implementar acções sobre o exterior, mas...
  - podem ser usados para guardar numa tabela separada acções-a-levar-a-cabo. Podem depois haver procedimentos que, periodicamente verificam essa tabela separada.
- E.g. Uma base de um armazém com as tabelas
  - inventario(item, quant)*: Que quantidade há de cada produto
  - quantMin(item, quant)* : Qual a quantidade mínima de cada produto
  - reposicoes(item, quant)*: Quanto encomendar sempre que está em falta
  - aencomendar(item, quant)* : Coisas a encomendar (lido por procedimento)

# Exemplo de Acções Externas

```
create trigger aenc_trigger after update of quant on inventario
referencing old row as orow, new row as nrow
for each row
    when nrow.quant <= some (select quant
                                from quantMin
                                where quantMin.item = orow.item)
        and orow.quant > some (select quant
                                from quantMin
                                where quantMin.item = orow.item)
begin
    insert into aencomendar
        (select item, quant
         from reposicoes
         where reposicoes.item = orow.item)
end
```

# IFPs: Active Database Systems

- Triggers are ok:
  - ◆ as a kind of hack, for imposing directly in the database (rather than by applications) integrity conditions not otherwise expressible in SQL
  - ◆ to support reactive behaviours in closed databases, where the result of the behaviour is always reflected in the database

# IFPs: Active Database Systems

- Triggers are ok:
  - ◆ as a kind of hack, for imposing directly in the database (rather than by applications) integrity conditions not otherwise expressible in SQL
  - ◆ to support reactive behaviours in closed databases, where the result of the behaviour is always reflected in the database
- **But for IFP applications**, with sources and sinks, it is simply **not good enough**
  - ◆ To re-execute the SQL queries over and over again is not feasible
  - ◆ The procedures that have to look at the tables and act are outside the semantics of the system

## Data Stream Management Systems

# Data Stream Management Systems

- Even when taking into account external event sources, **active database systems** (like traditional DBMS), are **built around a persistent storage** where all the relevant data is kept and whose updates are relatively infrequent.

# Data Stream Management Systems

- Even when taking into account external event sources, **active database systems** (like traditional DBMS), are **built around a persistent storage** where all the relevant data is kept and whose updates are relatively infrequent.
- Database community developed a new class of systems oriented toward **processing large streams of data in a timely way**: **data stream management systems** (DSMSs).

# Data Stream Management Systems

- Even when taking into account external event sources, **active database systems** (like traditional DBMS), are **built around a persistent storage** where all the relevant data is kept and whose updates are relatively infrequent.
- Database community developed a new class of systems oriented toward **processing large streams of data in a timely way**: **data stream management systems** (DSMSs).
- DSMSs differ from conventional DBMSs in several ways.

# Data Stream Management Systems

- Even when taking into account external event sources, **active database systems** (like traditional DBMS), are **built around a persistent storage** where all the relevant data is kept and whose updates are relatively infrequent.
- Database community developed a new class of systems oriented toward **processing large streams of data in a timely way**: **data stream management systems** (DSMSs).
- DSMSs differ from conventional DBMSs in several ways.
  - ◆ **Streams** are usually **unbounded**.

# Data Stream Management Systems

- Even when taking into account external event sources, **active database systems** (like traditional DBMS), are **built around a persistent storage** where all the relevant data is kept and whose updates are relatively infrequent.
- Database community developed a new class of systems oriented toward **processing large streams of data in a timely way**: **data stream management systems** (DSMSs).
- DSMSs differ from conventional DBMSs in several ways.
  - ◆ **Streams** are usually **unbounded**.
  - ◆ **No assumption** can be made on **data arrival order**.

# Data Stream Management Systems

- Even when taking into account external event sources, **active database systems** (like traditional DBMS), are **built around a persistent storage** where all the relevant data is kept and whose updates are relatively infrequent.
- Database community developed a new class of systems oriented toward **processing large streams of data in a timely way**: **data stream management systems** (DSMSs).
- DSMSs differ from conventional DBMSs in several ways.
  - ◆ **Streams** are usually **unbounded**.
  - ◆ **No assumption** can be made on **data arrival order**.
  - ◆ Size and time constraints make it **difficult to store and process data stream elements** after their arrival; **one-time processing is the typical mechanism** used to deal with streams.

# Data Stream Management Systems

- Users of a **DSMS** install **standing** (or **continuous**) **queries**, that is, queries that are **deployed once and continue to produce results until removed**.

# Data Stream Management Systems

- Users of a **DSMS** install **standing** (or **continuous**) **queries**, that is, queries that are **deployed once and continue to produce results until removed**.
- Standing queries can be executed **periodically or continuously** as new stream items arrive.
  - ◆ **Users do not have to explicitly ask for updated information;**
  - ◆ **The system actively notifies it according to installed queries.**
  - ◆ This form of interaction is usually called database-active human-passive (**DAHP**)

# Data Stream Management Systems

- Users of a **DSMS** install **standing** (or **continuous**) **queries**, that is, queries that are **deployed once and continue to produce results until removed**.
- Standing queries can be executed **periodically or continuously** as new stream items arrive.
  - ◆ **Users do not have to explicitly ask for updated information;**
  - ◆ **The system actively notifies it according to installed queries.**
  - ◆ This form of interaction is usually called database-active human-passive (**DAHP**)
- **DSMS** is viewed as one that, given a set of input streams and some continuous queries, **continuously output streams !**
  - ◆ Queries can be viewed as **algebraic expressions operating on streams** (as queries in DBMS are algebraic expressions on relations)

# (A side note on persistent queries)

- Persistent (and continuous) queries already existed in DBMS for a long time.

Do you remember where?

# (A side note on persistent queries)

- **Persistent (and continuous) queries already existed in DBMS for a long time.**

**Do you remember where?**

- **In materialized views!**

- ◆ The query (the view) is always present in the database, and must be run continuously in order to keep the materialized view updated
- ◆ But this was thought for sporadic updates, and in practice materialized views usually are not updated “continuously”, but, rather, periodically

# Data Stream Management Systems

- Several implementations were proposed for DSMSs. They differ in the semantics they associate to standing queries.

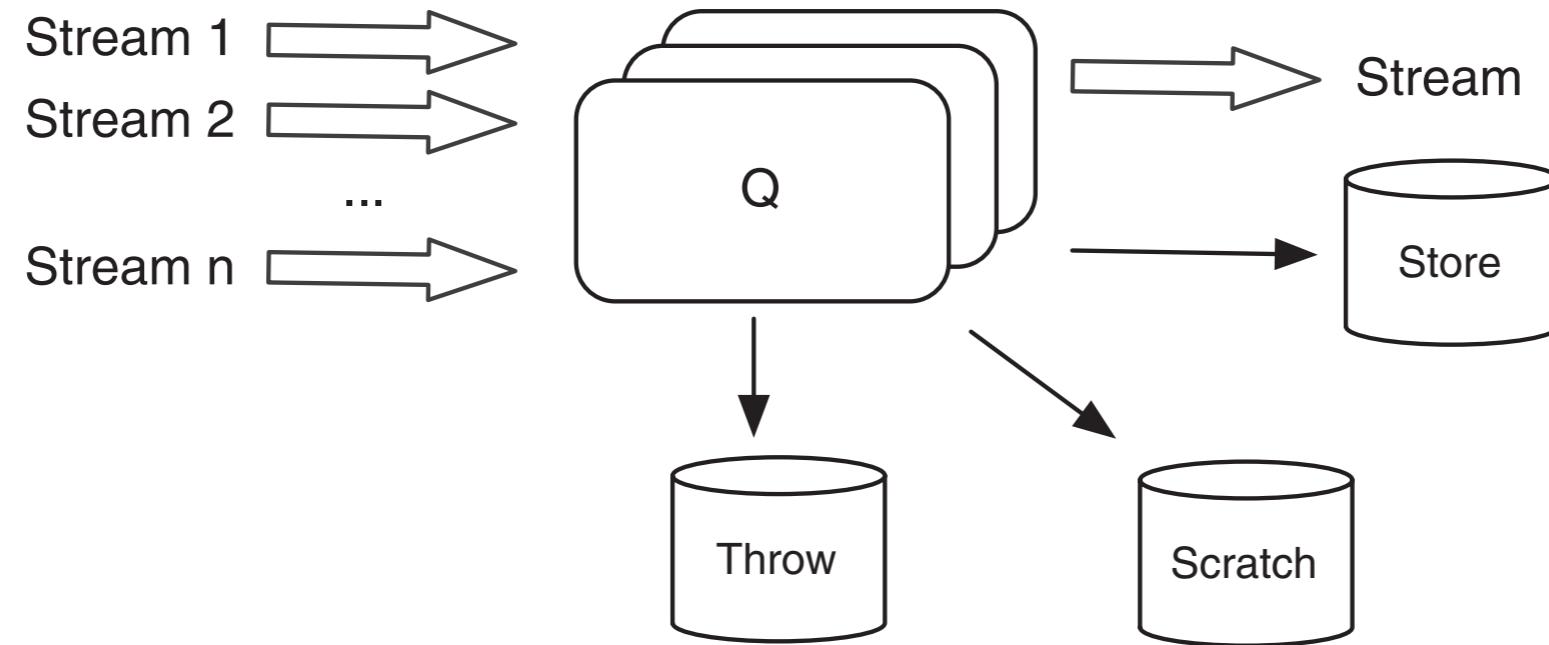
# Data Stream Management Systems

- Several implementations were proposed for DSMSs. They differ in the semantics they associate to standing queries.
- The answer to a query can be seen as
  - ◆ an **append-only** output stream **or**;
  - ◆ as an entry in a storage that is **continuously modified** as new elements flow inside the processing stream.

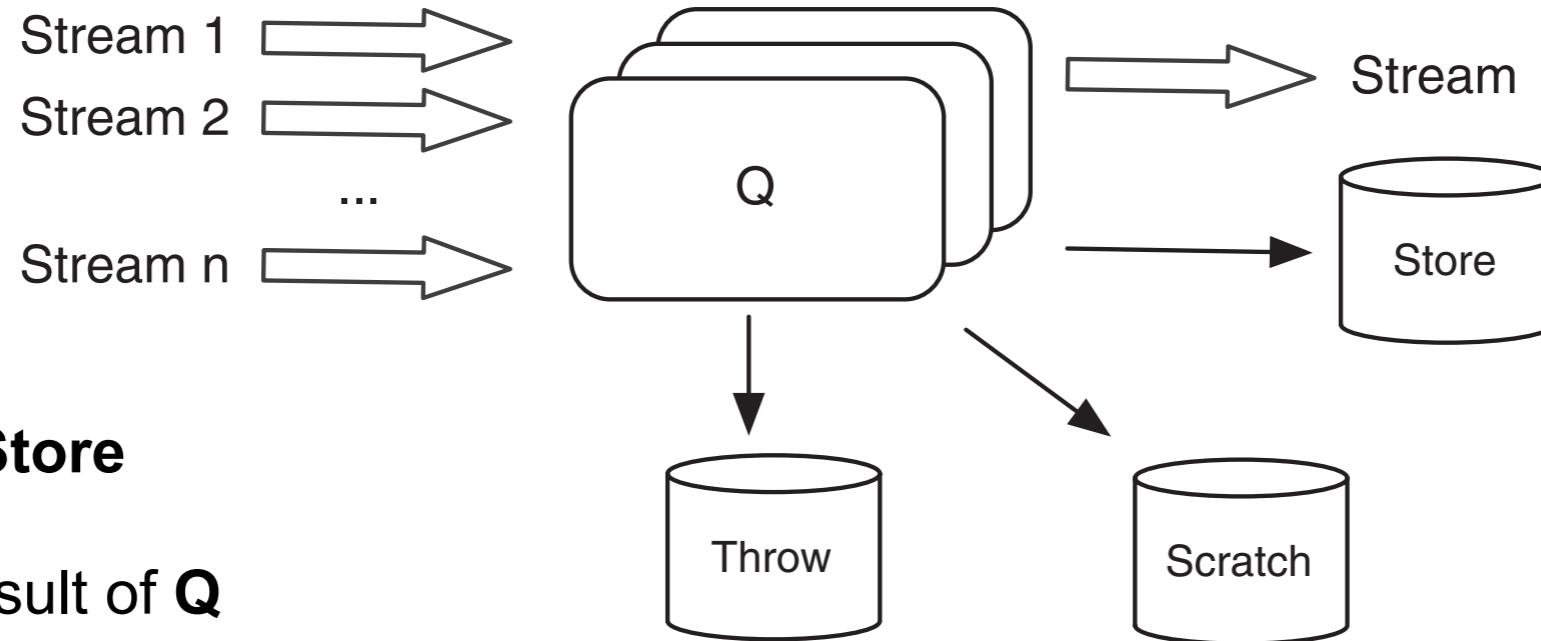
# Data Stream Management Systems

- Several implementations were proposed for DSMSs. They differ in the semantics they associate to standing queries.
- The answer to a query can be seen as
  - ◆ an **append-only** output stream **or**;
  - ◆ as an entry in a storage that is **continuously modified** as new elements flow inside the processing stream.
- An answer can be
  - ◆ **Exact**, if the system is supposed to have enough memory to store all the required elements of input streams' history;
  - ◆ **Approximate**, if computed on a portion of the required history.

# Data Stream Management Systems: Typical Model



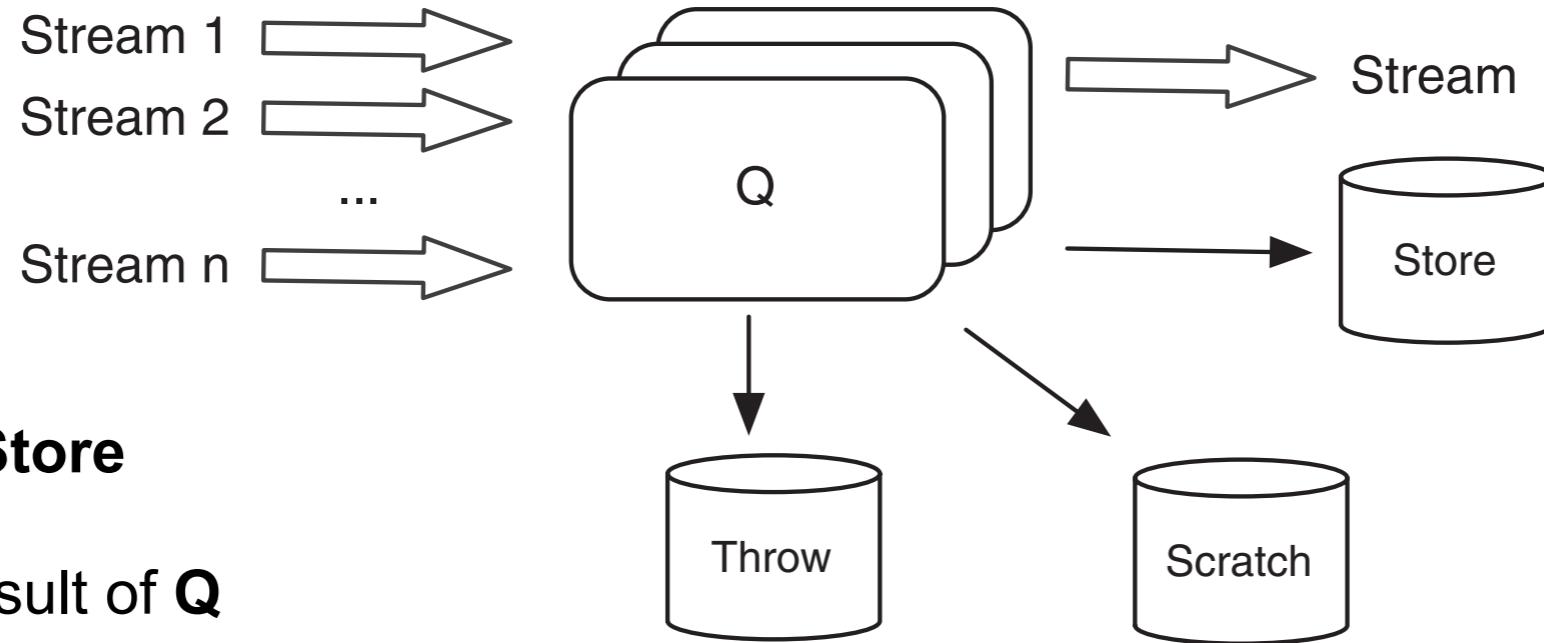
# Data Stream Management Systems: Typical Model



## ■ Stream and Store

- ◆ Are the result of **Q**
- ◆ **Stream**: the output stream, where elements are produced once and never changed
- ◆ **Store**: parts of the answer that may be changed later

# Data Stream Management Systems: Typical Model



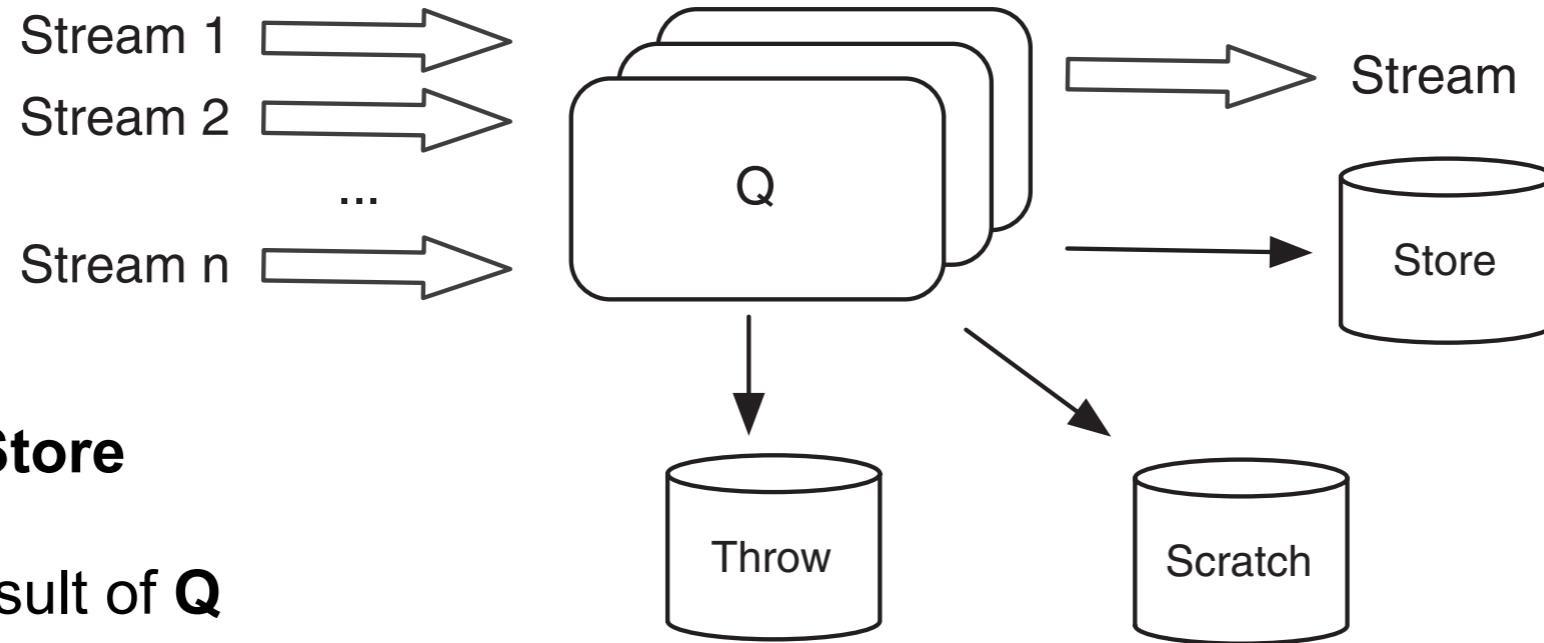
## ■ Stream and Store

- ◆ Are the result of **Q**
- ◆ **Stream**: the output stream, where elements are produced once and never changed
- ◆ **Store**: parts of the answer that may be changed later

## ■ Scratch: working memory of the system

- ◆ a repository to stored data that is not part of the answer, but may be needed (later) for computing it

# Data Stream Management Systems: Typical Model



## ■ Stream and Store

- ◆ Are the result of **Q**
- ◆ **Stream**: the output stream, where elements are produced once and never changed
- ◆ **Store**: parts of the answer that may be changed later

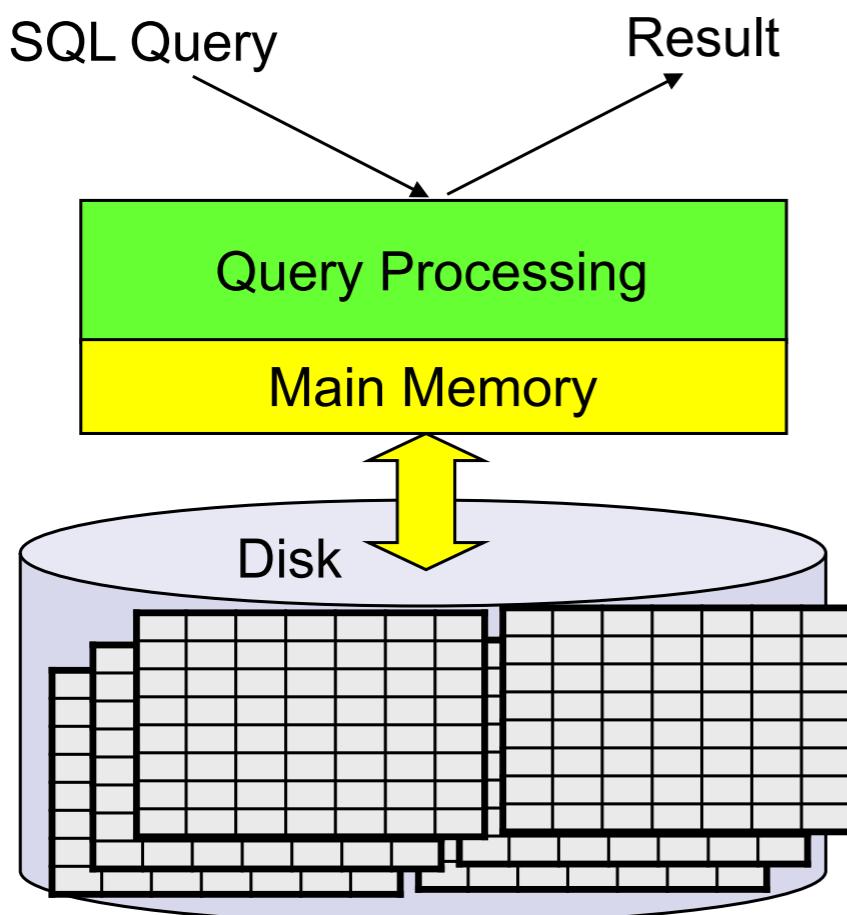
## ■ Scratch: working memory of the system

- ◆ a repository to stored data that is not part of the answer, but may be needed (later) for computing it

## ■ Throw: A kind of recycler bin (not important for now)

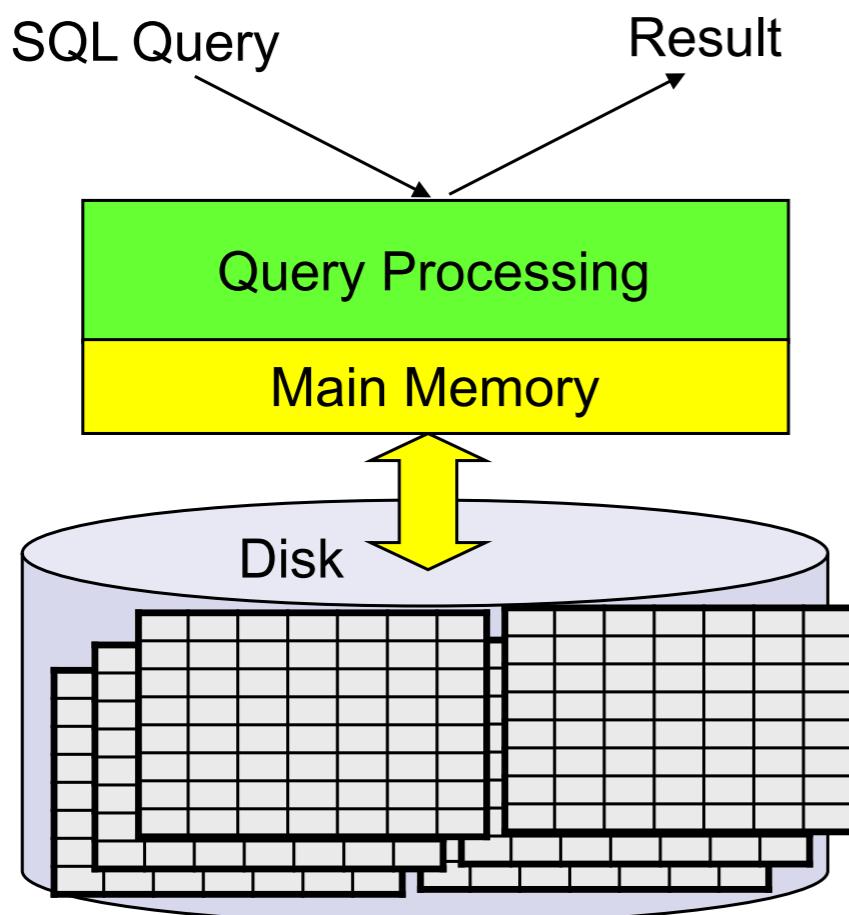
# DBMS versus DSMS

## DBMS



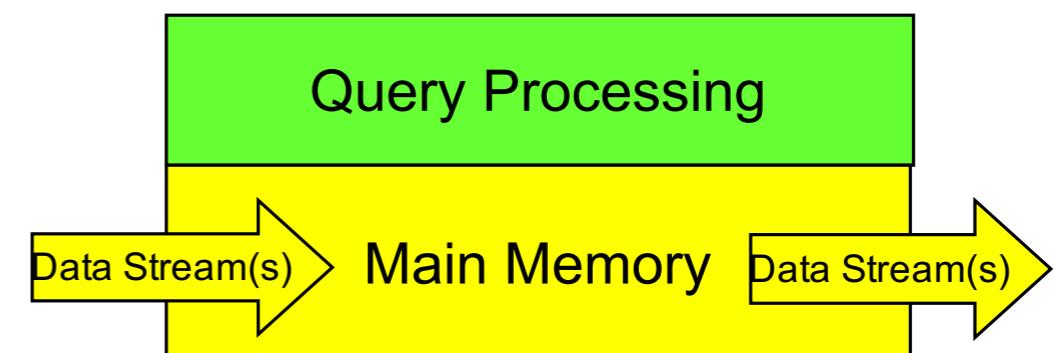
# DBMS versus DSMS

## DBMS



## DSMS

Continuous Query (CQ)      Result



# DBMS versus DSMS

# DBMS versus DSMS

- Persistent data
- Transient data

# DBMS versus DSMS

- Persistent data
- Finite sets of tuples
- Transient data
- Infinite sequence of tuples

# DBMS versus DSMS

- **Persistent data**
- Finite sets of tuples
- Random access
- **Transient data**
- Infinite sequence of tuples
- Sequential access

# DBMS versus DSMS

- **Persistent data**
- Finite sets of tuples
- Random access
- All types of updates
- **Transient data**
- Infinite sequence of tuples
- Sequential access
- Only append updates

# DBMS versus DSMS

- **Persistent data**
- Finite sets of tuples
- Random access
- All types of updates
- **Transient queries**
- **Transient data**
- Infinite sequence of tuples
- Sequential access
- Only append updates
- **Persistent queries**

# DBMS versus DSMS

- **Persistent data**
- Finite sets of tuples
- Random access
- All types of updates
- **Transient queries**
- Multi-pass query evaluation
- **Transient data**
- Infinite sequence of tuples
- Sequential access
- Only append updates
- **Persistent queries**
- One-pass query evaluation

# DBMS versus DSMS

- **Persistent data**
- Finite sets of tuples
- Random access
- All types of updates
- **Transient queries**
- Multi-pass query evaluation
- Fixed query plan
- **Transient data**
- Infinite sequence of tuples
- Sequential access
- Only append updates
- **Persistent queries**
- One-pass query evaluation
- Adaptive query plan

# DBMS versus DSMS

- **Persistent data**
- Finite sets of tuples
- Random access
- All types of updates
- **Transient queries**
- Multi-pass query evaluation
- Fixed query plan
- Allows blocking operator
- **Transient data**
- Infinite sequence of tuples
- Sequential access
- Only append updates
- **Persistent queries**
- One-pass query evaluation
- Adaptive query plan
- No blocking operators

# Relational data streams

- Each data stream consists of relational tuples
  - ◆ exactly as in databases

# Relational data streams

- **Each data stream consists of relational tuples**
  - ◆ exactly as in databases
- **The stream can be modeled as an append-only relation**
  - ◆ but repetitions must be allowed, and
  - ◆ order is very important!

# Relational data streams

- **Each data stream consists of relational tuples**
  - ◆ exactly as in databases
- **The stream can be modeled as an append-only relation**
  - ◆ but repetitions must be allowed, and
  - ◆ order is very important!
- **Order is based on timestamps or arrival order**

# Relational data streams

- **Each data stream consists of relational tuples**
  - ◆ exactly as in databases
- **The stream can be modeled as an append-only relation**
  - ◆ but repetitions must be allowed, and
  - ◆ order is very important!
- **Order is based on timestamps or arrival order**

Detection and notification of **complex patterns of elements involving sequences and ordering relations** are usually **out of the scope of these systems**

# Complex Event Processing Systems

# Streams versus Events

# Streams versus Events

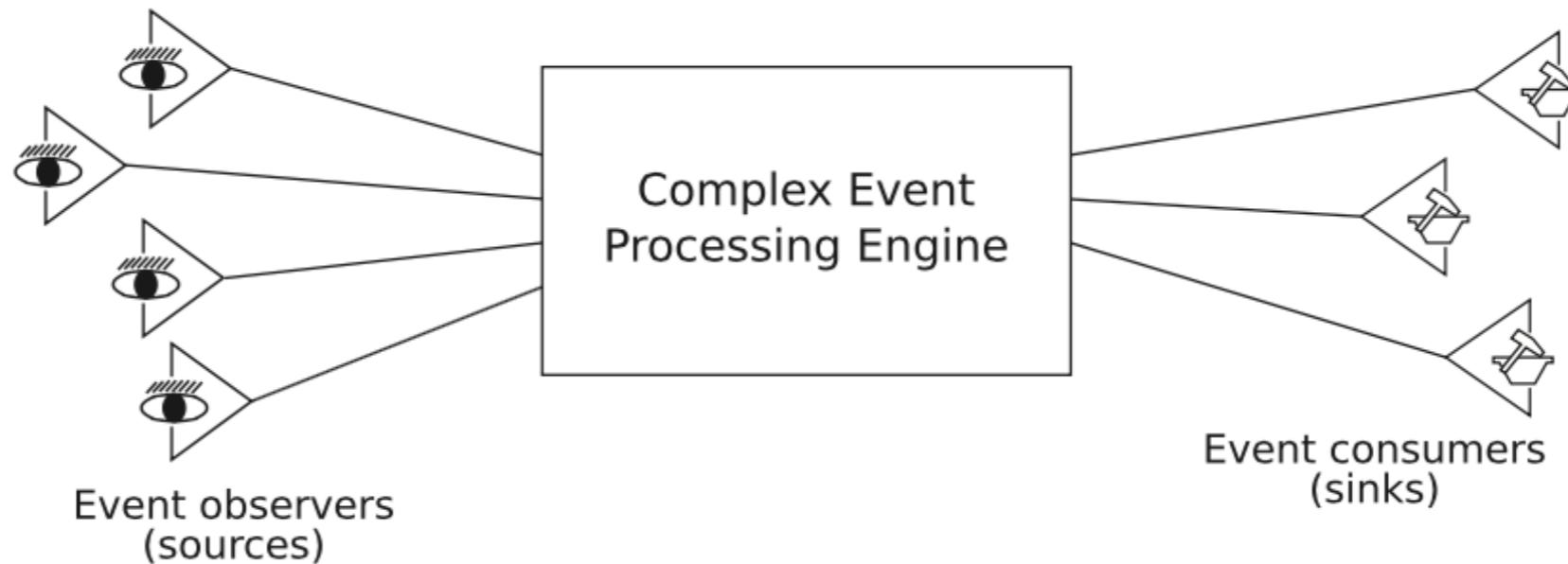
- **DSMS don't assume any meaning of the data in the streams**
  - ◆ The semantics is on the users/programmers shoulders
  - ◆ In fact, data is seen as continuous updates on a schema whose meaning is solely on the database programmer

# Streams versus Events

- **DSMS don't assume any meaning of the data in the streams**
  - ◆ The semantics is on the users/programmers shoulders
  - ◆ In fact, data is seen as continuous updates on a schema whose meaning is solely on the database programmer
- **Complex Event Processor take a very precise meaning of the information that arrives: Events:**
  - ◆ notification of things that happen
  - ◆ they can happen in the external world or in the system itself
  - ◆ events are instantaneous, and cannot be deleted (things don't "unhappen"!)

# Complex Event Processors

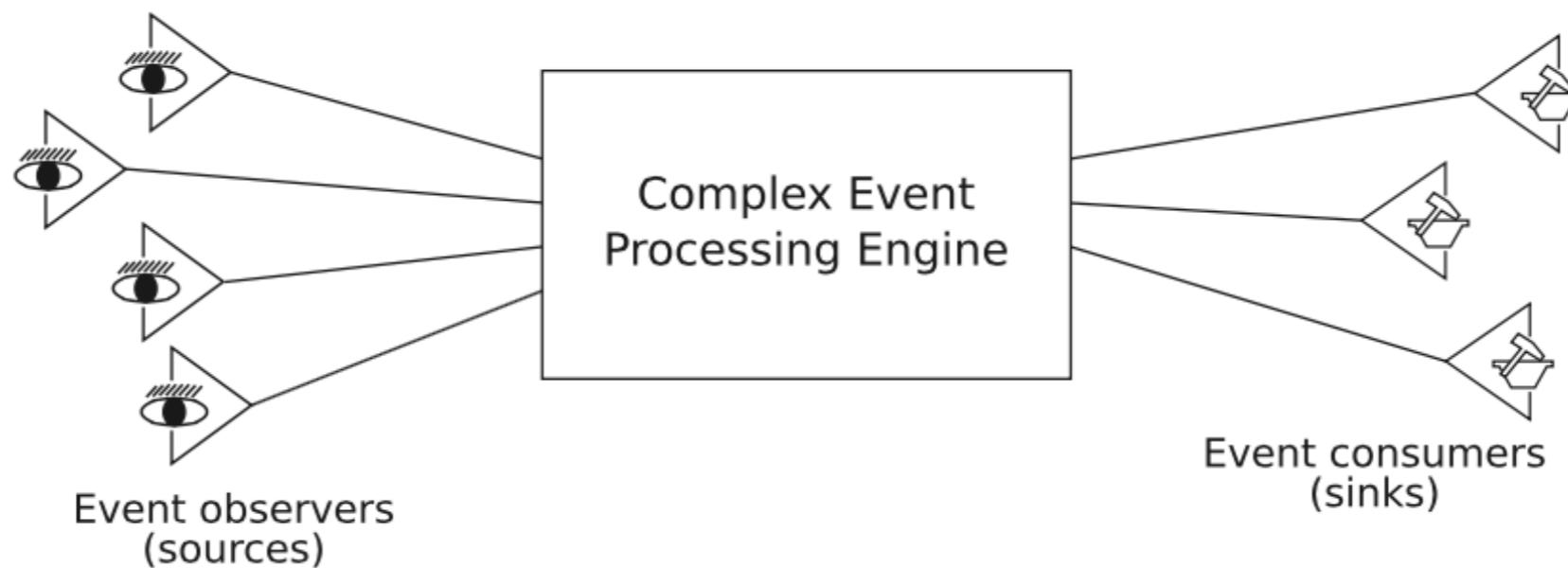
## ■ A CEP engine:



# Complex Event Processors

## ■ A CEP engine:

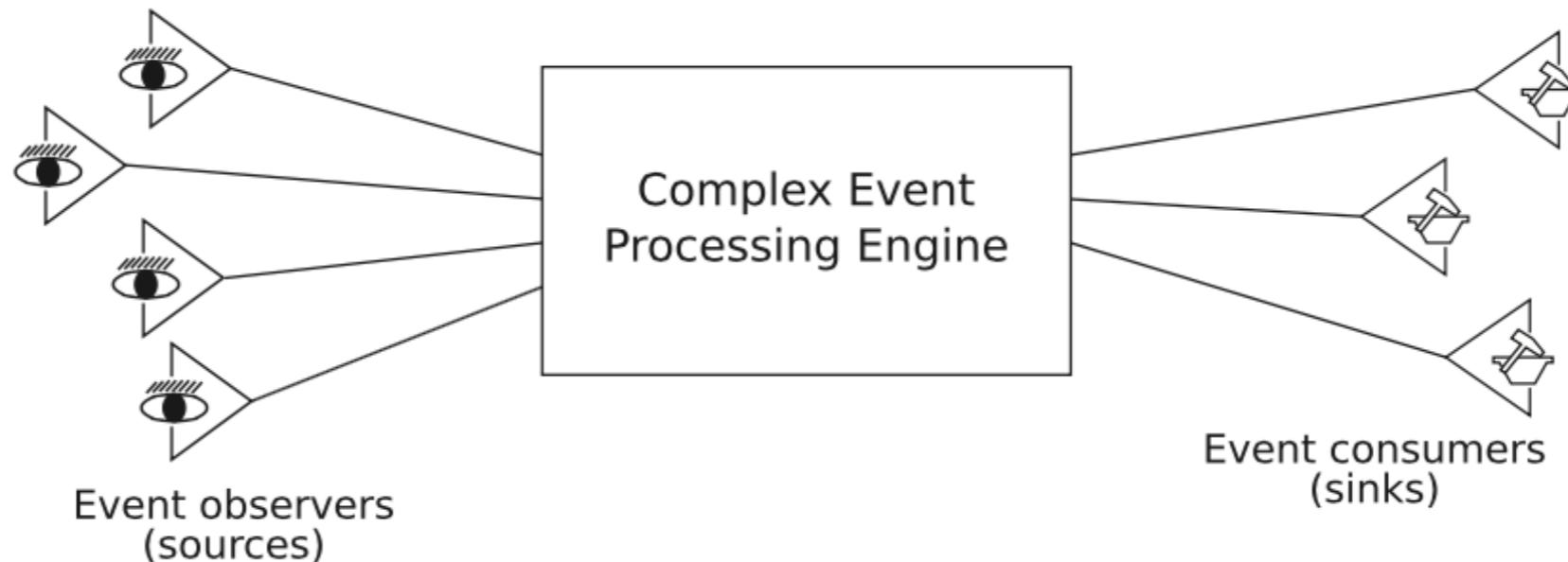
- ◆ sees event notifications that come from **event observers** (or **event sources**)



# Complex Event Processors

## ■ A CEP engine:

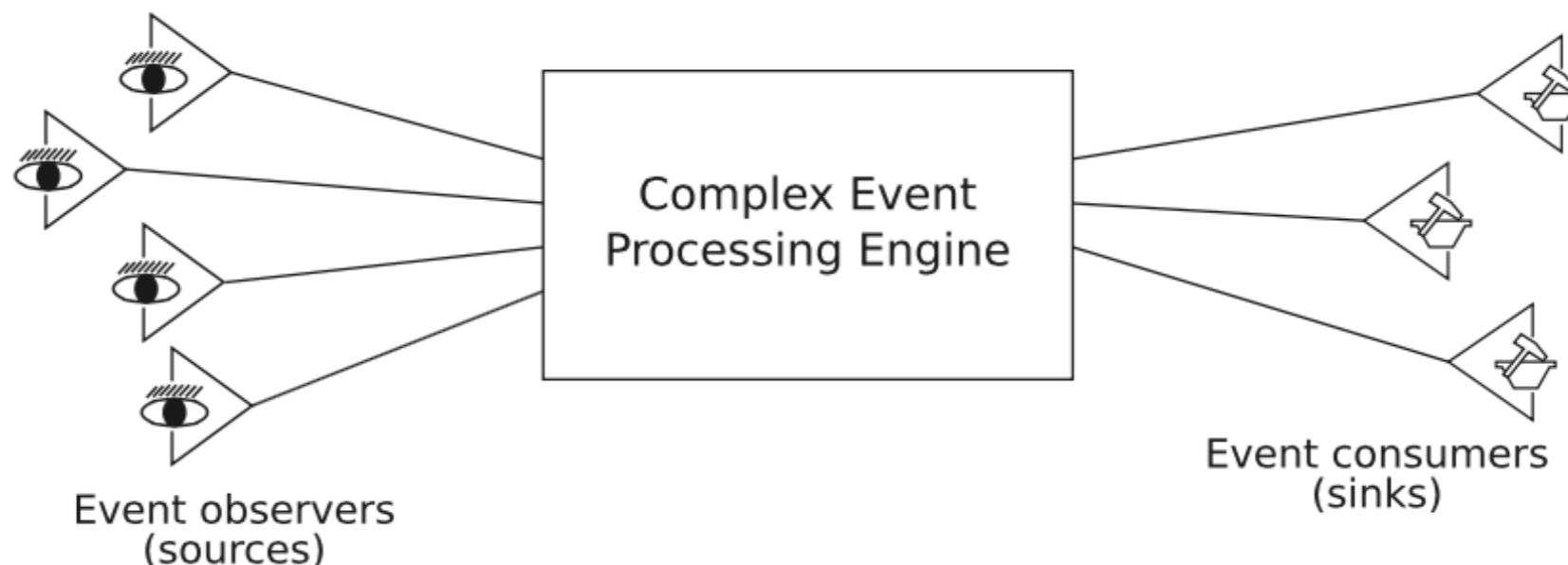
- ◆ sees event notifications that come from **event observers** (or **event sources**)
- ◆ filters and combines the notifications to understand what is happening in terms of higher-level **composite events**



# Complex Event Processors

## ■ A CEP engine:

- ◆ sees event notifications that come from **event observers** (or **event sources**)
- ◆ filters and combines the notifications to understand what is happening in terms of higher-level **composite events**
- ◆ notifies interested clients (**event consumers**) of what is happening in terms of the composite events



# Publish-Subscribe

- CEP has its roots in publish-subscribe messaging patterns

# Publish-Subscribe

- CEP has its roots in publish-subscribe messaging patterns
  - ◆ message senders (**publishers**) don't send messages directly to recipients (**subscribers**) - they don't even (have to) know who the recipients are

# Publish-Subscribe

## ■ CEP has its roots in publish-subscribe messaging patterns

- ◆ message senders (**publishers**) don't send messages directly to recipients (**subscribers**) - they don't even (have to) know who the recipients are
- ◆ instead they send them to a **system in the middle**, and somehow describe what the messages are about - to which classes they belong to

# Publish-Subscribe

## ■ CEP has its roots in publish-subscribe messaging patterns

- ◆ message senders (**publishers**) don't send messages directly to recipients (**subscribers**) - they don't even (have to) know who the recipients are
- ◆ instead they send them to a **system in the middle**, and somehow describe what the messages are about - to which classes they belong to
- ◆ users **subscribe** to classes of event

# Publish-Subscribe

## ■ CEP has its roots in publish-subscribe messaging patterns

- ◆ message senders (**publishers**) don't send messages directly to recipients (**subscribers**) - they don't even (have to) know who the recipients are
- ◆ instead they send them to a **system in the middle**, and somehow describe what the messages are about - to which classes they belong to
- ◆ users **subscribe** to classes of event
- ◆ the **system** is responsible for **receiving** the messages from the publishers, grouping them by classes, and **delivering the relevant** messages to the **relevant subscribers**

# Publish-Subscribe

- CEP has its roots in publish-subscribe messaging patterns
  - ◆ message senders (**publishers**) don't send messages directly to recipients (**subscribers**) - they don't even (have to) know who the recipients are
  - ◆ instead they send them to a **system in the middle**, and somehow describe what the messages are about - to which classes they belong to
  - ◆ users **subscribe** to classes of event
  - ◆ the **system** is responsible for **receiving** the messages from the publishers, grouping them by classes, and **delivering the relevant** messages to the **relevant subscribers**
- Two flavors publish-subscribe: topic and content-based
  - ◆ **Topic-based systems** allow sinks to subscribe only to predefined topics.
  - ◆ **Content-based systems** allow subscribers to use complex event filters.

# Composite Events

- **Content-based** publish-subscribe system allow **subscribers to use complex event filters**, based on event content for specifying what they are interested in

# Composite Events

- Content-based publish-subscribe system allow **subscribers to use complex event filters**, based on event content for specifying what they are interested in
- CEP systems extend this by allowing the filters to depend on the history and on relations between received events

# Composite Events

- Content-based publish-subscribe system allow **subscribers to use complex event filters**, based on event content for specifying what they are interested in
- CEP systems extend this by allowing the filters to depend on the history and on relations between received events
  - ◆ Composite events are events derived from the received ones, based on the content, and on the relation to other events already received

# Composite Events

- Content-based publish-subscribe system allow **subscribers to use complex event filters**, based on event content for specifying what they are interested in
- CEP systems extend this by allowing the filters to depend on the history and on relations between received events
  - ◆ Composite events are events derived from the received ones, based on the content, and on the relation to other events already received
    - E.g. a fire is detected (**composite derived event**) in case three different sensors located in an area smaller than 100m<sup>2</sup> report a temperature higher than 60°C, within 10s from each other

# Complex Events Algebras

# Complex Events Algebras

- Operators defining (derived) events from (basic) events

# Complex Events Algebras

- Operators defining (derived) events from (basic) events
- E.g. (some of) the SNOOP operators
  - ◆ Sequence:  $e_1;e_2$  - occurs when  $e_2$  occurs after  $e_1$  occurred some time before

# Complex Events Algebras

- Operators defining (derived) events from (basic) events
- E.g. (some of) the SNOOP operators
  - ◆ **Sequence:**  $e_1;e_2$  - occurs when  $e_2$  occurs after  $e_1$  occurred some time before
  - ◆ **Conjunction:**  $e_1 \wedge e_2$  - occurs when  $e_1$  and  $e_2$  occurs simultaneously

# Complex Events Algebras

- Operators defining (derived) events from (basic) events
- E.g. (some of) the SNOOP operators
  - ◆ **Sequence:**  $e_1;e_2$  - occurs when  $e_2$  occurs after  $e_1$  occurred some time before
  - ◆ **Conjunction:**  $e_1 \wedge e_2$  - occurs when  $e_1$  and  $e_2$  occurs simultaneously
  - ◆ **Disjunction:**  $e_1 \vee e_2$  - occurs when either  $e_1$  or  $e_2$  occurs

# Complex Events Algebras

- Operators defining (derived) events from (basic) events
- E.g. (some of) the SNOOP operators
  - ◆ **Sequence:**  $e_1;e_2$  - occurs when  $e_2$  occurs after  $e_1$  occurred some time before
  - ◆ **Conjunction:**  $e_1 \wedge e_2$  - occurs when  $e_1$  and  $e_2$  occurs simultaneously
  - ◆ **Disjunction:**  $e_1 \vee e_2$  - occurs when either  $e_1$  or  $e_2$  occurs
  - ◆ **Negation:**  $\text{Not}(e_1, e_2, e_3)$  - occurs when  $e_3$  occurs,  $e_1$  occurred before, and  $e_2$  has not occurred in between

# Complex Events Algebras

- Operators defining (derived) events from (basic) events
- E.g. (some of) the SNOOP operators
  - ◆ **Sequence:**  $e_1;e_2$  - occurs when  $e_2$  occurs after  $e_1$  occurred some time before
  - ◆ **Conjunction:**  $e_1 \wedge e_2$  - occurs when  $e_1$  and  $e_2$  occurs simultaneously
  - ◆ **Disjunction:**  $e_1 \vee e_2$  - occurs when either  $e_1$  or  $e_2$  occurs
  - ◆ **Negation:**  $\text{Not}(e_1, e_2, e_3)$  - occurs when  $e_3$  occurs,  $e_1$  occurred before, and  $e_2$  has not occurred in between
  - ◆ **Quantification:**  $\text{Any}(n, e)$  - occurs whenever event  $e$  occurs  $n$  times
  - ◆ ...

# Complex Events Algebras

- Operators defining (derived) events from (basic) events
- E.g. (some of) the SNOOP operators
  - ◆ **Sequence:**  $e_1;e_2$  - occurs when  $e_2$  occurs after  $e_1$  occurred some time before
  - ◆ **Conjunction:**  $e_1 \wedge e_2$  - occurs when  $e_1$  and  $e_2$  occurs simultaneously
  - ◆ **Disjunction:**  $e_1 \vee e_2$  - occurs when either  $e_1$  or  $e_2$  occurs
  - ◆ **Negation:**  $\text{Not}(e_1, e_2, e_3)$  - occurs when  $e_3$  occurs,  $e_1$  occurred before, and  $e_2$  has not occurred in between
  - ◆ **Quantification:**  $\text{Any}(n, e)$  - occurs whenever event  $e$  occurs  $n$  times
  - ◆ ...
- The name of the game in CEPs is the **definition of derived complex events** that describe the desired output, base on the basic events of the input

# CEP as distributed service

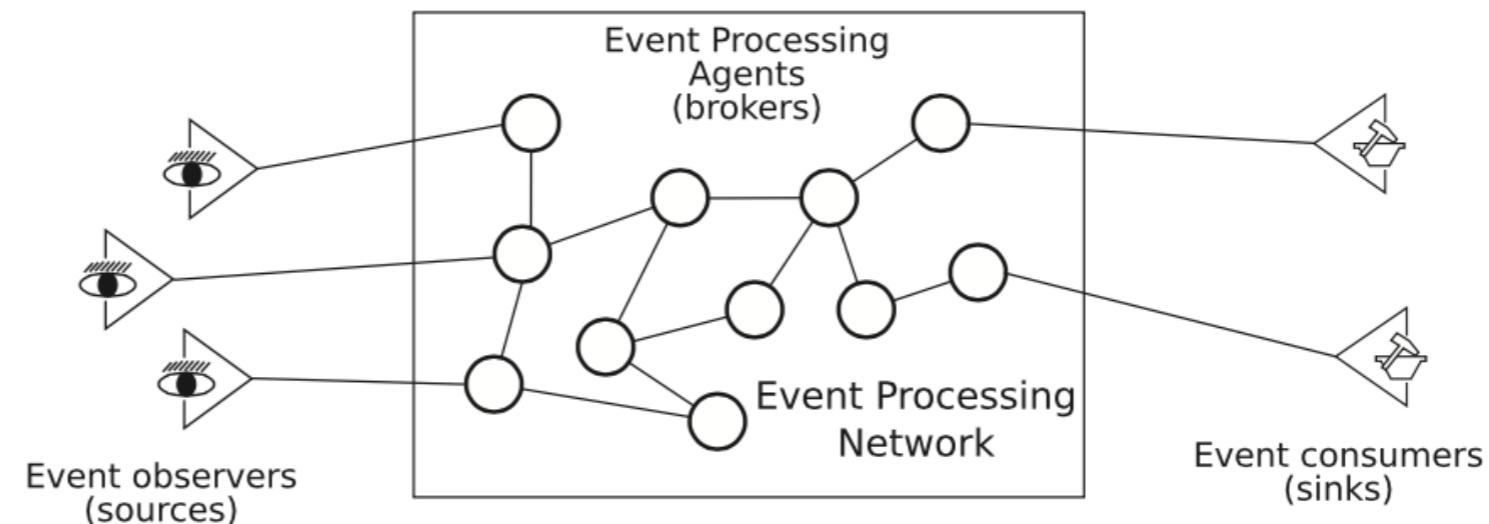
- CEP focused, from the start, on the features that DSMS lack (even before DSMS appeared!)
  - ◆ But, at least in the beginning, not so much on the features where DSMS excels

# CEP as distributed service

- CEP focused, from the start, on the features that DSMS lack (even before DSMS appeared!)
  - ◆ But, at least in the beginning, not so much on the features where DSMS excels
- They were mainly thought as a distributed service, able to deal with distributed and heterogeneous information sources

# CEP as distributed service

- CEP focused, from the start, on the features that DSMS lack (even before DSMS appeared!)
  - ◆ But, at least in the beginning, not so much on the features where DSMS excels
- They were mainly thought as a distributed service, able to deal with distributed and heterogeneous information sources
  - ◆ This suggests a distributed architecture with **event brokers** connected in an **event processing network**



## Framework for IFP Systems

# Framework for IFP Systems: (\*) models

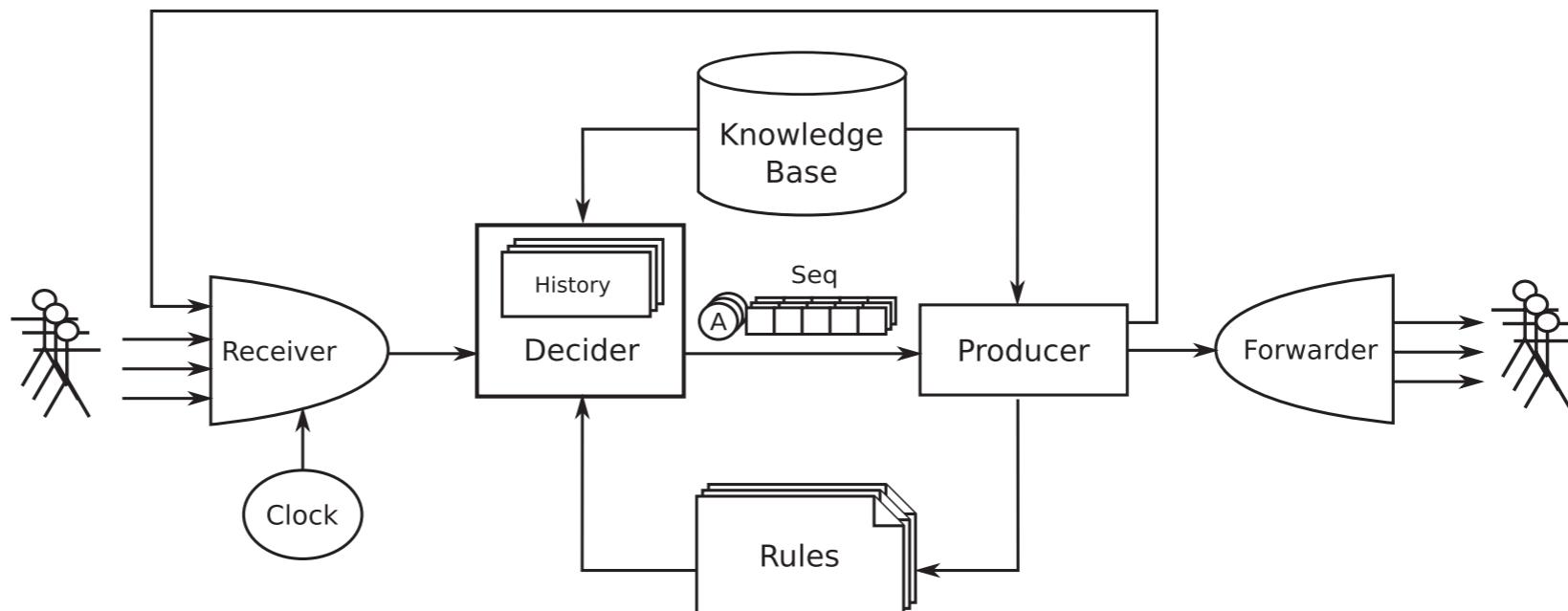
- **Functional Model**
- **Processing Model**
- **Deployment Model**
- **Interaction Model**
- **Data Model**
- **Time Model**
- **Rule Model**
- **Language Model**

# Framework for IFP Systems: functional model

- An IFP engine takes **flows of information coming from different sources** as its input, processes them, and **produces other flows of information** directed toward a set of sinks. **Processing rules** describe how to **filter**, **combine**, and **aggregate** incoming information to produce outgoing information.

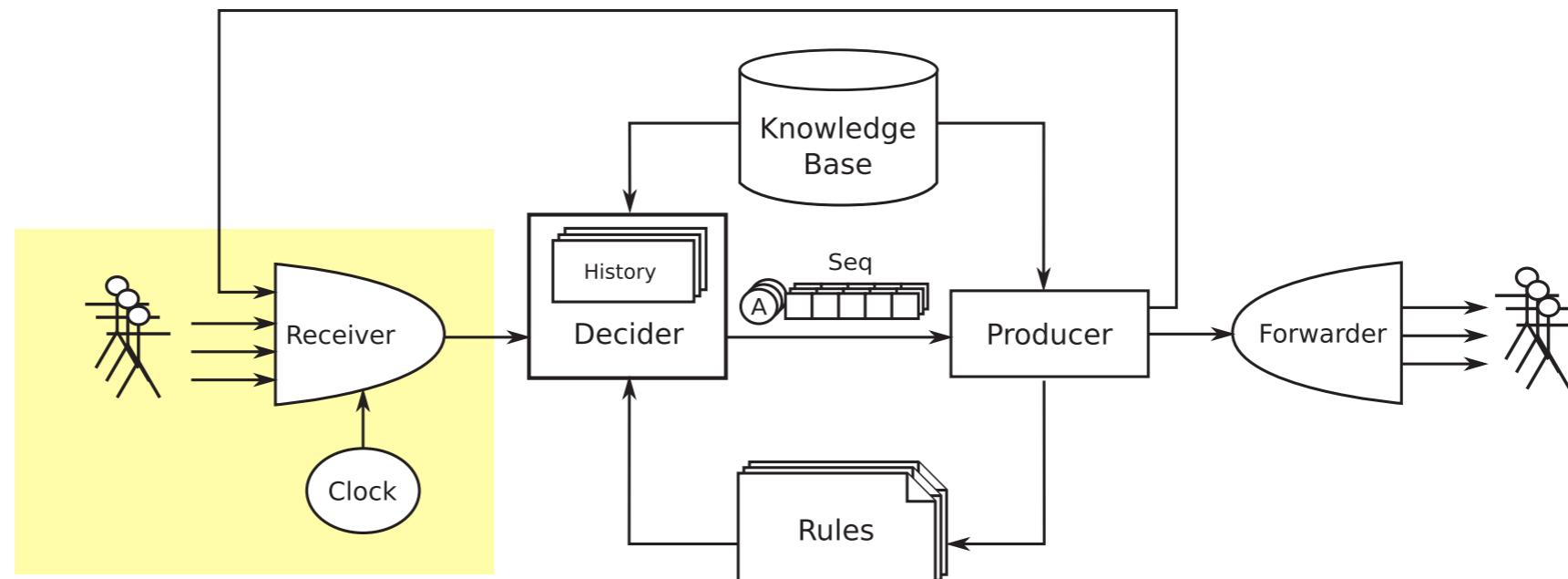
# Framework for IFP Systems: functional model

- An IFP engine takes **flows of information coming from different sources** as its input, processes them, and **produces other flows of information** directed toward a set of sinks. **Processing rules** describe how to **filter**, **combine**, and **aggregate** incoming information to produce outgoing information.
- This general behavior can be decomposed in a **set of elementary actions** performed by the different components



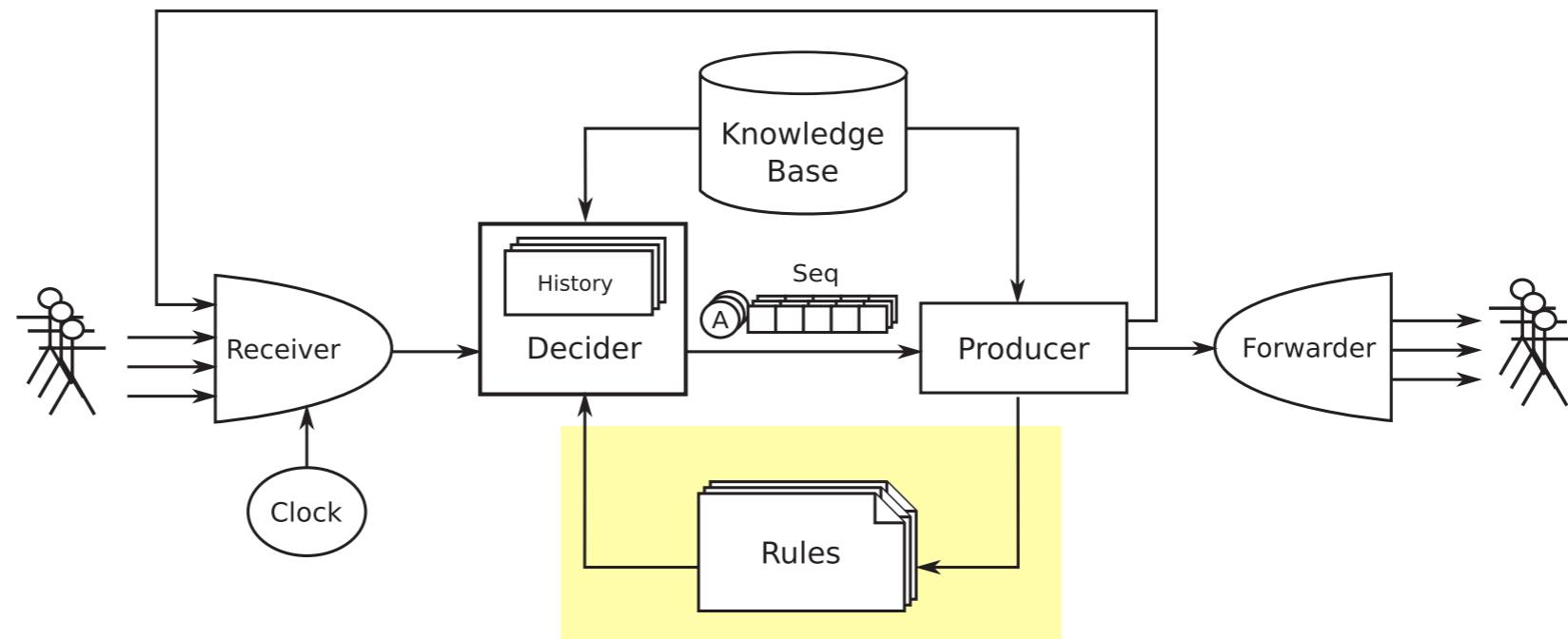
# Framework for IFP Systems: functional model

- **Receiver**, manage the channels connecting the sources with the IFP engine.
  - ◆ It implements the transport protocol adopted by the engine to move information around the network.
  - ◆ It also acts as a demultiplexer, receiving incoming items from multiple sources and sending them, **one by one**, to the next component in the IFP architecture



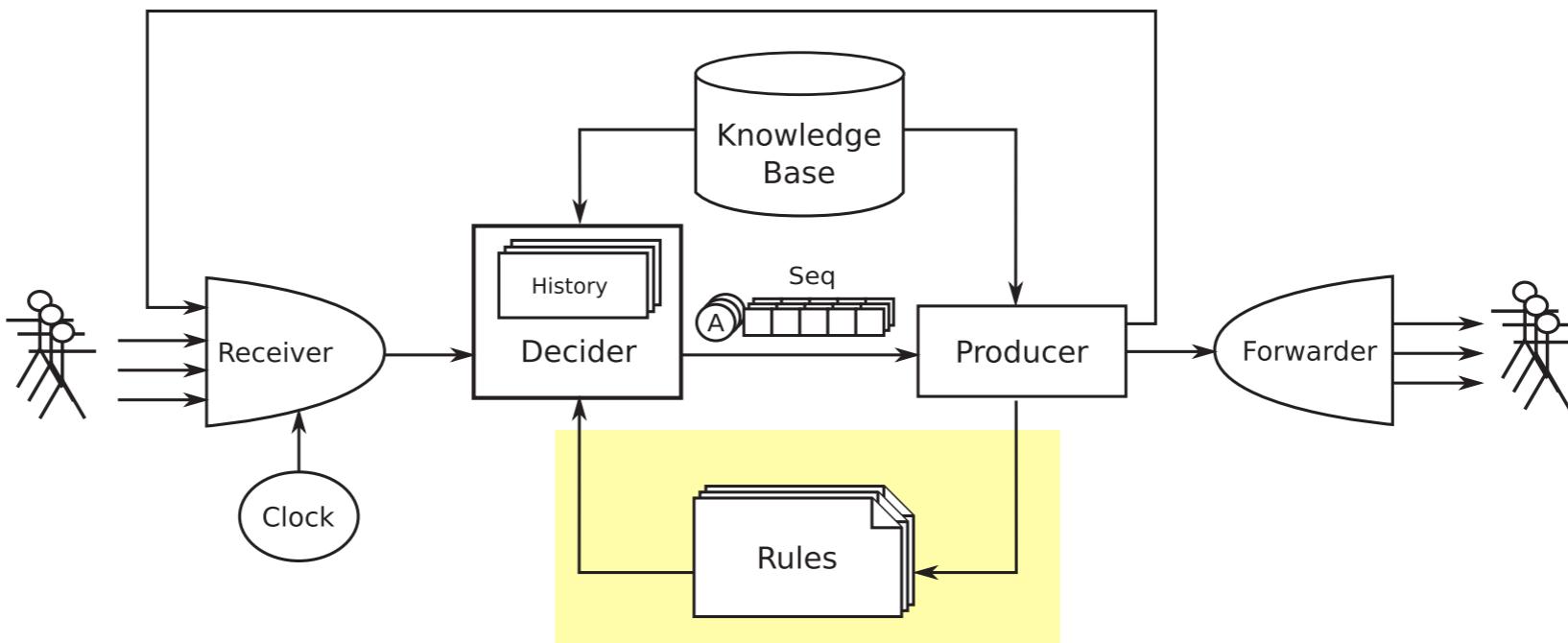
# Framework for IFP Systems: functional model

- The **information items** coming from the *external sources* or generated by *the clock* enter the main processing pipe, where they are elaborated **according to the processing rules** currently stored into the rules store



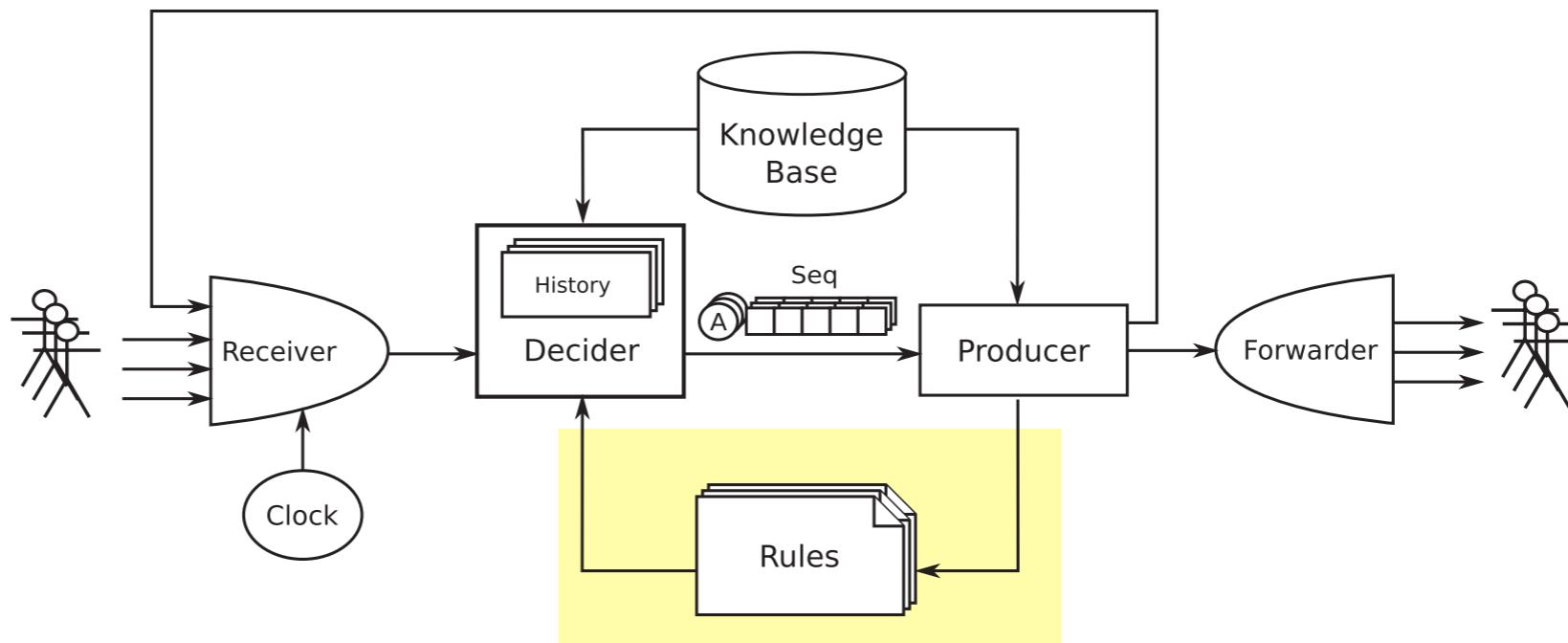
# Framework for IFP Systems: functional model

- The **information items** coming from the *external sources* or generated by *the clock* enter the main processing pipe, where they are elaborated **according to the processing rules** currently stored into the rules store
- The rules are viewed as composed by two parts:  $C \rightarrow A$ , where **C** is the **condition part**, while **A** is the **action part**.



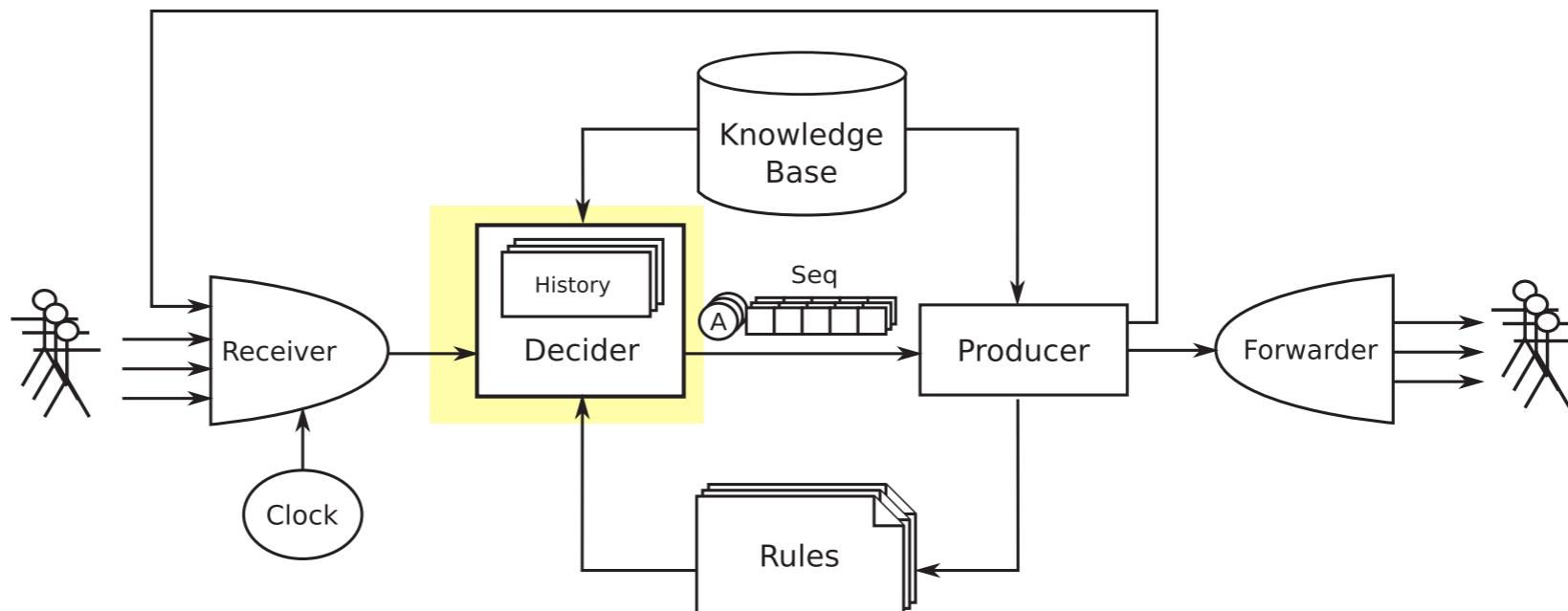
# Framework for IFP Systems: functional model

- The **information items** coming from the *external sources* or generated by *the clock* enter the main processing pipe, where they are elaborated **according to the processing rules** currently stored into the rules store
- The rules are viewed as composed by two parts:  $C \rightarrow A$ , where **C** is the **condition part**, while **A** is the **action part**.
- The information processing has two phases: a **detection** phase and a **production** phase



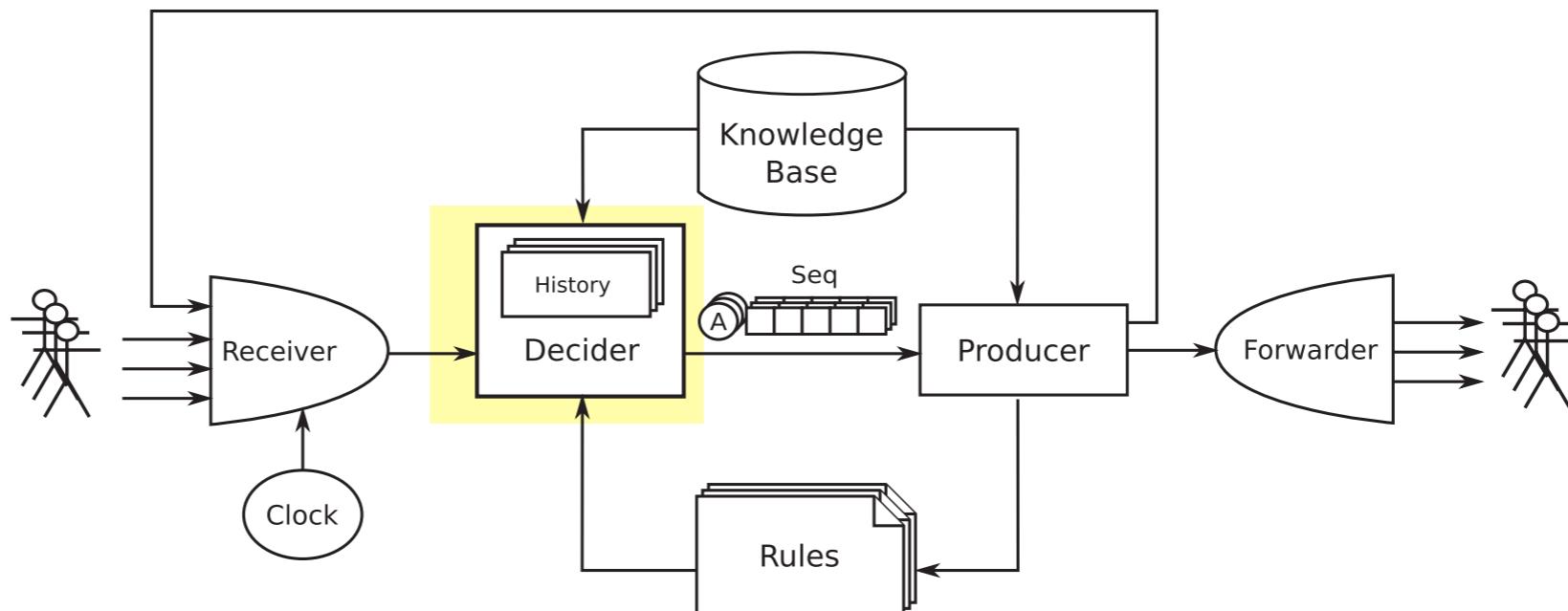
# Framework for IFP Systems: functional model

- The **decider**, which gets incoming information from the receiver, **item by item** and **looks at the condition part of rules to find those enabled**. The action part of each triggered rule is then passed to the producer for execution.



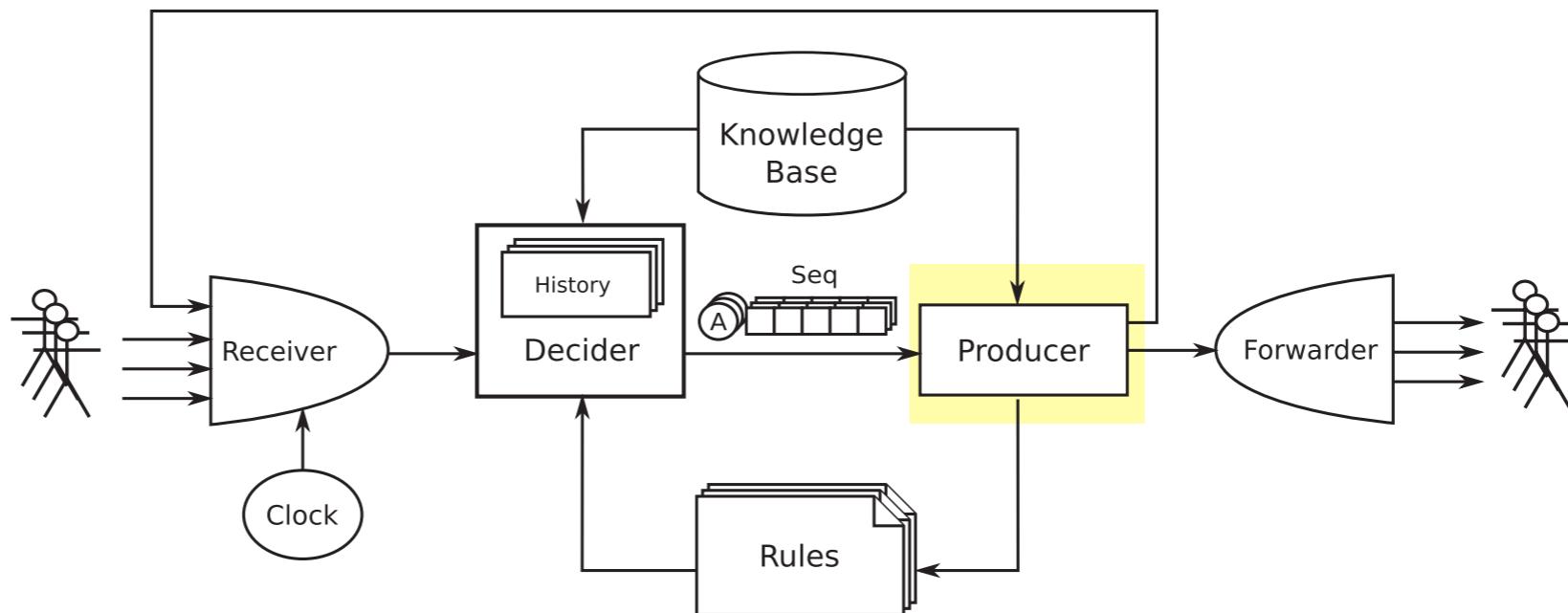
# Framework for IFP Systems: functional model

- The **decider**, which gets incoming information from the receiver, **item by item** and **looks at the condition part of rules to find those enabled**. The action part of each triggered rule is then passed to the producer for execution.
- The **decider** may need to **accumulate information items into a local storage** until the constraints of a rule are entirely satisfied. The presence of such memory through the **history** component inside.



# Framework for IFP Systems: functional model

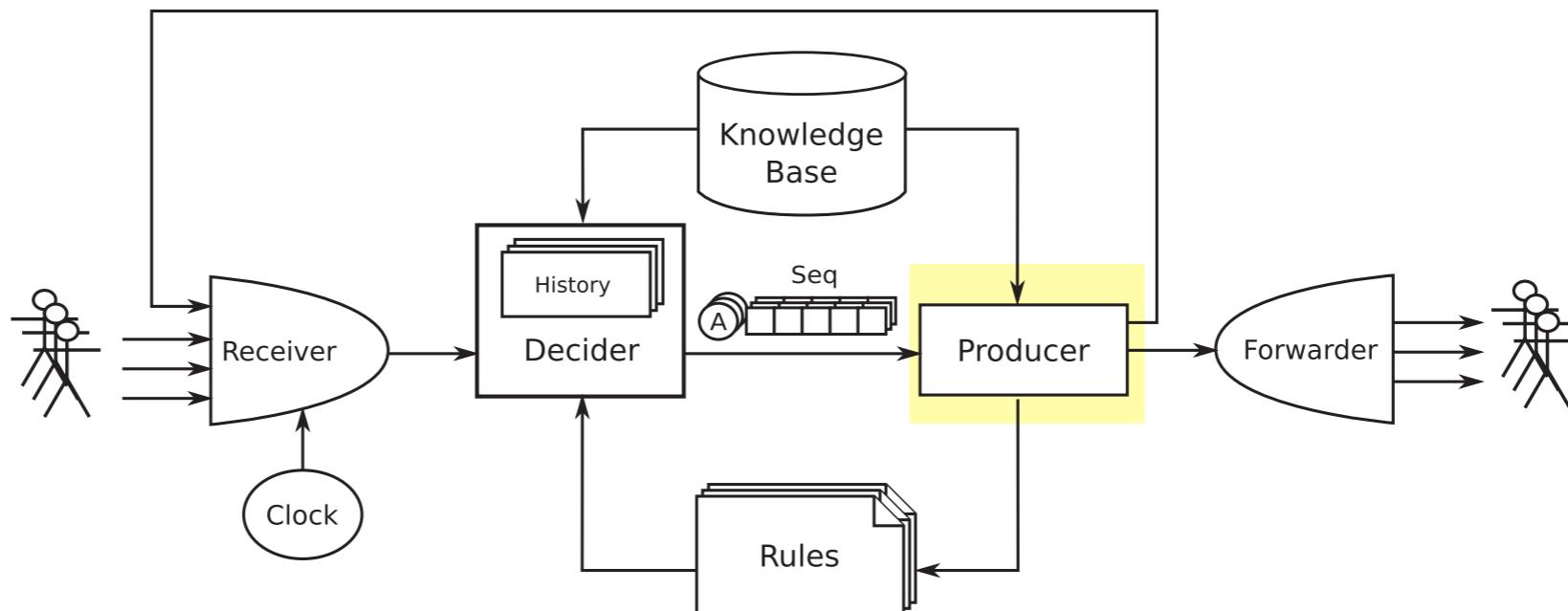
- The detection phase ends by passing to the **producer** an action **A** and a sequence of information items **seq** for each triggered rule, that is, **those items** (accumulated in the history by the decider) **that actually triggered the rule**.



# Framework for IFP Systems: functional model

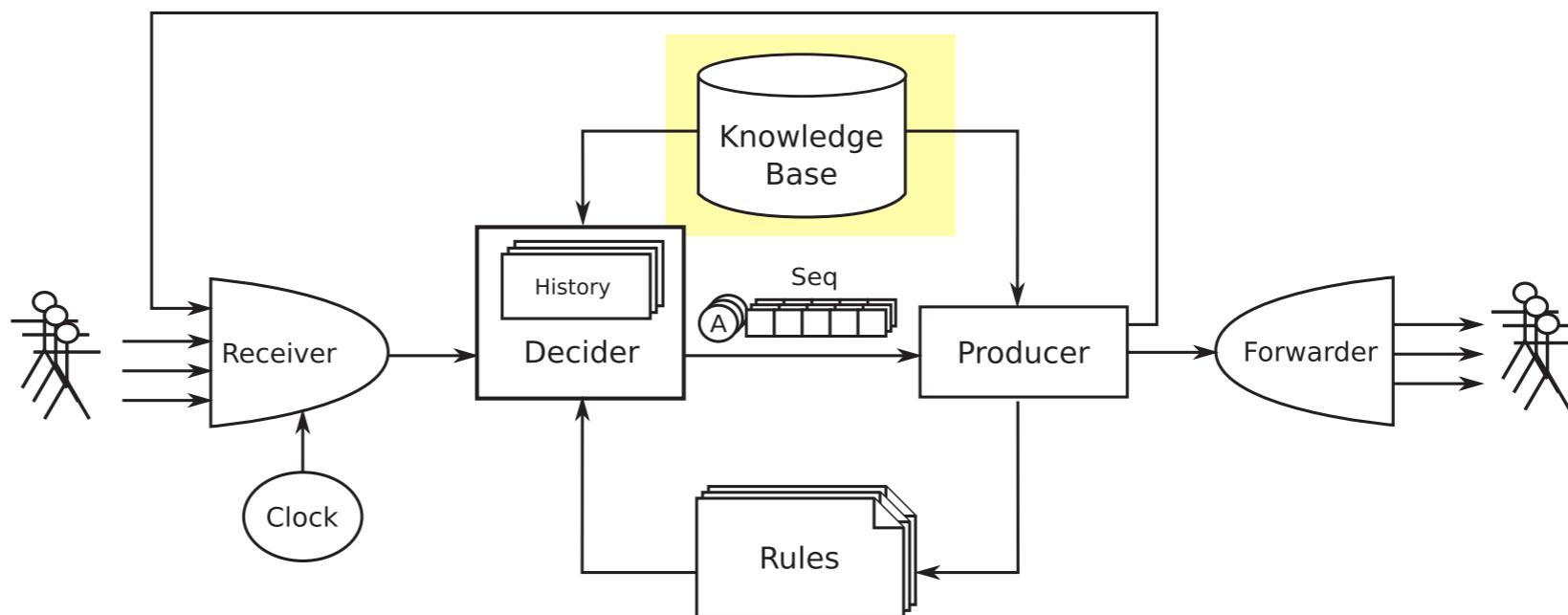
- The detection phase ends by passing to the **producer** an action **A** and a sequence of *information items seq for each triggered rule*, that is, **those items** (accumulated in the history by the decider) **that actually triggered the rule**.
- The **maximum allowed length of the sequence seq** is an important aspect for characterizing the expressiveness of the IFP engine:

- ◆ Single item;
- ◆ Bounded
- ◆ Unbounded



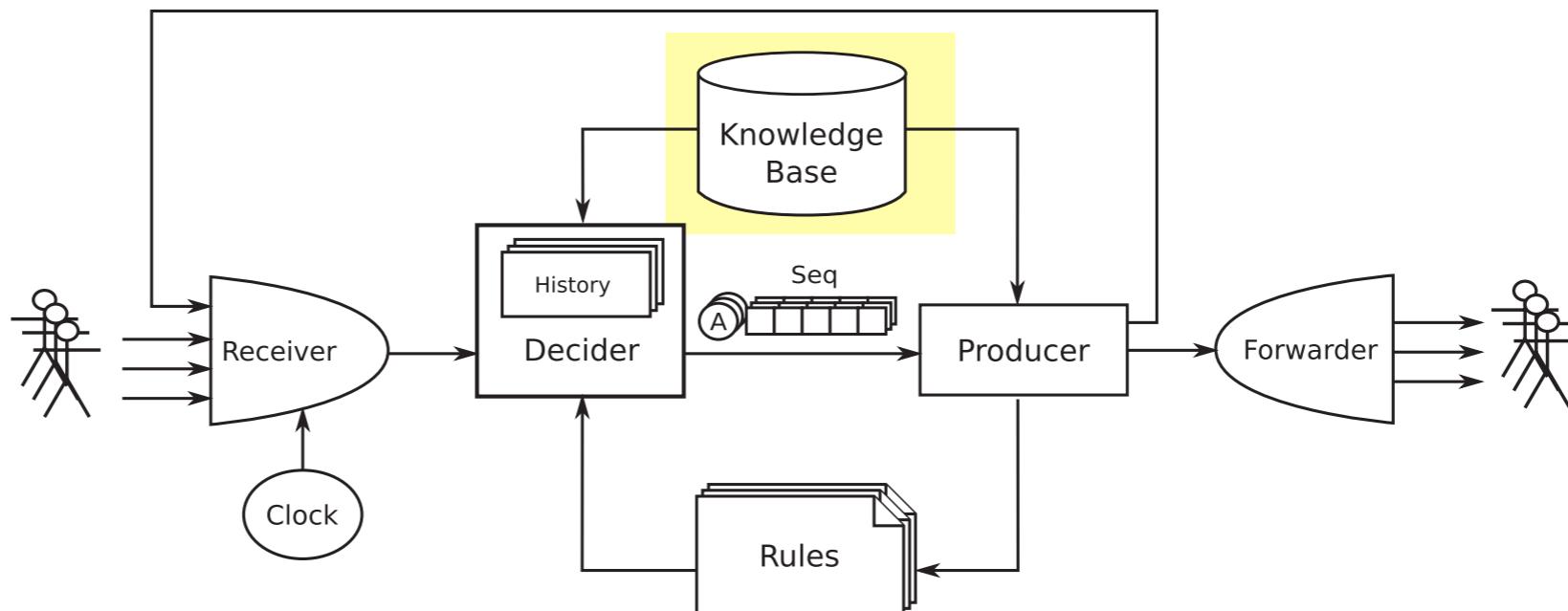
# Framework for IFP Systems: functional model

- The **knowledge base** represents, a **read-only memory that contains information used during the detection and production phases.**



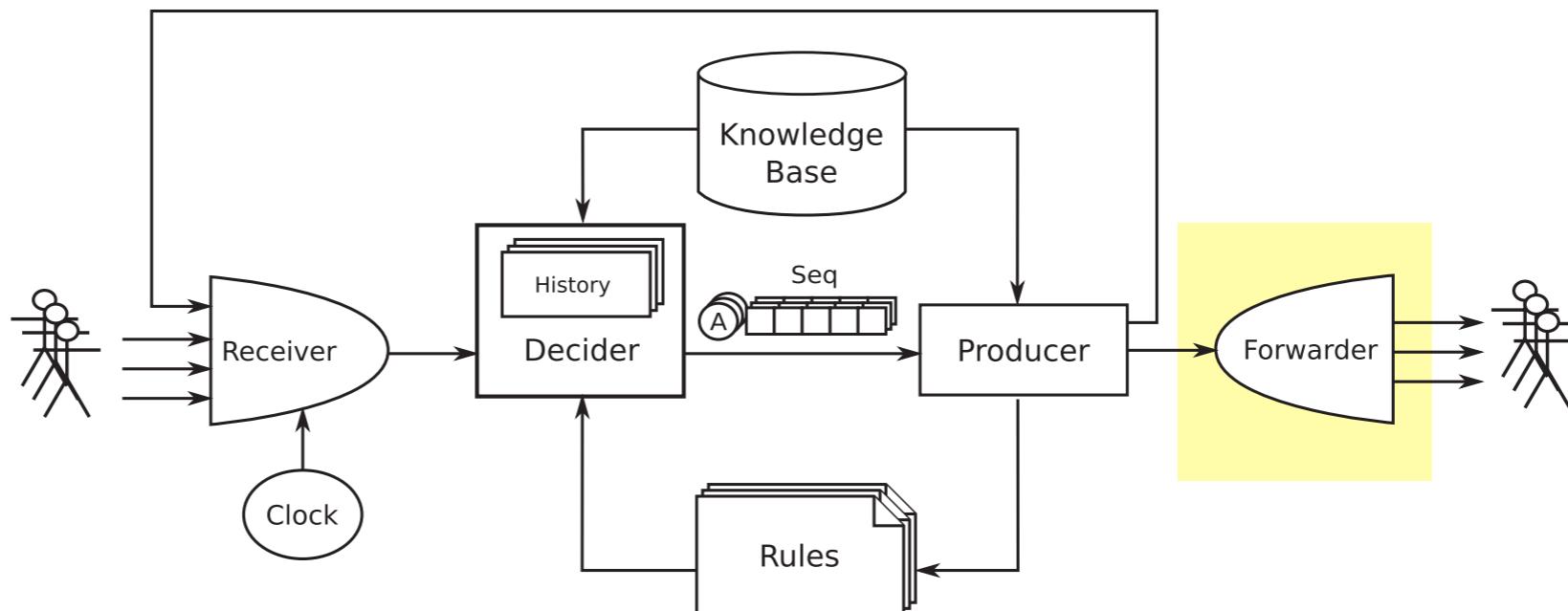
# Framework for IFP Systems: functional model

- The **knowledge base** represents, a **read-only memory that contains information used during the detection and production phases.**
  - ◆ This component is not present in all IFP engines
  - ◆ Usually, it is part of those systems developed by the database research community which allow for accessing persistent storage (typically in the form of database tables) during information processing



# Framework for IFP Systems: functional model

- The **forwarder** is the component in charge of delivering the information items generated by the producer to the expected sinks.
- Like the receiver, it implements the protocol to transport information along the network up to the sinks.



# Framework for IFP Systems: functional model - Overview

# Framework for IFP Systems: functional model - Overview

- Each time a **new item** (including those periodically produced by the clock) **enters** the engine through the receiver, a **detection-production cycle** is performed.

# Framework for IFP Systems: functional model - Overview

- Each time a **new item** (including those periodically produced by the clock) **enters** the engine through the receiver, a **detection-production cycle is performed**.
- **Detection phase** evaluates all the rules currently present in the rules store to **find those whose condition part is true**.
  - ◆ Together with the newly arrived information, this first phase may also use the information present in the knowledge base.

# Framework for IFP Systems: functional model - Overview

- Each time a **new item** (including those periodically produced by the clock) **enters** the engine through the receiver, a **detection-production cycle is performed**.
- **Detection phase** evaluates all the rules currently present in the rules store to **find those whose condition part is true**.
  - ◆ Together with the newly arrived information, this first phase may also use the information present in the knowledge base.
- At the end of detection phase, **we have a set of rules that have to be executed**, each *coupled with a sequence of information items*: those that triggered the rule and that were accumulated by the decider in the history.

# Framework for IFP Systems: functional model - Overview

- Each time a **new item** (including those periodically produced by the clock) **enters** the engine through the receiver, a **detection-production cycle is performed**.
- **Detection phase** evaluates all the rules currently present in the rules store to **find those whose condition part is true**.
  - ◆ Together with the newly arrived information, this first phase may also use the information present in the knowledge base.
- At the end of detection phase, **we have a set of rules that have to be executed**, each *coupled with a sequence of information items*: those that triggered the rule and that were accumulated by the decider in the history.
- The **producer** takes this information and **executes each triggered rule**.

# Framework for IFP Systems: functional model - Overview

- The **producer** takes this information and **executes each triggered rule**.

# Framework for IFP Systems: functional model - Overview

- The **producer** takes this information and **executes each triggered rule**.
- In executing the rules, the **producer may combine the items that triggered the rule** (as received by the decider), **together with the information present in the knowledge base**, to produce new information items

# Framework for IFP Systems: functional model - Overview

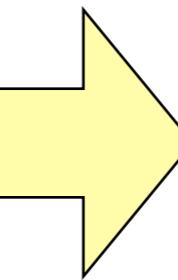
- The **producer** takes this information and **executes each triggered rule**.
- In executing the rules, the **producer may combine the items that triggered the rule** (as received by the decider), **together with the information present in the knowledge base**, to produce new information items
- Usually, these new items are sent to sinks (**through the forwarder**), but in some engines, **they can also be sent internally, to be processed again (recursive processing)**.

# Framework for IFP Systems: functional model - Overview

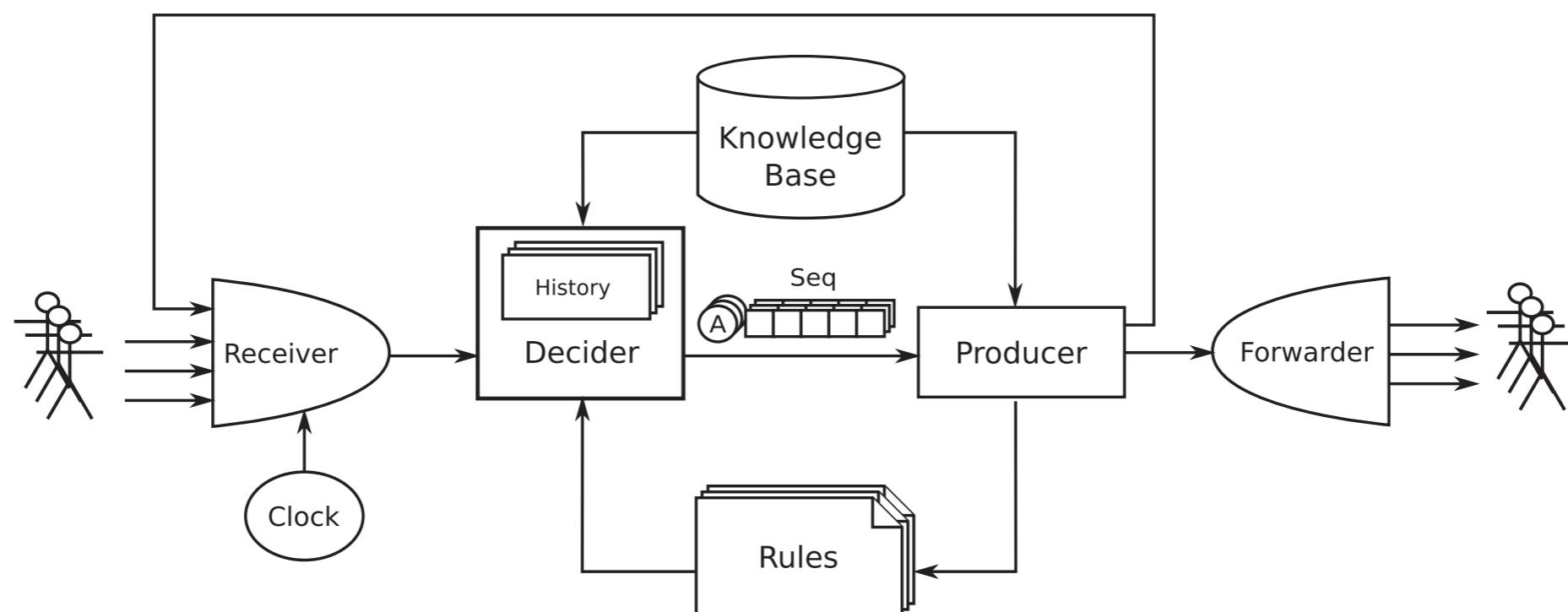
- The **producer** takes this information and **executes each triggered rule**.
- In executing the rules, the **producer may combine the items that triggered the rule** (as received by the decider), **together with the information present in the knowledge base**, to produce new information items
- Usually, these new items are sent to sinks (**through the forwarder**), but in some engines, **they can also be sent internally, to be processed again (recursive processing)**.
- Some engines **allow actions to change the rule set** by adding new rules or removing them.

# Framework for IFP Systems: Processing model

- The last information item entering the decider
- The set of deployed rules
- The information stored in the history
- The information stored in the knowledge base

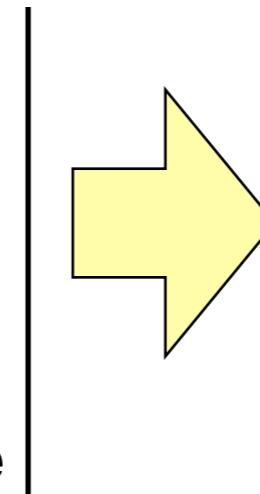


the **output** of a single  
detection-production cycle



# Framework for IFP Systems: Processing model

- The **last information item** entering the decider
- The set of deployed rules
- The **information stored in the history**
- The **information stored in the knowledge base**



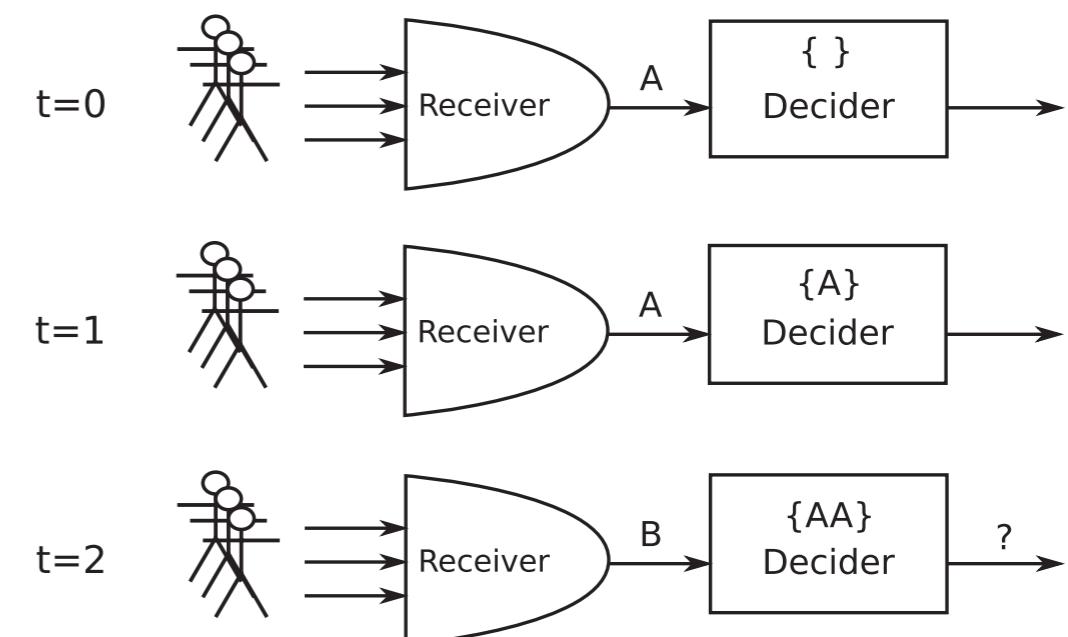
the **output** of a single  
**detection-production cycle**

- Depends on:
  - ◆ **Selection policy**
  - ◆ **Consumption policy**
  - ◆ **Load shedding**

# Processing model - Selection Policy

## ■ Selection policy.

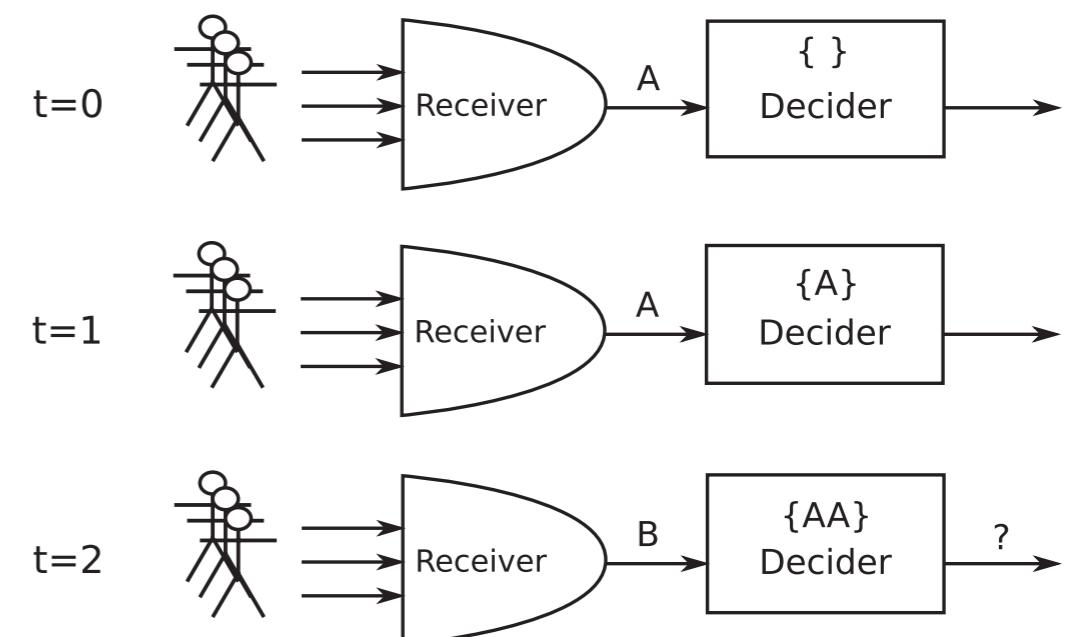
- ◆ In the presence of situations in which a **single rule  $R$  may fire more than once**, picking different items from the history, the **selection policy specifies if  $R$  has to fire once or more times** and which items are actually selected and sent to the producer.



# Processing model - Selection Policy

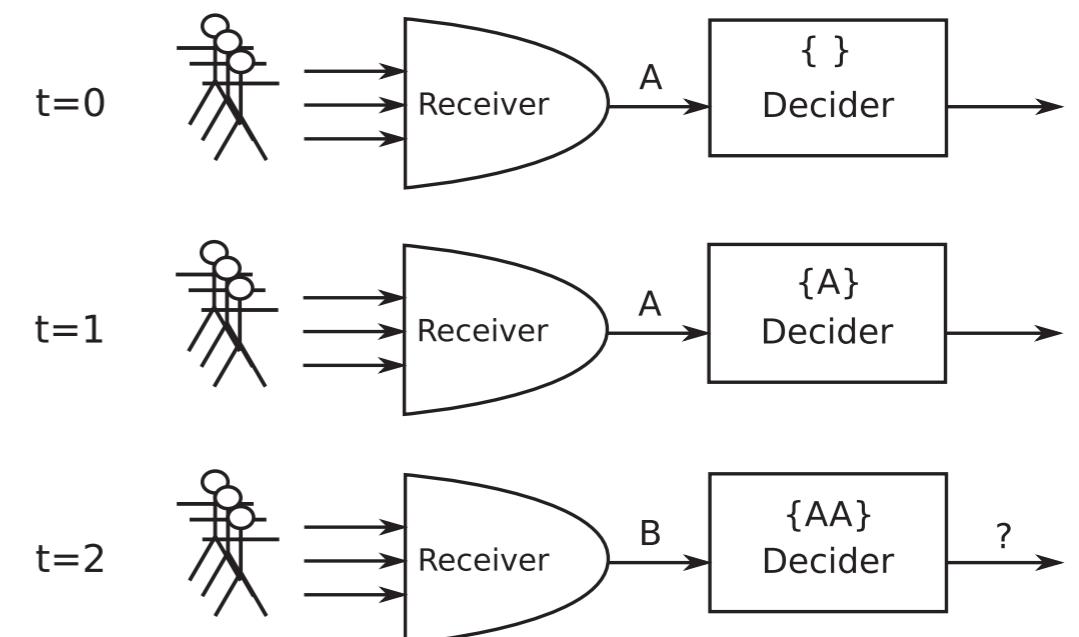
## ■ Selection policy.

- ◆ In the presence of situations in which a **single rule  $R$  may fire more than once**, picking different items from the history, the **selection policy specifies if  $R$  has to fire once or more times** and which items are actually selected and sent to the producer.
- ◆ Example: information items sent by the receiver to the decider at different times, together with the information stored by the decider into the history



# Processing model - Selection Policy

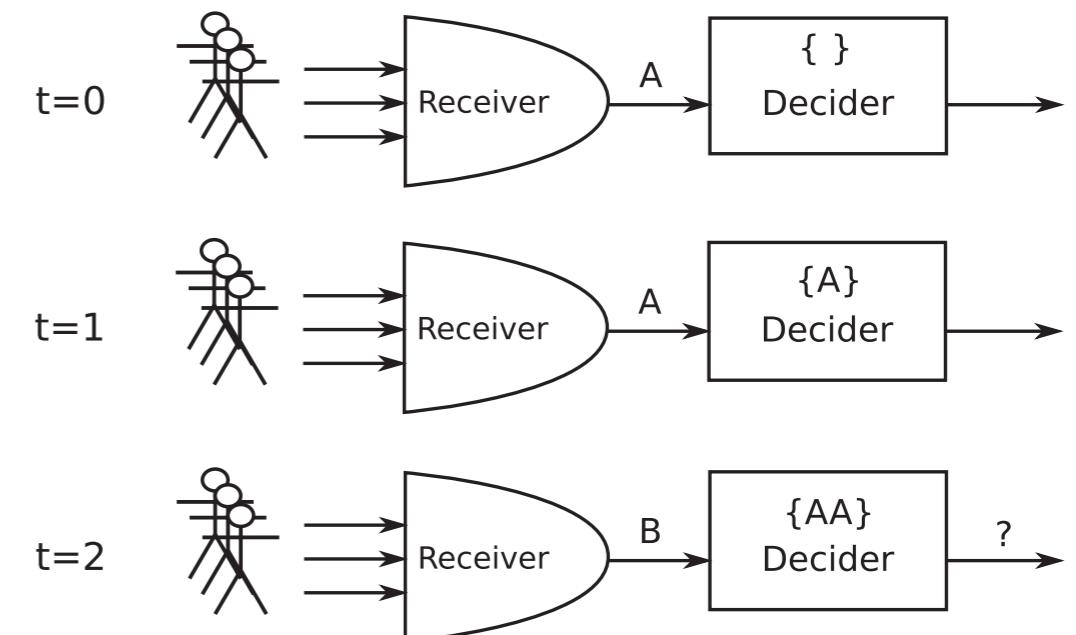
- Scenario with a single rule:
  - ◆ Condition part:  $A \wedge B$



# Processing model - Selection Policy

- Scenario with a single rule:
  - ◆ Condition part:  $A \wedge B$
- At  $t = 0$ , information A exits the receiver, starting a detection-production cycle.

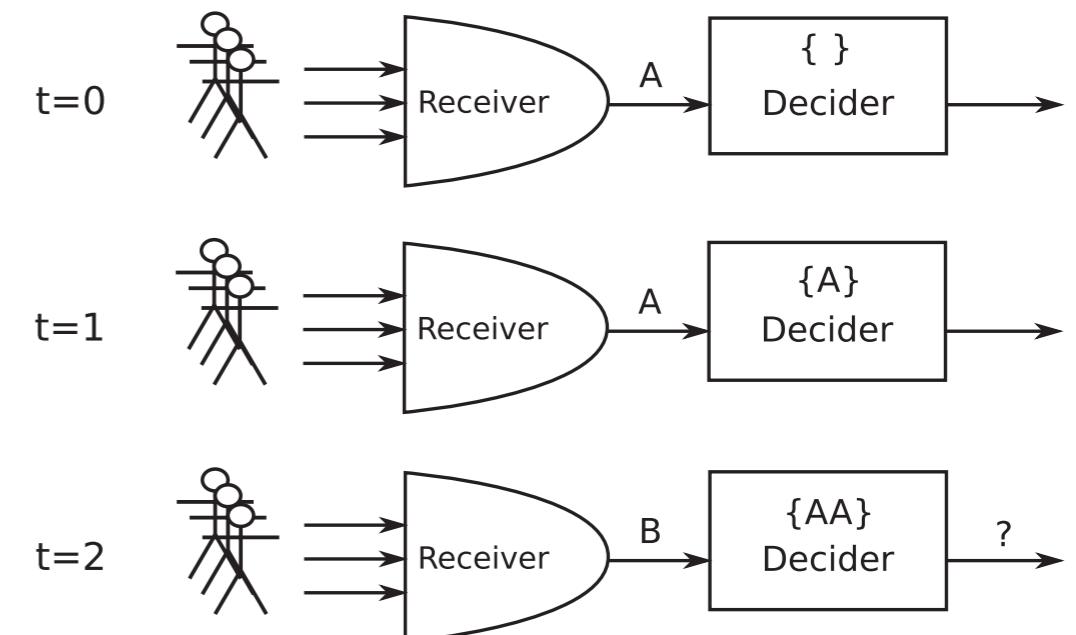
At this time, the pattern  $A \wedge B$  is not detected, but the decider has to remember that A has been received



# Processing model - Selection Policy

- Scenario with a single rule:
  - ◆ Condition part:  $A \wedge B$
- At  $t = 0$ , information A exits the receiver, starting a detection-production cycle.

At this time, the pattern  $A \wedge B$  is not detected, but the decider has to remember that A has been received
- At  $t = 1$ , a new A exits the receiver.



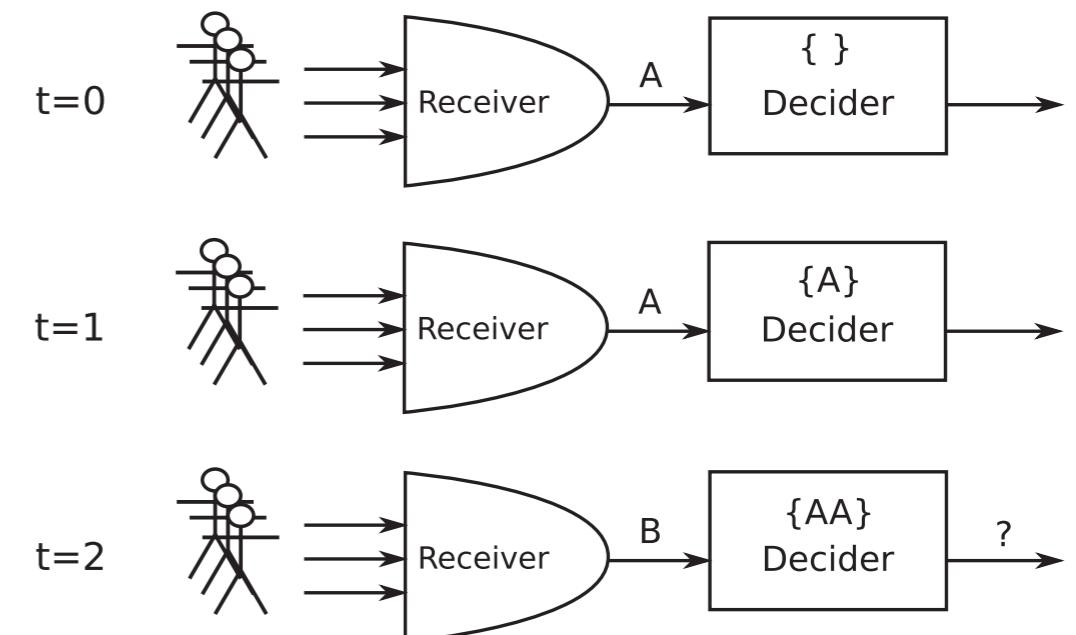
# Processing model - Selection Policy

- Scenario with a single rule:
  - ◆ Condition part:  $A \wedge B$
- At  $t = 0$ , information A exits the receiver, starting a detection-production cycle.

At this time, the pattern  $A \wedge B$  is not detected, but the decider has to remember that A has been received

- At  $t = 1$ , a new A exits the receiver.
- At  $t = 2$ , when information B exits the receiver.

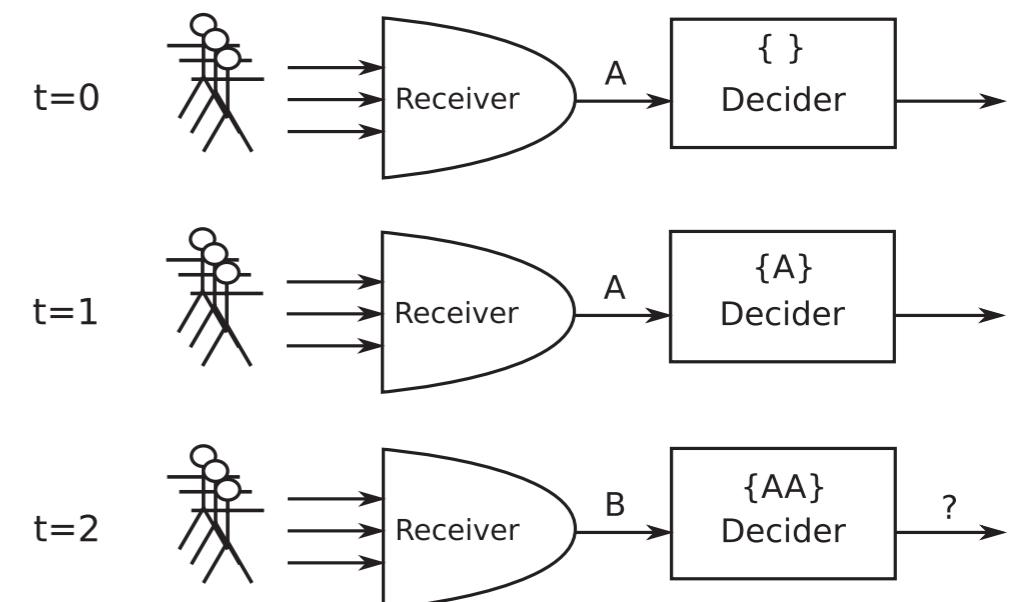
- ◆ not only is the condition part of the rule is satisfied,
- ◆ but it is satisfied by two possible sets of items



# Processing model - Selection Policy

- Selection Policy = ***multiple selection***

- ◆ **Each rule to fire more than once** at each detection-production cycle
  - ◆ The decider would **send two sequences of items to the producer**
    - A( $t = 0$ ), B
    - A( $t = 1$ ), B



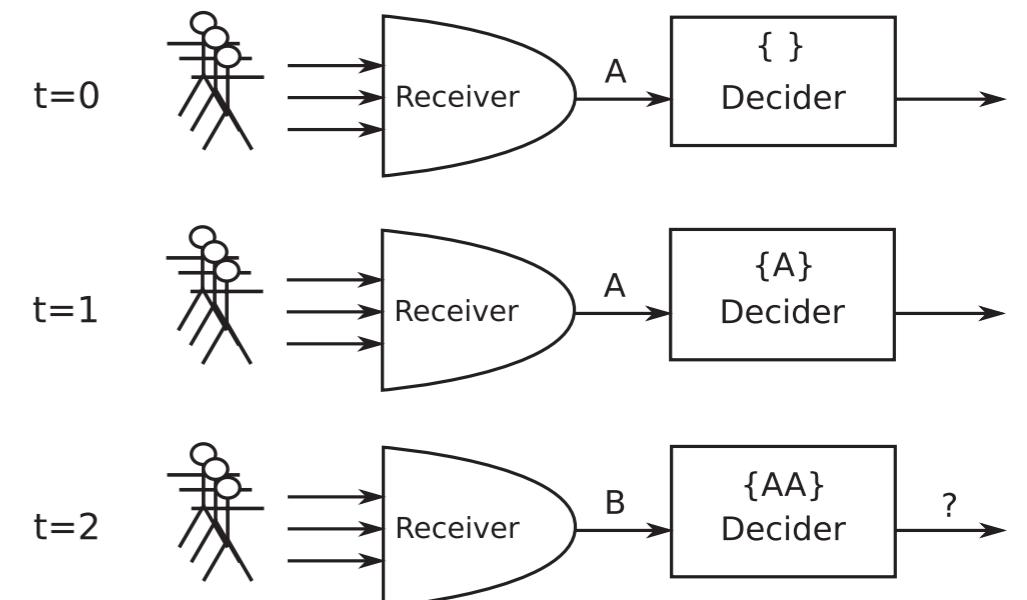
# Processing model - Selection Policy

## ■ Selection Policy = *multiple* selection

- ◆ **Each rule to fire more than once** at each detection-production cycle
- ◆ The decider would **send two sequences of items to the producer**
  - A( $t = 0$ ), B
  - A( $t = 1$ ), B

## ■ Selection Policy = *single* selection

- ◆ Allow **rules to fire at most once** at each detection-production cycle
- ◆ The decider would **choose one of the two As received**, sending a **single sequence** to the producer
  - Which one? A family of policies !



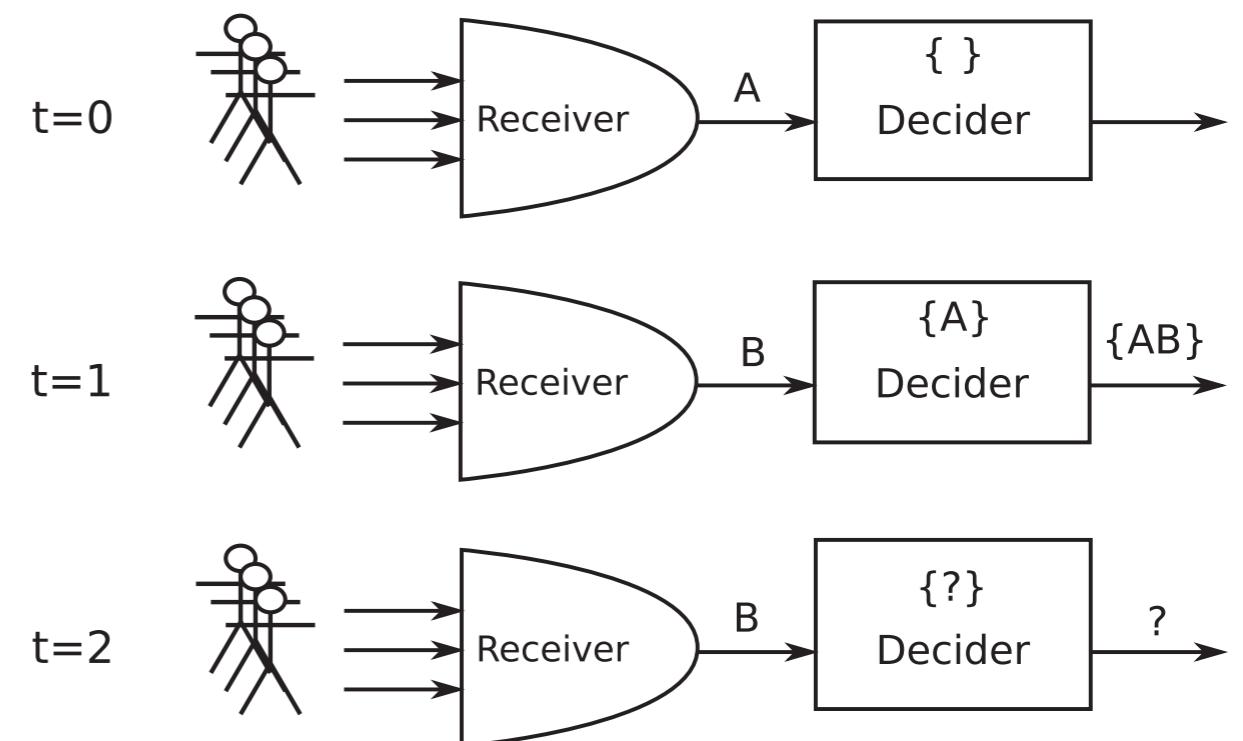
# Processing model - Selection Policy

- Selection Policy = ***multiple selection***
- Selection Policy = ***single selection***
- ***Programmable policy***
  - ◆ by including special language constructs that **enable users to decide**, often rule by rule, if they have to **fire once or more than once at each detection-production cycle**,
  - ◆ and in case fire once, which elements have to be actually selected among the different possible combinations

# Processing model - Consumption Policy

## ■ Consumption policy. (related with the selection policy)

- ◆ Specifies whether or not an **information item selected** in a given detection-production cycle **can be considered again in future** processing cycles.

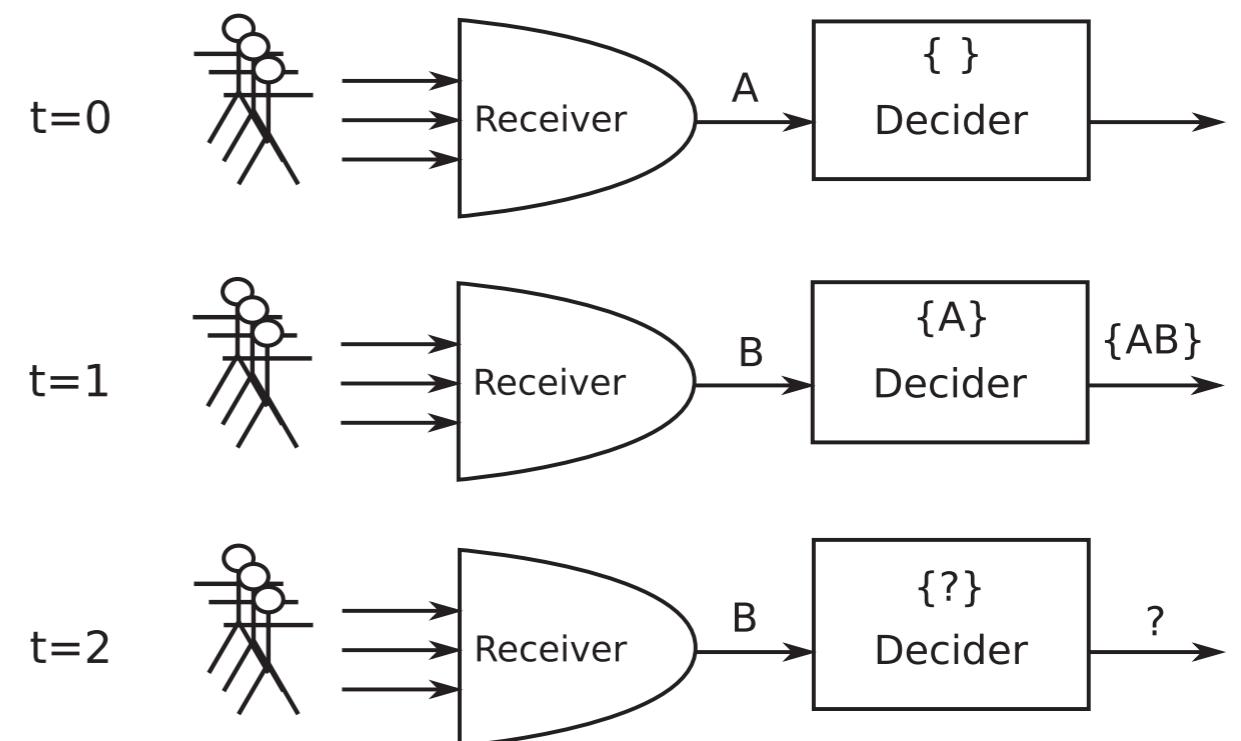


# Processing model - Consumption Policy

- **Consumption policy.** (related with the selection policy)
  - ◆ Specifies whether or not an **information item selected** in a given detection-production cycle **can be considered again in future** processing cycles.

- Scenario with a Single rule:

- ◆ Condition part:  $A \wedge B$



# Processing model - Consumption Policy

- **Consumption policy.** (related with the selection policy)
    - ◆ Specifies whether or not an **information item selected** in a given detection-production cycle **can be considered again in future** processing cycles.
  - Scenario with a Single rule:
    - ◆ Condition part:  $A \wedge B$
    - ◆ At time  $t = 2$ , a new instance of B enters the system. In such a situation, the consumption policy determines if preexisting items A and B have still to be taken into consideration to decide if rule  $A \wedge B$  is satisfied.
- $t=0$        $t=1$        $t=2$
- 
- ```
graph LR; Person((Person)) --> Receiver1[Receiver]; Receiver1 --> Decider0["{} Decider"]; Decider0 --> Out0["?"]; subgraph Time0 [t=0]; Person; Receiver1; Decider0; end; subgraph Time1 [t=1]; Person; Receiver2[Receiver]; Decider1["{A} Decider"]; Decider1 --> Out1["{AB}"]; end; subgraph Time2 [t=2]; Person; Receiver3[Receiver]; Decider2[" {?} Decider"]; Decider2 --> Out2["?"]; end;
```

# Processing model - Consumption Policy

- **Consumption policy.** (related with the selection policy)
  - ◆ Specifies whether or not an **information item selected** in a given detection-production cycle **can be considered again in future** processing cycles.

# Processing model - Consumption Policy

- **Consumption policy.** (related with the selection policy)
  - ◆ Specifies whether or not an **information item selected** in a given detection-production cycle **can be considered again in future** processing cycles.
- The systems that adopt the **zero consumption policy** do not *invalidate used information items*, which can trigger the same rule more than once.

# Processing model - Consumption Policy

- **Consumption policy.** (related with the selection policy)
  - ◆ Specifies whether or not an **information item selected** in a given detection-production cycle **can be considered again in future** processing cycles.
- The systems that adopt the **zero consumption policy** do not *invalidate used information items*, which can trigger the same rule more than once.
- Conversely, the systems that adopt the **selected consumption policy** consume all the items once they have been selected by the decider. This means that *an information item can be used at most once for each rule*.

# Processing model - Consumption Policy

- **Consumption policy.** (related with the selection policy)
  - ◆ Specifies whether or not an **information item selected** in a given detection-production cycle **can be considered again in future** processing cycles.
- The systems that adopt the **zero consumption policy** do not *invalidate used information items*, which can trigger the same rule more than once.
- Conversely, the systems that adopt the **selected consumption policy** consume all the items once they have been selected by the decider. This means that *an information item can be used at most once for each rule*.
- DSMSs adopted zero consumption policy

# Processing model - Consumption Policy

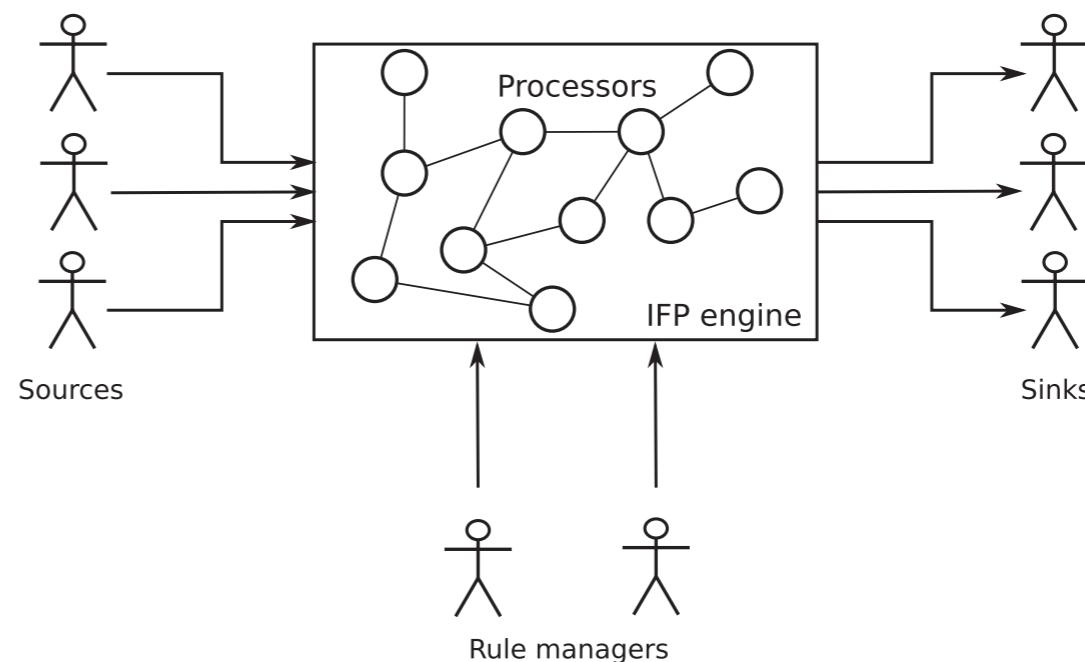
- **Consumption policy.** (related with the selection policy)
  - ◆ Specifies whether or not an **information item selected** in a given detection-production cycle **can be considered again in future** processing cycles.
- The systems that adopt the **zero consumption policy** do not *invalidate used information items*, which can trigger the same rule more than once.
- Conversely, the systems that adopt the **selected consumption policy** consume all the items once they have been selected by the decider. This means that *an information item can be used at most once for each rule*.
- DSMSs adopted zero consumption policy
- CEP systems may adopt either policies or **programmable selection**

# Processing model - Load shedding

- Load shedding is a technique adopted by some IFP systems to **deal with bursty inputs.**
- It can be described as an **automatic drop of information items** when the **input rate becomes too high for the processing capabilities of the engine**

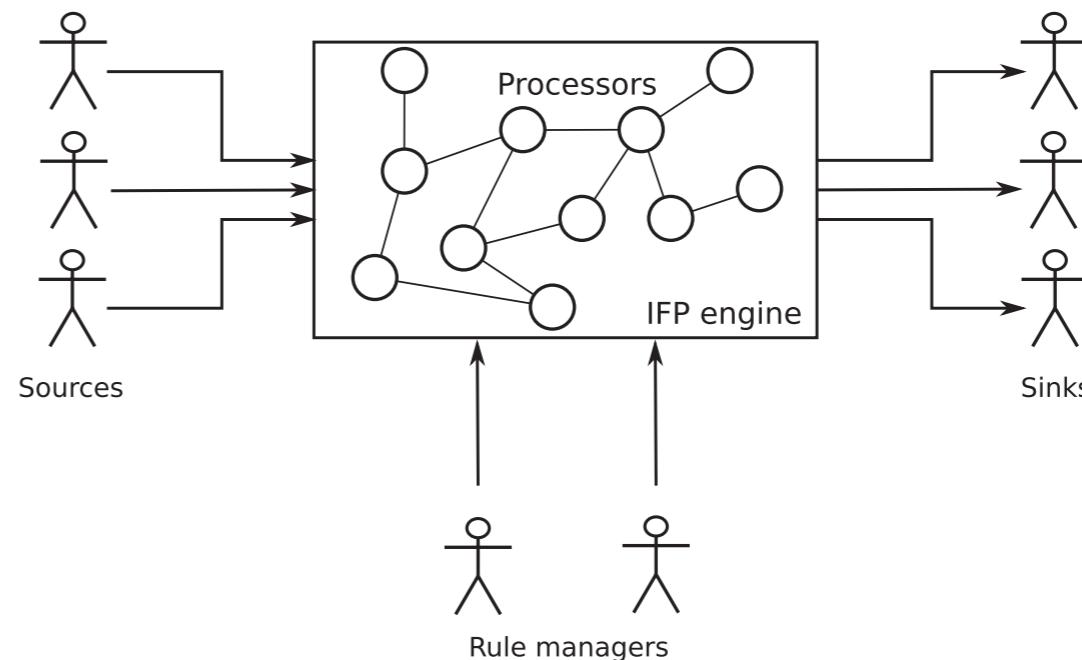
# Framework for IFP Systems: Deployment model

- Several IFP applications include a **large number of sources** and **sinks**—possibly dispersed over a wide geographical area—producing and consuming a large amount of information that the IFP engine has to process in a timely manner



# Framework for IFP Systems: Deployment model

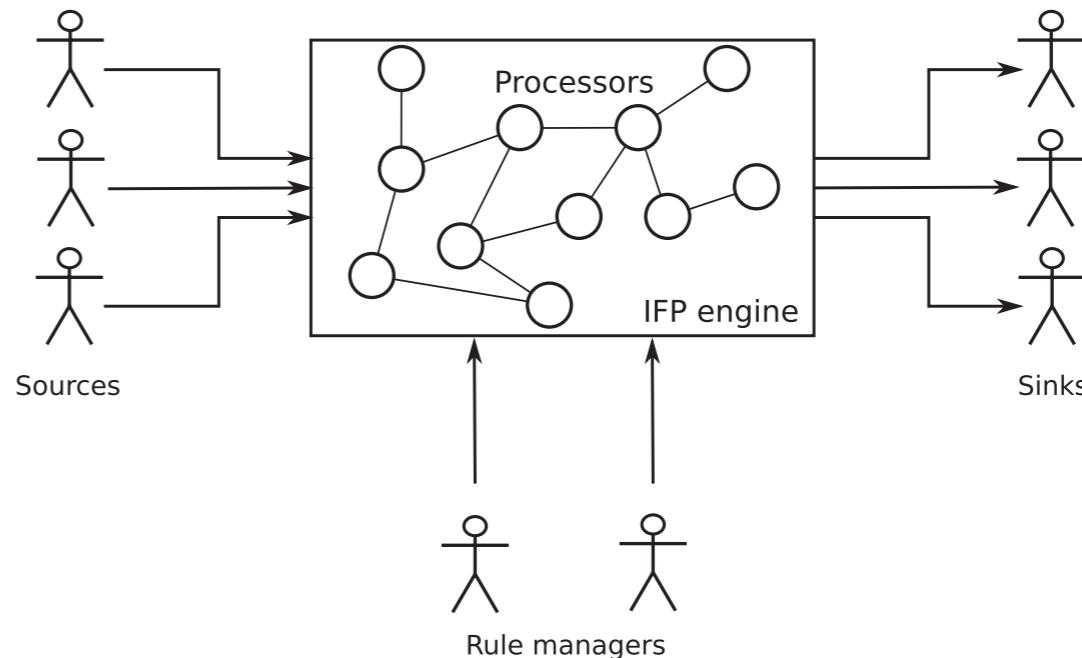
- Several IFP applications include a **large number of sources** and **sinks**—possibly dispersed over a wide geographical area—producing and consuming a large amount of information that the IFP engine has to process in a timely manner
- **Deployment architecture** of the engine, that is, how the components that implement the functional architecture can be **distributed over multiple nodes** to achieve **scalability**.



# Framework for IFP Systems: Deployment model

- Several IFP applications include a **large number of sources** and **sinks**—possibly dispersed over a wide geographical area—producing and consuming a large amount of information that the IFP engine has to process in a timely manner
- **Deployment architecture** of the engine, that is, how the components that implement the functional architecture can be **distributed over multiple nodes** to achieve **scalability**.

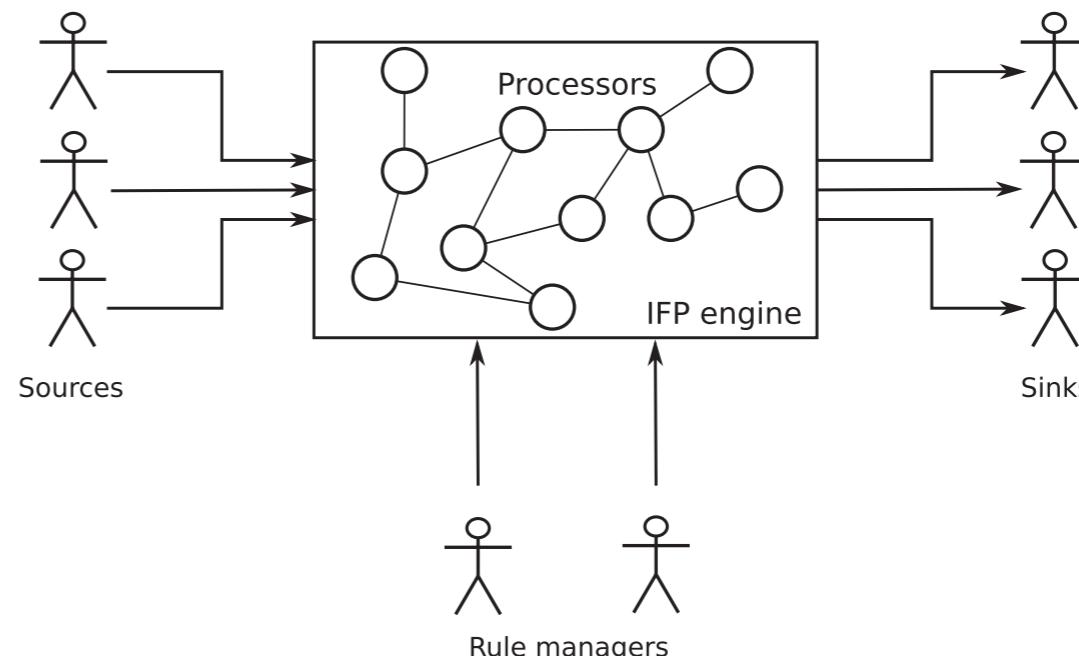
- Centralized



# Framework for IFP Systems: Deployment model

- Several IFP applications include a **large number of sources** and **sinks**—possibly dispersed over a wide geographical area—producing and consuming a large amount of information that the IFP engine has to process in a timely manner
- **Deployment architecture** of the engine, that is, how the components that implement the functional architecture can be **distributed over multiple nodes** to achieve **scalability**.

- Centralized
- Distributed
  - ◆ Clustered
  - ◆ Networked



# Framework for IFP Systems: Interaction model

- The **interaction model**, refer to the characteristics of the interaction among the main components that form an IFP application

# Framework for IFP Systems: Interaction model

- The **interaction model**, refer to the characteristics of the interaction among the main components that form an IFP application
- **Sub Models:**
  - ◆ **Observation model** refers to the interaction between information **sources** and the IFP engine
  - ◆ **Notification model** refers to the interaction between the engine and the **sinks**
  - ◆ **Forwarding model** defines the characteristics of the interaction among processors in the case of a distributed implementation of the engine

# Framework for IFP Systems: Interaction model

- The **interaction model**, refer to the characteristics of the interaction among the main components that form an IFP application
- **Sub Models:**
  - ◆ **Observation model** refers to the interaction between information **sources** and the IFP engine
  - ◆ **Notification model** refers to the interaction between the engine and the **sinks**
  - ◆ **Forwarding model** defines the characteristics of the interaction among processors in the case of a distributed implementation of the engine
- **Interaction styles**
  - ◆ Push
  - ◆ Pull

# Framework for IFP Systems: Interaction model

- The **interaction model**, refer to the characteristics of the interaction among the main components that form an IFP application
- **Sub Models:**
  - ◆ **Observation model** refers to the interaction between information **sources** and the IFP engine
  - ◆ **Notification model** refers to the interaction between the engine and the **sinks**
  - ◆ **Forwarding model** defines the characteristics of the interaction among processors in the case of a distributed implementation of the engine
- **Interaction styles**
  - ◆ Push
  - ◆ Pull

**push style is the most common** in the observation model, notification model, and especially in the forwarding model

# Framework for IFP Systems: Data model

## ■ Data Model

- ◆ How they represent single information items flowing from sources to sinks
- ◆ How they organize them in flows

# Framework for IFP Systems: Data model

- **Data Model**
  - ◆ How they represent single information items flowing from sources to sinks
  - ◆ How they organize them in flows
- One of the aspects that **distinguishes DSMSs from CEP systems** is that the former manipulate **generic data items**, while the latter manipulate **event notifications**. - **the nature of items**

# Framework for IFP Systems: Data model

## ■ Data Model

- ◆ How they represent single information items flowing from sources to sinks
  - ◆ How they organize them in flows
- One of the aspects that **distinguishes DSMSs from CEP systems** is that the former manipulate **generic data items**, while the latter manipulate **event notifications**. - **the nature of items**
- Another aspect is their **format**
- ◆ tuples, either typed or untyped
  - ◆ records, organized as sets of key-value pairs
  - ◆ objects, as in object-oriented languages or databases
  - ◆ XML documents

# Framework for IFP Systems: Data model

## ■ Data Model

- ◆ How they represent single information items flowing from sources to sinks
- ◆ How they organize them in flows

## ■ A final aspect regarding information items is the ability of an **IFP system to deal with uncertainty**

- ◆ The information received from sources has an associated degree of uncertainty

# Framework for IFP Systems: Data model

## ■ Data Model

- ◆ How they represent single information items flowing from sources to sinks
- ◆ How they organize them in flows

## ■ A final aspect regarding information items is the ability of an **IFP system to deal with uncertainty**

- ◆ The information received from sources has an associated degree of uncertainty

## ■ Flows

- ◆ Homogeneous information flows (same flow have the same format)
- ◆ May also manage heterogeneous flows

# Framework for IFP Systems: Time model

## ■ Time Model

- ◆ The ability of the IFP system of associating some kind of **happened-before relationship**
- ◆ This issue is **very relevant**, for IFP systems characterized by the **event notifications**.

# Framework for IFP Systems: Time model

## ■ Time Model

- ◆ The ability of the IFP system of associating some kind of **happened-before relationship**
- ◆ This issue is **very relevant**, for IFP systems characterized by the **event notifications**.

## ■ Scenarios

- ◆ **Stream-only**. Time Model not relevant: data flows into the engine within streams, but **time stamps** (when present) are used mainly to order items at the frontier of the engine and they **are lost during processing**.

# Framework for IFP Systems: Time model

## ■ Time Model

- ◆ The ability of the IFP system of associating some kind of **happened-before relationship**
- ◆ This issue is **very relevant**, for IFP systems characterized by the **event notifications**.

## ■ Scenarios

- ◆ **Stream-only**. Time Model not relevant: data flows into the engine within streams, but **time stamps** (when present) are used mainly to order items at the frontier of the engine and they are lost during processing.
- ◆ **Absolute-time**. Times stamps that represent an absolute time, usually interpreted as the **time of occurrence** of the related event. Time stamps are fully exposed to the rule language. (**buffering mechanism is required to cope with out-of-order arrivals**)

# Framework for IFP Systems: Time model

## ■ Scenarios

- ◆ **Stream-only**. Time Model not relevant: data flows into the engine within streams, but **time stamps** (when present) are used mainly to order items at the frontier of the engine and they are lost during processing.
- ◆ **Absolute-time**. Times stamps that represent an absolute time, usually interpreted as the **time of occurrence** of the related event. Time stamps are fully exposed to the rule language. (**buffering mechanism is required to cope with out-of-order arrivals**)
- ◆ **Label for partial order**. usually reflecting some kind of causal relationship, that is, the fact that the occurrence of an event was caused by the occurrence of another event

# Framework for IFP Systems: Time model

## ■ Scenarios

- ◆ **Stream-only**. Time Model not relevant: data flows into the engine within streams, but **time stamps** (when present) are used mainly to order items at the frontier of the engine and they are lost during processing.
- ◆ **Absolute-time**. Times stamps that represent an absolute time, usually interpreted as the **time of occurrence** of the related event. Time stamps are fully exposed to the rule language. (**buffering mechanism is required to cope with out-of-order arrivals**)
- ◆ **Label for partial order**. usually reflecting some kind of causal relationship, that is, the fact that the occurrence of an event was caused by the occurrence of another event
- ◆ **Interval**. two time stamps taken from a global time, usually representing the time when the related event started and the time when it ended

# Framework for IFP Systems: Rule model

## ■ Rule Model

- ◆ Rules are much more complex entities than data.
- ◆ Many different approaches to represent rules, which depend on the adopted rule language. However, we can roughly classify them into two macro classes: **transforming rules and detecting rules**.

# Framework for IFP Systems: Rule model

## ■ Rule Model

- ◆ Rules are much more complex entities than data.
- ◆ Many different approaches to represent rules, which depend on the adopted rule language. However, we can roughly classify them into two macro classes: **transforming rules and detecting rules**.

## ■ **Transforming rules** define an execution plan composed of **primitive operators** connected to each other in a graph.

# Framework for IFP Systems: Rule model

## ■ Rule Model

- ◆ Rules are much more complex entities than data.
- ◆ Many different approaches to represent rules, which depend on the adopted rule language. However, we can roughly classify them into two macro classes: **transforming rules and detecting rules**.

## ■ **Transforming rules** define an execution plan composed of **primitive operators**

connected to each other in a graph.

- ◆ Each operator takes several flows of information items as inputs and produces new items, which can be forwarded to other operators or directly sent out to sinks

# Framework for IFP Systems: Rule model

## ■ Rule Model

- ◆ Rules are much more complex entities than data.
- ◆ Many different approaches to represent rules, which depend on the adopted rule language. However, we can roughly classify them into two macro classes: **transforming rules and detecting rules**.

## ■ **Transforming rules** define an execution plan composed of **primitive operators**

connected to each other in a graph.

- ◆ Each operator takes several flows of information items as inputs and produces new items, which can be forwarded to other operators or directly sent out to sinks
- ◆ Execution plan can be either user-defined (graph based) or compiled (SQL-like)

# Framework for IFP Systems: Rule model

## ■ Rule Model

- ◆ Rules are much more complex entities than data.
- ◆ Many different approaches to represent rules, which depend on the adopted rule language. However, we can roughly classify them into two macro classes: **transforming rules and detecting rules**.

## ■ **Transforming rules** define an execution plan composed of **primitive operators**

connected to each other in a graph.

- ◆ Each operator takes several flows of information items as inputs and produces new items, which can be forwarded to other operators or directly sent out to sinks
- ◆ Execution plan can be either user-defined (graph based) or compiled (SQL-like)
- ◆ Transforming rules are often used with homogeneous information flows

# Framework for IFP Systems: Rule model

- **Detecting rules.** Are those that present an explicit distinction between a **condition** and an **action** part.

# Framework for IFP Systems: Rule model

- **Detecting rules.** Are those that present an explicit distinction between a **condition** and an **action** part.
  - ◆ **Condition:** logical predicate that captures **patterns of interest in the sequence** of information items

# Framework for IFP Systems: Rule model

- **Detecting rules.** Are those that present an explicit distinction between a **condition** and an **action** part.
  - ◆ **Condition:** logical predicate that captures **patterns of interest in the sequence** of information items
  - ◆ **Action:** hoc constructs to define how relevant information has to be **processed and aggregated** to produce new information

# Framework for IFP Systems: Rule model

- **Detecting rules.** Are those that present an explicit distinction between a **condition** and an **action** part.
  - ◆ **Condition:** logical predicate that captures **patterns of interest in the sequence** of information items
  - ◆ **Action:** hoc constructs to define how relevant information has to be **processed and aggregated** to produce new information
- **Another issue:** the ability to deal with uncertainty

# Framework for IFP Systems: Rule model

- **Detecting rules.** Are those that present an explicit distinction between a **condition** and an **action** part.
  - ◆ **Condition:** logical predicate that captures **patterns of interest in the sequence** of information items
  - ◆ **Action:** hoc constructs to define how relevant information has to be **processed and aggregated** to produce new information
- **Another issue:** the ability to deal with uncertainty
  - ◆ Some systems allow for distinguishing between **deterministic** and **probabilistic rules**

# Framework for IFP Systems: Language model

## ■ Language Type

# Framework for IFP Systems: Language model

## ■ Language Type

- ◆ **Transforming languages**: define **transforming rules**, specifying one or more operations that **process the input flows** by filtering, joining, and aggregating received information to **produce one or more output flows**

# Framework for IFP Systems: Language model

## ■ Language Type

- ◆ **Transforming languages**: define **transforming rules**, specifying one or more operations that **process the input flows** by filtering, joining, and aggregating received information to **produce one or more output flows**
  - **Declarative languages**: express processing rules in a declarative way, that is, by specifying the expected results of the computation rather than the desired execution flow (usually derived from relational languages)

# Framework for IFP Systems: Language model

## ■ Language Type

- ◆ **Transforming languages**: define **transforming rules**, specifying one or more operations that **process the input flows** by filtering, joining, and aggregating received information to **produce one or more output flows**
  - **Declarative languages**: express processing rules in a declarative way, that is, by specifying the expected results of the computation rather than the desired execution flow (usually derived from relational languages)
  - **Imperative languages**: define rules in an imperative way by letting the user specify a plan of primitive operators which the information flows have to follow. Each **primitive operator defines a transformation over its input**. Usually, systems that adopt imperative languages offer **visual tools** to define rules.

# Framework for IFP Systems: Language model

## ■ Language Type

- ◆ **Transforming languages**: define **transforming rules**, specifying one or more ...
  - Declarative languages
  - Imperative languages

# Framework for IFP Systems: Language model

## ■ Language Type

- ◆ **Transforming languages**: define **transforming rules**, specifying one or more ...
  - Declarative languages
  - Imperative languages
- ◆ **Detecting or pattern-based languages**: define **detecting rules by separately** specifying the firing conditions and the actions to be taken when such conditions hold.  
**Conditions** are usually defined as **patterns** that select matching portions of the input flows using logical operators, content, and timing constraints.

# Framework for IFP Systems: Language model

## ■ Language Type

- ◆ **Transforming languages**: define **transforming rules**, specifying one or more ...
  - Declarative languages
  - Imperative languages
- ◆ **Detecting or pattern-based languages**: define **detecting rules by separately** specifying the firing conditions and the actions to be taken when such conditions hold.  
**Conditions** are usually defined as **patterns** that select matching portions of the input flows using logical operators, content, and timing constraints.
- ◆ **Actions** define how the selected items have to be combined to produce new information  
(This type of languages is common in CEP systems).

# Framework for IFP Systems: Language model

- Representative example for declarative languages:
  - ◆ **Continuous Query Language - CQL**, created within the Stream project and currently adopted by Oracle

```
Select IStream(*)  
From F1 [Rows 5], F2 [Rows 10]  
Where F1.A = F2.A
```

# Framework for IFP Systems: Language model

- Representative example for declarative languages:
  - ◆ **Continuous Query Language - CQL**, created within the Stream project and currently adopted by Oracle

```
Select IStream(*)  
From F1 [Rows 5], F2 [Rows 10]  
Where F1.A = F2.A
```

- ◆ This rule isolates the last five elements of flow F1 and the last ten elements of flow F2 (using the **stream-to-relation operator [Rows n]**)

# Framework for IFP Systems: Language model

- Representative example for declarative languages:
  - ◆ **Continuous Query Language - CQL**, created within the Stream project and currently adopted by Oracle

```
Select IStream(*)  
From F1 [Rows 5], F2 [Rows 10]  
Where F1.A = F2.A
```

- ◆ This rule isolates the last five elements of flow F1 and the last ten elements of flow F2 (using the **stream-to-relation operator [Rows n]**)
- ◆ Then combines all elements having a common attribute A (using the **relation-to-relation operator Where**),

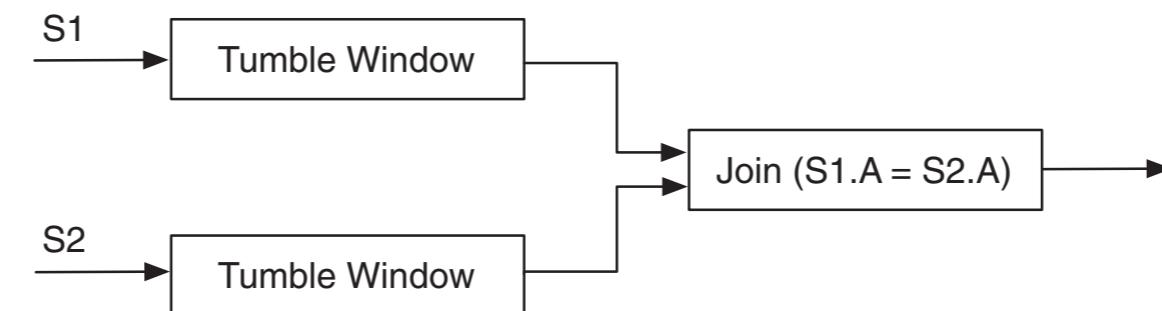
# Framework for IFP Systems: Language model

- Representative example for declarative languages:
  - ◆ **Continuous Query Language - CQL**, created within the Stream project and currently adopted by Oracle

```
Select IStream(*)  
From F1 [Rows 5], F2 [Rows 10]  
Where F1.A = F2.A
```
  - ◆ This rule isolates the last five elements of flow F1 and the last ten elements of flow F2 (using the **stream-to-relation operator** **[Rows n]**)
  - ◆ Then combines all elements having a common attribute A (using the **relation-to-relation operator** **Where**),
  - ◆ And produces a new flow with the created items (using the **relation-to-stream** operator **IStream**).

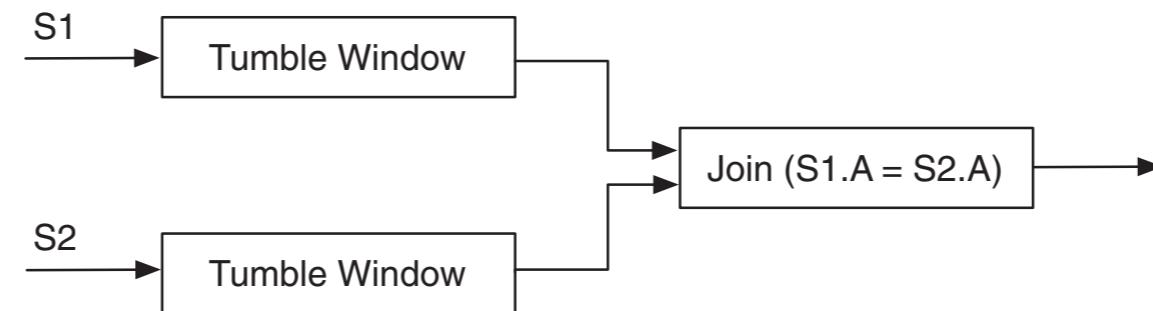
# Framework for IFP Systems: Language model

- Representative example for Imperative languages:
  - ◆ Aurora's Stream Query Algebra (SQuAI), which adopts a graphical representation called *boxes and arrows*



# Framework for IFP Systems: Language model

- Representative example for Imperative languages:
  - ◆ Aurora's Stream Query Algebra (SQuAI), which adopts a graphical representation called *boxes and arrows*



- ◆ The **tumble window** operator selects portions of the input stream according to given constraints, while **join** merges elements having the same value for attribute A.
- ◆ Similar to the previous example

# Framework for IFP Systems: Language model

- Representative example for Detecting languages:
  - ◆ **Subscription language of Padres**

A(X>0) & (B(Y=10);[timespan:5] C(Z<5)) [within:15] .

# Framework for IFP Systems: Language model

- Representative example for Detecting languages:

- ◆ **Subscription language of Padres**

A(X>0) & (B(Y=10);[timespan:5] C(Z<5)) [within:15] .

- ◆ A, B, and C represent item types or topics,

# Framework for IFP Systems: Language model

- Representative example for Detecting languages:

- ◆ **Subscription language of Padres**

A(X>0) & (B(Y=10);[timespan:5] C(Z<5)) [within:15] .

- ◆ A, B, and C represent item types or topics,
  - ◆ X, Y, and Z are inner fields of items. After the topic of an information item has been determined, filters on its content are applied,

# Framework for IFP Systems: Language model

- Representative example for Detecting languages:

- ◆ **Subscription language of Padres**

$A(X>0) \ \& \ (B(Y=10); [timespan:5] \ C(Z<5))_{[within:15]}.$

- ◆ A, B, and C represent item types or topics,
  - ◆ X, Y, and Z are inner fields of items. After the topic of an information item has been determined, filters on its content are applied,
  - ◆ The rule fires when an item of type A having an attribute  $X > 0$  enters the systems and also an item of type B with  $Y = 10$  is detected, followed (in a time interval of 5–15 s) by an item of type C with  $Z < 5$ .

# Framework for IFP Systems: Language model

## ■ Available Operators

- ◆ Single-Item Operators
- ◆ Logic Operators
- ◆ Sequences
- ◆ Iterations
- ◆ Windows
- ◆ Flow Management Operators
- ◆ Parameterization
- ◆ Flow creation
- ◆ Aggregates

# Framework for IFP Systems: Language model

- **Single-Item Operators:** those processing information items one by one. Two classes of single-item operators exist.
  - ◆ **Selection operators** - filter items according to their content, discarding elements that do not satisfy a given constraint (eg.g., only with temperature  $> 20^{\circ}\text{C}$ ).

# Framework for IFP Systems: Language model

- **Single-Item Operators:** those processing information items one by one. Two classes of single-item operators exist.
  - ◆ **Selection operators** - filter items according to their content, discarding elements that do not satisfy a given constraint (e.g., only with temperature  $> 20^{\circ}\text{C}$ ).
  - ◆ **Elaboration operators** - transform information items (e.g., converting from Celsius to Fahrenheit;
    - *projection* extracts only part of the information contained in an item).
    - *Renaming* - changes the name of a field in languages based on records.

# Framework for IFP Systems: Language model

- **Single-Item Operators:** those processing information items one by one. Two classes of single-item operators exist.
  - ◆ **Selection operators** - filter items according to their content, discarding elements that do not satisfy a given constraint (e.g., only with temperature  $> 20^{\circ}\text{C}$ ).
  - ◆ **Elaboration operators** - transform information items (e.g., converting from Celsius to Fahrenheit;
    - *projection* extracts only part of the information contained in an item).
    - *Renaming* - changes the name of a field in languages based on records.
- They are always present.

# Framework for IFP Systems: Language model

- **Logic Operators:** Are used to define rules that combine the **detection of several information items**. They are **order independent**.

# Framework for IFP Systems: Language model

- **Logic Operators:** Are used to define rules that combine the **detection of several information items**. They are **order independent**.
  - ◆ A ***conjunction*** of items  $I_1, I_2, \dots, I_n$  is satisfied when **all** the items  $I_1, I_2, \dots, I_n$  have been **detected**.

# Framework for IFP Systems: Language model

- **Logic Operators:** Are used to define rules that combine the **detection of several information items**. They are **order independent**.
  - ◆ A ***conjunction*** of items  $I_1, I_2, \dots, I_n$  is satisfied when **all** the items  $I_1, I_2, \dots, I_n$  have been **detected**.
  - ◆ A ***disjunction*** of items  $I_1, I_2, \dots, I_n$  is satisfied when **at least one** of the information items  $I_1, I_2, \dots, I_n$  has been **detected**.

# Framework for IFP Systems: Language model

- **Logic Operators:** Are used to define rules that combine the **detection of several information items**. They are **order independent**.
  - ◆ A **conjunction** of items  $I_1, I_2, \dots, I_n$  is satisfied when **all** the items  $I_1, I_2, \dots, I_n$  have been **detected**.
  - ◆ A **disjunction** of items  $I_1, I_2, \dots, I_n$  is satisfied when **at least one** of the information items  $I_1, I_2, \dots, I_n$  has been **detected**.
  - ◆ A **repetition** of an information item  $I$  of degree  $\langle m, n \rangle$  is satisfied when  $I$  is **detected at least  $m$  times** and not more than  $n$  times. (It is a special form of conjunction).

# Framework for IFP Systems: Language model

- **Logic Operators:** Are used to define rules that combine the **detection of several information items**. They are **order independent**.
  - ◆ A **conjunction** of items  $I_1, I_2, \dots, I_n$  is satisfied when **all** the items  $I_1, I_2, \dots, I_n$  have been **detected**.
  - ◆ A **disjunction** of items  $I_1, I_2, \dots, I_n$  is satisfied when **at least one** of the information items  $I_1, I_2, \dots, I_n$  has been **detected**.
  - ◆ A **repetition** of an information item  $I$  of degree  $\langle m, n \rangle$  is satisfied when  $I$  is **detected at least  $m$  times** and not more than  $n$  times. (It is a special form of conjunction).
  - ◆ A **negation** of an information item  $I$  is satisfied when  $I$  is **not detected**.

# Framework for IFP Systems: Language model

- **Logic Operators:** Are used to define rules that combine the **detection of several information items**. They are **order independent**.
  - ◆ A **conjunction** of items  $I_1, I_2, \dots, I_n$  is satisfied when **all** the items  $I_1, I_2, \dots, I_n$  have been **detected**.
  - ◆ A **disjunction** of items  $I_1, I_2, \dots, I_n$  is satisfied when **at least one** of the information items  $I_1, I_2, \dots, I_n$  has been **detected**.
  - ◆ A **repetition** of an information item  $I$  of degree  $\langle m, n \rangle$  is satisfied when  $I$  is **detected at least  $m$  times** and not more than  $n$  times. (It is a special form of conjunction).
  - ◆ A **negation** of an information item  $I$  is satisfied when  $I$  is **not detected**.
- The use of logic operators allows for the definition of expressions whose values **cannot be verified in a finite amount of time**, unless **explicit bounds** are defined for them

# Framework for IFP Systems: Language model

- The use of **logic operators** allows for the definition of expressions whose values cannot be verified in a finite amount of time, unless explicit bounds are defined for them.
  - ◆ **Repetition** and **negation**, which require elements to remain undetected in order to be satisfied. How long?

# Framework for IFP Systems: Language model

- The use of **logic operators** allows for the definition of expressions whose values cannot be verified in a finite amount of time, unless explicit bounds are defined for them.
  - ◆ Repetition and negation, which require elements to remain undetected in order to be satisfied. How long?
  - ◆ Existing systems combine these operators with other linguistic constructs known as windows.

# Framework for IFP Systems: Language model

- Logic operators are always present in pattern-based languages

(A & B) || (C & D)

# Framework for IFP Systems: Language model

- **Logic operators** are always present in **pattern-based languages**

(A & B) || (C & D)

- **Declarative and imperative languages** do not provide logic operators explicitly

- ◆ They usually allow conjunctions, disjunctions, and negations to be expressed using rules that transform input flows.

```
Select IStream(F1.A, F2.B)  
From F1 [Rows 50], F2 [Rows 50]
```

- ◆ the **From** clause specifies that we are interested in considering data from **both** flows F1 and F2.

# Framework for IFP Systems: Language model

- **Sequence Operators:** sequences are used to capture the arrival of a set of information items, but they **take into consideration the order of arrival**
  - ◆ Sequence defines an ordered set of information items  $I_1, I_2, \dots, I_n$ , which is satisfied when all the elements  $I_1, I_2, \dots, I_n$  have been detected in the specified order

# Framework for IFP Systems: Language model

- **Sequence Operators:** sequences are used to capture the arrival of a set of information items, but they **take into consideration the order of arrival**
  - ◆ Sequence defines an ordered set of information items  $I_1, I_2, \dots, I_n$ , which is satisfied when all the elements  $I_1, I_2, \dots, I_n$  have been detected in the specified order
- Sequence operator is **present** in many **pattern-based languages**

# Framework for IFP Systems: Language model

- **Sequence Operators:** sequences are used to capture the arrival of a set of information items, but they **take into consideration the order of arrival**
  - ◆ Sequence defines an ordered set of information items  $I_1, I_2, \dots, I_n$ , which is satisfied when all the elements  $I_1, I_2, \dots, I_n$  have been detected in the specified order
- Sequence operator is **present** in many **pattern-based languages**
- **Transforming languages** usually do **not provide it explicitly**

# Framework for IFP Systems: Language model

- **Sequence Operators:** sequences are used to capture the arrival of a set of information items, but they **take into consideration the order of arrival**
  - ◆ Sequence defines an ordered set of information items  $I_1, I_2, \dots, I_n$ , which is satisfied when all the elements  $I_1, I_2, \dots, I_n$  have been detected in the specified order
- Sequence operator is **present** in many **pattern-based languages**
- **Transforming languages** usually do **not provide it explicitly**
  - ◆ when the ordering relation is based on a time stamp field explicitly added to information items

```
Select IStream(F1.A, F2.B)
From F1 [Rows 50], F2 [Rows 50]
Where F1.time stamp < F2.time stamp
```

# Framework for IFP Systems: Language model

- **Iterations:** express possibly unbounded sequences of information items satisfying a given iterating condition.
  - ◆ Like sequences, iterations rely on the ordering of items.
  - ◆ However, they do not define the set of items to be captured explicitly but, rather, implicitly using the iterating condition.

# Framework for IFP Systems: Language model

- **Iterations:** express possibly unbounded sequences of information items satisfying a given iterating condition.
  - ◆ Like sequences, iterations rely on the ordering of items.
  - ◆ However, they do not define the set of items to be captured explicitly but, rather, implicitly using the iterating condition.
- An iteration example written in the Sase+ language:

```
PATTERN SEQ(Alert a, Shipment+ b[ ])
WHERE skip_till_any_match(a, b[ ]) {
    a.type = 'contaminated' and
    b[1].from = a.site and
    b[i].from = b[i-1].to }
WITHIN 3 hours
```

# Framework for IFP Systems: Language model

- An iteration example written in the Sase+ language:

```
PATTERN SEQ(Alert a, Shipment+ b[ ])
WHERE skip_till_any_match(a, b[ ]) {
    a.type = 'contaminated' and
    b[1].from = a.site and
    b[i].from = b[i-1].to }
WITHIN 3 hours
```

# Framework for IFP Systems: Language model

- An iteration example written in the Sase+ language:

```
PATTERN SEQ(Alert a, Shipment+ b[ ])
WHERE skip_till_any_match(a, b[ ]) {
    a.type = 'contaminated' and
    b[1].from = a.site and
    b[i].from = b[i-1].to }
WITHIN 3 hours
```

- It captures an **Alert** for a contaminated site (**item a**) and reports all possible series of infected **Shipments** (**items b[i]**)

# Framework for IFP Systems: Language model

- An iteration example written in the Sase+ language:

```
PATTERN SEQ(Alert a, Shipment+ b[ ])
WHERE skip_till_any_match(a, b[ ]) {
    a.type = 'contaminated' and
    b[1].from = a.site and
    b[i].from = b[i-1].to }
WITHIN 3 hours
```

- It captures an **Alert** for a contaminated site (**item a**) and reports all possible series of infected **Shipments** (**items b[i]**)
- **Pattern:** An **Alert** followed by a **sequence** of one or more **Shipments**

# Framework for IFP Systems: Language model

- An iteration example written in the Sase+ language:

```
PATTERN SEQ(Alert a, Shipment+ b[ ])
WHERE skip_till_any_match(a, b[ ]) {
    a.type = 'contaminated' and
    b[1].from = a.site and
    b[i].from = b[i-1].to }
WITHIN 3 hours
```

- It captures an **Alert** for a contaminated site (**item a**) and reports all possible series of infected **Shipments** (**items b[i]**)
- **Pattern:** An **Alert** followed by a **sequence** of one or more **Shipments**
  - ◆ Where
    - The type of alert is 'contaminated'
    - The first shipment's departure is the contaminated site and the next shipment

# Framework for IFP Systems: Language model

- An iteration example written in the Sase+ language:

```
PATTERN SEQ(Alert a, Shipment+ b[ ])
WHERE skip_till_any_match(a, b[ ]) {
    a.type = 'contaminated' and
    b[1].from = a.site and
    b[i].from = b[i-1].to }
WITHIN 3 hours
```

# Framework for IFP Systems: Language model

- An iteration example written in the Sase+ language:

```
PATTERN SEQ(Alert a, Shipment+ b[ ])
WHERE skip_till_any_match(a, b[ ]) {
    a.type = 'contaminated' and
    b[1].from = a.site and
    b[i].from = b[i-1].to }
WITHIN 3 hours
```

- Iteration is expressed using the **+ operator**, defining sequences of one or more Shipment information items.

# Framework for IFP Systems: Language model

- An iteration example written in the Sase+ language:

```
PATTERN SEQ(Alert a, Shipment+ b[ ])
WHERE skip_till_any_match(a, b[ ]) {
    a.type = 'contaminated' and
    b[1].from = a.site and
    b[i].from = b[i-1].to }
WITHIN 3 hours
```

- Iteration is expressed using the **+ operator**, defining sequences of one or more Shipment information items.
- The **iterating condition**  $b[i].from = b[i - 1].to$  specifies the collocation condition between each shipment and the preceding one.

# Framework for IFP Systems: Language model

- An iteration example written in the Sase+ language:

```
PATTERN SEQ(Alert a, Shipment+ b[ ])
WHERE skip_till_any_match(a, b[ ]) {
    a.type = 'contaminated' and
    b[1].from = a.site and
    b[i].from = b[i-1].to }
WITHIN 3 hours
```

- Iteration is expressed using the **+ operator**, defining sequences of one or more Shipment information items.
- The **iterating condition**  $b[i].from = b[i - 1].to$  specifies the collocation condition between each shipment and the preceding one.
- Shipment information items **need not be contiguous within the input flow**; intermediate items are simply discarded (*skip\_till\_any\_match*)

# Framework for IFP Systems: Language model

- Iteration is expressed using the **+ operator**, defining sequences of one or more Shipment information items.
- The **iterating condition**  $b[i].from = b[i - 1].to$  specifies the collocation condition between each shipment and the preceding one.
- Shipment information items **need not be contiguous within the input flow**; intermediate items are simply discarded (*skip\_till\_any\_match*)
- The **length** of captured **sequences** is **not known a priori** but depends on the actual number of shipments from site to site

```
PATTERN SEQ(Alert a, Shipment+ b[ ])
WHERE skip_till_any_match(a, b[ ]) {
    a.type = 'contaminated' and
    b[1].from = a.site and
    b[i].from = b[i-1].to }
    WITHIN 3 hours
```

# Framework for IFP Systems: Language model

- Iteration is expressed using the **+ operator**, defining sequences of one or more Shipment information items.
- The **iterating condition**  $b[i].from = b[i - 1].to$  specifies the collocation condition between each shipment and the preceding one.
- Shipment information items **need not be contiguous within the input flow**; intermediate items are simply discarded (*skip\_till\_any\_match*)
- The **length** of captured **sequences** is **not known a priori** but depends on the actual number of shipments from site to site
- To ensure the termination of pattern detection, a time bound is expressed using the **WITHIN** operator

```
PATTERN SEQ(Alert a, Shipment+ b[ ])
WHERE skip_till_any_match(a, b[ ]) {
    a.type = 'contaminated' and
    b[1].from = a.site and
    b[i].from = b[i-1].to }
WITHIN 3 hours
```

# Framework for IFP Systems: Language model

- **Iterations** are strictly related with the possibility for an IFP system to read its own output and to use it for **recursive processing**.
- In fact, if a system provides both a sequence operator and recursive processing, it can mimic iterations through recursive rules.

# Framework for IFP Systems: Language model

- **Windows.** It is often necessary to **define which portions of the input flows have to be considered** during the execution of operators.
- For this reason, almost all the languages used in existing systems define **windows**.
- **Windows** cannot be properly considered as operators; rather, they are language constructs that can be **applied to operators to limit the scope of their action**.

# Framework for IFP Systems: Language model

- **Windows.** It is often necessary to **define which portions of the input flows have to be considered** during the execution of operators.
- For this reason, almost all the languages used in existing systems define **windows**.
- **Windows** cannot be properly considered as operators; rather, they are language constructs that can be **applied to operators to limit the scope of their action**.
- Operators can be divided into two classes:
  - ◆ **blocking operators**, which need to read the whole input flows before producing results, and non-blocking operators, which can successfully return their results as items enter the system. Example: negation, repetitions (with an upper bound).

# Framework for IFP Systems: Language model

- **Windows.** It is often necessary to **define which portions of the input flows have to be considered** during the execution of operators.
- For this reason, almost all the languages used in existing systems define **windows**.
- **Windows** cannot be properly considered as operators; rather, they are language constructs that can be **applied to operators to limit the scope of their action**.
- Operators can be divided into two classes:
  - ◆ **blocking operators**, which need to read the whole input flows before producing results, and non-blocking operators, which can successfully return their results as items enter the system. Example: negation, repetitions (with an upper bound).
  - ◆ **non-blocking operators**. Example: conjunctions and disjunctions

# Framework for IFP Systems: Language model

- **Type of windows** (how they are defined)

# Framework for IFP Systems: Language model

- **Type of windows** (how they are defined)
  - ◆ Logical (or **time-based**): bounds are defined as a function of time
    - to force an operation to be computed only on the elements that arrived during the last five minutes.

# Framework for IFP Systems: Language model

## ■ **Type of windows** (how they are defined)

- ◆ Logical (or **time-based**): bounds are defined as a function of time
  - to force an operation to be computed only on the elements that arrived during the last five minutes.
- ◆ Physical (or **count-based**): bounds depend on the number of items included in the window:
  - to limit the scope of an operator to the last ten elements arrived.

# Framework for IFP Systems: Language model

## ■ Type of windows (how they are defined)

- ◆ Logical (or **time-based**): bounds are defined as a function of time
  - to force an operation to be computed only on the elements that arrived during the last five minutes.
- ◆ Physical (or **count-based**): bounds depend on the number of items included in the window:
  - to limit the scope of an operator to the last ten elements arrived.

## ■ Type of windows (the way their bounds move)

- ◆ Fixed windows
- ◆ Landmark windows
- ◆ Sliding windows
- ◆ Pane and tumble windows

# Framework for IFP Systems: Language model

- **Type of windows** (the way their bounds move)

# Framework for IFP Systems: Language model

## ■ **Type of windows** (the way their bounds move)

### ◆ **Fixed windows**: not move.



- They could be used to process the items received between 1/1/2010 and 31/1/2010.

# Framework for IFP Systems: Language model

## ■ **Type of windows** (the way their bounds move)

### ◆ **Fixed windows**: not move.



- They could be used to process the items received between 1/1/2010 and 31/1/2010.

### ◆ **Landmark windows**: have a fixed lower bound, while the upper bound advances every time a new information item enters the system.



- They could be used to process the items received since 1/1/2010.

# Framework for IFP Systems: Language model

## ■ Type of windows (the way their bounds move)

### ◆ Fixed windows: not move.



- They could be used to process the items received between 1/1/2010 and 31/1/2010.

### ◆ Landmark windows: have a fixed lower bound, while the upper bound advances every time a new information item enters the system.



- They could be used to process the items received since 1/1/2010.

### ◆ Sliding windows: They have a fixed size, that is, both lower and upper bounds advance when new items enter the system



# Framework for IFP Systems: Language model

## ■ Type of windows (the way their bounds move)

### ◆ Fixed windows: not move.



- They could be used to process the items received between 1/1/2010 and 31/1/2010.

### ◆ Landmark windows: have a fixed lower bound, while the upper bound advances every time a new information item enters the system.



- They could be used to process the items received since 1/1/2010.

### ◆ Sliding windows: They have a fixed size, that is, both lower and upper bounds advance when new items enter the system

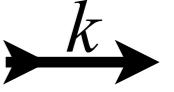


### ◆ Pane and tumble windows: variants of sliding windows, in which both the lower and the upper bounds move by k elements, as k elements enter the system

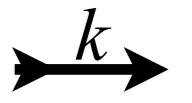


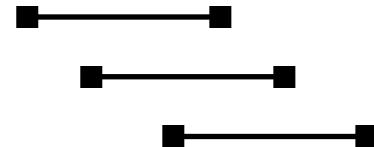
# Framework for IFP Systems: Language model

## ■ Type of windows (the way their bounds move)

- ◆ **Fixed windows**: not move. 
- ◆ **Landmark windows**: have a fixed lower bound, while the upper bound advances every time a new information item enters the system. 
- ◆ **Sliding windows**: They have a fixed size, that is, both lower and upper bounds advance when new items enter the system 
- ◆ **Pane and tumble windows**: variants of sliding windows, in which both the lower and the upper bounds move by  $k$  elements, as  $k$  elements enter the system 
  - **Pane** ( $\text{size} > k$ )
  - **Tumble** ( $\text{size} \leq k$ ): assures that every time the window is moved, all contained elements change; so each element of the input flow is processed at most once.

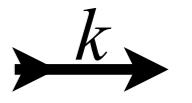
# Framework for IFP Systems: Language model

- **Type of windows** (the way their bounds move)
  - ◆ **Pane and tumble windows:** variants of sliding windows, in which both the lower and the upper bounds move by  $k$  elements, as  $k$  elements enter the system 
  - **Pane** ( $\text{size} > k$ ):
    - \* Calculate the average temperature in the last week, **every day** at noon



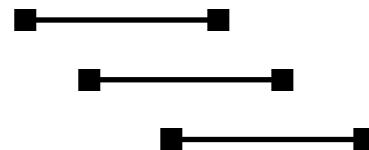
# Framework for IFP Systems: Language model

## ■ Type of windows (the way their bounds move)

- ◆ **Pane and tumble windows:** variants of sliding windows, in which both the lower and the upper bounds move by  $k$  elements, as  $k$  elements enter the system 

- **Pane** ( $\text{size} > k$ ):

- \* Calculate the average temperature in the last week, **every day** at noon



- **Tumble** ( $\text{size} \leq k$ ): assures that every time the window is moved, all contained elements change; so each element of the input flow is processed at most once.

- \* Calculate the average temperature in the last week, **every Sunday** at noon



# Framework for IFP Systems: Language model

## ■ Examples of windows:

- ◆ A **count-based, sliding window** over the flow F1 to count how many among the last 50 items received has  $A > 0$ . Results are streamed using the IStream operator

```
Select IStream(Count(*))
From F1 [Rows 50]
Where F1.A > 0
```

# Framework for IFP Systems: Language model

## ■ Examples of windows:

- ◆ A **count-based, sliding window** over the flow F1 to count how many among the last 50 items received has  $A > 0$ . Results are streamed using the IStream operator

```
Select IStream(Count(*))
From F1 [Rows 50]
Where F1.A > 0
```

- ◆ A time-base, **sliding window**. Does the same but considers the items received in the last minute.

```
Select IStream(Count(*))
From F1 [Range 1 Minute]
Where F1.A > 0
```

# Framework for IFP Systems: Language model

## ■ Examples of windows:

- ◆ A **count-based, sliding window** over the flow F1 to count how many among the last 50 items received has  $A > 0$ . Results are streamed using the IStream operator

```
Select IStream(Count(*))
From F1 [Rows 50]
Where F1.A > 0
```

- ◆ A time-base, **sliding window**. Does the same but considers the items received in the last minute.

```
Select IStream(Count(*))
From F1 [Range 1 Minute]
Where F1.A > 0
```

- Generally, **windows** are available in **declarative** and **imperative languages**.

Conversely, **only a few pattern-based** languages provide windowing constructs.

# Framework for IFP Systems: Language model

- Few languages that allow **users to define and use their own windows**.
- ESL provides *user-defined aggregates* to allow users to freely process input flows. In doing so, **users are allowed to explicitly manage the part of the flow they want to consider, that is, the window**

```
WINDOW AGGREGATE positive_min(Next Real): Real {  
    TABLE inwindow(wnext real);  
    INITIALIZE : { }  
    ITERATE : {  
        DELETE FROM inwindow  
        WHERE wnext < 0  
        INSERT INTO RETURN  
        SELECT min(wnext)  
        FROM inwindow  
    }  
    EXPIRE : { }  
}
```

the ITERATE clause removes every incoming element having a negative value and immediately returns the smallest value (using the INSERT INTO RETURN statement).

# Framework for IFP Systems: Language model

- **Flow Management Operators :** Declarative and imperative languages require ad hoc operators to **merge**, **split**, **organize**, and **process** flows of information:
  - ◆ **Join** operators
  - ◆ **Bag** operators
  - ◆ **Duplicate** operators
  - ◆ **Duplicate** operators
  - ◆ **Group by** operators
  - ◆ **Order by** operators

# Framework for IFP Systems: Language model

- **Flow Management Operators :** Declarative and imperative languages require ad hoc operators to **merge**, **split**, **organize**, and **process** flows of information:
- **Join** operators are used to merge two flows of information as in a traditional DBMS.  
Being a **blocking operator**, the join is usually applied to portions of the input flows which are processed as standard relational tables.

# Framework for IFP Systems: Language model

- **Flow Management Operators :** Declarative and imperative languages require ad hoc operators to **merge**, **split**, **organize**, and **process** flows of information:
- **Join** operators are used to merge two flows of information as in a traditional DBMS.  
Being a **blocking operator**, the join is usually applied to portions of the input flows which are processed as standard relational tables.
  - ◆ Example from CQL:

```
Select IStream(F1.A, F2.B)
From F1 [Rows 1000], F2 [Rows 1000]
Where F1.A = F2.A
```

# Framework for IFP Systems: Language model

- **Bag operators** combine different flows of information, considering them as bags of items:

# Framework for IFP Systems: Language model

- **Bag operators** combine different flows of information, considering them as bags of items:
  - ◆ **Union** merges two or more input flows of the same type, creating a new flow that includes all the items coming from them.

# Framework for IFP Systems: Language model

- **Bag operators** combine different flows of information, considering them as bags of items:
  - ◆ **Union** merges two or more input flows of the same type, creating a new flow that includes all the items coming from them.
  - ◆ **Except** takes two input flows of the same type and outputs all those items that belong to the first one but not to the second one. It is a **blocking operator**.

# Framework for IFP Systems: Language model

- **Bag operators** combine different flows of information, considering them as bags of items:
  - ◆ **Union** merges two or more input flows of the same type, creating a new flow that includes all the items coming from them.
  - ◆ **Except** takes two input flows of the same type and outputs all those items that belong to the first one but not to the second one. It is a **blocking operator**.
  - ◆ **Intersect** takes two or more input flows and outputs only the items included in all of them. It is a **blocking operator**.

# Framework for IFP Systems: Language model

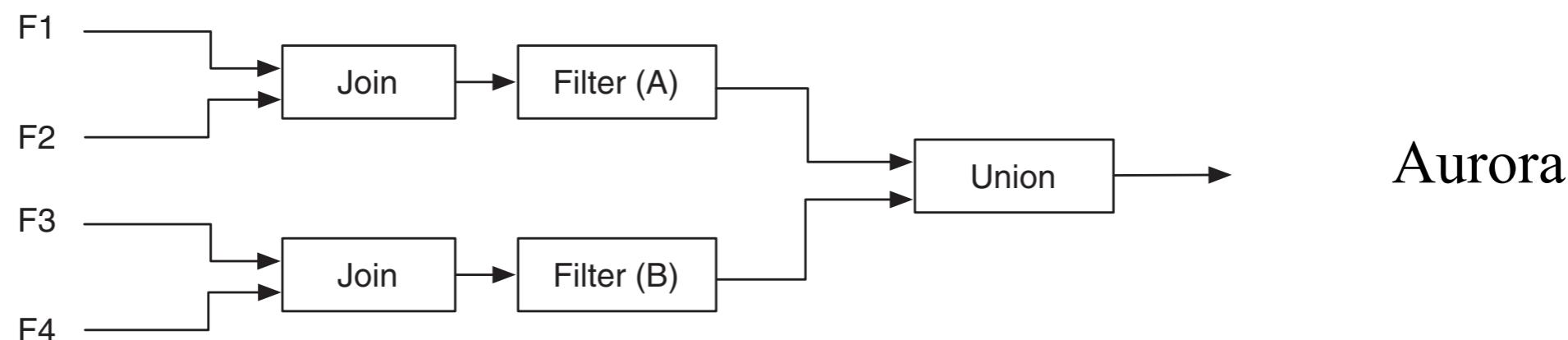
- **Bag operators** combine different flows of information, considering them as bags of items:
  - ◆ **Union** merges two or more input flows of the same type, creating a new flow that includes all the items coming from them.
  - ◆ **Except** takes two input flows of the same type and outputs all those items that belong to the first one but not to the second one. It is a **blocking operator**.
  - ◆ **Intersect** takes two or more input flows and outputs only the items included in all of them. It is a **blocking operator**.
  - ◆ **Remove-duplicate** removes all duplicates from an input flow.

# Framework for IFP Systems: Language model

## ■ Union example:

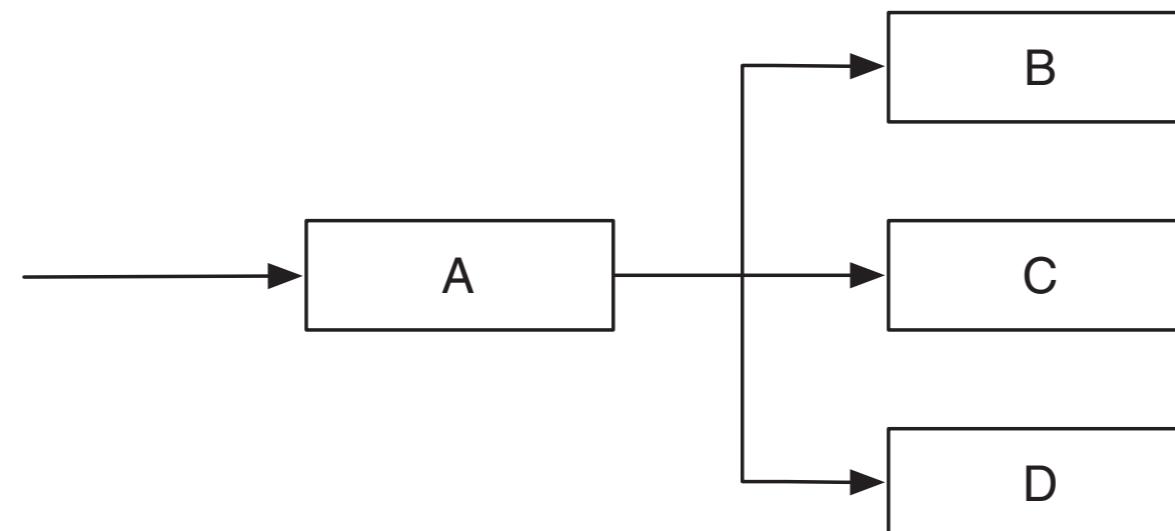
```
Select IStream(*)  
From F1 [Rows 1000], F2 [Rows 1000]  
Where F1.A = F2.A  
Union  
Select IStream(*)  
From F3 [Rows 1000], F4 [Rows 1000]  
Where F3.B = F4.B
```

CQL



# Framework for IFP Systems: Language model

- **Duplicate operators** allow a single flow to be duplicated in order to use it as an input for different processing chains
  - ◆ **Aurora** rule that takes a single flow of items, processes it through the operator A, then **duplicates** the result to have it processed by three operators B, C, and D, in parallel



# Framework for IFP Systems: Language model

- **Group-by operators** are used to split information flows into partitions in order to apply the same operator (usually an aggregate) to the different partitions.
  - ◆ CQL Example:

```
Select IStream(Count(*))  
From F1 [Rows 1000]  
Group By F1.B
```

# Framework for IFP Systems: Language model

- **Group-by operators** are used to split information flows into partitions in order to apply the same operator (usually an aggregate) to the different partitions.

- ◆ CQL Example:

```
Select IStream(Count(*))  
From F1 [Rows 1000]  
Group By F1.B
```

- **Order-by operators** are used to impose an ordering to the items of an input flow.
  - ◆ They are **blocking operators**, so they are usually applied to well-defined portions of input flows.

# Framework for IFP Systems: Language model

- **Parameterization:** In many IFP applications, it is necessary to filter some flows of information based on information that are part of other flows.
- Example: the fire protection system of a building could be interested in being notified when the temperature of a room exceeds 40°C but **only if some smoke has been detected in the same room.**

# Framework for IFP Systems: Language model

- **Parameterization:** In many IFP applications, it is necessary to filter some flows of information based on information that are part of other flows.
- Example: the fire protection system of a building could be interested in being notified when the temperature of a room exceeds 40°C but **only if some smoke has been detected in the same room.**
  - ◆ **Declarative and imperative languages** address this kind of situations by **joining** the two information flows

# Framework for IFP Systems: Language model

- **Parameterization:** In many IFP applications, it is necessary to filter some flows of information based on information that are part of other flows.
- Example: the fire protection system of a building could be interested in being notified when the temperature of a room exceeds 40°C but **only if some smoke has been detected in the same room.**
  - ◆ **Declarative and imperative languages** address this kind of situations by **joining** the two information flows
  - ◆ **Pattern-based languages** do not provide the join operator. This condition can be expressed only if the language allows filtering to be parametric.

(Smoke(Room=\$X)) & (Temp(Value>40 AND Room=\$X))      Padres

# Framework for IFP Systems: Language model

# Framework for IFP Systems: Language model

- **Flow creation:** operators for creating **new information flows from a set of items**. In particular, flow creation operators are used in **declarative languages** to address two issues:

# Framework for IFP Systems: Language model

- **Flow creation:** operators for creating **new information flows from a set of items**. In particular, flow creation operators are used in **declarative languages** to address two issues:
  - ◆ Some have been designed to deal indifferently with information flows and with relational tables a la DBMS. Flow creation operators can be used to **create new flows from existing tables**, for example, when new elements are added to the table.

# Framework for IFP Systems: Language model

- **Flow creation:** operators for creating **new information flows from a set of items**. In particular, flow creation operators are used in **declarative languages** to address two issues:
  - ◆ Some have been designed to deal indifferently with information flows and with relational tables a la DBMS. Flow creation operators can be used to **create new flows from existing tables**, for example, when new elements are added to the table.
  - ◆ Other languages use windows as a way of transforming (part of) an input flow into a table, which can be further processed by using the classical relational operators. Such languages use **flow creation operators to transform tables back into flows**.

# Framework for IFP Systems: Language model

- **Flow creation:**
- CQL provides three operators called *relation-to-stream* that allow for creating a new flow from a relational table T, at each evaluation cycle:
  - ◆ **(IStream)** streams all **new** elements added to T;  
Select IStream(\*)  
From F1 [Rows 10]
  - ◆ **(DStream)** streams all the elements **removed** from T;  
Select DStream(\*)  
From F1 [Rows 10]
  - ◆ **(Rstream)** streams **all** the elements of T at once,  
Select RStream(\*)  
From F1 [Rows 10]

# Framework for IFP Systems: Language model

- **Aggregates**: two kinds of aggregates
  - ◆ **Detection aggregates** are those used during the evaluation of the condition part of a rule. In our functional model, they are **computed and used by the decider**.

# Framework for IFP Systems: Language model

- **Aggregates**: two kinds of aggregates
  - ◆ **Detection aggregates** are those used during the evaluation of the condition part of a rule. In our functional model, they are **computed and used by the decider**.
    - Example: used in detecting all items whose value exceeds the average one (i.e., the aggregate), computed over the last ten received items.

# Framework for IFP Systems: Language model

- **Aggregates**: two kinds of aggregates
  - ◆ **Detection aggregates** are those used during the evaluation of the condition part of a rule. In our functional model, they are **computed and used by the decider**.
    - Example: used in detecting all items whose value exceeds the average one (i.e., the aggregate), computed over the last ten received items.
  - ◆ **Production aggregates** are those used to compute the values of information items in the output flow. In our functional model, they are **computed by the producer**.
    - Example: to output the average value among those part of the input flow.

# Framework for IFP Systems: Language model

- **Aggregates**: two kinds of aggregates
  - ◆ **Detection aggregates** are those used during the evaluation of the condition part of a rule. In our functional model, they are **computed and used by the decider**.
    - Example: used in detecting all items whose value exceeds the average one (i.e., the aggregate), computed over the last ten received items.
  - ◆ **Production aggregates** are those used to compute the values of information items in the output flow. In our functional model, they are **computed by the producer**.
    - Example: to output the average value among those part of the input flow.
- Almost all existing languages have **predefined aggregates**, which include minimum, maximum, and average

# Framework for IFP Systems: Language model

- **Aggregates**: two kinds of aggregates
  - ◆ **Detection aggregates** are those used during the evaluation of the condition part of a rule. In our functional model, they are **computed and used by the decider**.
    - Example: used in detecting all items whose value exceeds the average one (i.e., the aggregate), computed over the last ten received items.
  - ◆ **Production aggregates** are those used to compute the values of information items in the output flow. In our functional model, they are **computed by the producer**.
    - Example: to output the average value among those part of the input flow.
- Almost all existing languages have **predefined aggregates**, which include minimum, maximum, and average
- Some languages also offer facilities for creating user-defined aggregates (UDAs)

## Further Reading and Summary



Q&A

# Further Reading

## ■ Recommend Readings

- ◆ Processing Flows of Information: From Data Stream to Complex Event Processing,  
GIANPAOLO CUGOLA and ALESSANDRO MARGARA [Pages 1 - 25]

## ■ Supplemental readings:

# Further Reading and Summary



## Q&A