POLITECNICO DI MILANO

Facoltà di Ingegneria dell'Informazione

Corso di Laurea Specialistica in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

DEFINIZIONE DI UNA STRUTTURA AD OGGETTI PER LA RAPPRESENTAZIONE INTERNA DELLE INTERROGAZIONI DEL LINGUAGGIO PERLA

Candidato: Alessandro PERRUCCI matr. 662787

Relatore: Chiarissimo Professor Fabio A. SCHREIBER

Anno Accademico 2006/2007

Indice

Introduzione

Nel luglio 2005 un gruppo di università italiane, coordinate dal Politecnico di Milano, ha dato vita al progetto ART DECO (Adaptive InfRasTructures for DECentralized Organizations) [1] [2]. Finanziato dal Ministero dell'Università e della Ricerca, il progetto mira allo sviluppo di tecniche e strumenti per la diffusione delle "networked enterprise" tra le piccole e medie imprese italiane.

Uno dei casi di studio di ART DECO riguarda il processo di produzione e trasporto dei vini di qualità da parte di una importante azienda vinicola. Per garantire la qualità ed integrità del prodotto, l'azienda può aver interesse a mantenere sotto controllo alcuni importanti parametri del ciclo produttivo, come ad esempio la temperatura delle uve in vigna o l'umidità del terreno, ma anche le vibrazioni a cui le bottiglie di vino sono sottoposte durante il trasporto. Inoltre si vorrebbe garantire la completa tracciabilità degli interventi effettuati su ciascuna partita di vino prodotta, dalla raccolta dell'uva fino alla fase di etichettatura.

Per soddisfare le richieste dell'azienda è necessaria una vasta rete di sensori di tipo eterogeneo [3], dalle reti di sensori wireless per monitorare la temperatura del prodotto in ogni fase della sua vita, alle etichette RFID da applicare sulle bottiglie per identificarle durante il trasporto, fino ai dispositivi GPS per conoscerne la posizione in ogni momento. I dati così ottenuti, oltre a consentire di rilevare tempestivamente sbalzi dei parametri di rilievo, assicurandone la qualità, consentono, incrociandoli tra loro, di fornire agli acquirenti le informazioni sulla genesi di ogni singola bottiglia di vino.

A causa dell'elevato numero di tecnologie in gioco, ognuna con una sua interfaccia di controllo e di interrogazione, la quantità di lavoro e conoscenze richiesta all'utente finale per poter recuperare ed incrociare i dati provenienti da sensori di tipo eterogeneo sarebbe tale da rendere in pratica il sistema difficilmente utilizzabile. Quindi ci si è posto il problema di renderne omogenea la visione da parte dell'utente finale.

Una soluzione a questo problema è la definizione di un linguaggio di alto livello completamente dichiarativo, che consenta all'utente di interrogare un sistema pervasivo in modo molto simile a come interrogherebbe una base di dati [4].

Un progetto del Politecnico di Milano ha definito un linguaggio, di nome PERLA (PERvasive LAnguage) [5], che consente l'interrogazione di un sistema pervasivo utilizzando una sintassi simile a quella dell'SQL standard.

Il vantaggio nell'utilizzare questo linguaggio è evidente: la complessità del sistema è completamente nascosta all'utente, che non si dovrà preoccupare delle caratteristiche tecniche dei singoli sensori, ma potrà concentrarsi sulle informazioni da essi ottenute.

Il punto di partenza per la realizzazione del progetto è stato TinyDB [6]. Il progetto però ha limitazioni importanti, e ciò ha spinto a realizzare un nuovo linguaggio.

Nel capitolo 1 si presenterà il progetto e se ne descriveranno le motivazioni e gli obiettivi. Verrà inoltre introdotto il linguaggio PERLA e si elencheranno le principali motivazioni che hanno spinto alla realizzazione di un nuovo linguaggio.

Per descrivere la semantica del linguaggio è stata adottata una infrastruttura creata dal Politecnico di Milano e costituita da tre livelli ognuno dei quali, a partire dai dispositivi fisici reali, fornisce un livello di astrazione sempre maggiore. Nel capitolo 2 verrà fornita una breve descrizione dell'architettura e dei livelli dai quali è costituita, soffermandosi sui componenti di maggiore importanza per questo progetto.

Nel capitolo 3 si elencheranno le caratteristiche principali del linguaggio PERLA, e in particolare la gestione delle strutture dati e dei diversi tipi di query ammesse. Inoltre verrà introdotta l'importante funzione *pilot join*.

Il capitolo 4 rappresenta il cuore del presente elaborato. Al suo interno verrà analizzato il funzionamento di ogni singola clausola del linguaggio e, a partire dalla grammatica in forma EBNF, si spiegherà il modo in cui le classi relative possono essere realizzate. Per ogni clausola si fornirà inoltre il diagramma delle classi.

Nel capitolo 5 si fornirà un esempio di traduzione di una query in una opportuna struttura ad oggetti. Seppur semplice, la query proposta rappresenta un esempio realistico del caso di studio relativo al monitoraggio della produzione di vini. Si illustrerà quindi uno schema che mostra tutti gli oggetti istanziati dal parser e i valori dei loro attributi.

Il capitolo conclusivo spiega come il progetto è stato implementato in Java e descrive lo stato dell'arte attuale, spiegando inoltre quali progetti saranno sviluppati prossimamente per arrivare ad un prodotto funzionante.

Obiettivi del progetto

L'obiettivo principale del progetto è la definizione di una struttura a oggetti in grado di supportare l'esecuzione delle interrogazioni del linguaggio PERLA.

PERLA (PERvasive LAnguage) [5] è un linguaggio completamente dichiarativo che permette all'utente di interrogare un sistema pervasivo in modo simile a come interrogherebbe una base di dati utilizzando SQL. Un sistema pervasivo è una grande rete eterogenea composta da diversi dispositivi, ognuno dei quali può utilizzare diverse tecnologie, come ad esempio reti di sensori wireless (WSN), sistemi RFID, GPS e molti altri tipi di sensori.

Per linguaggio dichiarativo si intende un linguaggio con cui si stabilisce che cosa deve fare il programma, cioè che relazione intercorre fra ingresso e uscita, a differenza dei linguaggi imperativi, con i quali si stabilisce come un problema deve essere risolto, cioè come derivare un'uscita da uno o più ingressi.

Lo scopo principale del linguaggio PERLA è quindi nascondere all'utente l'elevata complessità del sistema e della programmazione di basso livello, fornendogli una interfaccia simile a quella di una base di dati, consentendogli di recuperare i dati dal sistema in modo semplice e veloce.

Il linguaggio PERLA è stato pensato per essere eseguito al di sopra di una architettura costituita da tre livelli: un livello di accesso fisico ai dispositivi, un livello degli oggetti logici e un livello applicativo. All'interno di quest'ultimo è presente il parser: esso è il componente che riceve le query sottoposte dall'utente in formato di stringhe, e ne verifica la correttezza sintattica. Obiettivo del parser è quello di tradurre ciascuna query in una struttura ad oggetti semanticamente equivalente, ma che risulti facilmente utilizzabile dal motore di esecuzione della query.

Lo scopo di questo progetto è quindi la progettazione e la realizzazione di una libreria di oggetti che, istanziati dal parser, forniscano una rappresentazione interna delle query sottoposte dall'utente.

Introduzione al linguaggio e motivazioni

Prima di presentare le caratteristiche del linguaggio è opportuno introdurne la storia, insieme alle motivazioni che hanno portato alla sua definizione. Infatti non è la prima volta che si tenta di astrarre una rete di sensori wireless per potervi interagire come se si trattasse di una base di dati.

Il più famoso progetto di questo tipo è TinyDB [6], la cui importanza deriva dall'esser stato il primo tentativo di estrarre informazioni da una rete di sensori wireless, utilizzando l'approccio delle basi di dati: esso permette infatti all'utente di interrogare la rete tramite un linguaggio simile ad SQL.

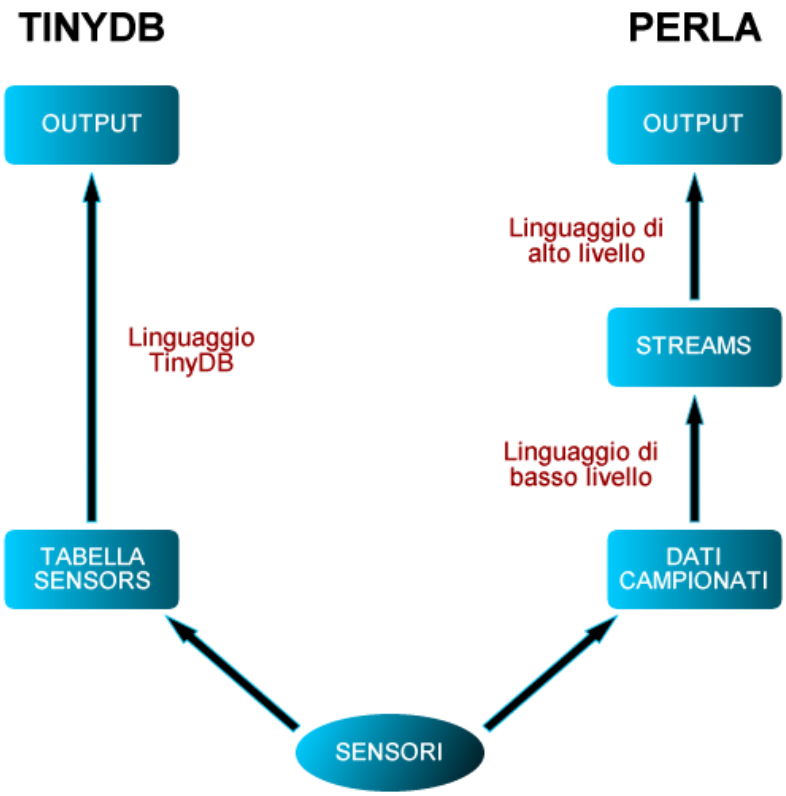
In ogni implementazione di TinyDB è presente una tabella dei sensori di nome *sensors*, di dimensione variabile, che contiene i campionamenti dei sensori provenienti da tutti i nodi. Ogni attributo della tabella si riferisce ad un tipo di sensore (per esempio temperatura, pressione, ecc.) e sono inoltre presenti un attributo per identificare univocamente un nodo della rete, e un attributo per il timestamp. Ogni record della tabella è il risultato di un campionamento eseguito da un nodo della rete. Se un sensore non è presente in un nodo, tutti i record che si riferiscono a quel nodo conterranno un valore nullo nel campo corrispondente all'attributo. Il risultato di una query sulla tabella dei sensori può essere memorizzato in una tabella di materializzazione, di dimensione fissa.

Nonostante l'idea di Madden sia molto innovativa, il progetto ha alcune limitazioni nell'implementazione, che lo rendono nella pratica un prodotto difficilmente utilizzabile.

Anche PERLA utilizza un'astrazione che consente di vedere una rete di sensori wireless come un database, ma ci sono alcune differenze rispetto a quella utilizzata da TinyDB. Una prima importante differenza fa riferimento alla classe di dispositivi supportati dai due linguaggi: TinyDB può essere eseguito su di una WSN omogenea di nodi che lavorano con sistema operativo TinyOS, mentre PERLA è stato pensato per poter operare contemporaneamente con diverse classi di dispositivi (dagli RFID ai nodi di una rete di sensori, fino a nodi più complessi, quali i PDA).

La seconda differenza riguarda il processo di campionamento: alla base di PERLA c'è l'idea che il concetto di campionamento non possa essere completamente nascosto all'utente. TinyDB astrae il processo di campionamento tramite il concetto di tabella dei sensori, senza permettere all'utente di modificare i parametri di campionamento, a parte il periodo. Ciò rende il sistema facile da capire e utilizzare per l'utente, ma adatto solamente a situazioni in cui i dispositivi della rete sono di tipo omogeneo e le loro caratteristiche sono già note durante la scrittura del programma.

In un sistema eterogeneo l'operazione di campionamento deve essere gestita in modo più complesso, dando all'utente la possibilità di definire criteri complessi per decidere quali nodi dovranno eseguire il campionamento. Proprio per l'importanza che PERLA riserva alla fase di campionamento, il linguaggio è suddiviso in due parti: una di basso livello che gestisce le operazioni di campionamento, e una di alto livello che manipola i dati campionati per produrre il risultato dell'interrogazione. Ciò consente una maggiore flessibilità e potenza del linguaggio, a scapito della semplicità d'uso.

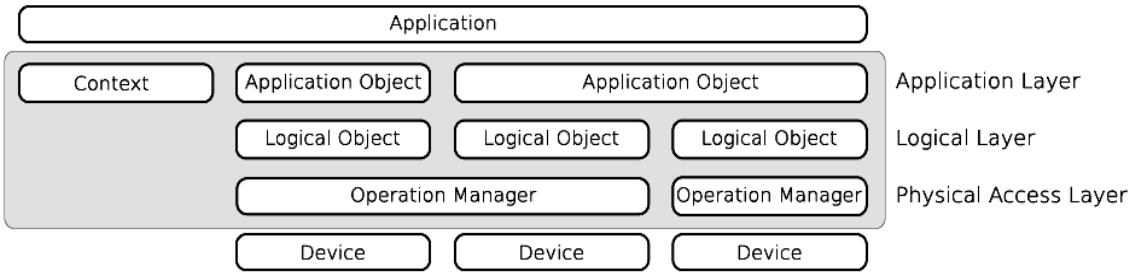


Architettura per sistemi pervasivi

In questo capitolo verrà introdotta l'architettura per sistemi pervasivi [7] sulla quale il linguaggio PERLA è eseguito, e verranno presentati i tre livelli di astrazione forniti dall'architettura. Verrà inoltre spiegato il modo in cui il linguaggio è integrato con l'architettura esistente.

Introduzione all'architettura

L'architettura presentata ha due obiettivi principali: il primo è la definizione di una astrazione per i diversi dispositivi che costituiscono il sistema pervasivo, mentre il secondo è l'integrazione dei dati provenienti dai dispositivi.



La figura mostra i tre strati introdotti per descrivere i dispositivi con 3 diversi livelli di astrazione:

- Livello di accesso fisico ai dispositivi, costituito dal software per l'accesso ai dispositivi;
- Livello degli oggetti logici, costruito sul livello precedente, che fornisce l'astrazione di un dispositivo singolo o di un aggregato omogeneo di dispositivi fisici;
- Livello applicativo, che definisce le astrazioni a livello dell'applicazione usate dai programmi applicativi per accedere ai dati dei dispositivi fisici.

I componenti software che popolano ogni livello comunicano con il livello superiore inviando eventi e con quello inferiore invocando comandi sincroni. La

comunicazione può avvenire solo tra livelli adiacenti; inoltre gli elementi del livello più alto possono comunicare tra loro.

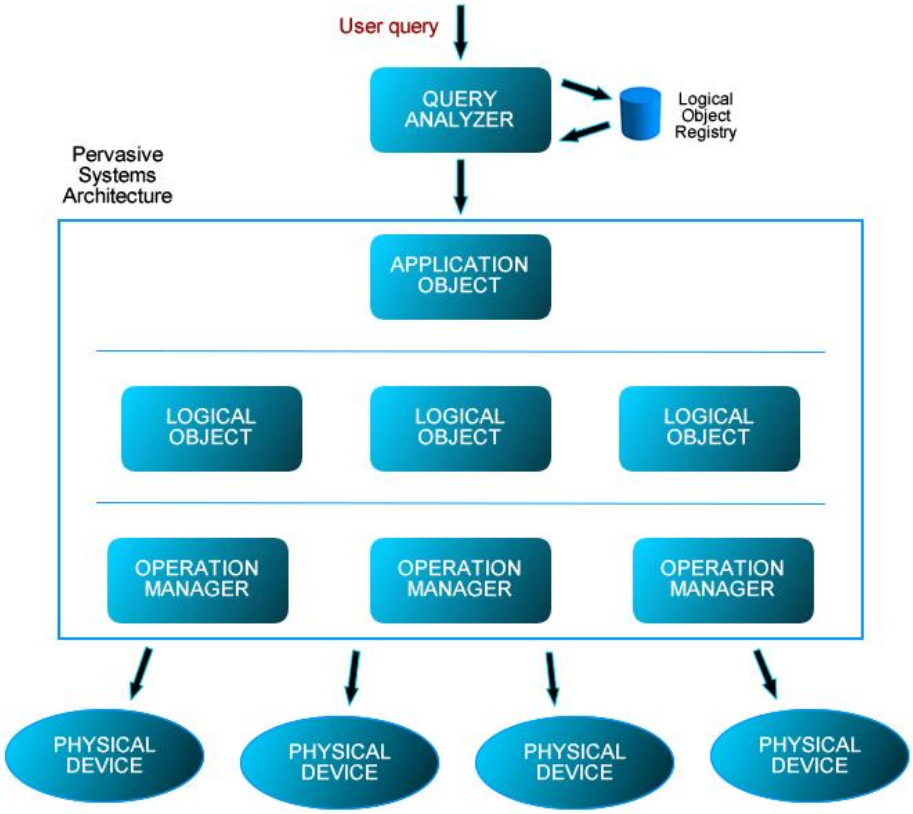
Ogni livello contiene una serie di componenti. Verranno descritti solo quelli importanti per questo progetto, omettendo tutti gli altri componenti software definiti nell'architettura.

I componenti del livello di accesso fisico ai dispositivi sono i punti di accesso ai dispositivi. Un sistema pervasivo contiene dispositivi di tipo eterogeneo, ognuno dei quali utilizza una determinata tecnologia e richiede un software specifico per accedere ad esso. L'obiettivo di un *operation manager* è gestire la comunicazione con un dispositivo appartenente ad una specifica tecnologia fornendo una interfaccia di accesso standard, così che tutte le tecnologie di dispositivi abbiano la stessa interfaccia.

Nel livello superiore esistono gli oggetti logici, che forniscono una virtualizzazione di ogni singolo dispositivo o di un gruppo omogeneo di essi, e mettono a disposizione i dati recuperati tramite il gestore delle operazioni corrispondente alla tecnologia del dispositivo considerato. Gli oggetti logici devono esporre una interfaccia standard che consenta di recuperare i dati campionati e l'identificativo dell'oggetto, ma anche di ricevere eventuali eventi di notifica.

I dati contenuti negli oggetti logici vengono a loro volta recuperati dagli oggetti applicativi. Essi si trovano nel livello applicativo, e definiscono le operazioni che possono essere invocate dall'esterno e gli eventi che possono essere sollevati.

Le interrogazioni sono gestite da un componente dedicato, il *query analyzer*, posto sopra il livello applicativo. Questo componente accetta in ingresso una interrogazione e ne fa il parsing; se essa non contiene nessun errore sintattico è necessario recuperare l'insieme di oggetti logici coinvolti nell'esecuzione dell'interrogazione. Per fare ciò il parser utilizza un componente dedicato posto nel livello applicativo, il registro degli oggetti logici, che mantiene un elenco degli oggetti logici presenti nel sistema. Infine l'interrogazione è suddivisa in sottointerrogazioni, che vengono inviate agli esecutori. I risultati vengono raccolti dall'analizzatore di interrogazioni ed esposti all'utente che ha sottoposto l'interrogazione.


User query
QUERY ANALYZER
Logical Object Registry
Pervasive Systems Architecture
APPLICATION OBJECT
LOGICAL OBJECT
LOGICAL OBJECT
LOGICAL OBJECT
OPERATION MANAGER
OPERATION MANAGER
OPERATION MANAGER
PHYSICAL DEVICE
PHYSICAL DEVICE
PHYSICAL DEVICE
PHYSICAL DEVICE

Caratteristiche del linguaggio PERLA

Analizziamo ora le principali caratteristiche del linguaggio PERLA. Come anticipato nella sezione riguardante la storia di PERLA, il linguaggio è costituito da due parti:

- una parte di basso livello che gestisce le operazioni di campionamento;
- una parte di alto livello che manipola i dati campionati per produrre il risultato dell'interrogazione.

Una delle caratteristiche del linguaggio è infatti la possibilità di stabilire i parametri di campionamento, in modo che sia l'utente stesso a creare le tabelle su cui, utilizzando la parte di alto livello, andrà a fare l'interrogazione. La parte di basso livello non consente solo all'utente di stabilire la frequenza di campionamento per ogni nodo, ma anche di stabilire in quali condizioni il campionamento deve essere effettuato. Un esempio immediato è la richiesta di attivazione del campionamento solo nei nodi nei quali la carica della batteria è sufficiente, cioè maggiore di un valore di soglia stabilito dall'utente.

Un'altra caratteristica è la possibilità di iniziare un campionamento su un nodo se e solo se un certo valore è stato campionato su un altro nodo. Si consideri per esempio il monitoraggio della temperatura in una determinata zona, con sensori di temperatura posizionati su piattaforme mobili. E' possibile richiedere l'attivazione dell'operazione di campionamento su un nodo se e solo se la piattaforma su cui è posizionato si trova in quella zona, permettendo un notevole risparmio di energia. Il nome di questa funzione è *Pilot Join*. È possibile distinguere due tipi di pilot join:

- Pilot join basata su eventi. Quando accade un evento (per esempio l'inserimento di un record in una tabella di stream) un certo insieme di nodi deve iniziare il campionamento. Per esempio si supponga, nell'esempio delle piattaforme mobili, che la temperatura debba essere misurata ogni volta che una piattaforma attraversa i limiti della zona: in questo caso servirà una pilot join basata sugli eventi.

- Pilot join basata su condizione. Un campionamento continuo deve essere effettuato su tutti i nodi connessi ad una base station tra quelle che soddisfano determinati criteri. La lista di base station che soddisfano la condizione è ottenuta estraendo una finestra di data lunghezza da uno stream, e tale lista viene aggiornata periodicamente, con una frequenza minore di quella di campionamento. Si consideri di nuovo l'esempio delle piattaforme mobili. Si supponga che una query in esecuzione stia monitorando le posizioni di tutte le piattaforme e le stia inserendo in uno stream. Si supponga inoltre che il comportamento desiderato per il sistema sia il campionamento continuo dei sensori di temperatura montati sulle piattaforme la cui ultima posizione monitorata sia all'interno della zona. In questo caso è richiesta una pilot join basata su condizione.

L'implementazione della pilot join basata su condizione richiede la presenza di una struttura dati che contenga la lista dei record usati come condizione per attivare e disattivare il campionamento dai nodi. Per questo motivo in PERLA esistono due tipi di strutture dati:

- Stream. Le tabelle di stream sono quelle più importanti. Si tratta di una lista di record prodotti da una o più interrogazioni. Possono inoltre essere usate per implementare la pilot join basata su evento: l'inserimento di ogni nuovo record nella tabella di stream è l'evento usato per avviare la query di basso livello nei nodi associati al record inserito.

- Snapshot. Sono tabelle di dimensione fissa ottenute, come indica il nome, considerando l'insieme dei record prodotti da una query in un periodo di tempo fissato. Durante ogni periodo tutti i record generati dalla query sono inseriti in un buffer, e alla fine del periodo la tabella di snapshot viene svuotata e riempita con i record presenti nel buffer. Questo tipo di tabella è stato previsto per implementare la pilot join basata su condizione: il contenuto della tabella di snapshot è la lista delle base station che al momento soddisfano la condizione, e

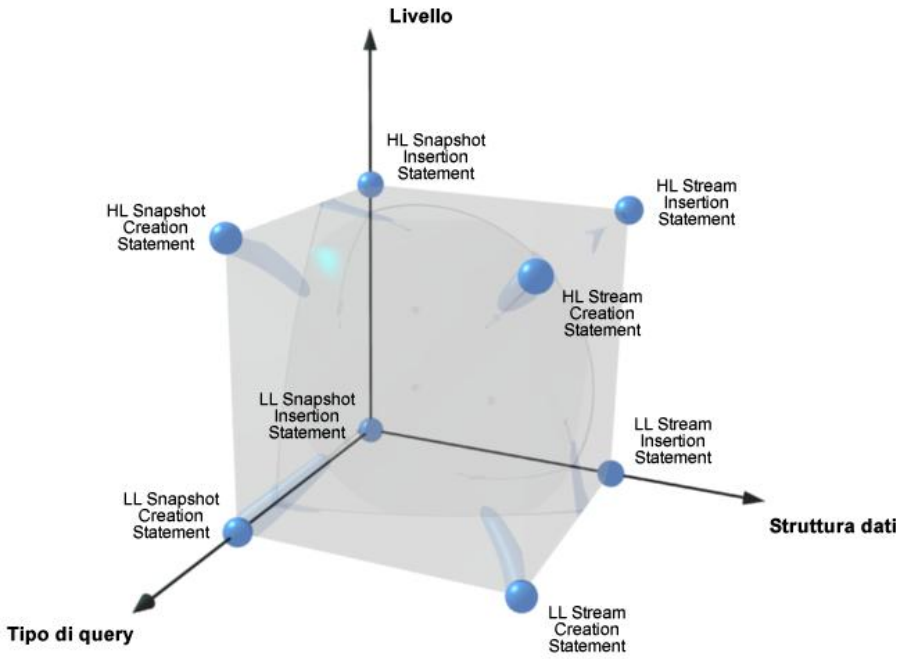
alle quali sono collegati i nodi sui quali nel prossimo periodo verrà attivato il campionamento.

Tutte le query, sia di basso che di alto livello, possono essere classificate in due gruppi:

- Query di creazione, per definire la struttura delle tabelle
- Query di inserimento, per definire come i dati devono andare a riempire le tabelle.

Poiché sono presenti due tipi di strutture dati nel sistema (stream e snapshot), sia le query di creazione che quelle di inserimento possono essere usate per creare e popolare i due tipi di tabella. È tuttavia importante notare le differenze tra query di inserimento che si riferiscono a stream e query di inserimento riferite a snapshot. Infatti quando si ha a che fare con stream, è necessario specificare la frequenza con cui la query deve essere elaborata e il suo risultato inserito nella struttura dati di destinazione. Nel caso di snapshot invece la frequenza è fissa e pari alla durata dello snapshot.

In base a quanto detto fin ad ora, se una query può essere di basso o di alto livello, di creazione o di inserimento, e relativa a stream o a snapshot, appare evidente che esisteranno 8 tipi di query diverse.



Struttura a oggetti per le query

Dopo aver brevemente presentato il linguaggio PERLA, in questo capitolo si fornirà una spiegazione del significato di ogni clausola, e per ognuna di esse si descriverà la classe Java implementata, corredata da un opportuno diagramma delle classi.

Una nota importante riguarda i nomi delle classi: ogniqualvolta una clausola di alto e di basso livello hanno lo stesso nome ma una diversa struttura sintattica, i nomi delle classi associate saranno preceduti dalle sigle *LL* e *HL* rispettivamente per il basso e per l'alto livello.

Per evitare di appesantire i diagrammi delle classi, si è deciso di utilizzare la seguente convenzione: ogniqualvolta esiste in una classe un attributo di tipo privato, corredato dai relativi metodi *getter* e *setter* di tipo pubblico, nei diagrammi delle classi verrà mostrato il solo attributo, come se fosse pubblico. In realtà in nessuna classe sono presenti attributi pubblici.

Classi di utilità generale

Prima di analizzare le clausole del linguaggio è opportuno descrivere le classi di utilità generale, di cui si farà ampio uso in tutte le classi associate al linguaggio.

Collezioni di elementi

Spesso sarà necessario gestire insiemi, ordinati e non, di elementi. I metodi più semplici per la gestione degli insiemi ordinati sono l'utilizzo di vettori, o l'uso di classi già presenti nelle librerie di Java, come ad esempio java.util.ArrayList. In particolare quest'ultima soluzione risulta assai utile e di facile utilizzo nei casi in cui si voglia manipolare un insieme ordinato di elementi, tuttavia non è improbabile che in un futuro più o meno prossimo si possa decidere di sostituirla con un'altra classe. Si è ritenuto quindi opportuno incapsulare questa struttura all'interno di un'altra classe, chiamata **GenericCollection**, che esponga metodi per l'accesso e la modifica dei dati contenuti nell'ArrayList. GenericCollection è una classe astratta e generica contenente al suo interno un ArrayList di tipo generico, quindi non è istanziabile e per utilizzarla è

necessario creare una classe figlia non astratta, specificando il tipo dei dati contenuti nell'ArrayList. I metodi esposti, oltre al costruttore, sono:

- public void add (ValueType val): consente di aggiungere un nuovo elemento alla fine della collezione
- public ValueType getValue (int index): restituisce l'elemento nella posizione indicata da index
- public boolean remove (int index): rimuove dalla collezione l'elemento nella posizione indicata da index

Esistono casi in cui si vorrebbe non solo accedere ad un elemento di una collezione mediante un indice, ma anche utilizzando una chiave. Per questi casi è stata prevista una seconda classe, di nome **GenericDictionary**, del tutto simile a GenericCollection, ma che incapsula sia un ArrayList che una HashMap, contenenti gli stessi valori. La classe espone metodi che consentono l'accesso ai dati sia utilizzando un indice, sia mediante l'utilizzo di una chiave. Per ogni tipo di funzionalità (aggiunta di un elemento, rimozione, ecc.) è stato quindi previsto un metodo che accetta come parametro un indice, e un metodo al quale è necessario fornire una chiave. Tutti i metodi che modificano una struttura modificano anche l'altra, in modo che i dati contenuti nelle due strutture siano congruenti. Anche la classe GenericDictionary è astratta, e per istanziarla è necessario creare una classe figlia, specificando sia il tipo dei dati che dovrà contenere, sia il tipo della chiave, che generalmente sarà una stringa.

Entrambe le classi implementano l'interfaccia java.util.Iterable ed espongono il metodo iterator(), che restituisce un oggetto di tipo Iterator, consentendo così di scorrere più agevolmente gli elementi che compongono l'insieme. Per motivi di coerenza il metodo *remove()* dell'Iterator non è supportato e genera un'eccezione.

D'ora in avanti lo standard per l'utilizzo delle classi GenericCollection e GenericDictionary sarà il seguente: ogniqualvolta una classe *xxx* deve essere collegata a più oggetti *yyy*, si creerà una nuova classe *yyyList*, ottenuta estendendo la classe GenericCollection o GenericDictionary e specificando *yyy* come tipo di dati. La classe *xxx*

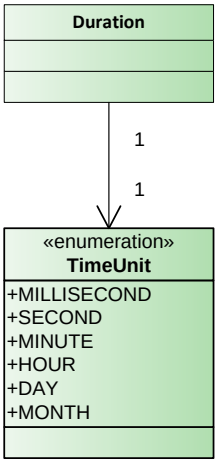
conterrà una variabile privata di tipo *yyyList*, inizializzata all'interno del costruttore della classe, e alla quale sarà possibile accedere utilizzando un metodo getter pubblico. Si consideri il seguente esempio: un oggetto di tipo Query deve essere collegato a più oggetti di tipo LLStatement. Si crea quindi un oggetto LLStatementList estendendo la classe GenericCollection e si pone in Query una variabile di tipo LLStatementList.

Durate temporali

Come abbiamo visto nei capitoli precedenti, una delle operazioni che riveste maggiore importanza nel linguaggio è il campionamento. È quindi necessario stabilire in che modo il tempo deve essere gestito.

Utilizzando il linguaggio PERLA è possibile sottoporre interrogazioni rappresentando il tempo con un numero seguito da una unità di misura temporale, che può variare dai millisecondi ai mesi. E' stata quindi necessaria la creazione di una enumerazione, di nome **TimeUnit**, contenente tutte le possibili unità di tempo utilizzabili dall'utente. Per quanto riguarda la durata, si è ritenuta opportuna la creazione di una classe apposita, di nome **Duration**. Al suo interno è presente un valore intero positivo, e una variabile di tipo TimeUnit.

```
<Time Unit> →
  { MILLISECONDS | SECONDS | MINUTES | HOURS | DAYS | MONTHS |
  'MS' | 'S' | 'M' | 'H' | 'D' | 'MT' }

<Duration> →
  <Float Constant> <Time Unit>
```



Espressioni e condizioni

Esistono diverse clausole del linguaggio che richiedono l'utilizzo di espressioni e condizioni. La trattazione di questo problema esula dagli scopi del presente lavoro, tuttavia è necessario fornirne una spiegazione sintetica senza la quale alcune scelte progettuali potrebbero non essere chiare. Nonostante espressioni e condizioni siano abbastanza diverse dal punto di vista dell'utente, dal punto di vista del linguaggio una condizione è semplicemente una espressione il cui valore è di tipo booleano.

Le espressioni vengono rappresentate utilizzando una struttura ad albero, in cui i nodi foglia contengono valori costanti, aggregati, funzioni, campi di un oggetto logico (nel contesto delle query di basso livello) o campi di una struttura dati (nel contesto delle query di alto livello). I nodi interni rappresentano invece le operazioni contenute nell'espressione (per esempio operazioni algebriche e logiche, funzioni, ecc.). Visitando con un opportuno algoritmo tutti i nodi dell'albero, è possibile effettuare le necessarie verifiche di tipo e calcolare il risultato dell'espressione, che sarà riportato nel nodo radice. Quindi ogniqualvolta una classe avrà come attributo una condizione o una espressione, si utilizzerà un oggetto di tipo **Node**, che rappresenta un nodo di tipo generico.

Strutture dati

Come introdotto nei precedenti capitoli, in PERLA esistono due tipi di strutture dati: gli Stream e gli Snapshot. Ogni tabella è identificata da un nome ed è composta da un elenco di campi. Inoltre è importante sapere se verrà utilizzata per fornire i dati in output, o solamente come ingresso per altre interrogazioni.

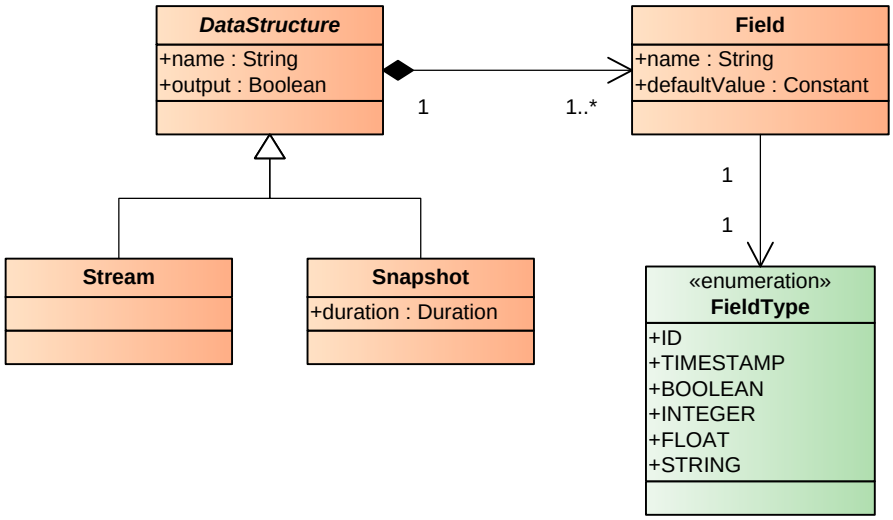
Ogni campo è identificato da un nome e dal tipo di dato che può contenere; è inoltre possibile specificare un valore di default da applicare nel caso in cui il sistema non disponga di un dato da inserire nel campo. I tipi di dato che un campo può contenere sono elencati nell'enumerazione **FieldType**.

```
<Create Stream Clause> →
  CREATE [OUTPUT] STREAM
  <Data Structure Name> '(' <Field Definition List> ')'
```

```
<Create Snapshot Clause> →
  CREATE [OUTPUT] SNAPSHOT
  <Data Structure Name > '(' <Field Definition List> ')'
  WITH DURATION <Duration>

<Field Definition List> →
  <Field Definition> { ',' <Field Definition> }*

<Field Definition> →
  <Data Structure Field> <Field Type> [DEFAULT <Signed Constant>]

<Field Type> →
  { ID | TIMESTAMP | BOOLEAN | INTEGER | FLOAT | STRING }
```

La classe **Field** rappresenta un campo, ed espone i metodi per leggere e modificare il nome, il tipo di dato e il valore di default. La classe astratta **DataStructure** espone, tra i vari metodi, il metodo *getFieldsDefinition()*, che restituisce una collezione di campi (ottenuta estendendo la classe GenericDictionary), ai quali è possibile accedere mediante indice o utilizzando come chiave di ricerca il nome del campo.

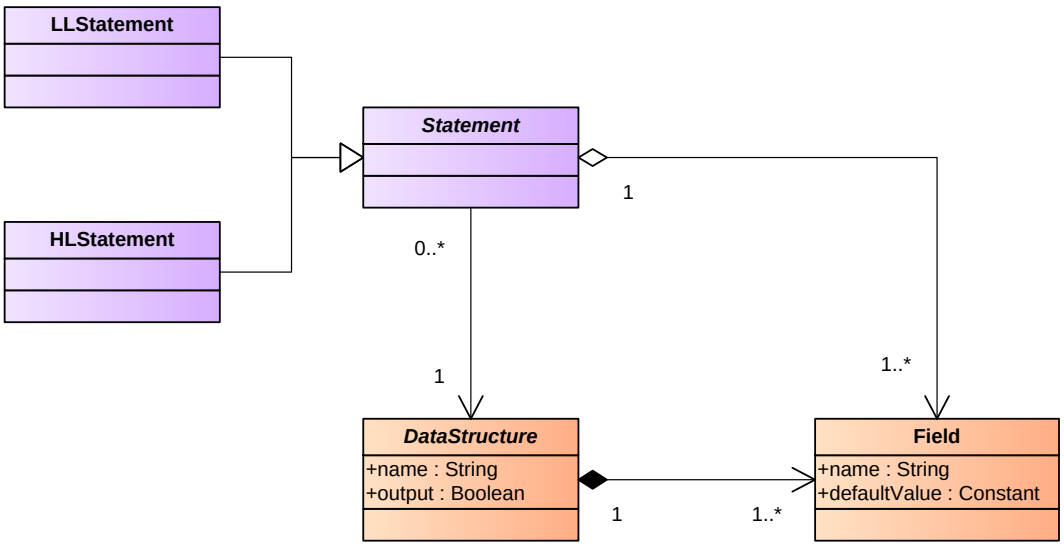
La classe DataStructure, che rappresenta una struttura dati di tipo generico, ha due classi figlie: **Stream** e **Snapshot**. Quest'ultima ha come attributo la durata della snapshot stessa, mentre la classe DataStructure ha come attributi il nome della tabella e un flag che indica se bisogna restituire la tabella come output per l'utente o come input per un'altra interrogazione.



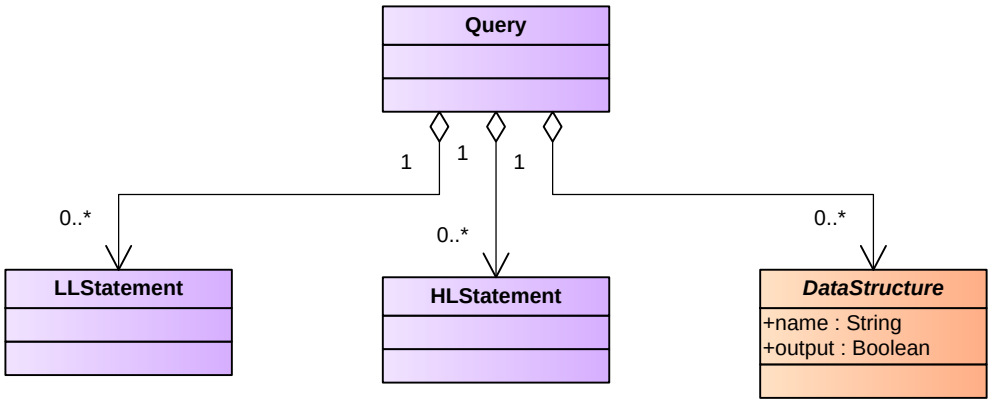
Statement

Tutte le asserzioni supportate dal linguaggio possono essere classificate in due gruppi: quelle di basso livello e quelle di alto livello. Come spiegato nei capitoli precedenti, la differenza è la fonte dei dati su cui lavorano. Infatti le interrogazioni di basso livello prendono i dati in ingresso direttamente dai nodi della rete, comunicando con loro tramite le interfacce fornite dagli oggetti logici. Le interrogazioni di alto livello invece elaborano i dati provenienti da altre interrogazioni, sia di basso che di alto livello.

Esisteranno quindi due tipi di statement, **LLStatement** e **HLStatement**, entrambi figli della classe astratta **Statement**. Quest'ultima è associata ad una DataStructure, che rappresenta la struttura di destinazione dei dati, ed espone il metodo *getFields()*, che restituisce una collezione di campi ottenuta derivando la classe GenericDictionary; questi sono i campi della struttura dati di destinazione in cui i dati devono essere inseriti. Se l'utente non specifica nessun campo della struttura di destinazione, i dati verranno inseriti in tutti i campi; in questo caso il numero e il tipo dei campi da inserire deve coincidere con quello della struttura di destinazione.



Ogni query sottoposta dall'utente è composta da un insieme di interrogazioni di alto e di basso livello, e dalla definizione di un certo numero di strutture dati. Di conseguenza la classe **Query** contiene una collezione di oggetti LLStatement, una di HLStatement e una terza collezione di oggetti DataStructure.


Query
1
1
1
0..*
0..*
0..*
LLStatement
HLStatement
DataStructure
+name : String
+output : Boolean

Statement di basso livello

Analizziamo ora gli statement di basso livello. Come già accennato, uno statement di basso livello ha il compito di recuperare i dati direttamente dai nodi della rete e, dopo averli analizzati e opportunamente filtrati, renderli disponibili per l'utente o per uno o più statement di alto livello.

Uno statement di basso livello è composto da diverse clausole, alcune delle quali opzionali, o comunque non necessarie nella maggior parte delle query di uso comune. Per mantenere un certo ordine nell'esposizione è utile analizzarle in base alla loro funzione. Uno statement di basso livello è infatti composto da tre parti principali:

- una sezione di campionamento, che specifica quando eseguire il campionamento dei sensori della rete e la memorizzazione dei dati in un buffer locale;
- una sezione di gestione dei dati, attivata periodicamente, che esegue una selezione sul buffer locale per produrre i risultati da inserire nella struttura dati di destinazione;
- una sezione contenente le condizioni di esecuzione, cioè le regole per stabilire se un certo oggetto logico dovrà prendere parte alla query

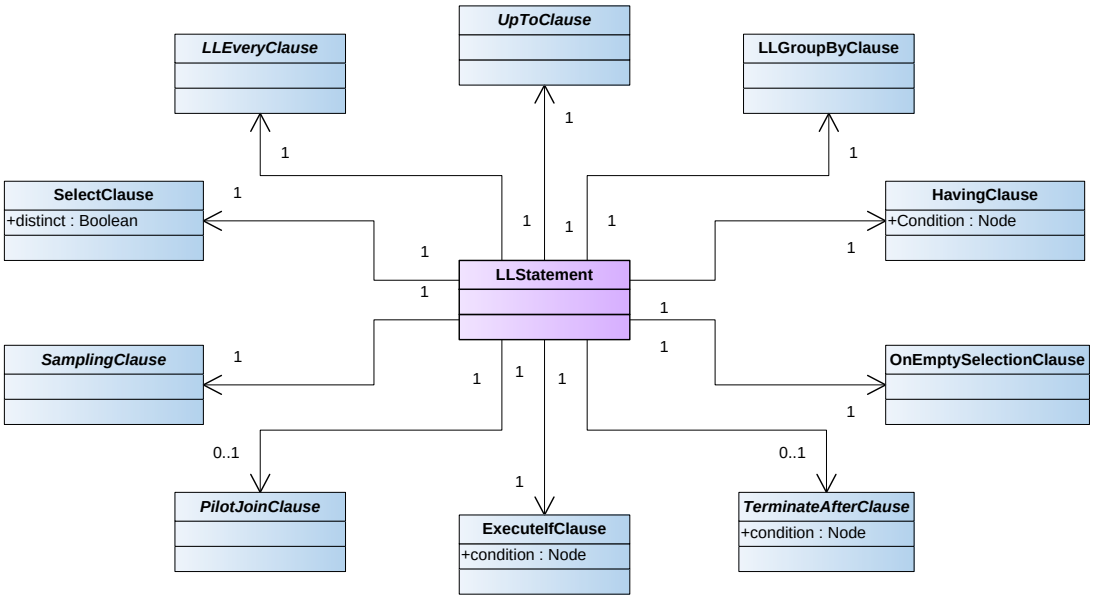
La sezione di campionamento comprende due clausole: **Sampling** e **Refresh**. La prima descrive le operazioni di campionamento dei dati direttamente dai nodi della rete, mentre la seconda permette di specificare se la frequenza di campionamento deve rimanere costante per tutta la durata della query o se deve essere periodicamente rivalutata.

Nella sezione di gestione dei dati sono presenti tutte le clausole per l'analisi dei dati campionati e per la generazione dei risultati da inserire nella tabella di destinazione. La più importante tra queste è la clausola **Select**, simile alla clausola Select dell'SQL standard, che definisce una proiezione dei record presenti nel buffer locale. La select è attivata periodicamente, con una frequenza definita dalla clausola **Every**. Esiste poi un insieme di clausole aggiuntive, che aggiungono maggiori funzionalità alla sezione di gestione dei dati. Esse sono la clausola **Up To**, che permette di recuperare più record alla

volta dal buffer locale, la clausola **Group By**, che permette di raggruppare i record nel buffer locale prima che la selezione venga eseguita, la clausola **Having**, che permette di filtrare gruppi di record, e la clausola **On Empty Selection**, che consente all'utente di specificare il comportamento della query quando la selezione non produce nessun record in uscita.

La sezione delle condizioni di esecuzione è costituita dalle clausole **Pilot Join**, il cui significato è stato spiegato nei precedenti capitoli, e **Execute If**, che può essere usata per imporre condizioni sull'esecuzione della query. Esiste inoltre la clausola **Terminate After**, che può essere usata per imporre un tempo limite oltre il quale l'oggetto logico abbandonerà l'esecuzione della query.

```
<LL Select Definition> →
  <Select Clause>
  [<Group By Clause>
  [<Up To Clause>]
  [<Having Clause>]
  [<On Empty Selection Clause>]
  <Sampling Clause>
  [<Pilot Join Clause>]
  [<Execute If Clause>]
  [<Terminate After Clause>]
```



Clausola Sampling

La clausola Sampling gestisce il campionamento dei dati dai nodi della rete. Una query di basso livello è eseguita su un singolo nodo della rete, e recupera i dati comunicando tramite l'interfaccia dell'oggetto logico che lo rappresenta. Il campionamento consiste nella lettura di uno o più attributi dell'oggetto logico e nell'inserimento del record così ottenuto in un buffer locale.

Esistono due tipi di campionamento: basato sugli eventi, in cui il campionamento avviene in corrispondenza di determinati eventi, e basato sul tempo, in cui invece è eseguito periodicamente con una certa frequenza. La classe **SamplingClause** è quindi astratta, e da essa derivano le classi **SamplingEventBased** e **SamplingTimeBased**. Nella classe SamplingEventBased è presente il metodo *getEvents()*, che restituisce una collezione di **Event**, ognuno dei quali rappresenta un evento, caratterizzato da un nome.

La classe SamplingTimeBased è più complessa in quanto supporta una definizione del periodo di campionamento basata sul costrutto *if-elseif-else*: l'utente deve poter indicare una serie di condizioni, e per ognuna di queste, il periodo di campionamento da adottare. Deve inoltre specificare un periodo di campionamento di default da utilizzare nel caso in cui nessuna delle condizioni indicate sia soddisfatta. La classe espone quindi un metodo per accedere ad una collezione ordinata di oggetti **SamplingIfEvery**, ognuno dei quali presenta al suo interno una condizione, un periodo di campionamento e una unità di tempo. L'ultimo elemento di questa collezione ha nella condizione il valore *true*, e rappresenta il periodo di campionamento di default. È prevista la possibilità di inserire una espressione come periodo di campionamento: per questo motivo la variabile in questione non è di tipo intero, ma di tipo Node.

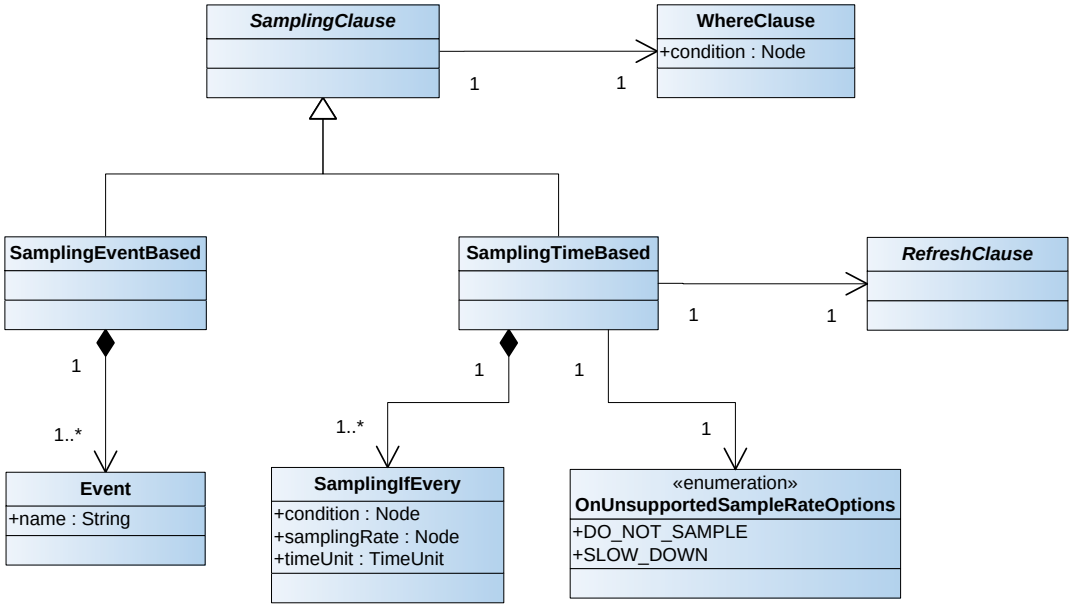
Nel campionamento basato sul tempo è possibile utilizzare due clausole aggiuntive. La prima permette di definire il comportamento da adottare nel caso in cui la frequenza di campionamento richiesta sia troppo alta e quindi non supportata. Le opzioni disponibili in questo caso (ossia diminuire la velocità di campionamento oppure non campionare, che è quella predefinita) sono contenute nell'enumerazione **OnUnsupportedSampleRateOptions**. La seconda clausola permette all'utente di specificare se la frequenza di campionamento è fissa o se deve essere rivalutata

periodicamente. All'interno della classe è quindi presente una variabile di tipo **RefreshClause**, il cui utilizzo verrà chiarito nel prossimo paragrafo.

La clausola di campionamento è completata da una clausola Where, che permette di definire una condizione di filtraggio. Se specificata, questa condizione viene rivalutata ogniqualvolta un record viene prodotto e, se soddisfatta, il record viene inserito nel buffer locale, altrimenti viene scartato. Per questo motivo esiste la classe **WhereClause**, che contiene al suo interno la condizione di filtraggio. Se l'utente non include la clausola Where all'interno della sua query, l'oggetto WhereClause viene comunque creato, e l'attributo relativo alla condizione di filtraggio viene impostato alla condizione *true* (di fatto non viene applicato alcun filtraggio).

```
<Sampling Clause> →
  SAMPLING
  {
      <On Event Clause> |
      <Sampling IfEvery Clause>
          [<On Unsupported SR Clause>][<Refresh Clause>]
  }
  [<Where Clause>]

<On Event Clause> →
  ON EVENT <Event List>

<Event List> →
  <Logical Object Event> { ',' <Logical Object Event> }*

<Sampling IfEvery Clause> →
  {
      { <Sampling If Clause> <Sampling Every Clause> }*
          ELSE <Sampling Every Clause> |
      <Sampling Every Clause>
  }

<Sampling If Clause> →
  IF <Condition>
```

```
<Sampling Every Clause> →
  EVERY <Expression> <Time Unit>

<On Unsupported SR Clause> →
  ON UNSUPPORTED SAMPLE RATE { DO NOT SAMPLE | SLOW DOWN }

<Where Clause> →
  WHERE <Condition>
```



Clausola Refresh

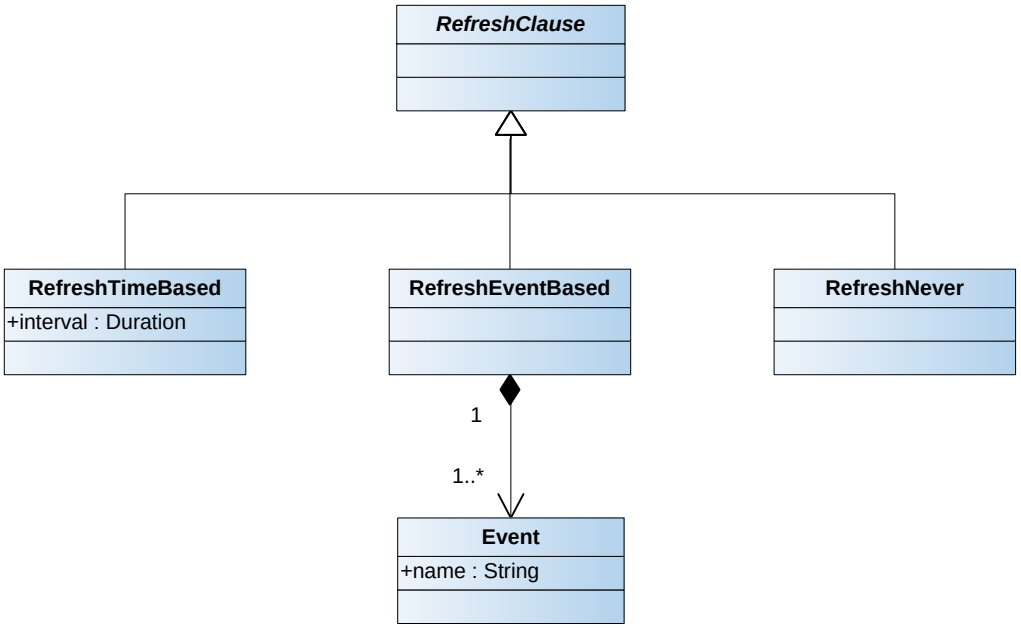
Nel precedente paragrafo è stata anticipata la presenza di una clausola la cui funzione è quella di indicare ogni quanto tempo la frequenza di campionamento debba essere ricalcolata. L'utente ha la possibilità di scegliere tra una delle seguenti opzioni:

- refresh basato sul tempo, ossia periodicamente con una certa frequenza;
- refresh in corrispondenza di determinati eventi;
- nessun refresh.

```
<Refresh Clause> →
  REFRESH { <On Event Clause> | EVERY <Duration> | NEVER }
```

Esiste quindi una classe **RefreshClause** di tipo astratto, dalla quale discendono tre classi, ognuna delle quali è associata ad una delle opzioni elencate:

- una classe **RefreshTimeBased**, con al suo interno una variabile di tipo Duration che rappresenta la frequenza di aggiornamento;
- una classe **RefreshEventBased**, che espone il metodo getEvents(). Questo restituisce una collezione di **Event**, ognuno dei quali rappresenta un evento ed è accessibile tramite un indice o una chiave, che probabilmente sarà il nome dell'evento;
- una classe **RefreshNever**, che è la classe predefinita nel caso in cui l'utente non specifichi la clausola Refresh.



Clausola Select

La clausola Select è probabilmente la più importante dell'intera interrogazione. Essa fa parte della sezione della query che si occupa della gestione dei dati. Mentre la sezione di campionamento ha la funzione di leggere gli attributi degli oggetti logici ed

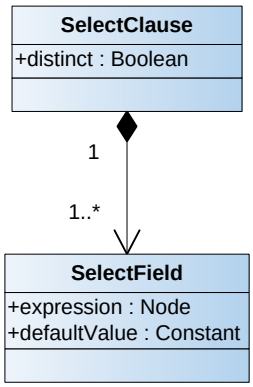
inserirli nel buffer locale, la sezione di gestione dei dati si occupa di stabilire quali tra questi dati devono far parte dell'output, quindi effettua una selezione e proiezione dei record del buffer locale. In particolare la clausola Select si occupa della proiezione: come per la clausola Select dell'SQL standard, è necessario specificare una serie di campi, che dal punto di vista grammaticale corrispondono ad espressioni che possono contenere nomi di attributi dell'oggetto logico, operatori standard e di aggregazione, o valori costanti. Nel caso in cui l'utente non specifichi le clausole Up To e Group By, la clausola Select restituisce un solo record.

L'utente ha la possibilità di specificare le parole chiave *DISTINCT* e *ALL*, che hanno la stessa funzione dell'SQL standard. Inoltre può specificare per ogni campo della Select un valore di default, la cui funzione verrà chiarita quando verrà introdotta la clausola On Empty Selection.

```
<Select Clause> →
  SELECT [DISTINCT | ALL] <Field Selection List>

<Field Selection List> →
  <Field Selection> { ',' <Field Selection> }*

<Field Selection> →
  { <Expression> [DEFAULT <Signed Constant>] }
```

La classe **SelectClause** ha come attributo il flag *distinct*, che vale *true* se l'utente ha specificato nella query la parola chiave *DISTINCT*, e *false* se ha specificato la parola *ALL* o non ha specificato alcuna parola chiave. Essa inoltre espone il metodo per l'accesso ad una lista di oggetti **SelectField**, ognuno dotato di un attributo di tipo Node contenente l'espressione su cui fare la proiezione, e di un valore di default di tipo Constant, che rappresenta una costante di tipo generico.

La clausola Select è valutata periodicamente con una frequenza definita all'interno della clausola Every della query di inserimento.

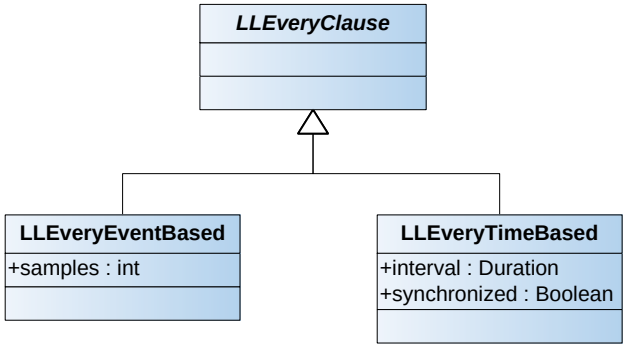


Clausola Every

Nei capitoli precedenti è stata spiegata la differenza tra statement di creazione di una tabella e statement di inserimento dei dati in una tabella creata. In quest'ultimo caso bisogna specificare il nome della tabella di destinazione e il sottoinsieme dei campi per i quali la selezione fornirà dei valori. Inoltre, se la struttura di destinazione è una tabella di stream, sarà necessario fornire altri parametri, ossia come e quando la selezione dovrà essere eseguita e il suo risultato aggiunto nella tabella di destinazione. Questi parametri sono contenuti nella clausola Every, della quale esistono due versioni: una basata sul tempo e l'altra basata sugli eventi. Nella prima è possibile specificare una durata temporale, e la selezione è attivata ad intervalli regolari di durata pari a quella specificata; nella versione ad eventi invece è necessario indicare un numero di campioni, e la selezione è attivata ogni volta che nel buffer locale viene inserito un numero di nuovi record pari a quanto indicato nella clausola stessa.

```
<LL Every Clause> →
  EVERY { <Duration> [SYNCHRONIZED] | <Integer Constant> SAMPLES
| ONE }
```

La classe **LLEveryClause** è quindi di tipo astratto, e da essa discendono le classi **LLEveryEventBased** e **LLEveryTimeBased**. Quest'ultima contiene, oltre all'attributo precedentemente descritto, un flag di nome *synchronized* che, se vale *true*, indica che il sistema forzerà la prima esecuzione della selezione ad un tempo multiplo della durata specificata.



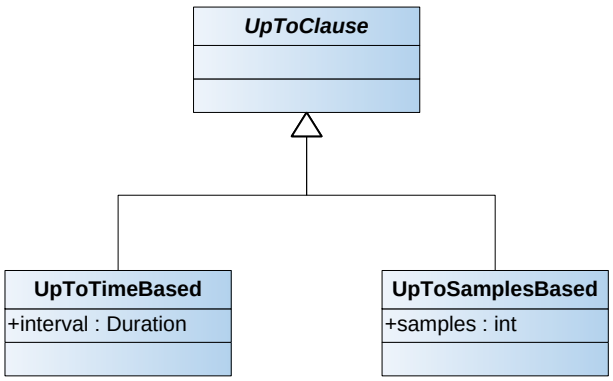
Clausola Up To

La clausola Up To permette di recuperare più di un record dal buffer locale, dando alla clausola Select una semantica un po' più simile a quella dell'SQL standard. È possibile specificare una durata o un numero di campioni per identificare la porzione di buffer da restituire. Per esempio, se si specifica una durata di 1 minuto, la selezione restituirà tutti i record del buffer locale relativi all'ultimo minuto.

```
<Up To Clause> →
  UP TO { <Duration> | <Integer Constant> SAMPLES | ONE}
```

Esiste quindi una classe astratta di nome **UpToClause**, estesa dalle classi **UpToTimeBased** e **UpToSamplesBased**. La prima contiene una variabile di tipo Duration con valore pari alla dimensione della finestra, mentre la seconda ha un attibuto intero positivo che rappresenta il numero di campioni che devono essere recuperati.

Se nella query fornita dall'utente la clausola Up To non è specificata, verrà istanziato un oggetto UpToSamplesBased con numero di campioni uguale a 1.



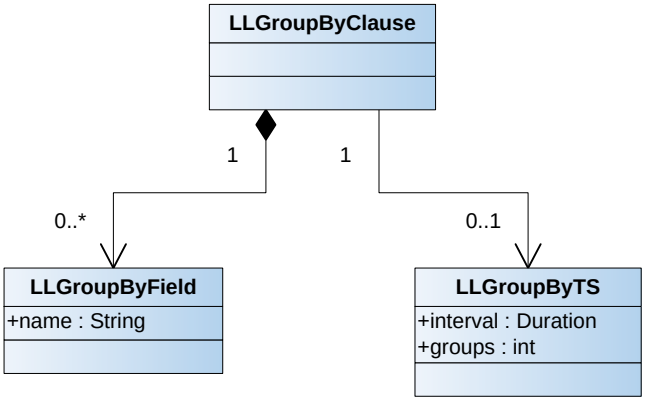
Clausola Group By

La clausola Group By consente di raggruppare i record nel buffer locale prima che il processo di selezione venga eseguito. Il linguaggio supporta due tipi di raggruppamento: per attributo e per timestamp. Nel primo caso il buffer viene suddiviso in più buffer, ognuno dei quali si riferisce ad un singolo valore dell'attributo. Nel secondo caso invece è necessario specificare una durata *d* e un numero di gruppi *n*. Il buffer locale viene quindi replicato in *n* buffer identici, e i record relativi ai primi *d*i* secondi vengono rimossi dalla i-esima copia del buffer. In seguito la selezione viene effettuata su ognuno dei buffer ottenuti.

Esistono quindi una classe **LLGroupByField** per il raggruppamento per attributo, del quale sarà necessario specificare il nome, e **LLGroupByTs** per il raggruppamento basato sul timestamp, in cui bisognerà indicare una durata e un numero di periodi.

È possibile effettuare nella stessa query un raggruppamento su uno o più attributi, oppure solo per timestamp, o ancora simultaneamente per timestamp e per uno o più attributi. Per questo motivo la classe che rappresenta la clausola, **LLGroupByClause**, avrà un metodo per l'accesso ad una collezione di oggetti di tipo LLGroupByField, che rappresentano l'insieme degli attributi sui quali si vuole raggruppare, e una variabile collegata ad un oggetto di tipo LLGroupByTS, che viene opzionalmente istanziato.

```
<Group By Clause> →
  GROUP BY <Field Grouping By List>

<Field Grouping By List> →
  { <Field Grouping By> | <Field Grouping By TS> }
  { ‘,’ <Field Grouping By> }*

<Field Grouping By> →
  <Logical Object Field>

<Field Grouping By TS> →
  TIMESTAMP ‘(’ <Duration> ‘,’ <Integer Constant> GROUPS ‘)’
```

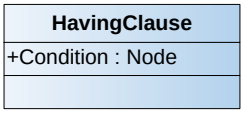


Clausola Having

Come è stato spiegato nei precedenti paragrafi, tramite la clausola Up To è possibile recuperare più di un record dal buffer locale. L'utente ha la possibilità di fornire una condizione per filtrare i record così ottenuti utilizzando la clausola Having.

```
<Having Clause> →
  HAVING <Condition>
```

La classe associata alla clausola è **HavingClause**, il cui unico attributo è la condizione di filtraggio, di tipo Node. Nel caso in cui l'utente non specifichi la clausola Having, l'oggetto HavingClause viene comunque creato, ma contiene il valore *true* nella condizione.



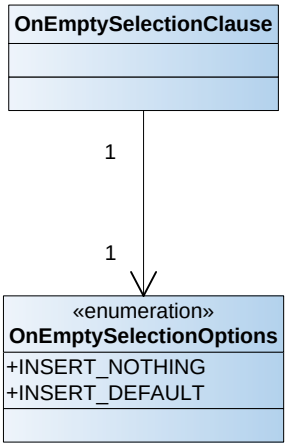
Clausola On Empty Selection

La clausola On Empty Selection permette all'utente di specificare il comportamento della query nel caso in cui il processo di selezione non produca alcun record (per esempio se la clausola Having filtra tutti i record). Le opzioni possibili sono due: non inserire nulla oppure inserire un record di default, composto dai valori di default specificati nella clausola Select (se non sono stati specificati, si prendono i valori

di default degli attributi). Se la clausola non viene esplicitamente specificata nella query, il comportamento predefinito è non inserire nulla.

```
<On Empty Selection Clause> →
  ON EMPTY SELECTION { INSERT NOTHING | INSERT DEFAULT }
```

Le opzioni sopra presentate sono elencate nell'enumerazione **OnEmptySelectionOptions** con i nomi *INSERT_NOTHING* e *INSERT_DEFAULT*. La classe **OnEmptySelectionClause** possiede un attributo che assume uno di questi valori.



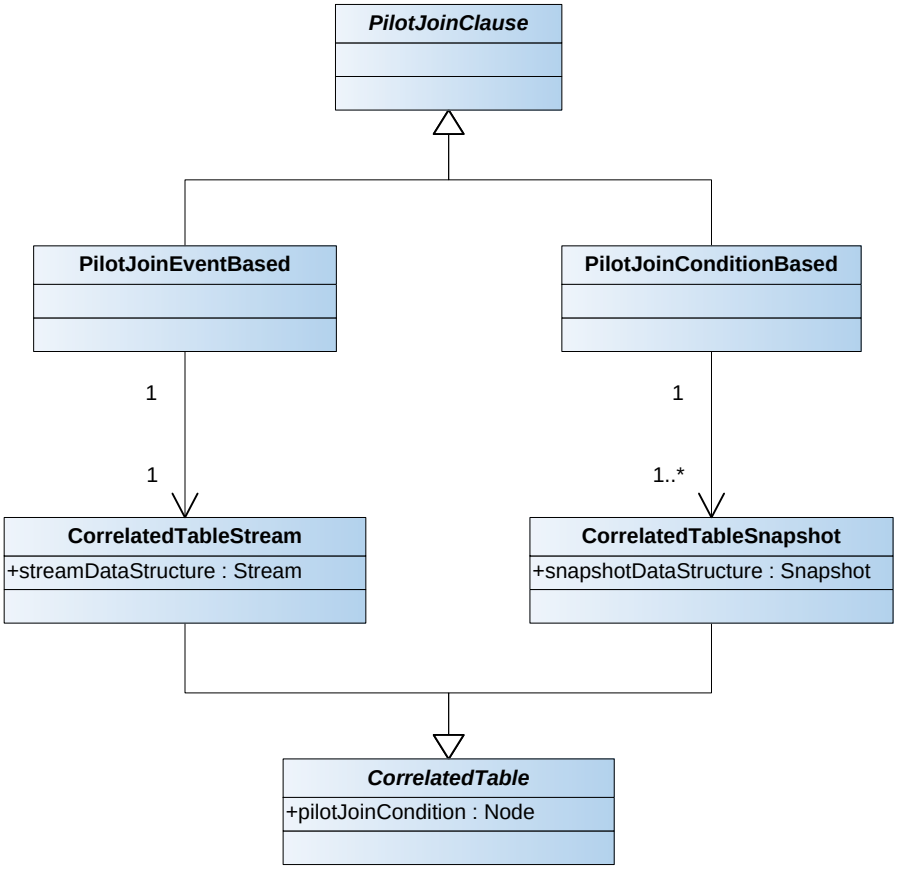
Clausola Pilot Join

La clausola Pilot Join è una delle clausole necessarie per definire l'insieme degli oggetti logici che saranno coinvolti nell'esecuzione di una query. Essa consente una esecuzione di una query su un nodo solo se una determinata condizione è verificata. La condizione è basata sul contenuto di una struttura dati esistente. Se la struttura dati è una tabella di stream, la pilot join sarà basata su eventi, e quando un nuovo record viene aggiunto alla tabella in questione, gli oggetti logici corrispondenti a quel record iniziano ad eseguire la query. Se invece la struttura dati è una tabella di snapshot, l'insieme degli oggetti logici che eseguono la query viene rivalutato periodicamente con un periodo pari alla durata dello snapshot.

L'operazione di pilot join può essere eseguita con più strutture dati simultaneamente, ma con alcune limitazioni: una pilot join basata su condizione può riferirsi ad un numero arbitrario di tabelle di snapshot, mentre una pilot join basata su evento può essere definita in relazione ad una sola tabella di stream; non è possibile definire una pilot join "ibrida", cioè con una tabella di stream e una o più tabelle di snapshot.

```
<Pilot Join Clause> →
  PILOT JOIN <Correlated Table List>

<Correlated Table List> →
  <Correlated Table> { ',' <Correlated Table> }*

<Correlated Table> →
  <Data Structure Name> ON <Condition>
```

È evidente che la classe **PilotJoinClause** è astratta, e ha due classi figlie: **PilotJoinEventBased** e **PilotJoinConditionBased**. La classe **CorrelatedTable** rappresenta invece la tabella coinvolta in una pilot join; essa è di tipo astratto e ha come attributo la condizione su cui fare la join, di tipo Node. Da CorrelatedTable derivano le classi **CorrelatedTableStream**, che ha come attributo lo Stream a cui si riferisce, e **CorrelatedTableSnapshot**, che invece ha come attributo uno Snapshot. La classe PilotJoinEventBased è quindi collegata ad una sola CorrelatedTableStream, mentre un oggetto PilotJoinConditionBased può essere collegato a più CorrelatedTableSnapshot. Per questo motivo la classe espone un metodo per accedere ad una collezione di CorrelatedTableSnapshot.



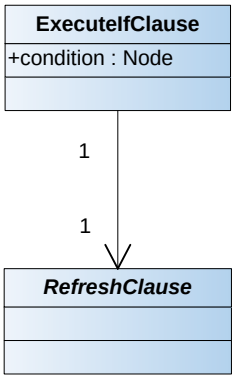
Clausola Execute If

La seconda clausola che può essere utilizzata per imporre condizioni sull'esecuzione di una query è Execute If. Essa permette di definire l'insieme degli oggetti logici che parteciperanno alla query ed è essenzialmente una condizione che viene valutata prima che la query sia distribuita ai nodi della rete.

È possibile aggiungere una clausola Refresh alla clausola Execute If per indicare se e quando il sistema dovrà rivalutare la condizione per aggiornare la lista degli oggetti logici che eseguono la query. La clausola Refresh, che è dello stesso tipo di quella utilizzata nella clausola Sampling, può essere omessa. In questo caso la condizione viene valutata solo nel momento in cui l'utente sottopone la query e, se soddisfatta, l'oggetto logico farà parte della query per tutta la sua durata, altrimenti verrà escluso per tutta la sua durata.

```
<Execute If Clause> →
  EXECUTE IF <Condition> [<Refresh Clause>]
```

La classe **ExecuteIfClause** ha solo l'attributo riguardante la condizione di esecuzione. Ad essa è collegato l'oggetto **RefreshClause**, analizzato in uno dei precedenti paragrafi. Se l'utente omette la clausola Execute If, l'oggetto viene comunque creato, ma la condizione assume il valore booleano true. Se invece omette la clausola Refresh, viene creato un oggetto RefreshNever.



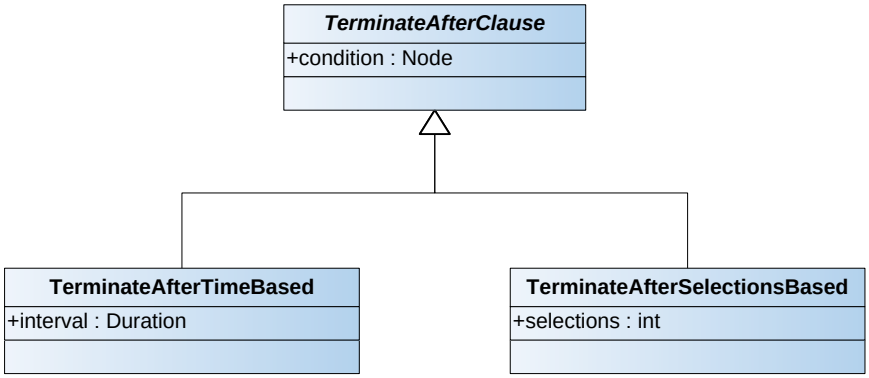
Clausola Terminate After

L'ultima clausola di uno statement di basso livello a disposizione dell'utente è Terminate After, che può essere usata per impostare un timeout dopo il quale l'oggetto logico abbandonerà l'esecuzione della query. Il timeout può essere specificato sia in termini di tempo, sia come numero di attivazioni del processo di selezione.

```
<Terminate After Clause> →
  TERMINATE AFTER { <Duration> | <Integer Constant> SELECTIONS }
```

La classe **TerminateAfterClause** è astratta e ha due classi figlie: **TerminateAfterTimeBased**, che ha come attributo la durata temporale del timeout, e **TerminateAfterSelectionsBased**, in cui il timeout è rappresentato da un numero intero positivo di attivazioni della sezione di gestione dati dello statement.

La clausola TerminateAfter è opzionale, e se non viene specificata ogni oggetto logico coinvolto nella query continuerà ad eseguirla, finché l'utente richiederà l'eliminazione dell'intera query sottoposta. In questo caso l'oggetto TerminateAfterClause non verrà creato.



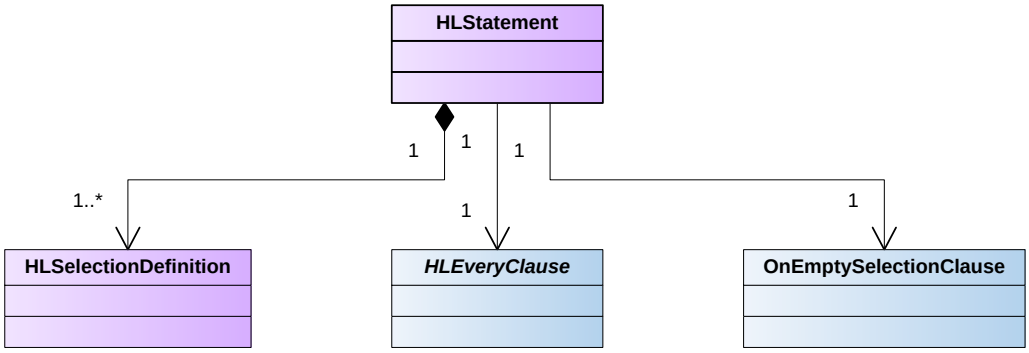
Statement di alto livello

Analizziamo ora gli statement di alto livello. Mentre gli statement di basso livello si occupano del campionamento dei dati direttamente dai sensori per creare le tabelle di stream e snapshot, quelli di alto livello recuperano i dati dalle tabelle di stream create dagli statement di basso livello, senza preoccuparsi del campionamento. Dal punto di vista sintattico e semantico sono quindi simili all'SQL standard. Uno statement di alto livello è composto dalle seguenti parti:

- un insieme di selezioni, separate dalla parola chiave *UNION*: come nell'SQL è infatti possibile unire i dati provenienti da diversi processi di selezione. Ovviamente tutte le selezioni devono generare tabelle con lo stesso numero e tipo di campi;
- una clausola Every che stabilisca ogni quanto tempo deve essere attivato il processo di selezione;
- una clausola On Empty Selection che stabilisca come comportarsi nel caso in cui nessun record sia stato prodotto dalla selezione.

```
<HL Select Definition> →
  <HL Single Select Definition>
  { UNION [ALL] <HL Single Select Definition> }*
  [<On Empty Selection Clause>]
```

La classe **HLStatement** espone il metodo *getUnionClause()*, che consente l'accesso alla collezione di oggetti **HLSelectionDefinition**, ognuno dei quali rappresenta una selezione. Ha inoltre come attributi un oggetto **HLEveryClause** e un oggetto **OnEmptySelection**.



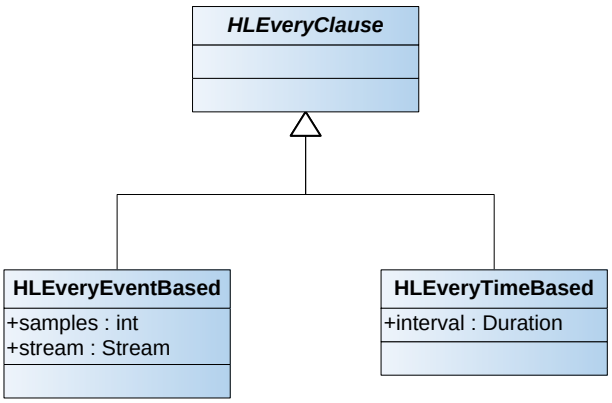
Clausola Every

La semantica della clausola Every negli statement di alto livello è simile a quella degli statement di basso livello. Come per il basso livello esistono due versioni: una basata sul tempo e l'altra basata sugli eventi. In quella basata sul tempo è possibile specificare una durata temporale, e la selezione è attivata ad intervalli regolari di durata pari a quella specificata; nella versione ad eventi invece è necessario indicare non solo un numero di campioni, ma anche il nome di una tabella. Gli statement di alto livello infatti possono lavorare si più di una tabella di input, a differenza degli statement di basso livello che possono lavorare sui dati provenienti da un solo oggetto logico. L'utente deve quindi specificare il nome della struttura dati da usare come fonte di eventi. In pratica quando nella tabella specificata viene inserito il numero di campioni indicato, il processo di selezione viene attivato. A differenza della clausola Every di basso livello non è possibile specificare la parola chiave *SYNCHRONIZED*, in quanto tutte le query di alto livello partono insieme all'avvio della query.

```
<HL Every Clause> →
  EVERY { <Duration> | <Integer Constant> SAMPLES IN
  <Data Structure Name> | ONE IN <Data Structure Name> }
```

La struttura ad oggetti è simile a quella utilizzata nella clausola Every di basso livello: la classe **HLEveryClause** è di tipo astratto, e da essa discendono le classi **HLEveryEventBased**, che ha come attributi il nome della struttura dati e il numero di campioni che devono essere inseriti al suo interno per attivare la selezione, e

HLEveryTimeBased, il cui unico attributo è l'intervallo di tempo tra una attivazione della selezione e la successiva.

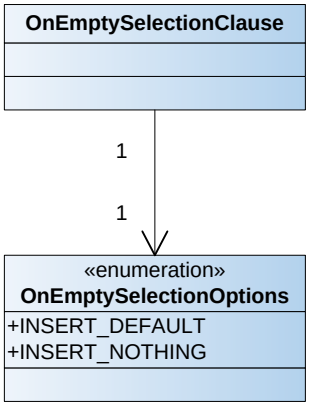


Clausola On Empty Selection

Anche negli statement di alto livello è possibile specificare la clausola On Empty Selection, che assicura che, in caso di necessità, almeno un record venga prodotto ogni volta che il processo di selezione viene attivato.

```
<On Empty Selection Clause> →
  ON EMPTY SELECTION { INSERT NOTHING | INSERT DEFAULT }
```

Per l'implementazione di questa clausola vengono utilizzate le stesse classi usate per la clausola On Empty Selection di basso livello.

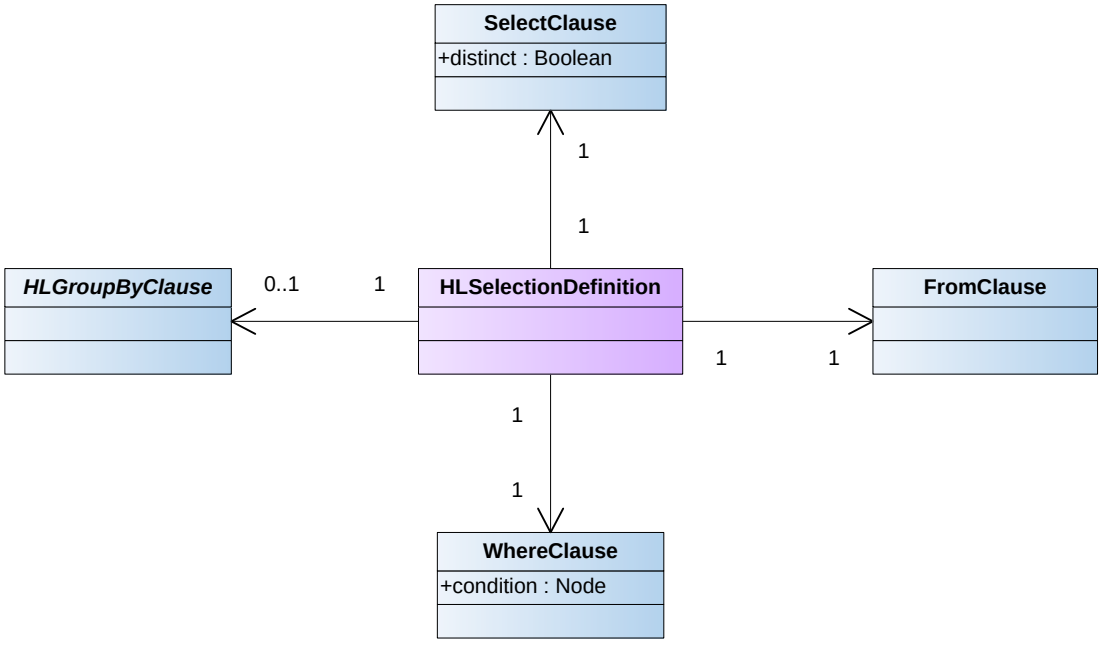


Selezione di alto livello

La selezione di alto livello è molto simile a quella dell'SQL standard. L'utente deve infatti specificare la parola chiave *SELECT* seguita da una lista di campi separati da una virgola e la parola chiave *FROM* seguita dalle tabelle da usare come fonti dei dati. Può inoltre specificare una clausola *Where* per filtrare il risultato e una clausola *Group By* per raggruppare i record.

```
<HL Single Select Definition> →
  <Select Clause>
  <From Clause>
  [<Where Clause>]
  [<Group Clause> [<Having Clause>]]
```

L'oggetto **HLSelectionDefinition** è quindi collegato agli oggetti **SelectClause**, **FromClause**, **WhereClause** e **HLGroupByClause**.



Clausola Select

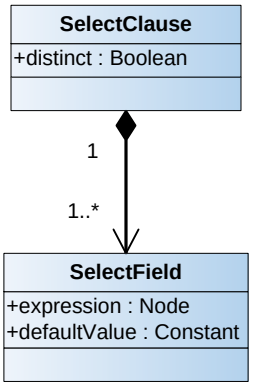
All'interno della singola selezione la clausola Select è, insieme alla clausola From, l'unica che l'utente deve fornire obbligatoriamente. Come nell'SQL standard, è necessario specificare una serie di campi, che dal punto di vista grammaticale

corrispondono ad espressioni che possono contenere nomi di campi delle tabelle da cui si esegue la selezione, operatori standard e di aggregazione, o valori costanti.

L'utente ha la possibilità di specificare le parole chiave *DISTINCT* e *ALL*, che hanno la stessa funzione dell'SQL standard. Inoltre può specificare per ogni campo della Select un valore di default, eventualmente utilizzato dalla clausola On Empty Selection.

```
<Select Clause> →
  SELECT [DISTINCT | ALL] <Field Selection List>

<Field Selection List> →
  <Field Selection> { ',' <Field Selection> }*

<Field Selection> →
  { <Expression> [DEFAULT <Signed Constant>] }
```

Poiché sintatticamente e semanticamente uguale alla clausola Select di basso livello, si è scelto di riutilizzare la stessa struttura ad oggetti.



Clausola From

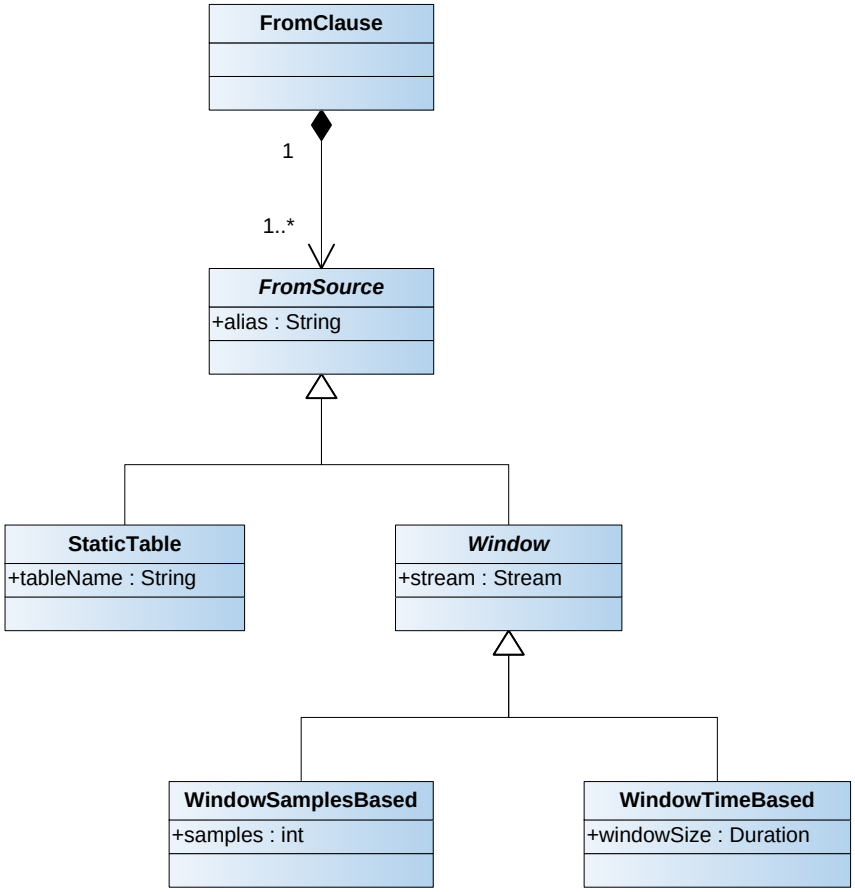
La struttura della clausola From è la stessa utilizzata nell'SQL standard: la parola chiave *FROM* è seguita da una lista di elementi separati da virgole. Ognuno di questi elementi non è però il nome della tabella di un database, ma una finestra di una tabella di stream. È possibile definire la dimensione di questa finestra sia indicandone la durata temporale, sia specificando un numero di record. L'utente può inoltre stabilire un alias

per ogni finestra, cioè un nome da poter utilizzare altrove nella query per riferirsi a quella determinata finestra. La definizione di un alias è obbligatorio solo quando la stessa tabella di stream è usata più volte nella stessa clausola From.

Oltre a finestre di stream, è possibile anche specificare come fonte dei dati tabelle statiche, il cui contenuto è fisso e noto a tutti i sensori della rete. Queste tabelle sono utili per memorizzare parametri stabiliti durante la fase di realizzazione della rete, per esempio per mantenere le connessioni fisiche tra i dispositivi.

```
<From Clause> →
  FROM <Window Definition List>

<Window Definition List> →
  <Window Definition> { ',' <Window Definition> }*

<Window Definition> →
  <Data Structure Name>
  ['(' <Duration> | <Integer Constant> SAMPLES | ONE ')']
  [AS <Data Structure Name>]
```

La classe **FromClause** espone il metodo *getFromSources()* per l'accesso ad una collezione di oggetti **FromSource**, ognuno dei quali ha come attributo una stringa per memorizzarne l'alias. Se l'utente non ha stabilito un alias nella query, la stringa è inizializzata con il nome della struttura dati. È possibile accedere ai singoli oggetti della lista utilizzando un indice, oppure l'alias. La classe FromSource è astratta, e da essa derivano le classi **StaticTable**, che ha come attributo il nome della tabella statica, e la classe astratta **Window**, il cui unico attributo è la tabella di stream a cui la finestra viene applicata. Le classi **WindowSamplesBased** e **WindowTimeBased** estendono la classe Window, e rappresentano rispettivamente una finestra di dimensione pari al numero di record indicato, e una finestra la cui lunghezza è data da un periodo di tempo.

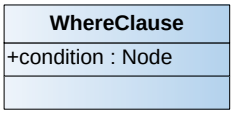


Clausola Where

Dopo aver recuperato i dati tramite le clausole Select e From, la clausola Where permette di filtrarli applicando una condizione.

```
<Where Clause> →
  WHERE <Condition>
```

La classe **WhereClause** ha come unico attributo una condizione, di tipo Node, che può essere una qualunque espressione. Se l'utente non specifica la clausola Where, l'oggetto viene comunque creato, e la condizione assume il valore *true*.

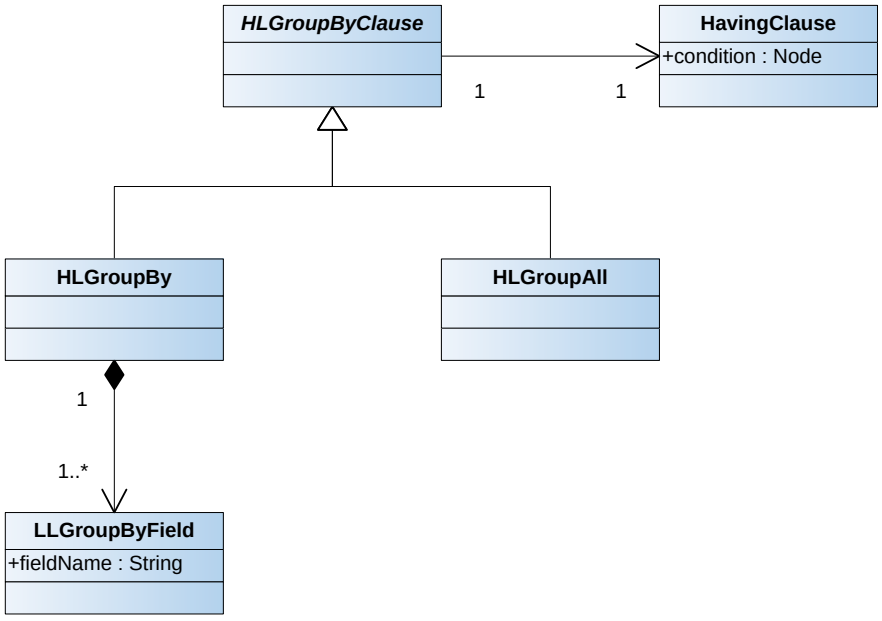


Clausola Group By

Mediante l'opzione Group By l'utente ha la possibilità di raggruppare i record. La parola chiave *GROUP BY* può essere seguita dalla parola *ALL*, o da una lista di nomi di campi separati da virgole. Nel primo caso tutti i record recuperati dalla fonte dei dati vengono raggruppati e il risultato è composto da un solo record, mentre nel secondo caso la semantica della query è uguale a quella dell'SQL standard. Dopo aver raggruppato i record, è possibile filtrare il risultato ottenuto specificando la clausola Having.

```
<Group Clause> →
  GROUP { BY <Window Field List> | ALL }

<Window Field List> →
  <Window Field> { ',' <Window Field> }*
```

La classe **HLGroupByClause** è astratta, e da essa discendono le classi **HLGroupAll**, e **HLGroupBy**. Quest'ultima espone il metodo *getFields()*, che consente l'accesso ad una lista di oggetti **HLGroupByField**, ognuno dei quali ha come attributo il nome del campo. Ad ogni oggetto HLGroupByField è possibile accedere utilizzando un indice o una chiave, corrispondente al nome di ogni campo. La classe HLGroupByClause è completata da un oggetto **HavingClause**, per il quale è stata riutilizzata la stessa classe usata per la clausola Having di basso livello.



Diagrammi UML di riepilogo

Si propongono ora tre diagrammi delle classi di riepilogo, costituiti da tutti i diagrammi illustrati precedentemente, uniti tra loro in modo da dare una visione complessiva della struttura ad oggetti realizzata.

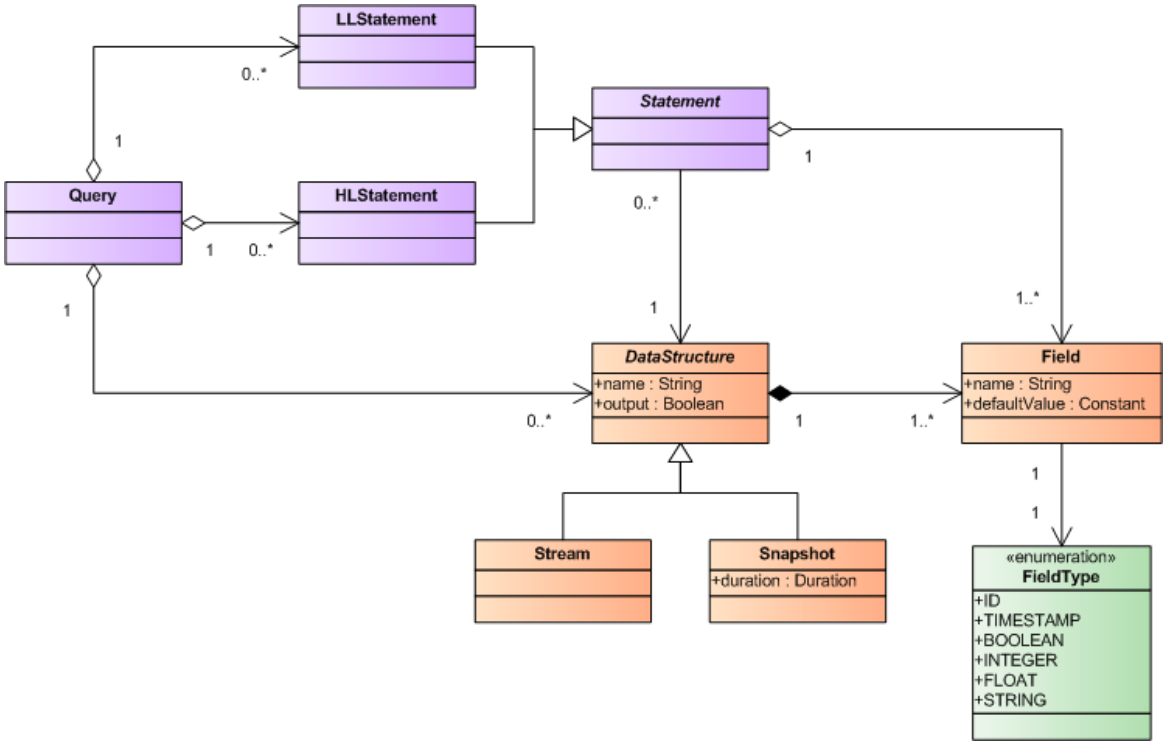


Diagramma delle classi relative alle query e alle strutture dati

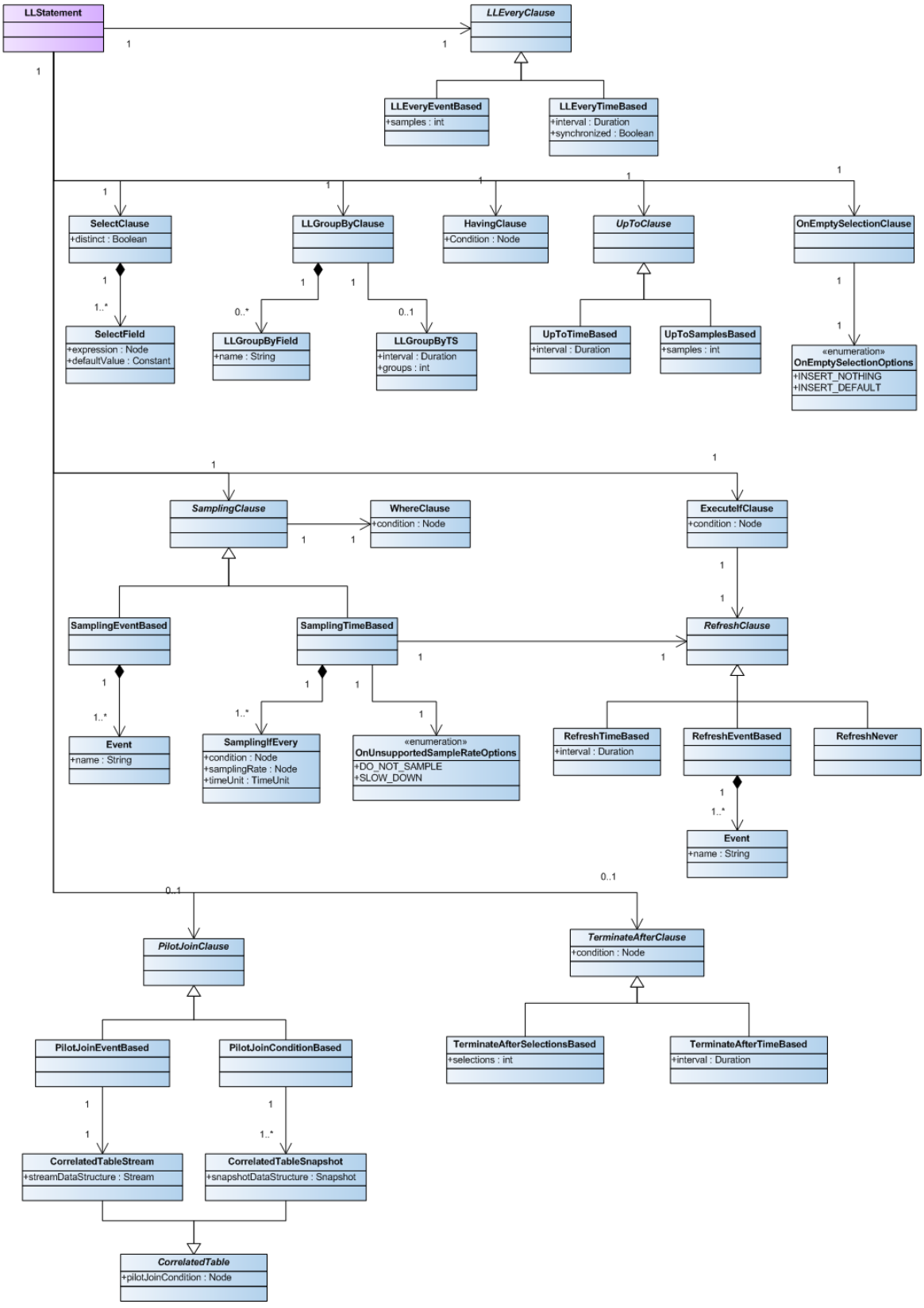


Diagramma delle classi di basso livello

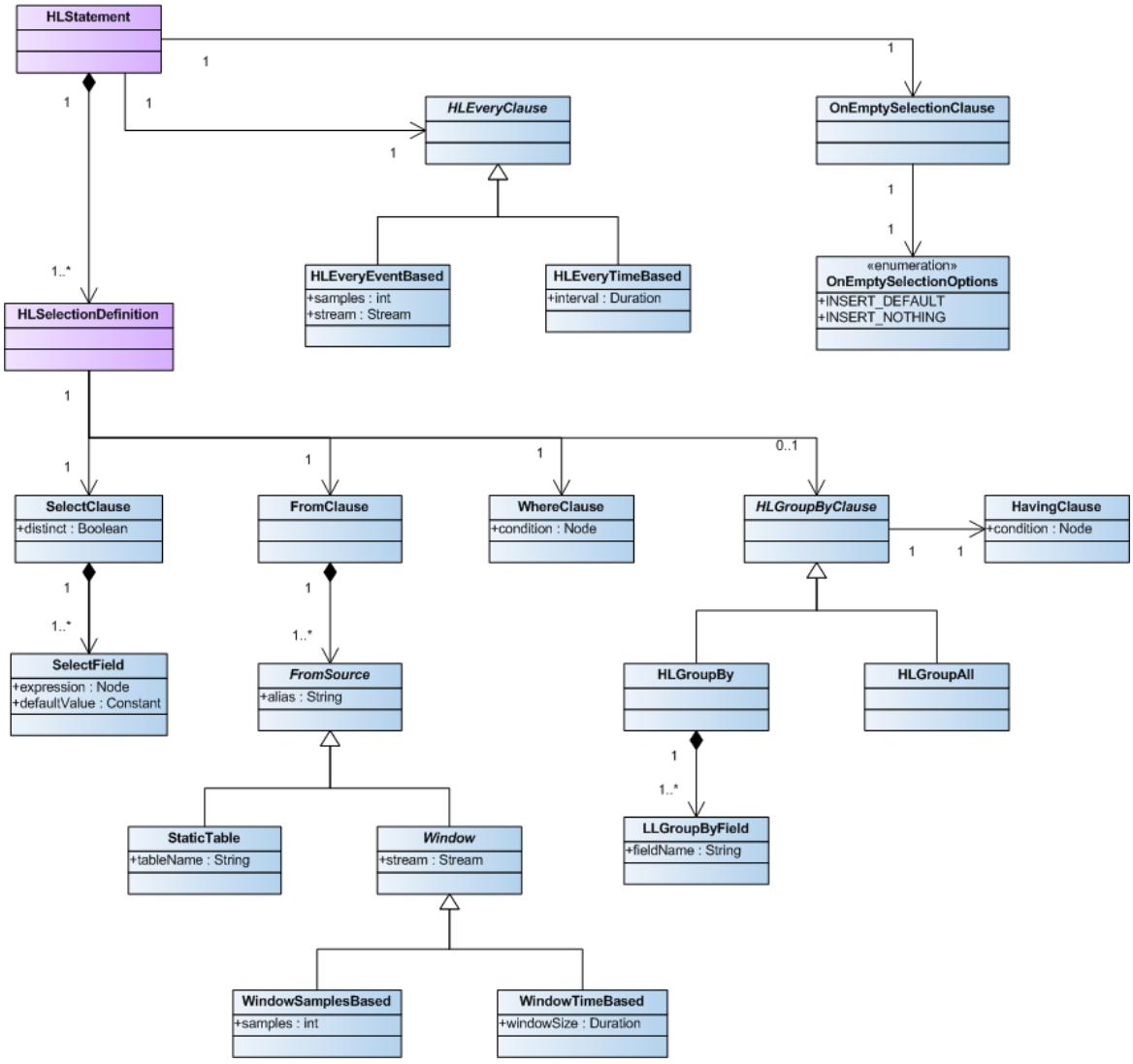


Diagramma delle classi di alto livello

Un esempio

In questo capitolo si analizzerà una semplice query facente parte del caso di studio del progetto ART DECO. La query, inserita dall'utente, verrà analizzata dal parser che la tradurrà in una struttura ad oggetti. L'esempio presentato è volutamente molto semplice e sfrutta solo una parte delle caratteristiche del linguaggio analizzate nei capitoli precedenti.

Consideriamo una semplice query per il monitoraggio delle condizioni ambientali di una vigna. In particolare si vuole campionare la temperatura e l'umidità ogni 30 minuti, e restituire questi valori accompagnati dalla posizione del sensore, solo se la temperatura è in intervallo critico. Per avere un risultato più accurato, ogni nodo dovrà campionare i suoi sensori ogni 10 minuti e considerare la media dei tre valori.

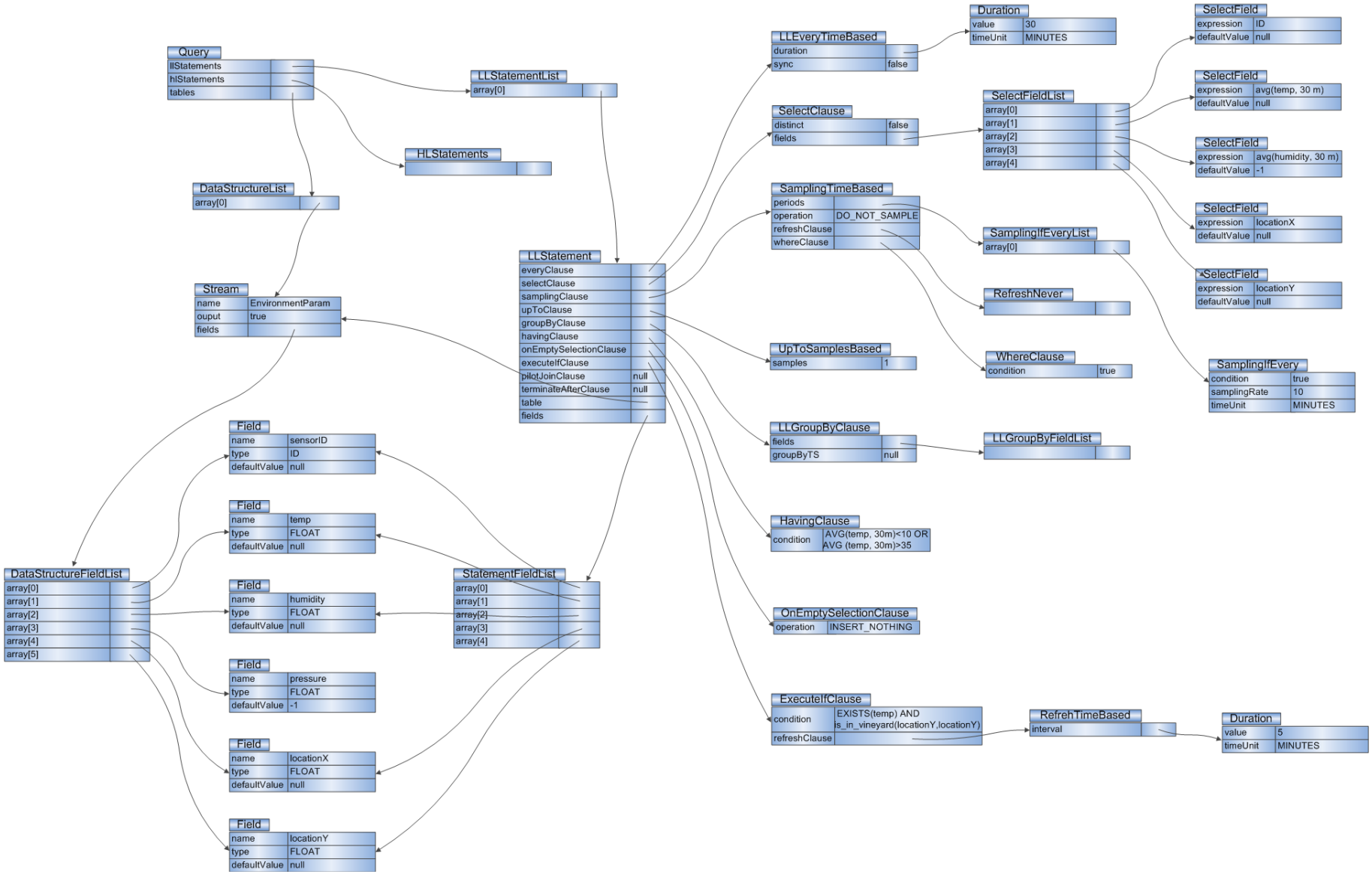
```
CREATE OUTPUT STREAM EnvironmentParam (sensorID ID, temp FLOAT, humidity
FLOAT, pressure FLOAT DEFAULT -1, locationX FLOAT, locationY FLOAT)

INSERT INTO STREAM EnvironmentParam (sensorID, temp, humidity,
locationX, locationY)
LOW:
   EVERY 30 m
   SELECT ID, AVG(temp, 30 m), AVG(humidity, 30 m) DEFAULT -1,
      locationX, location
   HAVING AVG(temp, 30 m)<10 OR AVG(temp, 30 m)>35
   SAMPLING EVERY 10 m
   EXECUTE IF EXISTS(temp) AND is_in_vineyard(locationX, locationY)
      REFRESH EVERY 5 m
```

Per la query proposta è sufficiente un unico statement di basso livello. Essa è composta da due parti. La prima crea uno stream di output di nome *EnvironmentParam* e costituita dai campi *sensorID* di tipo ID e *temp*, *humidity*, pressure, *locationX* e *locationY* di tipo float; per il campo pressure definisce un valore di defalut da utilizzare in caso di mancanza di un valore. La seconda inserisce al suo interno ogni 30 minuti il risultato di una select, che campiona i dati ogni 10 minuti e li filtra utilizzando la clausola

Having. Questa assicura che solo i record contenenti un valore critico per la temperatura siano restituiti all'utente. Alla query partecipano solo i nodi che rispettano determinati requisiti, specificati nella clausola Execute If: le funzione *EXISTS(temp)*, di tipo built in, restituisce *true* se sul nodo è presente un sensore di temperatura, mentre *is_in_vineyard(locationX, locationY)*, definita dall'utente, controlla se il nodo che ha campionato è situato all'interno della vigna da monitorare. I valori di queste due funzioni vengono aggiornati ogni 5 minuti.

La select non inserisce alcun valore nel campo *pressure* dello stream, quindi per il riempimento dei campi verrà utilizzato il valore di default -1, specificato durante la creazione dello stream.

Analizzata dal parser, questa query viene tradotta nella struttura ad oggetti in figura. Si noti come anche per alcune clausole non utilizzate viene istanziato l'oggetto corrispondente. Per esempio la clausola Up To non è stata specificata, e quindi è stato creato un oggetto *UpToSamplesBased* con l'attributo *samples* uguale a 1. Altre clausole, come Pilot Join e Terminate After, non richiedono la creazione dell'oggetto corrispondente in caso di assenza nella query.


Query
llStatements
hlStatements
tables
LLStatementList
array[0]
HLStatements
DataStructureList
array[0]
Stream
name EnvironmentParam
ouput true
fields
LLStatement
everyClause
selectClause
samplingClause
upToClause
groupByClause
havingClause
onEmptySelectionClause
executeIfClause
pilotJoinClause null
terminateAfterClause null
table
fields
LLEveryTimeBased
duration
sync false
Duration
value 30
timeUnit MINUTES
SelectClause
distinct false
fields
SelectFieldList
array[0]
array[1]
array[2]
array[3]
array[4]
SelectField
expression ID
defaultValue null
SelectField
expression avg(temp, 30 m)
defaultValue null
SelectField
expression avg(humidity, 30 m)
defaultValue -1
SelectField
expression locationX
defaultValue null
SelectField
expression locationY
defaultValue null
SamplingTimeBased
periods
operation DO_NOT_SAMPLE
refreshClause
whereClause
SamplingIfEveryList
array[0]
RefreshNever
WhereClause
condition true
SamplingIfEvery
condition true
samplingRate 10
timeUnit MINUTES
UpToSamplesBased
samples 1
LLGroupByClause
fields
groupByTS null
LLGroupByFieldList
HavingClause
condition AVG(temp, 30m)<10 OR AVG (temp, 30m)>35
OnEmptySelectionClause
operation INSERT_NOTHING
ExecuteIfClause
condition EXISTS(temp) AND is_in_vineyard(locationY,locationY)
refreshClause
RefreshTimeBased
interval
Duration
value 5
timeUnit MINUTES
DataStructureFieldList
array[0]
array[1]
array[2]
array[3]
array[4]
array[5]
StatementFieldList
array[0]
array[1]
array[2]
array[3]
array[4]
Field
name sensorID
type ID
defaultValue null
Field
name temp
type FLOAT
defaultValue null
Field
name humidity
type FLOAT
defaultValue null
Field
name pressure
type FLOAT
defaultValue -1
Field
name locationX
type FLOAT
defaultValue null
Field
name locationY
type FLOAT
defaultValue null

Conclusioni

In questo lavoro è stato analizzato nel suo complesso il linguaggio PERLA. A partire dalla grammatica è stata creata una possibile struttura ad oggetti che possa rappresentare ogni query supportata dal linguaggio. Successivamente le classi progettate sono state realizzate utilizzando Java.

Per ogni classe è stato implementato il metodo toString() in modo che, dando la query in input al parser, il metodo toString() della classe Query restituisca una stringa equivalente dal punto di vista semantico a quella inserita dall'utente. Se in un oggetto manca un attributo di tipo opzionale, per esempio se l'utente non ha specificato un valore di default, la funzione restituisce tra parentesi angolari il nome delle clausole mancanti invece del valore *null*.

Si riporta a titolo di esempio il metodo toString() della classe WindowTimeBased.

```
public String toString() {
	String stringToReturn = "";
	String aliasName;

	// Aggiunge il nome dello stream
	if((this.getStream() == null) || (this.getStream().getName() == null)) {
		stringToReturn += "<stream_name>";
		aliasName = "<alias>";
	} else {
		stringToReturn += this.getStream().getName();
		aliasName = this.getStream().getName();
	}

	// Aggiunge la parentesi aperta
	stringToReturn += "(";

	// Aggiunge la durata della finestra
	if(this.windowSize == null)
		stringToReturn += "<duration>";
	else
		stringToReturn += this.windowSize;

	// Aggiunge la parentesi chiusa e la parola chiave AS
	stringToReturn += ") AS ";

	// Aggiunge l'alias
	if(this.getAlias() == null)
		stringToReturn += aliasName;
	else
		stringToReturn += this.getAlias();

	// Restituisce la stringa costruita
	return stringToReturn;
}
```

Stato dell'arte

Al momento è stata realizzato lo scheletro del parser utilizzando JavaCC. Il prossimo passo consiste nel completare il parser inserendo il codice relativo alla creazione degli oggetti realizzati nel presente lavoro. Un altro progetto si sta invece occupando della progettazione e realizzazione degli oggetti logici. La fase immediatamente successiva sarà la progettazione del buffer locale e del motore di esecuzione delle query.

Appendice A: Grammatica

Creation statements

```
CREATE [OUTPUT] STREAM <Data Structure Name>
    ( <Data Structure Field> <Field Type> [DEFAULT <Signed Constant>], … )


CREATE [OUTPUT] SNAPSHOT <Data Structure Name>
    ( <Data Structure Field> <Field Type> [DEFAULT <Signed Constant>], … )
WITH DURATION <Duration>
```

Low level creation statements

```
CREATE [OUTPUT] STREAM <Data Structure Name>
    ( <Data Structure Field> <Field Type> [DEFAULT <Signed Constant>], … )
AS LOW:
    EVERY
    {
        <Duration> [SYNCHRONIZED] |
        <Integer Constant> SAMPLES | ONE
    }
    #LLSelection


CREATE [OUTPUT] SNAPSHOT <Data Structure Name>
    ( <Data Structure Field> <Field Type> [DEFAULT <Signed Constant>], … )
    WITH DURATION <Duration>
AS LOW:
    #LLSelection
```

Low level insertion statements

```
INSERT INTO STREAM <Data Structure Name> [( <Data Structure Field>, … )]
LOW:
    EVERY
    {
        <Duration> [SYNCHRONIZED] |
        <Integer Constant> SAMPLES | ONE
    }
    #LLSelection


INSERT INTO SNAPSHOT <Data Structure Name> [( <Data Structure Field>, … )]
LOW:
    #LLSelection
```

Low level selection statement

```
#LLSelection:

SELECT [DISTINCT | ALL]
    Expression:
    {
        ID | GROUP_TS |
        EXISTS ( { <Logical Object Field> | ALL } ) |
        <Logical Object Field> |
        <Pilot Join Field> |
        <Constant> |
        <Function> ( [<Expression>, …] ) |
        <Aggregation operator> ( <Attribute>,
            {<Duration> | <Integer Value> SAMPLES | ONE} [, <Condition>] )
    }
    [DEFAULT <Constant>], …

[GROUP BY
    {
        TIMESTAMP (<Duration> , <Integer Constant> GROUPS) |
        <Logical Object Field>
    }
    [, <Logical Object Field> … ]
]

[HAVING <Condition>]

[UP TO { <Duration> | <Integer Constant> SAMPLES | ONE } ]

[ON EMPTY SELECTION { INSERT NOTHING | INSERT DEFAULT }]

SAMPLING
    {
        ON EVENT <Logical Object Event>, …
    |
        {
            IF <Condition> EVERY <Expression> <Time Unit>, …
            ELSE EVERY <Expression> <Time Unit>
         |
            EVERY <Expression> <Time Unit>
         }
         [ON UNSUPPORTED SAMPLE RATE { DO NOT SAMPLE | SLOW DOWN }]
         [REFRESH {ON EVENT <Logical Object Event>,… | EVERY <Duration> | NEVER}]
    }
    [WHERE <Condition>]

[PILOT JOIN <Data Structure Name> ON <Condition>, … ]

[EXECUTE IF <Condition>
    [REFRESH {ON EVENT <Logical Object Event>, … | EVERY <Duration> | NEVER}]
]

[TERMINATE AFTER { <Duration> | <Integer Constant> SELECTIONS }]
```

High level creation statements

```
CREATE [OUTPUT] STREAM <Data Structure Name>
    ( <Field Name> <Field Type> [DEFAULT <Default Value>], … )
AS HIGH:
    EVERY
    {
        <Duration> |
        <Integer Constant> SAMPLES IN <Data Structure Name> |
        ONE IN <Data Structure Name>
    }

#HLSelection:

CREATE [OUTPUT] SNAPSHOT <Data Structure Name>
    ( <FieldName> <FieldType> [DEFAULT <DefaultValue>], … )
WITH DURATION <Duration>
AS HIGH:
    #HLSelection
```

High level insertion statements

```
INSERT INTO STREAM <Data Structure Name> [( <Data Structure Field>, … )]
HIGH:
    EVERY
    {
        <Duration> |
        <Integer Constant> SAMPLES IN <Data Structure Name>
        ONE IN <Data Structure Name>
    }
    #HLSelection

INSERT INTO SNAPSHOT <Data Structure Name> [( <Data Structure Field>, … )]
HIGH:
    #HLSelection
```

High level selection statement

```
#HLSelection:

#HLSelectionClause [UNION [ALL] #HLSelectionClause] …
[ON EMPTY SELECTION { INSERT NOTHING | INSERT DEFAULT }]


#HLSelectionClause:

SELECT [DISTINCT | ALL]
    Expression:
    {
        <Window Field> |
        <Constant> |
        <Function> ( [<Expression>, …] ) |
        <Aggregation operator> ( <Attribute> [, <Condition>] )
    }
    [DEFAULT <Constant>], …

FROM <Data Structure Name> [( <Duration> | <Integer Constant> SAMPLES | ONE )]
    [AS <Data Structure Name>], …

[WHERE <Condition>]

[GROUP { BY { <Window Field>, … } | ALL }
    [HAVING <Condition>]
]
```

Bibliografia

[1] http://artdeco.elet.polimi.it

[2] http://www.ricercaitaliana.it/firb/dettaglio_firb-RBNE05C3AH.htm

[3] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirc, “A survey on sensor networks”, *IEEE Communications Magazine*, vol. 40, no. 8, pp. 102–114, August 2002

[4] P. Atzeni, S. Ceri, S. Paraboschi and R. Torlone, “Basi di dati: modelli e linguaggi di interrogazione, seconda edizione”, McGraw-Hill Italia, pp. 1-395, 2006

[5] M. Fortunato and M. Marelli, “Design of a declarative language for pervasive systems”, Tesi di laurea specialistica, Politecnico di Milano, pp. 1-161, 2007

[6] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, “TinyDB: an acquisitional query processing system for sensor network”, *ACM Trans. Database Syst.*, vol. 30, no. 1, pp. 122–173, 2005

[7] L. Baresi, D. Braga, M. Comuzzi, F. Pacifici, and P. Plebani, “A service-based infrastructure for advanced logistics”, *in IW-SOSWE ’07: 2nd international workshop on Service oriented software engineering*. New York, NY, USA: ACM Press, 2007, pp. 47–53