Accesso a DB tramite prepared statement

Generalità

- I prepared statement sono gestiti attraverso l'**AgentData** della libreria **iaf/ia/iadata**
- I prepared statement riconosciuti dall'AgentData sono configurati in appositi file .json
- Ogni file json può contenere più di un prepared statement
- L'AgentData viene configurato andando ad indicare due parametri:
    - tpevodb: agente di database
    - path: path ove risiedono i file di configurazione
- Es:

```
<agent lib="/usr/lib/libiadata.so.0.0.0" create="agentData" live="1"
name="prepstat">
    <param name="agdb" value="tpevodb"/>
    <param name="path" value="/code/cpp/uml/proled/config/iadata"/>
</agent>
```

Key

- Ogni prepared statement è identificata attraverso la seguente codifica:
    - {Main Key}.{Sub Key}
        - {Main Key}: identifica il nome del file senza estensione
        - {Sub Key}: identifica lo specifico prepared statement all'interno del file
    - L'univocità a livello di Main Key è assicurata dal fatto che non possono esistere nella medesima cartella due file con lo stesso nome
    - L'univocità a livello di chiave completa {Main Key}.{Sub Key} deve essere assicurata dal configuratore, che non deve permettere che per lo stesso Main Key esistano due Sub Key uguali
    - Es:
        - nel file **topology.json** è presente un prepared statement codificato con **nodi**
        - la chiave di tale prepared statement è: **topology.nodi**

JSON

- Il file .json permette di configurare più di un prepared statement
- Si possono aggregare, nello stesso file, prepared statement che hanno fini comuni ad esempio
    - insieme di tabelle di database associate a certe entità logicamente correlate
    - insieme di operazioni comuni (select, update, insert, delete)
    - etc...
- Il file contiene un array di oggetti, ognuno dei quali ha i seguenti attributi:
    - **subkey**: chiave associata allo specifico prepared statement all'interno del file .json
    - **sql**: query sql che definisce il prepared statement
- Es: file operalang.json, abbiamo 3 prepared statement identificati come segue:
    - operalang.insert
    - operalang.update
    - operalang.select

```
[
    {
       "subkey":"insert",
       "sql":"insert into operalang (codice,descrizio) values
```

```
($[codice],$[descrizio]) returning *"
    },
    {
        "subkey":"update",
        "sql":"update operalang set descrizio=$[descrizio] where codice=$[codice]
returning *"
    },
    {
        "subkey":"select",
        "sql":"select * from operalang where codice=$[codice]"
    }
]
```

Parametri e Query

- I parametri di un prepared statement sono definiti in **forma espressiva** (attraverso nomi descrittivi) e non in **forma numerata**
- Questo modello permette di poter gestire i parametri in modo più intuitivo e semplificato, dal punto di vista della programmazione della chiamata
- L'AgentData si occupa di convertire la forma espressiva di ogni parametro nella corrispondente forma numerata (richiesta dal sistema di database)
- La sintassi utilizzata per definire un parametro in un prepared statement è la seguente:
  - **$[.]** ⇒ il nome del parametro è inserito tra parentesi quadre, precedute dal simbolo **$**
  - Es: **$[codice]**
  - *il nome del parametro, tra parentesi quadre, non deve essere necessariamente il nome di un campo di database, ma può essere un qualsiasi nome riconosciuto dall'utilizzatore*.
  - **Risulta però conveniente dal punto di vista progettuale e di uniformità nelle conoscenze dei sistemi, utilizzare i nomi dei campi di database**
- Di seguito un esempio di query in forma espressiva (da configurare nel sistema) e la corrispondente in forma numerata:

```
--Forma espressiva
SELECT * FROM nodi WHERE id=$[idnodo] AND codprg=$[codice progetto];

--Forma numerata
SELECT * FROM nodi WHERE id=$1 AND codprg=$2;
```

- Se un parametro è ripetuto più volte con lo stesso nome, verrà connesso con lo stesso numero
- Le query di insert, update, delete devono essere sempre seguite dall'espressione **returning** *, ad esempio:

```
INSERT INTO operalang (codice,descrizio) VALUES ($[codice], $[descrizio]) returning
*
```

AgentData e funzionalità

- L'AgentData si occupa di gestire i prepared statement sql del sistema; fonisce verso gli utilizzatori una interfaccia semplificata per l'accesso ai prepared statement
- L'agente fornisce la possibilità di eseguire più prepared statement appartenenti al medesimo file, in una unica chiamata
- Le query preconfezionate sono definite in appositi file json. Ogni file json e' un array di oggetti del tipo:
  - {"subkey":"xxx", "sql":"select ..."} ove subkey = una sotto-chiave della specifica prepared

statement, sql = sql del prepared statement.

Esecuzione dei prepared statement

- Al fine di eseguire un prepared statement si effettua una **ask** come segue:
  - **request** = **KEY** (chiave del prepared statement che si vuole eseguire, oppure Main Key)
  - **p1** = rec(*lista parametri*) (es: rec(codice,descrizio)) ove *lista parametri* contiene la lista dei nomi dei parametri richiesti nel prepared statement ed i rispettivi valori; il rec può essere costituito da più righe ad esempio nel caso di inserimento o update di più righe per la stessa tabella; in tal caso è necessario attivare esplicitamente la transazione
  - **nume** = 0 valore di default ⇒ operazione semplice (per nume > 0 vedi paragrafo Funzioni integrative)
  - **retval** = risultato della query (nel caso di query insert, update, delete si inserisce in fondo alla query "returning *", in modo che il sistema ritorni un json contenente il dato aggiornato o il dato cancellato)
  - Es
    - file **operacom.json**

```
[
    {
        "subkey":"select",
        "sql":"select * from operacom where codice=$[codice]"
    },
    {
        "subkey":"selectkey",
        "sql":"select * from operacom where key=$[key]"
    }
]
```

- **Chiamata**
  - agente: **AgentData**
  - tipo di chiamata: **ask**
  - request: **operacom.selectkey**
  - p1: **{"key":"errormsg"}**

- **Risposta**

```
[
  {
    "codice":3,
    "key":"errormsg",
    "tipo":"M"
  }
]
```

- Se **KEY (request) viene posta ad un valore di {Main Key}** (solo parte relativa al nome del file), **il sistema può restituire il risultato dell'esecuzione di più di un prepared statement** associato a più di una {Sub Key} presente nel file
  - **L'utilizzo diretto della Main Key nella request, permette di semplificare l'assegnazione dei permessi**
    - infatti i permessi utente potrebbero essere associati all'intero file piuttosto che alle singole prepared statement

Esecuzione multi-query implicita

- request = **{Main Key}**
- p1 = (**_subkey**, lista parametri) (singolo oggetto json ⇔ rec con una sola riga) ove
  - **_subkey** = la lista delle Sub Key a cui siamo interessati, divise da virgola senza spazi (è possibile, naturalmente, inserire una unica Sub Key)
    - se _subkey non viene specificato (vuoto, oppure assente) ⇒ l'agente restituisce tutte le subkey della Main Key
  - **lista parametri**: lista completa di tutti i parametri richiesti da ogni Sub Key
    - se due Sub Key hanno un parametro in comune (stesso nome espressivo), questo va specificato una sola volta nella lista ed il corrispondente valore verrà utilizzato da entrambe
- retval: rec(lista key) ove
  - lista key:
    - un attributo per ogni KEY richiesta, con il nome **{Main Key}.{Sub Key}**
    - il valore di ogni attributo è il JSON corrispondente all'esecuzione del prepared statement
- **Questo modello è particolarmente indicato per estrarre (select), con una unica chiamata, i dati da più tabelle in relazione tra loro**
- Es:
  - file **topology.json**

```
[
    {
        "subkey":"nodiprogetto",
        "sql":"select codice,id,tipoarm,h,hftmax,css from nodi where
codprg=$[codprg] and id=$[idnodo] order by id"
    },
    {
        "subkey":"ramiprogetto",
        "sql":"select codice,id,nodosx,nododx,lc,leq from rami where
codprg=$[codprg] and id=$[idramo] order by id"
    },
    {
        "subkey":"prgmezzi",
        "sql":"select * from prgmezzi where codprg=$[codprg] order by codice"
    }
]
```

- **Chiamata**
  - agente: **AgentData**
  - tipo chiamata: **ask**
  - request: **topology**
  - p1: **{"_subkey":"ramiprogetto,nodiprogetto","codprg":1748,"idnodo":1,"idramo":1}**

- **Risposta**

```
[
  {
    "topology.ramiprogetto":{
      "codice": 53730,
      "id": 1,
      "nodosx": 49641,
      "nododx": 49642,
      "lc": 350.0,
      "leq": 350.0
```

```
    },
    "topology.nodiprogetto":{
      "codice": 49641,
      "id": 1,
      "tipoarm": "A",
      "h": 27.0,
      "hftmax": 42.0,
      "css": 2826
    }
  }
]
```

Esecuzione multi-query esplicita

- request = **{Main Key}**
- p1 = (**_subkey**, **_params**) (una riga per ogni Sub Key) ove
  - **_subkey** = codice di una specifica Sub Key nel file della Main Key
  - **_params**: oggetto JSon contenente i parametri richiesti dalla specifica Sub Key
    - ogni Sub Key è analizzata indipendentemente dalle altre
- retval: rec(lista key) ove
  - lista key:
    - un attributo per ogni KEY richiesta, con il nome **{Main Key}.{Sub Key}**
    - il valore di ogni attributo è il JSON corrispondente all'esecuzione del prepared statement

Esempio

- Consideriamo il seguente file operacomins.json

```
[
    {
        "subkey": "operacom",
        "sql": "insert into operacom (codice,key,tipo) values
($[codice],$[key],$[tipo]) returning *"
    },
    {
        "subkey": "operamsg",
        "sql": "insert into operamsg (codcom,codlang,msg) values
($[codcom],$[codlang],$[msg]) returning *"
    },
    {
        "subkey": "operalang",
        "sql": "insert into operalang (codice,descrizio) values
($[codice],$[descrizio]) returning *"
    }
]
```

- Vogliamo inserire un nuovo **operacom** ed i relativi messaggi in **operamsg** per i vari linguaggi **operalang**, per cui utilizzeremo due subkey:
  - operacom: una riga per il nuovo tipo di messaggio
  - operamsg: una riga per ogni traduzione prevista nella specifica lingua
- Possiamo eseguire
  - **ask**
  - **request** = operacomins
  - **nume** = 2

- transazione: assicuriamo l'integrità dei dati nell'inserimento nelle tabelle operacom e operamsg
- pre-processing: vogliamo ottenere il numeratore per il campo codice di operacom e poi associare il codice rilevato alle righe di operamsg come valore di chiave esterna
  - {"_subkey":"operacom", "_params":[{ ... **"codice" : "_getnume"** ... },...]}
  - {"_subkey":"operamsg", "_params":[{ ... **"codcom" : "$[operacom.codice]"** ... },...]} (operacom di operacom.codice fa riferimento al nome della query e non al nome della tabella)

◦ **p1** è il seguente json

```
[
  {
    "_subkey":"operacom",
    "_params":{
      "codice":"_getnume",
      "tipo":"M",
      "key":"errmsg"
    }
  },
  {
    "_subkey":"operamsg",
    "_params":[
      {
        "codcom":"$[operacom.codice]",
        "codlang":"it",
        "msg":"E' tutto sbagliato"
      },
      {
        "codcom":"$[operacom.codice]",
        "codlang":"en",
        "msg":"All is wrong"
      },
      {
        "codcom":"$[operacom.codice]",
        "codlang":"es",
        "msg":"Todo es erato"
      }
    ]
  }
]
```

Applicazione Funzioni Avanzate

Indipendentemente dal tipo di query (monoquery, multiquery implicete o esplicite) è possibile richiedere al sistema una combinazione delle seguenti funzionalità:

- Transazione
- Preprocessing
- Merge dei risultati

Per attivare una o tutte le funzioni indicate bisogna utilizzare il **nume** secondo la tabella seguente:

Nume	Bit	Descrizione	Transazione	Preprocessing	Merge dei risultati
0	000	Nessuna Funzione			
1	001	Transazione	**X**		

Nume	Bit	Descrizione	Transazione	Preprocessing	Merge dei risultati
2	010	Preprocessing		**X**	
3	011	Transazione + Preprocessing	**X**	**X**	
4	100	Merge			**X**
5	101	Transazione + Merge	**X**		**X**
6	110	Preprocessing + Merge		**X**	**X**
7	111	Transazione + Preprocessing + Merge	**X**	**X**	**X**

Come si osserva dalla tabella le transazioni sono attivate con nume=1, il preprocessing con nume = 2, il merge con nume = 4; gli altri valori del nume combinano le tre funzioni in modo opportuno (vedi tabella delle funzioni avanzate sopra).

Transazioni

- **Questo modello è particolarmente indicato nel caso di applicazione di modifiche ai dati attraverso operazioni di insert, update, delete attuate su più tabelle in relazione tra loro**
  - I dati vengono preparati inserendo, per ogni Sub Key, i valori rilevati, nei rispettivi parametri attesi dal prepared statement
  - Si costruisce un JSon per ogni Sub Key, contenente i parametri
  - **Per assicurare l'integrità dei dati**
    - è possibile eseguire le operazioni all'interno di una transazione
    - a tale scopo ⇒ **nume = 1** esegue i prepared statement all'interno di una transazione
    - se anche una sola query non va a buon fine, il sistema esegue una rollback della transazione altrimenti esegue commit

Pre-Processing dei parametri

- Impostando nume = 2 si richiede all'agente di eseguire la transazione con preprocessing dei parametri
- L'agente interpreta il contenuto di ogni parametro e ricerca i seguenti valori:
  - **_getnume** ⇒ se un parametro ha il valore **_getnume**, l'utilizzatore intende sostituire il valore del parametro con un numertore richiesto al tpe_db
    - il numeratore richiesto sarà: **getnume({MainKey}.{SubKey}, nomeparametro)**
    - l'agente chiederà tanti numeratori quante sono le righe della tabella dei parametri e assegnerà ad ogni riga il rispettivo valore
    - **_getnume(tabnume,fldnume)** ⇒ si desidera estrarre il numeratore esattamente dalla tabnume, fldnume indicati tra parentesi (es: _getnume(rilievo,codice))
  - **_uuidv4** ⇒ se un parametro ha il valore **_uuid** (v4), l'utilizzatore intende sostituire il valore del parametro con una chiave uuid v4 generata dal sistema
    - la uuid viene generata con la classe iaf stringutil::generate_uuid_v4()
    - viene generata una uuid per l'attributo relativo, per ogni riga del rec dei parametri
    - la uuid generata ha una dimensione fissa di 37 caratteri
  - **_uuidv7** ⇒ se un parametro ha il valore **_uuidv7**, l'utilizzatore intende sostituire il valore del parametro con una chiave uuid v7 generata dal sistema (la chiave v7 rispetto a v4 permette un ordinamento cronologico in base al codice)
    - la uuid viene generata con la classe iaf stringutil::generate_uuid_v7()
    - viene generata una uuid per l'attributo relativo, per ogni riga del rec dei parametri
    - la uuid generata ha una dimensione fissa di 37 caratteri
  - **_uuidshort** ⇒ se un parametro ha il valore **_uuidshort**, si intende sostituire il parametro con una chiave uuid corta
  - **_uuidveryshort** ⇒ se un parametro ha il valore **_uuidveryshort**, si intende sostituire il parametro con una chiave molto corta della lunghezza di 10 caratteri
  - **$[X.Y]** ⇒ sostituisce il valore del parametro relativo, con il valore del parametro riferito dalla coppia

        - X=Sub Key della Main Key in elaborazione
        - Y=nome parametro della Sub Key
            - es: nodi.codice: X=nodi, Y=codice ⇒ sostituisce il valore del paramtro a cui è stato assegnato **nodi.codice**, con il valore del parametro **codice** specificato per la Sub Key **nodi**
        - Utile nei casi in cui si assegni una chiave esterna a RUN-TIME tramite la _getnume
    - **_datanow**, **_oranow**, **_timenow**, l'agente imposta rispettivamente
        - la data corrente (YYYYMMDD)
        - l'ora corrente (HH:MI:SE)
        - il time epoch corrente (secondi da 1 Gennaio 1970)

Merge dei risultati

- Funzionalità applicata dall'agente solo in caso di multiquery e non in caso di monoquery
- Esegue il merge di tutti i dataset di una multiquery in un unico dataset dove vengono accodati nella sequenza di esecuzione i dataset ottenuti
- Il **retval** prende in questo caso una struttura tabella che rappresenta il merge di tutti i risultati
- Particolarmente utile per fare il merge di dati provenienti da più database in una unica struttura tabellare

Altre funzioni di AgentData

Lista parametri

- Tramite **ask** è possibile richiedere la lista dei parametri di una {Main Key}.{Sub Key}:
    - **request = keyparams**
    - p1 = rec(key,_subkey), possiamo avere i seguenti casi
        1. key = chiave completa {Main Key}.{Sub Key} ⇒ il campo _subkey è trascurato
        2. key = Main Key e _subkey = vuoto ⇒ restituisce i parametri di tutti i _subkey della Main Key
        3. key = Main Key e _subkey = lista di subkey divisi da virgola ⇒ restituisce i parametri dei subkey specificati
    - nume = numero di righe nei parametri
    - retv = rec(parametri), per i tre casi
        1. un json con una colonna per ogni parametro il cui nome e' proprio il nome del parametro, il valore e' vuoto
        2. un json con una colonna per ogni subkey della Main Key identificata con {Main Key}.{Sub Key} contenente il json dei parametri della specifica key
        3. un json con una colonna per ogni subkey della Main Key indicata in _subkey, identificata con {Main Key}.{Sub Key} contenente il json dei parametri della specifica key
    - **request = keyparamsmqi (keyparams per multiquery implicite)**
    - p1 = rec(key,_subkey) come **keyparams**
    - retv = rec(parametri) restituisce in una unica riga tutti i parametri del sottoinsieme di key richieste senza replicazioni; questo parametro può essere utilizzato per le multi-query implicite

Reload del repository

- Tramite **tell** è possibile fare ricaricare dal repository di file json tutte le configurazioni, nel caso in cui vengano modificati:
    - request = reload
    - p1 = rec(ciao=ciao)
    - retv = true se l'operazione ha successo, false altrimenti

Costruzione automatica file JSon Insert / Update

Attraverso una chiamata **tell** all'agente sarà possibile creare automaticamente i prepared statement di:

- insert, update, delete

di apposite liste di tabelle. Di seguito vediamo i 3 casi.

Insert

- E' possibile costruire file json contenenti operazioni di insert per gruppi tabelle correlate, attraverso l'utilizzo delle seguenti chiamate tell (i file sono creati nel path definito in configurazione dell'agente):
  - request = tabinsert
  - p1 = rec(key,tabs) ove
    - key = Main Key (nome del file senza estensione da creare)
    - tabs = lista delle tabelle interessate separate da virgola senza spazi
  - nume = 1 ⇒ i nuovi dati vengono inseriti in append nel file (se esiste), altrimenti il file viene sovrascritto
  - retv = true se il file e' stato creato nel path di destinazione definito

Update

- E' possibile costruire file json contenenti operazioni di update per gruppi tabelle correlate, attraverso l'utilizzo delle seguenti chiamate tell (i file sono creati nel path definito in configurazione dell'agente):
  - request = tabupdate
  - p1 = rec(key,tabs) ove
    - key = Main Key (nome del file senza estensione da creare)
    - tabs = lista delle tabelle interessate separate da virgola senza spazi
  - nume = 1 ⇒ i nuovi dati vengono inseriti in append nel file (se esiste), altrimenti il file viene sovrascritto
  - retv = true se il file e' stato creato nel path di destinazione definito
  - La condizione di update (where codice=...) viene applicata come segue:
    - se esiste un campo codice, viene utilizzato nella codizione di update come segue
      - nome espressivo: _codice (codice preceduto da underscore)
      - condizione aggiunta: **where codice = $[_codice]**
    - se non esiste un campo codice, l'agente ricerca tutti i campi che iniziano con **cod** e applica tutte le condizioni
      - nome espressivo: _codXXX (codice di chiave esterna preceduto da underscore)
      - condizione aggiunta: **where codXXX = $[_codXXX] and codYYY = $[_codYYY] and ...**

Delete

- E' possibile costruire file json contenenti operazioni di delete per gruppi tabelle correlate, attraverso l'utilizzo delle seguenti chiamate tell (i file sono creati nel path definito in configurazione dell'agente):
  - request = tabdelete
  - p1 = rec(key,tabs) ove
    - key = Main Key (nome del file senza estensione da creare)
    - tabs = lista delle tabelle interessate separate da virgola senza spazi
  - nume = 1 ⇒ i nuovi dati vengono inseriti in append nel file (se esiste), altrimenti il file viene sovrascritto
  - retv = true se il file e' stato creato nel path di destinazione definito

- La condizione di delete (where codice=...) viene applicata come segue:
  - se esiste un campo codice, viene utilizzato nella codizione di update come segue
    - nome espressivo: _codice (codice preceduto da underscore)
    - condizione aggiunta: **where codice = $[_codice]**
  - se non esiste un campo codice, l'agente ricerca tutti i campi che iniziano con **cod** e applica tutte le condizioni
    - nome espressivo: _codXXX (codice di chiave esterna preceduto da underscore)
    - condizione aggiunta: **where codXXX = $[_codXXX] and codYYY = $[_codYYY] and ...**

Tutte le operazioni

- E' possibile creare un file json con tutte le operazioni (insert, update, delete) richiamando la seguente
  - request = **taballop**
  - p1 = rec(key,tabs) ove
    - key = Main Key (nome del file senza estensione da creare)
    - tabs = lista delle tabelle interessate separate da virgola senza spazi
  - nume = 1 ⇒ i nuovi dati vengono inseriti in append nel file (se esiste), altrimenti il file viene sovrascritto
  - retv = true se il file e' stato creato nel path di destinazione definito

Esempio Insert (stesso per update)

- **Chiamata**
  - agente: **AgentData**
  - tipo chiamata: **tell**
  - request: **tabinsert**
  - p1: **{"key":"nodimng","tabs":"nodi,nodisup,nodiatt"}**

Utilizzo di più database

- I file Json degli esempi precedenti non forniscono alcuna indicazione sull'agente di database da utilizzare
  - In questi casi l'agentdb utilizzato è quello specificato nell'AgentData
  - Tutti i prepared statement verranno quindi eseguiti attraverso tale agentdb
- E' possibile associare ciascuna prepared statement ad un differente agentdb
  - A tal fine si può aggiungere, opzionalmente, l'attributo **"db"** nel relativo oggetto prepared statement
  - L'AgentData utilizzerà l'agentdb di default (quello assegnato all'agente) per i prepared statement che non forniscono indicazione esplicita dell'attributo **db**
    - mentre per i prepared statement che forniscono l'attributo **db** l'AgentData eseguirà il prepared statement utilizzando l'agentdb indicato dall'attributo stesso
  - Questa possibilità è utile per:
    - recuperare dati da differenti database
    - inserire dati in differenti database
    - replicare dati da un database ad un'altro

Esempio Json per più database

- Ipotizziamo di:

    - dover inserire un nuovo utente in un sistema informativo
    - che il sistema informativo sia composto dai seguenti database:
        - userdb: database di gestione della protezione (nome dell'agentdb: **userdb**)
        - oeedb: database di gestione dell'oee (nome dell'agentdb: **tpevooee**)
        - mldb: database di gestione del machine ledger (nome dell'agentdb: **tpevoml**)
    - l'inserimento dell'utente deve avvenire su tutti i database indicati
- Vediamo di seguito un esempio di file JSON **opins.json**:

```
[
    {
            "db":"userdb",
        "subkey":"operato",
        "sql":"insert into operato (codice,pwd,nominati,email) values
($[codice],$[pwd],$[nominati],$[email]) returning *"
    },
    {
        "db":"tpevooee",
        "subkey":"operato_oee",
        "sql":"insert into operato (codice,nominati,email) values
($[codice],$[nominati],$[email]) returning *"
    },
    {
        "db":"tpevoml",
        "subkey":"operato_ml",
        "sql":"insert into operato (codice,nominati,email) values
($[codice],$[nominati],$[email]) returning *"
    }
]
```

- Attraverso una unica richiesta di esecuzione di multi-query implicita è possibile inserire l'utente in tutte e tre i database
    - si assegnano gli attributi codice, pwd, nominati, email dell'utente
    - si esegue la ask con request = operains
    - si pone nume = 1 per attivare le transazioni
        - l'AgentData avvierà una transazione per ogni database, in modo da garantire che tutte le query vadano a buon fine, altrimenti annullerà la transazione su tutti i db assicurando cosi l'integrità dei dati
- **Esempio Chiamata**
    - agente: **AgentData**
    - tipo chiamata: **ask**
    - request: **opins**
    - p1: **{"codice":"n.berga","pwd":"aioaio","nominati":"Nicola Bergantino","email":"n.bergantino@youhh.com"}**
    - nume: **1**

Parameters Augmentation

L'AgentData implementa un algoritmo che permette di completare i parametri di una prepared statement, non forniti dal richiedente:

- utilizzando valori default specificati in configurazione
- andando a prelevare i dati dal database attraverso una prepared statement

- impostando i dati a NULL

Di seguito la specifica dell'algoritmo:

- Data una prepared S sia:
  - P l'insieme dei parametri richiesti da S
  - Q l'insieme dei parametri forniti dal richiedente
  - sia M = P - Q l'insieme dei parametri presenti in P ma non forniti in Q
  - se M = 0 significa che tutti i parametri P richiesti da S sono stati forniti in Q
- Se Q ha una sola riga e M > 0, l'AgentData applica un algoritmo di parameters augmentation
  - tale algoritmo cerca di assegnare automaticamente i parametri in M, come segue:
    - applicazione dei parametri di default definiti nella subkey
    - impostazione dei parametri a partire dal risultato derivante dall'esecuzione di altre prepared statement (key)
    - impostazione dei parametri a valori NULL
  - Siano S(1), S(2), ..., S(n) le prepared statement da cui prelevare i parametri
  - Siano R(1), R(2), ..., R(n) i risultati dell'esecuzione delle n prepared statement
  - M = M - R(1) - R(2) - ... - R(n) è l'insieme dei parametri rimasti inassegnati dopo l'applicazione di R(1), ..., R(n)
  - Se M > 0 ed è richiesto il setting a NULL
    - l'algoritmo assegna tutti i parametri in M al valore NULL
- Per poter applicare l'algoritmo di parameters augmentation è necessario utilizzare due attributi dell'oggetto json corrispondente alla prepared statement:
  - **paramsdef**: è un oggetto JSON con un attributo per ogni parametro con il relativo valore di default
  - **paramsql**: specifica la lista delle **key** da eseguire per recuperare i parametri (lista di elementi divisi da virgola senza spazi)
  - **paramsnull**: se posto a **1** indica che si richiede il settaggio a NULL dei parametri residui (dopo l'applicazione di P e di R(1), ..., R(n))
- **Esempio**
  - vogliamo inserire una nuova linea di produzione
  - creiamo una mainkey = linea e il corrispondente file linea.json
  - per completare l'inserimento di una linea dobbiamo inserire:
    - ubicaz: tutti i dati richiesti, con setting automatico a NULL di quelli non forniti
    - lottim: solo il campo codice, non richiesto l'algoritmo di parameters augmentation
    - oeeconf (ORA): oeeconf orario, si preleva la configurazione da valori default
    - oeeconf (DAY): oeeconf giornaliero, si preleva la configurazione da oeeconf.codice="day"
  - di seguito linea.json

```
[
{"subkey":".","sql":".","paramsnull":0,"paramsql":".","paramsdef":{},"events":{}},
    {
        "subkey":"ubicaz",
        "sql":"insert into ubicaz
(codice,descrizio,identif,tiposet,tipocom,tipodepo,codriso,impianto,utente,datareg,
stato,codcate,codoper,locked,codqua,nqua,codubib,capacita,occupazione,pos,lim1,lim2
,curop,codturno,agpausesap,plmaster,plslave,updw,kbterm,blink,visnum,codfase,codazi
,schema,costo,defincdec,oeew,opts,thmonoee,thmonsetup,thmondownt,thmonscrap,thdayoe
e,thdaysetup,thdaydownt,thdayscrap,laps,linubi,defnpop,deftcpol,saveoeeshift,tmstar
t,started,mancost,amcilr,amsop,storcompo,supname,locname,extcost,tmstartto,ystart,y
startto,codtunext,tmtunext,tmtu) values
($[codice],$[descrizio],$[identif],'P',$[tipocom],$[tipodepo],$[codriso],$[impianto
],$[utente],$[datareg],$[stato],$[codcate],$[codoper],$[locked],$[codqua],$[nqua],$
[codubib],$[capacita],$[occupazione],$[pos],$[lim1],$[lim2],$[curop],$[codturno],$[
agpausesap],$[plmaster],$[plslave],$[updw],$[kbterm],$[blink],$[visnum],$[codfase],
$[codazi],$[schema],$[costo],$[defincdec],$[oeew],$[opts],$[thmonoee],$[thmonsetup]
,$[thmondownt],$[thmonscrap],$[thdayoee],$[thdaysetup],$[thdaydownt],$[thdayscrap],
```

```
$[laps],$[linubi],$[defnpop],$[deftcpol],$[saveoeeshift],$[tmstart],$[started],$[mancost],$[amcilr],$[amsop],$[storcompo],$[supname],$[locname],$[extcost],$[tmstartto],$[ystart],$[ystartto],$[codtunext],$[tmtunext],$[tmtu]) returning *",
        "paramsnull":1
    },
    {
        "subkey":"lottim",
        "sql":"insert into lottim (codice) values ($[codice]) returning *",
    },
    {
        "subkey":"oeeconf_hourly",
        "sql":"insert into oeeconf
(codice,codriso,codperiodo,durperiodo,nrperiodi,tmdisp,tpriso,tmlast,codmm,codrisodef,bgnper,availab,calcnop,kweek,lim1,lim2,mudoee,dircalcsca,disabled,nptots) values
($[codice] ||
$[codperiodo],$[codice],$[codperiodo],$[durperiodo],$[nrperiodi],$[tmdisp],$[tpriso],$[tmlast],$[codmm],$[codrisodef],$[bgnper],$[availab],$[calcnop],$[kweek],$[lim1],$[lim2],$[mudoee],$[dircalcsca],$[disabled],$[nptots]) returning *",
        "paramsdef":{
"codice":"ora","codperiodo":"ORA","durperiodo":3600,"nrperiodi":24,"tmdisp":3600,"tpriso":"M",
"tmlast":"_timenow","codmm":0,"codrisodef":"","bgnper":0,"availab":1,"calcnop":1,"kweek":"",
              "lim1":0,"lim2":0,"mudoee":"","dircalcsca":1,"disabled":1,"nptots":0
            }
    },
    {
        "subkey":"oeeconf_daily",
        "sql":"insert into oeeconf
(codice,codriso,codperiodo,durperiodo,nrperiodi,tmdisp,tpriso,tmlast,codmm,codrisodef,bgnper,availab,calcnop,kweek,lim1,lim2,mudoee,dircalcsca,disabled,nptots) values
($[codice] ||
$[codperiodo],$[codice],$[codperiodo],$[durperiodo],$[nrperiodi],$[tmdisp],$[tpriso],$[tmlast],$[codmm],$[codrisodef],$[bgnper],$[availab],$[calcnop],$[kweek],$[lim1],$[lim2],$[mudoee],$[dircalcsca],$[disabled],$[nptots]) returning *",
        "paramsql":"ubicazdef.oeeconfd"
    }
]
```

- le key ubicazdef.oeeconfh e ubicazdef.oeeconfd sono presenti nel file ubicazdef.json, come segue

```
[
    {
        "subkey":"oeeconfh",
        "sql":"select * from oeeconf where codice='ora'"
    },
    {
        "subkey":"oeeconfd",
        "sql":"select * from oeeconf where codice='day'"
    }
]
```

- L'inserimento di una linea di produzione può essere svolta attraverso la seguente chiamata
  - **Chiamata**
    - agente: **AgentData**
    - request: **linea**

- p1: **{"codice":"test1","descrizio":"test1","tiposet":"P"}**
- nume: **2**

IMPORTANTE : i parametri aggiunti con parameters augmentation possono essere soggetti a pre-processing dei parametri; ad esempio in un parametro default potrei impostare il valore di preprocessing **_getnume**

Events

- E' possibile associare ad ogni prepared statement un evento da inviare ad un agente qualora la prepared statement fosse stata eseguita con successo (es: un inserimento, una modifica, ...).
- E' possibile configurare uno (singolo oggetto) o più agenti (array json) destinatari dell'evento.
- A seguito dell'esecuzione con successo del prepared statement l'AgentData verifica se vi è associato un evento, in tal caso richiama la tell di ogni agente configurato, con la rispettiva request.
  - La tell eseguita fornirà nell'attributo "p1" della richiesta l'oggetto JSON (myrec) risultante dall'esecuzione del prepared statement (es: nel caso di isnert, delete, update sarà il valore di "returning *", mentre nel caso di select, sarà il risultato della select).
- Per configurare gli eventi è sufficiente integrare nel JSON del prepared statement il seguente attributo:
  - **events** = oggetto json o array json, cosi composto:
    - **to** = nome dell'agente destinatario.
    - **request** = request.
    - **param** = l'attributo param definisce il valore di tipo stringa che viene inserito nel "p2" della richiesta all'agente destinatario. Se non specificato il "p2" viene impostato a NULL, altrimenti viene impostato al valore specificato.
- Esempio di configurazione evento:

```
[
    {
        "subkey":"operacom",
        "sql":"insert into operacom (codice,key,tipo) values
($[codice],$[key],$[tipo]) returning *",
        "paramsdef":{"codice":"_getnume"},
    "events":{"to":"mngevent","request":"newopera","param":""}
    },
    {
        "subkey":"operamsg",
        "sql":"insert into operamsg (codcom,codlang,msg) values
($[codcom],$[codlang],$[msg]) returning *",
        "paramsdef":{"codcom":"$[operacom.codice]"}
    }
]
```

- Se nell'attributo **to** dell'oggetto events, si inserisce la seguente istruzione **$me** l'agente invierà la chiamata a se stesso (esempio per attivare certe operazioni su DB a seguito di certe query)

Riga dummy

- Nel caso in cui, per il primo oggetto del file JSON, non si specifichino gli attributi opzionali (*db, paramsql, paramsnull*) e se questi attributi fossero poi forniti gli oggetti successivi nell'array, allora è necessario indrodurre una **riga dummy**
- La **riga dummy** permette all'algoritmo di conversione da JSON a myrecson della IAF di intercettare tutti gli attributi degli oggetti presenti

- La **riga dummy** va inserita come prima riga del json, di seguito un esempio:

```
[
{"subkey":".","sql":".","paramsnull":0,"paramsql":".","paramsdef":{},"events":{}},
    {...}
]
```

Generazione di csv da Prepared Statement

- Utilizzando la tell è possibile eseguire qualsiasi prepared statement come con la ask
- In questo caso l'agente ritorna True se la chiamata multi-mono query restituisce un JSON altrimenti ritorna False
- E' possible generare file CSV a partire da una prepared statement, basta aggiungere nel JSON della richiesta i seguenti attributi:
    - _csvfile: path completo del file csv
    - _decimalpoint: (opzionale) si puo' indicare il carattere separatore dei decimali
    - _separator: (opzionale) si può indicare il carattere separatore dei campi nel file csv (default: ;)
- Se presente l'attributo _csvfile, l'agente crea il file csv

Attributi utili di una subkey

Rinominare nomi di campi

Con l'attributo **renamemap** del file JSON è possibile rinominare nomi dei campi della risposta di un prepared statement in modo arbitrario, ad esempio per stampare un CSV con nomi di campi significativi.

Vediamo il seguente esempio:

```
[
  {
        "subkey":"stab03",
        "db":"terranuovadb",
        "sql":"with d as (select d.* from dimension d inner join oeeconf o on
o.codriso=d.d1 where tm>$[tmbgn] and tm<$[tmend] and d2='ORA' and o.disabled='0'
and o.codperiodo='ORA'),e as (select u.descrizio as linea,round(sum(m_rep_nptc-
m_rep_stc)/60) as lavorato,round(sum(m_rep_d)/60) as previsto from d inner join
fact f on f.codice=d.kfact inner join ubicaz u on u.codice=d.d1 group by
u.descrizio) select  'Terranuova' as dplant,*,round((case when previsto>0 then
lavorato/previsto else 0 end)::numeric, 2) as e from e order by linea",
        "renamemap":{"dplant":"Stabilimento","linea":"Centro di
lavoro","lavorato":"Tempo Lavorato (min)", "previsto":"Tempo Previsto (min)",
"e":"Efficienza (%)"}
    }
]
```

In questo caso **renamemap** rinomina i campi *dbplant, linea, lavorato, previsto, e* assegnando nomi esplicativi.

Normalizzare query lunghe

Per poter normalizzare query lunghe in cui si adoperano ritorni a capo e spazi per poterle leggere meglio nel file

JSON si può attivare l'attributo **normalize**.

L'attributo **normalize** se posto a 1 attiva un algoritmo di normalizzazione della stringa SQL.

Questa operazione è particolarmente utile per la visualizzazione dei LOG delle stringhe lunghe.

```
{
        "subkey":"ubigiac",
        "sql":"SELECT
                g.codubi,
                l.codart,
                ca.tipopal as codpal,
                ug.xabs as x,
                ug.yabs as y,
                ug.zabs as z,
                CASE WHEN tp.stackable = 1 then 2 else 1 END as npallet,
                u.tipocom,
                u.tiposet,
                0.0 as a,
                0.0 as b
            FROM
                giacese g JOIN lottim l on l.codice=g.codlot
                JOIN catego ca ON l.codart = ca.codice
                JOIN ubicaz u on u.codice=g.codubi
                JOIN tipopal tp on tp.codice = l.umext
                LEFT JOIN ubigeo ug on ug.codubi = g.codubi
            WHERE
                g.codubi=$[codubi]
            GROUP BY
                g.codubi,
                l.codart,
                u.tipocom,
                u.tiposet,
                ca.tipopal,
                ug.xabs,
                ug.yabs,
                ug.zabs,
                tp.stackable
            ORDER BY
                max(g.codice)",
        "paramsdef":{"codubi":"LGV"},
        "normalize":1
    }
```

La normalize riduce tutti gli spazi doppi o i ritorni a capo o i segni di tabulazione ad un unico spazio, in modo da avere la query su una unica riga.

IMPORTANTE: Utilizzare con attenzione nel caso in cui gli spazi servano per impostare valori o condizioni.

Q&A

- E' possibile attivare la procedura di parameters augmentation per settare automaticamente attributi default indicati espressamente per valore oppure che utilizzano i valori default del campo nel db ? (tipo il

null)
    - **SI** basta utilizzare l'attributo **paramsdef** dell'oggetto subkey specifico (vedi Parameters Augmentation)
- Come va gestito nel prepared statement le query con condizione where che richiede che un parametro sia parte di un certo insieme di parametri ? (esempio: where codice in ('A', 'B', ...) and ...)
    - di seguito la risposta:

```
[
        ...
        {
        "subkey":"ubiany",
        "sql":"select * from ubicaz WHERE codice = ANY(string_to_array($[lista
codici], ','))"
    },
        ...
]
```

- Come gestire un prepared statement con una like ?
    - di seguito la risposta:

```
[
  ...
    {
        "subkey":"ubilike",
        "sql":"select * from ubicaz WHERE codice like $[codice] || '%'"
    }
]
```

- Nel caso in cui una richiesta multi-query non sia vincolata da una transazione è possibile eseguire le query in parallelo ?
    - *Nella versione attuale le query parallele sono possibili con le multi-query implicite che non prevedono transazioni; per farlo sarebbe necessario installare le librerie omp per il threading implicito*
- Come potremo orientarci all'interno dei vari path di file json quando le query saranno davvero tante ? Come faremo a capire se una query è già presente ? come faremo a indiviuare dove si torva una mainkey o una subkey ? Come potremo testarle o inserirne di nuove ?
    - E' necessario costruire un agente che si occupa di:
        1. tracciare tutte le key presenti nel sistema fornendo:
            1. commento, sql, path
        2. ricercare una key, subkey
        3. ricercare in modo intelligente un commento, una sql
        4. inserire / modificare / testare nuove key o nuove mainkey