# Data bases 2

Silviu Filote

August 24, 2021

## Contents

# 1 Trigger

Struttura del trigger:

```
create trigger NomeTrigger
{before | after}
{ insert | delete | update [of Colonne] } on
Tabella
[referencing
      {[old table [as] AliasTabellaOld]
       [new table [as] AliasTabellaNew] } |
      {[old [row] [as] NomeTuplaOld]
       [new [row] [as] NomeTuplaNew] }]
[for each { row | statement }]
[when Condizione]
ComandiSQL
```

Figure 1: struttura del trigger

Modo di esecuzione:

- **BEFORE**, il trigger viene eseguito prima che avvenga la modifica sulla base dati ("*prima che succeda update/insert/delete fai questo*")
  - posso cambiare valore della tupla prima che venga inserita nel database

$$New.Punti = 30$$

```
create trigger ProductLevel_Before_childProduct
before insert into Product
for each row
when new.SuperProduct is not null
set new.Level = 1 + ( select Level
                      from Product
                      where Code = new.SuperProduct )
```

Quando il trigger viene azionato = condizione

Il valore che inserirò viene modificato

Figure 2: modifico il valore prima che questo debba essere inserito

- **AFTER**,, il trigger viene eseguito dopo che è stata modificata la base dati (”*dopo update/insert/delete fai questo*”)
  - in questo caso la modifica alla base dati è già avventa per riuscire ad individuare la tupla inserita procediamo in questa maniera:

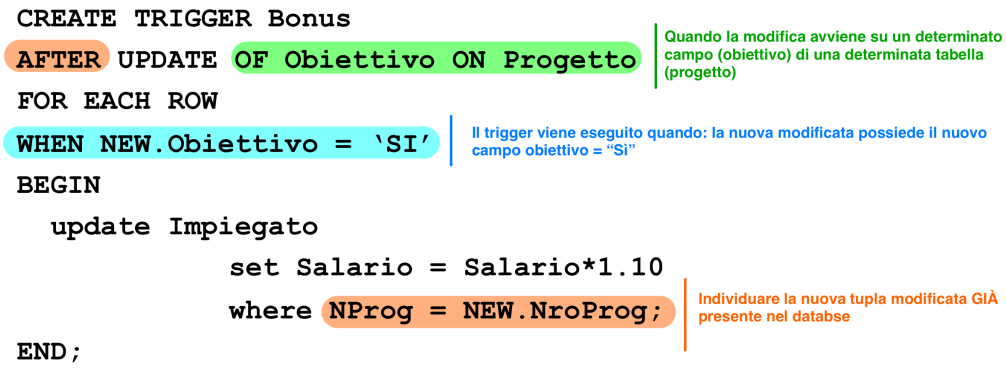



Figure 3: identifico la tupla attraverso NEW

Granularità degli eventi:

- **for each row**, per ogni tupla *insert, delete, update* viene considerato il trigger
  - si hanno due transition variables **old** e **new**, che rappresentano rispettivamente il valore precedente e successivo alla modifica della tupla che si sta valutando
  - **new, old** non sono puntatori, sono solo varialibili valore (simile ad uno struct)

| OLD-NEW table | | | | | | | |
|---|---|---|---|---|---|---|---|
| Stime | NO2 | NOx | PM10 | Pioggia | Temperatura | Umidità | Ozono |
| 1 | 5.3228 | 6.2275 | 6 | 0 | -0.89157 | 35.478 | 7.9571 |

- **for each statement**, vengono considerate tutte le tuple che sono state *insert, delete, update*
  - Per il modo statement-level, si hanno due transition **old_table e new_table**, che contengono rispettivamente il valore vecchio e nuovo di tutte le tuple modificate

```
Create trigger CalcolaVistaOneShot
after insert into VIAGGIO
for each statement
begin

   delete from VIAGGI;

   insert into VIAGGI
   select dip, sum( km ), sum( km * costo-km )
   from VIAGGIO join AUTO on ( targa-auto = targa )
   group by dip;

end
```

Elimino tutta la tabella

Poi la ricostruisco

Figure 4: for each statement

| OLD_TABLE e NEW_TABLE | | | | | | | |
|---|---|---|---|---|---|---|---|
| Stime | NO2 | NOx | PM10 | Pioggia | Temperatura | Umidità | Ozono |
| Min | 5.3228 | 6.2275 | 6 | 0 | -0.89157 | 35.478 | 7.9571 |
| Max | 66.274 | 173.02 | 71.143 | 156.4 | 29.54 | 98.408 | 153.06 |
| Mean | 21.412 | 33.942 | 21.422 | 22.623 | 14.814 | 66.32 | 66.555 |

- ***il NEW e OLD sono variabili valore e non puntatori, quello che si deve modificare é il valore che trovo tramite questi puntatori e non il putatore stesso***
- **ne posso modificare il valore *new.variabile* con il *before trigger*, in questo modo il valore che verrá inserito dopo il trigger saá quello modificato**
- La condizione “Code=new.Code” serve a individuare la tupla appena inserita (che deve essere modificata).
- “set new.Level = . . . ” sarebbe un grave errore! “new” non è un puntatore all’oggetto creato

Proprietà

- Conflitti tra trigger
  - Vengono eseguiti i trigger BEFORE statement-level
  - Vengono eseguiti i trigger BEFORE row-level
  - Si applica la modifica e si verificano i vincoli di integrità definiti sulla base di dati
  - Vengono eseguiti i trigg FTER row-level
  - Vengono eseguiti i trigg FTER statement-level
- Proprietà dei trigger
  - Terminazione ⇒ triggering graph
    - Se il grafo è aciclico si ha la garanzia che il sistema è terminante
    - Se il grafo ha dei ciclico , c'è la possibilità che il sistema sia nonterminante (ma non è detto)
  - Confluenza
  - Determinismo delle osservazi
- Nel corpo del trigger posso inserire perazioni multiple staccate, eseguo 3 update nello stesso trigger
- When è la condizione per la quale se viene soddisfatta viene eseguito il trigger
- la ricorsio viene generata quando **l'azione e l'evento sono uguali**
  - delete di un nodo specifico, e da li provoca la ricorsione

```
create trigger DeleteProduct
after delete on Product
for each row
delete from Product
where SuperProduct = old.Code
```

Come avviene la ricorsione: evento = azione

Figure 5: ricorsione

- si possono inoltre dichiarare variabili e dare a loro un valore tramite *M:=()*
- si possono inoltre fare dei controlli

```
Create trigger ValutaDomanda
after insert on DomandaBorsa
for each row
declare M, C number:
M := ( select avg(Voto) from Esame
       where  Matricola=new.Matricola and Data<= new.DataDomanda )
C := ( select sum(NumeroCrediti) from  Esame JOIN Corso ON
       Esame.CodCorso=Corso.CodiceCorso
       where  Matricola=new.Matricola and Data<= new.DataDomanda )
begin
if (M >= 27 and C >= 50)
 then (
       update DomandaBorsa set stato=”accolta” where Matricola=new.Matricola;
       insert into Graduatoria values (new.Matricola, M, C, NULL); )
 else
       update DomandaBorsa set stato=”respinta” where Matricola=new.Matricola;
end
```

Figure 6: esempio

```
SELECT
    id,
    avg(sal)
FROM
    StreamData
WHERE
    ...
GROUP BY
    id
HAVING
    avg(sal)>=10.0
    AND avg(sal)<=50.0
LIMIT 100
```

Figure 7: SQL schema

**Semantica**

- *AFTER/BEFORE insert **of** Colonna **into** tabella*
- *AFTER/BEFORE update **of** Colonna **on** tabella*
- *AFTER/BEFORE delete **of** Colonna **on** tabella*

**Predicati**

- *Exsists: se ritorna un risultato*
- *In: se é dentro la lista*
- *Not*
- *Rollback*
- *When 3 < (Select count(*))*

# 2 DTD

A DTD is a Document Type Definition.
A DTD defines the structure and the legal elements and attributes of an XML document.

Il DTD può essere dichiarato internamento al file XML stesso utilizzando la notazione

*<!DOCTYPE root-element [element-declarations]>*

```
<?xml version="1.0"?>
<!DOCTYPE note [
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
]>
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend</body>
</note>
```

Figure 8: DTD interno

Oppure esternamente utilizzando un'altro file

```
<?xml version="1.0"?>
<!DOCTYPE note SYSTEM "note.dtd">
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

(a) document must contain a reference to the DTD file

```
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
```

(b) And here is the file "note.dtd", which contains the DTD

## 2.1 Elements

*<!ELEMENT element-name category>*

| Category | Description |
|---|---|
| EMPTY | Empty Elements |
| (#PCDATA) | Character data elements |
| ANY | Any contents |

- Elements with one or more children, the children must appear in the same sequence in the document:

*<!ELEMENT element-name (child1,child2,...)>*

*<!ELEMENT note (to,from,heading,body)>*

```
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
```

Figure 9: I figli devono rispettare l'ordine

- Declaring Only One Occurrence of an Element:

*<!ELEMENT element-name (child-name)>*

- Declaring multiple occorrences of an element (minimum One Occurrence)

*<!ELEMENT element-name (child-name+)>*

- Can occur zero or more times

*<!ELEMENT element-name (child-name*)>*

- Can occur zero or one time

*<!ELEMENT element-name (child-name?)>*

- Either, or element

*<!ELEMENT note (to,from,header,(message|body))>*

- Declaring Mixed Content

*<!ELEMENT note (#PCDATA|to|from|header|message)*>*

| Occurrences | Description |
|---|---|
| child-name | Only One Occurrence of an Element |
| child-name + | multiple occorrences of an element |
| child-name * | Can occur zero or more times |
| child-name ? | Can occur zero or one time |

## 2.2 Attributes

*<!ATTLIST element-name attribute-name attribute-type attribute-value>*

*<!ATTLIST payment type CDATA "check">*

The **attribute-type** can be one of the following:

| Type | Description |
|---|---|
| CDATA | The value is character data |
| (*en1*\|*en2*\|..) | The value must be one from an enumerated list |
| ID | The value is a unique id |
| IDREF | The value is the id of another element |
| IDREFS | The value is a list of other ids |
| NMTOKEN | The value is a valid XML name |
| NMTOKENS | The value is a list of valid XML names |
| ENTITY | The value is an entity |
| ENTITIES | The value is a list of entities |
| NOTATION | The value is a name of a notation |
| xml: | The value is a predefined xml value |

The **attribute-value** can be one of the following:

| Value | Explanation |
|---|---|
| *value* | The default value of the attribute |
| #REQUIRED | The attribute is required |
| #IMPLIED | The attribute is optional |
| #FIXED *value* | The attribute value is fixed |

A Default Attribute Value:

```
<!ELEMENT square EMPTY>
<!ATTLIST square width CDATA "0">
```

# 3 XSD

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="note">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="to" type="xs:string"/>
      <xs:element name="from" type="xs:string"/>
      <xs:element name="heading" type="xs:string"/>
      <xs:element name="body" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

</xs:schema>
```

| Elements | Description |
|---|---|
| *<xs:complexType>* | contains other elements |
| *<xs:simpleType>* | can contains only text |

The syntax for defining a simple element is:

*<xs:element name=”xxx” type=”yyy”/>*

types:

- xs:string
- xs:decimal
- xs:integer
- xs:boolean
- xs:date
- xs:time

Simple elements may have a default value OR a fixed value specified

*<xs:element name=”color” type=”xs:string” default=”red”/>*

*<xs:element name=”color” type=”xs:string” fixed=”red”/>*

## 3.1 Attributes

The syntax for defining an attribute is: (types are the same as before)

*<xs:attribute name=”xxx” type=”yyy”/>*

Restrictions on Values, the value of age cannot be lower than 0 or greater than 120:

```
<xs:element name="age">
  <xs:simpleType>
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="120"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Restrictions on a Set of Values. Lista di valori possibili:

```
<xs:element name="car">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="Audi"/>
      <xs:enumeration value="Golf"/>
      <xs:enumeration value="BMW"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

(a) Metodi uguali

```
<xs:element name="car" type="carType"/>

<xs:simpleType name="carType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Audi"/>
    <xs:enumeration value="Golf"/>
    <xs:enumeration value="BMW"/>
  </xs:restriction>
</xs:simpleType>
```

(b) Metodi uguali

The only acceptable value is ONE of the LOWERCASE letters from a to z:

```
<xs:element name="letter">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="[a-z]"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

## 3.2 Complex Element

There are four kinds of complex elements:

- empty elements
- elements that contain only other elements
- elements that contain only text
- elements that contain both other elements and text

**Note:** Each of these elements may contain attributes as well!

```
<xs:element name="employee">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

The "employee" element can have a type attribute that refers to the name of the complex type to use:

```
<xs:element name="employee" type="personinfo"/>
<xs:element name="student" type="personinfo"/>
<xs:element name="member" type="personinfo"/>

<xs:complexType name="personinfo">
  <xs:sequence>
    <xs:element name="firstname" type="xs:string"/>
    <xs:element name="lastname" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

Mixed values:

```
<xs:element name="letter">
  <xs:complexType mixed="true">
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="orderid" type="xs:positiveInteger"/>
      <xs:element name="shipdate" type="xs:date"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

- Dopo *complexType* puó presentarsi uno dei seguenti
  - All Indicator ⇒ the child elements can appear in any order, and that each child element must occur only once

    $<xs:all>$
  - Choice Indicator ⇒ specifies that either one child element or another can occur

    $<xs:chioce>$
  - Sequence Indicator ⇒ specifies that the child elements must appear in a specific order

    $<xs:sequence>$
  - Any indicator ⇒ enables us to extend the XML document with elements not specified by the schema

    $<xs:any>$
  - Any attribute ⇒ enables us to extend the XML document with attributes not specified by the schema

    $<xs:anyAttribute>$
  - minOccurs Indicator and maxOccurs Indicator ⇒ indicator specifies the maximum/minimum number of times an element can occur

```
<xs:element name="person">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="full_name" type="xs:string"/>
      <xs:element name="child_name" type="xs:string"
      maxOccurs="10" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

# 4 XPath

Open the document: returns the root element and all its content.

*doc('name.xml')*

Selecting Nodes:

| Expression | Description |
|---|---|
| *nodename* | Selects all nodes with the name "*nodename*" |
| / | Selects from the root node |
| // | Selects nodes in the document from the current node that match the selection no matter where they are |
| . | Selects the current node |
| .. | Selects the parent of the current node |
| @ | Selects attributes |

| Path Expression | Result |
|---|---|
| bookstore | Selects all nodes with the name "bookstore" |
| /bookstore | Selects the root element bookstore<br><br>**Note:** If the path starts with a slash ( / ) it always represents an absolute path to an element! |
| bookstore/book | Selects all book elements that are children of bookstore |
| //book | Selects all book elements no matter where they are in the document |
| bookstore//book | Selects all book elements that are descendant of the bookstore element, no matter where they are under the bookstore element |
| //@lang | Selects all attributes that are named lang |

Predicates are used to find a specific node or a node that contains a specific value.
Predicates are always embedded in square brackets.
In the table below we have listed some path expressions with predicates and the result of the expressions:

| Path Expression | Result |
|---|---|
| /bookstore/book[1] | Selects the first book element that is the child of the bookstore element.<br><br>**Note:** In IE 5,6,7,8,9 first node is[0], but according to W3C, it is [1]. To solve this problem in IE, set the SelectionLanguage to XPath:<br><br>*In JavaScript:*<br>*xml*.setProperty("SelectionLanguage","XPath"); |
| /bookstore/book[last()] | Selects the last book element that is the child of the bookstore element |
| /bookstore/book[last()-1] | Selects the last but one book element that is the child of the bookstore element |
| /bookstore/book[position()<3] | Selects the first two book elements that are children of the bookstore element |
| //title[@lang] | Selects all the title elements that have an attribute named lang |
| //title[@lang='en'] | Selects all the title elements that have a "lang" attribute with a value of "en" |
| /bookstore/book[price>35.00] | Selects all the book elements of the bookstore element that have a price element with a value greater than 35.00 |
| /bookstore/book[price>35.00]/title | Selects all the title elements of the book elements of the bookstore element that have a price element with a value greater than 35.00 |

Selecting Unknown Nodes

| Wildcard | Description |
|---|---|
| * | Matches any element node |
| @* | Matches any attribute node |
| node() | Matches any node of any kind |

| Path Expression | Result |
|---|---|
| /bookstore/* | Selects all the child element nodes of the bookstore element |
| //* | Selects all elements in the document |
| //title[@*] | Selects all title elements which have at least one attribute of any kind |

| Path Expression | Result |
|---|---|
| //book/title \| //book/price | Selects all the title AND price elements of all book elements |
| //title \| //price | Selects all the title AND price elements in the document |
| /bookstore/book/title \| //price | Selects all the title elements of the book element of the bookstore element AND all the price elements in the document |

| Funzioni | Description |
|---|---|
| *distinc-values()* | valori non duplicati |
| *count()* | conteggio elementi |
| *text()* | stampa la stringa |
| *position()* | posizione nodo |
| *last()* | numero elementi |
| *name()* | nome del nodo |
| *[ ]* | filtro |

- [ ] is an overloaded operator
- Filtering by position (if numeric value) :
  - /book[3]
  - /book[3]/author[1]
  - /book[3]/author[2 to 4]
- Filtering by predicate :
  - //book[author/firstname = "ronald"]
  - //book[@price <25]
  - //book[count(author[@gender="female"] )>0]
- Existential filtering:
  - //book[author]

# 5 XPath Axes

An axis represents a relationship to the context (current) node, and is used to locate nodes relative to that node on the tree.

- Axis can be missing
  - By default the child axis
    - $x/child::person -> $x/person
- Short-hands for common axes
  - Descendent-or-self
    - $x/descendant-or-self::*/child::name-> $x//name
  - Parent
    - $x/parent::* -> $x/..
  - Attribute
    - $x/attribute::year -> $x/@year
  - Self
    - $x/self::* -> $x/.

| AxisName | Result |
| --- | --- |
| ancestor | Selects all ancestors (parent, grandparent, etc.) of the current node |
| ancestor-or-self | Selects all ancestors (parent, grandparent, etc.) of the current node and the current node itself |
| attribute | Selects all attributes of the current node |
| child | Selects all children of the current node |
| descendant | Selects all descendants (children, grandchildren, etc.) of the current node |
| descendant-or-self | Selects all descendants (children, grandchildren, etc.) of the current node and the current node itself |
| following | Selects everything in the document after the closing tag of the current node |
| following-sibling | Selects all siblings after the current node |
| namespace | Selects all namespace nodes of the current node |
| parent | Selects the parent of the current node |
| preceding | Selects all nodes that appear before the current node in the document, except ancestors, attribute nodes and namespace nodes |
| preceding-sibling | Selects all siblings before the current node |
| self | Selects the current node |

# 6 XQuery

- **Data model**
  - Instance of the data model
    - a sequence composed of zero or more items
  - The empty sequence often considered as the "null value"
  - Items
    - Nodes or
    - Atomic values
  - Nodes can be
    - document
    - element
    - attribute
    - text
    - namespaces
    - PI
    - comment
- **Sequences**
  - Can be ***heterogeneous*** (nodes and atomic values)

    $$(< a/ >, 3)$$

  - Can contain duplicates (by value and by identity)

    $$(1, 1, 1)$$

  - Are not necessarily ordered in document order
  - Nested sequences are automatically flattened

    $$(1, 2, (3, 4)) = (1, 2, 3, 4)$$

  - Single items and singleton sequences are the same

    $$1 = (1)$$

- **XML nodes**
  - Every node has a unique node identifier
  - Nodes have children and an optional parent $\rightarrow$ conceptual "tree"
  - Nodes are ordered based of the topological order in the tree

- **Combining sequences**
  - *Union, Intersect, Except*
  - ***Work only for sequences of nodes, not atomic values***
  - Eliminate duplicates and reorder to document order

```
$x := <a/>, $y := <b/>, $z := <c/>
($x, $y) union ($y, $z) => (<a/>, <b/>, <c/>)
```

- **Atomic values**
  - Instances of all XML Schema atomic types, *Typed* (schema validated)
    - xs:integer, xs:boolean, xs:date
  - *untyped* atomic values (non schema validated)
- **XQuery expressions**
  - Constants, Variable, FunctionCalls, PathExpr, ComparisonExpr, ArithmeticExpr, LogicExpr, FLWRExpr, ConditionalExpr, QuantifiedExpr, TypeSwitchExpr, InstanceofExpr, CastExpr, UnionExpr, IntersectExceptExpr, ConstructorExpr, ValidateExpr

## 6.1 Query in XQuery

- **Variables**
  - Si definiscono in questo modo → ***$*** + ***Name***
  - Vengono precedute da
    - ***let***
    - ***for***
  - Il risultato deve essere poi restituito da ***return***
  - Variable destroyed after spaces

```
let $x := ( 1, 2, 3 )
return count($x)
```

- **Functions**
  - XQuery uses functions to extract data from XML documents
  - The ***doc() function*** is used to open the "books.xml" file
  - ***empty(item*)*** → return boolean
  - ***index-of(item*, item)*** → return xs:unsignedInt
  - ***distinct-values(item*)*** → return item*
  - ***distinct-nodes(node*)*** → return node*
  - ***union(node*, node*)*** → return node*
  - ***except(node*, node*)*** → return node*
  - ***string-length(xs:string)*** → return xs:integer
  - ***contains(xs:string, xs:string)*** → return xs:boolean
  - ***date(xs:string)*** → return xs:date
  - ***add-date(xs:date, xs:duration)*** → return xs:date

  - In-place XQuery functions, such as
    - Can be recursive and mutually recursive
    - Senza {} verrebbe interpretato il contenuto come una stringa

```
declare function ns:foo($x as xs:integer) as element()*
    { <a>{$x+1}</a> }
```

List of return elements

Senza {} verrebbe interpretato il contenuto come una strings

- **Arithmetic expressions**
  - Apply the following rules
    - atomize all operands
    - if an operand is untyped, cast to xs:double (if unable, → error)
    - if the operand types differ but can be promoted to common type, do so
      - xs:integer can be promoted to xs:decimal

Examples:
- -1 - (4 * 8.5)
- $b mod 10
- <a>42</a> + 1

- **Logical expressions**
  - Possono essere
    - *expr1* ***and*** *expr2*
    - *expr1* ***or*** *expr2*
  - Return true, false
  - Rules:
    - first compute the Boolean Effective Value (BEV) for each *operand/expr*
    - then use standard two value Boolean logic on the two BEV's as appropriate
  - ***false and error*** → false or error!
    - *non-deterministic: it is impossible to foresee the result!!!)*
- **Comparison**
  - $1 = (1, 2, 3)$ → **True, perché 1 é contenuto dentro l'insieme**
  - $1 \quad != (1, 2, 3)$ → **True, questo perché 1 viene confrontato elemento per elemento**

| | | |
|---|---|---|
| Value | for comparing single values | `eq, ne, lt, le, gt, ge` |
| General | Existential quantification + automatic type coercion | `=, !=, <=, <, >, >=` |
| Node | for testing identity of single nodes | `is, isnot` Memory comparation |
| Order | testing relative position of one node vs. another (in document order) | `<<, >>` |

(c) Schema

| | |
|---|---|
| <a>42</a> eq "42" | true |
| <a>42</a> eq 42 | error |
| <a>42</a> eq 42.0 | error |
| <a>42</a> = 42 | true |
| <a>42</a> = 42.0 | true |
| <a>42</a> eq 42 | true |
| <a>42</a> eq  42 | false |
| <a>baz</a> eq 42 | type error |
| () = 42 | false |
| (<a>42</a>,43) = 42 | true |
| ns:shoesize(5) eq ns:hatsize(5) | true |
| (1,2) = (2,3) | true |
| (1,2) != (2,3) | true |
| (1,2) != (1,2) | true |

(d) esempi

- **FLWOR expressions**
  - FOR
  - LET
  - WHERE
  - ORDERR BY
  - RETURN

- **Simple iteration expressions**

  **for** *variable* **in** *expression1*
  **return** *expression2*

  - iteratively bind the variable ***to each root node*** of the forest returned ***by expression1***
  - for each such binding, evaluate expression2
  - concatenate the resulting sequences
  - nested sequences are automatically flattened

  Example: for $x in doc("bib.xml")/bib/book
  return $x/title

- **LET Expression**

  let *variable* := *expression1*
  return *expression2*

  - bind the variable to ***the result of the expression1*** ( an entire sequence of values)
  - add this binding to the current environment
  - evaluate and return expression2

  Example : let $x :=doc("bib.xml")/bib/book
  return count($x)

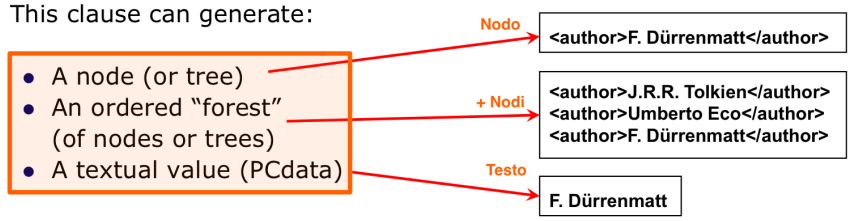
- **WHERE Expression**
  - express a condition over nodes
  - only nodes satisfaying the condition are further considered.
  - **not()** is implemented by a function

doc("books.xml")//book[publisher="Bompiani" and @avalilable="Y"]

```
for $book in doc ("books.xml")//book
where $book/publisher="Bompiani"
        and $book/@avalilable="Y"
return $book
```

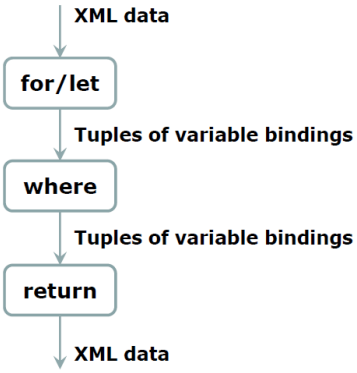
## • RETURN Expression

- This clause can generate:
  - A node (or tree)
  - An ordered "forest" (of nodes or trees)
  - A textual value (PCdata)




- It can contain node constructors
  - <result>
    literal text content
    </result>
  - <result>
    { $x/name } {-- Braces "{}" used to delineate evaluated content --}
    </result>
  - <result>
    some content here {$x/name} and some more here
    </result>

## • FLWR expressions: evaluation




- **for** : iteration, "individual" bindings
  - Every item in a sequence generates a different binding (whose value is that item)
- **let** : "collective" binding
  - A sequence of items is collectively bound to one variable (whose value is the whole sequence)
- **where** : filtering expressions
  - Independently evaluated on each tuple of bindings to keep or discard it
- **return** : construct results
  - Executed once for each tuple of bindings

- **A FWR query**

```
for $book in doc("books.xml")//book
where $book/price>60
return   <expensiveBook>
         { $book/title }
         </expensiveBook>
```

```
<expensiveBook><title>TCP/IP Illustrated</title></expensiveBook>
<expensiveBook>
  <title>The Economics of Technology and Content for Digital TV</title>
</expensiveBook>
```

```
for $book in doc("books.xml")//book
where $book/price>60
return   <expensiveBook>
         { $book/title/text() }
         </expensiveBook>
```

```
<expensiveBook>TCP/IP Illustrated</expensiveBook>
<expensiveBook>
  The Economics of Technology and Content for Digital TV
</expensiveBook>
```

- **Similar query with LET**

```
let $books := doc("books.xml")//book[price>60]
return   <expensiveBooks>
         { $books/title }
         </expensiveBooks>
```

```
<expensiveBooks>
  <title>TCP/IP Illustrated</title>
  <title>The Economics of Technology and Content for Digital TV</title>
</expensiveBooks>
```

```
let $books := doc("books.xml")//book
where $books/price>60
return   <expensiveBooks>
         { $books/title }
         </expensiveBooks>
```

```
<expensiveBooks>
  <title>TCP/IP Illustrated</title>
  <title>Data on the Web</title>
  <title>The Economics of Technology and Content for Digital TV</title>
</expensiveBooks>
```

- **ORDER BY Expression**

```
for $book in doc("books.xml")//book
order by $book/title
return
    <book>
    { $book/title,
      $book/publisher }
    </book>
```

Asceding if u are not specifying

Evaluated

**The ordering will be applied when u have a list of results. So first u have to have the full list**

- **Counting combined with let clause**

```
for $e in  doc("books.xml")//publisher
let $book :=  doc("books.xml")//book[publisher = $e]
where count($book) > 100
return $e
```

- **distinct-values()**

```
for $e in  distinct-values( doc("books.xml")//publisher )
let $book :=  doc("books.xml")//book[publisher = $e]
where count($book) > 100
return $e
```

- **Conditional expressions**

```
if ( $book/@year <1980 )
then (<old-book>{$x/title}</old-book>)
else (<new-book>{$x/title}</new-book>)
```

- **FLWR expressions**
  - **FLWR expression**:

```
for $x in //book
let $y := $x/author
where $x/title="Ulysses"
return count($y)
```

  - **Equivalent to**:

```
for $x in //bib/book
return (  let $y := $x/author
          return if ($x/title="Ulysses" )
                 then count($y)
                 else ()            )
```

- **Selections**

```
for $b in doc("bib.xml")//book
where $b/publisher = "Springer Verlag" and
        $b/@year = "1998"
return $b/title
```

- **Joins**

```
for $b in doc("bib.xml")//book,
    $p in doc("pubs.xml")//publisher
where $b/publisher = $p/name
return ( $b/title , $p/address)
```

- **Order by Example**

```
for $b in doc("bib.xml")//book
where $b/publisher = "Springer Verlag"
order by $b/@year
return $b/title
```

Enforce ordering that are not inside the result

- **Functions**
  - XQuery uses functions to extract data from XML documents
  - The ***doc() function*** is used to open the "books.xml" file
  - ***empty(item*)*** → return boolean
  - ***index-of(item*, item)*** → return xs:unsignedInt
  - ***distinct-values(item*)*** → return item*
  - ***distinct-nodes(node*)*** → return node*
  - ***union(node*, node*)*** → return node*
  - ***except(node*, node*)*** → return node*
  - ***string-length(xs:string)*** → return xs:integer
  - ***contains(xs:string, xs:string)*** → return xs:boolean
  - ***date(xs:string)*** → return xs:date

- ***add-date(xs:date, xs:duration)*** → return xs:date

- ***contains(string(.), "stringa")*** → return xs:boolean
- ***ends-with(local-name(.), "stringa")*** → return xs:boolean
- ***exists($variaible)*** → return xs:boolean
- ***name()*** → return nome del nodo
- ***max()*** → prendo il valore massimo numerico possibilmente
- ***sum()*** → somma
- ***avg()*** → media
- ***div*** → per dividere 2 operandi

## 6.2 DML XQuery

```
DTD for users.xml
<!ELEMENT users (user_tuple*)>
<!ELEMENT user_tuple (userid, name, rating?)>
<!ELEMENT userid (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT rating (#PCDATA)>

DTD for items.xml
<!ELEMENT items (item_tuple*)>
<!ELEMENT item_tuple (itemno, description, offered_by, start_date?, end_date?,
                                                                    reserve_price?)>
<!ELEMENT itemno (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT offered_by (#PCDATA)>
<!ELEMENT start_date (#PCDATA)>
<!ELEMENT end_date (#PCDATA)>
<!ELEMENT reserve_price (#PCDATA)>

DTD for bids.xml
<!ELEMENT bids (bid_tuple*)>
<!ELEMENT bid_tuple (userid, itemno, bid, bid_date)>
<!ELEMENT userid (#PCDATA)>
<!ELEMENT itemno (#PCDATA)>
<!ELEMENT bid (#PCDATA)>
<!ELEMENT bid_date (#PCDATA)>
```

- **Inserimento:** ***return do insert - into doc("prova.xml")***

```
do insert
    <user_tuple>
        <userid>U07</userid>
        <name>Annabel Lee</name>
    </user_tuple>
into doc("users.xml")/users
```

```
let $uid := doc("users.xml")/users/user_tuple[name="Anna Lee"]/userid
let $topbid := max(doc("bids.xml")/bids/bid_tuple[itemno=1002]/bid)
where $topbid*1.1 <= 200
return do insert <bid_tuple>
                    <userid>{data($uid)}</userid>
                    <itemno>1007</itemno>
                    <bid>{$topbid*1.1}</bid>
                    <bid_date>1999-02-01</bid_date>
                </bid_tuple>
into doc("bids.xml")/bids
```

- **Modifica:** ***return do replace value of $variabile with***

```
let $user := doc("users.xml")/users/user_tuple[name="Annabel Lee"] return do replace
value of $user/rating with "B"
```

- **Cancellazione:** ***return do delete $variabili***

```
let $user := doc("users.xml")/users/user_tuple[name="X Y"]
let $items := doc("items.xml")/items/item_tuple[offered_by=$user/userid]
let $bids := doc("bids.xml")/bids/bid_tuple[userid=$user/userid]
return (
        do delete $user,
        do delete $items,
        do delete $bids
)
```

An alternative solution is:

```
let $user := doc("users.xml")/users/user_tuple[name="X Y"]
let $items := doc("items.xml")/items/item_tuple[offered_by=$user/userid]
let $bids := doc("bids.xml")/bids/bid_tuple[userid=$user/userid]
return do delete $user, $items, $bids
```

- **Inserimento forzato in coda:** ***altrimenti verrá inserito in maniera non deterministica***

```
do insert <comment>This is a bargain !</comment>
as last into doc("items.xml")/items/item_tuple[itemno=1002]
```

- **Verifica dentro in lista**

```
for $keyword at $i in ("car", "skateboard", "canoe"),
    $parent in doc("part-tree.xml")//part[@name=$keyword]
let $descendants := $parent//part
for $p in ($parent, $descendants)
return do replace value of $p/@partid with $i*1000+$p/@partid
```

- **Creazione variabile e utilizzo**

```
let $x := (<result>1</result>,
           <result>2</result>,
           <result>3</result>,
           <result>4</result> )
return $x
```

```
<result>1</result>
<result>2</result>
<result>3</result>
<result>4</result>
```

- **Foreach che soddisfa**

```
for $n in //Country
where every $m in $n/*/Medal satisfies count( $m/Athlete ) = 1
return <OnlySingle>
          { $n/@name }
       </OnlySingle>
```

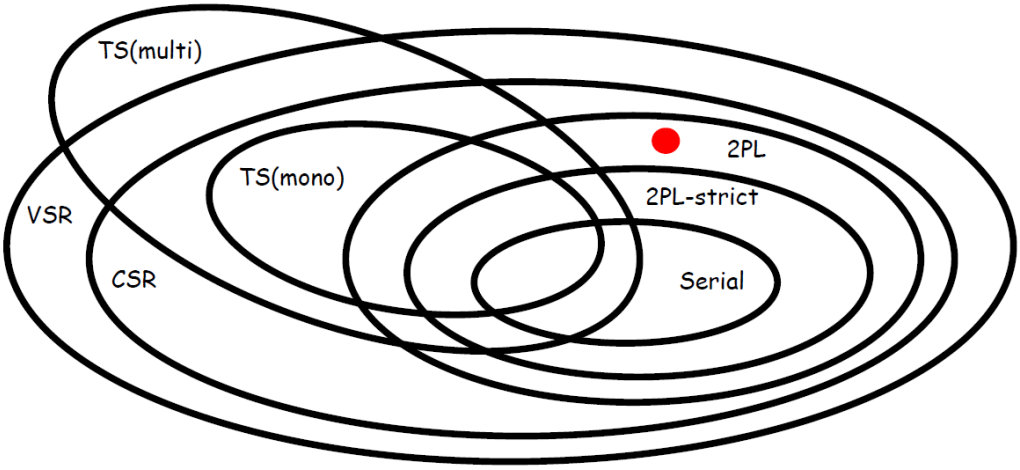
# 7 Metodologie di controllo di uno schedule

Per classificare uno schedule è molto più semplice partire da un gruppo interno, questo perchè essendo un sottoinsieme dell'insieme esterno de rappresenta l'appartenenza.
Per Esempio:

*se uno schedule è 2PL allora sarà sicuramente anche CSR e VSR, ossia*

$$2Pl \Rightarrow CSR \text{ } e \text{ } 2PL \Rightarrow VSR \text{ } ma$$

$$CSR \not\Rightarrow 2PL \text{ } e \text{ } VSR \not\Rightarrow 2PL$$




## 7.1 Conflict-serializzabilità (CSR)

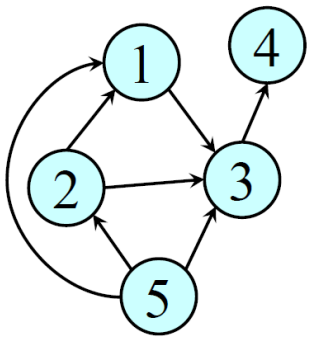
- Action $a_i$ **is conflicting** with $a_j$ se sono soddisfatte tutte le seguenti condizioni:
  - devono appartenere a transazioni diverse $i \neq j$
  - operano sulla stessa variaibile
  - e una delle due azioni è un **write operation**
  - Esempio:
    - $rw$ e $wr$ $\rightarrow$ read - write conflicting
    - $ww$ $\rightarrow$ write - write conflict
- ***Per testare la conflict serializability*** si utilizza il **Conflict graph**
  - I nodi indicano le transazioni $(T_1, T_2, ..., T_n)$

  - Un arco da una transazione $T_i$ verso $T_j$ implica l'esistenza di un conflitto tra un'azione $a_i$ di $T_i$ e un'azione di $a_j$ di $T_j$, la freccia viene rivolta verso l'ultima azione di conflitto.
  - ***I CONFLITTI SI GENERANO ANCHE SE LE AZIONI NON SONO CONSECUTIVE***
  - **Teorema** un schedule si dice CSR se e solo se il conflict graph risulta essere **aciclico**
    - NB: guardare le frecce dei nodi per verificarne la ciclicità

- ***Ogni CSR è anche VSR, ma non è sempre vero il contrario.***
- *I nodi che fanno parte dei cicli devono avere archi entranti e uscenti*
- *I nodi che hanno archi di sola entranta o di uscita non vanno considerati nel ciclo*
- Procedura da seguire:
  1. Consideriamo le operazioni relative a ogni risorsa separatamente
  2. Costruire il grafo
  3. Gli archi appartenenti a due transazioni può avvenire nei 2 sensi
     - $r2 \rightarrow w5$ viene fatto l'arco da 2 a 5
     - $r5 \rightarrow w2$ *viene fatto l'arco da 5 a 2*
  4. E vedere se presenta cicli

S: $w_3$
U: $w_5$ $w_2$ $w_1$
X: $r_1$ $r_5$ $w_3$
Y: $r_2$ $w_3$ $r_4$
Z: $w_5$ $r_5$

(e) operazioni relative a ogni risorsa




(f) Conflict graph

## 7.2 View-serializzabilità (VSR)

- **Def:** una scritura $w_i(x)$ si dice ***cieca/blind write,*** se non è l'ultima azione sulla risorsa $x$ e l'azione successiva su $x$ è una scrittura $w_j(x)$
- **Proprietà**
  - Ogni schedule $S \in VSR$ ma $S \notin CSR$ ha nel grafo dei conflitti cicli a cui concorrono archi che corrispondono a coppie di *scritture cieche*
  - Tali *scritture cieche* possono essere invertite senza impatto sulla relazione ***legge da*** e sulle ***scritture finali***
    - ma con l'effetto di invertire un arco nel grafo dei conflitti
  - Con questi scambi si può rendere aciclico il grafo, e si può ispezionarlo per individuare uno schedule seriale view-equivalent a quello inziale
- *Cerco di rendere aciclico il grafo utilizzando la proprietà delle* ***scritture cieche***
  1. Individuo l'*arco* comune ai cicli
  2. Cerco di riscrivere le *blind write* in modo da rendere il grafo aciclico
  3. Individuo un schedule seriale *view equivalent* che deve passare da tutti i nodi $\rightarrow VSR$
     - si può riconoscere un cammino che tocca tutti i nodi del grafo dei conflitti dello schedule modificato
     - *I nodi che hanno archi di sola entranta o di uscita non vanno considerati nel ciclo*
       - Questi nodi possono essere aggiunti nell'ordine che si vuole per la creazione dello schedule serializzabile

## 7.3 Locking a due fasi (2PL)

- le ***read operations***
  - sono precedute da **r_lock** (SHARED LOCK)
  - e seguite da **unlock**
- le ***write operations***
  - sono precedute da **w_lock** (EXCLUSIVE LOCK)
  - e seguite da **unlock**

| REQUEST | RESOURCE STATE | | |
|---|---|---|---|
| | FREE | R_LOCKED | W_LOCKED |
| `r_lock` | OK R_LOCKED | OK R_LOCKED | NO W_LOCKED |
| `w_lock` | OK W_LOCKED | NO R_LOCKED | NO W_LOCKED |
| `unlock` | ERROR | OK DEPENDS | OK FREE |

Figure 10: conflict table

- **Def:** Il 2PL non ci permette di conoscere con precisione i momenti di rilascio (*quando termina effettivamente l'operazione*).
- *É* invece possibile con 2PL strict
- **Def:** Il 2PL impone di non acquisire alcun lock dopo il primo rilascio (prima il lock delle risorse e poi unlock $\rightarrow$ 2 fasi totali). Se non viene riespettata questa regola lo schedule non è **2PL**
  1. Growing Phase: si possono solo acquisire risorse
  2. Shrinking Phase: si possono solo rilasciare le risorse acquisite
- Procedimento
  - Diagrammiamo lo schedule separando le azioni in base alla risorsa a cui si riferiscono
  - Numeriamo progressivamente gli istanti in cui avvengono le azioni delle varie transazioni
  - Costruiamo un sistema di disequazioni

∗ *Dove secondo la **definizione** non sappiamo il momento esatto del rilascio (**unlock** sulla risorsa)*

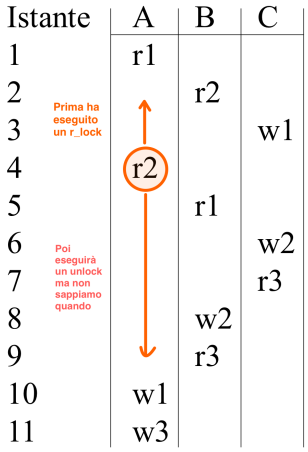



Figure 11: esempio

- Non si possono eseguire richieste di lock dopo un unlock
- Si possono fare i lock prima degli istanti visibili nell'esecizion come ipotesi
- Si possono fare lock su più oggetti contemporaneamente
- Le ***read operations*** hanno un ***shared lock*** → guardare tabella dei conflitti
- se una transazione prima legge e poi scrive un oggetto, modifica il **r_lock** in **w_lock** (*lock escalation*)

## 7.4 2PL Strict

- Viene aggiunta una regola al **2PL**
  - *I lock possono solamente essere rilasciati dopo un com-mit/rollback*
- ***Def:*** *assumiamo che tutte le transazioni effettuino il commit/release immediatamente dopo l'ultima operazione della transazione stessa*
  - se dovesse presentarsi una operazione di rilascio di un lock all'istante 4 mentre la transazione termina al 8' viola questa definizione e si puó concludere che non é **2PL Strict**

## 7.5 Timestamp (TS Singolo, TS Multi)

- Vengono utilizzati 2 contatori per ogni risorsa
  - **RTM(x)** → per la lettura dell'oggetto x
  - **WTM(x)** → per la scrittura dell'oggetto x
- **TS Mono**
  - ***read(x, ts)***
    - Viene effettuata una lettura dell'oggetto x al timestamp ts
    - Se $ts < WTM(x)$ la richiesta viene rigettata e la transazione uccisa
    - Nel caso contrario la transazione è eseguita e $RTM(x) = max(RTM(x), ts)$
  - ***write(x, ts)***
    - Viene effettuata una scrittura dell'oggetto x al timestamp ts
    - Se $ts < WTM(x)$ **oppure** $ts < RTM(x)$ la richiesta viene rigettata e la transazione uccisa
    - Nel caso contrario la transazione è eseguita e $WTM(x) = ts$
- **Multiversion Concurrency Control - TS Multi teoria**
  - L'idea alla base di questo protocollo
    - Le scritture generano nuove coppie, ognuna con un nuovo $WTM$
    - Le letture accedono alla giusta copia
  - Ogni oggetto $x$, possiede $N > 1$ coppie attive con un proprio $WTM_N(x)$
  - C'è un solo un unico globale $RTM(x)$, per oggetto $x$
  - Le vecchie coppie sono eliminate quando non ci sono più transazioni che accedono
- **TS Multi**
  - ***read(x,ts)***
    - *É* sempre eseguita e una coppia $x_k$ è selezionata per la lettura
    - Se $ts > WTM_N(x)$, allora $k = N$, quindi prendo l'ultima coppia appena scritta

        * Altrimenti viene presa la versione $k$ in questo modo $WTM_k(x) < ts < WTM_{k+1}(x)$
            · prendo quella immediatamente inferiore a $WTM_{k+1}$
        * $RTM = max(ts, RTM)$
    - ***write(x,ts)***
        * Se $ts < RTM(x)$, allora la richiesta non viene accettata
        * Altrimenti viene creata una nuova copia (N viene incrementato), con $WTM_N(x) = ts$

- *Una volta che la transazione viene uccisa, le operaqzioni svolte dalla stessa transazione dopo non vengono effettuate*

- *Esempio:*

$$w_2(y) \quad WTM(y) = \{0, 2\}$$

*vado a generare un'altra coppia*

$$r_2(x) \quad RTM = 4(0)$$

*vado a leggere la coppia 0 con RTM = max(ts, RTM) = 4*

- ***Nel caso in cui le opeazioni di lettura e scrittura sulle risorse hanno pedici sempre crescenti (lo si nota nella prima fase di CSR) allora automaticamente é → TS Mono e TS Multi***

## 7.6 Update Lock

- Non si possono avere piú update lock sulla stessa risorsa
- In base all'esercizio su puó fare l'upgrade da **UP a XL**

| Request | State | | |
|---|---|---|---|
| | SL | UL | XL |
| SL | OK | OK | No |
| UL | OK | No | No |
| XL | No | No | No |

Figure 12: Schema

## 7.7 Hierarchical Locking

- Ci sono 5 modalitá di esecuzione di una transazione:
  - **SL:** shared lock
  - **XL:** exclusive lock
  - **ISL:** Intention of locking in shared mode
  - **IXL:** Intention of locking in exclusive mode
  - **SIXL:** Lock in shared mode with intention of locking in exclusive mode (SL + IXL)
    - Acquisisco il **SL** sul nodo root (*lo leggo*) con l'intenzione di scrivere sui nodi dentro **IXL**
- **SL e XL** sono eseguiti sui nodi foglia
- *per accedere ai nodi foglia devo acquisire il lock suo nodi root e lo faccio con le* ***intenzioni***
- L'accesso all'albero viene fatto passando da nodi della root per poi arrivare alle foglie
- Si cerca di utilizzare un lock sempre meno restrittivo sui nodi root per poter essere utilizzati anche dalle altre transazioni e la tempo stesso selvaguardare l'interesse della transazioni in corso
- I lock vengono mantenuti anche sui nodi root
- Si puó effettuare il downgrading di un lock
- Posso acquisire i lock all'inizio di una transazioni per poi effettaure l'operazione stessa dopo per mantenere uno schedule 2PL
- si possono fare gli upgrade (da **SL a XL** e da (**ISL a IXL**))

| Request \ Resource state | ISL | IXL | SL | SIXL | XL |
|---|---|---|---|---|---|
| ISL | OK | OK | OK | OK | No |
| IXL | OK | OK | No | No | No |
| SL | OK | No | OK | No | No |
| SIXL | OK | No | No | No | No |
| XL | No | No | No | No | No |

Figure 13: Schema

## 7.8 Obermark's algorithm

- L'algoritmo é distribuito
- Ogni nodo decide di inviare le proprie waiting condition

$$E_h \to T_i \to T_m \to T_n \to T_o \to T_p \to ...T_j \to E_k$$

*questa espressione viene tradotti in:*

$$E_h \to T_i \to T_j \to E_k$$

- Il messaggio viene inviato al nodo $k$ ($E_k$) se e solo se $i > j$
- Le frecce indica effettivamente in che direzione va l'arco da a

$t_3 \to E_2$ *vuol dire che esiste un arco che va da* $t_3$ *a* $E_4$
$E_3 \to t_2$ *vuol dire che esiste un arco che va da* $E_3$ *a* $t_4$

- $t_n \to t_j$ *indica che* $t_n$ *aspetta che* $t_j$ *liberi la risorsa*
- $t_n \to E_m$ *indica che* $t_n$ *aspetta l'esecuzione della transazione sul nodo* $m$
- $E_m \to t_n$ *indica che* $t_n$ *é stato invocato da una transazione sul nodo* $m$
- Procedimento:
  1. Individuare quali sono i nodi che inviano i messaggi
  2. inviare i messaggi ai nodi di destinazione ($i > j$)
  3. Aggiungere tale messaggio al porprio nodo
  4. Vedere se aggiungendo vengano generati o meno messagi e ripetere i passi fino a 4
  5. Un volta terminati i possibili invi controllare se nei nodi si formano cicli
  6. Nel caso di cicli viene confermato il deadlock
- I nodi della transazione vengono aggiunti se e solo se non ci sono nel nodo di arrivo e vengono collegati al $E_n$ nuovo
- Se il nodo della transazione giá esiste indipendenetemente che le frecce siano sbagliate' cosí come la $E$ sia sbagliata si effettua solo il collegamento

## 7.9 Obermark's algorithm applied on a schedule

- Assunzioni
  - Ogni transazione inizia con il primo accesso alla risorsa
  - Il lock viene mantenuto fino alla fine della transazione
  - Meccanisco RPC
- Le frecce sono utilizzate per
  - Rappresentare e seguire il corso di una transazione di riferimento
    - Tutti i nodi della $T_3$ sono tra loro collegati in maniera temporale
  - Per indicare una waiting resource per liberarsi
- le letture tra loro sono consentite perché si tratta di shared lock non c'é bisogno di acquisire nulla
- Ovviamente le letture devono aspettare che si concludano le scritture prima di leggere
- Mentre le scritture devono attendere la fine delle altre scritture/letture
- Le operazioni della stessa transazione solo tra loro collegate

| Request | Resource state | | | | |
|---|---|---|---|---|---|
| | ISL | IXL | SL | SIXL | XL |
| ISL | OK | OK | OK | OK | No |
| IXL | OK | OK | No | No | No |
| SL | OK | No | OK | No | No |
| SIXL | OK | No | No | No | No |
| XL | No | No | No | No | No |

Figure 14: Schema

# 8 Esercizi sul log

- I comandi delle transazioni:
  - **begin:** b
  - **commit:** c
  - **abort:** a
- Le operazioni delle transazioni:
  - **insert:** i
  - **delete:** d
  - **update** u
- Le operazioni di recovery:
  - **dumb:** Dump
  - **checkpoint:** ckpt

- Records relevant to transactional commands:
  - `B(T), C(T), A(T)`
- Records relevant to operations
  - `I(T,O,AS), D(T,O,BS), U(T,O,BS,AS)`
- Records relevant to recovery actions
  - `DUMP, CKPT(T1,T2,…,Tn)`
- Record fields:
  - `T`: transaction identifier
  - `O`: object identifier
  - `BS`, `AS`: before state, after state

Figure 15: Schema

- Passi da seguire
  1. Individuare il checkpoint
  2. Le transazioni all'interno del checkpoint vengono annotate all'interno di **UNDO**
  3. Tutte le operazioni svolte della transazioni dopo il checkpoint
     - vengono aggiunte tutte le transazioni con il **begin** al **UNDO set**
     - spostare tutte le tranazioni dal **UNDO** al **REDO** quelle che eseguono un **commit**

4. Partendo dalla prima operazione
    * Le transazioni che fanno parte dell'**UNDO** devono essere riportate allo stato iniziale
    * *Le operazioni di insert diventano oprazioni di delete dell'oggetto creato*
    * Le transzioni che fanno parte del **REDO set** vengono rieseguito nell'ordine corretto riportato nel log
    * le transazioni che non sono riportate nei set non vengono toccate

- **REDO:** before state
- **UNDO:** after state
- il *ready* record é la decisione presa localmente, puó indicare (dipende dall'esercizio)
    * un commit
    * un abort

# 9 Physical Execises

- Ignorare il *caching* implica che si debba accedere al file system per forza (da esercizio)
- Numero di accessi dipende dai predicati di un query, rispettivamente dal numero di accessi che si devono effettivamente fare per accedere e risolvere il filtro (Where) in questione

## 9.1 Hash-based access structures

- **Varibili di esercizio**
  - $T$ numero di tuple previste per il File
  - $F$ numero medio di tuple memorizzate in ciascuna pagina
  - $B$ numero blocchi per file = $\frac{T}{(0.8 \times F)}$ utilizzando l'80% dello spazio disponibile
  - ***Costo di accesso:*** *rappresenta la probabilitá che un bucket sia pieno e richieda un'ulteriore lettura di un* ***overflow block***
- **Approfondimenti**
  - ***Bucket:*** *é un* ***blocco*** *su un disco o un'insieme di blocchi*
- **Risoluzione esercizio**
  - Si tratta di una struttura che si basa su una *key filed*
  - Dati $n$ filtri all'interno della query, dove i filtri sono seguiti da *Where*
    - Dobbiamo andare a scegliere quelli che si basano su tale *key filed*
    - Cercare di ignorare i filtri che non appartengono alla *key filed*
  - Dato il *costo medio di accesso singolo*: $c_s$
  - Si contano i numeri di filtri effettivi che si fanno per eseguire la query in questione: $\boldsymbol{n}$
  - Il numero di accessi totale alla struttura

$$n \cdot c_s$$

## 9.2 B+ tree

- **Variabili di esercizio**
  - ***Fan-out:*** *numero di discendenti per nodo*
  - ***Balanced tree:****le lunghezze dei cammini dal nodo radice ai nodi foglia sono tutti uguali.*
- **Approfondimenti**
  - Gli accessi che si devono effettuare all'interno di un *tree* sono per ordine
    1. Root
    2. Intermediate nodes (1 o piú)
    3. Nodi foglia
    4. Il blocco con la tupla rispettiva
  - solitamente il root access é salvato in cache
- **Risoluzione esercizio**
  1. L'ipotesi più ottimistica è quella di considerare la struttura come un ***balanced tree***
  2. Ogni nodo possiede quindi lo stesso numero di ***fan out***
  3. La profondità del ***B+ tree*** é data da

$$profonditá = log_{\,fan\ out}\ (\ tuple\ totali\ per\ file)$$

  4. Dobbiamo quindi effettuare *profondit/'a* accessi al disco per un eventuale valore

$$Numeri\ accessi\ totali = profonditá$$
$$Costo\ totale = Numeri\ accessi\ totali + eventuali$$

- **NOTA BENE**

$$profonditá = log_{\,fan\ out}\ (\ tuple\ totali\ per\ file)$$

$$fan\ out\ = \sqrt[profonditá]{tuple\ totali}$$

$$n\ tuple\ totali = n\ puntatori\ a\ quelle\ tuple$$

## 9.3 Scegliere l'indice piú efficiente

- Ci vengono rappresentati $x$ indici formati da $y$ attributi

$$Idx1(A,\ C,\ D)$$
$$Idx2(B,\ C,\ E)$$
$$Idx3(B,\ D,\ A)$$

- Gli indici fanno affidamento sui attributi di cui sono composti
  - L'attributo che si trova in *prima posizione* é quello su cui fanno piú affidamento e sono i piú importanti

$$Idx1(\mathbf{A},\ C,\ D)$$
$$Idx2(\mathbf{B},\ C,\ E)$$
$$Idx3(\mathbf{B},\ D,\ A)$$

  - La posizione é molto imporante indipendentemente dal risultato finale
  - Un indice che utilizza solo l'ultimo attributo ed é piú veloce ***"perde"*** contro uno che usa i primi 2

- Ci viene fornita la query dove avrá un determinato numero di filtri

```
select *          from R
where B = k1 and C = k2 and D = k3 and E = k4
```

Figure 16: query

- Dobbiamo stimare la selettività consentita da ciascuno degli indici disponibili, considerando ***solo i predicati menzionati nella clausola where***
  - **Non viene scelto l'indice *Idx1* perché l'attributo *A* non fa parte** dei filtri richiesti e pertanto essendo che l'indice si basa principalmente su quel attributo non viene considerato
    - **NB:** In questo caso *Idx1* eseguirebbe una scansione sequenziale completa della tabella

- **Variabili Esercizio**
  - ***R:*** numero di tuple totali all'interno della tabella
  - ***Val(attributo):*** numero di valori univoci per *attributo*
  - ***tuples to scan (on average)***

- **Risoluzione esericizio**

  1. Vengono scartati gli indici che
     - Non usano il primo attributo
     - Non usano gli attributi nelle prime posizioni
  2. Gli indici che non utilizzano gli attributi intermedi calcoleranno il ***tuples to scan (on average)*** fino all'ultimo attributo nella posizione iniziale prima del distacco, perchè poi quest'ultimi utilizzeranno una scansione forzata completa prima di arrivare agli attributi che sono presenti nelle posizioni finali
  3. Viene calcolato il ***tuples to scan (on average)*** per ogni indice scelto

$$\frac{R}{Val(attributo) \cdot Val(attributo)}$$

$$Idx2 = \frac{R}{Val(B) \cdot Val(C) \cdot Val(E)}$$

$$Idx2 = \frac{10.000.000}{100 \cdot 1000 \cdot 10} = 10$$

$$Idx3 = \frac{R}{Val(B) \cdot Val(D)}$$

$$Idx3 = \frac{10.000.000}{100 \cdot 2000} = 50$$

  4. viene ora scelto l'indice che ha come ***tuples to scan (on average)*** il numero inferiore ossia *Idx2*
     - *Questo vuol dire che data la combinazione di $B, C$ e $E$ vengono restituite 10 tuple dalla tabella che soddisfanno la query*

- ***tuples to scan (on average):*** 0.0004
  - La probabilitá di avere dei valori restituti dall'esecuzione della query utilizzando determinati attributi é molto bassa
  - Si preferisce in caso di uguaglianza di scegliere l'indice che presenta come attributi intermedi una laga scala di valori

## 9.4 Break even analysis

- La tabella é compostato dai blocchi e i blocchi a loro volta contengono delle tuple
- Fan out all'interno di un albero rappresenta
  - Per i nodi intermedi $\rightarrow$ *dei puntatori*
  - Per le foglie $\rightarrow$ *le tuple effettive*
- Se possediamo $n$ tuple totali abbiamo bisogno di $n$ puntatori
  - Il calcolo del *fan out* si svolge sul numero totale di tuple presenti nel file

$$profonditá = log_{\ fan\ out}\ (\ tuple\ totali\ per\ file)$$

$$fan\ out\ =\ \sqrt[profonditá]{tuple\ totali}$$

$$n\ tuple\ totali = n\ puntatori\ a\ quelle\ tuple$$

- **Variabili**
  - $T_{RA}$ = tempo random access
  - $T_{SA}$ = tempo sequential access
- **Sequentially ordered sequential structure**
  - Deve scansionare l'intera tabella a un costo fisso
    - La tabella é costituita da $n_{blocchi}$
    - Per cui dovremo andare a scansionare gli $n_{blocchi}$
  - *Al primo blocco si accede in maniera random, mentre gli* $n_{blocchi}$ *in maniera sequenziale, andiamo a togliere* $-1$ *che sarebbe il blocco di rifermento*

$$tempo\ tot = 1 \cdot T_{RA} + (n_{blocchi} - 1) \cdot T_{SA}$$

- **Answering via the B+ index**
  - I nodi foglia sono collegati in una catena nell'ordine imposto dalla chiave.
  - Valori numerici all'interno dell'intervallo → **N**
  - Numero puntatori/tuple per nodo (**fan out**) per nodo → **k**
    - Quindi 100 valori per ogni nodo foglia

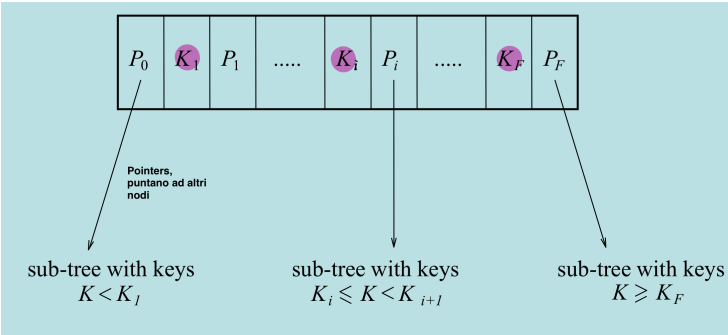



Figure 17: schema

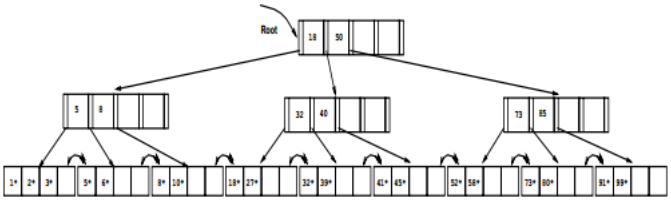



Figure 18: schema

  - L'accesso alla tupla di rifermento avviene
    - Raggiungimento prima foglia
    - Dato l'intervallo di interesse andiamo ad analizzare l'intera catena di valori
    - un $T_{RA}$ per ogni valore chiave, al fine di recuperare la tupla effettiva

$$tempo\ tot = deapth\ \cdot T_{RA} + \frac{N}{k} \cdot T_{RA} + N \cdot T_{RA}$$

$$tempo\ tot = (deapth + \frac{N}{k} + N) \cdot T_{RA}$$

– **Break even**

  * Tirare entrambe fuori in $T_{RA}$
  * Fare opportuni approssimazioni
  * *B+ tree* valore meglio sugli intervalli
  * *Hash bashed* sui valori random

$$tempo\ tot_{sequentially} = tempo\ tot_{B+}$$

Dal quale riusciamo a calcolare l'incognita $N$