Advanced Databases

8

Active Databases

Active Databases

- A database that supports active rules (also called *triggers*)
- Outline:
  - Trigger definition in SQL:1999
  - Trigger definition in DB2 and Oracle
  - Design methods for trigger-based systems
  - Advanced features of triggers
  - Several examples

The Trigger Concept

- Paradigm: Event-Condition-Action
  - When an event occurs
  - If the condition is true
  - The action is executed
- The rule model offers an effective way for representing reactive computations
- Other examples of rules in the DBMS world:
  - Integrity constraints
  - Datalog rules
  - Business rules
- Problem: difficult to realize complex database systems using triggers well

Event-Condition-Action

- **Event**
  - **Normally a modification of the state of the database:** `insert`, `delete`, `update`
  - **When the event occurs, the trigger is *activated***
- **Condition**
  - **A predicate identifying the situations in which the application of a trigger is necessary**
  - **When the condition is evaluated, the trigger is *considered***
- **Action**
  - **A generic update statement or a stored procedure**
  - **When the action is elaborated, the trigger is *executed***
- **DBMSs already provide all the required components. One only needs to integrate them**

Triggers in SQL:1999, Syntax

- SQL:1999 (SQL-3) was strongly influenced by DB2 (IBM); the other systems do not fully comply (as they exist since the mid eighties).
- Each trigger is characterized by:
  - name
  - name of the target (monitored) table
  - execution mode (**`before`** or **`after`**)
  - monitored event (**`insert`**, **`delete`** or **`update`**)
  - granularity (statement-level or row-level)
  - names and aliases for transition values and transition tables
  - action
  - creation timestamp

Triggers in SQL:1999, Syntax

```
create trigger TriggerName
   < before | after >
   < insert | delete | update [of Column] > on Table
   [referencing
        <[old table [as] OldTableAlias]
         [new table [as] NewTableAlias] > |
        <[old [row] [as] OldTupleName]
         [new [row] [as] NewTupleName] >]
   [for each < row | statement >]
   [when Condition]
   SQLStatements
```

Kinds of events

- BEFORE
  - The trigger is considered and possibly executed before the event (i.e., the database change)
  - Before triggers cannot change the database state; at most they can change ("condition") the transition variables in row-level mode (set t.new=expr)
  - Normally this mode is used when one wants to check a modification before it takes place, and possibly make a change to the modification itself.
- AFTER
  - The trigger is considered and possibly executed after the event
  - It is the most common mode, suitable for most applications.

Granularity of events

- Statement-level mode (default mode, **`for each statement`** option)
  - The trigger is considered and possibly executed only once for each statement that activated it, independently of the number of modified tuples
  - Closer to the traditional approach of SQL statements, which normally are set-oriented
- Row-level mode (**`for each row`** option)
  - The trigger is considered and possibly executed once for each tuple modified by the statement
  - Writing of row-level triggers is simpler

The `referencing` clause

- The format depends on the granularity
  - For row-level mode, there are two *transition variables* (**`old`** and **`new`**) that represent the value either prior to or following the modification of the tuple under consideration
  - For statement-level mode, there are two *transition tables* (**`old table`** and **`new table`**) that contain either the old or the new value of all modified tuples
- Variables **`old`** and **`old table`** can't be used with triggers whose event is **`insert`**
- Variables **`new`** and **`new table`** can't be used with triggers whose event is **`delete`**
- Transition variables and transition tables enable tracking the changes that activate a trigger

Example of a Row-level Trigger

```
create trigger AccountMonitor
   after update on Account
   for each row
   when new.Total > old.Total
   insert values
        (new.AccNumber,new.Total-old.Total)
        into Payments
```

Example of a Statement-level Trigger

```
create trigger FileDeletedInvoices
   after delete on Invoice
   referencing old_table as OldInvoiceSet
   insert into DeletedInvoices
   (select *
    from OldInvoiceSet)
```

Execution of Multiple Triggers: Conflicts between Triggers

- If several triggers are associated to the same event, SQL:1999 prescribes the following policy
  - BEFORE triggers (statement-level and row-level) are executed
  - The modification is applied and the integrity constraints defined on the DB are checked
  - AFTER triggers (row-level and statement level) are executed
- If there are several triggers belonging to the same category, the order of execution chosen by the system is based upon their definition timestamp (older triggers have higher priority).

Recursive Execution Model

- SQL:1999 states that triggers are handled within a Trigger Execution Context (TEC)
- The execution of a trigger action may produce events that activate other triggers, which will have to be evaluated within a new, internal TEC.
  - At this point, the state of the enclosing TEC is saved, and the enclosed TEC is executed; this is a recursive process.
  - At the end of the enclosed TEC's execution, the state of the enclosing TEC is recovered and its execution is resumed.
- Trigger execution halts after a given recursion depth by rising a "nontermination exception"
- Any failure during a chain of trigger activated by a given statement S causes the partial rollback of S and of all changes due to the triggers of the chain.

Example: Salary Management

Employee

RegNum	Name	Salary	DeptN	ProjN
50	Smith	59.000	1	20
51	Black	56.000	1	10
52	Jones	50.000	1	20

Department

DeptNum	MGRRegNum
1	50

Project

ProjNum	Objective
10	NO
20	NO

Example Trigger T1: Bonus

Event: `update` of `Objective` in `Project`

Condition: `Objective = 'YES'`

Action: Increase by 10% the salary of the employees involved in the project

```
CREATE TRIGGER Bonus
AFTER UPDATE OF Objective ON Project
FOR EACH ROW
WHEN NEW.Objective = 'YES'
BEGIN
  update Employee
       set Salary = Salary*1.10
       where ProjN = NEW.ProjNum;
END;
```

Example Trigger T2: CheckIncrement

Event: `update` of `Salary` in `Employee`
Condition: New salary greater than manager's salary
Action: Decrease salary and make it the same as the manager's

```
CREATE TRIGGER CheckIncrement
AFTER UPDATE OF Salary ON Employee
FOR EACH ROW DECLARE X number;
BEGIN  SELECT Salary into X
       FROM Employee JOIN Department
       ON Employe.RegNum = Department.MGRRegNum
       WHERE Department.DeptNum = NEW.DeptN;
       IF NEW.Salary > X
              update Employee set Salary = X
              where RegNum = NEW.RegNum;
       ENDIF;
END;
```

Example Trigger T3: CheckDecrement

Event: `update` of `Salary` in `Employee`
Condition: Decrement greater than 3%
Action: Decrement salary only by 3%

```
CREATE TRIGGER CheckDecrement
AFTER UPDATE OF Salary ON Employee
FOR EACH ROW
WHEN (NEW.Salary < OLD.Salary * 0.97)
BEGIN
   update Employee
   set Salary=OLD.Salary*0.97
   where RegNum = NEW.RegNum;
END;
```

Activation of T1

```
update Project
set Objective = 'yes' where ProgNum = 10
```

Event: update of the `Objective` attr. in `Project`

Cond.: true

Project	
ProjNum	**Objective**
10	yes
20	no

Action: increase Black's salary by 10%

RegNum	Name	Salary	DeptN	ProjN
50	Smith	59.000	1	20
51	Black	61.600	1	10
52	Jones	50.000	1	20

Activation of T2

Event: update of **`Salary`** in **`Employee`**

Condition: true (employee Black's salary is greater than manager Smith's salary)

Action: Black's salary is modified and set to Smith's salary

RegNum	Name	Salary	DeptN	ProjN
50	Smith	59.000	1	20
51	Black	59.000	1	10
52	Jones	50.000	1	20

- T2 is activated again – the condition is false
- T3 is activated

Activation of T3

Event: update of **`Salary`** in **`Employee`**

Condition: true (Black's salary was decreased by more than 3%)

Action: Black's salary is decreased by only 3%

RegNum	Name	Salary	DeptN	ProjN
50	Smith	59.000	1	20
51	Black	59.752	1	10
52	Jones	50.000	1	20

- T2 is activated again – the condition is true

Activation of T2

RegNum	Name	Salary	DeptN	ProjN
50	Smith	59.000	1	20
51	Black	59.000	1	10
52	Jones	50.000	1	20

Activation of T3

- The trigger condition is false
  - Salary was decreased by less than 3%
- Trigger activation has reached termination

Triggers in DB2

- They follow the SQL:1999 syntax and semantics
- Some examples:
- 1. “Conditioner” (acts before the update and integrity checking)

```
create trigger LimitaAumenti
before update of Salario on Impiegato
for each row
when (New.Salario > Old.Salario * 1.2)
set New.Salario = Old.Salario * 1.2
```

- 2. “Re-installer” (acts after the update)

```
create trigger LimitaAumenti
after update of Salario on Impiegato
for each row
when (New.Salario > Old.Salario * 1.2)
set New.Salario = Old.Salario * 1.2
```

Triggers in Oracle

- They follow a different syntax (multiple events allowed, no table variables, where clause only legal with row-level triggers)

create trigger *TriggerName*
{ **before** | **after** } event [, event [,event]]
[[**referencing**
[**old** [**row**] [**as**] *OldTupleName*]
[**new** [**row**] [**as**] *NewTupleName*]]
for each { **row** | **statement** } [**when** *Condition*]]
SQLStatements

Event ::= { **insert** | **delete** | **update** [**of** *Column*] } **on** *Table*

- They have also a rather different conflict semantics and no limitation on expressive power of before trigger's action.

Conflicts between Triggers in Oracle

- If several triggers are associated to the same event, ORACLE has the following policy:
  - BEFORE statement-level triggers are executed
  - BEFORE row-level triggers are executed
  - The modification is applied and the integrity constraints defined on the DB are checked
  - AFTER row-level triggers are executed
  - AFTER statement-level triggers are executed
- If there are several triggers belonging to the same category, the order of execution depends on the creation time of the trigger.
- "Mutating table exception": occurs when the chain of triggers activated by a before trigger T tries to change the state of T's target table. Forces a statement rollback.

Example of Trigger in Oracle

Event: `update` of AvailableQty in Warehouse
Condition: Quantity below treshold and no pending order
Action: Do the order

```
create trigger Reorder
after update of AvailableQty on Warehouse
when (new.AvailableQty < new.ThresholdQty)
for each row
 declare
  X number;
   select count(*) into X
   from PendingOrder
   where Part = new.Part;
  if X = 0
  then
   insert into PendingOrder
    values(new.Part,new.ReorderQty,sysdate)
  end if;
 end;
```

Design - Trigger Properties

- It's important to ensure that interferences among triggers and chain activations do not produce anomalies in the behavior of the system
- 3 classical properties
  - Termination: for any initial state and sequence of modifications, a final state is produced (no infinite activation cycles)
  - Confluence: triggers terminate and produce a unique final state, independent of the order in which triggers are executed
  - Determinism of observable behavior: triggers are confluent and produce the same sequence of messages
- Termination is by far the most important property

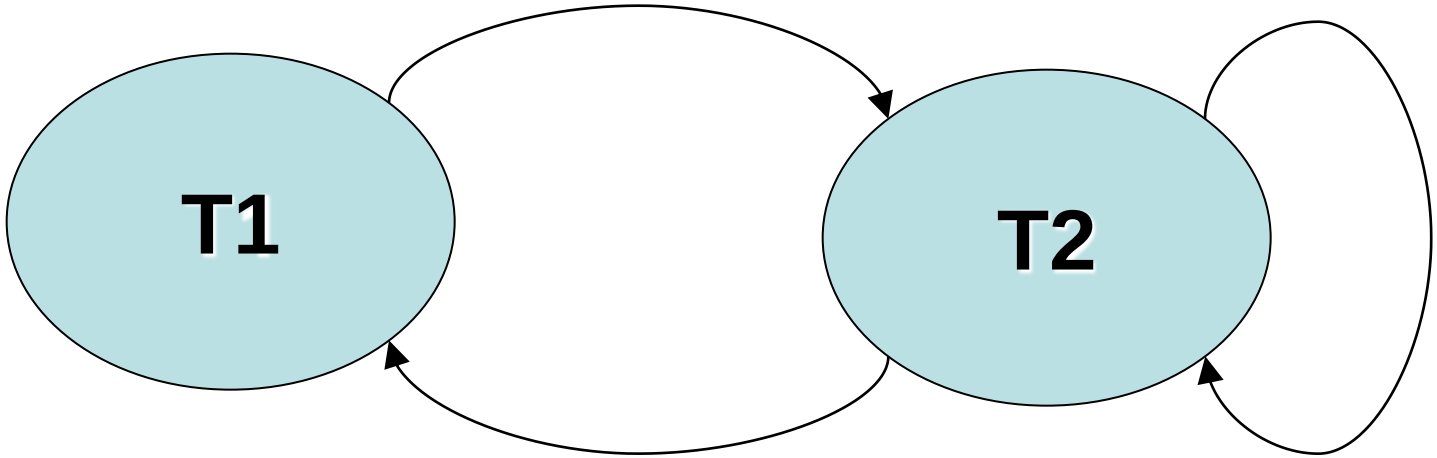
Termination Analysis

- There are several conceptual tools, most of which graph based
- The simplest is the triggering graph
  - A node for each trigger
  - An arc from a node t_i to a node t_j if the execution of t_i's action can activate trigger t_j (can be done with a simple syntactic analysis)
- If the graph is acyclic, the system is guaranteed to terminate
  - There cannot be infinite trigger sequences
- If the graph has some cycles, it *may* be non-terminating (but it might as well be terminating)

Example with Two Triggers

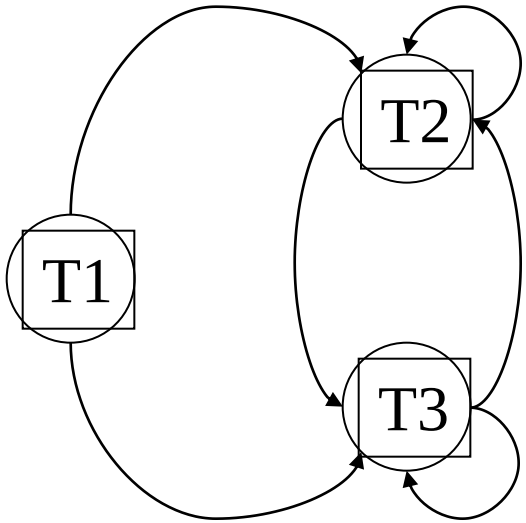
```
T1:    create trigger AdjustContributions
       after update of Salary on Employee
       referencing new table as NewEmp
       update Employee
       set Contribution = Salary * 0.8
       where RegNum in (   select RegNum
                           from NewEmp)

T2:    create trigger CheckBudgetThreshold
       after update on Employee
       when 50000 < (select sum(Salary+Contribution)
                     from Employee)
       update Employee
       set Salary = 0.9*Salary
```

Triggering Graph for Previous Triggers



- There are two cycles, but the system is terminating.
- To make it non-terminating, it suffices to reverse the direction of the comparison in T2's condition or to multiply by a factor greater than 1 in T2's action.

Termination Graph for the Salary Management Triggers



- The graph is cyclic, but the repeated execution of the triggers reaches termination anyway

Problems in Designing Trigger Applications

- Triggers add data management functions in a transparent and reusable manner
  - Such functions would otherwise be distributed over all applications
- But understanding the interactions between triggers is rather complex
- DBMS vendors use triggers to realize internal services, by introducing mechanisms for their automatic generation
  - Examples:
    - Constraint management
    - Data replication
    - View maintenance

Applications of Active Databases

- I**nternal** rules (generated by by the system and not visible by the user)
  - Integrity constraint management
  - Computation of derived and replicated data
  - Versioning management, privacy, security,
  - Action logging, event recording
- **External** rules (generated by the database administrator, they express application-specific knowledge)
  - Personalization, adaptation
  - Context-awareness
  - Business rules

Referential Integrity Management

- *Repair strategies* for violations of referential integrity constraints
  - The constraint is expressed as a predicate in the condition part

Ex:

```
CREATE TABLE Employee (
… …
FOREIGNKEY(DeptN) REFERENCES Department(DeptNum)
  ON DELETE SET NULL,
… …);
```

- Operations that can violate this constraint:
  - **INSERT** in Employee
  - **UPDATE** of Employee.DeptN
  - **UPDATE** of Department.DeptNum
  - **DELETE** in Department

Actions in the Employee Table

Event: insert in **Employee**
Condition: the new **DeptN** value is not among those in the **Dept** table
Action: insertion is inhibited, reporting error

```
CREATE TRIGGER CheckEmpDept
BEFORE INSERT ON Employee
FOR EACH ROW
WHEN (not exists select * from Department
        where DeptNum = NEW.DeptN)
BEGIN raise_application_error(-20000, 'Invalid
   Department'); END;
```

- For the update of **DeptN** in **Employee**, the trigger only changes in the event part

Deletion in the Department Table

Event: delete in **`Department`**
Condition: the deleted **`DeptNum`** is used in the **`Employee`** table
Action: Set null policy (the employee's dept is set to null)

```
CREATE TRIGGER CheckDeptDeletion
AFTER DELETE ON Department
FOR EACH ROW
WHEN (exists select * from Employee
        where DeptN = OLD.DeptNum)
BEGIN
        UPDATE Employee
        SET DeptN=NULL
        WHERE DeptN = OLD.DeptNum;
END;
```

(note: condition could be omitted)

Updates in the Department Table

Event: update of `DeptNum` in `Department`
Condition: the new `DeptNum` value is used in the `Employee` table
Action: cascade policy (`DeptN` in Employee is also modified)

```
CREATE TRIGGER CheckDeptUpdate
AFTER UPDATE OF DeptNum ON Department
FOR EACH ROW
WHEN (exists select * from Employee
        where DeptN = OLD.DeptNum)
BEGIN
        update Employee set DeptN = NEW.DeptNum
        where DeptN = OLD.DeptNum;
END;
```

(note: condition could be omitted)

Triggers for Materialized View Maintenance

- Consistency of views wrt tables on which they are defined
  - Base table updates must be propagated to views
- Replication management:

```
CREATE MATERIALIZED VIEW EmployeeReplica
REFRESH FAST AS
SELECT * FROM
DBMaster.Employee@mastersite.world;
```

- Materialized view maintenance is managed via triggers

Recursion Management

- Triggers for recursion management
  - Recursion not yet supported by current DBMSs
- Ex.: representation of a hierarchy of products
  - Each product is characterized by a **super-product** and by a depth **level** in the hierarchy
  - Can be represented by a recursive view (**`with recursive`** construct in SQL:1999)
  - Alternatively: use triggers to build and maintain the hierarchy

`Product(Code,Name,Description,SuperProduct,Level)`

- Hierarchy represented by **`SuperProduct`** and **`Level`**
- Products not contained in other products: **`SuperProduct=NULL`** and **`Level=0`**

Deletion of a Product

- In case of product deletion, all its sub-products must be deleted as well

```
CREATE TRIGGER DeleteProduct
    AFTER DELETE ON Product
    FOR EACH ROW
    BEGIN
     delete from Product
      where SuperProduct = OLD.Code;
    END;
```

Insertion of a New Product

- In case of an insertion, the appropriate Level value must be calculated

```
CREATE TRIGGER ProductLevel
AFTER INSERT ON Product FOR EACH ROW
BEGIN
  IF NEW.SuperProduct IS NOT NULL
    UPDATE Product
      SET Level = 1 + (select Level from Product
                       where Code = NEW.SuperProduct)
  ELSE
    UPDATE Product
      SET Level = 0
      WHERE Code = NEW.Code;
  ENDIF;
END;
```

Access Control

- Triggers can be used to strengthen access control
- It is convenient to define only those triggers that correspond to conditions that can’t be directly verified by the DBMS
- Using BEFORE gives the following advantages
  - Access control is executed before the trigger event is executed
  - Access control is executed only once and not for each tuple on which the trigger event is verified

ForbidSalaryUpdate Trigger

```
CREATE TRIGGER ForbidSalaryUpdate
BEFORE INSERT ON Employee
DECLARE
   not_weekend EXCEPTION; not_workingHours EXCEPTION;
BEGIN
/*if weekend*/
  IF (to_char(sysdate, 'dy') = 'SAT'
   OR to_char(sysdate, 'dy') = 'SUN')
  THEN RAISE not_weekend;
  END IF;
/*if outside working hours (8-18) */
  IF (to_char(sysdate, 'HH24') < 8
   OR to_char(sysdate, 'HH24') > 18)
  THEN RAISE not_workingHours;
  END IF;
```

ForbidSalaryUpdate Trigger (cont'd)

```
  EXCEPTION
  WHEN not_weekend
    THEN raise_application_error(-20324,'cannot
    modify Employee table during week-end');
  WHEN not_workingHours
    THEN raise_application_error(-20325,'cannot
    modify Employee table outside working
    hours');
END;
```

Evolution of active databases

- Execution modes (immediate, deferred, detached)
- Rule administration: priorities, grouping, dynamic activation and deactivation
- Instead-of clause
- New rule events (system-defined, temporal, user-defined)
- Complex events and event calculus
- The big news: streams.

Execution Modes

- The execution mode describes the connection between the activation (event) and the consideration and execution phases (condition and action)
- Condition and action are always evaluated together
- Normally the trigger is Immediate: considered and executed with the activating event
- Alternative execution modes:
  - Deferred: the trigger is handled at the end of the transaction
    - Example: triggers that check satisfaction of integrity constraints that require the execution of several operations
  - Detached: the trigger is handled in a separate transaction
    - Example: efficient management of variations of stock indices values after several exchanges

Priorities, Activations, and Groups

- Definition of priority
  - Allows specifying the execution order of triggers when there are several triggers activated at the same time
  - SQL:1999 states an order based on the execution mode and granularity; when these coincide, the choice depends on the implementation
- Activation/deactivation of triggers
  - Not in the standard, but often available
- Organization of triggers in groups
  - Some systems offer trigger grouping mechanisms, so as to activate/deactivate by groups

Instead of clause

- Alternative to `BEFORE` and `AFTER`
- Another operation than the one that activated the event is executed
- Very dangerous semantics (the application does one thing, the system does another thing)
- Implemented in several systems, often with strong limitations
  - In Oracle it can only be used for updates on views, so as to solve the view update problem when there is ambiguity

Extended Events/1

- System events and DDL commands
  - System: server-error, shutdown, etc.
  - DDL: authorization updates
  - In both cases some DBMSs already have these services that perform complex monitoring
- Temporal events (also periodical events)
  - Example: on July 23rd 2006 at 12, every day at 4
  - Useful for several applications
  - Difficult to integrate them because they are in an autonomous transactional context
  - They can be simulated via software components outside the DBMS that use time management services from the operating system

Extended Events/2

- "User-defined" events
  - Example: "TemperatureTooHigh"
  - Useful in some applications, but normally not offered
  - They too can be easily simulated
- Queries
  - Example: who reads the salaries
  - Normally too heavy to handle

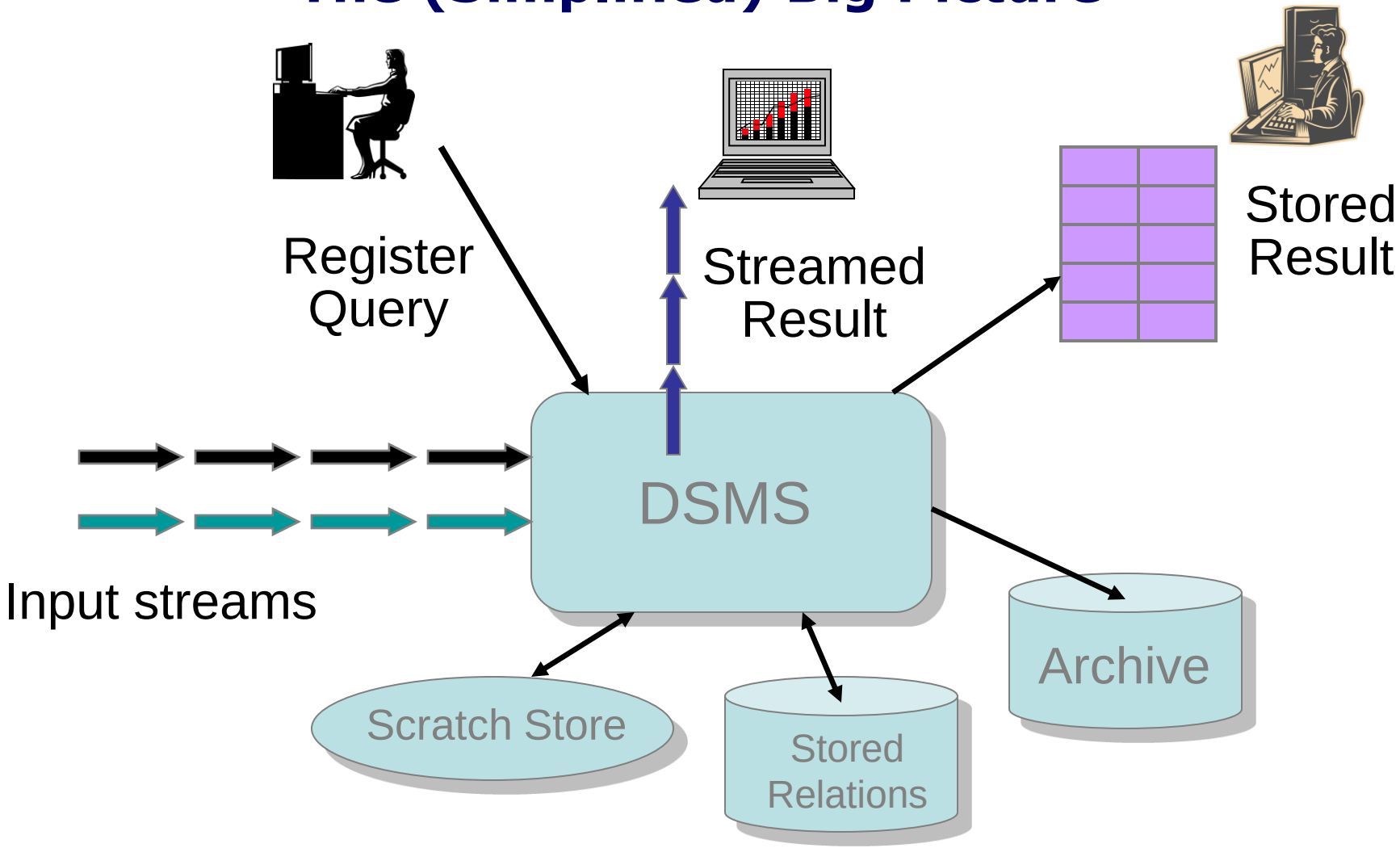
Event expressions

- Boolean combinations of events
  - SQL:1999 allows the specification of several events for a trigger, in disjunction
    - Any event among these is sufficient
  - Some researchers proposed more complex composition models
    - Very complex to handle
    - No strong motivation for introducing them

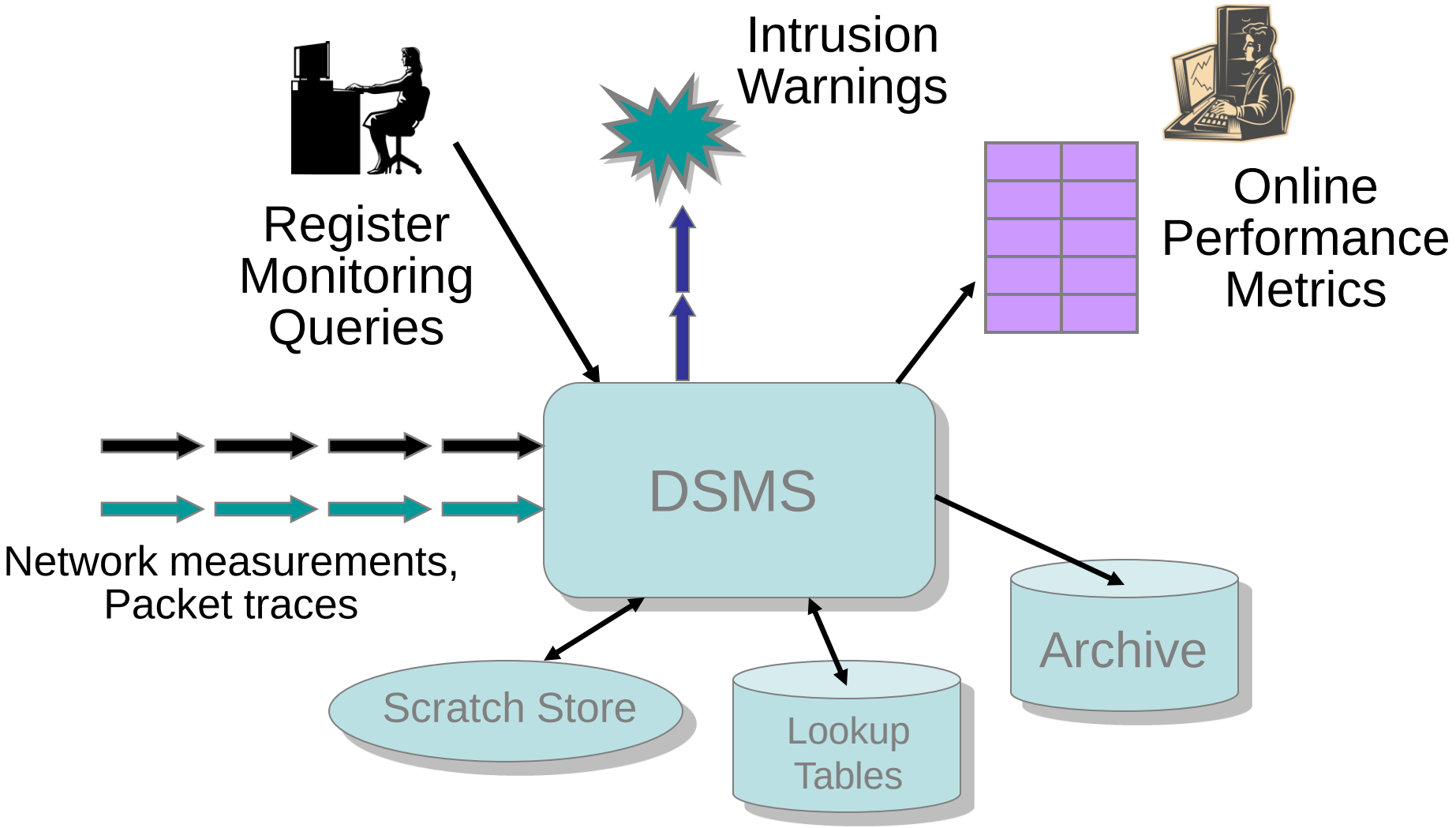
Data Streams

- Traditional DBMS -- data stored in finite, persistent data sets
- New applications -- data as multiple, continuous, rapid, time-varying data streams
  - Network monitoring and traffic engineering
  - Security applications
  - Telecom call records
  - Financial applications
  - Web logs and click-streams
  - Sensor networks
  - Manufacturing processes

The (Simplified) Big Picture



(Simplified) Network Monitoring



Active Databases

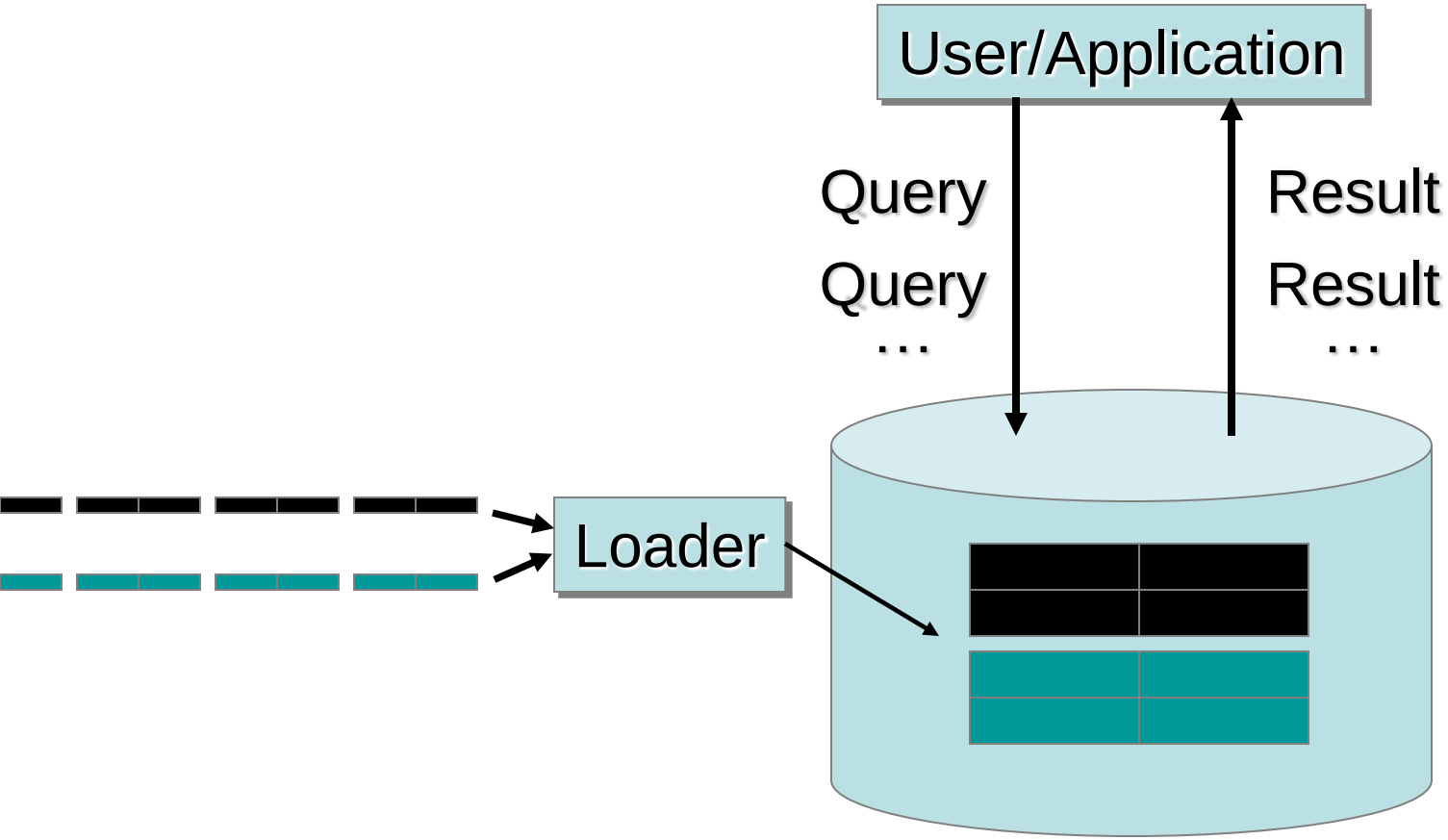
Why they are related to active databases?

- In an active database: the event is a query, the trigger system starts a reaction
- In a stream database: the event is a flow of data, the data stream system starts a reaction
  - Data streams enable the massive control of multiple data-driven events

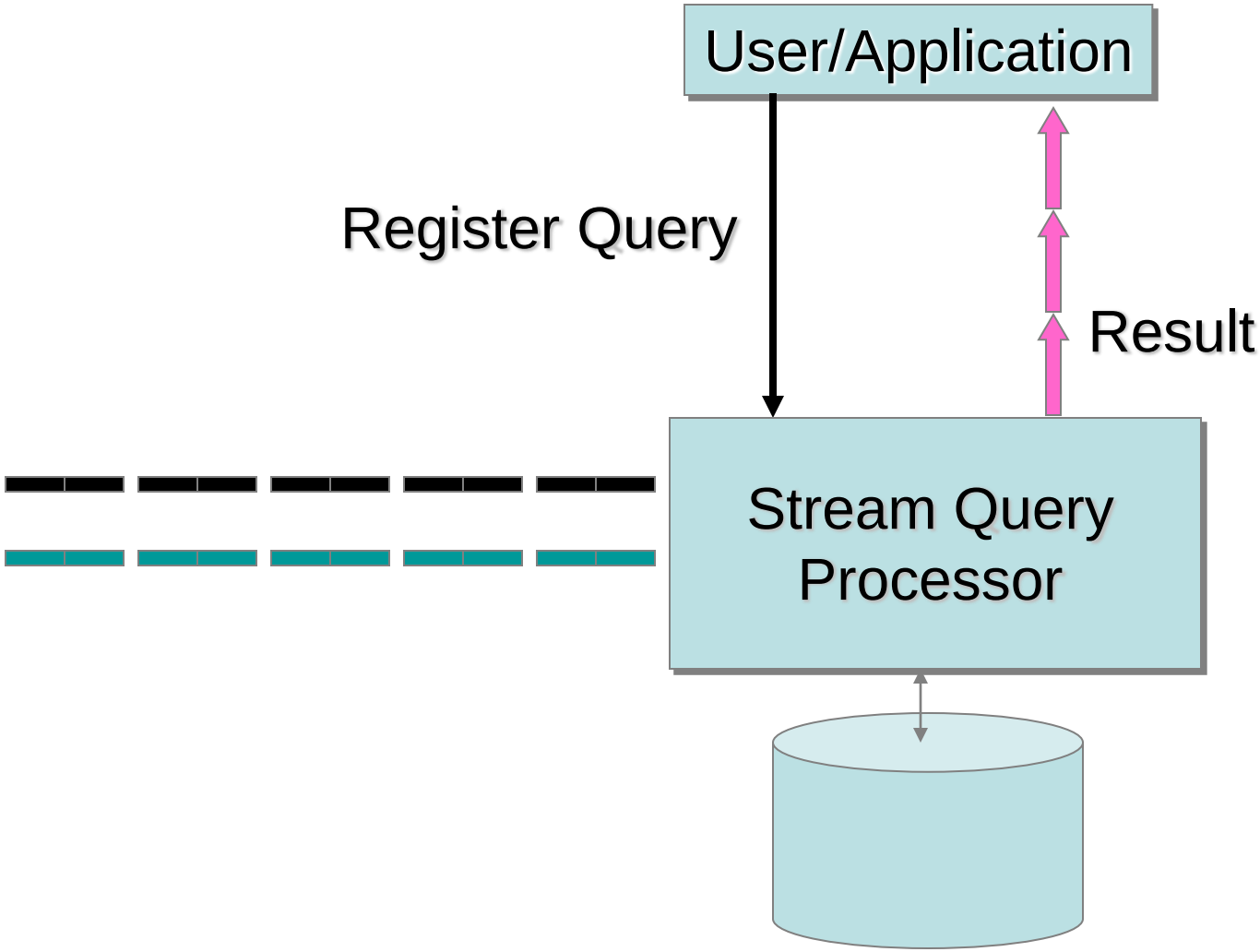
Challenges

- Multiple, continuous, rapid, time-varying streams of data
- Queries may be continuous (not just one-time)
  - Evaluated continuously as stream data arrives
  - Answer updated over time
- Queries may be complex
  - Beyond element-at-a-time processing
  - Beyond stream-at-a-time processing

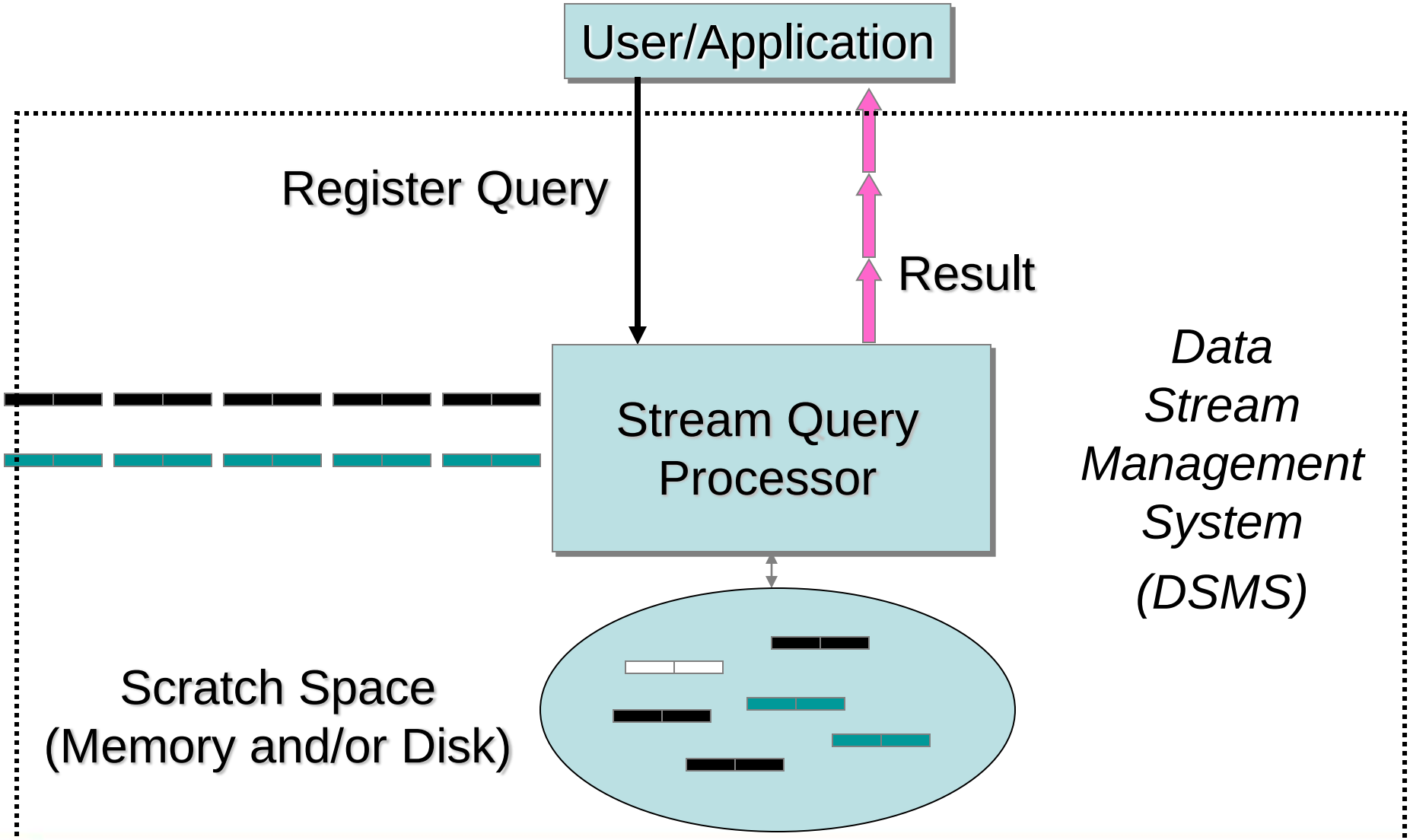
Using Traditional Databases



New Approach for Data Streams



New Approach for Data Streams



DBMS versus DSMS

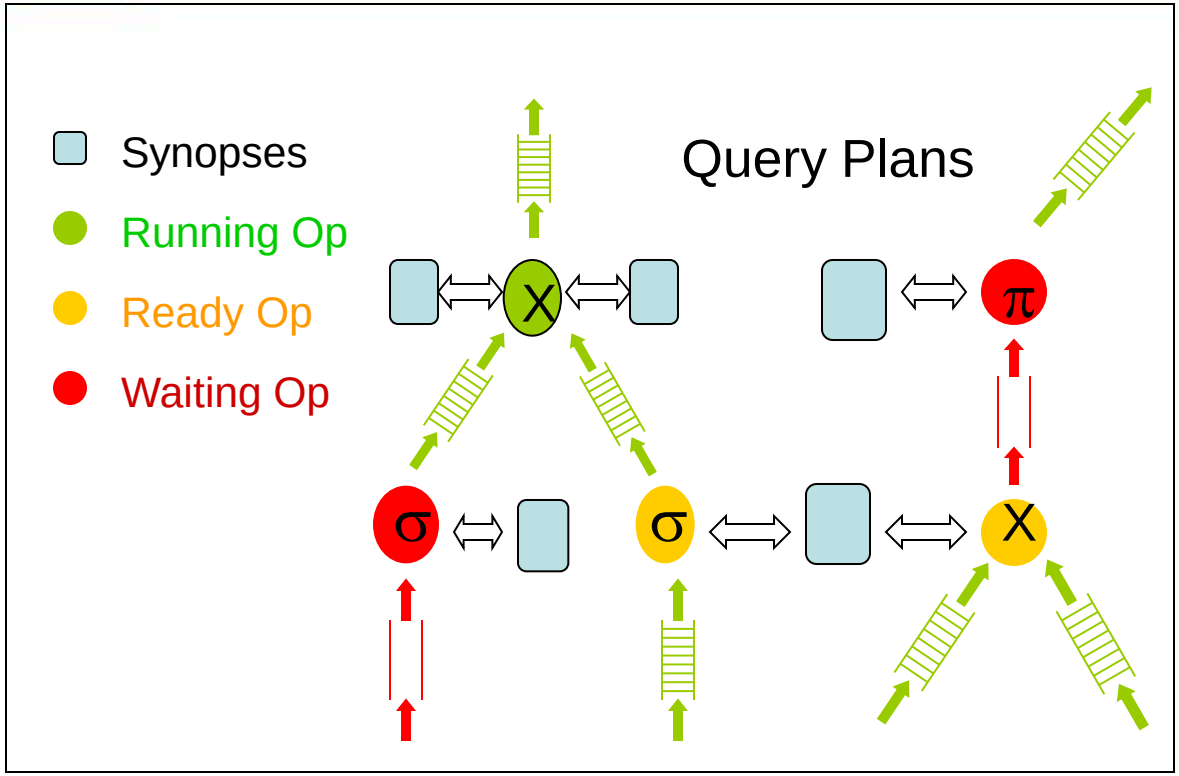
- Persistent relations
- One-time queries
- Random access
- Access plan determined by query processor and physical DB design
- "Unbounded" disk store

- Transient streams (and persistent relations)
- Continuous queries
- Sequential access
- Unpredictable data arrival and characteristics
- Bounded main memory

Data Stream Queries -- Basic Issues

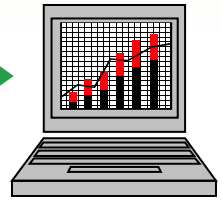
- Specifying queries over streams
  - SQL-like versus dataflow network of operators
- Answer availability
  - One-time
  - Multiple-time
  - Continuous (“standing”), stored or streamed
- Registration time
  - Predefined
  - Ad-hoc
- Semantic issues
  - Blocking operators, e.g., *aggregation, order-by*
  - Streams as sets versus lists

Query Processing Architecture



Input Data Streams

Output Stream



Applications register continuous queries

Users issue continuous and ad-hoc queries

Administrator can monitor query execution and adjust run-time parameters

Multiple Queries

- An engine can possibly support a large number of continuous queries
- Multi-query optimization: detecting common operations on common streams and factoring them
  - It is convenient, because the same stream is typically inspected according to many queries
  - It is possible, because queries are registered and rarely change

Operations on Streams

- Filtering streams (move tuples to different streams, remove certain tuples from the stream)
- Join streams (compare tuples from different streams - based upon matching conditions)
- Merging streams (according to time of arrival or logical conditions)
- Map streams (change tuples “on the fly” based upon value-based mappings)
- Compute stream aggregates (moving average, summation over given time intervals)
- Metronome operations: output data at given times
- Heartbeat operations: ouput data synchronized with given feed beats
- Managing disorder or late/missing data

Data window

- A portion of stream, used to put a boundary to (endless) streams.
- Windows can be defined by defining their starting/ending time (logical windos) or their total number of tuples (physical windows)
- Windows can be overlapping (incremented by a given step, smaller than the window size) or tumbling (the step is equal to the window size).

Query Evaluation -- Approximation

- Why approximate?
  - Streams are coming too fast
  - Exact answer requires unbounded storage or significant computational resources
  - Ad hoc queries reference history
- Issues in approximation
  - How is approximation controlled?
  - How is it understood by user?
- Accuracy-efficiency-storage tradeoff

Query Evaluation -- Adaptivity

- Why adaptivity?
  - Queries are long-running
  - Fluctuating stream arrival & data characteristics
  - Evolving query loads
- Issues in adaptivity
  - Adaptive resource allocation (memory, computation)
  - Adaptive query execution plans

StreamBase

- Product designed specifically for managing streams
- Proposes StreamSQL, an SQL extension with streams, windows, and tables as ingredients, and the major stream operations
  - Integrates stream processing with table processing
  - The company pushes the language standardization
- Has a powerful graphic environment for specifing dataflow of tuples and operations upon them
  - Claims very fast development of applications
- Provides partitioning by splitting to guarantee scalability of applications

Coral8

- Product designed specifically for managing streams
- Proposes CCL (Continuous Computation Language), also an SQL extension with streams, windows, and tables as ingredients, and the major stream operations
- Has an engine supporting complex event processing
- Has an integrated engine with IBM DB2, thus supporting native XML processing

Sample Applications

- Network management and traffic engineering (e.g., Sprint)
  - Streams of measurements and packet traces
  - Queries: detect anomalies, adjust routing
- Telecom call data (e.g., AT&T)
  - Streams of call records
  - Queries: fraud detection, customer call patterns, billing

Sample Applications (cont'd)

- Network security
  (e.g., iPolicy, NetForensics/Cisco, Netscreen)
  - Network packet streams, user session information
  - Queries: URL filtering, detecting intrusions & DOS attacks & viruses
- Financial applications
  (e.g., Traderbot)
  - Streams of trading data, stock tickers, news feeds
  - Queries: arbitrage opportunities, analytics, patterns

Sample Applications (cont'd)

- Web tracking and personalization (e.g., Yahoo, Google, Akamai)
  - Clickstreams, user query streams, log records
  - Queries: monitoring, analysis, personalization
- Truly massive databases (e.g., Astronomy Archives)
  - Stream the data once (or over and over)
  - Queries do the best they can