

Design of PerLa, a Declarative Language and a Middleware Architecture for Pervasive Systems

Fabio A. Schreiber, Romolo Camplani, Marco Fortunato, Marco Marelli, Guido Rota

Abstract

This project aims at defining a completely declarative language to deal with wireless sensor networks and, more generally, with pervasive systems. In this work the main problems to be considered for the design of such a language are presented and discussed. Then, a software architecture for pervasive systems is introduced and the *PerLa* language proposal is presented. They allow to collect data from different nodes of a pervasive system and to manage them, hiding the complexity due to the large underlying heterogeneity of devices, which can span from passive RFID(s) to ad-hoc sensor boards and to portable computers. An important feature of the presented middleware is to make the integration of new technologies in the system easy, through the use of device self-description. Finally, some simple examples are reported to show how the language can be used by a final user. The considered case study is related to the monitoring of the wine production process from the vineyard to the table.

Data	03/03/2010
Tipo di prodotto	Deliverable R.A. 11.1b
Stato	Final
Unità responsabile	Politecnico di Milano
Unità coinvolte	PoliMI
Autore da contattare	Fabio A. Schreiber
Versione	2.0

Lista delle modifiche

Versione	Data	Stato	Note
1.0	10/01/2008		First draft
2.0	03/03/2010	Final	Revised version

Index

Sommario	4
1 Goals and phases of the project	7
1.1 Projects goals evolution	7
1.2 Short survey of system features and relations with similar projects.....	9
1.3 Definition of pervasive computing	12
2 Main issues in language design	13
2.1 Data representation issues.....	13
2.2 Physical devices issues	15
2.3 Functional characteristics of the language.....	18
2.4 Non functional characteristics of the language.....	18
3 Architecture for pervasive systems	21
3.1 Logical object abstraction.....	21
3.2 <i>PerLa</i> Middleware.....	22
3.3 Functionality Proxy Component.....	24
3.3.1 Channel virtualization.....	24
3.3.2 Channel multiplexing	25
3.3.3 FPC functionalities	26
3.3.4 FPC architecture	27
3.4 FPC Factory and Registry.....	28
3.5 Device self description file	29
4 Language design	31
4.1 Query analysis	31
4.2 Query decomposition (from the system point of view)	32
4.3 Query graph	33
4.4 Execution of a user query on the architecture for pervasive systems	39
5 Language syntax and semantics	45
5.1 Types of SQL Statements	45
5.2 Creation statements.....	49
5.3 Insertion statements	50
5.4 Low level SELECT definition	51
5.4.1 Sampling section	52
5.4.2 Data management section.....	54
5.4.3 Execution conditions	63
5.4.4 Low level statements UML	65
5.5 High level SELECT definition	67
5.6 Actuation statements.....	69
5.7 Expressions and conditions.....	71

5.8	Constants and identifiers	76
5.9	Summary of the grammar	78
5.9.1	Creation statements	79
5.9.2	Low level creation statements	79
5.9.3	Low level insertion statements	79
5.9.4	Low level selection statement	80
5.9.5	High level creation statements.....	81
5.9.6	High level insertion statements	81
5.9.7	High level selection statement.....	82
5.9.8	Actuation query level statement	82
6	Examples of queries	83
6.1	Low Level queries with event based sampling.....	83
6.1.1	Example 1	83
6.1.2	Example 2.....	85
6.1.3	Example 3.....	86
6.1.4	Example 4.....	87
6.1.5	Example 5.....	88
6.2	Low level queries with time based sampling.....	89
6.2.1	Example 6.....	89
6.2.2	Example 7.....	91
6.3	Low level and High level queries	92
6.3.1	Example 8.....	92
6.3.2	Example 9.....	93
6.4	Queries with pilot join	95
6.4.1	Example 10.....	95
6.4.2	Example 11.....	96
6.4.3	Example 12.....	98
6.5	Actuation queries.....	99
6.5.1	Example 13.....	99
6.5.2	Example 14.....	100
6.6	Other queries.....	100
6.6.1	Example 15.....	100
6.6.2	Example 16.....	102
6.7	Case study queries	103
6.7.1	Vineyard monitoring	103
6.7.2	Transport monitoring.....	105
7	Conclusions	107
	Appendixes	109
A	Parser implementation	110
B	Language EBNF	117
	Bibliography	139



Sommaro

Esistono molti contesti applicativi le cui fasi operative vengono monitorate tramite un elevato numero di dispositivi, muniti di appositi sensori, tra loro molto differenti per tecnologia e modalità di funzionamento. Si consideri, a titolo di esempio, il processo di produzione e trasporto dei vini di qualità che costituisce uno dei casi di studio di *ART DECO*, un progetto finanziato dal *Ministero dell'Università e della Ricerca italiano*.

Il processo deve essere monitorato in ogni sua fase, a partire dalla coltivazione dell'uva nel vigneto fino al trasporto su camion ed al mantenimento delle bottiglie nelle cantine. Ognuna di queste fasi richiede degli impianti di sensoristica molto diversi tra loro: le reti di sensori wireless possono essere la soluzione ideale per il monitoraggio di parametri quali la temperatura e l'umidità nel vigneto; l'applicazione di etichette *RFID* alle bottiglie può essere appropriata per la loro identificazione durante e dopo il trasporto; i dispositivi *GPS* permettono invece di conoscere l'attuale posizione dei camion che trasportano le bottiglie.

E' evidente l'interesse che può esistere nel considerare nel loro insieme tutti i dati che possono essere raccolti tramite questi impianti di monitoraggio. Ciò, però, risulta difficile in quanto le diverse tecnologie utilizzate non sono fra loro integrate e ciascuna di esse presenta all'utente delle interfacce di controllo ed interrogazione differenti.

Il lavoro presentato in questo report ha come obiettivo quello di analizzare il problema sopra descritto e porre le basi per una sua possibile risoluzione: l'idea principale è quella di definire *PerLa*, un linguaggio di alto livello completamente dichiarativo, che permetta all'utente di interrogare un sistema pervasivo utilizzando una sintassi molto simile a quella dello *SQL* standard. Con il termine "*sistema pervasivo*" si intende una rete eterogenea formata da dispositivi molto differenti tra loro: reti di sensori, palmari, etichette *RFID*, ecc. Alla base della definizione di *PerLa* vi è l'idea di astrarre il sistema pervasivo fino a considerarlo alla stregua di una base di dati, con l'obiettivo di mascherare la complessità della rete sottostante e mettere in risalto le informazioni che vengono raccolte, nonché le relazioni che sussistono tra di esse. In questo modo l'utente, non dovendosi preoccupare delle peculiarità dei singoli dispositivi, può interrogare l'intero sistema in maniera facile e veloce.

Lo scopo iniziale del progetto era quello di creare un linguaggio dichiarativo per l'interrogazione di reti costituite solamente da sensori tra loro omogenei. Si voleva riprendere l'idea di *TinyDB*, che storicamente è stato il primo progetto ad aver proposto la possibilità di astrarre una rete di sensori come una base di dati. Nonostante l'idea proposta fosse molto innovativa ed abbia suscitato molto interesse in ambito accademico, questo progetto è rimasto un prototipo senza mai essersi trasformato in un'applicazione commerciale.

Si è pensato quindi di utilizzare *TinyDB* come base di studio da cui partire per tentare di definire un sottoinsieme minimo di funzionalità necessarie in un qualsiasi sistema per

l'interrogazione di reti di sensori. L'obiettivo originale era quello di implementare un sistema che supportasse tali funzionalità e che fosse facilmente portabile su diverse architetture fisiche. In seguito, durante la fase di analisi dei requisiti, lo scopo del progetto si è modificato: l'idea di poter interrogare contemporaneamente diversi dispositivi, tra loro eterogenei sia nel tipo che nella tecnologia, ha spostato il focus sulla definizione di un linguaggio dichiarativo per sistemi pervasivi.

Il *Capitolo 1* presenta alcuni progetti simili, classificandoli secondo i diversi approcci utilizzati ed evidenziando le differenti peculiarità, potenzialità e limitazioni. A partire dall'analisi di questi progetti sono stati definiti gli aspetti fondamentali del linguaggio che viene proposto in questo report. Essi sono presentati nel *Capitolo 2* e sono classificabili nelle seguenti quattro categorie:

- Aspetti relativi alla rappresentazione e all'astrazione dei dati
- Aspetti relativi alla gestione dei dispositivi fisici
- Aspetti relativi alle caratteristiche funzionali del linguaggio
- Aspetti relativi alle caratteristiche non funzionali del linguaggio

L'eterogeneità dei dispositivi considerati ha portato alla necessità di suddividere il linguaggio in due parti:

- Un **Linguaggio di Basso Livello**, il cui scopo è quello di descrivere le operazioni di campionamento e le manipolazioni da effettuare sui dati raccolti dal singolo nodo (raggruppamenti, aggregazioni e filtraggi).
- Un **Linguaggio di Alto Livello**, che ha il compito di descrivere le manipolazioni da effettuare a partire dai dati prodotti dal basso livello.

Entrambi i linguaggi hanno una sintassi simile ad *SQL*, ma la semantica del linguaggio di basso livello si differenzia in buona misura da quella dello *SQL* standard, sia perché introduce clausole per la gestione del campionamento, sia perché è stato studiato per agevolare la generazione di dati aggregati a partire dai dati campionati. La semantica del linguaggio di alto livello è invece molto simile a quella dei linguaggi di interrogazione per database di streaming. È anche prevista una terza parte del linguaggio per la gestione di query di attuazione.

Il *Capitolo 3* presenta l'infrastruttura adottata per descrivere la semantica di *PerLa*. A livello applicativo, un analizzatore di query manipola le interrogazioni sottomesse dall'utente e, utilizzando un componente dedicato (*registry*), ottiene informazioni circa gli oggetti logici che compongono il sistema. Quindi, a partire da queste informazioni e da quanto richiesto dall'utente, seleziona gli oggetti logici che dovranno prender parte all'esecuzione della query. Grazie all'architettura appena descritta, l'intero sistema pervasivo può essere astratto come un insieme di oggetti logici che astraggono i dispositivi fisici. In questo modo, ogni richiesta ad un nodo del sistema si traduce, di fatto, in una richiesta al corrispondente oggetto logico, tramite l'interfaccia da esso esposta. Proprio la definizione precisa di questa interfaccia permette di descrivere la semantica del linguaggio in termini di interazioni con gli oggetti logici. Ad esempio, una richiesta di campionamento ad un nodo che possiede un sensore di temperatura viene vista dal linguaggio come la lettura di un attributo dell'interfaccia esposta da quel nodo. Similmente, il passaggio di un tag *RFID* da un certo lettore viene visto dal linguaggio come un evento scatenato dalla stessa interfaccia.

Il *Capitolo 4* descrive come le query sottoposte dall'utente possano essere rappresentate tramite grafi, i cui nodi definiscono le strutture dati e le componenti di alto e di basso livello delle query stesse. Viene in seguito presentato il meccanismo con cui tale grafo è utilizzato dal sistema per decomporre, distribuire ed eseguire la query sui vari nodi.

La definizione dei concetti appena descritti permette una migliore comprensione della semantica del linguaggio, che viene presentato in maniera formale nel *Capitolo 5* tramite una grammatica *EBNF*, scritta in modo tale da essere facilmente leggibile e comprensibile al lettore, ma non direttamente adatta come input per i software generatori di parser.

Il *Capitolo 6* propone vari esempi di query per aiutare il lettore a comprendere meglio la semantica del linguaggio e le varie funzionalità offerte da quest'ultimo. Inoltre riprende il caso di studio relativo al monitoraggio dei vini e propone alcuni esempi realistici di utilizzo del linguaggio.

Nel *Capitolo 7*, oltre a essere presentate le conclusioni sono suggeriti alcuni possibili miglioramenti che potranno essere apportati sia al linguaggio che al middleware.

L'*Appendice A* fornisce alcuni dettagli implementativi relativi alla realizzazione del parser per il linguaggio *PerLa*, mentre l'*Appendice B* riporta la grammatica completa del linguaggio, in forma non ambigua, scritta utilizzando la sintassi del generatore di parser *JavaCC*.

La documentazione aggiornata del progetto e delle attività ad esso correlate è reperibile sul sito <http://perla.dei.polimi.it>

1 Goals and phases of the project

The main idea of the project is to define a completely declarative language that allows the user to query a pervasive system in a similar way as *SQL* allows to query traditional databases. As better explained in the following, the “*pervasive system*” term refers to a large heterogeneous network composed of many devices, belonging to different technologies like wireless sensors networks (*WSN*) [1] [2], *RFID* systems, *GPS* and other kinds of sensors.

We aim at providing a database like abstraction of the whole network in order to hide the high complexity of low level programming and allowing users to retrieve data from the system in a fast and easy way.

The original goal of our project was to define a *SQL* like declarative language only for *WSNs*. We aimed at defining at least a working subset of the functionalities supported in similar existing systems. Our idea was to develop an architecture deployable on different *WSN* technologies with little effort, but we were not initially interested in supporting query execution over different device technologies at run time. So, the goal was to develop a software executable on homogeneous sets of nodes, either in terms of technologies or of types of sensors.

The first idea changed many times during the initial phase of the project; in this chapter we describe the main steps that led us to the final goal of defining a declarative language for pervasive systems. Then we report a short survey on existing similar projects.

1.1 Projects goals evolution

Firstly, we considered the possibility of developing a system really supporting heterogeneity, also at runtime. Allowing the user to write queries, whose execution involves *WSN* nodes belonging to different technology classes, seemed very interesting.

To reach this goal we had to deal with two main issues: implementing communication between nodes implemented with different technologies and understanding the additional features required by the language to support runtime heterogeneity. Communication issues should be resolved by a middleware that provides a set of *APIs* to manage the different hardware platforms in a uniform way. At this point, the language becomes a user friendly interface to that *API*. An implementation of the language is then a compiler or an interpreter that translates queries into a set of *API* calls.

The effort in language design does not change with the introduction of the full heterogeneity requirement. In fact the main part of the work is still the research of all the common features of the different technologies.

The next extension to our project goals has been the introduction of new kinds of devices. We realized that our considerations about the language are not strictly limited to wireless sensor networks, and can be easily extended to any kind of mobile device equipped with sensors and communication tools (for example *PDA*s). In fact the only requirement for a device to be used in our architecture is the ability of sampling some physical magnitude and sending collected

data to a base station. The only effect, rising from enlarging the set of supported devices, is the need of finding a more general method to manage the language non functional characteristics. In fact, while in *WSNs* the main issue to be handled is the power management, other aspects have to be considered in other types of devices.

From the point of view of the language functional characteristics, some problems came out when we tried to include also the possibility of managing *RFID* tags. There are many types of *RFID* technologies (*HF*, *UHF*, *UWB*, etc.), but in any case simpler tags are not equipped with sensors and cannot perform data manipulation or data transmission.

As we will better describe in the following, we found two possible solutions to this problem taking into account that the information of interest is the presence of a tag into the range area of an *RFID* reader.

The first one abstracts an *RFID* tag as it were a sensor, and the *ID* of the last reader which sensed the tag is considered as the sampled data. On the opposite, the second solution abstracts an *RFID* reader as it were a sensor, and the *ID* of the last tag detected by the reader is considered as the sampled data. In both cases, the communication with the network node that submits queries is done by readers, so it is clear that the first solution provides a greater level of abstraction and then requires a greater amount of middleware to be implemented.

Moreover, with the introduction of *RFID* devices, we had to define also an event based semantics for data managing at the language level. In fact, differently from other kinds of nodes, the sampling operation on passive *RFID* is done when the tag is sensed by a reader and not at fixed instants. A time based semantics for *RFID* sampling can also be defined, but the event based one must be considered too.

The last step in the project goals definition has been the extension of the declarative language – named *PerLa* from *PERvasive Language* - to generic pervasive systems. Although our first idea was to implement the language on some existing platforms for pervasive systems management, we realized that all the existing systems don't provide the needed heterogeneity support. In fact, the most common approach is to define a high level interface that must be implemented by all the devices that aim at joining the system. This interface often assumes that Java programming is supported at the device level or, at least, that a TCP/IP stack is available to communicate with the device. These requirements can be too strong when dealing with a generic pervasive system, since also very small or ad-hoc designed devices must be supported.

Thus, we decided to design and implement a new middleware infrastructure in order to provide a more flexible support to handle heterogeneity. This middleware is based on the concept of logical object, that is the basic abstraction to define the sampling operation on a generic device and to homogeneously access heterogeneous nodes. As it will be described in the following section, the formal syntax and semantics of *PerLa* language have been completely defined over the concept of “logical object”.

This approach allowed us to split the data management issues, that are handled by the language, from the low level details issues related to devices access, that are fully handled by the middleware platform.

The full updated documentation on the *PerLa* project can be found at <http://perla.dei.polimi.it>

1.2 Short survey of system features and relations with similar projects

It is not the first time that a database like abstraction approach is used in wireless sensor networks. In fact, in the last years, there was an increasing interest of the academic world on the problem of finding a high level model for programming this kind of systems.

Many different projects on *WSNs* and pervasive systems have been developed and they introduced a lot of different network abstractions and high level programming models. Some of this projects are *TinyDB* [3] [4] [5], *DNS* [6], *Cougar* [7], *Maté* [8], *Impala* [9], *Sina* [10], *DsWare* [11], *MaD-WiSe* [12] [13], *Kairos* [14], *SWORD* [15] and *GSN* [16]. There is also a certain number of surveys that try to classify each of the previous projects depending on the type of the provided abstraction [17] [18] [19]. In particular, S. Hadim and N. Mohamed [19] evaluated a set of representative middleware, analyzing the different approaches with their advantages and disadvantages. They also wrote out a clear summarizing table that is surely helpful to get the state of art on high level *WSNs* programming techniques.

For our purposes, we will mainly focus on two classification criteria. The first one is related to the architecture the project works on, while the second one is related to the kind of language provided for interacting with the system.

The architectural-based classification depends on:

- **The type of supported nodes** (sensors, *RFID*, *PDA*, etc.)
- **The specific hardware model** for each of the previous types (*Crossbow MOTE* [20], *Intel MOTE* [21], , ...; *HF tags*, *UHF tags*, *UWB tags*, ...)
- **The supported heterogeneity level**

The heterogeneity level can be evaluated on two different dimensions:

- **Deploy-time heterogeneity**: this dimension indicates the ability of the system to be deployed on different kinds of hardware platforms, so it can be considered as an index of the system hardware portability. For example, we will say that a system supports the deploy-time heterogeneity if it can be deployed on a *MICA MOTES WSN*, but can also be re-compiled with little effort to be used on a different kind of network.
- **Run-time heterogeneity**: this dimension indicates the ability of the system to manage a network composed of different kinds of nodes (*RFID*, sensors, etc.).

We can then find out three macro groups of projects based on the supported architecture:

Architectural-based heterogeneity		Deploy-time heterogeneity	
		Little	Full
Run-time Heterogeneity	Little	Homogenous system	Partially Heterogeneous system
	Full	-	Heterogeneous system

Figure 1: Heterogeneity levels

Most of the existing *WSN* projects we have analyzed work on homogeneous systems. Sometimes they provide deploy-time heterogeneity through the underlying operative system portability (that is often *TinyOS* [22]).

The only project we found that is really intended to work simultaneously with different kinds of devices is *GSN (Global Sensor Network)*, under development at the *EPFL of Lausanne*.

The second classification we are interested in is related to the type of language provided to the final user for interacting with the system. There are mainly two kinds of languages:

- **Declarative languages** focus on the results the user expects from the system rather than the way these results should be computed. The most famous and diffused declarative language to query databases is *SQL*. For this reason, many of the analyzed projects define *SQL* like languages to query the network. *TinyDB* is the most famous and cited work on declarative languages for sensor networks since it was the first attempt to design a processing system for extracting information from a *WSN*, using a database abstraction. *TinyDB* is written in *nesC* [23] over *TinyOS* operating system and runs on *MICA Motes* architecture. The innovative idea was to abstract each node of the network as a source of tuples representing the results of sensor samplings. The union of all these tuples is a logical and distributed table called “*Sensors*”, over which some queries can be performed. The provided language has an *SQL*-like syntax but it is properly tuned in order to support some features related to the *WSN* context. More specifically, both temporal and non temporal aggregations are allowed and support for events management is provided. Moreover, the query duration can be specified through the *LIFETIME* clause that is used to establish the sensors sampling rate, granting in this way an appropriate power consumption.
- **Procedural languages** are imperative programming languages allowing the definition of the steps that should be executed by each network node to obtain the expected results. Often, systems using a procedural language provide a middleware with the *APIs* that can be called by the language itself. While in the case of declarative languages the provided network abstraction is often a database, in the case of procedural languages many different abstractions have been adopted. For example, some projects are based on the concept of Virtual Machine (*MATE* and *MAGNET*) while others are based on the idea of event programming (*IMPALA*).

In some cases the system provides both a procedural language and a declarative language. An example is *SINA* that is composed of 4 layers. The first one is the *SINA* middleware. Then the *SQTL* (*Sensor Query and Tasking Language*) is a procedural language built over the middleware. The third level provides a built in declarative query language, defined in terms of *SQTL* scripts. Finally the application level can query the network using the declarative language, but also generating and directly executing *SQTL* scripts.

The previous considerations highlight that providing an homogeneous interface to access physical heterogeneous networks and reducing the programming effort required in order to support new devices are the two main goals of any monitoring project. Achieving the first goal makes the interaction between the final user and the system easier, while achieving the second goal simplifies the developers workload. Existing projects usually focuses on a single aspect, without dealing with the other one. A comparison among some of the above cited projects is shown in Figure 2, where the *x-axis* and the *y-axis* represent the supported Devices homogeneity and the Additional programming effort respectively. *TinyDB* and *DSN* are customized for *Motes* and they don't manage device heterogeneity at all, but the programming effort required to run *TinyDB* on a different family of *Motes* is reduced to the deployment of *TinyOS* on the new technology. On the contrary, *GSN* and *SWORD* are designed to work with heterogeneous devices and they provide high level homogeneous interfaces to access the physical nodes. However, introducing new devices in a *GSN* or *SWORD* environment requires to implement the high level communication protocols imposed by the middleware.

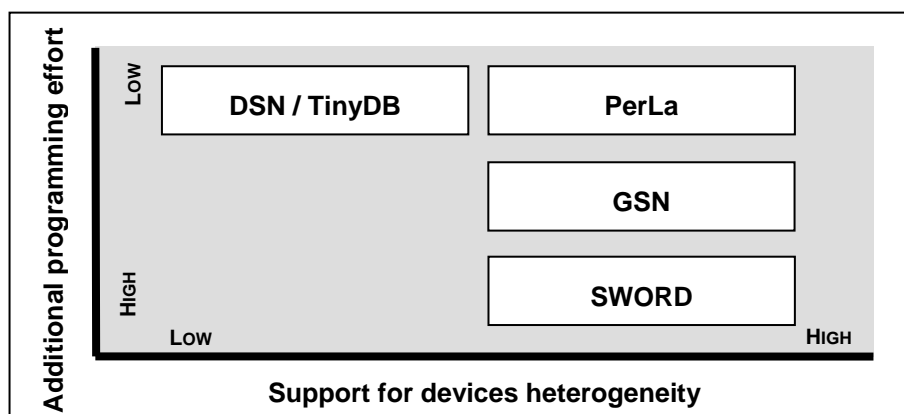


Figure 2: Monitoring projects comparison

PerLa project aims at reaching the best trade-off between the two above described aspects: a solution composed of two main entities is proposed in order to provide a complete support for heterogeneous networks. As said before, the absence of a common high level abstraction to deal with different kinds of physical devices was the main reason that led to the design of the first entity, called *PerLa* language.

Similarly, the absence of a common physical interface to access different devices in the same way was the main reason that led to the development of the second entity, called *PerLa* middleware. The goal of the whole project is then to provide a simple user interface to interact

with a pervasive system, hiding hardware heterogeneity, masking the efforts to support *Plug & Play* features and reducing the coding effort to integrate new technologies.

1.3 Definition of pervasive computing

We said that the target of this work is not just a single wireless sensor network but, rather, a full pervasive system. This chapter is intended to explain the meaning of this widely used and generic term, in order to clarify the boundaries of our project. The term “Pervasive computing” refers to a wide set of applications and it encompasses many research areas, from networking and distributed computing to software engineering, knowledge management and databases. Many are the existing projects, both in the academic and the industrial world, that could be considered part of the pervasive computing research or strictly related to it. Each of them focuses on some aspects of the whole problem and proposes a solution to handle them. The idea of ubiquitous computing, that is the key concept pervasive systems are based on, was firstly introduced by Mark Weiser in [24]. He thought to ubiquitous computing as a human-machine interaction model, characterized by the replacement of existing desktop PCs with a widespread diffusion of hardware and software components into everyday life objects. In this way, technology becomes almost hidden to the final user and it plays a background role. In fact, it is distributed on a widespread network of heterogeneous devices, called pervasive system.

Afterward, Kindberg and Fox [25] pointed out two principles a system should respect in order to be considered pervasive, accordingly to the definition given by Mark Weiser. These two characteristics are physical integration and spontaneous interoperability. The first one refers to the ability of hardware and software components to integrate and hide themselves into everyday life objects, interacting with the environment in a transparent way; the second refers to the devices ability to start communicating whenever required by the context, without an explicit developer scheduling. Existing projects related to pervasive systems seldom cover both the aspects defined by Kindberg and Fox: they mainly focus on a single feature. The works that try to reach a high level of integration typically deal with low level hardware and software and they are rarely interested in obtaining spontaneous interoperability features. Vice versa, projects that concentrate their efforts on spontaneous interoperability usually consider the integration issues only at a high abstraction level. They are not interested in the details allowing device integration and they require the devices to expose high level interfaces in order to communicate with the system (e.g.: *XML* over *HTTP* protocols). A complete pervasive application is composed of many architectural layers requiring different abilities, from low level hardware programming to the design of high level abstractions. Physical integration is achieved operating on low level layers, while spontaneous interoperability is more related to the highest levels. This is probably the reason that makes difficult and unusual achieving both goals within the same research project.

2 Main issues in language design

In this chapter we will present the main issues we found out during the first phase of the project, when trying to define the features of the language. Those issues are the questions that have to be answered to clearly bound the goals and the characteristics of the language to be designed.

We classified the issues into four categories, identifying issues related to data representation and abstraction, issues dealing with physical devices and issues strictly related to the language expressiveness, both in terms of functional and non functional features.

2.1 Data representation issues

In this section we outline the ways in which data of interest will be presented to the user. We also describe abstractions that can be used to hide physical devices and to provide a database view of the whole network. The logical steps that allow the manipulations of samplings, coming from different sensors in order to build an output table, are presented too.

First of all we decided to use a database like abstraction, but there are some differences with the *TinyDB* abstraction. In fact we think that the concept of sampling cannot be completely hidden to the user. *TinyDB* abstracts the sampling process through the concept of “*sensors*” table, without allowing the user to modify any sampling parameter except for the sampling rate. This choice makes the system easy to be used and understood by the user, but it is only feasible when all the devices in the network are homogeneous and their features are already known at design time.

In our heterogeneous system the sampling operation should be managed in a more complex way, giving the possibility of defining some complex criteria to decide which nodes will perform the sampling operation. Moreover we would like to allow the user to define other sampling details, for example, in terms of sampling rate and characteristics of the node performing the sampling.

Another problem to be managed is that different nodes can have different attributes and different limitations on the maximum achievable sampling rate. As we will better described in the following, the above considerations led us to split the language in two parts: the lower level is intended to manage the sampling operations while the higher level has the role to manipulate the sampled data producing the query results. Due to the great differences between the two parts, two different languages should be designed. The low level one deals with sampling operations and can only perform some data manipulations on data sampled from a single sensor. So its semantics cannot be very similar to that of standard *SQL*; anyhow, we tried to define an *SQL* like syntax for it. On the contrary the high level language is very similar to standard *SQL* because it must perform relational operations on the data streams produced by the low level part.

Note that the high level language has similar functions as the *TinyDB* language, because both are used to manipulate data streams coming from sensors. The low level language has not a counterpart in *TinyDB* and can be thought as a mechanism to generate data streams corresponding to the *TinyDB* “*sensors*” table (Figure 3).

The abstraction provided by the low level language does not only handle the parameters of the sampling operation, but also deals with the problem of deciding when sampling should be executed on different network nodes.

Users should be able to define the set of nodes that will take part in the query, specifying some conditions on the type (*e.g.*: *RFID*, *WSN*), the characteristics (*e.g.*: available sensors) and the current state (*e.g.*: battery level) of the nodes.

Thinking about the above problem and analyzing some case studies, we realized that in many real situations a sampling on a node should be started if and only if a certain value has been retrieved from a sampling done on another node. For example, suppose that a user requires a temperature monitoring in a certain zone. If temperature sensors are mounted over moving platforms, it is clear that the sampling operation on a node should be activated only when its platform enters the zone. Platforms locations are detected by position sensors that are nodes physically different from the temperature nodes and not directly connected to them.

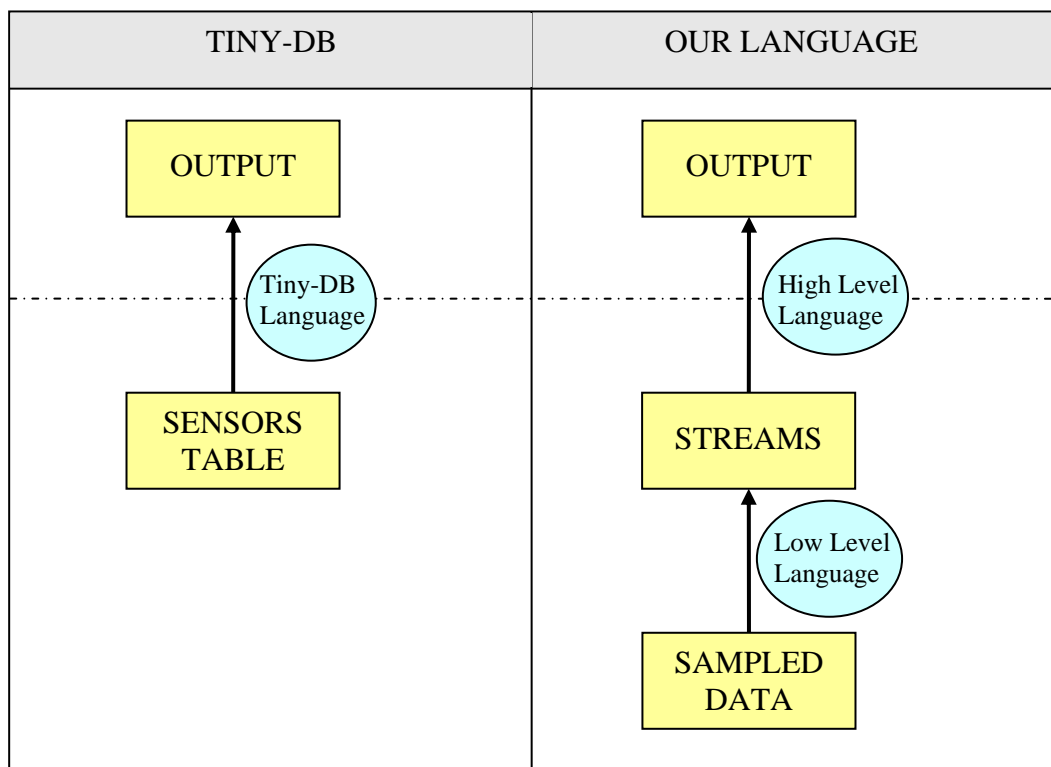


Figure 3: Comparison between TinyDB and our project

The above consideration suggested us that a specific operation should be supported to allow sampling activation depending on data sampled from other nodes. More specifically our idea is

to force a sampling operation to start when a certain record is inserted into a stream by another query.

This operation can be logically thought as a join, followed by a filtering: in line of principle the same result of the above example can be obtained by continuously sampling all the platforms positions and all the temperature nodes, then calculating the join between the platforms location stream and the temperature stream and finally eliminating all the records pertaining to platforms currently out of the considered zone. This solution is not feasible in a real system because it requires the sampling of many sensors whose results are not used to compute the output stream. This is obviously too expensive from the point of view of power consumption.

The new operation, that we will call “*Pilot Join*” in the following, was designed to avoid the above problem. To obtain this goal, a latency between the event causing the sampling start and the first sample must be accepted.

Analyzing some real situations we found out that two kinds of *pilot join* are possible:

- **Event based Pilot Join.** When an event happens (i.e. a record is inserted into a stream) a certain set of nodes should start sampling (typically for a fixed period). Suppose that, in the example of moving platforms, the temperature must be sensed once every time a platform crosses the zone borders. In this case an event based *pilot join* is required.
- **Condition based Pilot Join.** A continuous sampling (with frequency f) should be done on all the nodes that are connected to a base station that is in a list of base stations satisfying given criteria. This list is obtained extracting a fixed window from a stream and is updated with a frequency lower than f . Consider again the moving platforms example. Suppose that a running query is sensing (with low frequency) the position of all platforms and inserting them into a stream. Suppose also that the required behavior of the system is the continuous sampling of temperature sensors mounted over platforms whose last monitored position is in the zone. In this case a condition based *pilot join* is required.

The implementation of the condition based *pilot join* requires the definition of a data structure to store the list of records used as condition to activate and deactivate nodes sampling. This consideration led us to distinguish two table types that will be handled by the language.

- **Stream Tables.** Stream is the most important type of table; it is an unbounded list of records produced by a query. Streams can be used to implement the event based join: the insertion of each new record in the stream is the event used to start the low level query in the nodes matching the record.
- **Snapshot Tables.** A snapshot table is a set of records produced by a query in a given period. During each period, all the records generated by the query are stored in a buffer; at the end of the period the snapshot table is cleared and filled with the records in the buffer. This type of table can be used to implement the condition based join: the snapshot table content is the list of nodes that are currently satisfying the condition and that will be sampled during the next period.

2.2 Physical devices issues

In this paragraph we focus our attention on issues related to physical devices. We show the importance of giving a well defined meaning to the sampling concept. To reach this goal an

abstraction of the sampling operation should be provided to hide the large hardware differences existing among different types of nodes.

Another problem that is introduced is the contrast between the ideal vision of hardware at the language level and the real hardware behavior (latency, failures, etc.). This will show the importance of formally defining a concept of timestamp in order to hide the above contrast.

The semantics of the sampling operation should be analyzed for all the device types that will be supported in our system in order to provide a good high level abstraction of the sampling concept. For network nodes having some sensors on board and able to probe some measures (temperature, pressure, speed, etc.) an intuitive meaning of sampling can be provided. When a sampling operation is invoked, the required sensor is instantaneously probed and the read value is returned.

Consider now a node having on board also a storage device (*RAM, ROM, flash, etc.*). In this case the sampling operation can be extended to the reading of a section of that memory. Note that, from the user point of view, there are no differences between a sampled value obtained from probing a sensor and a value read from memory. In fact, in both cases, the user requests a sampling and the device returns a value.

So, a first abstraction of the sampling operation is a function-call with one parameter and a return value. The parameter is used to communicate to the device the name of the attribute to be sampled. It is not important if the attribute is then mapped to a real sensor or to a storage cell.

Particular attention is needed to define the sampling concept for *RFID* devices. First of all, it should be noted that both the tag and the reader can be seen as the device to be sampled. If tags are considered as the devices, sampled data will be readers *IDs*. At the opposite, if readers are considered as the devices, sampled data will be tags *IDs*. Although we will support both the previous cases in our language, in the following we refer only to the first case because the software architecture for pervasive systems is thought to manage tags in this way. To better define an *RFID* sampling semantics it is important to remark that different kinds of *RFID* technologies exist.

HF-RFID (High Frequency RFID) is a technology working on short ranges: a tag is sensed when it passes near a reader (few centimeters). In this case, a first idea to define the sampling is to assume the *ID* of the tag currently passing near the reader as the sampled value. Obviously this is not a good idea because there is a high probability that, when a sampling is required, no tag is passing.

We found two possible solutions to this problem. The first one requires the introduction of a cached value: when a sampling is required the returned value is the cached one, i.e. the *ID* of the last reader that sensed the tag. Note that with this approach the sampling operation can be seen again as a function call.

On the contrary, the second solution requires a radical change in the abstraction: in fact the interaction model between the device and the upper software layer becomes synchronous. The idea is to abstract the sampling operation as an event: when the tag is sensed by a reader, an event flag is raised and a parameter of this event reports the *ID* of the reader. Note that, with the described abstraction, the sampling operation is not invoked by the user: there is not a concept

of sampling rate and the execution of the next sampling operation is logically decided by the tag device.

In our language we will provide statements to support both sampling abstractions.

UHF-RFID (Ultra High Frequency RFID) is a technology working on long distances: the reader is able to detect the presence of a tag in a range of few meters. Since the reader to which a tag is connected changes slowly, in this case the sampled value can be abstracted by the current reader *ID*. Note that the sampling operation can return a null value if the tag is not in the range area of any reader in the system.

UWB-RFID (Ultra Wide Band RFID) is a fairly recent technology having a very long line-of-sight read range (dozens meters). These devices are active or semi-passive and the sampling operation semantics can be defined in a similar way to sensor nodes.

The last type of devices we consider in this summary are *GPS (General Positioning System)* sensors. They can be simply thought as a special kind of sensor node, having position as its unique attribute.

As we will better show in the next chapters, also a full wireless network can be abstracted as a single device. In this case the meaning of the sampling operation depends on the sampling semantics of the nodes composing the network.

Another issue related to physical devices deals with the contrast between the ideal vision of hardware at the language level and the real hardware behavior. In a *WSNs* query language time plays an important role: time values are used to specify sampling rates, to define queries execution length, to extract windows from streams, etc... Then, maintaining a timestamp shared by all nodes in the network is an important issue. This is not a trivial problem, but some algorithms can be found in the literature [26] to synchronize nodes' clock. Obviously, the result that can be obtained is not the sharing of an absolute time, but only a division of the time in fixed size intervals (called "*epochs*"). The shared timestamp is the number of the current epoch.

The computation performed by network nodes to manipulate sampled data requires a certain amount of time. In addition, the communication channels are not ideal and delays or packets loss can affect the transmission of data through the network. Therefore, latency can be found from the instant in which a record is generated by a query Q and the instant in which the same record is available for another query, using as input stream the output of Q .

The language should be able to hide this kind of problems as much as possible. To do this, a precise temporal semantics should be defined for all the operations provided by the language. As we will show in the following chapters, the concept of *native timestamp* can be introduced to formally specify a temporal semantics. We define the "*native timestamp*" of a sampled data the timestamp of the epoch in which the sampling was done. Then, we extend the concept to all records generated in the system, defining some rules to assign a native timestamp to a record obtained from a manipulation of other records. The idea of native timestamps is especially useful to give an exact semantics to the *pilot join* operation.

2.3 Functional characteristics of the language

The functional characteristics of the language are the features that precisely define what the system can and what the system cannot do. They identify the set of operations and activities that can be required by user queries.

We identified a minimum set of functionalities, which must be implemented even in the first implementation of the language:

- **Statements for the definition of sampling parameters (time, mode, etc.).** These clauses are very important since they allow the extraction of raw data from the network nodes. In fact obtaining high quality streams of raw data is essential because all other system functionalities work on them. The goal of these statements is to define when a node should or should not sample, the interval between two consecutive samplings and the condition under which sampled data must be held or deleted.
- **Statements for the definition of operations to generate query output manipulating raw data.** Also this set of clauses must be supported in a basic version of the language because, with the previous one, it forms the minimum kernel of instructions and clauses allowing queries definition. The goal of these statements is to provide classical *SQL* operators (filtering, aggregation, join, etc.) adapted for streaming systems.

We also identified a set of extra functionalities that are not strictly needed but, if implemented, can increase the system potential: statements for the definition of in-network data mining operations and statements for the definition of actuation commands. The latter functionalities are not implemented in the first release of the language and will be described in the last chapter of this document as future works.

2.4 Non functional characteristics of the language

The non functional characteristics of the language are the features that allow the definition of constraints on the offered services and of the system *QoS (Quality of Service)* [27] [28]. The main non functional requirements in *WSNs* are related to power management, but many other aspects can be considered, taking into account the wide heterogeneity of supported devices: different node types, different nodes latency and different available sensors.

Given the amount and the diversity of the possibly useful non functional constraints, it is not feasible to manage each possible constraint with a reserved clause. Thus, we think that a generalized abstract management could be a better solution: few clauses should be provided by the language to support all the non functional characteristics that can be considered now or discovered in the future.

The solution we present aims at providing a simple and general framework for policies management, allowing the user to express non functional requirements in a compact and abstract way.

Each physical device has a set of parameters that can be monitored and used to control query execution. For example, some sensor nodes can detect the power level reading the current battery voltage. The main idea in policy management is that this kind of sensed information can be used to take some decisions related to the quality of retrieved data rather than to the operations needed to produce query results. As an example, the power level can be used to decide which nodes of the network should execute the query.

Three main problems should be considered. The first one is that every device technology has a different set of parameters: for example the concept of power level cannot be applied to the *HF-RFID* technology because tags are passive devices powered by the *RFID* reader magnetic field.

The second problem is that the same parameter can be detected in different ways by different technologies: for example some devices can infer the power level starting from the number of executed operations, rather than measuring the battery voltage. Obviously these details should be hidden to users: they only want to express conditions like “execute the query only on devices having a high power level”.

The last problem is the different importance that a certain parameter can assume in different technologies. Consider still the battery level: it is very important in *WSNs* because charging or substituting the battery of a node is often difficult. The same information is still useful, but less critical, if related to a *PDA*, that can usually be easily recharged.

The abstract model we now propose as a solution will be clearer in the next chapter, when the architecture for pervasive systems will be presented. The idea is to abstract policies parameters as attributes of some logical object in the language: for a user there are no differences in requiring a power level or a temperature sample to a node. Every technology abstraction (logical object) should provide a mapping from its physical parameters to user-level policies. For example, the logical object of a sensor node will convert the battery voltage into a percentage indicating the current power state of the device. User queries will express conditions on that percentage.

Adopting the above solution, an important question has to be discussed: what should the query executor do if a policy-attribute that appears in a query is not managed by a technology abstraction? Note that this problem also exists when we consider standard attributes: for example, what should an executor do if a query requires a temperature sampling to a device that doesn't have any temperature sensor on board?

A possible, radical solution to that problem is excluding from the query execution all the devices that cannot provide a value for any of the attributes appearing in the query.

An alternative solution could be the definition of a well known set of policy attributes that should be implemented by all the logical objects. To maintain generality a default value for every well known attribute can be introduced so that, if an attribute is not supported by a device, the default value is used. For all other attributes (not present in the list) the default type value can be used when they are involved in selections, while devices should be excluded from the query execution when conditions refer to one or more non supported attributes.

A third solution, similar to the previous one, consists in using *NULL* values for all missing attributes, both in selections and in conditions, and then managing them with a **three valued logic** instead of a *Boolean logic*. This approach is the one used in standard *SQL* language [29] to deal with *NULL* values and can be summarized as follow:

- When an algebraic operator is applied to some operands, the produced result is *NULL* whenever at least one of the operands is *NULL*.
- *NULL* values are neither equal nor comparable with any other value (also if it is *NULL*, too). The comparison with a *NULL* always returns the *UNKNOWN* logical value.

- When a logical operator is applied to some operands, its logical value is computed using classical truth tables for three valued logic.
- The user can require that a condition must be (or not be) *TRUE*, *FALSE* or *UNKNOWN*. If this request is not explicitly specified, the default behavior is checking if the condition is *TRUE*.

We decided to adopt the latter solution because it is the more general and it maintains compatibility with the standard *SQL* approach. It is worth noticing that this feature of the *SQL* language is seldom used (and actually not known by common *SQL* users), but it is very useful in our language due to the above mentioned missing attributes issue.

Independently of the chosen way of implementing policy management, non functional clauses can be integrated in the declarative language to control some aspects of queries execution. For example, they can be used:

- *To decide if a node should participate to a query*
- *To set the current sample rate*
- *To set the rate used for sending data out of the node*
- *To retrieve information about network nodes*

3 Architecture for pervasive systems

In this chapter the architecture for pervasive systems, designed and implemented to support *PerLa* language, is explained: the concept of logical object is deeply explored; then, the main components of the architecture are presented and the ideas to provide a physical implementation of the logical object abstraction are described; finally, the solutions adopted to achieve the *Plug & Play* behavior are presented.

3.1 Logical object abstraction

The logical object abstraction is not a real software interface, but just a set of functionalities that allow to represent every kind of devices at a high abstraction level. A logical object representing a given device must show the list of information (data and metadata) that can be read or written and the type of sampling that can be performed on the device. Note that a logical object can be defined both to abstract a single node and a group of them: in the last case the final user controls all the nodes as they were a unique entity.

The main idea is that the formalization of language semantics has to be completely based on the logical object concept. We report in the following the main functional features of a logical object.

Retrieving or setting attributes. Attributes can be used to retrieve both the state of the node and the sampled data or to modify some parameters on the device. They can be classified in three categories, depending on the type of value they abstract:

- *Static attributes*: represent constant values describing the characteristics of the node (e.g.: device type, maximum sampling rate, etc). These values don't change during the logical object life cycle.
- *Probing dynamic attributes*: when these kinds of fields are read, the logical object must refer to the physical device in order to produce the requested value before returning it. Similarly, when a write operation is required (if supported), the logical object has to interact with the device to set the new attribute value. Probing attributes can be mapped on a real sensor (e.g.: temperature, pressure, etc), on a real actuator (e.g.: position of a stepper motor) or on a memory area (e.g.: information stored in a device *RAM*, *ROM*, flash, etc), but the logical object user don't perceive any difference.
- *Non probing dynamic attributes*: when these kinds of fields are read, the logical object returns a local cached value without dealing with the physical device. The cached value is periodically updated by the logical object. Writing operations are not supported on these attributes.

Firing notification events. Some operations defined in *PerLa Language* are activated after an event is raised. Thus, the logical object interface must be able to fire some events coming from

physical devices (*e.g.*: when an *RFID* reader senses an *RFID* tag or when a device identifies a certain pattern in the signal produced by an attached sensor).

Getting the list of supported attributes and events. A functionality should be provided to allow the query analyzer to discover if a logical object supports a certain attribute or event. The type of each attribute, both in terms of data type (integer, string, etc) and attribute type (static, probing or not probing) should also be provided.

The final user who submits *PerLa* queries is provided with a logical object view of the network: physical access to devices is completely hidden and the user writes queries thinking only about the set of available logical objects. This feature allows the language to be completely independent of the real middleware implementation. The main advantage of this approach is the possibility of developing and using the language without taking care of low level issues.

Consider a classical monitoring device having some sensors on board (*e.g.*: temperature, pressure, etc...): in this case a logical object can wrap the device, exposing an attribute for each physical measure that can be sensed. Probably, no events can be raised by this logical object and only time based sampling can be performed on it. A sampling operation is then abstracted as the reading of one or more attributes.

On the other hand, with the *RFID* technology, accordingly to the two approaches introduced in the previous subsection, logical objects can be defined in two different ways. In fact, a logical object should wrap the device and expose the sampled variable as an attribute. Thus, both logical objects wrapping an *RFID* reader and logical objects wrapping an *RFID* tag can exist: in the first case the exposed attribute is the *ID* of the last sensed tag, while in the second case the exposed attribute is the *ID* of the last reader that saw the wrapped tag.

Both approaches require the logical object to expose an event in order to signal the activation of a tag, allowing in this way the *event based sampling* to be performed. A sampling operation on such a logical object consists of reading the attribute when the event is raised. Note that the reading can also be performed without waiting for an event, obtaining in this way a *time based sampling*.

3.2 *PerLa* Middleware

PerLa middleware is a collection of software layers, each of them managing a different aspect of the network or of the language. Figure 4 shows the overall *PerLa* system architecture and the main blocks composing the middleware. The two interfaces that delimit the area covered by the middleware are highlighted. The upper one is simple and well defined; it represents the final user access point to the system and it must expose two main functionalities: receiving *PerLa* language queries and returning the user the streams produced by the queries execution.

The lower interface is, instead, more critical and it is the main point characterizing our middleware with respect to other similar projects. In fact, we tried to push this interface as near to the devices as possible. Existing systems, like *SIEMENS SWORD* or *Global Sensor Network*, define an interface between middleware and physical nodes that is simple and clear, but difficult to implement on small devices (*e.g.*: an *HTTP* interface can be too hard to implement if the physical device is a small *C-based DsPic* connected to the middleware via an industrial

bus). Our goal is to allow also this kind of devices to be attached to our infrastructure, without requiring an excessive developing effort. For this reason, the bounds of the lower interface cannot be easily and accurately delimited in terms of a set of *Java* functions or network protocols; in fact, a set of low level routines are provided to the developer as a *C-Java* library, that has to be extended with device specific sampling routines in order to support a new technology. For this reason, a dashed line, instead of a solid one, is used in Figure 4 to represent the lower interface.

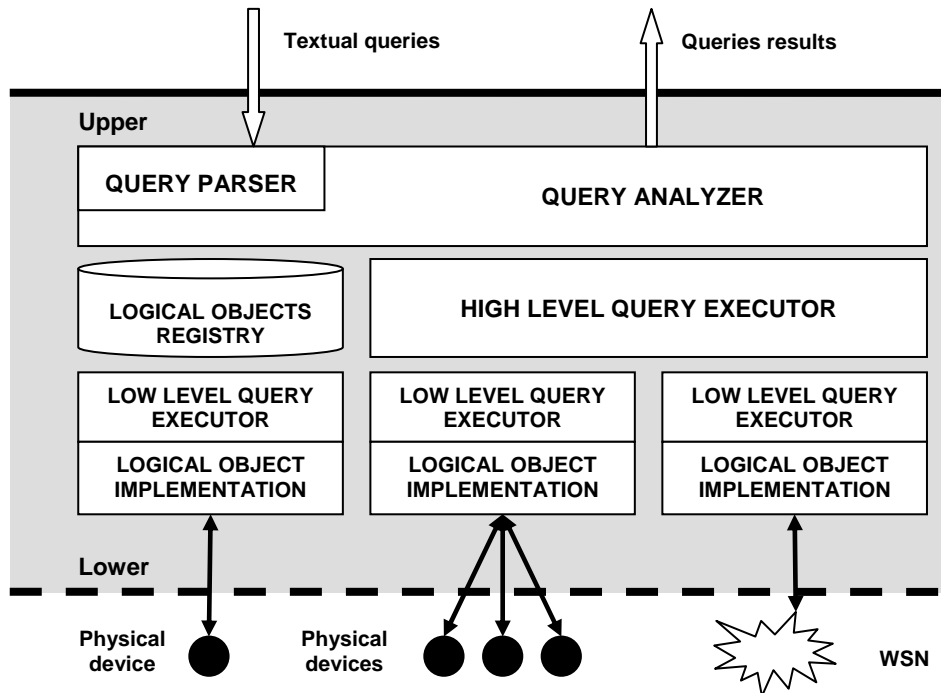


Figure 4: TinyDB event used to terminate a query

In the following of this section, the main blocks composing the middleware are briefly introduced, while the key concepts are presented in details in the next sections.

The **Query analyzer** and **Query Parser** are the highest level components of the middleware and they directly interface with the user. They receive textual queries as input to verify their syntax, perform semantic validation and, finally, transform the queries in a suitable format for distribution and execution. As better explained in Section 4, *PerLa Language* is composed of two classes of queries: low level queries (*LLQ*) allow to define the behavior of the devices, while the high level ones (*HLQ*) allow to manipulate the streams generated by *LLQs*.

The **Logical Object implementation** and the **LLQ Executor** aim at managing the interaction with physical devices. The logical objects implementation, the communications between the physical nodes and the corresponding logical objects and the execution of *LLQs* are the main tasks delegated to this set of components.

The **HLQ Executor** is the middleware component charged to execute *HLQs* and it is just a *DSMS* engine. In fact, this kind of queries are independent of physical nodes and their abstractions.

The **Logical Object Registry** has the role of maintaining the list of currently instantiated logical objects. When the language parser receives a *LLQ*, the set of logical objects that should execute it has to be identified. This is done by sending a request to the registry, in order to discover which devices currently in the system satisfy the condition specified in the *LLQ*. To produce a response to this request, the registry must maintain a list of all the active logical objects and it must also be able to contact each of them, in order to retrieve the current value of the attributes needed to evaluate the specified condition.

3.3 Functionality Proxy Component

An *FPC* is a *Java* object whose role is to provide a physical implementation of a logical object abstraction: it can be used by other pieces of *Java* code to interact with the device itself. A hardware device can host an *FPC* only if it complies with two main constraints: first it must be able to run a *Java Virtual Machine* and, second, it must be connected to the *PerLa* middleware through a *TCP/IP* network.

Although some physical devices are powerful enough to directly host their *FPCs*, many nodes don't satisfy the above conditions: they cannot run a *Java Virtual Machine*, they are not provided with a *TCP/IP* interface or, sometimes, they don't have an operating system at all. In all this cases, the *Java* object wrapping the device is instantiated on the nearest machine that is *Java* and *TCP/IP* enabled. The communications between the *FPC* and the physical device are fully managed by the middleware: in fact, a channel manager component provides a virtual channel that abstracts the peculiarities of the adopted physical channel.

3.3.1 Channel virtualization

Channel virtualization is the process that allows to establish a bidirectional point-to-point channel between an *FPC* and the middleware code directly running on the device. Thus, the channel manager component has been implemented in two versions: the *Java* version is deployed on the *FPC* side, while the *C* version is deployed on the physical nodes.

On the *Java* side, the channel is abstracted as a stream on which only read and write operations can be performed. The middleware is then charged to convert and transmit all the data pushed on the stream, using a protocol shared with the device. Communications use a physical link that connects the device with the *Java* machine; thus, the channel manager must respect the constraints imposed by that link. Consider, as an example, a radio channel that operates in duty cycle mode to reduce power consumption: in this case the channel manager must support buffering, in order to mask the real behavior of the physical link.

Consider, as a second example, a device that is connected to the nearest *Java* machine through two cascaded links (for instance, a slave device can be attached to a master *CAN-BUS* board, which, in turn, communicates with a *Java* machine through a radio link). In this case, the channel manager should also take into account the routing required to transport the messages from a *Virtual Channel* side to the other one: more generally, the channel manager must deal with heterogeneous physical networks.

Another important feature of the channel virtualization component is the ability of automatically setting up the virtual channel when a new device joins the system: only some configuration files are required to be installed on the physical device, in order to inform the middleware code about the supported protocols. Both *Java* and *C* implementations of the channel manager now support the most used bus interfaces (*i.e.* serial bus, *TCP/IP* sockets, *CAN-BUS*), but a developer can integrate the middleware with new modules in order to support other protocols.

3.3.2 Channel multiplexing

In the previous scenario we assumed that a physical channel always exists between any *device-FPC* couple. This is not a realistic assumption in a real environment, because it often happens that a certain number of nodes are cabled on a shared broadcast bus (*e.g.*: *CAN-bus*), that must be used to communicate with the nearest Java machine. To manage this situation we introduced a channel multiplexing facility. We called *Adapter Server* the component charged to perform the multiplexing and the demultiplexing operations on the *FPC* side, while we called *Adapter Client* the symmetric component on the other side.

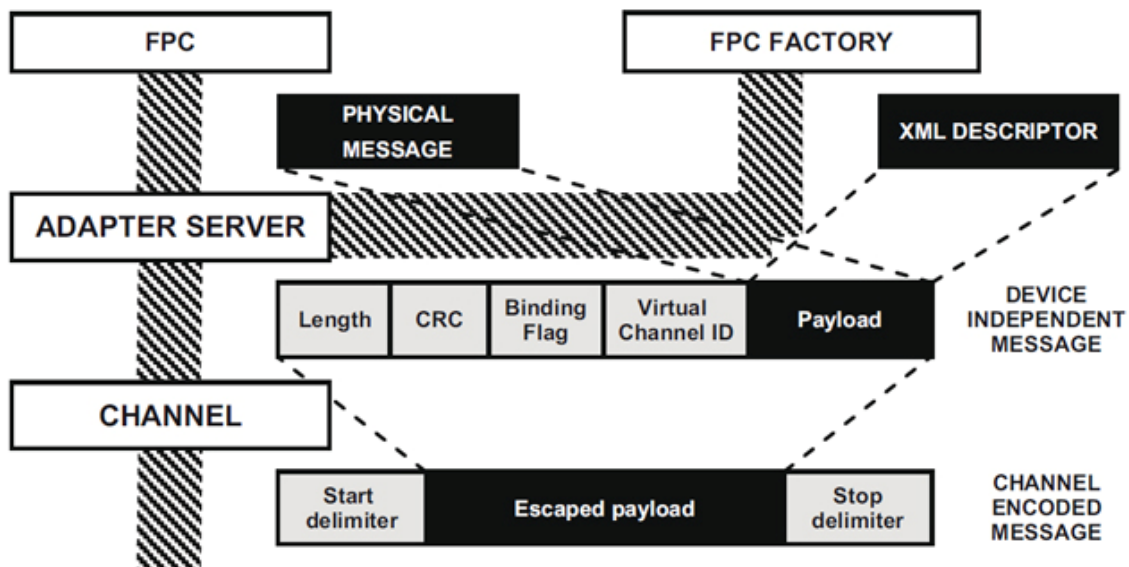


Figure 5: FPC-Device communication protocol

The set of exchanged messages between the *FPC* and the physical device is now presented to complete the description of the lowest middleware layers. Figure 5 shows the format of the messages used at different levels of the protocol stack. The data physically sent on the link connecting the *FPC* and the device is a sequence of escaped packets, bounded by a *Start* and a *Stop* delimiters. The format of these messages is fixed by the middleware and a complete *Channel Manager* implementation is provided both for the Java side and the C side. A full duplex point to point packet oriented communication is provided by these components.

The payload of each *Channel Encoded Message* is one *Device Independent Message*, the structure of which is defined by the middleware. This kind of packets contains the information used by the adapter component to perform multiplexing and demultiplexing. In particular, a

binding flag is used to distinguish between the type of carried payload, that can be either an *XML* device self-description or a *Physical Message*. When the *Adapter Server* receives a message, the binding flag is checked: if true, the *XML* descriptor contained in the payload is delivered to the *FPC Factory*; if false, the *Physical Message* contained in the payload is delivered to the correct *FPC* instance. The above described architecture allows to set up a virtual point-to-point connection between a *Low Level Driver* and the correspondent *FPC*. The messages exchanged on this channel are called *Physical Messages* and their structure is not imposed by the middleware. On the contrary, they are formatted accordingly to the information contained in the self-description file of the device. This choice has the goal of making the implementation of the *Low Level Driver* easier. In fact, if the message structure is appropriately defined to match the data structures used by the native *APIs* (provided by the device vendor), the effort required to the developer is certainly reduced.

3.3.3 FPC functionalities

Consider a *C* based physical device equipped with a temperature sensor, connected to a *Java* machine through an *RS232* bus. A functionality that must certainly be exposed by the *FPC* interface is a temperature attribute, that causes the execution of a physical sampling operation when it is read. A final user should be able to require the value of the temperature attribute and the *FPC* should act as a proxy, by forwarding the request to the middleware code directly running on the device. The technical details of the sampling operation are completely masked to the user: they are entirely managed by the *FPC* and by the middleware code running on the device.

When the value of the temperature attribute is required, the *FPC* pushes a message on the virtual channel in order to require the execution of the sampling operation. The middleware code on the device recognizes the message and calls the proper sampling routine, that is the only *C* code the device manufacturer has to write to grant the integration with *PerLa* middleware. The portion of *FPC Java* code that is dependent on the device features is automatically generated by a middleware component (the *FPC Factory*), starting from a declarative self description file distributed with the device.

The *FPC*, as described till now, is just a simple proxy to the functionalities of a physical device. However, it is more complex and it contains the logic needed to best exploit the device capabilities. Suppose now that, in the previous example, the temperature has to be sampled once every 30 seconds: if the device is equipped with a timer and it is able to autonomously schedule the execution of the sampling routine every 30 seconds, an explicit sampling request sent by the *FPC* to the device every 30 seconds is certainly a waste of network resources. Similarly, if only an aggregated value of the samples collected during the last hour is needed every hour, sending all the gathered data to the *FPC* is a waste again if the device can autonomously perform the required aggregation.

The variables considered to decide if a computation can be executed directly on the device (rather than executing it on the *FPC*) depends on the type of the required operator. The evaluation of algebraic and distributive operators requires a limited amount of memory and the time performances are the only factor influencing the choice. Instead, the ability of a device to evaluate an holistic operator is related to the amount of available memory.

A goal of the *FPC* is, then, to find the best trade-off between the network load on the *Virtual Channel* and the computation balancing between the *FPC* and the physical device: only

requests that can be really processed directly by the device must be injected on the *Virtual Channel*. The information needed to evaluate this trade-off are obtained from the device self configuration file.

Another critical aspect an *FPC* has to deal with is related to the management of multiple and concurrent requests, coming from different threads that operate on the same device. In fact, an *FPC* must be able to detect if all the pending requests can be processed by the device or if some of them must be aborted. Moreover, the *FPC* can translate two close requests, submitted by different threads, into a single physical operation whenever the time interval between the two requests is less than the time needed to physically perform the sampling activity. In this cases, the same sampled datum is used to satisfy both requests.

3.3.4 FPC architecture

Figure 6 provides an overview of the *Functionality Proxy Component* and it represents the interactions between this component and the software running on the abstract device. It also distinguishes the software modules belonging to the middleware from the software modules written by the developer who is integrating the device with *PerLa* middleware.

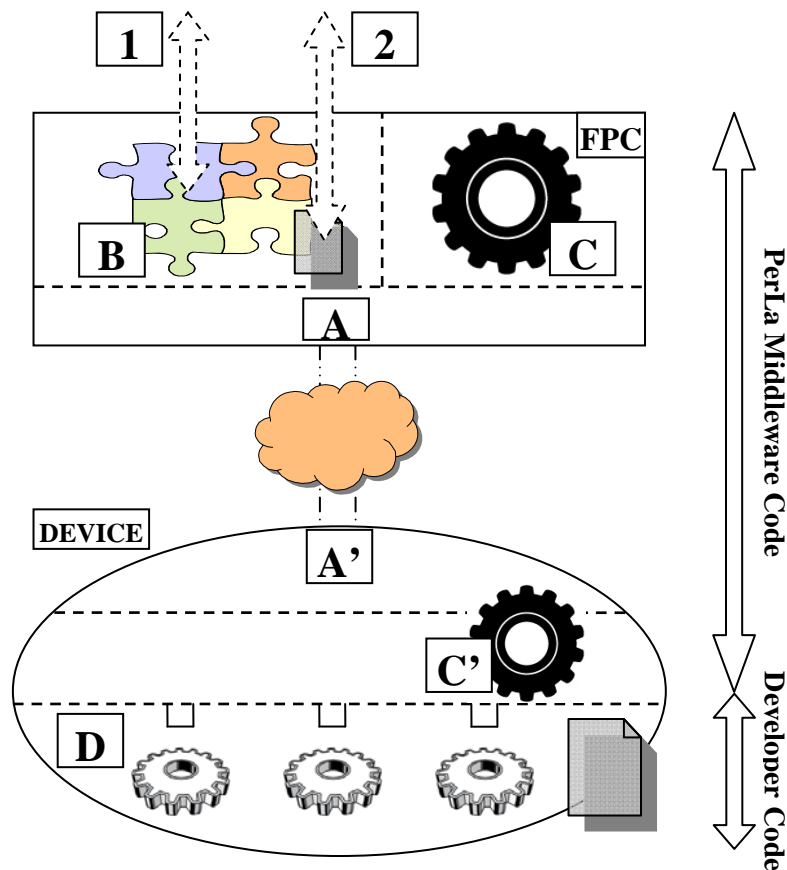


Figure 6: FPC architecture overview

The upper interface of an *FPC* is composed of two classes of methods: the first one (1) allows to retrieve the metadata extracted from the device self configuration file, while the second one

(2) allows to require the execution of a sampling operation. Note that, exposing a simple property for each sensor is not enough to support all the functionalities described before. In fact, more details have to be specified for each submitted request: the sampling operation is abstracted through a method that takes as parameters the set of sensors to be sampled, the number of required samples, the sampling rate and the filters and aggregations to be applied. As shown in Figure 6, the *FPC* core is composed of three blocks. The first one (A) manages the communications over the *Virtual Channel*, by encoding and decoding the messages accordingly with a protocol defined by the middleware. The second block (B) is charged to plan the activities needed to satisfy all the pending request, taking into account physical devices abilities (that are declared in the self description file). Finally, the last block (C) is deputed to perform the computations over sampled data: this subcomponent is charged to perform the operations not supported by the physical devices. In fact, the middleware code deployed on the nodes contains a *C* implementation of the same subcomponent (C'), that can be used (if the device is powerful enough) to filter and aggregate sampled data before sending them to the *FPC*.

The code that must be written by the developer in order to allow a device to be integrated with *PerLa* middleware is marked as (D) in Figure 6: it includes the declarative self description file and the low level sampling routines.

3.4 FPC Factory and Registry

As presented above, an *FPC* allows a simple interaction with a physical device, hiding the communication details and masking low level programming issues. To reach a real *Plug & Play* behavior, the middleware should be able to create *FPCs* at runtime, to instantiate them on a *Java* and *TCP/IP* enabled machine and to set up the *Virtual Channel*. A middleware component, called *FPC Factory*, has been designed to achieve this goal. An instance of this object is deployed on each machine deputed to host the *FPCs*: given the information contained in a device self description file, the *FPC* code is automatically generated. This process is feasible because the *FPC* can be implemented by merging a certain number of modules, each of them managing a different aspect of the interaction with the physical node. The device description is used to decide which are the right modules to be selected from a library in order to provide a proper device abstraction.

To provide a full *Plug & Play* support, a physical device that is joining the system must be able to send its self description to the nearest factory and to start communicating with the generated *FPC* without any human interaction. A special *Virtual Channel* is reserved on each physical channel to communicate with the *FPC Factory*. Thus, the new device sends its self description on the reserved channel and, if no error occurs, the binding of the generated *FPC* with a new *Virtual Channel* is notified. All the subsequent communications between the physical device and the *FPC* are performed on the new *Virtual Channel*.

Another important middleware component, strictly related to the *FPC Factory*, is the *FPC Registry*: it is basically a *Main Memory Database* in which a reference to any existing *FPC* is maintained. Its main goal is to provide support for the evaluation of the language clause used to define the set of devices involved in the query (*EXECUTE IF*). After a new *FPC* is generated, the factory registers it: the device metadata (supported attributes and events, maximum sampling frequencies, etc...), the values of static attributes and the remote reference to the *FPC*

object are stored in the *FPC Registry*. In a real pervasive system, a physical device is often connected to different nodes that can host its *FPC* and more than one physical channel could exist between these nodes and the device. Moreover, some of the *Java* enabled nodes and some of the physical channels can be temporary unavailable due to power management policies or mobility problems. Thus, a mechanism to allow an already initialized device to reconnect to the middleware using a different physical channel has been implemented. Basically, the device executes the initialization protocol again, but the *FPC Factory* recognizes that an *FPC* for that device already exists in the middleware. This can be done simply by performing a search in the registry. To restore the communication between the device and its *FPC*, the found *FPC* is reinstated and a new *Virtual Channel* is established.

3.5 Device self description file

In the previous sections we referred to the device self description file many times. This section provides an overview on the contents of this file, that must be created for each new device that should be integrated with *PerLa* middleware. All the relevant information, needed to dynamically build the *FPC* and to establish a data connection between the physical device and the generated *FPC*, must be included in the file. More specifically, the developer has to specify:

the details needed to contact the available *FPC Factories* during the set up phase; they include the initialization parameters of the physical channels, required to establish a communication with the *FPC Factory*;

The list of available sensors and the physical measures that can be sampled by the device. For each attribute, the name, the data type at language level, the data type at physical level (e.g.: the encoding and the length of the value generated by the *ADC*), the conversion function and the constraints on the sampling rate should be specified;

The technical features of the device, such as the number and the size of available buffers, the number of available timers, etc... ;

The list of middleware functionalities, implemented in the *C* library, that can be executed directly on the device (e.g. the evaluable conditions, the set of supported aggregation functions, etc...).

Some of the information contained in the self description file (1) are used by the device itself during the setup phase. All other data are used by the *FPC*; thus they must be sent to the *FPC Factory* during the *Plug & Play* initialization phase.

Two examples are reported in the following, in order to show how the middleware can exploit all these information to optimize the queries execution. Suppose that the *EXECUTE IF* condition of a submitted query requires to execute the query itself only on those devices that are equipped with a temperature sensor: the *FPC Registry* detects if a given node complies with these constraints based on the list of attributes stated in the device self description file. Similarly, if the submitted query requires the computation of a given aggregation on a set of sampled values, the *FPC* can determine if the physical device can perform the needed aggregation. In particular, the execution is performed by the *FPC* if the software module needed to compute the aggregation is not available on the device, or if the estimated amount of memory exceeds the size of the device buffers.

4 Language design

In this chapter *PerLa* language is introduced: the types of supported queries are described and the logical process of queries execution is explained. Firstly, we analyze how queries are structured from the user point of view and how they will be decomposed when executed. Then, we introduce the idea of high and low level queries and of stream and snapshot tables and, using these concepts, we show how a user query can be seen as a graph. We also explain the meaning of abstraction level and we provide a formal classification of queries and sub-queries that can be conceptually found in the system. Finally, starting from a query graph example, we show how a user query is decomposed, sent to logical objects and executed.

The concepts presented in this chapter will be useful to better understand the language semantics, that is formally described in Chapter 5.

4.1 Query analysis

Queries expressed with our language are quite complex, because they should describe all data manipulations to transform a set of sensor sampling operations to an output data stream. Thus, we start the presentation of language design showing how user queries can be decomposed in simpler components. In the following paragraphs we will show in detail what these components are and the possible relationships among them. Then, in the next chapter, we will present how each component can be specified using language statements.

The query decomposition can be considered from two different and orthogonal points of view that of the user and that of system.

The user can perform a *syntactical decomposition*: the same query can be written as a single complex statement or as a set of simpler statements. For example, a simple query taking data from a sensor and inserting it in an output stream can be written either with a single statement or with a creation statement (to define the output stream) followed by an insertion one (to control the sampling operation). Allowing the user to define simple queries and combine them to build the final *SQL* statement is similar to introduce the concept of function in a programming language syntax.

As we shall see in the next chapter, not all the queries can be written as a single statement, but some choices in syntactical decomposition are left to users. Note that this decision is not very important at this point of the language definition, because it implies only modifications in the language syntax and in the parsing routine, but not in the query execution engine.

From the *system point of view* a user query can be decomposed in some blocks, each one describing a data structure (stream or snapshot) or the behavior of a sub-query (high or low level query). In the following of this chapter we present in details these blocks, showing their features and how they can be combined to build a valid user query.

The language syntax presented in this report is designed to allow user decomposition to be very similar to system decomposition: for example, if a user query contains a high level and a low level query, the user has to write two different statements.

4.2 Query decomposition (from the system point of view)

The building blocks composing a query can be classified in two classes: *table definition* and *data management*. The first type of block is used to define intermediate and final data structures that must be computed to produce query results. The second type of block is used to describe the set of operations needed to retrieve data from sensors or from existing data structures and to produce records for other data structures. Moreover, this second block also provides the statements to inject data into a logical object: this can be useful both to control physical actuators and to set some parameters or thresholds on the software executed on the device.

Two types of table are supported: *streams* and *snapshots*.

The *stream table* is the most common type of data structure and it is an unbounded table. Each record has a set of user-defined fields and a native timestamp field. The following operations are supported:

- **Insertion.** A new record can be inserted into the stream by a running query. The execution of this operation generates an insertion event that can be detected and used by other sub-queries.
- **Reading.** A window can be extracted from the stream by a running query. That window is defined by a timestamp value, that identifies the most recent record, and by a size, that can be expressed either in terms of a number of records or a time interval.

The *snapshot table* is a buffer characterized by a time value T that indirectly defines the buffer size. In fact, a snapshot taken a time t must contain all the records that were generated in the time window $(t - T, t)$. The behavior of this data structure can be understood thinking that there are two memory areas: a local buffer and an output buffer. The time is divided into intervals of length T (window). All the new records inserted in the snapshot are stored in the local buffer. When the window changes, the content of the local buffer is copied to the output buffer and then the local buffer is cleared. When the snapshot is read, the current content of the window (that is the set of records received during the previous interval) is returned.

Three types of blocks relative to data management exist: *low level queries*, *high level queries* and *actuation queries*.

A *low level query* is used to define the behavior of a single device (or a group of devices abstracted by a single logical object). The main role of low level statements is to precisely define the sampling operations, but they also allow the application of some *SQL* operators on sampled data. An object executing a low level query has to maintain a local buffer and to perform the following activities:

- **Sampling data** by reading some logical object attributes and inserting the read values into the local buffer. This operation has to be executed when required by the query: periodically or when an event happens.
- **Performing SQL operations** (selection, aggregation, filtering, grouping, etc) on the current content of the local buffer and inserting obtained records into the stream or the snapshot that was indicated as query output. This activity should be executed when required by the query (if the output is a stream) or periodically with period T (if the output is a snapshot with size T).

The local buffer is conceptually unbounded and its size increases indefinitely. Practically, the executor should be able to delete old records that are no longer needed to perform required *SQL* operations.

It is worth noticing that *all the operations that can be executed at low level are relative to data extracted from a single logical object*. The goals of these operations is to discard bad values and, optionally, to aggregate a group of sampled values before sending them to high level queries. Two different low level queries can insert their results in the same stream or snapshot, performing in this way a *SQL* union operation.

The second type of data management blocks is composed of *high level queries*. They use one or more streams (generated by low level queries or other high level queries) as input, they perform *SQL* operations on windows extracted from input streams and insert the generated records in an output data structure. The activation of a high level query can be specified either in terms of a time period or in terms of an event (insertion of a record into a stream).

The last type of data management blocks provides support for actuation operations. An *actuation query* is a statement that allows to set the value of a logical object attribute. Similarly to low level queries, a clause to define the set of logical objects involved in the query execution is provided. Each device is then charged to independently perform the required actuation. The effect of an actuation query can widely vary dependently on the type of logical object attribute that is set: a physical measure can be modified, the content of a memory cell can be written, a parameter of an algorithm running on the device can be changed, etc..

4.3 Query graph

In this paragraph we introduce a graphical notation to represent the concepts presented above; then, using this notation, we show how a user query can be seen as a graph, whose nodes are streams, snapshots, low level, high level and actuation queries. Finally, we list all the constraints that a query graph should comply to represent a valid query.

As shown in Figure 7, we use a diamond to indicate a low level query (*LLQ*), a circle to indicate a high level query (*HLQ*) and a triangle to indicate an actuation query (*AQ*). Streams and snapshots are represented with rectangles having a single and a double border respectively. An exiting arrow can be drawn to indicate that a data structure (stream or snapshot) is an output of the user query. The star is used to indicate a generic set of logical objects that can be sampled by a low level query or modified by an actuation query. Arrows are used to show a

dataflow from a query to a data structure or vice versa. Then, a dashed oriented line is used to indicate a *pilot join* operation.

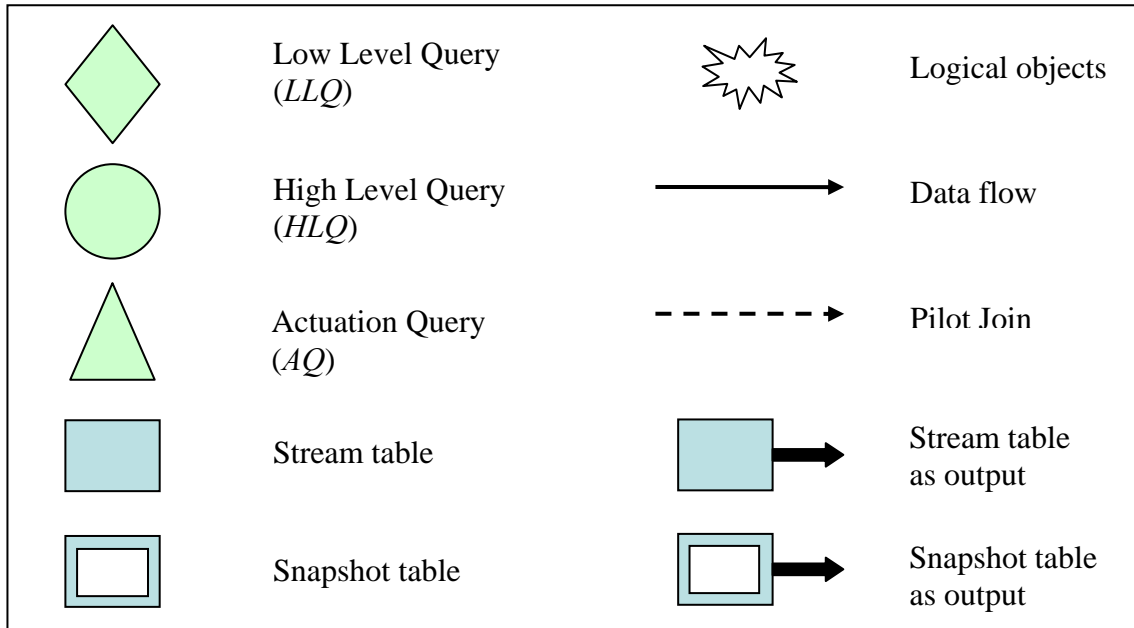


Figure 7: List of symbols used in query graphs

As an example, consider Figure 8; it is a graphical representation of a user query composed of two low level and two high level queries. *LLQ1* is used to sample a set of logical objects and to produce a stream. This stream is used as the source of both *HLQ1* and *HLQ2*. *HLQ1* generates a snapshot that is used to drive the *pilot join* of *LLQ2*: the list of logical objects that will execute *LLQ2* will be periodically updated looking at the snapshot content. Finally, *HLQ2* takes two streams in input and generates the output stream.

Considering again the mobile platforms example introduced in the Section 2.1, we now show that the graph in Figure 8 can represent the following query: “Execute a temperature sampling on the nodes mounted over the platform that is located nearest to a given point *P*. Then, return the sensed temperature and the location where the sample was taken.”

LLQ1 is the query that periodically (with period *T*) samples platforms position; *HLQ1* is activated every *T* instants to find the platform nearest to the point *P*. The *ID* of the base station mounted over that platform is then inserted in the snapshot. *LLQ2* is activated on all the logical objects abstracting temperature sensors and currently connected to the base station with the *ID* contained into the snapshot. Finally, *HLQ2* has the role of adding the platform location to the sampled temperatures using a join operation. Note that the sampling rate of *LLQ2* should be greater than the rate used to update the snapshot.

In Figure 9 the list of allowed connections in the query graph is shown. Low level queries can only have a set of logical objects as input and a single stream or snapshot as output. They can contain a *pilot join* driven by one stream table (*event based pilot join*) or one or more snapshot tables (*condition based pilot join*). High level queries can have one or more streams as input, and a single stream or snapshot as output.

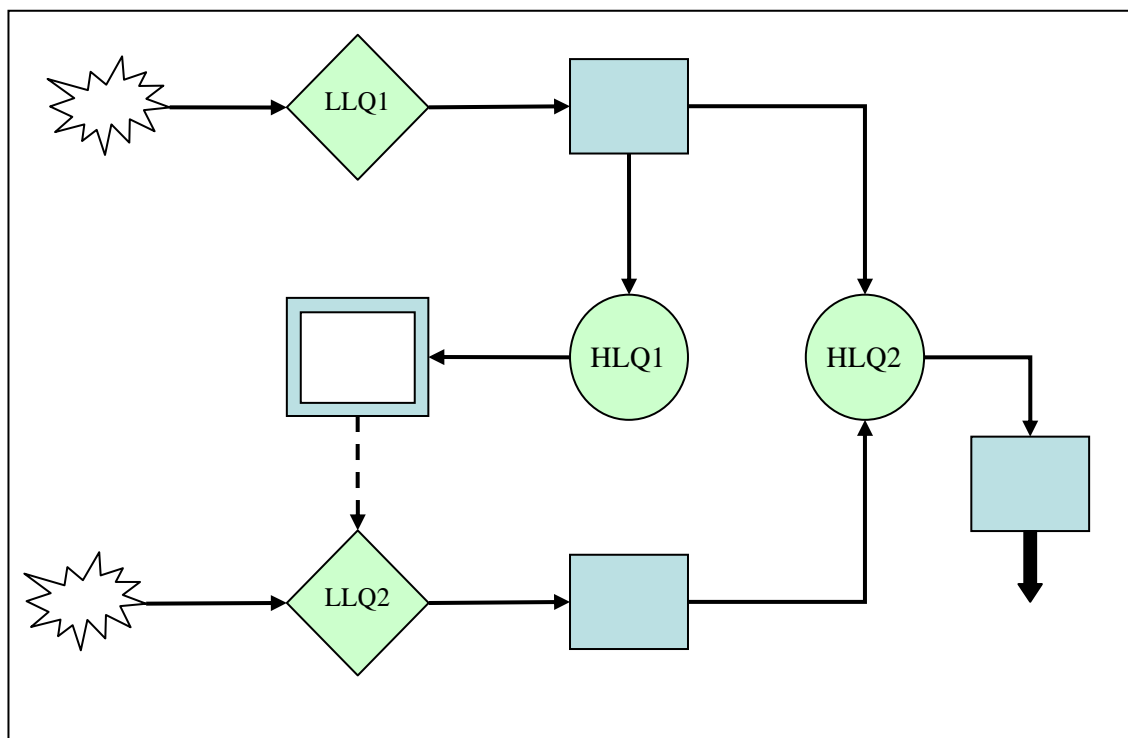


Figure 8: Example of query graph

Even if not shown from the figure, two important rules must be respected. The graph, having data structures and queries as nodes and data flow arrows as edges, must be acyclic (*DAG*). Loops can be present in the graph only considering also *pilot join* arrows as edges: in this case each loop must contain at least a dashed edge.

The second rule forces each data structure to be the output of a single high level query or of a set of low level queries. A query is not valid if two different high level queries (or a low level and a high level queries) both insert their results in the same data structure.

The query graph expressivity can be extended adding some labels to the graph edges, as shown in Figure 9 and Figure 10. These labels can be used to specify:

- **The sampling type** (event based or time based) of low level queries
- **The query activation type** (event based or time based) of low and high level queries
-

The semantics of the different sampling and activation types will be clearer when the complete language grammar will be presented and explained in Chapter 5.

We now consider the issues related to timestamp management. We shall use the graph representation of a query to describe how native timestamps are calculated and propagated. Low level queries are the first elements of the graph that generate native timestamp values. When activated, these queries compute some *SQL* operations on data contained in their local buffer and insert the obtained records in the output data structure. They set the native timestamp field of all the generated records equal to the current timestamp, i.e. the timestamp in which the query was activated.

Instead, high level queries use the following rules to attach a native timestamp value to the produced records. Each record is timestamped with:

- The native timestamp of the record that raised the event, if the query is event based
- The activation timestamp, if the query is activated periodically
-

To better explain how timestamps are managed by high level queries we now introduce two simple examples (Figure 11).

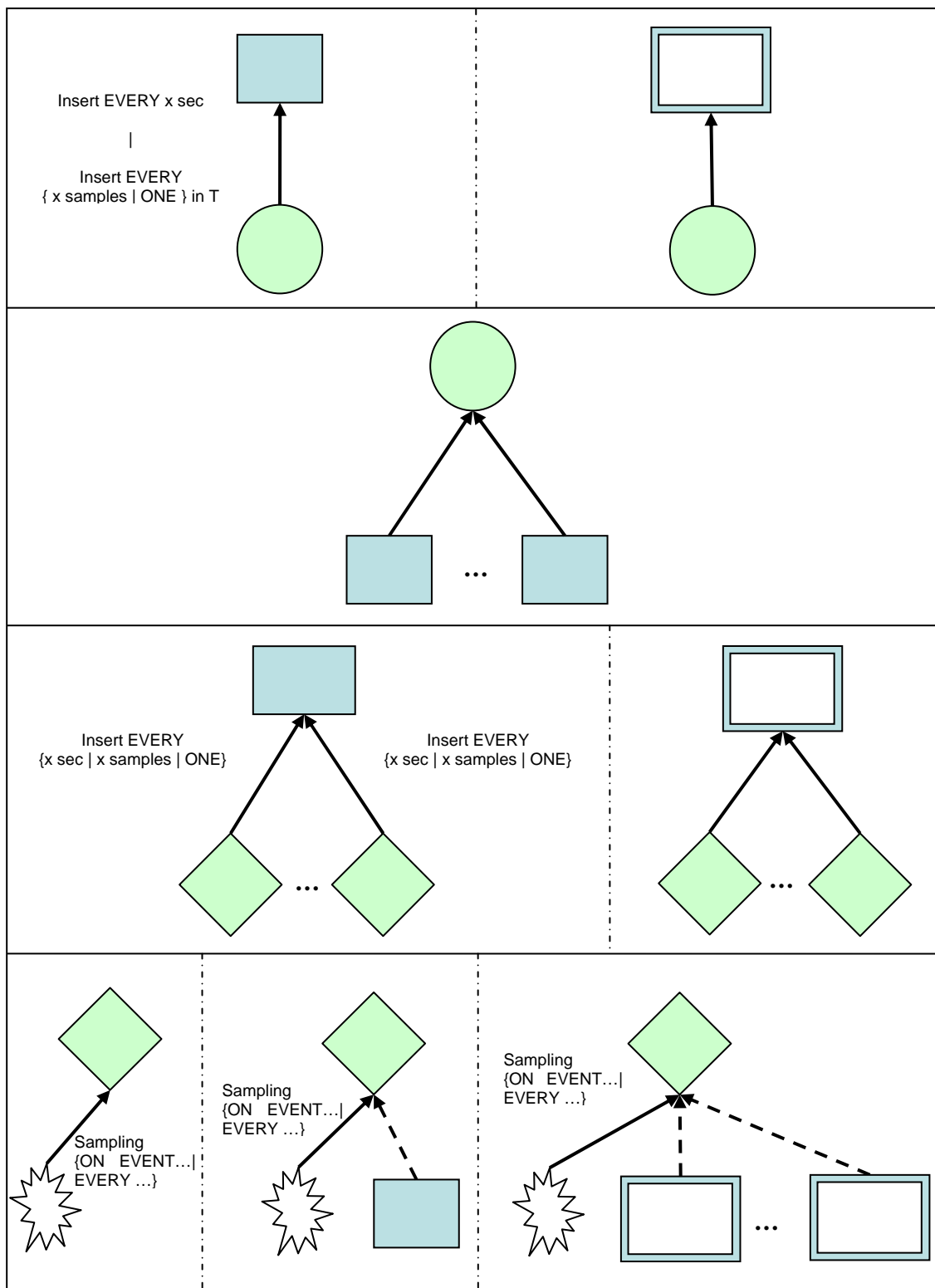


Figure 9: List of allowed connections in query graphs - A

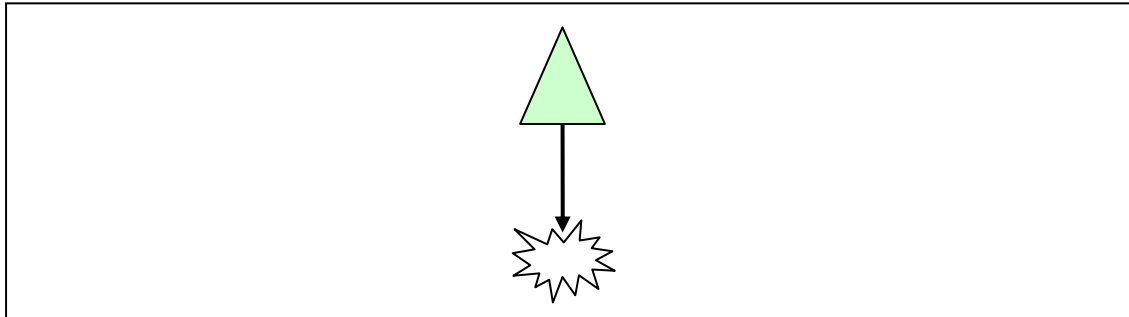


Figure 10: List of allowed connections in query graphs - B

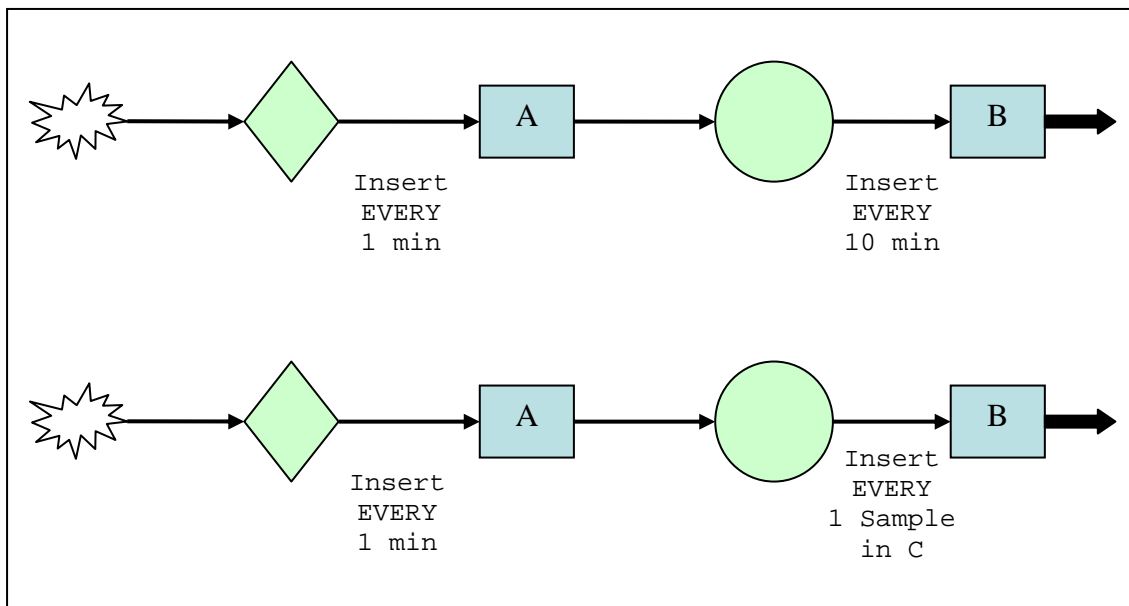


Figure 11: Query graphs and timestamps

The first query graph is composed of a low level query, activated every minute, and of a high level query, activated every *10 minutes*. Suppose that the high level query takes the last ten values produced by the low level query to calculate an average. If this query is activated exactly after ten minutes, it should work on the records having a native timestamp in the interval $(1 \text{ min}, 10 \text{ min})$. But probably, due to network latencies, an offset exists such that, at the time 10 min , only records produced before $(10 \text{ min} - \text{offset})$ are ready to be used.

To give a clear semantics to the query execution, the system must estimate the maximum value of that offset. Then, the query is activated at the time $(10 \text{ min} + \text{offset})$ and takes into account only the records having a native timestamp in the interval $(1 \text{ min}, 10 \text{ min})$. In this case, the native timestamp appended to the generated records is set equal to 10 min .

Note that if a record comes with a delay greater than the offset, the high level query is able to identify this anomalous latency and to immediately discard the record.

The second example shown in Figure 11 is similar to the previous one, but now the high level query is activated by an event. Suppose that a record is inserted in the stream used as events source (C) and that it is timestamped with the value 15 min . This event will be sensed by the high level query at a time greater than 15 min due to network latencies. The idea is that this query activation must work on data having a native timestamp less than or equal to 15 min . The generated records will be timestamped with the value 15 min , too. Note that, to obtain this result, the real query execution can be delayed by an offset due to the same reasons described for the previous example.

We defined some rules to assign a native timestamp to all records in the system. We also showed that a query must be executed after a certain delay from the ideal activation instant. This delay should be forecasted by the system and it is dependent on the position of the query in the graph.

This delay has also an impact on the *pilot join* operation semantics. In fact, it is clear that if a query inserts a record timestamped with T in a snapshot, the logical objects matching that record will start executing the query at a timestamp greater than T . This is the reason for which *the query generating the snapshot should be very slow with respect to the query containing the pilot join operation.*

Actuation statements are not continuous queries and, when submitted to the system, they are executed, one shot, as soon as possible. Thus, the definition of a temporal semantics involving the concept of native timestamp is not needed for this kind of queries.

4.4 Execution of a user query on the architecture for pervasive systems

In this section we explain how a user query is decomposed in order to be executed on the pervasive system. We analyze the architecture components that receive the obtained sub-queries and we show how the results are generated and sent to the user.

Note that with the term *sub-query* we refer to a command or a set of commands that can be executed by an object of the architecture and that was obtained from the user submitted query. The way in which a language implementation represents these sub-queries is not important for the purposes of this section: they can be a set of statements expressed in a *SQL* like language or simply procedural programs generated by the query analyzer.

The query decomposition, initially operated by the analyzer to extract the graph from the user query, is not the only action that is executed. In fact the obtained low level queries can refer to some abstract attributes that should be still translated into concrete attributes, as described in Chapter 2. So there are mainly five types of commands (or sub-queries) that can be found in the system:

- *Low level, concrete sub-queries*
- *Low level, abstract sub-queries*
- *High level sub-queries*
- *Actuation concrete sub-queries*
- *Actuation abstract sub-queries*

Note that the concept of abstract/concrete cannot be applied to high level sub-queries because they don't deal with sampling or actuation operations.

We now consider a simple example to show the flow of commands and responses that is established during a query execution. The query graph used for the example is represented in Figure 12: there are two low level queries that insert data into a stream, used as input for a high level query.

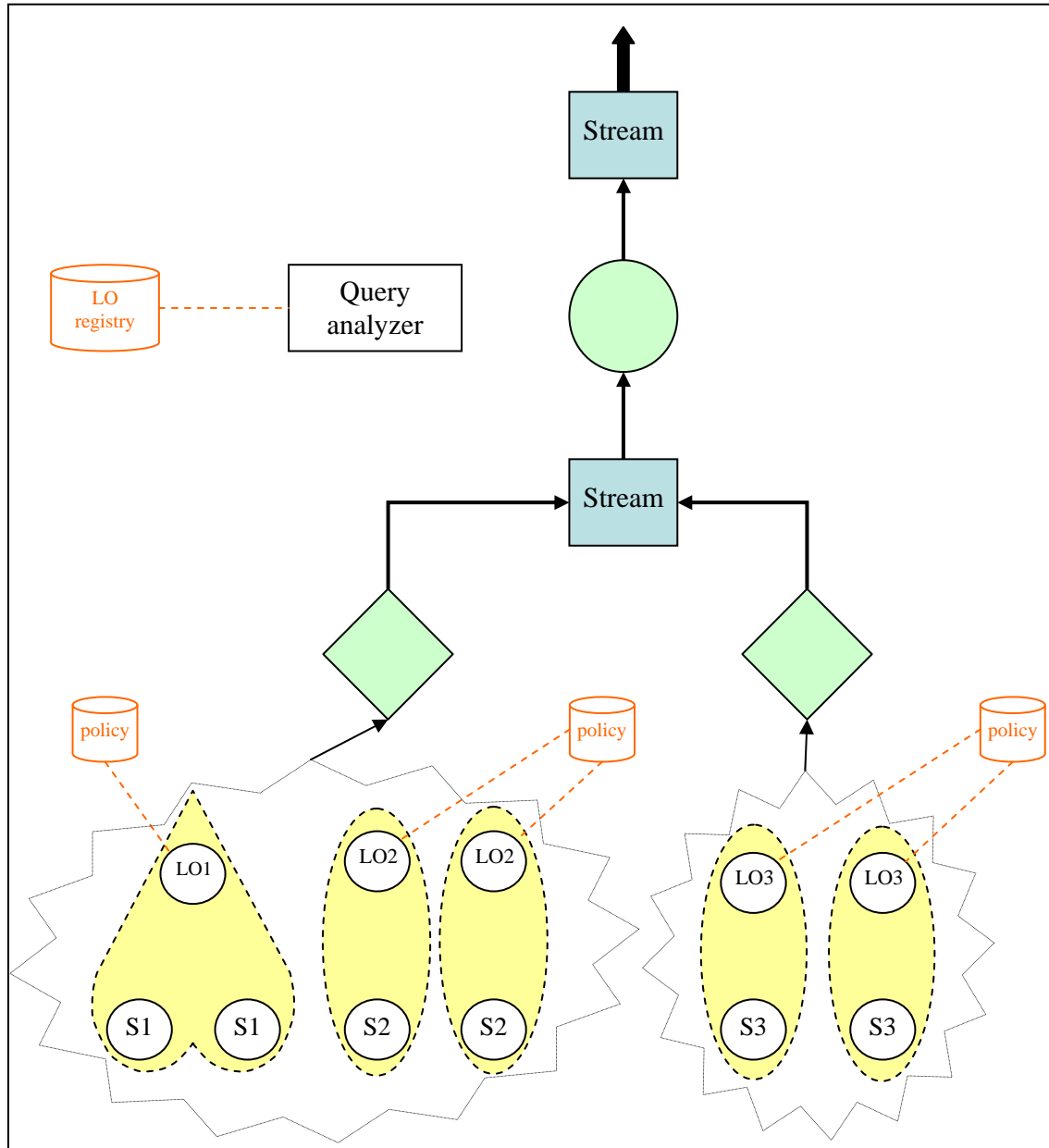


Figure 12: Example of query execution (1/3)

The figure also contains some other components that will be involved in the query execution:

- **The query analyzer**, that is a block representing all the software modules for the query parsing and distribution.
- **The logical object registry**, that represents all software modules and information needed to know the set of logical objects currently instantiated in the system.
- **Physical devices**, that are indicated with *S1*, *S2*, *S3* (note that same labels are used for homogeneous devices).
- **Logical objects**, that are indicated with *LO1*, *LO2*, *LO3* (note that same labels are used to indicate different instances of the same logical object class). Logical object *LO1* abstracts a group of homogeneous physical devices while *LO2* and *LO3* abstract a single device.
- **Policies**, that are an abstraction of the rules used by each logical object to perform the translation from abstract to concrete attributes.

The process of query decomposition and distribution is shown in Figure 13. Each thick arrow indicates that a set of commands is sent from a component to another one. The red arrow (*Q1*) is the query submitted by the user. The parsing of *Q1* produces:

- One “High level” sub-query (*Q2*) containing the statements for composing, filtering and aggregating data at the high level.
- Some “Low level, Abstract” sub-queries (*Q3*, *Q4*) containing the statements that are submitted to the logical objects and that define the behavior of sensor nodes.

It is worth noticing that the same low level query can be submitted to many different logical objects, also if they abstract different types of physical devices.

After the parser has analyzed the query *Q1*, buffers for the stream data structures are allocated. Then, the list of nodes that should take part in the query is generated using the information in the registry. Finally the sub-queries *Q3* and *Q4* are sent to all the logical objects in the list.

When a logical object receives an abstract command, it translates the command into a concrete one that can be understood by the object manager of the underlying physical device technology. The “Low level, Concrete” commands are indicated in the figure with the arrows labeled *Q5*, *Q6* and *Q7*.

It is to be noticed that the same abstract sub-query can be translated to different concrete sub-queries if it should be executed on different devices, recognizing different concrete policies constraints. For example, in the figure, the abstract query *Q3* has been translated to the concrete query *Q5* for a class of devices and to the concrete query *Q6* for another class of devices.

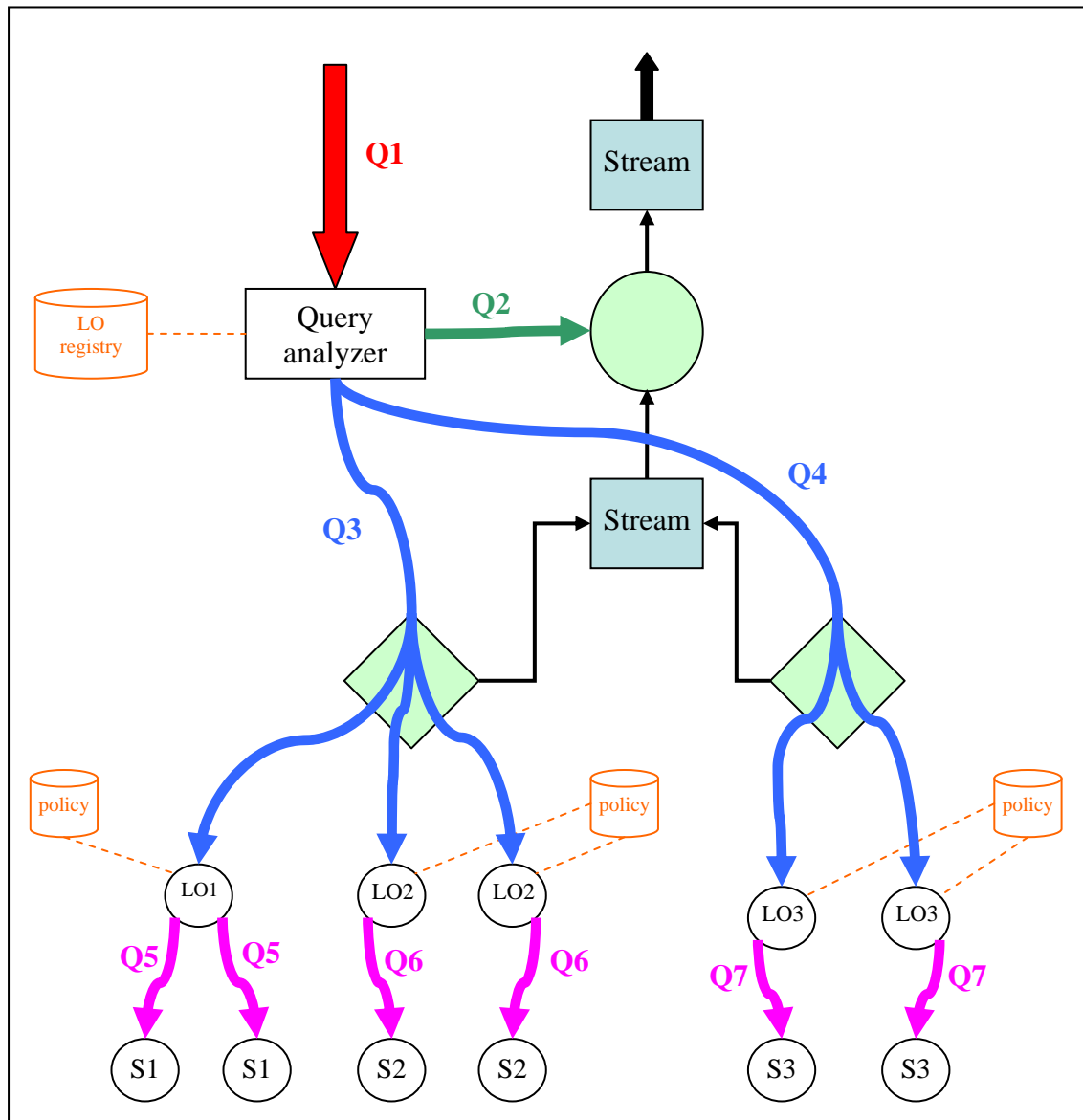


Figure 13: Example of query execution (2/3)

Figure 14 shows the data flow from the networks nodes to the output stream that is established after the query is initialized. Obviously, commands and data flows can change during the query execution if the list of logical objects involved in the query changes.

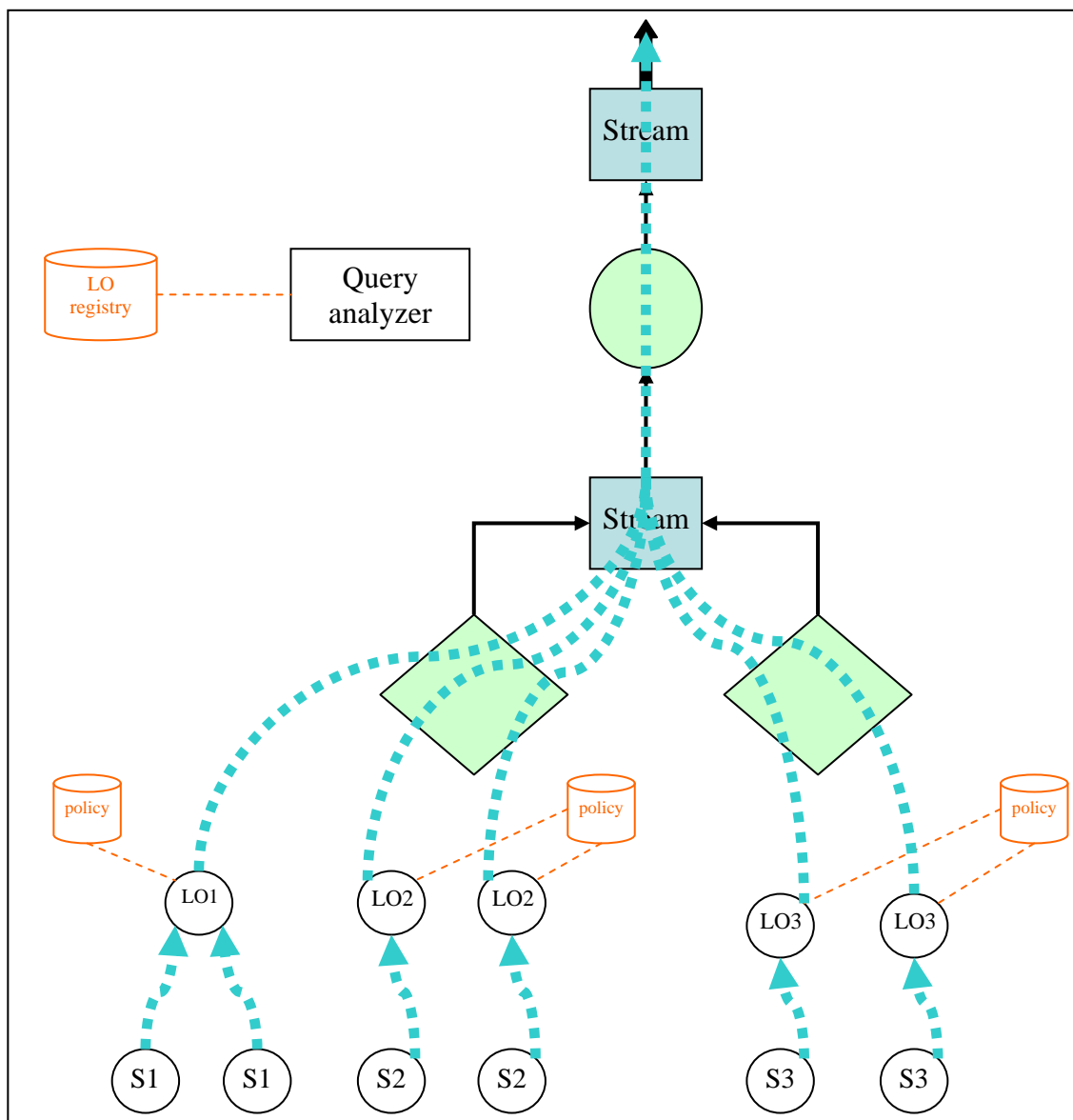


Figure 14: Example of query execution (3/3)

5 Language syntax and semantics

In this chapter we present a formal definition of *PerLa*. The syntax is defined through an *EBNF* grammar and each production is explained and commented.

The grammar is presented in a form to be clear to a human reader and not to be directly used with a parser generator software. In fact, we didn't take care, in this section, of maintaining the grammar in the *LL* or *LR* classes because we decided to ensure the readability. However, this has been done at the implementation level.

The language productions are grouped in small sets, each of them relative to a specific type of language statement. Firstly, we introduce the different types of language clauses; then we present the creation and the insertion statements that are needed to create and to populate data structures. We also describe low level and high level selections, needed to collect data from the pervasive system, and actuation statements, needed to modify some parameters on the devices. Finally, we show the structure of the expressions allowed in the language.

In this document, grammar productions are reported into a single bordered box and numbered for an easier reference. Terminals are written in uppercase to make them more readily identifiable (*e.g.*: *SELECT*), while non terminals are put into angle brackets (*e.g.*: *<Group By Clause>*). Terminals that are composed of one or two characters are put into quotes, so that they cannot be confused with grammar metacharacters (*e.g.*: '|').

Furthermore, when two non terminals with the same name are defined for the high level part and the low level part of the language, a prefix is used to distinguish between them. The "*LL*" prefix is used to identify low level productions, while the "*HL*" prefix is used to identify high level productions.

5.1 Types of SQL Statements

As discussed in Chapter 4, all the supported query statements can be classified in three classes: low level, high level and actuation statements. The main difference between *LLQ* and *HLQ* is the data sources they work on, while no data structure is involved in *AQ*. Low level queries take input data directly from the network nodes, communicating with them through the logical interfaces provided by the underlying architecture (logical objects). So, this kind of queries defines the behavior of network nodes and specifies when sensors have to be sampled and when collected data should be sent out from the node.

Vice versa high level queries elaborate data coming from other (high or low level) queries. Their semantics is very similar to the semantics of a query language for streaming databases.

```

1. <Sql Statement> →
   { <LL Sql Statement> |
     <HL Sql Statement> |
     <AQ Sql Statement> }

```

Both *LLQ* and *HLQ* can be classified in two groups that correspond to the *CREATE TABLE* and the *INSERT* statements of standard *SQL*. In fact, each interaction with the system should produce some results that have to be inserted into a data structure. So, at least two kinds of statements are needed: one to define the table structure (creation statement) and the other to define how data should be queried to fill the table (insertion statement).

As said before, since there are two kinds of data structures in the system (stream and snapshot), both creation and insertion statements can be used to create and to populate the two types of tables. As we shall show in the following, there are some differences between statements that refer to a stream table and statements which refer to a snapshot table: this is the reason that led to the definition of productions (3), (4), (6) and (7).

```

2. <LL Sql Statement> →
   { <LL Creation Statement> | <LL Insertion Statement> }

3. <LL Creation Statement> →
   { <LL Stream Creation Statement> |
     <LL Snapshot Creation Statement> }

4. <LL Insertion Statement> →
   { <LL Stream Insertion Statement> |
     <LL Snapshot Insertion Statement> }

5. <HL Sql Statement> →
   { <HL Creation Statement> | <HL Insertion Statement> }

6. <HL Creation Statement> →
   { <HL Stream Creation Statement> |
     <HL Snapshot Creation Statement> }

7. <HL Insertion Statement> →
   { <HL Stream Insertion Statement> |
     <HL Snapshot Insertion Statement> }

```

The productions introduced up to now identify eight different types of statement as shown in Figure 15. Every creation statement is composed of a creation clause that defines the name and the fields of the data structure to be created; it allows also an optional part that will be discussed in the following. The creation clause is obviously the same for low and high level queries.

All the insertion statements are composed of an insertion clause followed by a select definition clause. The first one defines the data structure in which query results will be inserted and is the

same at low and high level, while the second one is really the query part of the statement and is different for the low and the high level.

It is important to notice the differences between insertion statements that refer to streams and insertion statements that refer to snapshots. In fact, an *EVERY* clause is required when dealing with streams: it is used to define modality and frequency at which the query is computed and the results are inserted in the destination data structure. Otherwise, in the case of snapshots, that frequency is fixed equal to the size of the destination window (for example, if the window is a *30 seconds* buffer the query will be re-calculated every *30 seconds*).

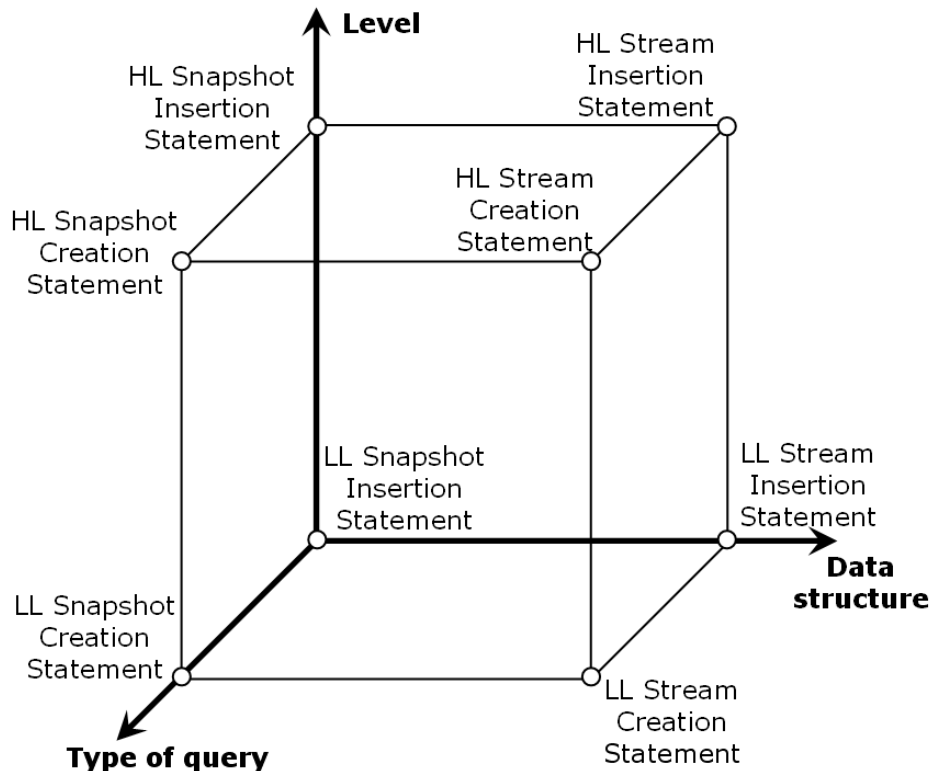


Figure 15: Classification of the language selection statements

The optional part of creation statements is a shortcut to define a creation query and a correspondent insertion query, writing only a single statement. Note that, when this optional block is specified, the creation query becomes just like an insertion query, except for the first part that requires a data structure creation.

It is not clear from the grammar how many insertion queries can be written to fill the same table. As explained in Section 4.3, there are no limits at the low level: a table can be populated by many low level statements at the same time and the union operator is applied among records coming from different queries. Vice versa, only one statement can be written at high level to fill a specified data structure.

The reason for this difference is that, at low level, the same information can be retrieved from very heterogeneous kinds of network nodes, on condition that different low level queries are used to appropriately set up involved technologies. For example, the monitoring of nodes position can be done both on devices having a *GPS* on board and devices equipped with an *RFID* tag, that indirectly calculate their position starting from the location of the nearest *RFID* reader. In this case, different queries should be written to manage these two kinds of nodes, but their results can be inserted in the *same* data structure.

- | | |
|-----|--|
| 8. | <LL Stream Creation Statement> → |
| | <Create Stream Clause |
| | [AS LOW ':' <LL Every Clause> <LL Select Definition>] |
| 9. | <LL Stream Insertion Statement> → |
| | <Insert Stream Clause |
| | LOW ':' <LL Every Clause> <LL Select Definition> |
| 10. | <LL Snapshot Creation Statement> → |
| | <Create Snapshot Clause |
| | [AS LOW ':' <LL Select Definition>] |
| 11. | <LL Snapshot Insertion Statement> → |
| | <Insert Snapshot Clause |
| | LOW ':' <LL Select Definition> |
| 12. | <HL Stream Creation Statement> → |
| | <Create Stream Clause |
| | [AS HIGH ':' <HL Every Clause> <HL Select Definition>] |
| 13. | <HL Stream Insertion Statement> → |
| | <Insert Stream Clause |
| | HIGH ':' <HL Every Clause> <HL Select Definition> |
| 14. | <HL Snapshot Creation Statement> → |
| | <Create Snapshot Clause |
| | [AS HIGH ':' <HL Select Definition>] |
| 15. | <HL Snapshot Insertion Statement> → |
| | <Insert Snapshot Clause |
| | HIGH ':' <HL Select Definition> |

Note that, in the previous productions, the *HIGH* and the *LOW* keywords have been introduced between the end of each level independent clause (creation, insertion) and the beginning of the next level dependent clause. The reason of this choice will be clearer when the parser implementation will be discussed.

5.2 Creation statements

We now consider creation statements that are very similar to the *CREATE TABLE* construct of standard *SQL*: they define the name of the data structure to be created and the list of fields. No other information must be provided for streams, while the size of the buffer must be specified for snapshots (we call this size "*the duration*" of the snapshot).

The name, the data type and optionally the default value have to be specified for each field. Besides general *SQL* types (*BOOLEAN*, *INTEGER*, *FLOAT* and *STRING*), *ID* and *TIMESTAMP* are supported because they have a particular meaning with respect to our architecture. *ID* is a special type that can be used to unambiguously identify specific nodes of the network. The real data type associated to *ID* can be a string, if logical objects are identified by an *URI*.

The *TIMESTAMP* data type is used for fields containing a time reference. It is the type of the "*Native Timestamp*" hidden field, that is present in each data structure, but it can also be used for other user-defined fields. The real type associated to timestamps is an integer value.

A particular remark is needed for the default value. We will see that an insertion clause can specify the value only for a subset of the data structure attributes: in this case the remaining fields of each inserted record are set to the default value of the attribute.

Note that the default value is optional in the creation statement: if not specified, a predefined default value is used, whose value depends on the field data type as shown in the following table:

Type	Default value
DOUBLE	0
INTEGER	0
BOOLEAN	FALSE
STRING	""
ID	/
TIMESTAMP	0

Figure 16: Data types and default values

The *OUTPUT* keyword can be used to specify that the created data structure is an output data structure. The system must know this information because the records inserted in the output tables are to be returned to the user who submitted the query. Vice versa, the content of the data structures that are not marked as output tables should only be transferred from the node that generated it to the nodes that will use it.

```

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

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

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

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

```

5.3 Insertion statements

Looking at productions (9), (11), (13) and (15), we can observe that each insertion statement is composed of an introductory part (the insertion clause) that defines the destination data structure, and a second part (the selection definition) that specifies the real query to be done on the sources.

The insertion clause is the same for low and high level queries and contains the name of the destination data structure and the subset of fields for which the selection statement will provide values. Obviously, all the elements in the field list must be attributes of the destination table. As said before, when the destination data structure is a stream, a third component appears in the insertion statement: it is the *EVERY* clause, used to specify how and when the selection part of the statement should be executed and its results appended to the destination table.

Production (22) is the *EVERY* clause for low level statements. If duration is specified, the selection part of the query is activated at regular intervals whose length is the duration itself. Otherwise, a number of samples can be specified to obtain a completely different behavior: the selection is activated when the specified number of samples is generated by the sampling part of the query. While the first approach is *time based*, this second approach is *event based* and the activation instant of the selection part cannot be foreseen in advance. The *EVERY* clause allows also the *ONE* option that is just a useful shortcut for *1 SAMPLES*.

The semantics of the *EVERY* clause in the high level statements is similar to the one presented above. More specifically, it is exactly the same when the duration is specified. However, when the event based approach is used, a table name must be provided as well the number of samples. In fact, high level statements can work on more than one input table, differently from the low level statements that work on data coming from a single logical object. So, in the high level case, the user must specify the name of the data structure that will be used as event source.

A last remark is the possibility to specify the keyword *SYNCHRONIZED* in the low level *EVERY* clause when the time based semantics is used: in this case the system will force the first execution of the selection part at a timestamp that is a multiple of the specified duration.

An example will clarify the concept. Suppose that a low level query, that inserts data into a table every *10 minutes*, is started after the evaluation of the *EXECUTE IF* clause at the *minute 27*. If the *SYNCHRONIZED* keyword is not specified, the first activation of the query will be at the *minute 37*. In the opposite case, the start of the query will be postponed by *3 minutes*, so that the first activation of the query will be at the *minute 40*.

The keyword *SYNCHRONIZED* is provided because there are some situations in which the user must exactly know when data will be inserted into a data structure (e.g.: when the content of that data structure is joined with another stream by a high level query). Note that the keyword is not allowed in the high level *EVERY* clause, because all the high level queries start together at query startup.

```

20. <Insert Stream Clause> →
    INSERT INTO STREAM <Data Structure Name> [<Field List>]

21. <Insert Snapshot Clause> →
    INSERT INTO SNAPSHOT <Data Structure Name> [<Field List>]

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

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

24. <Field List> →
    '(' <Data Structure Field> { ',' <Data Structure Field> }* ')'
```

5.4 Low level SELECT definition

Now we consider the set of rules that define the low level selection. Production (25) shows that the statement is quite complex because it is composed of many clauses. However, some of them are optional and most of the common queries .

The selection statement has the role of specifying when data should be collected from the logical object acting as a query source and how it is manipulated to produce output records. A low level query can be decomposed into three main blocks:

- **Sampling section** specifies when the sensors must be sampled and the conditions to decide if a sampled record is valid, before storing it into a local buffer;
- **Data management section** is periodically activated and executes a selection on the local buffer to produce the results that will be inserted in the destination data structure.
- **Execution conditions section** defines the rules to establish if a certain logical object should execute the query (*EXECUTE IF* and *PILOT JOIN* clauses);

```

25. <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>]

```

The simplest query a user can write contains only the *SELECT* and the *SAMPLING* clauses. All the network nodes will take part in such a query (this concept will be clearer when the *EXECUTE IF* clause will be presented).

5.4.1 Sampling section

The first clause we analyze is the *SAMPLING* one. First of all, we briefly recall what we mean with the term "sampling". Remember that a low level query is logically executed on a single network node from which data is collected, communicating through the interface provided by the logical object that wraps it. With the term "sampling" we mean that one or more attributes of the logical object are read and the obtained record is inserted into a local buffer, if it satisfies some conditions. Note that the reading of an attribute can imply a sensor sampling if the attribute is dynamic and probing.

As it can be seen in production (26) two types of sampling are supported: *event based* and *time based*. The first one is obtained writing the *ON EVENT* clause and forces a sampling whenever an event is raised by the logical object. The second is obtained writing the *SAMPLING IF EVERY* clause and forces the sampling to be executed periodically with a certain frequency. The *ON EVENT* clause requires the list of events that will be used to drive the sampling operation. The *IF EVERY* clause is more complex because it allows a parametric definition of the sampling rate. In fact, a construct like an *if-elseif-else* is provided and the user can specify different sampling rates for each branch of the construct. Moreover, the frequency can be expressed with a general expression, in order to calculate the optimal needed sampling rate (e.g.: the sampling rate can be defined as a function of the attribute giving the remaining power).

If at least one "if branch" is defined, the system evaluates the conditions in order of appearance until one of them is satisfied: at this point the sampling rate is set to the value established in the correspondent *EVERY* clause. If no condition is satisfied the sampling rate specified in the *ELSE* clause is used. Note that if a zero value or a negative value is obtained evaluating the sampling rate expression, the sampling operation will not be performed.

To clarify the distinction between event based and time based sampling, consider the following example. Suppose we want to retrieve the list of *RFID* readers that sense a specific *RFID* tag. The logical object that wraps the *RFID* tag probably has at least a static attribute (the tag *ID*), a dynamic non probing attribute (the last *RFID* reader who sensed the tag) and an event (that signal the sensing). In this situation, a reasonable sampling mode is the event based one: a record is appended to the local buffer whenever the event is raised.

The opposite situation is the case of a node with a temperature sensor on board, that should be sampled every *10 minutes*. In this case, the logical object that wraps the node has a dynamic probing attribute that returns the current temperature. To obtain the required sampling rate, a time based sampling mode must be used.

When the time based sampling is used, other two clauses can be specified in the statement. The first one allows to define the behavior of the node if the required sampling rate is too fast and then unsupported. The "*DO NOT SAMPLE*" option indicates that the query execution should be suspended until the required sampling rate become acceptable again. Otherwise the "*SLOW DOWN*" option allows keeping on the query execution, but with an automatic reduction of the sampling rate. If the clause is not specified, the option "*DO NOT SAMPLE*" is used as the default one.

The second clause is the *REFRESH* one and it allows the user to specify if the sample rate is fixed for the whole query life or if it should be periodically reevaluated. If activated, the refresh mode can be event based or time based, exactly as the sampling mode. If the time based mode is used, the duration is usually set to a very long interval with respect to the sampling frequency of the query. Note that the *REFRESH* clause can be useful also when no "*if branches*" are specified and the sampling rate is defined with a single expression: in this case the refresh forces a new evaluation of that expression.

The *SAMPLING* clause can be completed appending a *WHERE* clause, that allows the definition of a filtering condition. If specified, this condition is evaluated whenever a record is produced, before the insertion into the local buffer. If the condition is not satisfied, the record is immediately rejected without inserting it into the buffer. A discarded record is not considered anymore during the query execution. For example if the selection statement is activated every 3 records, the rejected one is not counted for the activation.

```

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

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

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

```

```

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

30. <Sampling If Clause> →
    IF <Condition>

31. <Sampling Every Clause> →
    EVERY <Expression> <Time Unit>

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

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

34. <Where Clause> →
    WHERE <Condition>

```

5.4.2 Data management section

With the *SAMPLING* clause we have defined the behavior of the low part of the query that reads logical objects attributes and inserts collected values into the local buffer. Now we present the *SELECT* clause that is the most important clause of the query.

The *SELECT* clause is evaluated with a frequency defined in the *EVERY* clause of the insertion query, as described before. The selection is executed on the local buffer that is ideally unbounded. In practice, all the fields specified in the *SELECT* clause are calculated on a bounded subset of the infinite buffer and then the system should be able to maintain only the needed portion of the buffer.

The idea behind the *SELECT* clause is the same as in standard *SQL*: they both have the role of specifying how data have to be collected from data sources (the unbounded buffer in our case; the tables in *SQL*) and manipulated to produce some records as result. However, there are important differences in the semantics, due to our decision of focusing especially on aggregates. In fact, we think that in many real situations the data of interest is directly an aggregation of some samplings rather than the list of all sampled records.

We describe the semantics of the data management section starting from the simpler case in which only the *SELECT* clause is specified. In this hypothesis, a single record is generated: each field of this record is an aggregation obtained from the current buffer content. From the grammar point of view, a field of the *SELECT* clause is an expression and can contain aggregation operators.

Differently from standard *SQL*, these operators have more than one parameter. The first one is the value on which the aggregation should be computed and can be an expression involving one or more logical object attributes. The second parameter is a duration or a number of records and

represents the portion of the buffer that is used to calculate the aggregation. For example, the element *MIN (temperature, 20 seconds)* requires the minimum temperature value present in the buffer and relative to the last 20 seconds, while the element *MIN (temperature, 20 SAMPLES)* refers to the last 20 records in the buffer (independently of the epoch in which they were generated). Note that aggregations relative to different portions of the buffer can be used in different fields of the *SELECT* clause or, even, in the same expression. An example is shown in Figure 17.a: the *SELECT* clause extracts a percentage value indicating how many times a tag was seen by an *RFID* reader during the last ten seconds with respect to the last minute.

A third optional parameter can be introduced in each aggregate to define a filter. This condition is applied to the records that have to be aggregated, before the aggregation is done. For example the element *COUNT (*, 60 seconds, temperature > 50)* requires the number of temperature samplings that exceeded a threshold of 50 degrees during the last 60 seconds (Figure 17.b). Note that in the same figure the element *COUNT (*, 60 seconds, temperature < 30)* is computed starting from the same buffer portion, but applying a different filter condition. An attribute can be present alone in the *SELECT* clause, out of any aggregation operator: it will be interpreted as a shortcut for *SUM (attribute, 1 SAMPLES)* and then the value of that attribute in the most recent record of the buffer will be returned (Figure 17.c).

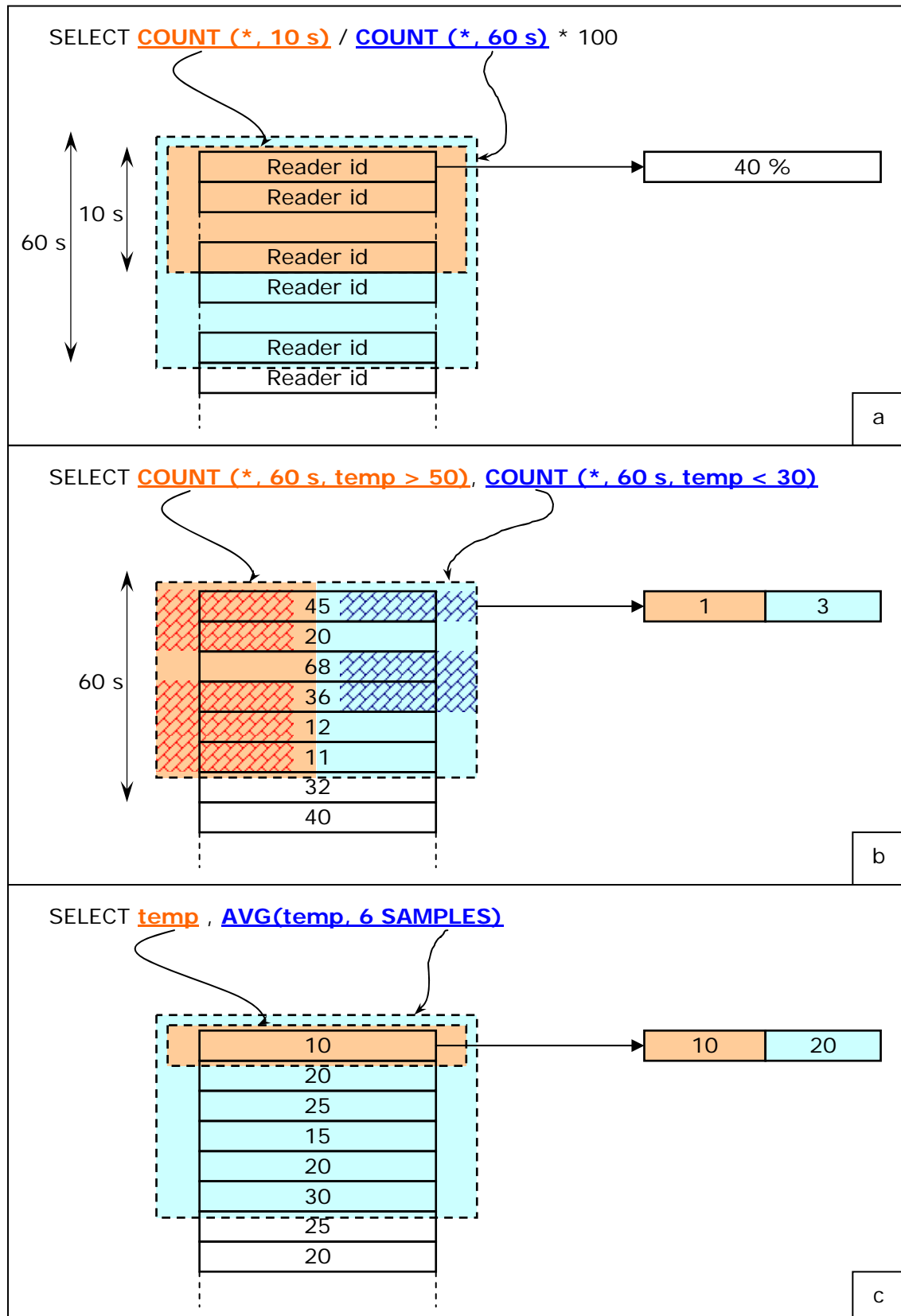


Figure 17: Low level buffer management and SELECT clause

In grammar production (37) each field of the *SELECT* clause is represented as an expression: Section 5.6 will precisely clarify the set of attributes and operators that can be used in that expression. At the moment, the reader should only know that all the aggregation and standard operators, all the logical object fields and the *ID* and *GROUP_TS* special fields can be used. *ID* returns an unambiguous identifier of the logical object that is executing the query; while *GROUP_TS* returns a timestamp value and its meaning will be clear when the *GROUP BY* clause will be introduced.

It is worth noticing that *DISTINCT* and *ALL* keywords have the same semantics as in standard *SQL*, but they are not useful when the *SELECT* clause is used alone without other optional clauses: in fact, as seen before, in this case only a record is generated by the *SELECT* clause. A default value for each element of the *SELECT* clause can be specified and its meaning will become clear in the following.

```

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

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

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

38. <LL Aggregate> →
    { <LL Aggregate Count> | <LL Aggregate Other> }

39. <LL Aggregate Count> →
    COUNT '('
    '*' ','
    { <Duration> | <Integer Constant> SAMPLES | ONE }
    [ ',' <Condition> ] ')'

40. <LL Aggregate Other> →
    <Aggregation Operator> '('
    <Expression> ','
    { <Duration> | <Integer Constant> SAMPLES | ONE }
    [ ',' <Condition> ] ')'

41. <Aggregation Operator> →
    { AVG | MAX | MIN | SUM }

```

We now extend the presented subset of the language with other clauses to add more functionality to the data management part of the statement. The *UP TO* clause allows to retrieve more than one record from the local buffer, giving to the *SELECT* clause a semantics more similar to the standard *SQL* one. A duration or a number of samples can be specified to identify the buffer window that will be returned. For example, if a duration of *1 minute* is specified in the *UP TO* clause, the selection returns all the records in the buffer that are relative

to the last minute. This clause doesn't modify the semantics of the aggregations that can appear in the *SELECT* clause. Note that if the *UP TO* clause is not specified, an *UP TO* of 1 record is taken as a default, falling again within the above explained situation of a simple *SELECT*. This justifies the semantics we gave to single attributes in the *SELECT* clause, when no optional clauses are present. Figure 18 better clarifies the behavior of the *UP TO* clause.

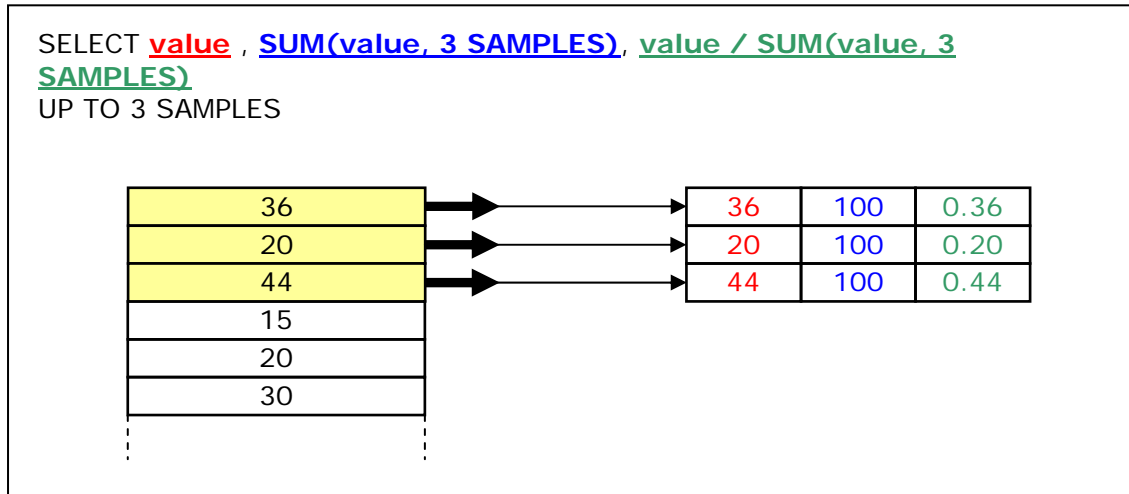


Figure 18: Low level buffer management and UP TO clause

42. <Up To Clause> →

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

The *GROUP BY* clause allows a grouping of the records in the local buffer before the selection process described above is computed. Two kinds of grouping are supported: *attribute based* and *timestamp based*. If the grouping is done on an attribute, the (ideally infinite) buffer is splitted into many (ideally infinite) buffers. Each of them refers to a single value of the attribute. Then, the *SELECT* clause is evaluated on each buffer exactly as described above (Figure 19.a).

The second type of grouping is based on timestamps. A duration d and a number of groups n should be specified. In this case the local buffer is ideally replicated into n identical buffers. Then the records relative to the first $d * i$ seconds are removed from the i -th copy of the buffer. Finally the *SELECT* clause is computed on each of the n buffers (Figure 19.b). The field *GROUP_TS* retrieves the initial timestamp of the buffer replica from which the output record is generated.

The grouping can be made on more than one attribute simultaneously with an obvious semantics.

```
43. <Group By Clause> →  
    GROUP BY <Field Grouping By List>  
  
44. <Field Grouping By List> →  
    { <Field Grouping By> | <Field Grouping By TS> }  
    { \,' <Field Grouping By> }*  
  
45. <Field Grouping By> →  
    <Logical Object Field>  
  
46. <Field Grouping By TS> →  
    TIMESTAMP \(' <Duration> \,' <Integer Constant> GROUPS \)'
```

The *UP TO* can be used together with the *GROUP BY* clause: in this case the number of records specified by the *UP TO* clause indicates the number of records that will be considered by the selection process from each group that is generated by the *GROUP BY* clause (Figure 20).

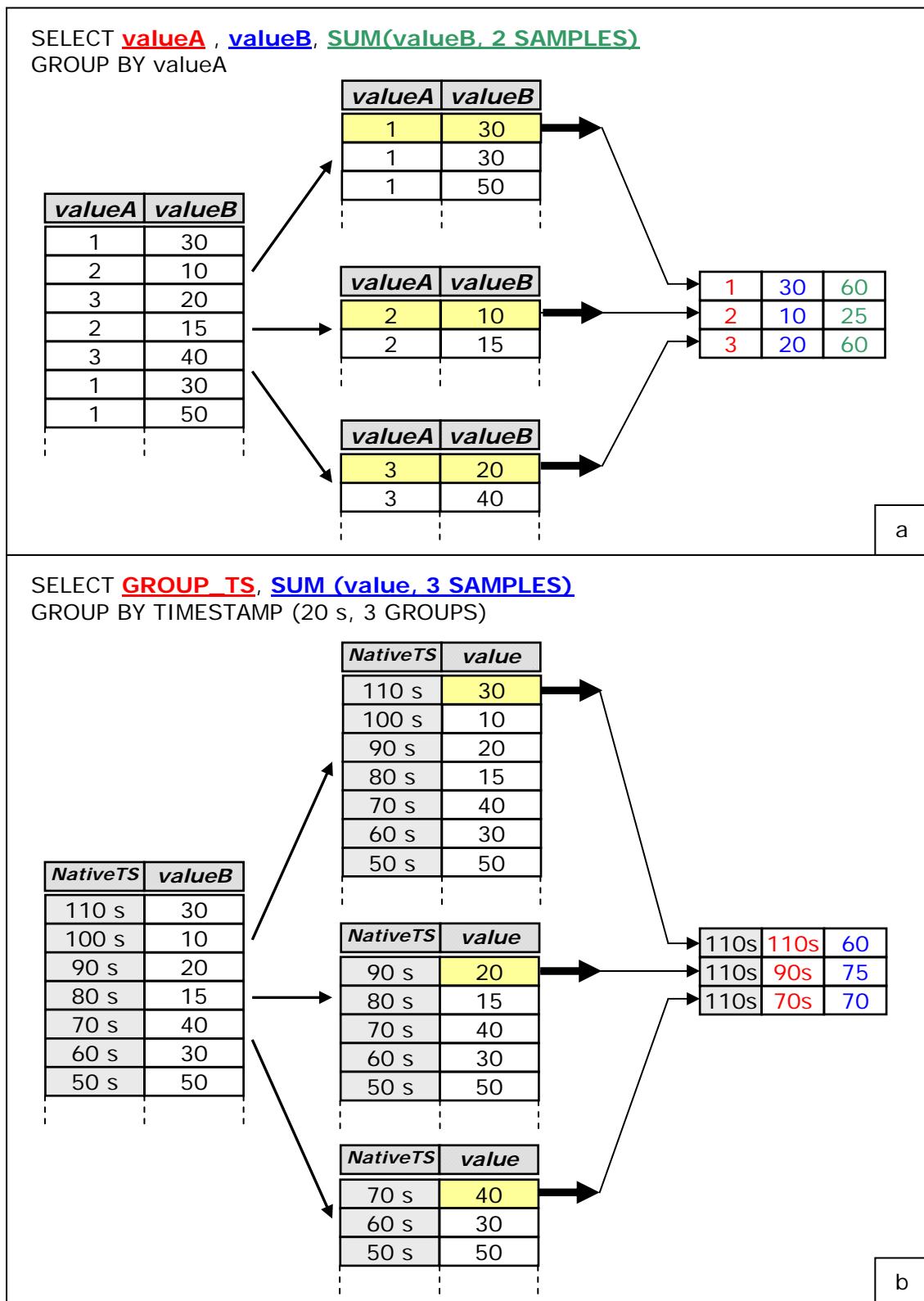


Figure 19: Low level buffer management and GROUP BY clause (1)

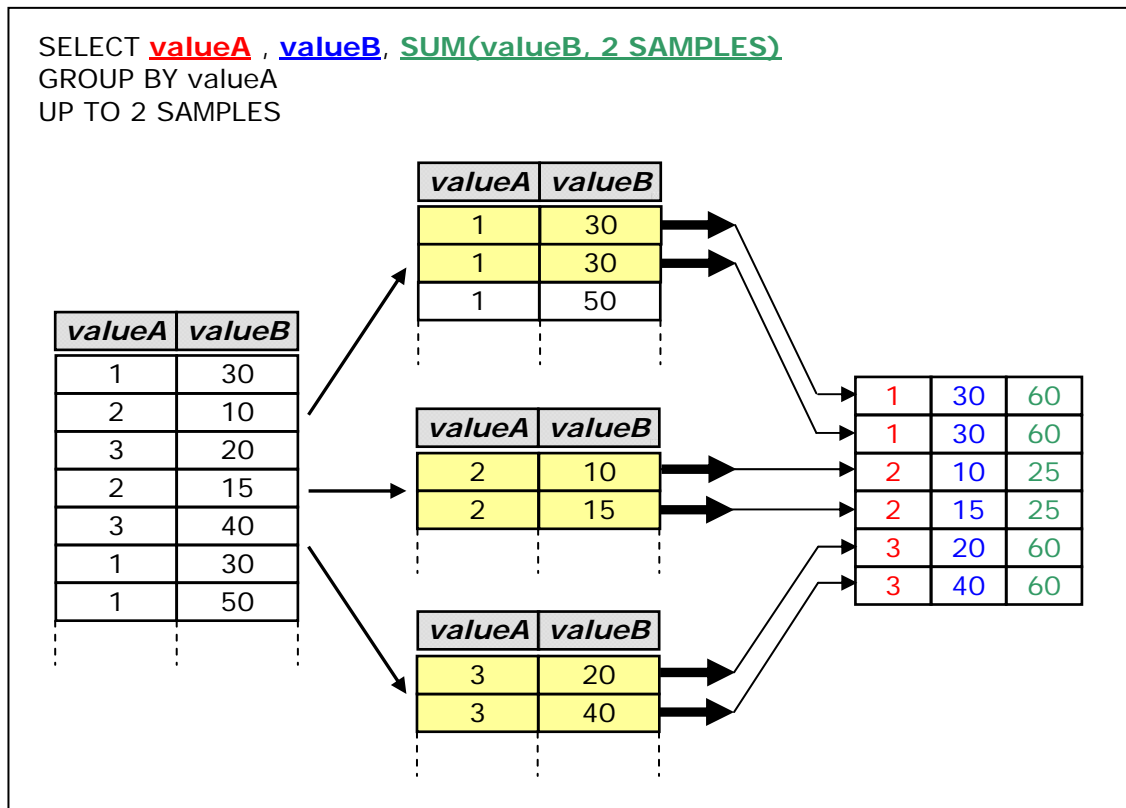


Figure 20: Low level buffer management and GROUP BY clause (2)

We said that when the selection process is activated, the first executed step is the application of the *GROUP BY* clause to split the content of the local buffer. Then, the *UP TO* clause is used to determine how many records from each portion of the buffer should be taken to produce output records. Finally, starting from each of these records, the expressions that appear in the *SELECT* clause are evaluated and the obtained records are inserted in the output data structure. The *HAVING* clause allows a filtering operation on the records identified by the *UP TO* clause, before output records are produced. In Figure 21 an example that includes the *HAVING* clause is shown: records not satisfying the *HAVING* condition are indicated with a prohibition symbol. Note that the *HAVING* clause covers both the *WHERE* and the *HAVING* clauses of traditional *SQL*. We decided to call this clause “*HAVING*”, instead of “*WHERE*”, because in many real queries it is used to filter aggregates, instead of single records.

47. <Having Clause> →
 HAVING <Condition>

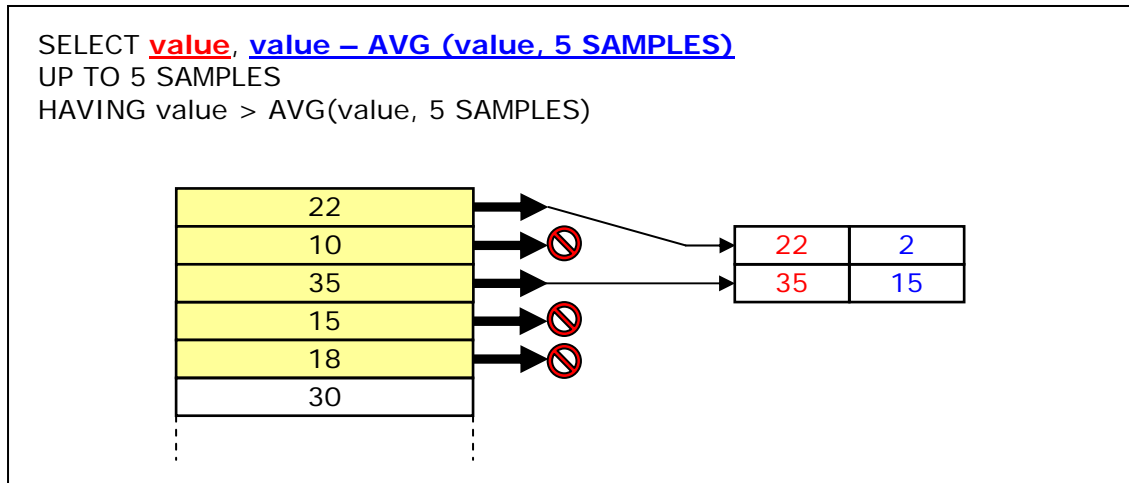


Figure 21: Low level buffer management and HAVING clause

The *EMPTY SELECTION* clause allows the user to specify the behavior of the query when the selection process doesn't produce any output record (this condition can happen, for example, when the *HAVING* clause filters all the records).

Two options are possible: *INSERT NOTHING* means that no records are inserted in the destination data structure when the buffer is empty, while *INSERT DEFAULT* means that, in the same situation, a default record is generated and inserted into the destination data structure. The fields of this record are set to the default values specified by the *SELECT* clause. (If these values are not specified, the default values of the attributes, defined with the data structure creation statement, are used).

If the *EMPTY SELECTION* clause is not specified, the *INSERT NOTHING* option is used as default.

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

Figure 22 shows a complete example of selection process in which all the clauses presented in this section are involved.

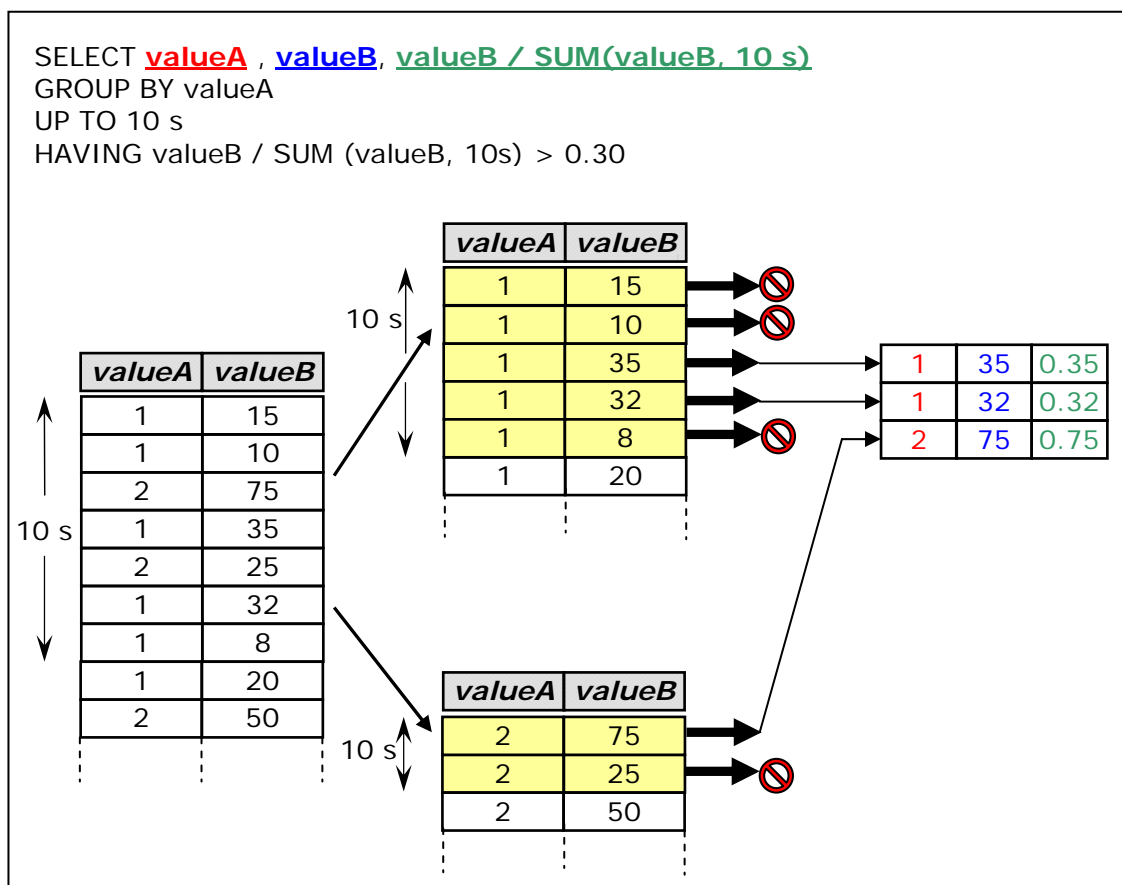


Figure 22: Complete example of low level buffer management

5.4.3 Execution conditions

To complete the analysis of the low level selection statement we now explain the clauses needed to define the set of logical objects that will be involved in the query execution: *PILOT JOIN* and *EXECUTE IF* clauses.

The first one is a special statement that allows a conditional execution of a query on a node. It is the implementation of the *pilot join* operation that was introduced in previous chapters and that was graphically represented as a dashed arrow in Figure 7. The condition, the query execution depends on, is based on the content of an existing data structure: if it is a stream, the *pilot join* is event based and, whenever a new record is appended to that table, the logical objects matching with that record start executing the query; if it is a snapshot, the set of logical objects that execute the query is reevaluated periodically with a period equal to the snapshot duration.

The grammar productions reported below show that a *pilot join* operation can be executed on more than one data structure simultaneously. As explained in Chapter 4 there are some limitations, depending on the type of the used data structures: at most one *pilot join* can be defined with a stream, while an arbitrary number of *pilot joins* can be defined with snapshots. It

is not possible to require simultaneously a *pilot join* with a stream and a *pilot join* with a snapshot.

The condition in production (51) must be a Boolean expression and can refer to logical object fields (including dynamic probing attributes) and to the correlated tables' fields. Aggregation operators cannot be present in the condition because it is evaluated before the query starts. Correlated table fields must be referred to in the condition using the syntax: “*data_structure_name.field_name*”.

Note that the most common use of the *pilot join* operation is to activate a certain low level query on the wireless sensor nodes that are currently connected to a base station present in a certain list: in this case, the *PILOT JOIN* clause has the following structure:

```
PILOT JOIN BaseStationList ON
currentBaseStation = baseStationList.baseStationID
```

where “*currentBaseStation*” is a dynamic non probing attribute of the logical object that indicates the *ID* of the base station the node is currently wireless connected to.

<pre>49. <Pilot Join Clause> → PILOT JOIN <Correlated Table List> 50. <Correlated Table List> → <Correlated Table> { ',' <Correlated Table> }* 51. <Correlated Table> → <Data Structure Name> ON <Condition></pre>
--

The second clause that can be used to impose conditions on the execution of a query is the *EXECUTE IF* clause; it allows to define the set of logical objects that will participate to the query and it is essentially a condition that is evaluated *before* the query is distributed to network nodes. This condition must comply with the same constraints indicated for the *PILOT JOIN* clause. Moreover, it cannot contain references to correlated table fields because the *EXECUTE IF* condition is evaluated before the analysis of the *PILOT JOIN* clause. A useful Boolean function that can be used in the condition is *EXISTS (attribute)*: it returns *TRUE* if the logical object supports the specified attribute, *FALSE* otherwise. If the keyword *ALL* is specified as parameter, the function returns *TRUE* if all the logical object attributes referred in the whole low level statement are supported. Note that this function is not bounded in the *EXECUTE IF* condition and can be used elsewhere in low level language expressions and conditions.

A *REFRESH* clause can be appended to the *EXECUTE IF* one in order to indicate if and when the system should reevaluate the condition to update the list of logical objects executing the query. The syntax of the *REFRESH* clause has been explained before, when the *SAMPLING* clause was presented. If the *REFRESH* clause is not specified, the condition is evaluated only when the user submits the query: if satisfied, the logical object will be involved in query execution for the whole query life time; otherwise, if the condition is not satisfied, the logical object is excluded from the query once and for all.

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

The last clause that can be specified in a low level selection statement is the *TERMINATE AFTER* one, that can be used to set the timeout after which the logical object will leave the query execution. This timeout can be specified both in terms of a time interval (“*duration*”) and of a number of activations of the query data management section. The timeout starts after the *EXECUTE IF* condition is evaluated if the *PILOT JOIN* clause is not specified. Otherwise, it starts after the *pilot join* condition becomes satisfied.

The *TERMINATE AFTER* clause is optional and, when not specified, each logical object involved in the query continues executing it, until the user requires the termination of the whole submitted query.

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

5.4.4 Low level statements UML

In this section, an *UML* activity diagram is reported in order to summarize the concepts presented in previous sections. It aims at explaining the main steps that should be performed by a query engine to execute a low level statement on a logical object. Logical steps reported in the diagram have the role of clarifying the language semantics and of guiding the user that is writing queries. Diagram is intended neither to explain how a real language implementation can work, nor to explain which entities of the architecture should execute the different activities.

Figure 23 represents the logical steps executed by the query engine when all the execution conditions are satisfied: the activity flow on the left-hand side models the sampling section of the query, while the activity flow on the right-hand side models the data management section.

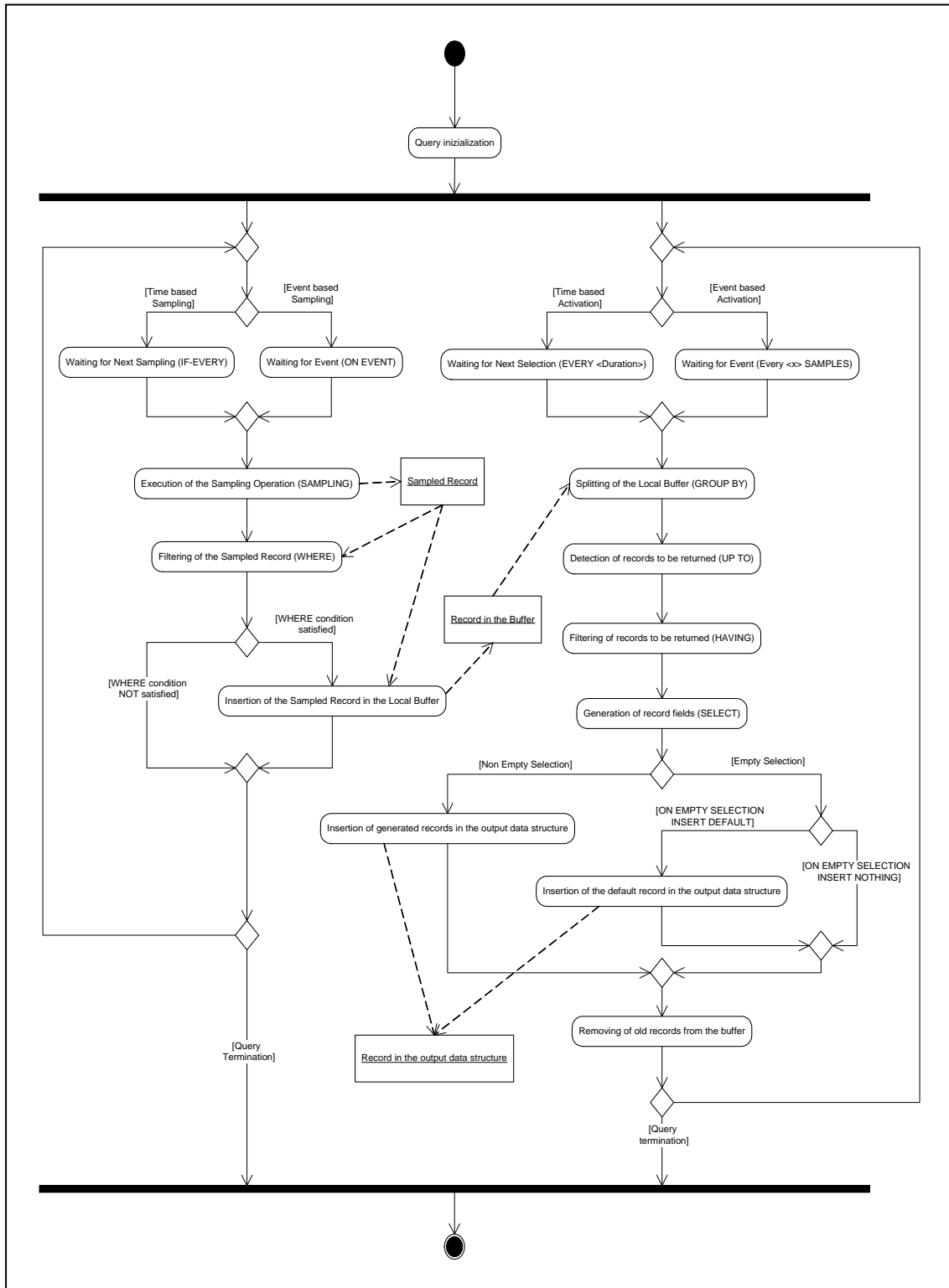


Figure 23: UML activity diagram of low level query buffer management

5.5 High level SELECT definition

The high level selection statement is syntactically and semantically similar to the standard *SQL*, but there are some important differences. The first one is related to the kind of data used as source. In fact, in standard *SQL*, each table listed in the *FROM* clause is a static table, while the sources of our high level queries are streams. We explain how these streams are managed when the *FROM* clause is presented.

The *UNION* keyword has exactly the same semantics as in standard *SQL* and allows the union of data coming from different selections. Obviously, even if not explicitly stated, all the *SELECT* clauses must extract the same number and type of fields.

As described for the low level language, the *ON EMPTY SELECTION* clause has been introduced also at high level to ensure that, if needed, at least one record is produced every time the *SELECT* statement is evaluated. This clause can be useful to manage all the situations in which the user would like to know if a missing record is due to a communication error or to an empty selection result.

```

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

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

```

The overall structure of the *FROM* clause is the same as in standard *SQL*: a list of elements separated by a comma character must be provided; but each of these elements is not just the name of a database table: it defines a window on a stream. Production (58) shows that the window size can be defined either in terms of a time interval or a number of records.

The *AS* keyword allows the definition of an alias for the window, that have to be used elsewhere in the query to refer to that window. If an alias is not specified the name of the source stream is taken as default. Obviously, the alias definition is mandatory when the same stream is used twice in the same *FROM* clause.

The windows list in the *FROM* clause allows also the specification of static non streaming tables, whose content is fixed and known by all the sensors in the network. These tables are not obtained extracting a window from a stream, so the user has just to write their name. Static non streaming tables are useful to store parameters and information that were fixed during the setup and the deployment phase of the network. For example, physical connections between devices can be maintained.

Consider the case in which the position of some containers is traced with *GPS*. Suppose that a base station is placed in each container to monitor the temperature of the contained objects. In this situation the mapping between *GPS* nodes and base stations mounted over the same container can be stored in a static non streaming table. This information can be used to force a

temperature sampling only in the containers located in a certain zone (using appropriately the *pilot join* feature).

Note that an alternative way to store static parameters, without using static tables, is to expose them through logical objects (in the example above the base station *ID* can be thought as an attribute of the logical objects wrapping the *GPS* devices).

```

56. <From Clause> →
    FROM <Window Definition List>

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

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

```

Conditions and expressions used in *SELECT*, *WHERE* and *HAVING* clauses have the same syntax and semantics previously described for low level statements, but there are some differences in attributes and aggregations; in fact, at the high level, attributes must be fields of a table contained in the *FROM* clause.

Aggregates are functions of one or two parameters. The first one is the name of the attribute the aggregation is relative to, while the second one is an optional *WHERE* clause that allows a filtering before the aggregation is computed. Note that the duration parameter that is supported at low level cannot be used at the high level. In fact, the aggregation is always calculated on the records obtained from the Cartesian product of the windows specified in the *FROM* clause.

So, the main difference between standard *SQL* aggregates and our high level aggregates is the possibility of defining the *WHERE* condition. This feature was introduced to allow users to write in a more compact way queries requiring nested statements when written in standard *SQL*. *The above consideration allowed us to avoid supporting nested queries in our language.*

Another difference is relative to the semantics of aggregates when the *GROUP BY* clause is not specified: in standard *SQL* all the source records are grouped together to produce a single output record, while in our language an output record is generated for each source record (Figure 24). To reproduce the standard *SQL* behavior we introduced the *ALL* keyword in the *GROUP BY* clause: if it is specified all source records are grouped together. Otherwise, whenever the *GROUP BY* clause specifies a list of fields, the semantics of the query is exactly the same as the standard *SQL* one.

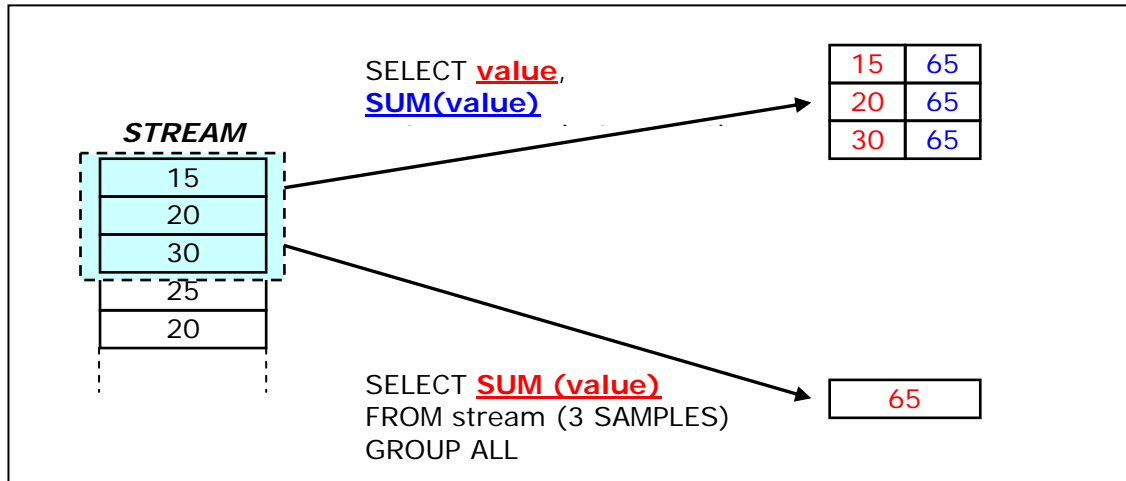


Figure 24: High level aggregations and GROUP BY clause

- ```

59. <HL Aggregate> →
 <HL Aggregate Count> | <HL Aggregate Other>

60. <HL Aggregate Count> →
 COUNT '(' '*' [',' <Condition>] ')'

61. <HL Aggregate Other> →
 <Aggregation Operator>
 '(' <Expression> [',' <Condition>] ')'

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

63. <Window Field List> →
 <Window Field>
 { ',' <Window Field> }*

```

## 5.6 Actuation statements

Actuation queries are simple statements that allow the user to set the value of one or more logical object parameters. From a syntactical point of view, an actuation query is basically composed of two blocks. The *SET* clause allows to define the list of attributes to be set and to specify the values of those attributes. The optional *ON* clause can be used to bound the set of logical objects involved in the query execution.

```
64. <AQ Sql Statement> →
 <SET Clause>
 [<ON Clause>]
```

The *SET* clause is a comma separated list of assignments. Each of those assignments specifies the value for a given attribute through an expression. As better explained in the following, the *Logical Object Field* production is just an identifier that indicates the name of a logical object attribute. The *Expression* has the same semantics already explained for the *SELECT* clause of low level statements, with the only limitation that aggregation operators cannot be adopted. In fact, the expressions contained in an actuation query are evaluated once and the only available data is the current set of logical object attributes values.

```
65. <SET Clause> →
 SET <Set Pairs List>

66. <Set Pairs List> →
 <Set Pair> { ',' <Set Pair> }*

67. <Set Pair> →
 <Logical Object Field> '=' <Expression>
```

The syntax and the semantics of the *ON* clause are quite similar to the syntax and the semantics of the *EXECUTE IF* clause of low level queries. The only difference is that the *REFRESH* clause cannot be specified, since an actuation query is executed one shot. Thus, the execution condition is evaluated once when the query is submitted and there is no meaning in evaluating it again. The *EXISTS* function can be used to limit the query execution only to the logical objects that expose a given attribute or all the attributes referenced in the *SET* clause.

If the *ON* clause is not specified, only the logical objects that exposes all the attributes assigned in the *SET* clause are charged to execute the query. The presence of the *ON* clause imposes a further filtering on the list of involved objects.

It should be noticed that, if an unsupported attribute is referenced in an expression, the null values semantics described above is applied.

A logical object attribute can accept only a range of values, but the evaluation of the assigned expression can result in a value that is out of that range. Moreover, the physical actuation operation can fail. In both this situations, the attribute value is not changed by the actuation query. However, the user can have a feedback by executing a simple one shot *LLQ* query on the logical object to retrieve the current attribute value.

```
68. <ON Clause> →
 ON <Condition>
```



## 5.7 Expressions and conditions

Expressions and conditions are required in many language clauses. Although they have a common general structure and they support the same set of the operators, different attributes and aggregations can be used in different clauses. In fact, some clauses belong to high level statements, some others belong to low level statements; some clauses are evaluated before the queries are started, some others are evaluated during query execution. For these reasons, the set of available objects depends on the context in which an expression or a condition appears.

In this section we present the formal grammar of expressions, the complete set of basic blocks and the rules to know which blocks are available in a given context.

Although expressions and conditions are conceptually quite different for the user, they are both described with the same grammar productions. In fact, a condition is just an expression whose value is a Boolean data type. The grammar defines the set of operators that are allowed into expressions and fix their priorities, but doesn't perform type checking: a query that is valid from the grammar point of view can contain type errors. As described in Appendix A, these errors are handled by the parser, but not taking advantage of grammar features.

69. <Condition> →  
 <Expression>

| OPERATORS | DESCRIPTION                   |
|-----------|-------------------------------|
| OR        | Boolean disjunction           |
| AND       | Boolean conjunction           |
| XOR       | Boolean exclusive disjunction |
| NOT       | Boolean negation              |
| IS NULL   | Is null value                 |
| BETWEEN   | Between an inclusive range    |
| LIKE      | Search for a pattern          |
| <         | Less than                     |
| >         | Greater than                  |
| <=        | Less than or equal            |
| >=        | Greater than or equal         |
| <>        | Not equal                     |
| !=        | Not equal                     |
|           | Bitwise disjunction           |
| &         | Bitwise conjunction           |
| ^         | Bitwise exclusive disjunction |
| <<        | Shift left                    |
| >>        | Shift right                   |
| !         | Bitwise negation              |
| +         | Addition                      |
| -         | Subtraction                   |
| *         | Multiplication                |
| /         | Division                      |
| ±         | Sign                          |
| ()        | Round brackets                |

Priority

Figure 25: Set of supported operators

We now report the grammar of expressions, starting from the non terminal *Expression* and entering into the smallest details until reaching basic blocks. In this way, operators are introduced in increasing priority order (Figure 25).

The first introduced operators' class is that of Logical operators. Note that, although we use the word "*Boolean*" for non terminals, these operators are intended to work with a three valued logic.

```

70. <Expression> →
 <Expression Boolean Term> { OR <Expression Boolean Term> }*

71. <Expression Boolean Term> →
 <Expression Boolean Factor> { AND <Expression Boolean Factor>
})*

72. <Expression Boolean Factor> →
 <Expression Boolean Test> { XOR <Expression Boolean Test> }*

73. <Expression Boolean Test> →
 { NOT }* <Expression Boolean Primary>
 [IS [NOT] { TRUE | FALSE | UNKNOWN }]

74. <Expression Boolean Primary> →
 <Expression Bit>
 [{ <Comparison Operator> <Expression Bit> |
 [IS [NOT] NULL] |
 BETWEEN <Expression Bit> AND <Expression Bit> |
 LIKE <String Constant> }]

75. <Comparison Operator> →
 { '<' | '>' | '=' | '>=' | '<=' | '<>' | '!=' }

```

The following productions introduce bitwise operators.

```

76. <Expression Bit> →
 { <Expression Bit Term> { '|' <Expression Bit Term> }

77. <Expression Bit Term> →
 <Expression Bit Factor> { '&' <Expression Bit Factor> }*

78. <Expression Bit Factor> →
 <Expression Bit Shift> { '^' <Expression Bit Shift> }*

79. <Expression Bit Shift> →
 <Expression Bit Test> [{ '<<' | '>>' } <Integer Constant>]

80. <Expression Bit Test> →
 { '!' }* <Expression Value>
 [IS [NOT] { TRUE | FALSE | UNKNOWN }]

```

The following productions introduce standard algebraic operators.

```

81. <Expression Value> →
 <Expression Value Term>
 { { '+' | '-' } <Expression Value Term> } *

82. <Expression Value Term> →
 <Expression Value Factor>
 { { '*' | '/' } <Expression Value Factor> } *

83. <Expression Value Factor> →
 ['+' | '-'] <Expression Value Primary>

```

“*Expression Value Primary*” non terminal represents what we have previously called “*basic block*” of the expression. As said before, the grammar productions that defines this non terminal depends on the context in which the expression is used. In order to model this feature with the grammar, the whole set of productions describing an expression should be replicated once for each possible context, changing only the definition of the “*Expression Value Primary*” non terminal. To reduce grammar complexity and improve readability we decided to define that non terminal once, allowing every possible choice in all the contexts. In the following we will report a table that precisely shows which choices are valid in each context.

```

84. <Expression Value Primary> →
 { ID | GROUP_TS |
 <Logical Object Field> |
 <Pilot Join Field> |
 <Window Field> |
 <Constant> |
 <Function Call> |
 <Exists Attribute> |
 <LL Aggregate> |
 <HL Aggregate>
 }

```

*ID* and *GROUP\_TS* are special fields whose meaning was described above. A “*Logical Object Field*” is the name of a logical object attribute and, then, it is defined as an identifier. A “*Pilot Join Field*” is the name of a column of a data structure used to drive the *pilot join* operation. The user must specify such a field indicating both the data structure name and the field name, separated with a dot. A “*Window field*” is a column of a window extracted from a data structure: it can be referred both with an identifier and using the dot notation (in this case the identifier on the left of dot is the alias of the window).

```

85. <Logical Object Field> →
 <Identifier>

86. <Pilot Join Field> →
 <Data Structure Name> \.' <Data Structure Field>

87. <Window Field> →
 [<Window Alias> \.'] <Data Structure Field>

88. <Data Structure Name> →
 <Identifier>

89. <Data Structure Field> →
 <Identifier>

90. <Window Alias> →
 <Identifier>

```

“*Constant*” non terminal refers to numeric, logic, string or *NULL* values and it will be defined in the next section. It is worth noticing that numeric values are unsigned because their sign is already defined with the unary operators (“+” or “-”) of production (83).

“*Function Call*” non terminal has been introduced to allow the user calling an external function from our language (e.g.: string functions: *TRIM*, *LENGTH*, etc). A function can have an arbitrary number of parameters and the language grammar allows passing arbitrary expressions to these parameters. Obviously, a real implementation of the language should define the supported set of external functions and provide their implementations.

```

91. <Function Call> →
 <Identifier> \(' [<Function Parameter List>] \)'

92. <Function Parameter List> →
 <Expression> { \,' <Expression> } *

```

“*Exist Attribute*” is a predefined function that allows to inspect the logical object structure. As said before, it takes the name of a field as the only parameter and returns a Boolean value indicating if the specified field exists or not in the logical object that is executing the query.

```

93. <Exists Attribute> →
 EXISTS \(' { <Logical Object Field> | ALL } \)'

```

Figure 26 reports a table that specifies which of the previous basic blocks can be used in each context: four different sets of basic blocks are identified. Expressions (and conditions) that appears as parameter of aggregates or functions are not reported in the table: the rule that has to be applied in this case is that these expressions should comply with the same constraints of the expression in which the aggregate or the function is contained. For example, a low level

aggregate cannot be part of an expression that is used as a parameter of a function contained in the *EXECUTE IF* clause, but the same aggregate can be used if the function is contained in the *SELECT* clause.

|                                                                           |                    | Low Level SELECT Clause | Low Level HAVING Clause | Low Level WHERE Clause | Low Level IF-EVERY Clause | Low Level PILOT JOIN Clause | Low Level EXECUTE IF Clause | Actuation ON Clause | Actuation SET Clause | High Level SELECT Clause | High Level WHERE CLAUSE | High Level HAVING Clause |
|---------------------------------------------------------------------------|--------------------|-------------------------|-------------------------|------------------------|---------------------------|-----------------------------|-----------------------------|---------------------|----------------------|--------------------------|-------------------------|--------------------------|
|                                                                           |                    | Expr.                   | Expr.                   | Cond.                  | E/C                       | Cond.                       | Cond.                       | Cond.               | Expr.                | Expr.                    | Cond.                   | Cond.                    |
| Logical Object Identifier                                                 | ID                 | ✓                       |                         |                        | ✓                         |                             |                             | ✓                   |                      |                          | ✗                       |                          |
| Low Level Group Timestamp                                                 | GROUP_TS           | ✓                       |                         |                        | ✗                         |                             |                             | ✗                   |                      |                          | ✗                       |                          |
| Logical Object Attribute (Static – Dynamic Probing – Dynamic Non Probing) | LogicalObjectField | ✓                       |                         |                        | ✓                         |                             |                             | ✓                   |                      |                          | ✗                       |                          |
| Pilot Join Data Structure Field                                           | PilotJoinField     | ✓                       |                         |                        | ✓                         |                             |                             | ✗                   |                      |                          | ✗                       |                          |
| Window Field                                                              | WindowField        | ✗                       |                         |                        | ✗                         |                             |                             | ✗                   |                      |                          | ✓                       |                          |
| Constant Value                                                            | Constant           | ✓                       |                         |                        | ✓                         |                             |                             | ✓                   |                      |                          | ✓                       |                          |
| External Function Call                                                    | FunctionCall       | ✓                       |                         |                        | ✓                         |                             |                             | ✓                   |                      |                          | ✓                       |                          |
| Exists Function                                                           | Exists Attribute   | ✓                       |                         |                        | ✓                         |                             |                             | ✓                   |                      |                          | ✗                       |                          |
| Low Level Aggregation With / Without Condition                            | LLAggregate        | ✓                       |                         |                        | ✗                         |                             |                             | ✗                   |                      |                          | ✗                       |                          |
| High Level Aggregation With / Without Condition                           | HLLAggregate       | ✗                       |                         |                        | ✗                         |                             |                             | ✗                   |                      |                          | ✓                       |                          |

Figure 26: Table of allowed expressions in different contexts

## 5.8 Constants and identifiers

For completeness, in this section we report a set of productions representing simple grammar rules used to define the valid syntax of identifiers, numerical and literal values, etc. Productions (94), (95), (96) and (97) define the sets of characters that can be used:

```

94. <Digit> →
 { '0' | '1' .. '9' }

95. <Literal> →
 { 'A' .. 'Z' | 'a' .. 'z' }

96. <Non Single Quote Char> →
 # all printable ASCII Chars, except single quote (') #

97. <Non Double Quote Char> →
 # all printable ASCII Chars, except double quote (") #

```

The following grammar rules define the syntax of identifiers in a classical way: an identifier starts with a literal or underscore and is composed only of literals, digits and underscores. Logical object fields, logical object events, data structure fields and data structure names are just identifiers. We introduced all of them to produce a clearer grammar.

```

98. <Identifier> →
 { <Literal> | '_' } { <Literal> | <Digit> | '_' } *

99. <Logical Object Field> →
 <Identifier>

```

Production (100) lists all the supported data types. Note that in this version of the language we have not introduced a *DataTime* type to limit the grammar complexity. In fact, this kind of data type is not absolutely necessary due to the existence of timestamp values.

```

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

```

The following productions define the syntax of valid constant values for each data type. The “*Signed Constant*” non terminal represents the complete set of values that can be used in the language.

```

101. <Boolean Constant> →
 { TRUE | FALSE }

102. <Integer Constant> →
 { <Digit> } +

103. <Float Constant> →
 { <Integer Constant> ['.' <Integer Constant>] |
 '.' <Integer Constant> } ['E' { '+' | '-' } <Integer Constant>]

```

```

104. <String Constant> →
 { <Single Quoted String Value> |
 <Double Quoted String Value> }

105. <Single Quoted String Value> →
 { ''' { <Non Single Quote Char> | '''' }* ''' }

106. <Double Quoted String Value> →
 { '"' { <Non Double Quote Char> | '\"' }* '"' }

107. <Null Constant> →
 NULL

108. <Signed Constant> →
 { <Boolean Constant> |
 ['+' | '-'] <Integer Constant> |
 ['+' | '-'] <Float Constant> |
 <String Constant> |
 <Null Constant> }

109. <Constant> →
 { <Boolean Constant> |
 <Integer Constant> |
 <Float Constant> |
 <String Constant> |
 <Null Constant> }

```

In many parts of the language the concept of duration appears: it is a period of time that can be expressed with a floating point value followed by a time unit.

```

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

111. <Duration> →
 <Float Constant> <Time Unit>

```

## 5.9 Summary of the grammar

A summary of the grammar presented in this chapter is reported in the following. The notation used for this summary is not as formal and precise as an *EBNF* grammar can be, but it is very intuitive and gives an overall view of the language. A full *EBNF* definition is reported in the Appendix B.

Keywords are written with a bold font, while fields that should be completed by the user are enclosed within angle brackets. Square brackets indicate that a clause is optional, and a sequence of three dots indicates that a user can insert a list of elements. Underlined keywords are used to indicate the default option, when an optional clause is not specified.



### 5.9.1 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>
```

### 5.9.2 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
```

### 5.9.3 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
```

### 5.9.4 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 }]

```

### 5.9.5 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
```

### 5.9.6 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
```

### 5.9.7 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>]
```

```
]
```

### 5.9.8 Actuation query level statement

```
SET <Logical Object Field> = <Expression>, ...
```

```
ON <Expression>
```

## 6 Examples of queries

In this chapter we present some queries written with *PerLa*. These queries are intended to explain all the features of the language.

For each example, we will report the request of the user, the interfaces of the involved logical objects, the code of the query and some comments to describe the behavior and to clarify possible doubts.

The first examples are very simple and are intended to exactly explain the sampling operation, highlighting the differences between event based and time based semantics. Query complexity grows in the next examples, where other features of the language are introduced. The last section of the chapter reports some examples related to a real case study.

In this chapter logical objects interfaces are described using a table, composed of four columns: the field name, the data type, the field type and a short description. Field types are indicated using the abbreviations reported in Figure 27.

| Field Type                    | Abbreviation |
|-------------------------------|--------------|
| ID                            | ID           |
| Static attribute              | S            |
| Non probing dynamic attribute | NP           |
| Probing dynamic attribute     | P            |
| Event                         | E            |

Figure 27: Abbreviations for allowed field types of logical objects

In many examples the requirements refer to some parameters (*e.g.*: the *ID* of a logical object, a threshold, etc.). In the queries these parameters are enclosed within square brackets, to indicate that a real value should be provided before queries are submitted to the query analyzer.

### 6.1 Low Level queries with event based sampling

#### 6.1.1 Example 1

Get the list of *RFID* readers that sense the passive tag with *ID* [tag], specifying also the transit timestamp.

| RFID tag                                  |           |            |                                                   |
|-------------------------------------------|-----------|------------|---------------------------------------------------|
| Logical object wrapping a single RFID tag |           |            |                                                   |
| Field Name                                | Data Type | Field type | Description                                       |
| ID                                        | ID        | ID         | Logical object identifier                         |
| lastReaderID                              | ID        | NP         | Cached ID of the last reader that sensed the tag  |
| lastReaderChanged                         | -         | E          | Notifies that the tag has been sensed by a reader |

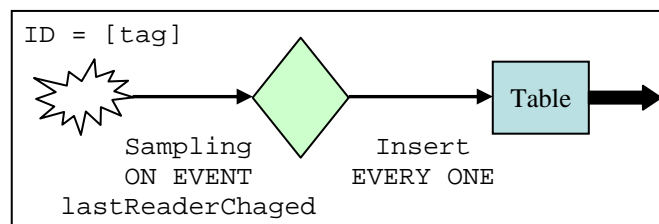
*Query*

```
CREATE OUTPUT STREAM Table (readerID ID, ts TIMESTAMP) AS
LOW:
 EVERY ONE
 SELECT lastReaderID, GROUP_TS
 SAMPLING ON EVENT lastReaderChanged
 EXECUTE IF ID = [tag]
```

*Alternative query*

```
CREATE OUTPUT STREAM Table (readerID ID, ts TIMESTAMP)

INSERT INTO STREAM Table (readerID, ts)
LOW:
 EVERY ONE
 SELECT lastReaderID, GROUP_TS
 SAMPLING ON EVENT lastReaderChanged
 EXECUTE IF ID = [tag]
```



The *EXECUTE IF* condition is specified in order to force the low level query to be executed only on the logical object wrapping the *RFID* tag having the specified *ID*. Whenever that tag is physically sensed by an *RFID* reader, the logical object updates the *lastReaderId* field and raises the *lastReaderChanged* event. The *SAMPLING* clause forces the query engine to be attached to that event and to perform a sampling operation whenever the event is raised.

Since a *WHERE* clause is not specified after the *SAMPLING* one, every sampling operation causes the insertion of a new record in the local buffer. The *EVERY ONE* clause means that the selection on the local buffer is performed once for each inserted record. Every time the *SELECT* clause is computed, the most recent record in the local buffer is inserted in the output stream: obviously, the *lastReaderId* field of that record contains the *ID* of the reader that sensed the tag.

Then, the behavior of the query is that a reader *ID* is immediately returned to the user every time the tag is sensed.

### 6.1.2 Example 2

Get the list of *RFID* tags that are sensed by the *RFID* reader with *ID* [reader], specifying also the transit timestamp.

| RFID tag                                  |           |            |                                                   |
|-------------------------------------------|-----------|------------|---------------------------------------------------|
| Logical object wrapping a single RFID tag |           |            |                                                   |
| Field Name                                | Data Type | Field type | Description                                       |
| ID                                        | ID        | ID         | Logical object identifier                         |
| lastReaderID                              | ID        | NP         | Cached ID of the last reader that sensed the tag  |
| lastReaderChanged                         | -         | E          | Notifies that the tag has been sensed by a reader |
| deviceType                                | STRING    | S          | Type of device                                    |

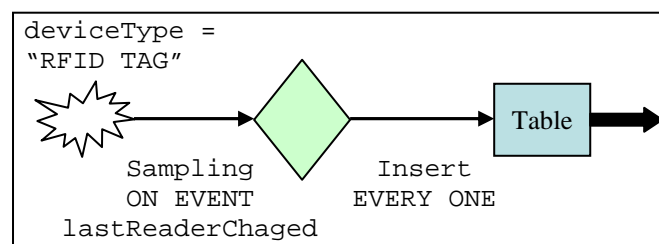
#### Query

```
CREATE OUTPUT STREAM Table (tagID ID, ts TIMESTAMP) AS
LOW:
 EVERY ONE
 SELECT ID, GROUP_TS
 SAMPLING ON EVENT lastReaderChanged
 WHERE lastReaderID = [reader]
 EXECUTE IF deviceType = "RRFID_TAG"
```

#### Alternative Query

```
CREATE OUTPUT STREAM Table (tagID ID, ts TIMESTAMP)

INSERT INTO STREAM Table (tagID, ts)
LOW:
 EVERY ONE
 SELECT ID, GROUP_TS
 SAMPLING ON EVENT lastReaderChanged
 WHERE lastReaderID = [reader]
 EXECUTE IF deviceType = "RFID_TAG"
```



This query requires to monitor an *RFID* reader and to report all the tags that are sensed by that reader. Suppose that each tag in the system is wrapped by a logical object and that no logical object wraps the *RFID* reader.

To reach the goal, the low level query must be executed on all logical objects wrapping a tag: whenever they are sensed by the reader, they send a record to the output stream.

The *EXECUTE IF* clause limits the set of logical objects that will take part to the query performing a filtering on the *deviceType* attribute: in this way a node that is not a tag (e.g.: a *WSN* node) is excluded.

The *WHERE* clause is needed because the *lastReaderChanged* event is raised whenever a tag is sensed by any of the *RFID* readers in the system, but we are interested only in a specific reader. Note that, when a different reader senses the tag, the *WHERE* clause is not satisfied, no new record is inserted in the local buffer and then the *SELECT* clause is not computed. On the opposite, when the *WHERE* condition is satisfied, a new record is inserted into the buffer and the behavior is the same as the previous query: the only difference is that, in this example, the *ID* of the logical object is returned instead of the last sensed reader *ID*.

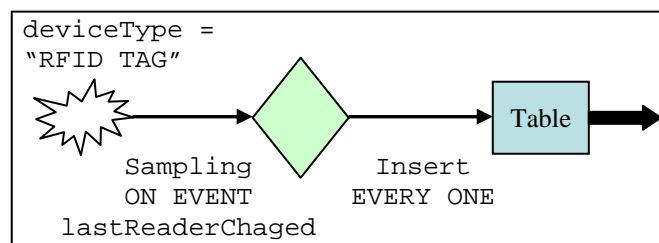
### 6.1.3 Example 3

Get the list of *RFID* tags that are sensed by the *RFID* reader with *ID* [reader], only if they had been already viewed at least once in the last *30 seconds*.

| RFID tag                                  |           |            |                                                   |
|-------------------------------------------|-----------|------------|---------------------------------------------------|
| Logical object wrapping a single RFID tag |           |            |                                                   |
| Field Name                                | Data Type | Field type | Description                                       |
| ID                                        | ID        | ID         | Logical object identifier                         |
| lastReaderID                              | ID        | NP         | Cached ID of the last reader that sensed the tag  |
| lastReaderChanged                         | -         | E          | Notifies that the tag has been sensed by a reader |
| deviceType                                | STRING    | S          | Type of device                                    |

### Query

```
CREATE OUTPUT STREAM Table (tagID ID) AS
LOW:
 EVERY ONE
 SELECT ID
 HAVING COUNT(*, 30 s) >= 2
 SAMPLING ON EVENT lastReaderChanged
 WHERE lastReaderID = [reader]
 EXECUTE IF deviceType = "RFID_TAG"
```





This query is very similar to the previous one: the only difference is the specification of the *HAVING* clause, needed to perform the required control on the last 30 seconds. When a certain tag “*T*” is sensed by the reader “*reader*” and the *WHERE* condition is satisfied, a record is appended to the local buffer and the selection is computed. Then, the *HAVING* clause is applied: the records produced during the last 30 seconds are counted. If the result of this operation is equal to one, no record is inserted in the output data structure because the tag was not sensed a second time during the last 30 seconds.

Note that a real implementation of the query executor should maintain a 30 seconds window of the ideally unbounded buffer.

The condition of the *WHERE* clause can be moved into the *HAVING* clause, without changing its semantics:

```
CREATE OUTPUT STREAM Table (tagID ID) AS
LOW:
 EVERY ONE
 SELECT ID
 HAVING lastReaderID = [reader]
 AND COUNT(*, 30 s, lastReaderId = [reader]) >= 2

 SAMPLING ON EVENT lastReaderChanged
 EXECUTE IF deviceType = "RFID_TAG"
```

With this version of the query, whenever the tag “*T*” is sensed by any reader “*R*” in the system, a new record is appended to the local buffer. The *EVERY ONE* clause causes the activation of the selection: the record to be returned is filtered with the conditions specified in the *HAVING* clause. If “*R*” is not “*reader*” or “*R*” is “*reader*”, but the count is less than two, the record is discarded and the set of records generated by the selection is empty. Since the clause *ON EMPTY SELECTION* is not specified, the default option *SEND NOTHING* is used and no record is inserted in the output data structure.

This approach is less human readable than the previous one and it requires the insertion of a record in the local buffer whenever the tag is sensed by a reader, even if this record is useless.

#### 6.1.4 Example 4

Perform a *10 minutes* monitoring of the tag with *ID* [tag]. At the end of this period, insert a single record in an output stream to report how many times the tag was sensed by the reader with *ID* [reader].

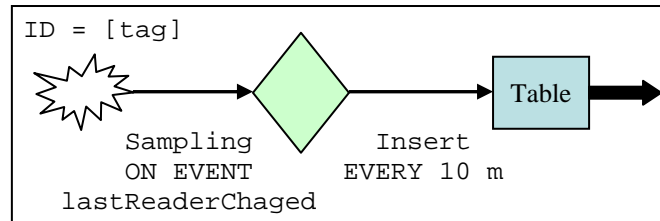
| RFID tag                                  |           |            |                                                   |
|-------------------------------------------|-----------|------------|---------------------------------------------------|
| Logical object wrapping a single RFID tag |           |            |                                                   |
| Field Name                                | Data Type | Field type | Description                                       |
| ID                                        | ID        | ID         | Logical object identifier                         |
| lastReaderID                              | ID        | NP         | Cached ID of the last reader that sensed the tag  |
| lastReaderChanged                         | -         | E          | Notifies that the tag has been sensed by a reader |

*Query*

```

CREATE OUTPUT STREAM Table (counter INTEGER) AS
LOW:
 EVERY 10 m
 SELECT COUNT(*, 10 m) DEFAULT 0
 ON EMPTY SELECTION INSERT DEFAULT
 SAMPLING ON EVENT lastReaderChanged
 WHERE lastReaderID = [reader]
 EXECUTE IF ID = [tag]
 TERMINATE AFTER 1 SELECTIONS

```



This example is a one shot query: it requires the monitoring of a tag for a short interval and, then, it terminates. Note that all the queries in the previous examples do not specify the *TERMINATE* clause and they continuously run until the user explicitly stops them.

The execution conditions and the selection section of this query are the same as previous ones. So we focus our description only on the data management section.

Although the sampling operation is still event based, differently from previous examples, the selection process is time based and it is activated 10 minutes after the query starts. The local buffer records that are relative to the last 10 minutes are counted, and the obtained result is inserted in the output stream. At this point, the query is stopped because the *TERMINATE AFTER* clause requires the execution of the selection process only once.

A particular remark is needed to explain the query behavior when the tag is never sensed during the monitoring period. In this case, when the selection is computed, the local buffer is empty and no record can be returned although the *UP TO* clause is implicitly set to *ONE*. Since the *INSERT DEFAULT* option of the *ON EMPTY SELECTION* clause is specified, a default record is generated and appended to the output stream. The *counter* field of that record will be set to the value *0*, as specified with the *DEFAULT* keyword in the *SELECT* clause.

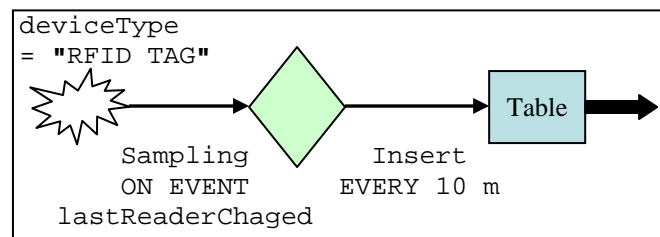
**6.1.5 Example 5**

Every *10 minutes* get a set of records, each of them related to one of the *10 minutes*. All the tags that were sensed by the reader with *ID* [reader] at least three times during a minute must be returned, indicating also the final timestamp and how many times the tag was seen.

| RFID tag                                  |           |            |                                                   |
|-------------------------------------------|-----------|------------|---------------------------------------------------|
| Logical object wrapping a single RFID tag |           |            |                                                   |
| Field Name                                | Data Type | Field type | Description                                       |
| ID                                        | ID        | ID         | Logical object identifier                         |
| lastReaderID                              | ID        | NP         | Cashed ID of the last reader that sensed the tag  |
| lastReaderChanged                         | -         | E          | Notifies that the tag has been sensed by a reader |
| deviceType                                | STRING    | S          | Type of device                                    |

### Query

```
CREATE OUTPUT STREAM Table (tagID ID, ts TIMESTAMP, counter INTEGER) AS
LOW:
 EVERY 10 m
 SELECT ID, GROUP_TS, COUNT(*, 1 m)
 GROUP BY TIMESTAMP(1 m, 10 GROUPS)
 HAVING COUNT(*, 1 m) >= 3
 SAMPLING ON EVENT lastReaderChanged
 WHERE lastReaderID = [reader]
 EXECUTE IF deviceType = "RFID_TAG"
```



The only feature of the language used in this query and not yet explained in the previous examples is the *timestamp grouping*. Every ten minutes, when the selection is computed, the content of the local buffer is logically replicated in ten copies (10 *GROUPS*). The first replica contains all the records having a native timestamp less or equal to the current timestamp; the second replica contains all the records having a native timestamp relative to at least one minute before; and so on, until the tenth replica that contains data with a native timestamp relative to at least nine minutes before.

At this point the *HAVING* clause is applied to each replica of the buffer: records that are not discarded are appended to the *Table* stream.

## 6.2 Low level queries with time based sampling

### 6.2.1 Example 6

Get the *ID* and the temperature of all temperature sensors placed in the area *Z*, sampling every *30 seconds* and assuming that all sensors are fixed.

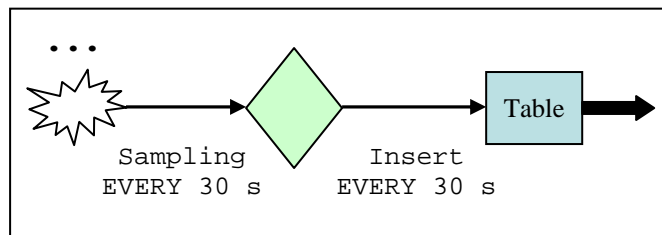
| WSN node                                                                     |           |            |                                |
|------------------------------------------------------------------------------|-----------|------------|--------------------------------|
| Logical object wrapping a single WSN node equipped with a temperature sensor |           |            |                                |
| Field Name                                                                   | Data Type | Field type | Description                    |
| ID                                                                           | ID        | ID         | Logical object identifier      |
| temp                                                                         | FLOAT     | P          | Temperature sensor             |
| locationX                                                                    | FLOAT     | S          | Sensor location – X coordinate |
| locationY                                                                    | FLOAT     | S          | Sensor location – Y coordinate |
| deviceType                                                                   | STRING    | S          | Type of device                 |

### Query

```

CREATE OUTPUT STREAM Table (sensorID ID, temperature FLOAT) AS
LOW:
 EVERY 30 s
 SELECT ID, temp
 SAMPLING
 EVERY 30 s
 EXECUTE IF deviceType = "SENSOR"
 AND is_in_zone_Z(locationX, locationY)
 AND EXISTS (temp)

```



This simple query introduces the time based sampling: the nodes involved in the query execution are sampled every 30 seconds (*SAMPLING EVERY 30 s*). The selection section of the query is activated with the same frequency, obtaining in this way a semantics that is exactly equivalent to requiring the sampling operation whenever a record is inserted in the local buffer (*EVERY ONE*). Note that this particular circumstance, in which the time based activation is equivalent to the event based one, is not usually true. In fact, for example, the two semantics become different just appending a *WHERE* condition to the *SAMPLING* clause: when a sampled record is discarded, the selection is computed only if the time based activation is used. Some considerations about the *EXECUTE IF* clause are needed. First of all, the query properly works only under the condition that temperature sensors are fixed or cannot enter or exit the area *Z*. In fact, if this condition is not satisfied, a node that is in the area *Z* while the *EXECUTE IF* clause is evaluated, keeps on sampling also after exiting the area. Vice versa, a node that is not in the area *Z* while the *EXECUTE IF* clause is evaluated, will not start sampling even if it will enter the area. The simplest way to handle moving nodes (*i.e.* nodes whose *locationX* and *locationY* logical object fields are dynamic) is to add a *REFRESH* clause to the *EXECUTE IF* condition.

The second consideration about the *EXECUTE IF* clause is relative to the evaluation of the condition when some logical objects do not expose the *locationX* or the *locationY* attributes: in this case *NULL* values are passed to the “*is\_in\_zone\_Z*” function. If this external routine returns *FALSE* the *EXECUTE IF* condition is not satisfied, while if it returns *NULL* the condition becomes *UNKNOWN*, and the *EXECUTE IF* is still not satisfied.

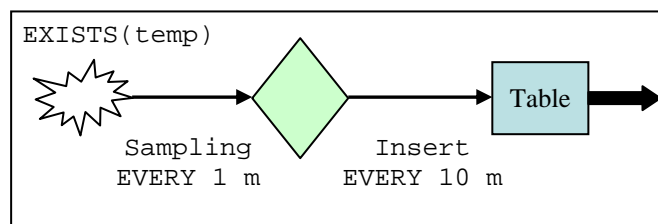
### 6.2.2 Example 7

Every minute, sample the temperature and the pressure on the nodes having at least a temperature sensor on board. Then, after 10 minutes, return the record (or the records) having the maximum temperature value.

| WSN node                                                                     |           |            |                           |
|------------------------------------------------------------------------------|-----------|------------|---------------------------|
| Logical object wrapping a single WSN node equipped with a temperature sensor |           |            |                           |
| Field Name                                                                   | Data Type | Field type | Description               |
| ID                                                                           | ID        | ID         | Logical object identifier |
| temp                                                                         | FLOAT     | P          | Temperature sensor        |
| pressure                                                                     | FLOAT     | P          | Pressure sensor           |

#### Query

```
CREATE OUTPUT STREAM Table (sensorID ID, temperature FLOAT, pressure FLOAT) AS
LOW:
 EVERY 10 m
 SELECT ID, temp, pressure
 HAVING temp = MAX (temp, 10 m)
 UP TO 10 m
 SAMPLING
 EVERY 1 m
 EXECUTE IF EXISTS (temp)
```



The language feature introduced with this example is the *UP TO* clause, that is needed because the results that have to be returned are not aggregate values. The *HAVING* clause is used to filter the records considered by the *UP TO* clause, which do not have the maximum temperature value.

Note that the *EXECUTE IF* condition requires only the existence of the *temp* attribute. If the logical object provides this attribute, but not the *pressure* one, it will participate to the query and it will return a *NULL* value when a pressure sample is required.

## 6.3 Low level and High level queries

### 6.3.1 Example 8

Every 30 seconds get the average temperature of the sensors placed in the area Z sampling them depending on the following criteria:

- High power level: sample every 10 seconds and returns the average of 3 values
- Medium power level: sample every 30 seconds
- Low power level: don't sample (don't participate in the query)

Sampling rates should be updated once every hour.

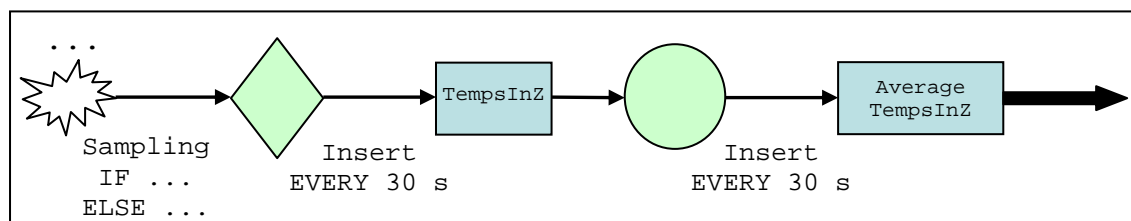
| WSN node                                                                     |           |            |                                                                |
|------------------------------------------------------------------------------|-----------|------------|----------------------------------------------------------------|
| Logical object wrapping a single WSN node equipped with a temperature sensor |           |            |                                                                |
| Field Name                                                                   | Data Type | Field type | Description                                                    |
| ID                                                                           | ID        | ID         | Logical object identifier                                      |
| temp                                                                         | FLOAT     | P          | Sampled temperature                                            |
| locationX                                                                    | FLOAT     | S          | Sensor location – X coordinate                                 |
| locationY                                                                    | FLOAT     | S          | Sensor location – Y coordinate                                 |
| powerLevel                                                                   | FLOAT     | P          | Current power levels of the devices expressed in a 1-100 scale |

#### Query

```
CREATE STREAM TempsInZ (temperature FLOAT) AS
LOW:
 EVERY 30 s
 SELECT AVG(temp, 30 s)
 SAMPLING
 IF powerLevel > 0.70 EVERY 10 s
 ELSE EVERY 30 s
 REFRESH EVERY 1 h

EXECUTE IF
 EXISTS (ALL) AND is_in_zone_Z(locationX,locationY) AND powerLevel > 0.3
 REFRESH EVERY 1 h
```

```
CREATE OUTPUT STREAM AverageTempsInZ (temperature FLOAT) AS
HIGH:
 EVERY 30 s
 SELECT AVG(temperature)
 FROM TempsInZ (30 s)
```



This example is solved using a low level statement to specify the sampling parameters and, if needed, to perform the time average of data collected by a single node. The required spatial average is then computed by an apposite high level query.

It is worth noticing how the sampling frequency is specified in terms of the current power level. If it is low ( $< 30\%$ ) the *EXECUTE IF* condition become *FALSE* and the logical object do not participate to the query. Otherwise, the sampling condition is evaluated to establish the sampling rate: if the power level is high ( $> 70\%$ ) the interval between two subsequent sampling operations is set to 10 seconds, else it is set to 30 seconds.

The correct behavior of the query can be obtained using the *REFRESH EVERY* clause, in order to force the execution of the above described process once every hour. Note that this clause must be specified both for the *EXECUTE IF* and the *SAMPLING* clause: otherwise, only a part of the whole process will be executed. Suppose, for example, to require only the refresh of the *EXECUTE IF* clause: a device having a high power level will not reduce its sample rate when the remaining energy becomes lower than the high threshold. Only when the power level reaches the low threshold the *EXECUTE IF* clause will be reevaluated and the query will be stopped.

In the low level query the *AVG* aggregate, calculated on the last 30 seconds, performs the average of a different number of temperature samples depending on the current sampling rate: one record is used if the sampling time is equal to 30 seconds, while 3 records are used if the sampling is equal to 10 seconds.

The high level query is activated every 30 seconds and computes the spatial average of all the records arrived in the *TempsInZ* stream during the last 30 seconds: in this way the considered window certainly contains at most one record for each logical object that is executing the query.

### 6.3.2 Example 9

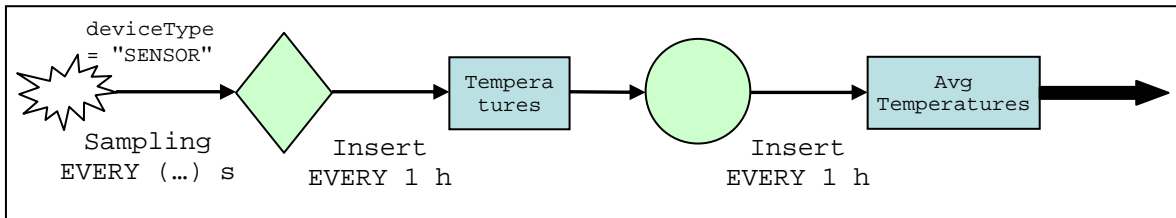
Every hour, get a record with the average temperature sensed at distances  $(0\text{ m} ; 5\text{ m})$ ,  $(5\text{ m} ; 10\text{ m})$ ,  $(10\text{ m} ; +\infty)$  from a fixed point  $P$ . Suppose that the temperature sensors are fixed, too. Use a greater sampling rate for sensors nearer to  $P$ .

| WSN node                                                                     |           |            |                                |
|------------------------------------------------------------------------------|-----------|------------|--------------------------------|
| Logical object wrapping a single WSN node equipped with a temperature sensor |           |            |                                |
| Field Name                                                                   | Data Type | Field type | Description                    |
| ID                                                                           | ID        | ID         | Logical object identifier      |
| temp                                                                         | FLOAT     | P          | Sampled temperature            |
| locationX                                                                    | FLOAT     | S          | Sensor location – X coordinate |
| locationY                                                                    | FLOAT     | S          | Sensor location – Y coordinate |
| deviceType                                                                   | STRING    | S          | Type of device                 |

### Query

```
CREATE STREAM Temperatures
 (distanceFromP FLOAT, TempSum FLOAT, TempCounter INTEGER) AS
LOW:
 EVERY 1 h
 SELECT dist_from_P(locationX, locationY), SUM(temp, 1 h), COUNT(*, 1 h)
 SAMPLING EVERY (1000 / dist_from_P(locationX, locationY)) s
 EXECUTE IF deviceType = "SENSOR"
```

```
CREATE OUTPUT STREAM AvgTemperatures
 (temp0_5 FLOAT, temp5_10 FLOAT, temp10_inf FLOAT) AS
HIGH:
 EVERY 1 h
 SELECT
 SUM(TempSum, distanceFromP < 5) / SUM (TempCont, distanceFromP < 5),
 SUM (TempSum, distanceFromP > 5 AND distanceFromP < 10) /
 SUM (TempCont, distanceFromP > 5 AND distanceFromP < 10),
 SUM(TempSum, distanceFromP > 10) / SUM (TempCont, distanceFromP > 10)
 FROM Temperature (1 h)
```



The only interesting point to be noted about this query is that the sampling rate is defined through an expression: in this way the interval between two sampling operations is a function of sensors location. No *REFRESH* clause is specified after the *SAMPLING* clause because the nodes are assumed to be fixed and, then, the value of the sampling rate expression is not time dependent.

The goal of the query is to compute an average of all the temperature samples taken during an hour by all the sensors placed at a certain distance from the point *P*. The simplest way to reach this goal is appending all the sampled values to a stream and, then, computing their average using a high level query. The solution presented above uses a more efficient approach: a pre-aggregation of data collected by a single logical object is computed directly in the low level query. Thus, each logical object inserts only a record in the *Temperatures* stream every hour, independently of its distance from the point *P*. Note that this record does not contain directly an average, but a sum and a count, because the average operator is not distributive.



## 6.4 Queries with pilot join

### 6.4.1 Example 10

There is a set of tanks, containing some temperature sensors. A *HF-RFID* tag and a base station are mounted on each tank.

Whenever a tank is seen by the *RFID* reader with *ID* [reader], sample all the temperature sensors contained in that tank for a minute and return an average value for each sensor.

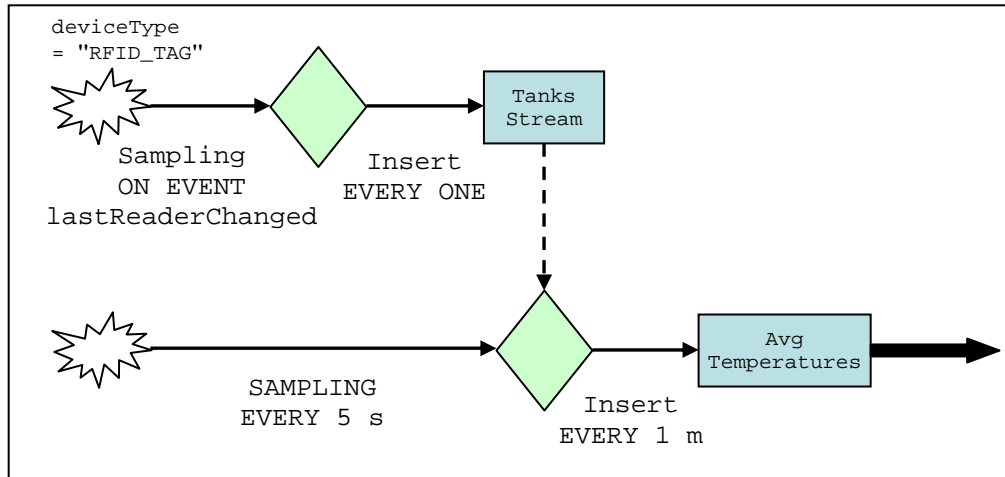
| HF-RFID tag                                  |           |            |                                                   |
|----------------------------------------------|-----------|------------|---------------------------------------------------|
| Logical object wrapping a single HI-RFID tag |           |            |                                                   |
| Field Name                                   | Data Type | Field type | Description                                       |
| ID                                           | ID        | ID         | Logical object identifier                         |
| lastReaderID                                 | ID        | NP         | Cashed ID of the last reader that sensed the tag  |
| lastReaderChanged                            | -         | E          | Notifies that the tag has been sensed by a reader |
| linkedBaseStationID                          | ID        | S          | ID of the base station mounted over the same tank |
| deviceType                                   | STRING    | S          | Type of device                                    |

| WSN node                                                                     |           |            |                                                               |
|------------------------------------------------------------------------------|-----------|------------|---------------------------------------------------------------|
| Logical object wrapping a single WSN node equipped with a temperature sensor |           |            |                                                               |
| Field Name                                                                   | Data Type | Field type | Description                                                   |
| ID                                                                           | ID        | ID         | Logical object identifier                                     |
| baseStationID                                                                | ID        | NP         | ID of the base station the WSN node is currently connected to |
| temp                                                                         | FLOAT     | P          | Sampled temperature                                           |

### Query

```
CREATE STREAM TanksStream (tagID ID, linkedBaseStationID ID) AS
LOW:
 EVERY ONE
 SELECT ID, linkedBaseStationID
 SAMPLING ON EVENT lastReaderChanged
 WHERE lastReaderID = [reader]
 EXECUTE IF deviceType = "RFID_TAG"
```

```
CREATE OUTPUT STREAM AvgTemperatures (baseStationID ID, temp FLOAT) AS
LOW:
 EVERY 1 m
 SELECT TankStream.baseStationID, AVG(temp, 1 m)
 SAMPLING EVERY 5 s
 PILOT JOIN TankStream ON TankStream.linkedBaseStationID = baseStationID
 TERMINATE AFTER 1 SELECTION
```



This example is intended to show the *pilot join* operation based on a stream. The first low level query is executed on each *RFID* tag and it inserts a record in *TankStream* whenever a tag is sensed by the reader with the *ID* [reader]. When this event happens, an instance of the second low level query is started on all the logical objects for which the *pilot join* condition is satisfied. This second query performs a sampling operation every 5 seconds, for a minute. Then, an average of the sampled temperatures is computed, a record is inserted in *AvgTemperatures* and the instance of the query is terminated.

#### 6.4.2 Example 11

There is a set of tanks, containing some temperature sensors. A *UHF-RFID* tag and a base station are mounted on each tank.

The temperature sensors contained in the tanks that are in the range area of the *RFID* reader with *ID* [reader] must be sampled once every minute.

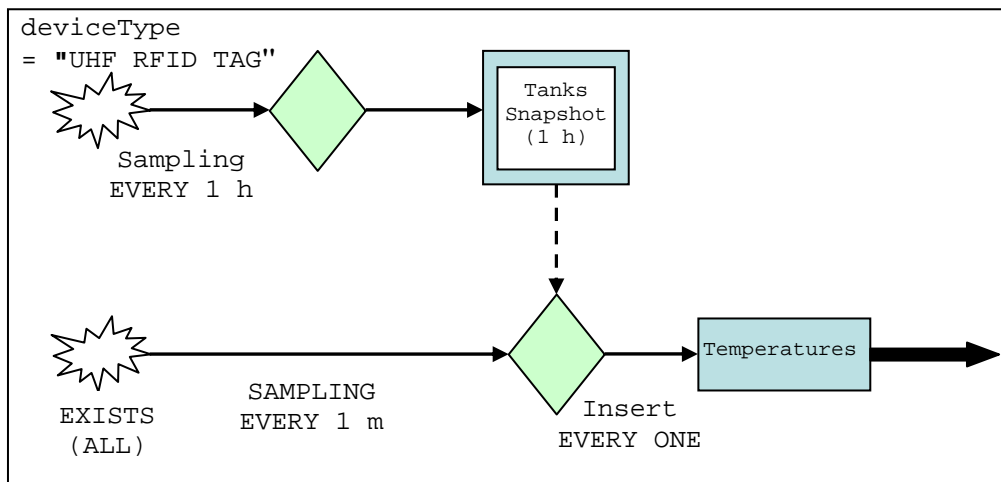
| UHF-RFID tag                                  |           |            |                                                    |
|-----------------------------------------------|-----------|------------|----------------------------------------------------|
| Logical object wrapping a single UHF-RFID tag |           |            |                                                    |
| Field Name                                    | Data Type | Field type | Description                                        |
| ID                                            | ID        | ID         | Logical object identifier                          |
| currentReaderID                               | ID        | P          | ID of the reader that is currently sensing the tag |
| linkedBaseStationID                           | ID        | S          | ID of the base station mounted over the same tank  |
| deviceType                                    | STRING    | S          | Type of device                                     |

| WSN node                                                                     |           |            |                                                               |
|------------------------------------------------------------------------------|-----------|------------|---------------------------------------------------------------|
| Logical object wrapping a single WSN node equipped with a temperature sensor |           |            |                                                               |
| Field Name                                                                   | Data Type | Field type | Description                                                   |
| ID                                                                           | ID        | ID         | Logical object identifier                                     |
| baseStationID                                                                | ID        | NP         | ID of the base station the WSN node is currently connected to |
| temp                                                                         | FLOAT     | P          | Sampled temperature                                           |

### Query

```
CREATE SNAPSHOT TanksSnapshot (tagID ID, linkedBaseStationID ID)
WITH DURATION 1 h AS
LOW:
 SELECT ID, linkedBaseStationID
 SAMPLING EVERY 1 h
 WHERE currentReaderID = [reader]
 EXECUTE IF deviceType = "UHF_RFID_TAG"
```

```
CREATE OUTPUT STREAM Temperatures (sensorID ID, temp FLOAT) AS
LOW:
 EVERY ONE
 SELECT ID, temp
 SAMPLING EVERY 1 m
 PILOT JOIN TanksSnapshot
 ON TanksSnapshot.linkedBaseStationID = baseStationID
 EXECUTE IF EXISTS (ALL)
```



This example is intended to show the *pilot join* operation based on a snapshot. The first low level query fills a snapshot data structure having a duration of one hour. The query is executed on all the logical objects wrapping *UHF-TAGs* and each of them produces at most one record for every snapshot interval.

The second low level query uses the previous snapshot to drive the *pilot join* operation: the sensor nodes, currently connected to a base station that is present in the snapshot list, execute

the query and sample their temperature sensor every minute. The *EXISTS (ALL)* is specified to guarantee that the nodes taking part to the query have a *baseStationID* attribute (*i.e.* they are *WSN* nodes) and a temperature attribute (*i.e.* they have a temperature sensor on board).

### 6.4.3 Example 12

There is a set of tanks, containing some temperature sensors. A *GPS* and a base station are mounted on each tank.

The temperature sensors contained in the tank nearest to point *P* must be sampled every *30 seconds*. The distances of the tanks from the point *P* must be revaluated every hour.

| GPS                                  |           |            |                                                   |
|--------------------------------------|-----------|------------|---------------------------------------------------|
| Logical object wrapping a GPS device |           |            |                                                   |
| Field Name                           | Data Type | Field type | Description                                       |
| ID                                   | ID        | ID         | Logical object identifier                         |
| linkedBaseStationID                  | ID        | S          | ID of the base station mounted over the same tank |
| locationX                            | FLOAT     | P          | Sensor location – X coordinate                    |
| locationY                            | FLOAT     | P          | Sensor location – Y coordinate                    |
| deviceType                           | STRING    | S          | Type of device                                    |

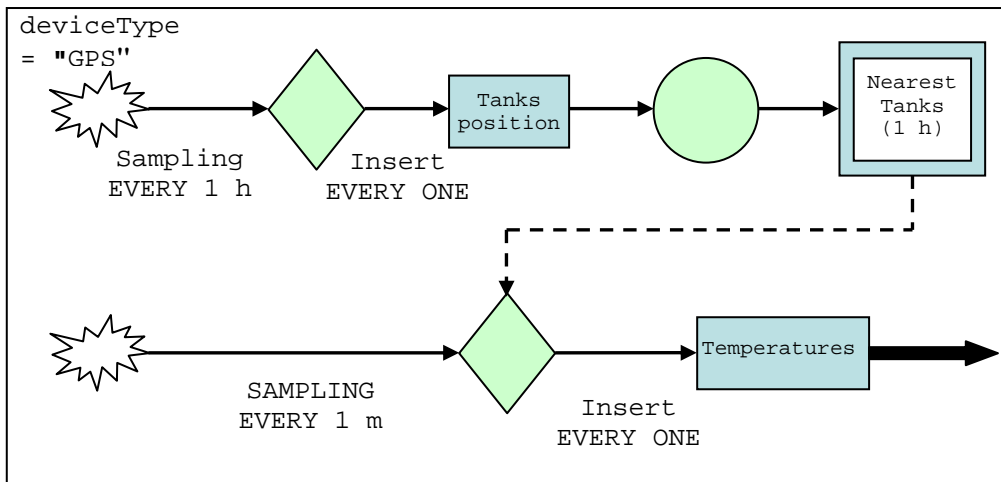
| WSN node                                                                     |           |            |                                                               |
|------------------------------------------------------------------------------|-----------|------------|---------------------------------------------------------------|
| Logical object wrapping a single WSN node equipped with a temperature sensor |           |            |                                                               |
| Field Name                                                                   | Data Type | Field type | Description                                                   |
| ID                                                                           | ID        | ID         | Logical object identifier                                     |
| baseStationID                                                                | ID        | NP         | ID of the base station the WSN node is currently connected to |
| temp                                                                         | FLOAT     | P          | Sampled temperature                                           |

### Query

```
CREATE STREAM TanksPositions (gpsID ID, linkedBaseStationID ID, distanceFromP
FLOAT) AS
LOW:
 EVERY ONE
 SELECT ID, linkedBaseStationID, dist_from_P(locationX, locationY)
 SAMPLING EVERY 1 h
 EXECUTE IF deviceType = "GPS"
```

```
CREATE SNAPSHOT NearestTank (gpsID ID, linkedBaseStationID ID)
WITH DURATION 1 h AS
HIGH:
 SELECT TanksPositions.gpsID, TanksPositions.linkedBaseStationID
 FROM TanksPositions (1 h)
 WHERE TanksPositions.distanceFromP = MIN(TanksPositions.distanceFromP)
```

```
CREATE OUTPUT STREAM Temperatures (sensorID ID, temp FLOAT) AS
LOW:
EVERY ONE
SELECT ID, temp
SAMPLING EVERY 1 m
PILOT JOIN NearestTank ON NearestTank.linkedBaseStationID = baseStationID
```



This query is very similar to the one in the previous example, with the only difference that a snapshot data structure is populated by a high level query instead of a low level one.

## 6.5 Actuation queries

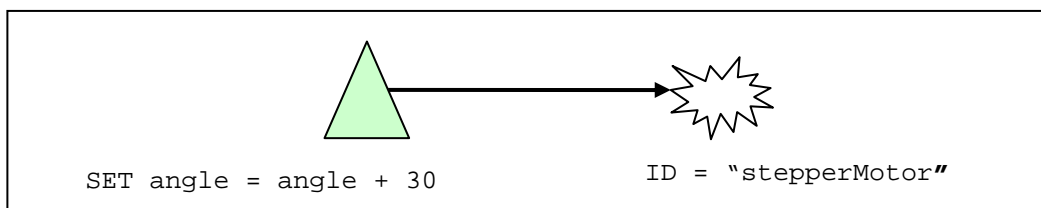
### 6.5.1 Example 13

Given a stepper motor, produce an anti clock wise rotation of thirty degrees starting from the current position.

| Stepper motor                             |           |            |                                               |
|-------------------------------------------|-----------|------------|-----------------------------------------------|
| Logical object wrapping the stepper motor |           |            |                                               |
| Field Name                                | Data Type | Field type | Description                                   |
| ID                                        | ID        | ID         | Logical object identifier                     |
| angle                                     | FLOAT     | P          | Current angular position of the stepper motor |

#### Query

```
SET angle = angle + 30 ON ID = stepperMotorID
```



This query is the simplest example of actuation query, used to control the value of a physical measure. Note that the *angle* logical object field is a probing attribute that can be both read or written: when it is read, the logical object returns the current stepper position; when it is written, the logical object interacts with the physical device to produce the required rotation.

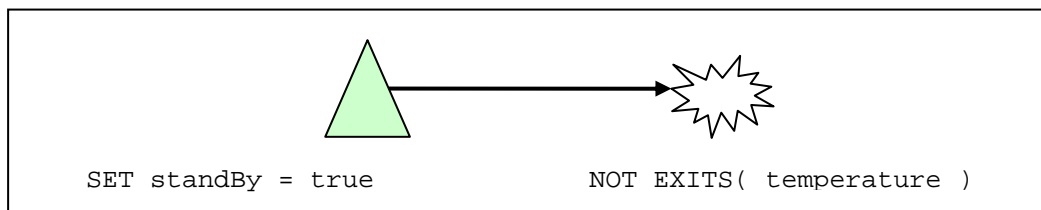
### 6.5.2 Example 14

Suppose that some devices in the pervasive system can be switched into stand-by status in order to reduce the power consumption when not used. Suppose also that the logical objects of these devices expose a writable *standBy* attribute to enable or disable the stand-by mode. The goal of the query shown in this example is to force the stand-by mode on all the devices that are not able to sample the temperature (*i.e.* that are not provided with the *temperature* attribute).

| Generic device                                                                                                                            |           |            |                           |
|-------------------------------------------------------------------------------------------------------------------------------------------|-----------|------------|---------------------------|
| Logical object wrapping a generic device.<br>* <i>standBy</i> and <i>temperature</i> attributes could not be supported by all the devices |           |            |                           |
| Field Name                                                                                                                                | Data Type | Field type | Description               |
| ID                                                                                                                                        | ID        | ID         | Logical object identifier |
| *temperature                                                                                                                              | FLOAT     | P          | Sampled temperature       |
| *standBy                                                                                                                                  | BOOLEAN   | P          | Stand by status           |

#### Query

```
SET standBy = true ON NOT EXISTS (temperature)
```



This query is executed on all the devices that are not provided with a *temperature* attribute, as stated by the *ON* clause. Moreover, the query can be executed only on the devices that support the *standBy* attribute, due to the semantics of actuation queries.

## 6.6 Other queries

### 6.6.1 Example 15

Given two temperature sensors, produce a record once a minute indicating the node that sensed the highest average temperature.

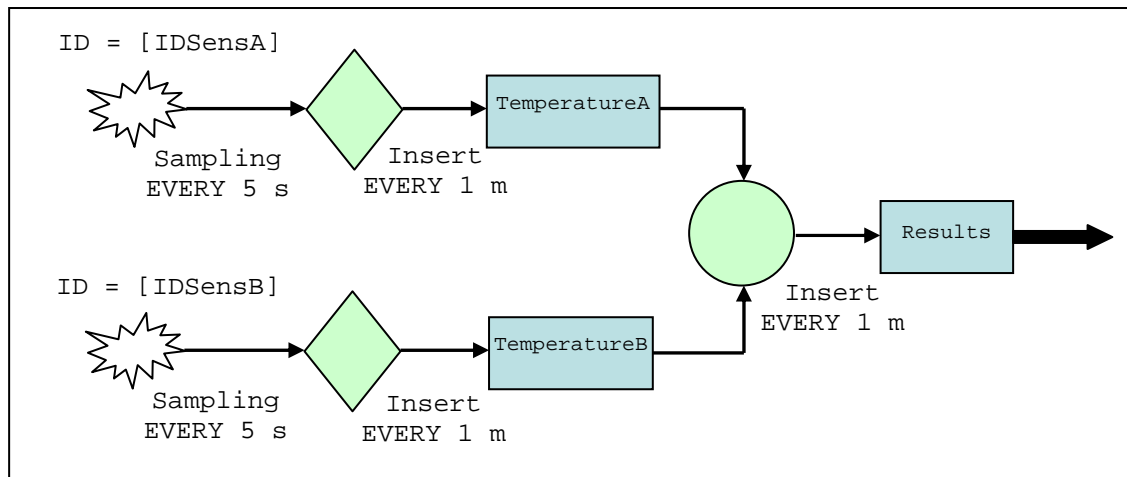
| RFID tag                                  |           |            |                           |
|-------------------------------------------|-----------|------------|---------------------------|
| Logical object wrapping a single RFID tag |           |            |                           |
| Field Name                                | Data Type | Field type | Description               |
| ID                                        | ID        | ID         | Logical object identifier |
| temp                                      | FLOAT     | P          | Sampled temperature       |

### Query

```
CREATE STREAM TemperatureA (temp FLOAT) AS
LOW:
 EVERY 1 m
 SELECT AVG(temp, 1 m)
 SAMPLING EVERY 5 s
 EXECUTE IF ID = [IDSensA]
```

```
CREATE STREAM TemperatureB (temp FLOAT) AS
LOW:
 EVERY 1 m
 SELECT AVG(temp, 1 m)
 SAMPLING EVERY 5 s
 EXECUTE IF ID = [IDSensB]
```

```
CREATE OUTPUT STREAM Results (sensor STRING) AS
HIGH:
 EVERY 1 m
 SELECT IIf(TemperatureA.temp > TemperatureB.temp, "A", "B")
 FROM TemperatureA (1 m), TemperatureB (1 m)
```



This example shows a high level query in which two windows are used in the *FROM* clause. The two low level queries produce a record every minute indicating the average temperature

sensed during the last minute. The high level query is activated every minute and works on two windows with size of one minute: each of them contains exactly a record and then the Cartesian product generated by the *FROM* clause is composed of only a record.

Consider a variation of the query in which the *FROM* clause uses windows with a size greater than a minute. In this case each window will contain more than one record and their Cartesian product will contain many value pairs. If we are interested only in the comparison of temperatures sampled by the two nodes in the same minute, a *WHERE* condition on native timestamps can be added.

### 6.6.2 Example 16

Every 10 minutes, get a record indicating the *ID* of the tag that was sensed by the *RFID* reader (with *ID* [reader]) the maximum number of times during the last 10 minutes.

| RFID tag                                  |           |            |                                                   |
|-------------------------------------------|-----------|------------|---------------------------------------------------|
| Logical object wrapping a single RFID tag |           |            |                                                   |
| Field Name                                | Data Type | Field type | Description                                       |
| ID                                        | ID        | ID         | Logical object identifier                         |
| lastReaderID                              | ID        | NP         | Cached ID of the last reader that sensed the tag  |
| lastReaderChanged                         | -         | E          | Notifies that the tag has been sensed by a reader |
| deviceType                                | STRING    | S          | Type of device                                    |

#### Query

```
CREATE STREAM Counters (tagID ID, counter INTEGER) AS
```

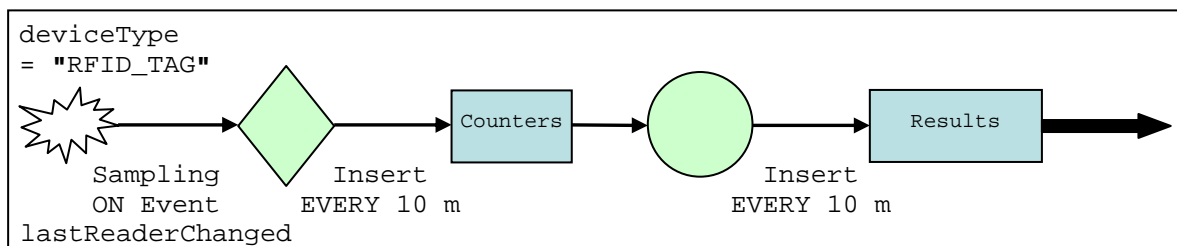
LOW:

```
EVERY 10 m
SELECT ID, COUNT(*, 10 m)
SAMPLING ON EVENT lastReaderChanged
WHERE lastReaderID = [reader]
EXECUTE IF deviceType = "RFID_TAG"
```

```
CREATE OUTPUT STREAM Results (tagID ID) AS
```

HIGH:

```
EVERY 10 m
SELECT tagID
FROM Counters (10 m)
WHERE Counters.counter = MAX(Counters.counter)
```





No new feature of the language is introduced in this example. The low level query is used to count how many times each tag is sensed by the reader [reader] during the ten minutes period. The high level query is used to find the tag that reported the maximum count.

## 6.7 Case study queries

In this section we present some queries that can be used in a real context. As a case study, we consider a company which produces high quality wines and gives a great attention to the production process quality. On one side, the company wants to keep under strict control some important parameters of the production cycle to guarantee the quality and the integrity of the product itself; on the other side it also requires the complete traceability of all the operations performed on each of the produced wine lots until the final labeling phase. The availability through the network of information and controls could start as early as with the monitoring of grapes in the vineyard in order to timely identify sudden changes in the values of relevant parameters (*e.g.*: the soil temperature and humidity).

In the following examples we suppose that the devices used for supporting the different phases of the production and delivery processes are distributed as schematically shown in Figure 28.

| WHERE    | WHAT                              | HOW               |
|----------|-----------------------------------|-------------------|
| vineyard | humidity , temperature, chemicals | sensors           |
| cellar   | humidity, temperature             | sensors           |
| bottle   | tracking information              | RFID tag          |
| pallet   | tracking information, temperature | RFID tag, sensors |
| truck    | position information              | GPS               |
| workers  | information system                | PDA               |

Figure 28: Devices usage and types of the case study

### 6.7.1 Vineyard monitoring

The first query we consider has the role of monitoring environment parameters of the area in which vine is cultivated. More specifically we want to sample temperature and humidity every 30 minutes, returning these values, together with the location of the sensor, only if the sensed temperature is in a critical range. In order to provide more accuracy in the results, each node is required to sample its sensors every ten minutes and to consider the average of the three last values.

| WSN node                                                                     |           |            |                                                                |
|------------------------------------------------------------------------------|-----------|------------|----------------------------------------------------------------|
| Logical object wrapping a single WSN node equipped with a temperature sensor |           |            |                                                                |
| Field Name                                                                   | Data Type | Field type | Description                                                    |
| ID                                                                           | ID        | ID         | Logical object identifier                                      |
| temp                                                                         | FLOAT     | P          | Sampled temperature                                            |
| humidity                                                                     | FLOAT     | P          | Sampled humidity                                               |
| locationX                                                                    | FLOAT     | S          | Sensor location – X coordinate                                 |
| locationY                                                                    | FLOAT     | S          | Sensor location – Y coordinate                                 |
| deviceType                                                                   | STRING    | S          | Type of device                                                 |
| powerLevel                                                                   | FLOAT     | P          | Current power levels of the devices expressed in a 1-100 scale |

| PDA                                  |           |            |                                |
|--------------------------------------|-----------|------------|--------------------------------|
| Logical object wrapping a worker PDA |           |            |                                |
| Field Name                           | Data Type | Field type | Description                    |
| ID                                   | ID        | ID         | Logical object identifier      |
| temp                                 | FLOAT     | P          | Sampled temperature            |
| locationX                            | FLOAT     | P          | Sensor location – X coordinate |
| locationY                            | FLOAT     | P          | Sensor location – Y coordinate |

In this situation, the user submitted query is only composed of a low level statement:

```
CREATE OUTPUT STREAM EnvironmentParameters
(sensorID ID, temp FLOAT, humidity FLOAT, locationX FLOAT, locationY FLOAT) AS
LOW:
 EVERY 30 m
 SELECT ID, AVG(temp, 30 m), AVG(humidity, 30 m), locationX, locationY
 SAMPLING
 EVERY 10 m
 EXECUTE IF EXISTS(temp) AND is_in_Vineyard(locationX, locationY)
 REFRESH EVERY 5 m
```

The *EXECUTE IF* clause requires the execution of the query on all the nodes currently placed in the yard and having on board a temperature sensor, but it does not restrict the query execution to wireless nodes only. Therefore, suppose that a worker is in the yard with a *PDA* which can sense the current temperature: this *PDA* will execute the query exactly as the wireless nodes (since the *PDA* has not a humidity sensor on board, *NULL* values will be produced).

In this example, it should be noticed that the *locationX* and *locationY* attributes are static for wireless nodes (and they are configured during the network deployment phase), while they are dynamic for *PDA*s: if a worker enters the yard, his *PDA* will start executing the query, possibly recovering data which should be provided by a “*dead*” sensor.

The next query shows an important peculiarity of our language: the ability of querying the network state as “*normal*” data. Suppose that, in the previous example, we want to monitor the

power state of the wireless nodes, in order to detect low powered devices and to allow the substitution of the batteries before nodes become inactive.

```
CREATE OUTPUT STREAM LowPoweredDevices (sensorID ID) AS
LOW:
 EVERY ONE
 SELECT ID
 SAMPLING EVERY 24 h
 WHERE powerLevel < 0.15
 EXECUTE IF deviceType = "WirelessNode"
```

If the user wants to know the number of low powered wireless nodes, a high level query performing the count aggregation can be written starting from the stream generated by the previous low level query:

```
CREATE OUTPUT STREAM NumberOfLowPoweredDevices (counter INTEGER) AS
HIGH:
 EVERY 24 h
 SELECT COUNT(*)
 FROM LowPoweredDevices(24 h)
```

### 6.7.2 Transport monitoring

Now, we consider the monitoring of wine during the transport. In particular, suppose that every truck is equipped with a *GPS* and a base station. Suppose also that each pallet has a temperature sensor used to sense the temperature of the contained bottles. The query requires to produce the list of pallets whose temperature exceeded a certain threshold while the truck was travelling through a given zone, which is considered particularly critical.

| GPS                                  |           |            |                                               |
|--------------------------------------|-----------|------------|-----------------------------------------------|
| Logical object wrapping a GPS device |           |            |                                               |
| Field Name                           | Data Type | Field type | Description                                   |
| ID                                   | ID        | ID         | Logical object identifier                     |
| linkedBaseStationID                  | ID        | S          | ID of the base station mounted over the truck |
| locationX                            | FLOAT     | P          | Sensor location – X coordinate                |
| locationY                            | FLOAT     | P          | Sensor location – Y coordinate                |
| deviceType                           | STRING    | S          | Type of device                                |

| WSN node                                                                     |           |            |                                                               |
|------------------------------------------------------------------------------|-----------|------------|---------------------------------------------------------------|
| Logical object wrapping a single WSN node equipped with a temperature sensor |           |            |                                                               |
| Field Name                                                                   | Data Type | Field type | Description                                                   |
| ID                                                                           | ID        | ID         | Logical object identifier                                     |
| baseStationID                                                                | ID        | NP         | ID of the base station the WSN node is currently connected to |
| temp                                                                         | FLOAT     | P          | Sampled temperature                                           |

```
CREATE SNAPSHOT TrucksPositions (linkedBaseStationID ID)
WITH DURATION 1 h AS
LOW:
 SELECT linkedBaseStationID
 SAMPLING
 EVERY 1 h
 WHERE is_in_CriticalZone(locationX, locationY)
 EXECUTE IF deviceType = "GPS"
```

```
CREATE OUTPUT STREAM OutOfTemperatureRangePallets (palletID ID) AS
LOW:
 EVERY 10 m
 SELECT ID
 SAMPLING EVERY 10 m
 WHERE temp > [threshold]
 PILOT JOIN TrucksPositions
 ON baseStationID = TrucksPositions.linkedBaseStationID
```

In this query the *pilot join* operation is used to activate the temperature sampling only on the pallets contained in the trucks that are driving into the critical zone. Note that the interface of logical objects wrapping *GPS* devices has an attribute *linkedBaseStationId* that retrieves the *ID* of the base station mounted on the same truck (this is a static attribute, whose value is defined during the network deployment time).

## 7 Conclusions

In this work we widely analyzed the main requirements a middleware should comply with in order to collect data from a pervasive system. We presented these features and discussed how they can be mapped on a fully declarative language. We also studied if and how similar projects handle the same issues. We described *PerLa* architecture, that is composed of two main elements: a language and a middleware.

A declarative SQL-like language has been designed to provide the final user with an overall homogeneous view on the whole pervasive system. The provided interface is simple and flexible and it allows users to completely control each physical node, masking the heterogeneity at data level. For each language clause we defined the syntax and explained the semantics. We also showed some query examples to better clarify the language semantics.

A middleware has been implemented in order to support the execution of *PerLa* queries and it is mainly charged to manage the heterogeneity at physical integration level. The key component is the *FPC (Functionality Proxy Component)*, that is a Java object able to represent a physical device in the middleware and to take the device's place whenever unsupported operations are required.

In this report we have not investigated some optimizations that can improve the performances of the middleware and the expressiveness of the language. Our future work on the middleware will be focused on supporting intelligent in-network processing protocols. More specifically, the current implementation is able to optimize the execution of temporal aggregations on data gathered from the same sensor, but it cannot perform spatial aggregations with in-network processing. We identified context and content based routing protocols as a possible way to solve this issue.

From the language point of view, there is a set of useful language functionalities that are not implemented yet in the grammar presented in this work and there are several ways to extend the language expressiveness. In the following the main open issues are briefly listed.

- **Termination of a query.** Although *WSNs* are often used to execute queries in a continuous way, we think that the language must however support a mechanism to stop a running query. At the moment we only introduced a clause in the low level language to allow specifying when an instance of a running low level query has to be stopped. This termination clause is related to a single logical object, but probably a clause should be added to require the termination of the whole user submitted query.
- **Extension of actuation queries.** The current version of the language supports a very simple version of actuation queries, that only allow to set some parameters on a physical device. This is enough to demonstrate that, from a conceptual point of view, sensors and actuators can be abstracted and managed in a similar way. But it seems

useful extending the semantics of actuation queries, in order to support also periodic actuations and to introduce some clauses having a semantics similar to the *PILOT JOIN* one.

- **Extension of the EXECUTE IF clause.** Some extensions to the *EXECUTE IF* clause can be explored to increase the language expressiveness. For example, an option to control the maximum number of logical objects that will take part in a query can be added.
- **Definition of in-network data mining operations.** Some data mining statements can be introduced in the language: they should be a set of clauses to require the execution of typical data mining primitives on intermediate streams, before they come to the node that submitted the query [30] [31]. The implementation of these statements is not strictly required to build a minimal query language for pervasive systems but it can be an additional useful service for the user.
- **Reduction of network traffic.** We plan to integrate some data reduction facilities into the language, in order to reduce the network traffic. As an example, an algorithm of data stream compression [32] can be adopted to reduce the amount of data exchanged between a device and its *FPC*.
- **Support for context.** Although the pilot join operation is a first attempt to support context-based queries, we plan to extend the language semantics with specific operators.

Finally, an interesting extension is the definition of a procedural language that can be used to add new functionalities to the declarative one. More specifically, this language should provide the constructs to define new *aggregation operators* and new customized *grouping operators* (to be used in *GROUP BY* clauses). For example it can be useful to define a new grouping operator, able to split a set of records (containing a location field) depending on a squared grid.

# **Appendixes**

## A Parser implementation

The parser is a component of the query analyzer and it has the role of analyzing language statements and representing them in a format that is optimized for further processing. It verifies if the text taken as input is a valid string of the language defined by the formal grammar presented in Chapter 5. If that is the case, it builds a data structure that describes all the clauses found in the input text; otherwise, an error is raised.

Many *compiler-compiler* software exist and they can be used to automatically build the parser. They take a grammar of the language as input (usually in *BNF* or *EBNF* form) and they produce the parser source code as output. Some pieces of code can usually be associated to each grammar rule and they will be executed when these rules are applied by the parser. In our project, these blocks are used to build the data structures that define the internal representation of queries.

Different parser generators can be classified on the basis of the programming language used to write the output code and on the basis of the grammar subset that is accepted as input. In fact, if only specific classes of grammars are allowed, high-performance algorithms can be used to parse the input string with a linear time complexity. The most common parser generators accept *LL*, *LR* or *LALR* grammars [33].

As said in Chapter 3 we decided to use the *Java* platform to implement the system components not directly interacting with the hardware. Among the *Java* based parser generators, we decided to use *JavaCC* (*Java Compiler Compiler*) that is an open source project able to generate top-down parsers. *LL(k)* is the accepted class of grammars, but *semantic lookahead* can be used to handle with other particular types of grammar [34].

As said before, the grammar explained in Chapter 5 was written to be human readable and, then, it can contain some ambiguities. To use that grammar with *JavaCC*, some changes were needed. More specifically, all the ambiguities was removed as well as the left side recursions. The obtained grammar, written with the *JavaCC* syntax, is reported in *Appendix B*. Although the *LL(k)* class is supported, we tried to write a grammar that is *LL(1)* as much as possible. In fact, *JavaCC* allows to specify the *lookahead* to be adopted at each choice point. In this way, the obtained parser is mainly *LL(1)* with some *LL(2)* choice points. Note that, when the *lookahead* is not explicitly specified, a default *lookahead* of one token is used by *JavaCC* and an automatic choice conflict control is provided.

Although in Chapter 5 we have specified the whole language through an *EBNF* grammar, parser generators usually require to split the language definition in two blocks: tokens and grammar productions. Tokens are regular expressions and they specify the set of terminal symbols used in the productions. Thus, the software produced by the generator is composed of two components, called *lexical analyzer* and *syntactical analyzer*: the first one scans the input text and tokenizes it, while the second one checks the obtained tokens sequence to verify if it complies with grammar productions.

In Chapter 5 grammar, many non terminals symbols are introduced only to improve readability: they are all defined as an identifier or as two identifiers separated by a dot (*e.g.*:



*DataStructureName*, *DataFieldName*, *LogicalObjectField*, *PilotJoinField*, etc.). In *JavaCC* grammar they have been substituted with only three non terminal symbols (*Identifier*, *QualifiedIdentifier* and *OptionallyQualifiedIdentifier*) to make easier the grammar rewriting w.r.t. *LL(1)* constrains. For clearness, whenever these three symbols are used, a comment is reported to specify the original non terminal they substitute.

There are few situations in which the specification of the *lookahead* clause was introduced. In the productions that allows the choice between a duration and a number of samples (or selections) a two tokens *lookahead* is needed. Consider for example the production that defines the *WindowSize* non terminal symbol:

```
void WindowSize() : {}
{ LOOKAHEAD(2) Duration() | SamplesNumber() }
```

If the window size is specified through a duration, a numeric constant (integer or float) followed by a unit of time is required. Otherwise, if the window size is defined in terms of a number of records, an integer constant followed by the *SAMPLE* keyword is required. If a float constant is found, it is certainly the beginning of a duration, but if an integer constant is found, decision cannot be taken until the next token is analyzed. So the choice point must be declared as *LOOKAHEAD (2)*.

Another situation in which an explicit specification of *lookahead* is needed is the definition of the *Constant* symbol:

```
void Constant() : {}
{ ConstantNull() | ConstantBoolean() | ConstantString() |
 LOOKAHEAD(1) ConstantInteger() | ConstantFloat() }
```

To better understand this production, the explanation should start from the lexical analyzer. Two tokens are defined to recognize numeric values: *CONSTANT\_INTEGER* and *CONSTANT\_FLOAT*. The first one corresponds to integer numbers, while the second one corresponds only to the numeric values containing a dot. At the syntactical analyzer level, two productions are defined: *ConstantInteger* is exactly the *CONSTANT\_INTEGER* token, while *ConstantFloat* is the union of *CONSTANT\_INTEGER* and *CONSTANT\_FLOAT* tokens. For these reasons, *JavaCC* signals a choice conflict in the production reported above. In fact, when a *CONSTANT\_INTEGER* token is received from the scanner, both *ConstantInteger* and *ConstantFloat* productions can be matched. However, the first one is certainly the correct choice, without the needing of looking ahead. Since *JavaCC* chooses the firstly written production, specifying *ConstantInteger* before *ConstantFloat* (in the *Constant* production) is enough. To suppress the warning signaling, an explicit *LOOKAHEAD (1)* has been specified. The last productions in which a numeric *lookahead* was used are relative to external function calls. The reasons that led us to using a *lookahead* of two tokens will be clearer in the following.

In Chapter 5 the different kinds of allowed expressions were not described through the grammar, but rather with a table (Figure 26). In *JavaCC* these constraints have been specified directly in the grammar, using the *semantic lookahead* feature. It allows to specify any arbitrary

Boolean expression, whose evaluation determines which option is taken at a choice point. We used this *JavaCC* feature as follows:

- We defined a Java enumeration (*ExpressionType*) to list the different kinds of expression.
- We added a parameter (*parExpressionType*) to the *Expression* production to indicate the type of the recognized expression.
- We propagated this information down to the production that defines the expression base blocks (*ExpressionValuePrimary*), through parameter passing.
- We placed *semantic lookahead* clauses in the *ExpressionValuePrimary* production to define the set of allowed base blocks, depending on the value of *parExpressionType*.

Note that, in the current version of *JavaCC*, some constraints must be respected when *semantic lookahead* is used: in particular, if local variables (or parameters) are used in a *semantic lookahead* specification within the *EBNF* production for a certain non terminal, this non terminal mustn't be used in *syntactic lookahead*, or in a *lookahead* of more than 1 token; otherwise a compiler error will be present in the generated source code. This limitation forced us to split the definition of function calls in two cases: with and without parameters. Since these two alternatives start with the same token, a *lookahead* specification has been introduced.

A final consideration about the grammar reported in *Appendix B* is relative to a pair of features that are not present in the grammar of Chapter 5: the possibility of writing a list of statements separated by a “;” character and the possibility of introducing some comments in the query, using a *C* like notation.

Figure 29, Figure 30, Figure 31, Figure 33 and Figure 33 report a UML class diagram of the internal representation of queries, that is built by the parser.

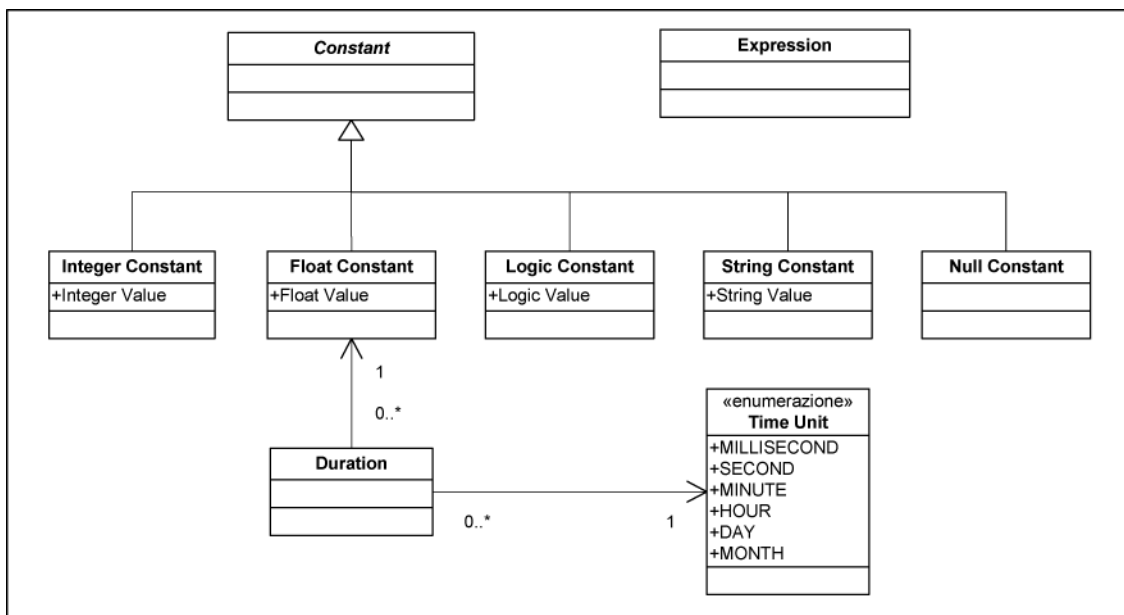


Figure 29: Internal representation of queries (1/5)

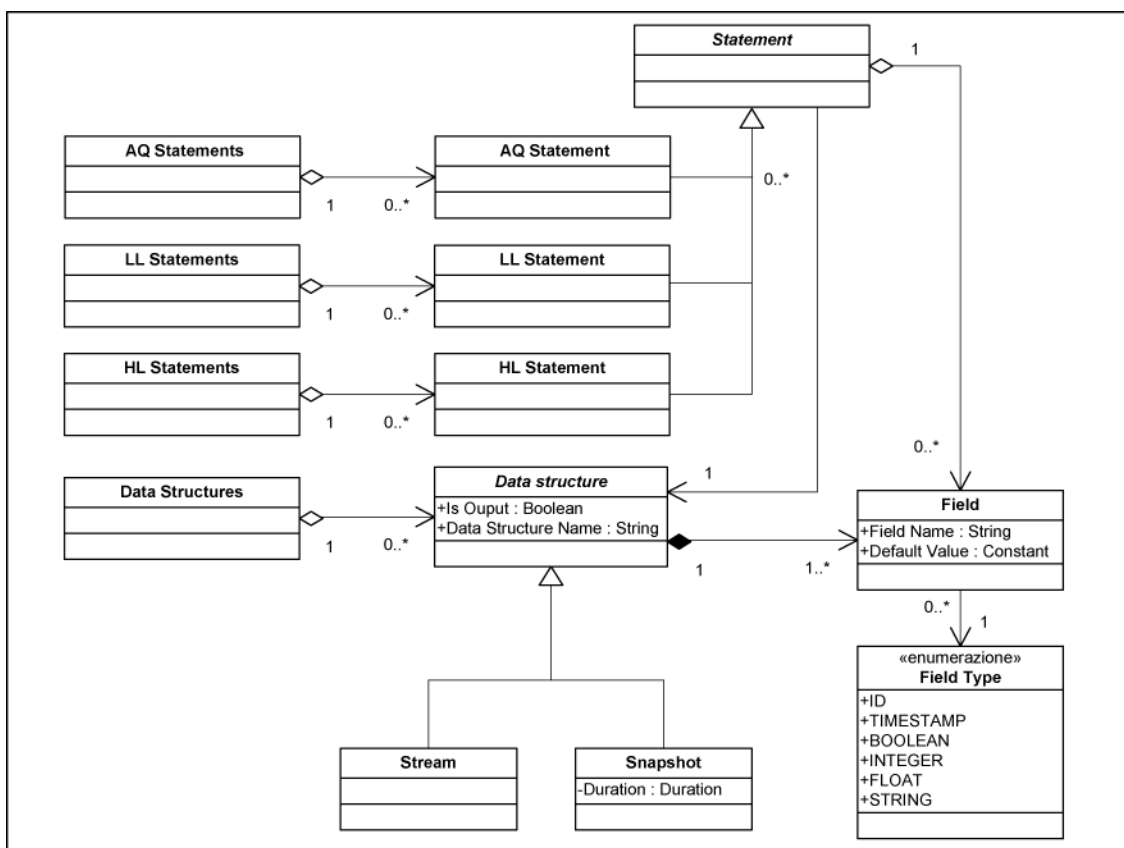


Figure 30: Internal representation of queries (2/5)

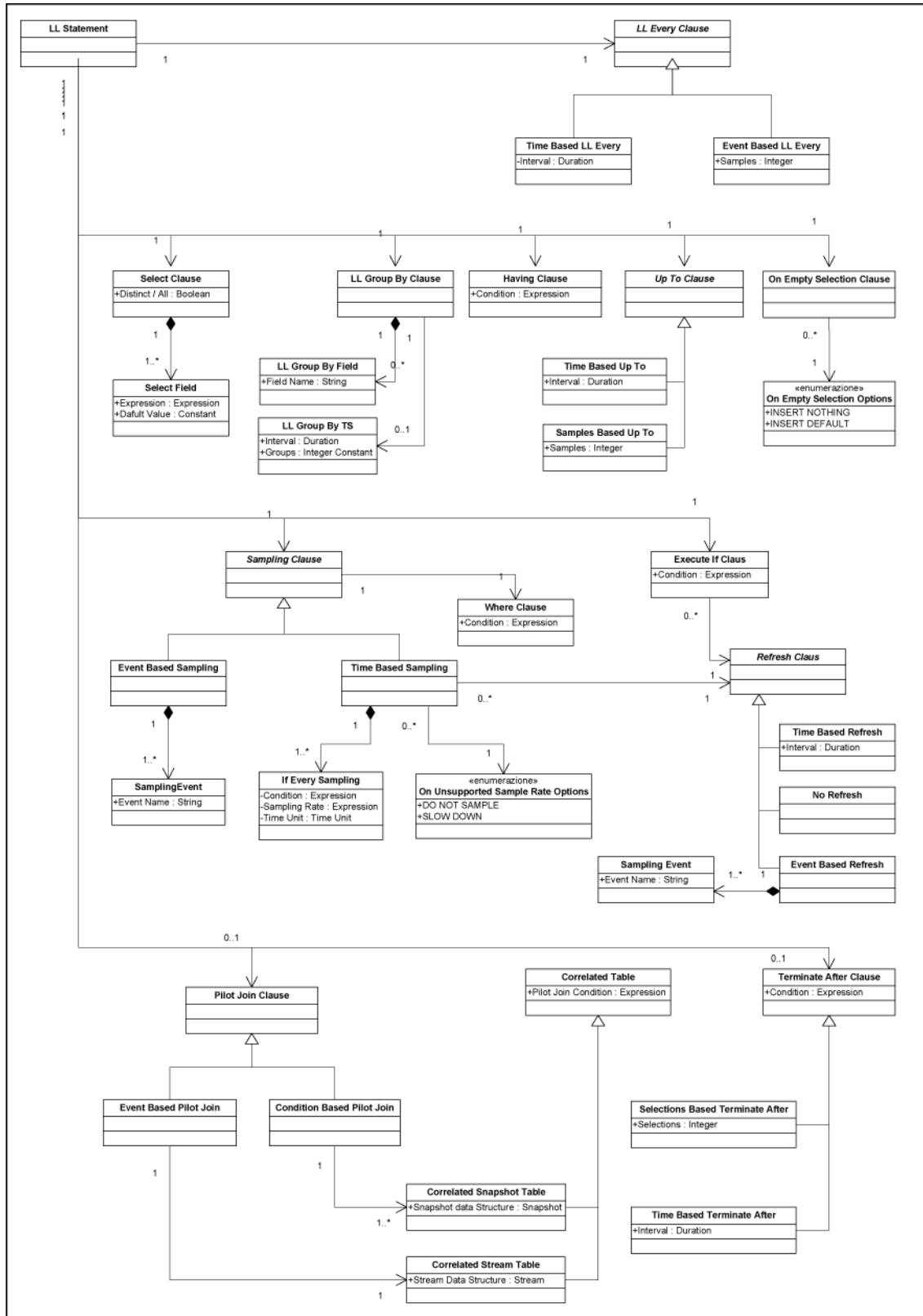


Figure 31: Internal representation of queries (3/5)

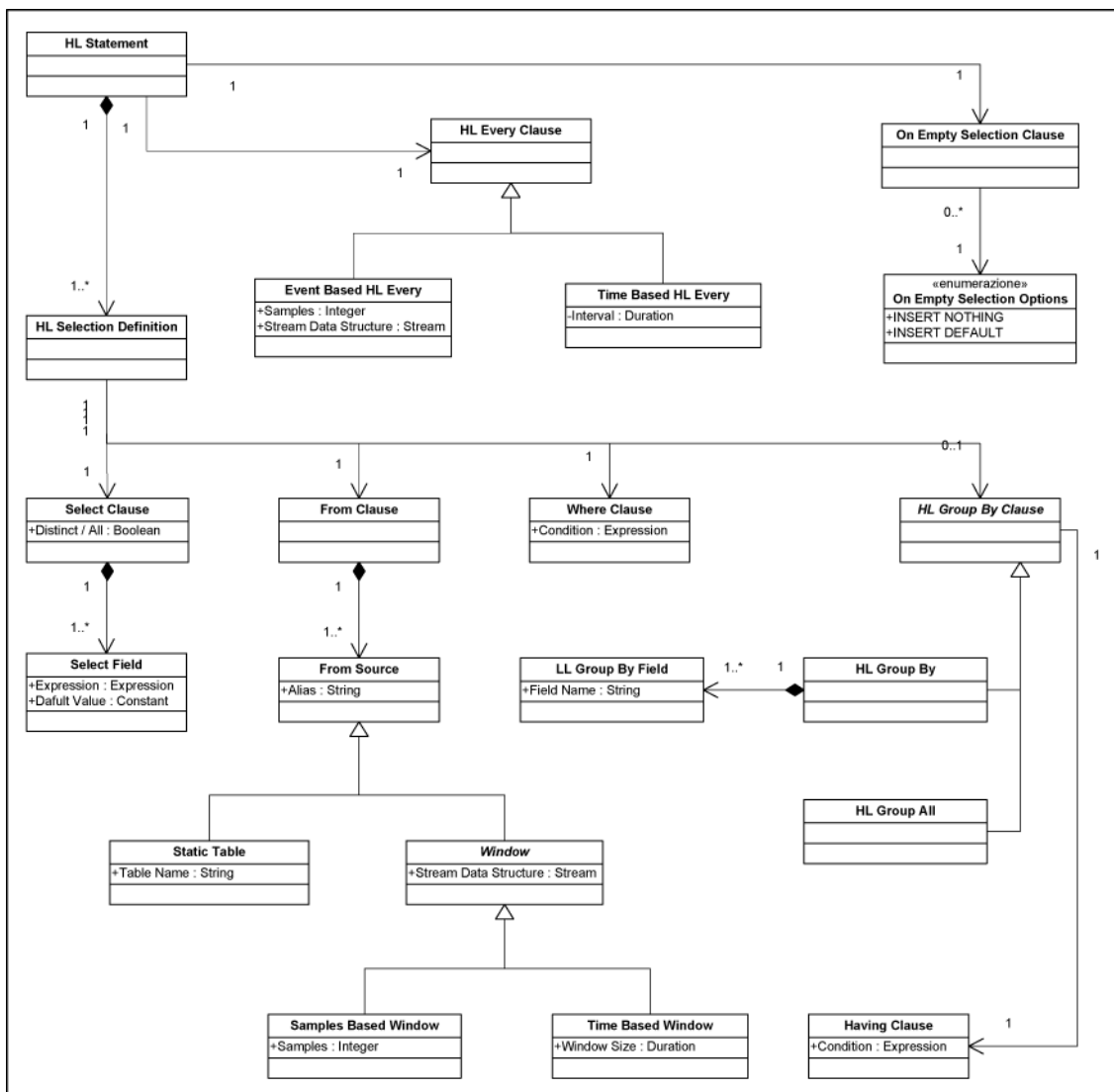


Figure 32: Internal representation of queries (4/5)

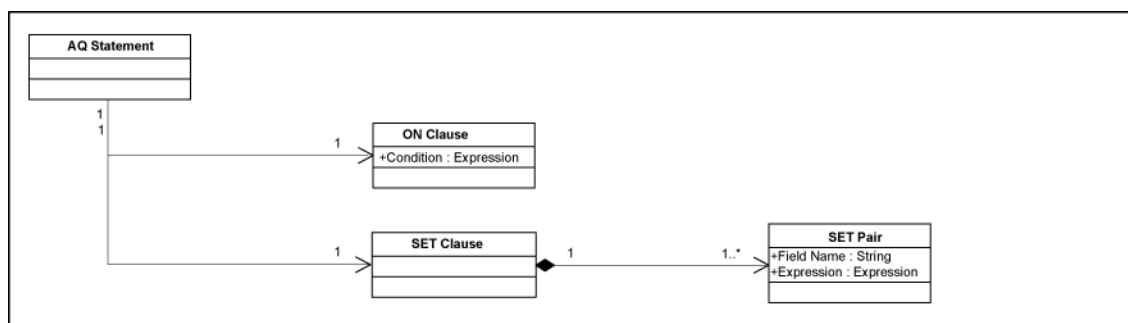


Figure 33: Internal representation of queries (5/5)



## B Language EBNF

```
/* SEPARATORS
 * Characters between tokens. They are not sent to the parser
 */
SKIP: { " " | "\r" | "\t" | "\n" }

/* COMMENTS
 * C-style comments (/ * ... * /): lexical states are used.
 */
MORE: { "/*" : COMMENT }
<COMMENT> MORE: { <~[]> }
<COMMENT> SKIP: { "*/" : DEFAULT }

/* Tokens of the *CREATE* clause */
TOKEN: {
 < KEYWORD_CREATE: "CREATE" > | < KEYWORD_OUTPUT: "OUTPUT" > |
 < KEYWORD_AS : "AS" >
}

/* Tokens of the *INSERT* clause */
TOKEN: { < KEYWORD_INSERT: "INSERT" > | < KEYWORD_INTO: "INTO" > }

/* Tokens of the *EVERY* clause */
TOKEN: {
 < KEYWORD_EVERY : "EVERY" > | < KEYWORD_IN: "IN" >
 |
 < KEYWORD_SYNCHRONIZED: "SYNCHRONIZED" >
}

/* Tokens of the *SELECT* clause */
TOKEN: {
 < KEYWORD_SELECT: "SELECT" > |
 < KEYWORD_DISTINCT: "DISTINCT" > |
 < KEYWORD_ALL : "ALL" > |
 < KEYWORD_GROUPTS: "GROUP_TS" >
}
```

```

/* Tokens of the *UNION* clause */
TOKEN: { < KEYWORD_UNION: "UNION" > /* | < KEYWORD_ALL: "ALL" > */ }

/* Tokens of the *WHERE* clause */
TOKEN: { < KEYWORD_WHERE: "WHERE" > }

/* Tokens of the *GROUP BY* clause */
TOKEN: {
 < KEYWORD_GROUP: "GROUP" > | < KEYWORD_BY: "BY" >
 /* | < KEYWORD_ALL: "ALL" > */ |
 < KEYWORD_GROUPS: "GROUPS" >
}

/* Tokens of the *HAVING* clause */
TOKEN: { < KEYWORD_HAVING: "HAVING" > }

/* Tokens of the *UP TO* clause */
TOKEN: { < KEYWORD_UP: "UP" > | < KEYWORD_TO: "TO" > }

/* Tokens of the *ON EMPTY SELECTION* clause */
TOKEN: {
 /* < KEYWORD_ON: "ON" > | */
 < KEYWORD_EMPTY: "EMPTY" > | < KEYWORD_SELECTION: "SELECTION" > |
 /* < KEYWORD_INSERT: "INSERT" > | */
 < KEYWORD_NOTHING: "NOTHING" > | < KEYWORD_DEFAULT: "DEFAULT" >
}

/* Tokens of the *SAMPLING* clause */
TOKEN: {
 < KEYWORD_SAMPLING: "SAMPLING" > |
 /* < KEYWORD_ON: "ON" > | */
 < KEYWORD_EVENT: "EVENT" > |
 /* < KEYWORD_IF: "IF" > | */
 /* < KEYWORD_EVERY: "EVERY" > | */
 < KEYWORD_ELSE: "ELSE" >
}

```



```
/* Tokens of the *ON UNSUPPORTED SAMPLING RATE* clause */
TOKEN: {
 /* < KEYWORD_ON: "ON" > | */
 < KEYWORD_UNSUPPORTED: "UNSUPPORTED" > |
 < KEYWORD_SAMPLE: "SAMPLE" > | < KEYWORD_RATE: "RATE" > |
 < KEYWORD_DO: "DO" > |
 /* < KEYWORD_NOT: "NOT" > | */
 < KEYWORD_SLOW: "SLOW" > | < KEYWORD_DOWN: "DOWN" >
}

/* Tokens of the *REFRESH* clause */
TOKEN: {
 < KEYWORD_REFRESH: "REFRESH" > |
 /* < KEYWORD_EVERY: "EVERY" > | */
 < KEYWORD_NEVER: "NEVER" >
}

/* Tokens of the *PILOT JOIN* clause */
TOKEN: {
 < KEYWORD_PILOT: "PILOT" > | < KEYWORD_JOIN: "JOIN" > |
 < KEYWORD_ON : "ON" >
}

/* Tokens of the *EXECUTE IF* clause */
TOKEN: {
 < KEYWORD_EXECUTE: "EXECUTE" > | < KEYWORD_IF: "IF" > |
 < KEYWORD_EXISTS : "EXISTS" >
}

/* Tokens of the *TERMINATE AFTER* clause */
TOKEN: {
 < KEYWORD_TERMINATE: "TERMINATE" > | < KEYWORD_AFTER: "AFTER" > |
 < KEYWORD_SELECTIONS: "SELECTIONS" >
}

/* Tokens of the *FROM* clause */
TOKEN: { < KEYWORD_FROM: "FROM" > /* | < KEYWORD_AS: "AS" > */ }
```

```

/* Tokens of the *SET* clause */
TOKEN: {
 < KEYWORD_SET: "SET" >
}

/* Tokens of the *ON* clause */
TOKEN: {
 /* < KEYWORD_ON: "ON" > */
}

/* Tokens for DATA STRUCTURES */
TOKEN: {
 < KEYWORD_STREAM: "STREAM" > | < KEYWORD_SNAPSHOT: "SNAPSHOT" >
}

/* Others tokens */
TOKEN: {
 /* Token used to introduced the low level query */
 < KEYWORD_LOW: "LOW" > |
 /* Token used to introduced the high level query */
 < KEYWORD_HIGH: "HIGH" > |
 /* Token used, within DURATION, to introduce a TIME QUANTUM */
 < KEYWORD_WITH: "WITH" > |
 /* Token used, within WITH, to introduce a TIME QUANTUM */
 < KEYWORD_DURATION: "DURATION" > |
 /* Token used to indicate sample number*/
 < KEYWORD_SAMPLES: "SAMPLES" > |
 /* Token used to indicate sample number equal to ONE*/
 < KEYWORD_ONE: "ONE" > |
 /* Token used in Boolean expressions (in order to verify their
 * value: TRUE, FALSE or UNKNOWN) or to compare with a NULL
 * value */
 < KEYWORD_IS : "IS" > | < KEYWORD_BETWEEN: "BETWEEN" > |
 < KEYWORD_LIKE: "LIKE" >
}

/* ALGEBRAIC OPERATORS */
TOKEN: {
 < OPERATOR_MULTIPLY: "*" > | < OPERATOR_DIVIDE: "/" > |
 < OPERATOR_PLUS : "+" > | < OPERATOR_MINUS : "-" >
}

/* LOGIC OPERATORS */
TOKEN: {
 < OPERATOR_NOT: "NOT" > | < OPERATOR_XOR: "XOR" > |
 < OPERATOR_AND: "AND" > | < OPERATOR_OR : "OR" >
}

```

```
/* BITWISE OPERATORS */
TOKEN: {
 < OPERATOR_BITWISE_NOT: "!" > | < OPERATOR_BITWISE_XOR: "^" > |
 < OPERATOR_BITWISE_AND: "&" > | < OPERATOR_BITWISE_OR: "|" > |
 < OPERATOR_BITWISE_LSH: "<<" > | < OPERATOR_BITWISE_RSH: ">>" >
}

/* COMPARISON OPERATORS */
TOKEN: {
 < OPERATOR_GREATER: ">" > | < OPERATOR_LESS: "<" > |
 < OPERATOR_GREATER_EQUAL: ">=" > |
 < OPERATOR_LESS_EQUAL: "<=" > |
 < OPERATOR_EQUAL: "=" > |
 < OPERATOR_NOT_EQUAL: "<>" | "!=" >
}

/* AGGREGATION FUNCTIONS */
TOKEN: {
 < FUNCTION_COUNT: "COUNT" > | < FUNCTION_AVG: "AVG" > |
 < FUNCTION_MAX : "MAX" > | < FUNCTION_MIN: "MIN" > |
 < FUNCTION_SUM : "SUM" >
 /* | < FUNCTION_TIMESTAMP: "TIMESTAMP" > */
}

/* TIME UNITS */
TOKEN: {
 < TIMEUNIT_S: "seconds" | "s" > |
 < TIMEUNIT_M: "minutes" | "m" > |
 < TIMEUNIT_H: "hours" | "h" > |
 < TIMEUNIT_MS: "milliseconds" | "ms" > |
 < TIMEUNIT_D: "days" | "d" > |
 < TIMEUNIT_MT: "months" | "mt" >
}
```

```
/* DATA TYPES */
TOKEN: {
 < TYPE_ID : "ID" > | < TYPE_TS : "TIMESTAMP" > |
 < TYPE_BOOLEAN: "BOOLEAN" > | < TYPE_INTEGER: "INTEGER" > |
 < TYPE_FLOAT : "FLOAT" > | < TYPE_STRING : "STRING" >
}

/* CONSTANTS
* Note: DO NOT directly use in the grammar the tokens introduced
* in this section. Correspondent productions must be used.
*/

/* NULL constant */
TOKEN: { < CONSTANT_NULL: "NULL" > }

/* BOOLEAN constants:
* They define both the Boolean set of values (TRUE and FALSE)
* and the three valued logic set of values (TRUE, FALSE and UNKNOWN)
*/
TOKEN: {
 < CONSTANT_BOOLEAN_TRUE: "TRUE" > |
 < CONSTANT_BOOLEAN_FALSE: "FALSE" > |
 < CONSTANT_BOOLEAN_UNKNOWN: "UNKNOWN" >
}

/* NUMERIC constants:
* They define integer and float values.
* Note: CONSTANT_FLOAT DO NOT corresponds to all numeric values,
* but only to the ones containing a dot ('.').
* e.g. '3.5' will be interpreted as a CONSTANT_FLOAT,
* while '3' will be interpreted as a CONSTANT_INTEGER.
*/
```



```

<NON_DOUBLE_QUOTED_STRING> TOKEN: {
 < CONSTANT_DOUBLE_QUOTED_STRING_VALUE: (~["\""] | "\"" "\\")* > :
 NON_DOUBLE_QUOTED_STRING_END
}
<NON_SINGLE_QUOTED_STRING_END> TOKEN: {
 < CONSTANT_SINGLE_QUOTED_STRING_END: "'" > : DEFAULT
}
<NON_DOUBLE_QUOTED_STRING_END> TOKEN: {
 < CONSTANT_DOUBLE_QUOTED_STRING_END: "\"" > : DEFAULT
}

/* IDENTIFIERS */
TOKEN: {
 < #DIGIT: ["0" - "9"] > |
 < #LITERAL: ["a" - "z"] | ["A" - "Z"] > |
 < #UNDERSCORE: "_" > |
 < IDENTIFIER: (<LITERAL> | <UNDERSCORE>)
 (<DIGIT> | <LITERAL> | <UNDERSCORE>)* >
}

/*****
GENERIC PRODUCTIONS(Signs, Constants, Field names, Table names, etc.)
*****/
void StatementSequence():{ } { Statement(";")* Statement()* <EOF> }

void Sign():{ } { <OPERATOR_PLUS> | <OPERATOR_MINUS> }

void LogicValue():{ } {
 <CONSTANT_BOOLEAN_TRUE> | <CONSTANT_BOOLEAN_FALSE> |
 <CONSTANT_BOOLEAN_UNKNOWN>
}

void ConstantNull():{ } { <CONSTANT_NULL> }

void ConstantBoolean():{ } {
 <CONSTANT_BOOLEAN_TRUE> | <CONSTANT_BOOLEAN_FALSE>
}

```

```
void ConstantFloat():{}{ (<CONSTANT_INTEGER> | <CONSTANT_FLOAT>) }

void ConstantInteger():{}{ <CONSTANT_INTEGER> }

void ConstantString():{}{
 (
 <CONSTANT_SINGLE_QUOTED_STRING_START>
 <CONSTANT_SINGLE_QUOTED_STRING_VALUE>
 <CONSTANT_SINGLE_QUOTED_STRING_END>
)
 |
 (
 <CONSTANT_DOUBLE_QUOTED_STRING_START>
 <CONSTANT_DOUBLE_QUOTED_STRING_VALUE>
 <CONSTANT_DOUBLE_QUOTED_STRING_END>
)
}

void Constant():{}{
 ConstantNull() | ConstantBoolean() | ConstantString() |
 LOOKAHEAD(1) ConstantInteger() |
 ConstantFloat()
}

void SignedConstant():{} { [Sign()] Constant() }

void ComparisonOperator(): {}{
 <OPERATOR_GREATER> | <OPERATOR_LESS> | <OPERATOR_GREATER_EQUAL> |
 <OPERATOR_LESS_EQUAL> | <OPERATOR_EQUAL> | <OPERATOR_NOT_EQUAL>
}

void FieldType():{}{
 <TYPE_ID> | <TYPE_TS> | <TYPE_BOOLEAN> | <TYPE_INTEGER> |
 <TYPE_FLOAT> | <TYPE_STRING>
}

void TimeUnit():{}{
 <TIMEUNIT_S> | <TIMEUNIT_M> | <TIMEUNIT_H> | <TIMEUNIT_MS> |
 <TIMEUNIT_D> | <TIMEUNIT_MT>
}
```

```

void AggregationOperator():{}{
 <FUNCTION_AVG> | <FUNCTION_MIN> |
 <FUNCTION_MAX> | <FUNCTION_SUM>
}

void Duration():{}{ ConstantFloat() TimeUnit() }

void SamplesNumber():{}{
 (ConstantInteger() <KEYWORD_SAMPLES>) | (<KEYWORD_ONE>)
}

void SelectionsNumber():{}{(ConstantInteger() <KEYWORD_SELECTIONS>) }

void WindowSize():{}{ LOOKAHEAD(2) Duration() | SamplesNumber() }

void Identifier():{}{ <IDENTIFIER> }

void QualifiedIdentifier():{}{<IDENTIFIER> "." <IDENTIFIER> }

void OptionallyQualifiedIdentifier():{}{<IDENTIFIER>["." IDENTIFIER]}

void Aggregate(ExpressionType parExpressionType):{}{
 (
 (<FUNCTION_COUNT> "(" "*") |
 (AggregationOperator() "(" Expression(parExpressionType))
)
 (
 LOOKAHEAD({parExpressionType == ExpressionType.LOW_LEVEL_ALL})
 ("," WindowSize() ["," Expression(parExpressionType)]) |
 LOOKAHEAD({parExpressionType == ExpressionType.HIGH_LEVEL_ALL})
 (["," Expression(parExpressionType)])
) ")"
}

```



```

void FunctionCall(ExpressionType parExpressionType):{}{
 <IDENTIFIER> (
 LOOKAHEAD(2) ("(" ")")
 |
 ("(" Expression(parExpressionType)
 ("," Expression(parExpressionType))* ")")
)
 }

void ExistsAttribute():{}{
 <KEYWORD_EXISTS>
 "(" (Identifier() /* LogicalObjectField */ | <KEYWORD_ALL>) ")"
}

/*****
EXPRESSIONS PRODUCTIONS
*****/
void Expression(ExpressionType parExpressionType):{}{
 ExpressionBooleanTerm(parExpressionType)
 (<OPERATOR_OR> ExpressionBooleanTerm(parExpressionType))*
}

void ExpressionBooleanTerm(ExpressionType parExpressionType):{}{
 ExpressionBooleanFactor(parExpressionType)
 (<OPERATOR_AND> ExpressionBooleanFactor(parExpressionType))*
}

void ExpressionBooleanFactor(ExpressionType parExpressionType):{}{
 ExpressionBooleanTest(parExpressionType)
 (<OPERATOR_XOR> ExpressionBooleanTest(parExpressionType))*
}

void ExpressionBooleanTest(ExpressionType parExpressionType):{}{
 (<OPERATOR_NOT>)* ExpressionBooleanPrimary(parExpressionType)
}

void LogicTest():{}{ [<KEYWORD_IS> [<OPERATOR_NOT>] LogicValue()] }

```

```

void ExpressionBooleanPrimary(ExpressionType parExpressionType):{}{
 ExpressionBit(parExpressionType)
 [
 (ComparisonOperator()ExpressionBit(parExpressionType)LogicTest())
 |
 (<KEYWORD_IS> [<OPERATOR_NOT>]
 (<CONSTANT_NULL> LogicTest() | LogicValue()))
 |
 (<KEYWORD_BETWEEN>ExpressionBit(parExpressionType)<OPERATOR_AND>
 ExpressionBit(parExpressionType) LogicTest())
 |
 (<KEYWORD_LIKE> ConstantString() LogicTest())
]
}

void ExpressionBit(ExpressionType parExpressionType):{}{
 ExpressionBitTerm(parExpressionType)
 (<OPERATOR_BITWISE_OR> ExpressionBitTerm(parExpressionType))*
}

void ExpressionBitTerm(ExpressionType parExpressionType):{}{
 ExpressionBitFactor(parExpressionType)
 (<OPERATOR_BITWISE_AND>ExpressionBitFactor(parExpressionType))*
}

void ExpressionBitFactor(ExpressionType parExpressionType):{}{
 ExpressionBitShift(parExpressionType)
 (<OPERATOR_BITWISE_XOR> ExpressionBitShift(parExpressionType))*
}

void ExpressionBitShift(ExpressionType parExpressionType):{}{
 ExpressionBitTest(parExpressionType)
 [(<OPERATOR_BITWISE_LSH> | <OPERATOR_BITWISE_RSH>) ConstantInteger()]
}

```

```

void ExpressionBitTest(ExpressionType parExpressionType):{}{
 (<OPERATOR_BITWISE_NOT>)* ExpressionValue(parExpressionType)
}

void ExpressionValue(ExpressionType parExpressionType):{}{
 ExpressionValueTerm(parExpressionType)
 ((<OPERATOR_PLUS> | <OPERATOR_MINUS>)
 ExpressionValueTerm(parExpressionType))*
}

void ExpressionValueTerm(ExpressionType parExpressionType):{}{
 ExpressionValueFactor(parExpressionType)
 ((<OPERATOR_MULTIPLY> | <OPERATOR_DIVIDE>)
 ExpressionValueFactor(parExpressionType))*
}

void ExpressionValueFactor(ExpressionType parExpressionType):{}{
 [Sign()] ExpressionValuePrimary(parExpressionType)
}

void ExpressionValuePrimary(ExpressionType parExpressionType):{}{
 Constant() | ExistsAttribute() |
 LOOKAHEAD(2) FunctionCall(parExpressionType) |
 ("(" Expression(parExpressionType) ")") |
 LOOKAHEAD({parExpressionType == ExpressionType.LOW_LEVEL_ALL })
 (<TYPE_ID> | <KEYWORD_GROUPTS> |
 Aggregate(parExpressionType) |
 OptionallyQualifiedIdentifier() /* PilotJoinField,
 LogicalObjectField */
)
 |
 LOOKAHEAD({parExpressionType == ExpressionType.HIGH_LEVEL_ALL})
 (Aggregate(parExpressionType) |
 OptionallyQualifiedIdentifier() /* WindowField */)
 |
 LOOKAHEAD({parExpressionType==pressionType.LOW_LEVEL_NO_AGGR})

```

```

 (<TYPE_ID> |
 OptionallyQualifiedIdentifier()
 /* PilotJoinField, LogicalObjectField */)
 |
 LOOKAHEAD(
 {parExpressionType==ExpressionType.LOW_LEVEL_NO_AGGR_NO_PILOT}
)
 (<TYPE_ID> | Identifier() /* LogicalObjectField */)
 }

/*****
* ELEMENTS LIST PRODUCTIONS
*****/
void FieldDefinitionList():{}{
 "(" FieldDefinition() ("," FieldDefinition())* ")"
}

void FieldDefinition():{}{
 Identifier() /* DataStructureField */
 FieldType() [<KEYWORD_DEFAULT> SignedConstant()]
}

void FieldList():{}{
 "(" Identifier() /* DataStructureField */
 ("," Identifier() /* DataStructureField */)* ")"
}

void EventList():{}{
 Identifier() /* LogicalObjectEvent */
 ("," Identifier() /* LogicalObjectEvent */)*
}

void CorrelatedTableList():{}{
 CorrelatedTable()("","CorrelatedTable()*
}

```

```

void CorrelatedTable():{}{
 Identifier() /* DataStructureName */
 <KEYWORD_ON> Expression(ExpressionType.LOW_LEVEL_NO_AGGR)
}

void WindowDefinitionList():{}{
 WindowDefinition() ("," WindowDefinition())*
}

void WindowDefinition():{}{
 Identifier() /* DataStructureName */
 ["(" WindowSize() ")"]
 [<KEYWORD_AS> Identifier() /* DataStructureName */]
}

void FieldGroupingByList():{}{
 (FieldGroupingBy() | FieldGroupingByTs())("," FieldGroupingBy())*
}

void FieldGroupingBy():{}{ Identifier() /* LogicalObjectField */ }

void FieldGroupingByTs():{}{
 <TYPE_TS> "(" Duration() "," ConstantInteger() <KEYWORD_GROUPS> ")"
}

void WindowFieldList():{}{
 OptionallyQualifiedIdentifier() /* WindowField */
 ("," OptionallyQualifiedIdentifier() /* WindowField */)*
}

/*****
STATEMENTS PRODUCTIONS
*****/
void Statement():{}{
 CreationStatement() |
 InsertionStatement() |
 ActuationStatement()
}

```

```
void CreationStatement():{}{
 <KEYWORD_CREATE> [<KEYWORD_OUTPUT>]
 (StreamCreationStatement() | SnapshotCreationStatement())
}

void StreamCreationStatement():{}{
 CreateStreamClause() [<KEYWORD_AS> StreamSelectionStatement()]
}

void SnapshotCreationStatement():{}{
 CreateSnapshotClause() [<KEYWORD_AS> SnapshotSelectionStatement()]
}

void CreateStreamClause():{}{
 <KEYWORD_STREAM> Identifier() /* DataStructureName */
 FieldDefinitionList()
}

void CreateSnapshotClause():{}{
 <KEYWORD_SNAPSHOT> Identifier() /* DataStructureName */
 FieldDefinitionList() <KEYWORD_WITH> <KEYWORD_DURATION> Duration()
}

void InsertionStatement():{}{
 <KEYWORD_INSERT> <KEYWORD_INTRO>
 (StreamInsertionStatement() | SnapshotInsertionStatement())
}

void StreamInsertionStatement():{}{
 InsertStreamClause() StreamSelectionStatement()
}

void SnapshotInsertionStatement():{}{
 InsertSnapshotClause() SnapshotSelectionStatement()
}
```

```
void InsertStreamClause():{}{
 <KEYWORD_STREAM> Identifier() /*DataStructureName*/ [FieldList()]
}

void InsertSnapshotClause():{}{
 <KEYWORD_SNAPSHOT> Identifier() /*DataStructureName*/ [FieldList()]
}

void StreamSelectionStatement():{}{
 (<KEYWORD_LOW> ":" LowEveryClause()
 LowSelectionStatement(ExpressionType.LOW_LEVEL_ALL))
 |
 (<KEYWORD_HIGH> ":" HighEveryClause()
 HighSelectionStatement(ExpressionType.HIGH_LEVEL_ALL))
}

void SnapshotSelectionStatement():{}{
 (<KEYWORD_LOW> ":"
 LowSelectionStatement(ExpressionType.LOW_LEVEL_ALL))
 |
 (<KEYWORD_HIGH> ":"
 HighSelectionStatement(ExpressionType.HIGH_LEVEL_ALL))
}

void LowEveryClause():{}{
 <KEYWORD_EVERY> (
 LOOKAHEAD(2)(
 Duration() [<KEYWORD_SYNCHRONIZED>])|(SamplesNumber()))
}

void HighEveryClause():{}{
 <KEYWORD_EVERY> (
 LOOKAHEAD(2)(Duration()) |
 (SamplesNumber()<KEYWORD_IN> Identifier()/*DataStructureName*/))
}
```

```

void LowSelectionStatement(ExpressionType parExpressionType):{}{
 SelectClause(parExpressionType)
 [GroupByClause()]
 [UpToClause()]
 [HavingClause(parExpressionType)]
 [OnEmptySelectionClause()]
 SamplingClause()
 [PilotJoinClause()]
 [ExecuteIfClause()]
 [TerminateAfterClause()]
}

void HighSelectionStatement(ExpressionType parExpressionType):{}{
 HighSelectionStatementDefinition(parExpressionType)
 (<KEYWORD_UNION> [<KEYWORD_ALL>]
 HighSelectionStatementDefinition(parExpressionType))*
 [OnEmptySelectionClause()]
}

void HighSelectionStatementDefinition(ExpressionType
parExpressionType) :{}{
 SelectClause(parExpressionType) FromClause()
 [WhereClause(parExpressionType)]
 [GroupClause() [HavingClause(parExpressionType)]]
}

void SelectClause(ExpressionType parExpressionType):{}{
 <KEYWORD_SELECT> [<KEYWORD_DISTINCT> | <KEYWORD_ALL>]
 FieldSelectionList(parExpressionType)
}

void FieldSelectionList(ExpressionType parExpressionType):{}{
 FieldSelection(parExpressionType)
 ("," FieldSelection(parExpressionType))*
}

```



```

void FieldSelection(ExpressionType parExpressionType):{}{
 Expression(parExpressionType) [<KEYWORD_DEFAULT> SignedConstant()]
}

void WhereClause(ExpressionType parExpressionType):{}{
 <KEYWORD_WHERE> Expression(parExpressionType)
}

void GroupByClause():{}{
 <KEYWORD_GROUP> <KEYWORD_BY> FieldGroupingByList()
}

void GroupClause():{}{
 <KEYWORD_GROUP>((<KEYWORD_BY> WindowFieldList())|(<KEYWORD_ALL>))
}

void HavingClause(ExpressionType parExpressionType):{}{
 <KEYWORD_HAVING> Expression(parExpressionType)
}

void UpToClause():{}{ <KEYWORD_UP> <KEYWORD_TO> WindowSize() }
void OnEmptySelectionClause():{}{
 <KEYWORD_ON> <KEYWORD_EMPTY><KEYWORD_SELECTION><KEYWORD_INSERT>
 (<KEYWORD_NOTHING> | <KEYWORD_DEFAULT>)
}

void SamplingClause():{}{
 <KEYWORD_SAMPLING>
 (
 OnEventClause()
 |
 (
 SamplingIfEveryClause()[OnUnsupportedSRClause()][RefreshClause()]
)
) [WhereClause(ExpressionType.LOW_LEVEL_NO_AGGR)]
}

```

```

void OnEventClause():{}{ <KEYWORD_ON> <KEYWORD_EVENT> EventList() }

void SamplingIfEveryClause():{}{
 (
 (SamplingIfClause() SamplingEveryClause())+
 [<KEYWORD_ELSE> SamplingEveryClause()]
)
 |
 (SamplingEveryClause())
}

void SamplingIfClause():{}{
 <KEYWORD_IF> Expression(ExpressionType.LOW_LEVEL_NO_AGGR)
}

void SamplingEveryClause():{}{
 <KEYWORD_EVERY>
 Expression(ExpressionType.LOW_LEVEL_NO_AGGR) TimeUnit()
}

void OnUnsupportedSRClause():{}{
 <KEYWORD_ON><KEYWORD_UNSUPPORTED><KEYWORD_SAMPLE><KEYWORD_RATE>
 (<KEYWORD_DO><OPERATOR_NOT><KEYWORD_SAMPLE>)
 |
 (<KEYWORD_SLOW><KEYWORD_DOWN>)
}

void RefreshClause():{}{
 <KEYWORD_REFRESH> (
 OnEventClause() | (<KEYWORD_EVERY> Duration()) | (<KEYWORD_NEVER>)
)
}

void PilotJoinClause():{}{
 <KEYWORD_PILOT> <KEYWORD_JOIN> CorrelatedTableList()
}

```

```
void ExecuteIfClause():{}{
 <KEYWORD_EXECUTE> <KEYWORD_IF>
 Expression(ExpressionType.LOW_LEVEL_NO_AGGR_NO_PILOT)
 [RefreshClause()]
}

void TerminateAfterClause():{}{
 <KEYWORD_TERMINATE> <KEYWORD_AFTER>
 (LOOKAHEAD(2) Duration() | SelectionsNumber())
}

void FromClause():{}{
 <KEYWORD_FROM> WindowDefinitionList()
}

void ActuationStatement():{}{
 SetClause()
 [OnClause()]
}

void SetClause():{}{
 <KEYWORD_SET> SetPairsList()
}

void SetPairsList ():{}{
 SetPair() ("," SetPair())*
}

void SetPair ():{}{
 Identifier() /* LogicalObjectField */
 <OPERATOR_EQUAL>
 Expression(ExpressionType.LOW_LEVEL_NO_AGGR_NO_PILOT)
}

void OnClause ():{}{
 <KEYWORD_ON>
 Expression(ExpressionType.LOW_LEVEL_NO_AGGR_NO_PILOT)
}
```



# Bibliography

- [1] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless sensor networks: a survey," *IEEE Communications Magazine*, vol. 40, no. 8, pp. 102–114, August 2002.
- [2] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless sensor networks: a survey," *Computer Networks*, vol. 38, no. 4, pp. 393–422, 2002.
- [3] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TinyDB: an acquisitional query processing system for sensor networks," *ACM Trans. Database Syst.*, vol. 30, no. 1, pp. 122–173, 2005.
- [4] <http://telegraph.cs.berkeley.edu/tinydb/tinydb.pdf>
- [5] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TAG: A Tiny Aggregation service for ad-hoc sensor networks," in *Proc. 5th USENIX OSDI*, Boston, MA, Dec. 2002.
- [6] D. Chu, L. Popa, A. Tavakoli, J. M. Hellerstein, P. Levis, S. Shenker, and I. Stoica, "The design and implementation of a declarative sensor network system," in *SenSys '07: Proceedings of the 5th international conference on Embedded networked sensor systems*. New York, NY, USA: ACM, 2007, pp. 175–188.
- [7] Y. Yao and J. Gehrke, "The cougar approach to in-network query processing in sensor networks," *SIGMOD Rec.*, vol. 31, no. 3, pp. 9–18, 2002.
- [8] P. Levis and D. Culler, "Maté: a tiny virtual machine for sensor networks," *SIGPLAN Not.*, vol. 37, no. 10, pp. 85–95, 2002.
- [9] T. Liu and M. Martonosi, "Impala: a middleware system for managing autonomic, parallel sensor systems," in *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM Press, 2003, pp. 107–118.
- [10] C. Srisathapornphat, C. Jaikao, and C.-C. Shen, "Sensor information networking architecture," in *ICPP '00: Proceedings of the 2000 International Workshop on Parallel Processing*. Washington, DC, USA: IEEE Computer Society, 2000, p. 23.
- [11] S. Li, S. Son, and J. Stankovic, "Event detection services using data service middleware in distributed sensor networks," in *Proceedings of 2nd Intl Workshop Information Processing in Sensor Networks (IPSN 03)*, ser. LNCS 2634. Springer, April 2003, pp. 502–517.
- [12] G. Amato, P. Baronti, and S. Chessa, "Mad-wise: programming and accessing data in a wireless sensor network," in *Proceedings of the International Conference on Computer as a tool EUROCON 2005*, 2005.
- [13] <http://mad-wise.isti.cnr.it>
- [14] R. Gummadi, N. Kothari, T. Millstein, and R. Govindan, "Kairos: A macroprogramming system for wireless sensor networks," in *Proceedings of the twentieth ACM symposium on Operating systems principles SOSP 05*, 2005.
- [15] <http://webdoc.siemens.it/CP/SIS/Press/SWORD.htm>

- [16] K. Aberer, M. Hauswirth, and A. Salehi, "Global sensor networks," School of Computer and Communication Sciences Ecole Polytechnique Federale de Lausanne (EPFL), Tech. Rep. LSIR-REPORT-2006-001, 2006.
- [17] G. Amato, A. Caruso, S. Chessa, V. Masi, and A. Urpi, "State of the art and future directions in wireless sensor networks data management," Istituto di Scienza e Tecnologie dell'Informazione del CNR, Pisa, Italy, Tech. Rep. ISTI-2004-TR-16, May 2004.
- [18] K. Henriksen and R. Robinson, "A survey of middleware for sensor networks: state-of-the-art and future directions," in *MidSens '06: Proceedings of the international workshop on Middleware for sensor networks*. New York, NY, USA: ACM Press, 2006, pp. 60–65.
- [19] S. Hadim and N. Mohamed, "Middleware: Middleware challenges and approaches for wireless sensor networks," *IEEE Distributed Systems Online*, vol. 7, no. 3, p. 1, 2006
- [20] <http://www.xbow.com>
- [21] <http://www.intel.com/research/exploratory/motes.htm>
- [22] <http://www.tinyos.net>
- [23] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: A holistic approach to networked embedded systems," in *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. New York, NY, USA: ACM Press, 2003, pp. 1–11.
- [24] M. Weiser, "The computer for the 21st century," *Scientific American*, vol. 265, no. 3, pp. 66–75, September 1991.
- [25] T. Kindberg and A. Fox, "System software for ubiquitous computing," *IEEE Pervasive Computing*, vol. 1, no. 1, pp. 70–81, January 2002. [Online]. Available: <http://dx.doi.org/10.1109/MPRV.2002.993146>
- [26] F. Bombardieri and A. Kumar, "Implementation of a time synchronization protocol for wireless sensor networks," *Report of Technologies of Information Systems course project, Politecnico di Milano*, 2006.
- [27] P. Mohapatra, J. Li, and C. Gui, "Qos in mobile ad hoc networks," *IEEE Wireless Communications*, pp. 44–52, June 2003.
- [28] J. Qingchun, "A framework for supporting quality of service requirements in a data stream management system," Ph.D. dissertation, Faculty of the Graduate School of the University of Texas at Arlington, August 2005.
- [29] J. Melton and A. R. Simon, *SQL: 1999 - Understanding Relational Language Components*. Morgan Kaufmann, 2001, ch. 2.
- [30] F. Crivellari, F. dalla Libera, S. Frasson, and F. A. Schreiber, "Computation of statistical functions in distributed information systems," *Inf. Syst.*, vol. 8, no. 4, pp. 303–308, 1983.
- [31] D. Talia, P. Trunfio, S. Orlando, R. Perego, and C. Silvestri, "Systems and techniques for distributed and stream data mining," Institute on Knowledge and Data Management, CoreGRID - Network of Excellence, Tech. Rep. TR-0045, July 2006.
- [32] C. Cappiello, F. Schreiber "Quality- and Energy-Aware Data Compression by Aggregation in WSN Data Streams". In *Proceedings of Fifth IEEE International Workshop on Sensor Networks and Systems for Pervasive Computing (PerSens 2009)*.
- [33] S. C. Reghizzi, *Linguaggi formali e Compilazione*. Pitagora, 2006, pp. 1–368.
- [34] <https://javacc.dev.java.net/>