TÉCNICO LISBOA

Real-Time Integration of Building Energy Data

Diogo Gonçalo Silva dos Anjos

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Professor Paulo Jorge Fernandes Carreira

Examination Committee

Chairperson:	Professor Miguel Nuno Dias Alves Pupo Correia
Supervisor:	Professor Paulo Jorge Fernandes Carreira
Member of the Committee:	Professor Mário Jorge Costa Gaspar da Silva

November 2015

À minha mãe Maria e ao meu pai Carlos, por terem investido tudo
na minha liberdade intelectual e disciplina de pensamento.

“It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness, it was the epoch of belief, it was the epoch of incredulity, it was the season of Light, it was the season of Darkness, it was the spring of hope, it was the winter of despair, we had everything before us, we had nothing before us ...”

— Charles Dickens, *A Tale of Two Cities*

Agradecimentos

"*No Man is an Island*"
— John Donne

Tudo na vida se resume a trabalho de equipa, por isso gostaria de deixar aqui algumas palavras de agradecimento a todos aqueles que, de uma forma ou de outra, considero terem sido essenciais para o meu desenvolvimento académico e pessoal — sem vocês, nada disto teria sido possível.

Em primeiro lugar, como não poderia deixar de ser, gostaria de expressar o imenso sentimento de admiração e gratidão que tenho pelos meus pais, Maria José Anjos e Carlos Anjos. Foi o espírito de sacrifico e tenacidade com que ambos me educaram que me permitiu chegar até aqui, estou-vos eternamente grato. É a vocês que dedico este trabalho.

Ao meu orientador, Professor Paulo Carreira, pela exigência e sentido critico com que me guiou na execução deste trabalho, que, juntamente com os seus elogios e incentivos, me levou a realizar uma tese de mestrado em que parte dos resultados alcançados foram cientificamente publicados.

Também ao INESC-ID, especialmente ao Data Management and Information Retrieval (DMIR) Group, tanto por ter financiado[1] parcialmente esta investigação, como também pelo ambiente e condições de trabalho que me proporcionou. Um agradecimento especial ao Professor Alexandre P. Francisco, por todas as discussões frutuosas que tivemos no âmbito deste projeto, as quais foram essenciais para melhor compreender as implicações e consequências dos resultados alcançados.

Aos meus grandes amigos de batalha, Pedro Costa e Pedro Carneiro, por terem estado sempre presentes, nos bons e nos maus momentos. Pelo espírito de entreajuda, pelo companheirismo e pela paciência mutua que sempre marcou o nosso grupo, mesmo nos momentos de maior pressão. Enfim, pela nossa amizade e por tudo aquilo que vivemos juntos na nossa passagem pelo Técnico. Foi um prazer enorme trabalhar com vocês.

Ao Instituto Superior Técnico, pela vanguarda, dinamismo e exigência com que a escola me soube formar desde o primeiro dia de aulas — foi o momento de viragem mais marcante da minha vida.

Ao meus país, Portugal, em especial ao sistema de ensino público, por, perante tantas encruzilhadas, continuar a lutar pelo ideal de uma escola de acesso livre, de ensino plural e exigente. Ideais aos quais eu devo muito a pessoa que sou hoje, e sem os quais não é possível construir uma sociedade livre e desenvolvida.

A todos vocês, o meu sincero agradecimento.

Instituto Superior Técnico
Lisboa, Outubro de 2015

[1] This work was partially supported by Fundação para a Ciência e Tecnologia (FCT) under the research project DATASTORM, EXCL/EEI-ESS/0257/2012 and PEstOE/EEI/LA0021/2013.

Resumo

Os Sistemas de Gestão de Energia (SGE) são usados para supervisionar o consumo de energia em edifícios, permitindo melhorar os seus índices de eficiência energética através da rápida deteção e correção de situações anómalos. Os SGE monitorizam a rede de medidores energéticos instalada no edifício, que está continuamente a produzir medições — data streams —, que refletem a variação do consumo ao longo do tempo, sendo processadas por cada SGE de forma a produzirem informação relevante ao plano de gestão energética do edifício. Estes dados devem ser processados em tempo *quase* real (aqui abreviado para *tempo real*), de forma a reduzir o período que medeia entre a deteção de um problema e a sua resolução — reduzindo o seu custo. O processamento de dados nos SGE é suportado por SGBD, que apenas processam dados armazenados em disco, introduzindo latências proibitivas à avaliação de dados em tempo real. Além disso, a linguagem SQL também não é a mais adequada para processar sensor data streams. Ou seja, a capacidade dos SGE para monitorizar o consumo energético em tempo real é seriamente afetada pelo uso de SGBD. Os Sistemas de Gestão de Data Streams (SGDS) existem para ultrapassar os problemas levantados pelos SGBD, processando data streams eficientemente. Em muitos domínios aplicacionais, a monitorização das respetivas redes de sensores é suportada por SGDS, obtendo com isso grandes melhorias de desempenho; no entanto, o mesmo conceito ainda não foi aplicado aos SGE. Este trabalho mostra que os SGE, para conseguirem processar data streams em tempo real, necessitam de ser suportados por SGDS. Propõe-se uma Arquitetura de Processamento de Dados suportada por um SGDS, que ilustra a implementação de um SGE capaz de monitorizar redes de medidores energéticos em tempo real. A solução foi validada por comparação com uma arquitetura baseada num SGBD, tendo os resultados superado o desempenho da arquitetura baseada no estado da arte, tanto na latência de avaliação dos dados como na capacidade da linguagem em expressar queries do domínio.

Palavras-chave: Sistemas de Gestão de Energia em Tempo Real, Monitorização de Redes de Medidores Energéticos, Processamento de Data Streams, Queries em Dados de Sensores, Eficiência Energética em Edifícios, Avaliação de SGDS e SGBD.

Abstract

Energy Management Systems (EMSs) are used to monitor energy consumption in buildings with the purpose of improving energy efficiency, by identifying savings opportunities and misuse situations. To achieve that, an EMS collects energy metering data streams from a network of energy meters deployed in a building. Sensor data must be processed in (*near*) real-time, to support a timely decision making process. Currently, EMSs are using traditional DBMSs to process these data, introducing a persistence step that translates to an unacceptable latency on data evaluation. Moreover, sensor data monitoring queries are not elegantly supported by the SQL query language, thus hampering the ability of an EMS to process energy metering data in real-time. Data Stream Management Systems (DSMSs) are used to process data streams efficiently in several domains. Many sensor network monitoring applications have been implemented upon DSMSs resulting in significant improvements on performance and overall resource usage. This thesis validates the hypothesis that, to process energy metering data streams in real-time, EMSs should be supported by DSMSs, instead of DBMSs. We introduce an EMS's Data Processing Architecture supported by a DSMS that supports the implementation of an EMS capable of performing real-time data processing. We validate our solution through a comparative evaluation against a DBMS based architecture. The results show that the DSMS-based EMS outperformed the state of the art approach, both in data evaluation latency and query language expressibility—demonstrating its adequacy to process energy metering data streams in real-time.

Keywords: Real-Time Energy Management Systems, Monitoring of Energy Metering Networks, Data Stream Processing, Sensor Data Querying, Buildings Energy Efficiency, DSMS and DBMS Benchmark.

Contents

List of Tables

List of Figures

Glossary

API	Application Programming Interface
CEP	Complex Event Processing
CQ	Continuous Query
DBMS	Database Management System
DPR	Datapoint Reading
DSMS	Data Stream Management System
DSPE	Data Stream Processing Engine
EMS	Energy Management System
EPL	Event Processing Language
ER	Entity-Relationship
ESPE	Event Stream Processing Engine
FIFO	First In, First Out
IoT	Internet of Things
JDBC	Java Database Connectivity
OLAP	Online Analytical Processing
OLTP	Online Transaction Processing
QoS	Quality of Service
SPE	Stream Processing Engine
pgSQL	PostgreSQL

Chapter 1

Introduction

The European Union has to import nearly 54% [Eurostat, 2013] of its energy demands while its greatest resource, energy efficiency, remains untapped [Commission, 2012]. To explore this asset, numerous initiatives have been put into place to reduce energy requirements in a manner that does not harm productivity (e.g. *Energy 2020* initiative [Commission, 2011]). Buildings account for 40% of energy consumption, ahead of other sectors, such as industry or transportation [Pérez-Lombard et al., 2008]. Therefore, small improvements on building energy consumption translate to major savings.

Energy efficiency in buildings can be achieved through: energy conservation (e.g. preserve the temperature of an heated room), equipment efficiency (e.g. replace incandescent light bulbs by lights made of LED), and intelligent energy management [Chwieduk, 2003]. This last topic concerns the monitoring of energy consumption and the careful tracing of its usage, in order to enable building managers to identify saving opportunities. Such monitoring is performed by an Energy Management System (EMS), through the gathering of data from the building energy metering network. The gathered sensor data is organized across several dimensions such as time, area, occupation, equipment state, expected consumption, among others, and then analyzed, to determine energy usage patterns. The information produced by the EMS is a crucial insight to determine the adjustments required to improve energy usage [Granderson et al., 2011].

One fundamental aspect of energy management is timeliness: faster decisions translate to less waste and larger savings. In other words, up-to-date information greatly improves the decision making process, because building managers are able to immediately diagnose and promptly respond to anomalous situations [Copin et al., 2010]. As depicted in Figure 1.1, (Near) Real-Time[1] Decision Making Applications are time-critical, and aspire at detecting volatile events (that have a very short lifespan), requiring (Near) Real-Time integration of a huge quantity of data, wherein each record relates to a very short period. This means that, EMSs must be capable of continuously processing massive quantities of energy-related data in real-time to improve the decision making process of building managers towards energy efficiency. However, there is a set of circumstances preventing this from being possible.

In recent years, many advances have been made in sensor technology, making it affordable and

[1] Throughout this document, the terms “Real-Time” and “(Near) Real-Time” are used interchangeably to denote the ability of a system to deliver up-to-date information.

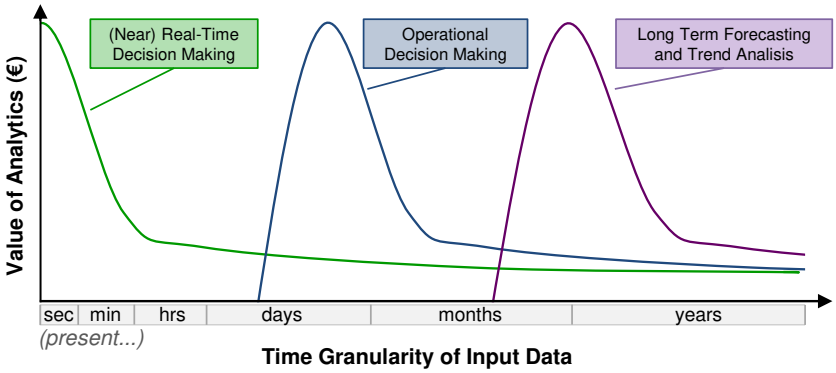

Figure 1.1: Optimal input data granularity per application domain (adapted from [Chandramouli et al., 2010]). For better results, (Near) Real-Time Decision Making Applications require data to be sampled in short periods, from seconds to minutes (left). Operational Decision Making Applications require data ranging from days to months (centre). Long Term Forecasting and Trend Analysis require data granularity that varies from months to years (right).

widely available, forming what we know as the *"Internet of Things"* (IoT) [Gubbi et al., 2013]. Such ubiquity of sensors is leading to pervasive sets of sensor data that, by being so large and complex, are not suitable of being timely processed by the traditional data processing systems, such as DBMSs [Golab and Özsu, 2003]. In fact, this problem is requiring a lot of attention from the community, since it belongs to a trend of issues known as the challenges of Big Data, which are imposing a paradigm shift on how data is being handled [Beyer and Laney, 2012]. The data processing architecture of an EMS is generally supported by a traditional DBMS [Kazmi et al., 2014, Section 3; Ma et al., 2010; Granderson et al., 2009, Section 2.2.2], which, as we already said, is not capable of processing sensor data streams in real-time. Therefore, we believe that existing EMSs are not prepared to provide useful energy management information in a timely manner, neither their software architecture or functionalities are conceived to be a truly real-time data processing system [Anjos et al., 2014]. That is, sensor data driven applications, such this one, are struggling to cope with a set of emerging challenges for which they were not initially conceived—processing large volumes of data streams in real-time. The lack of standard solutions to address these requirements is forcing the community to rethink the data processing infrastructure of these applications.

A set of disruptive solutions have been developed to address the challenges of Big Data. The Data Stream Management Systems (DSMSs) are the proposed solution to effectively process data streams, such as energy metering sensor data [Babcock et al., 2002; Stonebraker et al., 2005]. Across different domains, several monitoring applications that are sensor data driven, are being developed upon a DSMS (instead of a DBMS). For instance, the monitoring of data related to: stock market transactions [Chandramouli et al., 2010; Mukherjee et al., 2010], network traffic [Akhtar and Siddiqui, 2011], healthcare [Jiang et al., 2011; Zhang et al., 2010] and environment [Li et al., 2008]. Suggesting that this same approach must be followed to develop energy management applications.

This work shows that current EMSs, by being DBMS-supported, are not prepared to process energy metering data in real-time; having, for this purpose, to be supported by a DSMS, since these provide better performance and a more suitable query language to fulfil the requirements of this domain. By pointing out how to develop an EMS capable of evaluating sensor data streams in real-time, we contribute with a solution that states how energy management applications must be developed in order to

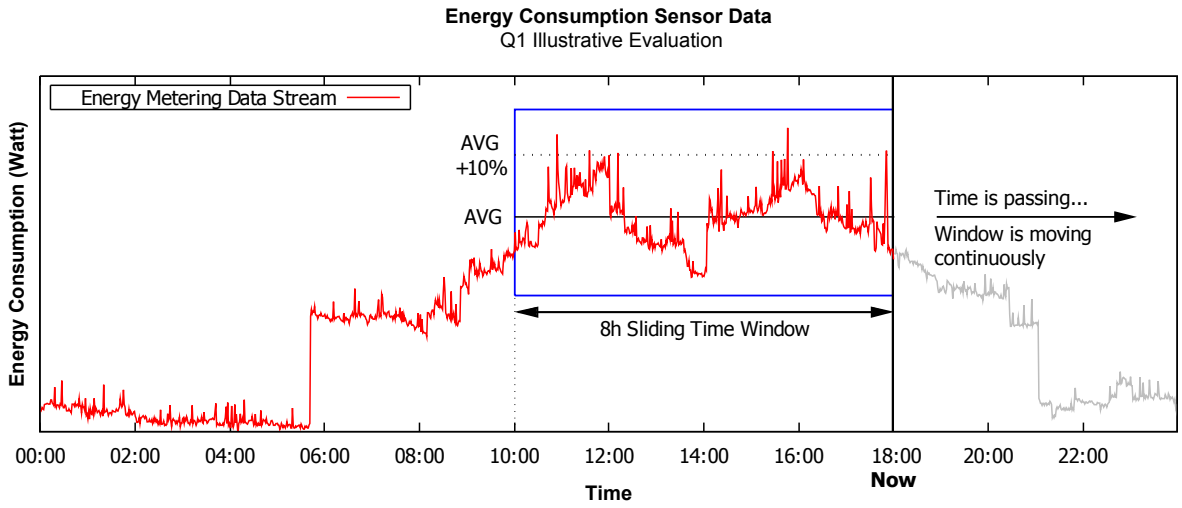

Figure 1.2: Illustrative evaluation of query Q1. The 24-hours energy consumption sample shows the volatility of energy metering data over time. The data stream covered by the *"last-8h-from-Now"* time window is changing (sliding) continuously, imposing a continuous evaluation of Q1 (i.e. AVG has to be repeatedly re-evaluated over an 8-hours data stream). Otherwise, it would be difficult to keep the query result updated, given the sensor data volatility.

overcome the issues of Big Data, and thereby capable of providing useful energy management information in a timely manner. Essential to improve energy efficiency in buildings and achieve major energy savings.

1.1 Motivation

As pointed out earlier, EMSs are highly dependent on querying data stream in real-time, so they can monitor the building energy consumption. However, a DBMS is an inadequate solution to handle this requirement. To understand why, consider the following queries:

```
Q1: Identify the places that are reporting an energy consumption 10% above
    the respective average over the last 8 hours.

Q2: Identify the faulty energy meters that are not being able of produce an
    energy consumption measurement every 60 seconds.

Q3: For each place, tell me how energy consumption is varying (%) from the
    average over the last 5 minutes.

Q4: For each place, give me its current and expected energy consumption.
    Being the expected value of each place given by the average consumption
    of the current hour, computed along last (sliding) month.
```

The evaluation of query Q1 is depicted in Figure 1.2, and, as we can see, these queries are distinguished by having to process time-variant data (sensor time-series), which leads to the following requirements: (i) queries must be evaluated *continuously* as time goes by, in order to keep the evaluation results updated (note that, *"the last 8 hrs"* of Q1 is not a static interval and is always changing); (ii) the time it takes to evaluate the query—*query evaluation latency*—should be small enough to allow the processing of new data as fast as it arrives—that is, in real-time; and (iii) queries must be capable of computing data aggregates over *moving subsets* of the data stream (such as the AVG of Q1 over the 8 hrs *sliding time window*).

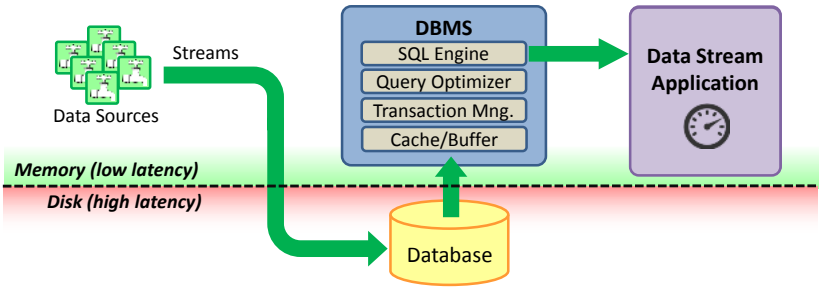

Figure 1.3: Data stream processing using a DBMS (adapted from [Chakravarthy and Jiang, 2009, p.2-4]). Data streams have first to be persisted in disk (high latency) before could be retrieved and processed in main memory (low latency). The disk I/O latency is prohibitive for data stream applications with real-time requirements.

DBMSs are not a suitable solution to cope with these requirements. They only run queries over persistent data, meaning that, to be queried, sensor data streams have first to be stored in disk to later be retrieved and evaluated in main memory (see Figure 1.3). This obligation of persist data before it can be processed imposes an unacceptable disk I/O latency for many data streaming applications, including EMSs. Being this a major performance bottleneck for data intensive applications such this one [Babcock et al., 2002]. In addition, the DBMS query language model is not the most appropriate to pose this type of queries. As we said, these queries must be running continuously, which strongly differs from the *one-time queries* that are provided by the DBMS. These type of queries was designed to manage static data held in DBMSs, having to be explicitly executed by the application, instead of spontaneously reacting to changes in the (time-variant) dataset. Moreover, queries are evaluated in a batch manner, having the entire dataset to be evaluated before the output result could be produced, which is not the most adequate strategy to process (potentially unbounded) data streams. Finally, DBMS's SQL language lacks operators to effectively manage time-variant data, such as time window operators, required to specify both the subset of the data stream under evaluation and how it must be dynamically updated.

DSMSs are in a better position to handle these type of queries, since they were specifically conceived to process continuous data streams. They do not require the data to be persistent in order to be evaluated, the arriving data is placed directly in memory to be processed on-the-fly by online queries (see Figure 1.4). By avoiding disk access overheads, DSMSs are capable of achieving query evaluation latencies that do not compromise the ability of the application to process data in real-time. Moreover, the query language model of a DSMS is the one that best suits the requirements of the monitoring queries under consideration. They provide *continuous queries*, a type of query which is in permanent evaluation, in order to promptly reflect any changes that occur in the (time-variant) data streams [Babu and Widom, 2001]. Continuous queries evaluate the arriving data streams incrementally, outputting a result for each data stream tuple that is evaluated. That is, the query output is maintained (or updated) incrementally, as new data is being evaluated: this makes sense, since we are in the presence of potentially unbounded data streams. Regarding the query language expressibility, DSMS's EPL (Event Processing Language) is rich in time related operators, such as the sliding time window of Q1, which simplifies the implementation of queries on time-variant data. As a result, DSMSs' continuous queries were specifically designed to process sensor data in real-time.

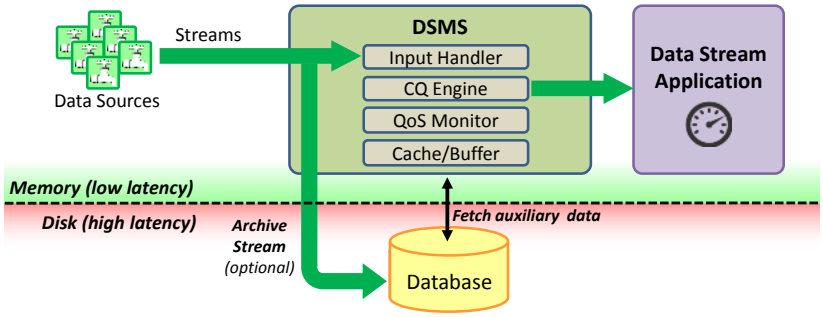

Figure 1.4: Data stream processing using a DSMS (adapted from [Chakravarthy and Jiang, 2009, p.2-4]). To be processed, data streams are placed directly in main memory. By bypassing disk I/O latencies, DSMSs are prepared to process data streams in real-time. Yet, this does not impedes data from being (optionally) persisted in disk, in order to make it available for future reference.

All this discussion leads us to the claim of this work. That current EMS solutions are not capable of processing energy metering data in real-time, since they rely on a traditional DBMS. Having for that purpose to be supported by a DSMS, since it outperforms a DBMS both in the query evaluation latency as in the query language expressibility.

1.2 Problem Statement

EMSs must be capable of process energy metering data in a timely manner, in order to provide up-to-date information able to improve energy efficiency in buildings.

The problem identified by this work is that: current EMSs are not capable of process energy metering data in real-time, since that, for that matter, they are supported by traditional DBMSs [Kazmi et al., 2014], which are known for its unsuitability to timely process sensor data streams, and lack of expressibility to pose sensor monitoring queries [Babcock et al., 2002]—seriously hampering the ability of an EMS to efficiently manage energy consumption in buildings. To solve this problem we propose the hypothesis that, an EMS supported by a DSMS would perform better on processing energy metering data, than the state of the art solutions supported by a DBMS. By better performance, we mean the ability to: (i) process energy metering data streams in real-time and (ii) provide a most suitable query language to evaluate these type of data within the requirements of this domain. The goal of this work was to assess and validate this hypothesis, through the development of an EMS's Data Processing Architecture supported by a DSMS, and a comparative evaluation of its performance against a DBMS based solution.

1.3 Methodology and Contributions

To achieve the goals proposed by this work, of introduce a novel EMS's Real-Time Data Processing Architecture supported by a DSMS, we conduct our research through the following methodology:

1. **Requirement Analysis.** Review the literature in order to identify the requirements of both sensor network monitoring queries and building energy management techniques demanding real-time data evaluation. The outcome was a representative set of *use-case queries* (scenarios) on the domain

of real-time monitoring of energy metering networks, comprising the type of data transformations that the proposed solution must be able to compute in a timely manner.

2. **Case Study.** Survey the properties of the energy metering data streams produced in the context of a real scenario—the IST Taguspark campus energy metering network. Several properties of sensor data were assessed, such as: the data structure, stream periodicity, number of network nodes, meters and network fault model, and other data quality dimensions. As a result, a *data schema* to model this domain in our solution was developed; we also gathered an extensive sample of energy metering data form the network, with the purpose of later testing our solution with a *real dataset*.

3. **EMS's Real-Time Data Processing Architecture.** Assess the related work on stream processing engines, to select the DSMS that most suits our needs; and data processing architectures aiming at process data in real-time. From this research, we developed the *proposed solution* for an EMS's Data Processing Architecture that, by being supported by a DSMS, is able of process energy metering data streams in real-time.

4. **Comparative Evaluation and Validation.** To validate our solution, two prototypes of the proposed architecture were implemented. The first, supported by a DSMS, represents our proposed solution, and the second, supported by a DBMS, represents a state of the art based solution. A comparative evaluation between the two prototypes was conducted. Having been evaluated the expressibility of each prototype query language to implement the *use-case queries* (earlier identified), and the ability of each solution to evaluate these queries in real-time, over the *energy metering dataset* previously gathered (see above). The DSMS solution outperformed the DBMS solution in both dimensions.

The **main contributions**[2] achieved by this work are as follows:

- We validate this work hypothesis: That, in order to process energy metering data streams in real-time, an EMS must be supported by a DSMS, and not by a DBMS, as they currently are. Being the inability of the DBMS to perform this task clearly highlighted by the benchmark results.

- We proposed an EMS's Real-Time Data Processing Architecture, which points out how energy management applications should be implemented so they can monitor buildings' energy consumption in a timely manner, towards energy efficiency. Moreover, the implementation feasibility of such architecture was validated by the respective prototype that we implemented.

- We give a comprehensive explanation of why a DBMS is not capable of properly monitor energy metering data, by identifying both the problems related with the query language and the performance bottlenecks of the query engine. This is relevant because, in the domain of energy management applications, recent literature [Kazmi et al., 2014] still points out DBMSs as a valid solution to support this feature.

[2] Part of the contributions achieved by this work have been published [Anjos et al., 2014].

- We identify the requirements of the class of queries used to monitor energy metering data. We propose a classification schema to catalogue these queries according to the identified features, that contributes to better understand the type of data transformations that must be taken into account by these applications.

Finally, the **lesson learned** from this research work, is that: energy management applications, aiming at monitoring energy metering data in real-time, must, for that purpose, be developed upon a DSMS.

1.4 Document Organization

The remaining of this manuscript is structured in six chapters. In Chapter 2, we introduce the fundamental concepts of streaming data processing, together with the general architecture and features of an EMS. To provide the background knowledge required for the coming sections. In Chapter 3, we survey the related work on data stream processing systems, to assess the type of DSMSs that are available, and how they can support our solution. Chapter 4, presents and details our proposed solution for an EMS's Real-Time Data Processing Architecture. Moreover, it also introduces the requirement analysis on the type of data transformations that must be performed by the system, together with the case study of the real building energy metering network that was assessed to validate the solution. Chapter 5, discusses the evaluation process of the proposed solution, introduces the results of such evaluation, and discuss them towards validating the thesis proposed by this work. Finally, Chapter 6, summarizes the goals of this work, together with the methodology used to achieve them. Details the contributions and lessons learned achieved along the research, points out how they affect the state of the art, and, to finalise, outlines directions for further research.

Chapter 2

Research Background

In this chapter, we introduce the background knowledge required to understand the work developed along this research. We review the literature on the concepts of stream data processing, to point out the requirements, properties, and techniques that should be taken into account to address the issues of sensor data processing. In addition, we survey the literature on Energy Management Systems (EMSs), in order to identify their main features and architectural components, essential to realize how state of the art solutions are dealing with the requirement of process energy metering data in real-time. It was this survey that points out traditional DBMSs as the current approach to support the data processing architecture of an EMS, which, as we will see, is an inadequate solution to process sensor data streams in real-time—leading us to the research problem of this work.

2.1 Concepts of Data Stream Processing

There is a large set of data intensive applications for which the traditional approach of handling and processing data through a DBMS is not feasible. This happens due the distinguishing characteristics of the data produced by the data sources—sensor networks—of these data driven applications. Each sensor of the network is *continuously* producing new data, forming a *sequence* of time-variant sensor measurements (a sensor time-series), which is known as a *data stream*. To be more specific, let us to define a data stream in the following manner (adapted from Sathe et al. [2013]):

Data Stream. A *sequence* of **data tuples** $(t_i, V_i)_{s \in S}$, where $V_i = (v_{i1}, v_{i2}, ..., v_{in})$ is the sequence of values produced by the n measurement points of sensor s of sensor network S at time t_i.

An example of a data stream is depicted in Figure 1.2, which illustrates a 24-hour data stream produced by an energy meter, showing the energy consumption variation over that period.

There are several types of data sources producing data streams, such as sensor networks providing RFID readings, social network publications, health related measures from biodevices, and energy consumption measurements from building energy metering network. All these data sources impose a common requirement: data streams have to be processed in real-time by the data stream monitoring application, in order to provide up to date information on what is happening in the sensor network.

As we said, traditional DBMSs are not an adequate solution to support data stream processing. They can only manage previously persisted data, such intermediate step of persistence introduces an unacceptable penalty on data evaluation latency, hampering the ability to evaluate the data stream in a timely manner. Moreover, the SQL query language is not prepared to support the class of queries required by data stream applications: continuous queries capable of provide a reach set of time related operators. Data Stream Management Systems (DSMSs) were developed to efficiently process data streams, they differ from DBMSs by being capable of process data streams continuously, through continuous queries, and in memory, without have to previously persist data in a database [Babcock et al., 2002]. The main requirements of data stream processing that imposes a distinction on how a DBMS and a DSMS process their data are detailed below.

2.1.1 Requirements of Real-Time Data Processing

The data stream computational model greatly differs from the traditional database model, making the existing DBMS unsuitable for processing data streams, they are not prepared to timely and continuously process data streams. DSMSs were specifically designed to process data streams, by effectively addressing the following requirements on real-time data stream processing [Golab and Özsu, 2003]:

Arrival Order. The system should allow to express window queries, the ones that let to specify data transformations over data stream tuples by their arrival order. For instance, evaluate a query just over the last fifty arriving tuples, or over the tuples that have arrived in the last ten minutes.

Bounded Memory. In most cases, is impossible to store all data stream in memory, the system must be able to maintain data structures that summarize the data stream seen so far (see below). Queries lying on these data summaries may not produce exact answers, but rather approximated ones (still accepted as a valid answers if the error is small enough).

Blocking Operators. Since we are in the presence of potentially unbounded data streams, query aggregate operators could not be evaluated in a blocking manner (i.e. have to see all input data before be able of produce an output), they must be capable of evaluating data in an online manner. Otherwise, the query will fail to produce any output in the presence of an unbounded data stream.

See Data Only Once. Due the semantics of the data model and restrictions on performance and memory consumption, data processing stream algorithms must be capable of evaluate a data stream by making a single pass through the data stream (that is, see each tuple only once).

Timely Data Evaluation. Applications aiming at monitoring a sensor network (such as, an EMS) must be capable of processing sensor data streams in real-time. Sensor data is continuously being updated (produced) and greatly changes over time, thus it must be evaluated as soon as it becomes available in order to produce up to date information on the sensor network current state and behaviour.

Graceful Degradation. During periods of massive overload the system must be capable of adapt itself (e.g. by reducing the accuracy of produced answers or by carefully discarding some tuples), in order to maintain or, at most, slightly degrade its performance in order to maintain its ability of evaluate data in real-time.

Scalability. The system must be capable of scale its performance in the presence of a scenario of increased workload by maintaining its performance, or then, be prepared to be easily rearranged in order to be capable of handling such growing amounts of work without declining its performance (in most cases, this means to be able of expanding its computation through a distributed environment).

2.1.2 Queries Over Data Streams

Data stream processing imposes a set of requirements which are hard to achieve through the type of queries provided by DBMSs—the one-time queries. To elegantly cope with these requirements, the DSMSs provide a most suitable class of queries to enhance the querying of data streams—the continuous queries. The two types of queries diverge from each other as follows [Babcock et al., 2002]:

One-Time Queries. Is the DBMS type of queries, the query execution life-cycle is summed-up to an one-time evaluation. More specifically, query will consume and evaluate all the dataset hold by the database (following an evaluation approach that is all-dataset oriented) to then output a single result set, reflecting the current state of the database according to query semantic. That is, the query evaluates all the dataset in a batch manner and outputs an unique snapshot with the results.

Continuous Queries. Is the class of queries provided by DSMSs, that, unlike one-time queries, are continuously evaluating the arriving data stream. The query evaluates a single data stream tuple at a time (following an incremental evaluation approach that is single-tuple oriented), and for each evaluated tuple an output tuple is produced. That is, the query evaluates the arriving data stream in a streaming manner (tuple-a-tuple), producing an result tuple for each evaluated tuple, producing then an output stream (instead of a single snapshot, such as one-time queries).

Regarding continuous queries (CQs), they start to evaluate data and produce results since the very moment they are installed in the DSMS. Thus, the results of evaluating a data stream are dependent on the query installation time, being this feature used to distinguish between the following two types of CQs:

Predefined Queries. Are the queries installed in the DSMS *before* it starts to process any data stream. By knowing the query specification in advance, the system is able to create a more efficient query evaluation plan, capable of reduce the data evaluation latency and enhance the accuracy of the produced results.

Ad-hoc Queries. Are the queries installed in the DSMS *after* it has already begun to process a data stream. Since the query specification is known only after some data have already passed and the optimization mechanisms have already been configured, the query evaluation latency may be penalized and the produced results may not be so accurate as the ones of predefined queries.

2.1.3 Memory Limitations and Unbounded Data Streams

By definition a data stream is potentially unbounded in its size, yet the same is not true for the amount of memory that is made available to process it. Meaning that we must find a solution to process data streams within a limited amount of main memory (i.e. primary storage). Otherwise, it will be impossible to process data streams in real-time, since the latency penalty of disk accesses (secondary storage) will not allow such thing, and even the disk is limited in their storage. However, process unbounded data streams within limited amounts of memory is a challenging a task. For instance, it is impossible to set a limitation for the memory that is required to join two unbounded data streams. Therefore, we must resort to some techniques in order to deal with unbounded data streams and memory limitations, which have the inevitable disadvantage of, in some cases, do not be possible of provide accurate results.

All the memory contention techniques, in on way or another, lie on the concept of "synthesizing" the full data stream (that does not fit in memory) into a dataset able to be managed in main memory, which as a consequence may lead to a reduction in the accuracy of the final results (although they continue to be produced in real-time). However, for the vast majority of data stream applications, an approximated answer (within an acceptable margin of error), rather than the exact one, is generally enough, since they are produced in real-time. The main techniques used to reduce the amount of memory required for data stream processing, are stated bellow [Babcock et al., 2002]:

Windows. One of the main techniques is to identify a sub-part of the entire data stream and then evaluate just this (smaller) sub-part. The concept of window is used to delimit the part of the data stream that must be evaluated, being discarded the data stream tuples that do not take place inside the window. An essential property of such approach is to specify the windows moving behaviour, that is, the policy which states how data stream tuples enter and leave the window. Different window moving behaviours lead us to several types of windows, for instance: in time based windows, the tuples inside the window change according to the elapsed time, while in size based windows, the tuples inside the window change according the quantity of tuples that are arriving to the system, etc. There are many types windows, see Section 2.1.6 for details.

Synopsis Data Structures. An other common technique is to maintain a synopsis of the arriving data stream, a type of data structure with a small memory footprint which works like a brief summary of the stream, and is used by the query engine to quickly provide the required information about the data stream in order to produce (typically approximated) query answers. In this way, the data stream is only used to keep the synopsis updated, may then the stream be discarded from memory, since the (unlimited) data stream is being represented by the (memory bonded) data synopsis. A data synopsis may be implemented in many different ways, being the usage of statistical techniques, to extract the statistical distribution of the data stream, some of the most used, for instance: **Histograms** and **Sketches** techniques (see Gama and Rodrigues [2007] for details). **Wavelets** are another method to aggregate information from unbounded data streams, they are a mathematical transformation that represents the data streams as a weighted sum of simpler data streams. This data stream decomposition does not imply information loss, the original data stream may

be reconstructed from the entire set of coefficients. However, to summarize the data stream, the "smallest" coefficients—those with lower impact on data stream reconstruction—may be removed. This will suppress small details in the reconstructed data stream, yet the most distinctive properties of the data stream will be preserved [Gama and Rodrigues, 2007].

Load Shedding. Also known as Sampling Processing, this technique is applied in the scenarios where the data stream arrival rate is greater than the data consumption rate—that is, when the system is receiving more data tuples than the mount of tuples that it can process. In such overwhelming scenario, the system input queue will start to grow endlessly until reach the limit of available memory, making no sense to try to process all the queued data stream tuples. Alternatively, it is more appropriate to start to discard some tuples from the queue, which will work as an attempt to "slow down" the data stream arrival rate, by artificially decreasing the data stream arrival rate. As a consequence, the query will be evaluated from a sample of the entire data stream, which will result in approximated answers, instead of accurate ones. This is one of the most simplest methods to summarize a data stream, solving many problems related to huge massive workload scenarios. Being the challenge of Load Shedding algorithms the ability of identify the optimal data stream tuple to be discarded: the one with the least contribution to the final result of the query, and thus the one that if discarded will affect less the accuracy of the query result.

2.1.4 Blocking Operators and Unbounded Data Streams

Blocking query operators are unable to produce any result until the entire input data stream has been evaluated. Sorting operators and aggregate operators such as SUM, COUNT, MIN, MAX, and AVG, are examples of operators that are typically evaluated in a blocking manner (through offline algorithms). As we already said, a data stream is potentially unbounded in its size, therefore the evaluation of a blocking operator in the presence of an unlimited data stream will fail to produce any output. This means that, in the context of data stream processing, blocking operators must be evaluated in a non-blocking manner, which may be achieved through the following techniques [Golab and Özsu, 2003]:

Windows. Windows may be used to delimit an infinite data stream to a finite sub-part of the data stream, providing to the blocking operator a dataset that is bounded in its size, and therefore could be evaluated in a blocking manner. See Section 2.1.6 for the several types of windowing schemes.

Incremental Evaluation. It is possible to evaluate some operators in an incremental manner, by producing intermediate evaluation results as new data stream tuples are evaluated, instead of just produce a final result at the moment the whole data stream has become available (which may never happen). For instance, the aggregate operator AVG may be evaluated incrementally by maintaining only two variables in memory (instead of keeping the entire stream data): a counter of the data stream tuples seen so far, and the sum of all those tuples; making it possible to produce an intermediate result for each new arriving tuple.

Punctuations. Is a technique that tries to take advantage of the data stream semantics through the identification of a specific set of tuples—punctuation tuples. These tuples may be used to mark a point in the data stream beyond which the stream is no longer relevant for the evaluation of a given blocking operator. By doing this, we are delimiting a possible unbounded data stream into the finite sub-parts of this data stream that are strictly necessary to evaluate the given operator in a non-blocking manner. It is easy to see that, the main disadvantage of this approach is that it tightly depends on the data stream semantics and its ability to provide tuples that may be used as punctuations, which in many cases is not possible.

2.1.5 Handling Past Data

The data stream processing model assumes that each arriving data stream tuple could be assessed only once, imposing a serious limitation to all ad-hoc queries mentioning tuples that were already discarded (i.e. that had already passed by the query engine). The most simple solution for this problem, yet tremendously restrictive, is by definition prevent ad-hoc queries from refer old tuples. As we said, it is a very stringent solution, even so for certain applications it may be considered as an acceptable one. A more elaborate solution is to maintain a summary data structure (i.e. a data synopsis) which captures the main features of the previously processed data streams, so it could be used by the queries to provide information on historic parts of the data stream [Babcock et al., 2002]. An ultimate solution is to store parts of the data stream that we believe that could eventually be required in the future, so they can be assessed when necessary.

2.1.6 Language Model

The set of query operators required to process data streams effectively, are as follows [Cugola and Margara, 2012]. Should be noted that these were the operators used to assess the language expressibility of the DSMSs surveyed in Section 3.5.

Single-Item Operators

The operators used to process data stream tuples individually, that is, one at a time.

`Selection` Filters the data stream tuples according to the value of their attributes, just keeping the tuples which match the selection restriction.

`Projection` Extracts from the data stream tuples just the required attributes.

`Extended Projection` Apply a transformation to the attributes of data stream tuples. Typically these transformations are performed by User-Defined Functions (UDFs), such as a function which converts a temperature value from Celsius to Fahrenheit.

`Renaming` Rename the names of the attributes composing a data stream tuple.

Logic Operators

Are used to express detecting rules in order to assess the existence of patterns on the data stream tuples. These rules do not contemplate the order in which the tuples are detected, therefore the pattern rules only rely on the detection (or non detection) of tuples, being ignored the order in which they appear.

`Conjunction` A conjunction of tuples *T1*, *T2*, ..., *Tn* is satisfied when *all* the tuples *T1*, *T2*, ..., *Tn* are detected in the data stream analysed so far.

`Disjunction` A disjunction of tuples *T1*, *T2*, ..., *Tn* is satisfied when *at least one* of the tuples *T1*, *T2*, ..., *Tn* are detected in the data stream analysed so far.

`Repetition` Repetition of tuple *T* is satisfied when it is detected more than *m* times and less than *n* times in *entire* data stream, being *m* and *n* customizable parameters.

`Negation` Negation of tuple *T* is satisfied if *T* was never detected along the *entire* data stream.

Its worth nothing that, `Repetition` and `Negation` are examples of operators which can not be evaluated in a blocking manner, otherwise in the presence of an unbounded data stream they would block forever. For instance, with an blocking evaluation approach the `Negation` operator is only satisfied if after consuming the *entire* data stream, tuple *T* remains undetected. For this reason, these operators have to be unblocked through the usage *windows* operators, in order to delimit the analyzed data stream into a finite set of tuples, or by being evaluated incrementally.

Sequence Operator

The `Sequence` is similar to the logic operators, it is also used to express rules in order to detect patterns of tuples in a data stream, however the tuples arrival order is taken into account. That is, a sequence rule specifying an *ordered* set of tuples *T1*, *T2*, ...,*Tn*, is only satisfied if those tuples appear along the data stream in the specific order *T1*, *T2*, ..., *Tn*.

Iteration Operator

The `Iteration` it's also used to detect an ordered set of tuples in a given data stream, however the detection rule is not defined explicitly (as in the `Sequence` operator), but instead implicitly, through an *iteration condition*, that states the set of restrictions that must be met by the detected sequence (similar to a regular expression). The size of captured sequences is unknown a priori, and may be different from sequences previously identified. For instance, the iteration condition: *"detect all the sequence of tuples that are delimited, upper and lower bound, by prime numbers and that contain between them more than two even numbers"*, may detect both <11,14,5,10,17> and <5,4,6,7> sequences. Note that, for the reasons already highlighted, this operator cannot be evaluated in a blocking manner.

Window Operators

`Window` operators are used to isolate a (finite) portion of the (potentially unbounded) data stream, being this the portion of the data stream that will be evaluated by the query. This approach is of major importance to reduce unbounded data streams to a finite sequence of tuples, so they could be evaluated by

a blocking operator; and also, to "select" just some parts of the entire data stream to be evaluated by a given query. The data stream tuples inside a window are changing continuously, being the dynamic of this change dictated by the semantic of the window operator, which specifies the movement of the window boundaries. Moreover, typically the length of a window could be measured in time (e.g. a window containing all tuples that arrived in last five minutes), or quantity of tuples (e.g. a window containing all the last fifty arriving tuples). The most common semantics for window operators are the following:

`Fixed` It's a static window with fixed lower and upper bounds. For instance, could be used to process data stream tuples received between [*8h*, *12h*].

`Landmark` It's a dynamic window with a fixed lower bound, while the upper bound advances as the new data stream tuples arrive. Could be used to process stream tuples between [*8h*, *now*].

`Sliding` It's a dynamic windows of fixed size where the lower and upper bounds advance as the new data stream tuples arrive. Is the most common type of windows, the emphasis given to the most recent data closely resembles with the domain requirements of the majority of data stream applications, where recent data should prevail over the old one on the query evaluation. For instance, could be used to process the last data stream tuples of the past *4 hours*, [*now–4h*, *now*].

`Tumble` It's a variation of a sliding window, where the changes in the lower and upper bound movement are always greater than the windows size, ensuring that whenever the window moves (i.e. "jumps ahead") all the elements inside the window are different.

`User defined` It's a window where the movement of lower and upper bound is explicitly managed by the user.

Flow Management Operators

Are operators used to merge, split, organize and correlate different data streams with each other. Note that, these operators cannot be assessed in a blocking manner.

`Join` Merge two data streams into one data stream, likewise the JOIN operation in databases.

`Union` Merge two or more input data streams in order to output one data stream including all the data tuples from the two or more input data streams.

`Except` Merge two data streams with the same data structure in order to produce one data stream with all the tuples belonging to the first stream, but do not belong to the second stream.

`Intersect` Merge two or more input data streams to produce one that include only the tuples that are common to all input streams.

`Duplicate` Produce two output data streams which are equal, in structure and content, to the input data stream.

`Remove-Duplicate` Produce an output stream without all the duplicate tuples of the input data stream.

`Group-By` Is used to aggregate data stream tuples according to a given attribute, the resulting clusters are used to be evaluated by aggregate operators (e.g. AVG).

`Order-By` Is used to produce the output data stream with the tuples of the input data stream sorted according to some criteria.

User-Defined Aggregate Functions

Aggregate operators are used to process a group of tuples grouped by some attribute. Typically, they are evaluated in a blocking manner and its usage must be combined with a windows operator. Apart from the common built-in aggregates, such as: AVG, MIN, MAX, and SUM, some systems allow the user to implement new ones, through user defined functions.

2.2 Energy Management Systems

An Energy Management System (EMS) is a monitoring tool that tracks buildings energy consumption with the purpose of enhancing energy efficiency, by identifying savings opportunities and misuse situations. Energy metering data streams are collected from the building sensor network, that is composed by energy meters, equipment and environmental sensors, and others. Such sensor data is then integrated and analysed in order to produce useful information capable of support the building energy management plan. This information should be updated continuously, at least on an hourly basis, and presented to building managers in an analytical and graphical manner, by means of dashboards and reports. The main goal of an EMS is to achieve high-levels of energy efficiency, by reducing buildings energy consumption without compromising the equipments performance, neither the occupant comfort [Motegi et al., 2004; Granderson et al., 2009].

Energy building consumption can be analysed from various perspectives, different concerns will produce distinct reports based on the same gathered data. Thereby, an EMS commonly provides the following comprehensive set of data evaluations features [Granderson et al., 2011]:

Performance indicators. Identify energy consumption patterns through the computation of profiling metrics based on past consumptions and behaviours. These metrics reflect the energy demand by seasons, operations, occupant activities, and so on. They are used to perform benchmarking and forecasting analysis, and also to identify the drivers of building energy consumption.

Normalization. Removes from an a energy consumption measurement the influence of an external variable (e.g. impact of the outside temperature), allowing consumption measurements to be compared with each other in a fair manner. Normalization, by improving the quality of input data, will improve the effectiveness of benchmarking and forecasting analysis.

Benchmarking. Compare the building energy consumption performance against another building with equivalent features (cross-sectional benchmarking), or against the building own historic performance (longitudinal benchmark), or standards (e.g. Energy Star [1]).

[1] `http://www.energystar.gov/`

Forecasting. Predicts near-future consumption and cost profiles. Through the identification of the main energy consumption drivers it is possible to build a predictive model of the building energy demand, allowing to plan preventive actions to properly handle the periods of high energy demands.

Fault detection and diagnostic. Identify faulty equipments that are not working properly or are consuming an unexpected amount of energy.

Statistical analysis. Perform regression analysis over the collected sensor time-series, to compute statistical indicators, such as: average, standard deviation, variance, and percentiles, etc. Required to summarize the energy consumption patterns of the building.

Load Profile. Illustrates the building energy consumption variation over a given period of time.

Financial Analysis. Estimate energy consumptions costs through the usage of tariff rates, and predicts the impact of applying a given energy saving policy.

The information produced by these techniques must be presented to the energy building managers in an understandable and effective manner, otherwise such information will not be useful to support the mangers decision making process. Effective data presentation methods are required so managers can succeed on drawing conclusions on building energy performance. Therefore, depending on managers needs, together with the type of information that has to be shown, EMS solutions provide different presentation methods that can be broadly categorized as follows [Cardoso, 2013]:

Historical Data Analysis. Is used to provide an *historical summary* of all gathered data, through aggregation queries which organize data according space and time dimensions. Much of this information lies on the computation of cumulative amounts of consumed energy and associated costs. Being the purpose of such information to highlight patterns over extended periods of time and space, instead of identify abnormal situations that happen in a volatile and isolated manner. This information is typically depicted with charts and other dashboards techniques that summarize all information in key performance indicators (KPIs), making the information to reflect historical trends in a manner that is easy and fast to understand.

Real-time Monitoring. Is used to provide *up to date information* about the building current energy demands, such as: by the minute information on energy being consumed, associated costs, equipments and places being used, and other timeliness indicators on energy consumption. Information provided in real-time is crucial to properly identify ephemeral and harmful situations (e.g. an abnormal peak of energy consumption in the first two minutes of every hour), so we could act as soon as possible to solve the problem and minimize their caused costs. As you can see in Figure 2.1, the faster an EMS can detect and alert for the occurrence of an harmful situation, the faster we can act upon the problem to correct it, and thus reduce their costs—that is, we must be capable of detect harmful situations in real-time, in order to minimize their detection times towards minimize their costs. To achieve this, gathered sensor data must be processed into useful information in real-time. However, the latency on data processing is a serious barrier to the ability of present up

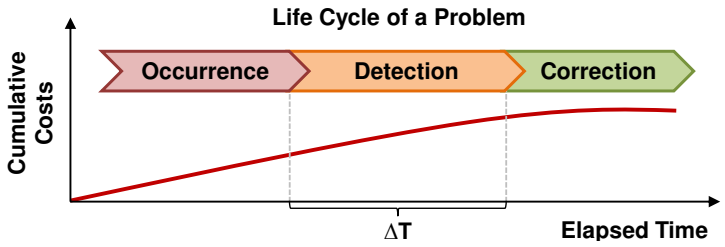

Figure 2.1: Life cycle of an harmful event and its cost. The cost caused by an harmful event is proportional to the time it takes to its detection and correction. Smaller detection times (ΔT) are essential to reduce the costs caused by an harmful event.

to date information in a timely manner. In fact, as we will see bellow, current EMS architectures are not the most adequate approach to cope with the requirement of continuously evaluate large amounts of sensor data streams in real-time.

Hybrid View. Is used to merge the best features of both last approaches: Historical Data Analysis and Real-Time Monitoring. Examples of this method applications are: a real-time comparison of the current energy consumption with the one predicted by analysing historical data, or visualize the impact of the current consumptions on the forecasting model by seeing, in real-time, how current demands influence the expected ones (that also, are constantly changing). To conclude, dashboards with real-time and historical data presentation capabilities, are quite suitable to monitor energy consumption in buildings. They are capable of present current status information mixed up with historical energy consumption trends, which is of major importance to take informed decisions on the building energy management plan.

2.2.1 Generic Architecture and Real-Time Deadlines

To understand how an EMS could monitor energy metering data in real-time, we review the literature [Kazmi et al., 2014, Section 3; Ma et al., 2010; Granderson et al., 2009, Section 2.2.2] to identify the EMS generic architecture depicted in Figure 2.2, essential to realize which architectural components must be addressed in order to develop a real-time EMS. According to the scope of this work, three board architectural tiers were identified, representing the three dimensions composing the latency between the gathering and evaluation of data, and the presentation of the derived information. The tree tiers influencing an EMS latency on stream data processing, are as follows:

Data Presentation Tier. Is responsible to present to the users, generally by means of dashboards, the monitoring information that is managed and computed by the EMS, being the refresh rate of dashboards an important issue for the matter of timely data presentation. Some dashboards can react instantaneously to any change that occurs in the data that they are presenting. That is, displayed information is dynamically updated as soon as new information becomes available, being this the best approach to present time-sensitive information. In contrast, some dashboards can only be updated periodically, by polling their data sources for data updates periodically, being this a not suitable approach to present information which varies widely over time [Eckerson, 2010].

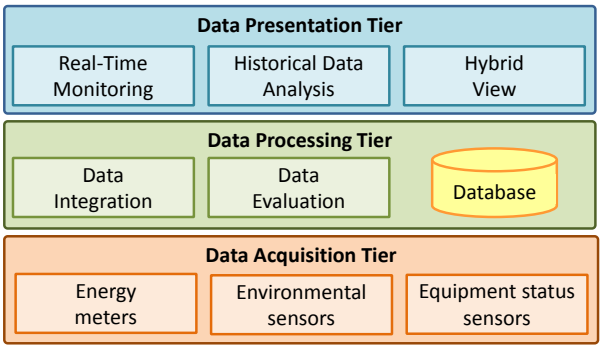

Figure 2.2: Generic architecture of an EMS. An EMS is generically composed by three broad components: (i) Data Presentation Tier, presents the information derived from the evaluation of collected data; (ii) Data Processing Tier, integrates and evaluates sensor data streams and static (persisted) data, this component is commonly supported by a traditional DBMS; (iii) Data Acquisition Tier, collects sensor data streams from the building sensor network.

Data Processing Tier. Comprises the integration and evaluation of acquired data, which could be of two different types: static or dynamic. Static data rarely undergoes changes and thus does not have to be processed in a regular basis, this data is about metadata on building properties, such as: area of each room, equipments per room, energy tariffs, etc. Typically, such data is made available through files and databases. Dynamic data is sensor data streams which are constantly being produced (i.e. updated) by their sources. Such time-variant data has to be processed by the EMS in a continuous manner, in order to keep the information updated. This requirement greatly increase the overhead of the data processing tier, making it challenging to generate results in real-time. Moreover, as we noted in literature, this tier computation is commonly supported by a DBMS, which is known to be an inadequate solution to timely process sensor data streams [Golab and Özsu, 2003; Babcock et al., 2002]. Being this an architectural aspect that severely hampers the ability of current EMSs to process energy metering data streams in real-time [Anjos et al., 2014].

Data Acquisition Tier. Is in charge of collecting the sensor data produced by the building sensor network, which generally can be originated by three different types of sources: energy meters, environmental and equipment sensors. Depending on the device, data may be gathered from sensors according to the following methods: pull-based or push-based (event driven). In the pull-based method, the EMS pulls data from the sensors by querying them periodically, being the pooling time an adjustable parameter. Since the EMS have to *explicitly* query all the devices by the availability of new data to be gathered, the time it takes to check all the sensors of the network may be to large for the system to be able of respond in real-time. In the push-based method, the sensors are the ones with the obligation of send (push) their new measurements to the EMS data processing tier (following an event driven approach). By avoiding both the pooling time and the time it takes to scan all the sensor network, the push method outperforms the pull method on collecting data in a timely manner, since an event driven type of data source is the most suitable one for data driven applications, such as an EMS.

To conceive an EMS capable of monitoring a network of energy meters in real-time, the time it takes to process data through the three tiers of its architecture must be taken into account. The least demanding

Real-Time Levels	Architectural Components of an EMS		
	Data Acquisition	Data Processing	Data Presentation
Real-Time	< 5 min. (Event Driven)	**< 5 min.† (Stream)**	Dynamically (Reactive)
Soft Real-Time	≤ 15 min. (Pull)	≤ 15 min. (Batch)	≤ 15 min. (Periodically)
No Real-Time	> 15 min. (Pull)	> 15 min. (Batch)	> 15 min (Periodically)

Table 2.1: Real-time deadlines of an EMS. The deadlines (in minutes) that each architectural component must meet to handle the arriving data, according to the real-time agreement level that an EMS aims to achieve. (†) Captures the scope of this work: conceive an EMS Data Processing Architecture capable of evaluate data streams with a latency below 5 minutes—that is, in real-time.

deadline (in minutes) that each tier must be able of meet, according to different levels of agreement for the real-time requirement, is illustrated in Table 2.1 [Motegi et al., 2004; Granderson et al., 2009, 2011]. The focus of this work is on Data Processing Tier, which, to respond in real-time, must be capable of processing energy metering data streams within a (least demanding) **deadline of 5 minutes**. The subjects of Real-Time Data Acquisition and Data Presentation are beyond the scope of this research.

2.2.2 State of the Art: Limitations

In this section we survey a representative sample of existing EMS solutions on the market. The purpose is to realize that the architectural design stated above breaks the ability of EMS solutions to monitor energy metering data in real-time. Therefore, each solution was assessed according to its ability to produce results in a timely manner for the functionalities identified in the beginning of this section, being the performance correlated with the query engine that supports their data processing infrastructure.

Many of existing solutions are proprietary, meaning that, since it is not possible to acquire a paid license to evaluate each of them, this kind of surveys are often limited to the documentation made available by the product owner, which in most cases is just a booklet with few technical details. For instance [2], rarely this kind of sources go further than just classifying the systems as a real-time one, not clarifying the time scale of their timeliness. Given that, from the surveyed solutions, we just consider the ones that, at least, give the minimal insight about the time they take to update data.

The results of such survey are depicted in Table 2.2. We assess the features of four EMS solutions: EEMSuite [McKinstry], EnergyWitness [Interval Data Systems], EnerwiseEM [Enerwise], and OpenEIS [Lawrence Berkeley]. Some features require more time to process data than others, being impossible to update data with the same regularity in all of them, whoever the ability of each solution to, in a general manner, produce results in real-time is summarized in the table row: “Real-Time Monitoring”. According to the deadlines of Table 2.1, the “EnergyWitness” solution is capable of computing some of their features in a timely enough manner (within 15 minutes) to be considered soft real-time, while their other features are not computed in real-time. Therefore, we classified this solution as capable of produce results in Soft Real-Time. The remaining three EMS solutions update their results hourly, and,

[2] `http://www.powerlogic.com/literature/3000BR0608R0409_EPO.pdf`

Features	Systems			
	EEMSuite	EnergyWitness	EnerwiseEM	OpenEIS
Data Presentation				
Real-Time Monitoring	○	●†	○	○
Historical Data	●	●	●	●
Data Processing				
Integrate data from different sources	●	●	○	—
Performance Indicators	●	●	●	●
Normalization	—	—	○	○
Benchmarking	●	—	●	—
Forecasting	●	●	○	○
Fault Detection and Diagnostic	●	—	●	●
Statistical Analysis	—	—	—	—
Load Profile	●	●	●	●
Financial Analysis	●	●	●	●
Data Acquisition				
Energy Meters	●	●	●	●
Equipment Status	○	●	—	○
Environmental Sensors	●	●	—	—

Table 2.2: Features and real-time capabilities of surveyed EMSs. Four EMS solutions were surveyed according to their features of data collection and processing. The ability of each solution to compute and display information in real-time is summarized by "Data Presentation" feature. ●: supported. ○: not supported. —: unknown information. (†) This solution responds in Soft Real-Time.

given that, they were classified as No Real-Time solutions.

To conclude, the data processing tier of the four surveyed solutions is supported by a DBMS, in accordance with the standard EMS architecture identified above (see Figure 2.2), confirming what we had already highlighted: that the state of the art approach to support the data processing tier of an EMS lies on the usage of a DBMS, which seriously limits the ability of the EMS to process energy metering data streams in real-time. Being this the research problem of this work—that is, the gap that we found in the state of the art, and for which we propose a solution.

Chapter 3

Related Work

There is a large number of sensor data driven applications that have to process large volumes of data streams in a timely manner, which is pushing to the limit the capabilities of their data processing systems. EMSs are amongst these applications, they are continuously gathering sensor data streams from the building sensor network, which have to be processed with low latency requirements, in order to be possible to monitor building energy consumption in real-time. In this chapter we assess the main systems and technique on stream data processing, in order to identify the systems that are available and and how they could be used to support the solution proposed by this work.

3.1 Stream Data Processing Approaches

Systems aiming at processing data streams in an efficient and effective manner must be capable of coping with the following requirements [Stonebraker et al., 2005]:

Real-Time Response. The system must process high-volumes of data in real-time is only possible with a system designed and fine-tuned for this specific purpose, such systems should be able to timely process data on demand.

High-level Language. The system must provide a query language capable of express data stream queries in an efficient manner, by being capable of easily pose complex data transformation on data stream tuples.

Scalability. The system must be capable of handle a scenario of increased workload maintaining its performance, or then, be prepared to be easily rearranged in order to be capable of cope with such growing amount of work without severely decline its performance.

Tolerance to Faulty Streams. The system must be prepared to deal with delayed, out-of-order, inconsistent, malformed, or data loss data streams.

Deterministic Response. The system must be prepared to always produce the same result for the same input. Among other aspects, this is required to support fault tolerance and recovery capabilities.

Data Integration. The system must be prepared to integrate static (persisted) data with dynamic data streams, by means of an uniform query language that avoids the need of manual intervention.

Availability. The system must be resilient to failures and capable of preserve the integrity and consistency of its data.

3.1.1 Database Management Systems

The inability of Database Management Systems (DBMSs) to process data streams in real-time is widely recognized [Stonebraker et al., 2005; Cugola and Margara, 2012]. DBMSs only processes data after they have been stored and indexed, which introduces an unacceptable penalty on data processing latency. To mitigate this issue, the following DBMS based alternatives have emerged, albeit in vain:

Main Memory Database Systems. This solution stores its data in main memory, and not in second storage, allowing to achieve better results on data evaluation latency than the ones of traditional DBMSs. Even so, this solution is still supported by the same paradigm of "processing data just after store", that by design is not suitable to cope with the requirement of real-time data evaluation of data stream applications. Moreover, traditional DBMSs are completely passive on how they interact with the application, they only respond with data if they are explicitly requested to do so (following a client-server approach), resulting in a interaction style known as: "Human-Active, Database-Passive" (HADP). The HADP model does not allow to spontaneously trigger routines within the database whenever any predefined condition is fulfilled [Garcia-Molina and Salem, 1992].

Active Database Systems. Is a type of database which attempts to solve the preceding issue by providing a mechanism of triggers that capable of spontaneously respond to an event that occurs within the scope of the database [Paton and Díaz, 1999]. However, those triggers are poorly scalable, and a considerable amount of them leads to a large impact on system performance.

Besides the main problem of DBMSs, of have not being designed for process data streams in real-time, they present another issue on this subject: the SQL query language that they provide is not suitable to properly query data streams. The query presents a set of inappropriate features for data stream processing: queries are not evaluated in a continuous manner, the language lacks time-related operators (e.g. sliding window operators), aggregate operators are evaluated in blocking manner (a problematic approach in the presence of unbounded data streams), and the query output is not produced incrementally, instead it is produced a single result-set each time query is evaluated [Arasu et al., 2002; Wang and Zaniolo, 2003]. These are just a few examples on the SQL unsuitability to query data streams, motivating for the need of a query language specifically designed to support data stream applications.

3.1.2 Stream Processing Engines

To effectively cope with the requirements stated above on stream data processing, a new type of system was developed: the Stream Processing Engines (SPEs) [Stonebraker et al., 2005; Babcock et al., 2002;

Abadi et al., 2003]. Although all the implementations of these systems share the same goal—to achieve stream data processing in real-time—different implementations were developed, and some with some sharp differences between them, such as: different architectures, underlying data model, processing mechanisms and query languages. The different implementations that exist between SPEs came from the effort of different scientific communities trying to, through their specific technical background and own view of the problem, develop their own solutions and contributions to this domain [Bass, 2007]. From the proliferation of these systems, two major types of SPEs emerged [Cugola and Margara, 2012]:

Data Stream Management Systems (DSMSs). A type of SPE with the main purpose of process the largest amount of data streams, produced by wide range of different data source source, in real-time. The main purpose of such systems is to process raw data streams that arrive directly from the sources, in order to extract useful information about the application domain [Gulisano, 2012, p.9]. DSMSs can be seen as a natural evolution of DBMSs to properly support data stream processing, in fact, at the beginning, many of these systems were developed from already existing DBMSs, for instance: TelegraphCQ [Chandrasekaran et al., 2003], NiagaraCQ [Chen et al., 2000], Cougar [Yao and Gehrke, 2002], and Nile [Aref et al., 2004].

Complex Event Processing Systems (CEPSs). A type of SPE that interprets input data streams as a streams of events (i.e. facts that happened), with the purpose of draw more complex facts from them. They assess a sequence of events in order identify complex patterns and produce more complex events (with an higher semantic level), exposing more intricate circumstances on what is happening on the domain being monitored. The main goal of those systems is to derive more meaningful situations from less complex ones, giving follow up to the type of analysis and information that is computed and produced by DSMSs [Babcock et al., 2002].

Besides the different semantic level on data evaluation of these two types of SPEs, there is another distinguishable aspect between them: the query language model. DSMSs usually rely on a declarative (e.g. CQL [Arasu et al., 2004]) or imperative language model (e.g. SQuAL [Cetintemel et al., 2006]), while CEPSs usually rely on pattern-based language (e.g. CEL [Demers et al., 2007]). Declarative languages (such as SQL) logically express the type of results that are expected for a given query, instead of describe the computation flow that would lead to these results, such as in imperative languages. Pattern-based languages are known to be defined as set of ECA rules (Event-Condition-Action rules), where each rule is composed by a *condition* (e.g. a regular expression) that if satisfied by a sequence of *events*, detected in the arriving data stream, will trigger a specific *action*.

It is important to point out the differences between DSMSs and CEPSs due the following reasons: (i) each system is specifically designed to effectively process or data streams or event streams, (ii) explain the misunderstandings [1] on the SPEs' concepts, that hinders the required cooperation between researchers in order to advance this field's state of the art, (iii) clarify, that ideally a SPE should be able to process both data and event streams, and (iv) show that the main difference between SPEs of first

[1] `http://epthinking.blogspot.pt/2007/09/event-processing-and-babylon-tower.html`

and second generation is the ability of the last ones to process both data and event streams, proofing the current maturity of such systems.

3.2 Stream Processing Engines: First Generation

In this section, we present the relevant work that was developed on streaming data computation. We highlight the pioneering SPEs, those that, at the beginning, most contributed to development of the field of data stream processing.

3.2.1 Data Stream Processing Engines

Many systems were developed to efficiently process data streams, the most relevant are presented below [Gulisano, 2012; Cugola and Margara, 2012; Bui, 2009; Fulop et al., 2010]: STREAM, Borealis, TelegraphCQ, and COUGAR. Among other things, these systems differ in the deployment model, which can be: *centralized*, if the data streams are computed in a single machine, such as STREAM; or *distributed*, is the data stream computation is spread by a distributed environment (i.e. a network of well connected machines), such as Borealis.

STREAM. [Arasu et al., 2004, 2005] Is the system that introduces CQL (Continuos Query Language), the first great effort to conceive a declarative query language that provides a precise, clear, and general purpose semantics to query continuous data streams. CQL presents a SQL-like syntax, being the core of the language composed by three types of operators: relation-to-relation (RtR), stream-to-relation (StR), and relation-to-stream (RtS). CQL is built on top of a SQL query engine, being SQL-92 used to implement the RtR operators, and SQL-99 used to implement the window features of StR and RtS operators. The windows introduced by CQL are used to handle input data streams (e.g. sliding window), which converts the data stream into relational tables capable of being evaluated by RtR operators. The StR operator is the one that implements this windowing mechanism. Finally, those processed values are converted to an output data stream through RtS operators. STREAM's scheduling protocol, decides which operator should be executed, and how many data tuples should be processed, according to criteria related to main memory consumption. Timestamps are implicit, they are managed by the system internal clock, being impossible to reference it from the CQL language. Massive workload scenarios load are handled through shedding techniques [Gulisano, 2012, Section 8.2.2][Cugola and Margara, 2012, Section 4.2].

NiagaraCQ. [Chen et al., 2000] Aims at applying continuous queries (CQs) to a set of XML documents (possibly distributed across internet), in order to provide an high-level abstraction able to detect changes on a set of XML documents (e.g. Wikipedia articles). CQs are defined trough an SQL-like declarative query language known as XML-QL[2], being the queries evaluated according two different approaches: *timer-based* or *change-based*. In the former, queries are evaluated periodically; in the later, queries' evaluation is triggered by a document change notification thrown by the

[2] `http://www.w3.org/TR/NOTE-xml-ql/`

source. These queries do not produce an output data stream, instead they produce a set of actions: table updates. The system main contribution is their novel architectural approach to achieve high scalability: it groups "similar" queries, i.e. query plans sharing the same logic, in order to reduce redundant computation overheads. Through shared query plans of grouped queries, is possible to greatly reduce the amounts of memory that is required for query evaluation, allowing to greatly reduce I/O overheads when compared with individual execution of each query plan. NiagaraCQ also introduces a dynamic regrouping protocol that allows the user to add/remove queries in runtime, without the need of regroup already installed queries. At a glance, NiagaraCQ can be seen as a sort of Active Database (AD), in the sense they also run queries and triggers. However, ADs do not have the ability to evaluate queries periodically, neither to support several change-based queries by document (ADs' triggers scale poorly). Moreover, by contrast with NiagaraCQ, ADs are not designed to monitor autonomous and heterogeneous data sources distributed across a wide geographical area [Gulisano, 2012, Section 8.2.4][Cugola and Margara, 2012, Section 4.2]

Borealis. [Cetintemel et al., 2006] This system was built on top of other two previous SPEs: Aurora* [Abadi et al., 2003] and Medusa [Cherniack et al., 2003]. The system main contributions are: the ability to add corrections to previous output results as new stream tuples are being processed, *dynamic query revision*; and also the ability of change, at runtime, the operators composing a query evaluation plan, *dynamic query modification*. SQuAL is the Borealis imperative query language used to specify how data streams must be evaluated, it is a graphical language where the user explicitly specifies the evaluation query plan, through a scheme of boxes and arrows. Boxes (or computation nodes) are used to represent the data transformation steps of a query evaluation plan, while the arrows, by connecting the boxes, describe the sequence (pipeline) of these transformations. Each node can have more than one input/output stream (or arrow), and has associated a QoS metric used by the system to adjust both the *scheduler* and *load shedding* algorithms. QoS indicators allows the scheduler to know which nodes are deviating from their QoS agreement, and therefore should be executed with priority in order to not compromise the overall system's performance. These QoS indicators are also used to by the load shedding algorithm to identify the boxes from which should be removed some data tuples. For better scalability Borealis is designed to efficiently distribute its load across the available computational resources—that is, to work in a distributed environment. Borealis main applications go through: real-time monitoring applications, environmental monitoring, surveillance, tracking, plant maintenance, and telecommunications data management [Fulop et al., 2010, Section 2.4.2.9].

TelegraphCQ. [Chandrasekaran et al., 2003] This system was built as an extension of a DBMS, PostgresSQL. TelegraphCQ takes advantage of the DBMS capabilities to store and manipulate data, making the required changes to properly process data streams [Cugola and Margara, 2012, p.35]. One of the main features of this system is the architecture, which is composed by a set of individual modules that use the Fjord [3] API [Madden and Franklin, 2002] to interconnect them in

[3] "Framework in Java for Operators on Remote Data Streams".

a non-blocking manner to form a query plan, where data can be processed in a push-based or pull-based model. These modules are generic components of different types (caching, adaptive routing, query processing, etc.) that consume and produce tuples, allowing the query plan to be distributed across several nodes of a distributed environment. This architecture seems to be very similar with the Borealis' boxes and arrows approach. However, in Borealis, these boxes are not generic and can not be changed by the user in order to add extra functionality, as they can be in TelegraphCQ. A SQL-like declarative query language is also introduced by TelegraphCQ: the StreaQuel, which provides a rich set of data window operators [Gulisano, 2012, Section 8.2.3][Cugola and Margara, 2012, Section 4.2].

COUGAR. [Yao and Gehrke, 2002; Bonnet et al., 2001] This system aims at evaluating *sensor queries* on *sensor databases*. Sensor queries can be processed in two different manners: *warehousing* and *distributed* approach. Warehousing approach collects data from sensor network into a centralized database, for posterior evaluation. This approach scales poorly due the population of the databse with huge amounts of raw data that is often not used in any query evaluation, moreover it is quite inefficient for sensors that run on batteries, the transmission of all collected values consumes a lot of energy, greatly reducing the network lifespan. The distributed approach take advantage of sensors computational capabilities, and thus the query evaluation computation is moved from the centralized database to the network (i.e. in-network query processing) drastically reducing sensors energy consumption. Therefore, a sensor network can be seen as a huge distributed database where each sensor holds and assesses part of the data, hence the term: sensor database. COUGAR focuses on the *distributed query evaluation* approach, being each type of sensor modelled as an Abstract Data Types (ADT), that provides an API to control its computation capabilities and encapsulated data. Local computation is cheaper than communication between sensors, thus, to better exploit sensors energy resources, the query evaluation plan is pushed to the sensor network, by specifying the exact computation that each node must perform, together with the data that must be exchanged between nodes. COUGAR main contribution is the minimization of data exchanged between network nodes along a query evaluation process, only being transferred the strictly necessary information, performing data evaluation as close as possible of the data sources. Such contribution is relevant to better understand how to achieve high throughputs in distributed SPEs [Gulisano, 2012, Section 8.2.5].

3.2.2 Event Stream Processing Engines

Event stream processing engines have their origin in the *content-based publish/subscribe* middleware, where subscribers express their interest in a given event (or pattern of events), by subscribing the content, in order to be notified whenever a match is found in the event streams produced by the publishers. The ability to identity event patterns is the main issue of these middleware [Cugola and Margara, 2012, Section 4.3][Eugster et al., 2003, Section 4.2]. It is worth noting that, the systems presented above for data stream processing were not designed to express queries to detect causal relationships over

sequence of events (i.e. to detect patterns of events), and even less identify patterns with intricate temporal constraints. Hence the emergence of systems capable of efficiently processing event streams—the Event Stream Processing Engines [Barga et al., 2007]. In this section, we present the relevant work that was developed on event streaming computation. We highlight the pioneer SPEs, those that, at first, most contributed to develop the field of event stream processing.

CEDR. [Barga et al., 2007] Is a general purpose event stream processing engine which introduced significant contributions to this field, such as: a novel temporal approach supporting a consistency model which adapts to the application requirements, and also a powerful declarative query language. The introduced temporal approach associates three dimensions to the event stream tuples (Valid Time, Occurrence Time, and System Time), allowing to create consistency models that are capable of ensure the correctness of produced results. For instance, the consistency model could be adapted to allow the system to tolerate faulty sensor networks (see [Barga et al., 2007, Section 4] for details). In CEDR, event streams are modelled as a flow of state update notifications, which can be compared with NiagaraCQ on monitoring XML content, however CEDR, more than detect changes on documents, also identifies causality patterns and more complex temporal relations in a event stream. Temporal model also allows the formal specification of each language operator, strongly contributing to the rich set of operators on temporal relations and pattern identification provided by the query language [Cugola and Margara, 2012, Section 4.3].

Cayuga. [Demers et al., 2007] Is a general purpose event monitoring system aiming at process large sets of event streams in a centralized manner. Besides its ability to scale in a centralized environment, another relevant contribution of this system is its SQL-like query language: CEL, a very expressive language used to write detection rules to assess complex patterns on event streams. Each rule can be seen as a set of restrictions that are converted by Cayuga into a set of non-deterministic automata that will be evaluated simultaneously. That is, each arriving event will be used as input in all automata, and if a sequence of events triggers all automata rules to their final state, then this means that a pattern was detected. Due the strictly connection that exists between automata, Cayuga does not support any kind of distributed processing. Therefore, to scale in a centralized environment, all auxiliary data structures reuse the shared automata instances in order to assess the available resources in an efficient manner [Cugola and Margara, 2012, Section 4.3].

SASE/SASE+. [Gyllstrom et al., 2007; Wu et al., 2006; Diao et al., 2007] Is an event monitoring system with the specific purpose of process streams of RFID events [4]. The system main contribution is the scalability of its performance—SASE throughput is up to 40k events/second, outperforming similar systems. SASE language allows the specification of detecting rules to identify RFID event patterns, where each rule specifies: the events that should be detected and how they relate each other, and the restrictions on their attributes. The language also provides an expiration date parameter to allow the definition of time windows. Rules are always evaluated through a fixed query plan of six data transformation blocks. The SASE drawbacks are related to the lack of language expressibility,

[4] Events that result from reading "Radio-Frequency Identification" (RFID) tags.

namely: the absence of aggregations (e.g. COUNT and AVG operators), only primitive events could be used to produce complex events (i.e. complex events could not be used to produce events even more complex), and the bounded size of operators' buffers, which limits the number of events that can be correlated. Those limitations prevents the system from being capable of detect some patterns. Subsequently, SASE language was augmented to SASE+, a general purpose language for event stream processing that goes beyond the RFID events, allowing the detection of patterns that cannot be detected by SASE [Cugola and Margara, 2012, Section 4.3].

3.3 Stream Processing Engines: Second Generation

The systems presented below belong to the second generation of stream processing engines, they differ from the systems previously presented by being designed to efficiently process both data and event streams. The following solutions are the most relevant ones on streaming data computation.

Esper. [5] [EsperTech, 2014] It is a leader open source SPE for general purpose data stream processing. Esper introduces a powerful declarative query language, known as "Event Processing Language" (EPL), with an SQL-like syntax. The language provides all common SQL operators (implemented in a non-blocking manner), together with a rich set of stream related operators, such as time windows and pattern detection rules. The continuous queries provided by Esper output their evaluation results as data streams, that can be reused as input data for another continuous queries, letting us to implement a *pipeline of queries* (chain of data transformations), to process the data streams that arrive from the sources. This pipeline of queries allows us to implement data processing layers in a structured manner, i.e. to implement complex blocks of data stream processing through an elaborate topology composed by simple continuous queries. Esper is provided as a library for the Java programming language, making it straightforward to be integrated with other Java applications. There is also an .Net version of the engine, the Nesper, allowing the engine to be integrated into applications that are being developed in C# programming language. Although a centralised SPE, Esper throughput scales up to 5×10^5 tuples/minute, with an maximum average of tuple evaluation latency below 3 ms [EsperTech, 2014, Section 20.1]. For more demanding scenarios, there is EsperHA, a distributed version of Esper, designed to be deployed in a distributed environment, in order to improve the performance of the original solution. [Cugola and Margara, 2012, Section 4.4].

Apache Storm. [6] [Apache, 2014] It is a distributed, fault tolerant, and extremely scalable system that focuses on the reliable processing of data streams. The Storm architecture is quite simple, it is composed by three main concepts: (i) the data stream sources, known as "spouts", (ii) the data transformation components, known as "bolts", (iii) and the graph structure which determines how "spouts" and "bolts" are interconnected, that is known as the "topology". The data stream is processed *continuously*, as it traverses the "topology" and passes through their "bolts" (unlike

[5] `http://www.espertech.com/products/esper.php`
[6] `https://storm.apache.org/`

equivalent systems, such as Hadoop, which process data in a *batch* manner). The transformation performed by each "bolt" may vary from simple MapReduce functions, to operations of filtering, aggregation, joining, counting, and interaction with external sources, such as service APIs and databases. Storm main contributions are their high performance and scalability, it can process up to 10^6 tuples/minute, moreover its fault tolerance allows to build strongly reliable systems. Data stream applications integrate with Storm through its API, allowing the application to be developed in virtually any programming language (provided the required language adapter). Java programming language is natively supported by Storm, however many other language adapters are available, such as: Scala, JRuby, Perl, and PHP. Strom may be deployed in a distributed or centralized environment. Centralized deployment allows Storm to be used in a single machine, while distributed deployment allow to take the most of a cluster of machines, such as cloud computing environments, as the "Amazon Elastic Compute Cloud" [7] (EC2).

Apache S4. [8] [Neumeyer et al., 2010] It is a distributed, general-purpose, stream processing engine supported by MapReduce and Actor Model techniques. Data processed through MapReduce operations is not a new approach in data analytic, yet this approach is commonly used to process data in a batch manner, which is not the most appropriate manner to process data streams. That said, the relevant contribution of S4 is to present an architecture that, likewise Apache Storm is capable of process data streams through MapReduce operations in a continuous (streaming) manner, avoiding the batch processing approach. Such architecture is based on the Actor Model concurrent computation technique, that supports the S4 main features of being an highly concurrent and scalable systems, working in a distributed and event driven manner. The manner how data transformations are expressed differ from both Esper and Storm, in the sense that S4 does not provide neither a declarative query language or a topology of data transformation nodes. Instead, the user specifies how data streams must be processed through the definition of data processing blocks, known as *Processing Elements* (PEs), for which the user specifies the data transformation that must be performed and the target data stream tuples. S4 will instantiate a PE for all the different data stream tuples. Being each instance in charge of consume and evaluate their target tuples, and push the produced results to the PEs that subscribed those types of data stream tuples. At first, S4 approach of process data streams may resemble the Storm topology, however they are different: Storm requires the data stream to pass through all the nodes (i.e. bolts) of topology, and therefore these bolts must be implemented accordingly; whereas S4 processing elements will strictly receive just the tuples which they are in charge of processing, facilitating the implementation of these elements. Arriving data stream tuples are *pushed* to S4, which requires the existence of input data queues, to hold the excess of data arriving to the processing elements. If a scenario of increased workload makes the queues grow to their maximum size, the system adopts the most restrictive load shedding approach of start dropping all new arriving tuples, which may not be the most suitable solution for some application fields. To achieve better performance, S4 maintains all

[7] `https://aws.amazon.com/ec2/`
[8] `http://incubator.apache.org/s4/`

their data structures in memory (to avoid disk access overheads), these structures are persisted periodically by the Fault Recover Mechanism. Meaning that, tuples that arrive after the checkpoint and before the failure will be lost, therefore the partially fault tolerance of S4.

3.4 Other Systems to Process Large Data Sets

There is a set of well known systems that although capable of efficiently process large sets of data, were not designed for streaming data processing—this work main research topic. Therefore, these systems were not formally assessed in this related work, for which we have only the following observations:

Apache Hadoop. [9] It is a MapReduce based system that processes large data sets in a distributed computing environment. We explore their ability to process massive sets of unbounded data streams, however we quickly realize that Hadoop is conceived for batch processing, diverging from the streaming computing paradigm in which we are interested. Because of this, Hadoop was not considered in this literature review.

Apache Spark. [10] [Zaharia et al., 2010, 2012] It is another system to process large data sets in a distributed manner, also in a batch model, which can be seen as an alternative to Hadoop. For the purposes of this work, Spark main feature is their **Spark Streaming** [11] extension which allows Spark to handle stream data processing, giving to this system the ability to integrate both batch and stream data processing in a single solution. However, Spark Streaming works as an intermediate component used to apply a "discretization" setp to the input data stream (i.e. split the data stream in batches), so it could be (batch) processed by the underlying query processing engine of Spark—in a "micro-batching" manner [12]. Meaning that, the data stream processing capabilities of Apache Spark (Streaming) are underneath supported by a batch processing engine. Since this work is concerned with stream processing engines, the Apache Spark and its Streaming extension were not considered in our survey.

3.5 Discussion

In accordance to the scope of this work, the most relevant features of the stream processing engines assessed above are: the ability of systems to perform real-time data stream processing, and the expressiveness of query language to pose queries on the application domain. All assessed SPEs seem to be capable of process data streams in rea-time, in particular those of second generation: Esper, Storm, and S4. That said, the surveyed feature which is more distinguishable among assessed SPEs is theirs query language model. This was the most critical concern of this survey, since we aim to proof that the query language of a DSMS is more suitable to query energy metering data streams, than the traditional SQL

[9] https://hadoop.apache.org/
[10] https://spark.apache.org/
[11] http://spark.apache.org/streaming/
[12] http://spark.apache.org/docs/latest/streaming-programming-guide.html

Systems	Language Model																											
	Single Tuple					Logic						Windows					Flow Management										Deployment Model	Open Source
	Selection	Projection	Renaming	Extended Projection		Conjunction	Disjunction	Repetition	Negation	Sequence	Iteration	Fixed	Landmark	Sliding	Tumble	User Defined	Join	Union	Except	Intersect	Remove Duplicates	Duplicate	Group By	Order By	User-Defined Aggregates		
DSPEs																												
STREAM	●	●	●	○		○	○	○	○	○	○	○	○	●	●	○	●	●	●	●	●	●	●	●	○		**C**	●
NiagaraCQ	●	●	●	●		○	○	○	○	○	○	○	○	○	○	○	●	●	—	●	○	○	●	○	●		**C**	●
Borealis	●	●	○	●		○	○	○	○	○	○	○	○	●	●	○	●	●	○	○	○	●	●	○	●		**C**	●
TelegraphCQ	●	●	●	—		○	○	○	○	○	○	●	●	●	●	●	●	●	●	●	●	●	●	●	—		**D**	●
COUGAR	●	●	●	●		—	—	—	—	—	—	○	○	●	○	○	●	●	—	●	—	○	●	—	●		**D**	●
ESPEs																												
CEDR	●	●	●	●		●	●	○	●	●	○	○	○	○	○	○	●	●	●	●	○	○	●	○	●		**C**	●
Cayuga	●	●	●	○		●	●	○	○	●	●	○	○	○	○	○	●	○	○	○	○	○	○	○	○		**C**	●
SASE/SASE+	●	○	○	●		●	●	○	●	●	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○		**C**	●
SPEs																												
ESPER	●	●	●	●		●	●	●	●	●	●	○	○	●	●	●	●	●	○	○	○	○	●	●	●		**C**†	●
Twitter Storm	●	●	●	●		●	●	—	—	●	●	●	—	●	—	—	●	●	—	—	—	●	●	●	●		**D**	●
Apache S4	●	●	●	●		●	●	—	—	●	●	—	—	●	—	—	●	—	—	—	—	—	●	—	●		**D**	●

Table 3.1: Summary of the main features provided by the surveyed SPEs. ●: supported. ○: not supported. **C**: Centralized. **D**: Distributed, scales through the workload distribution across a clustered of well connected nodes. —: unknown information. (†) EsperHA, an Esper version supporting distributed computation.

provided by DBMSs. Therefore, the language model expressibility of each surveyed SPE was assessed, being the results illustrated by Table 3.1 (the language model operators are described in Section 2.1.6).

Its worth noting that, the relation between language expressiveness and supported operators is not linear. There is some operator transformations that can be achieved through the combination of other operators [Cugola and Margara, 2012]. Moreover, MapReduce based systems supporting User Defined Functions (UDFs) written in a Touring-Complete language (e.g. Java), such as Storm and S4, are Touring-Complete. Meaning that, regardless of the algorithm complexity, these systems can potentially express any surveyed operator of the language model [Stewart et al., 2011]. However, for the sake of simplicity on queries implementation, we only consider that the language model of a surveyed system provides a given query operator, if the operator is provided by default, or if its semantic can be trivially mimicked through the composition of other available operators.

Finally, a couple of other features were taken into account in Table 3.1. The Deployment model of each SPE is tightly related with its ability to scale its performance in the presence of a massive workload scenario, and given our requirements on low latency data processing, such feature is also highlighted. The possibility of use and modify each SPE freely, without restrictions, by following an “Open source” approach, was another important feature that was taken into account.

3.6 Conclusion

Surveying the related work on the stream processing engines (SPEs) let us to assess: the different types of available systems, their main features and requirements, and how this information must be taken into account in the development of the solution proposed by this work. We found that, at an early stage, according to the requirements of each application domain, SPEs could be splited into two types of

systems: Data Stream Processing Engines (DSPEs), and Event Stream Processing Engines (ESPEs). Later on, the evolution and maturation of this technology lead to the development of SPEs, that, coarsely speaking, are capable of efficiently process both data and event streams. The main features covering these systems, as well their impact on the type of stream analytic that is performed, are as follows:

Language Model. Is a feature that is tightly related to the streaming processing capabilities of the systems. ESPEs rely on pattern-based languages to formulate pattern-detection rules to identify patterns on event streams—that is, to identify relations of causality between events. These languages provide a rich set of pattern oriented techniques such as Sequence detection and Logic operators. Whereas DSPEs specify their data transformation operations through declarative or imperative languages, providing powerful windowing mechanisms and flow management operators.

Time Model. Another distinguishable feature is related to how the system deals with the notion of time. ESPEs establish a partial (causal) or total order between tuples of event stream. This is required to implement the semantics "Happened-before" used to support the identification of relations of causality, such as the ones of Sequence and Iteration operators. In turn, time model of DSPEs is a quite more relaxed, the notion of time is only relevant to specify the time windows boundaries, in order to identify which data stream tuples are inside (or outside) the window. Being, in most cases, the ordering of tuples within the window not relevant.

Load Shedding. Is the ability of the system to do not compromise its response time during periods of overload, by dropping some tuples from the input data queue. Each dropped tuple implies a degradation in the quality of the produced results, yet this approach allows the system to continue to respond in real-time. We realize that "purely" ESPEs are very reluctant to this approach, and typically they do not support any Load Shedding technique. This happens because the queries computed by such engines are commonly related to pattern detection, which produces a type of query responses for which is difficult to be interested in approximated answers: a pattern is or is not detected, an approximated response for this is typically not reasonable. Moreover, the drop of the "'wrong" tuple may invalidate the detection of an whole pattern, and thus produce a completely wrong result. The same does not apply to DSPSs, for the type of queries assessed by these systems is reasonable to accept approximated answers since they are produced in real-time. For instance, in most applications, is reasonable to accept an approximated answer for the "5-minutes Moving Average" over a data stream attribute, provided the result is computed in a timely manner.

Programming Model. For the matter of this work, the most relevant issue is to understand how these systems can be used to implement our solution, and since the solution will be supported by a second-generation stream processing engine, this discussion will be focused on the SPEs: Esper, Storm, and S4. Storm is based on an imperative programming model, meaning that is the user that explicitly specifies the sequence of data transformations that must be applied to the data stream, through the implementation of a *graph of data transformations* (i.e. the Storm "bolts-topology"). Moreover, Storm is not a pure query engine, in the sense that there is no query optimizer to automatically produce an optimal query plan. Regarding S4, its programming model is somewhat

different from Storm, but it also lacks a query optimizer capable of generate an optimal query plan to evaluate a given query. On the other hand, Esper provides a SQL-like declarative query language, which allows to specify the data transformations that are desired, instead of having to specify how data must be processed, such as in the imperative language approaches of Storm and S4. In Esper a data transformation graph does not have to be explicitly defined by the user, instead it could be automatically produced and optimized by the query engine optimizer, providing a transparent decoupling between the logical level (query specification) and the physical level (query execution). This is an advantage of Esper over Storm and S4, for which the difficulty of implement and optimize a graph of data transformations grows quickly with the complexity of data analytics. Esper hides from the user the complexity of formulate query evaluation plans; however, this does not prevent users from implementing a data transformation graph (or topology)—which can be achieved through a pipeline of continuous queries. Rather, it means that the functionality of each query (i.e. each node of topology) could be implemented through a declarative language. Meaning that, there is a sort of quite complex data transformations for which the query plan could be automatically generated from a declarative query specification.

Our proposed solution is based on a graph of continuous queries, which will work as a pipeline of data transformations. Coarsely speaking, in our solution data streams are processed as they cross and make progress through the graph. Each graph node is a data processing step of the pipeline, that will consume the arriving data stream and output the computed results. Due the claims of this work, the Programming Model of the streaming data processing system used to support this work solution is of utmost importance. We demand real-time data processing, therefore it is crucial to formulate optimized query evaluation plans. We also aim to demonstrate that the traditional SQL query evaluation model is not the most suitable to query sensor data streams. Therefore, to benchmark SQL with another declarative SQL-like query language would be useful to in a smoothly and fair manner point out the SQL issues to support queries in this domain. Taking all of this into account, **Esper** was the Data Stream Management System (DSMS) that we choose to support the solution proposed by this work.

Chapter 4

Solution

Our solution aims at creating a data processing architecture to integrate energy related data in real-time. Existing architectures, supported by DBMSs, process data in a batch manner through a *pipeline of data transformation steps*, impeding data to be processed in a timely manner [Vassiliadis and Simitsis, 2008; Bruckner et al., 2002; Karakasidis et al., 2005; Nguyen and Min; Vassiliadis, 2009]. This work proposes a Data Processing Architecture supported by a DSMS—instead of a DBMS—that allows EMSs to process data streams in real-time. The proposed solution adapts the *pipeline* stated before in order to process data continuously—in stream—allowing the freshness of data to be measured in seconds.

The chapter begins by introducing the fundamental concepts of the proposed *Data Processing Architecture* in Section 4.1, being followed by Section 4.2 which describes the *Requirement Analysis* that identifies the data transformation operations that should be implemented by the prototype version of the proposed architecture, essential to validate the feasibility of its implementation as well the claims of this work. To finalize, Section 4.3 covers the features of the *Experimental Environment* that was used to deploy the proposed solution. In particular: the *dataset* and the *energy metering network* that was employed in our experiments, the *data schema* used by the solution to manipulate the energy metering data, and also the *Simulator* that we had to build in order to mimic the behaviour of a real energy metering network.

4.1 Architecture Overview

This section introduces the main components of the solution and how they interact. The solution architecture is depicted by Figure 4.1 which identifies the three main architectural components of an EMS, and how they interact with each other: Data Acquisition Tier, Data Processing Tier, and Data Presentation Tier. The bulk of the complexity lies in the Data Processing Tier, for which we propose the Data Processing Architecture depicted, defined by the following main components: Adapters, Data Integration and Evaluation component, and Data Queues. We believe this is the best solution to implement an EMS capable of process data in data real-time, for the following reasons:

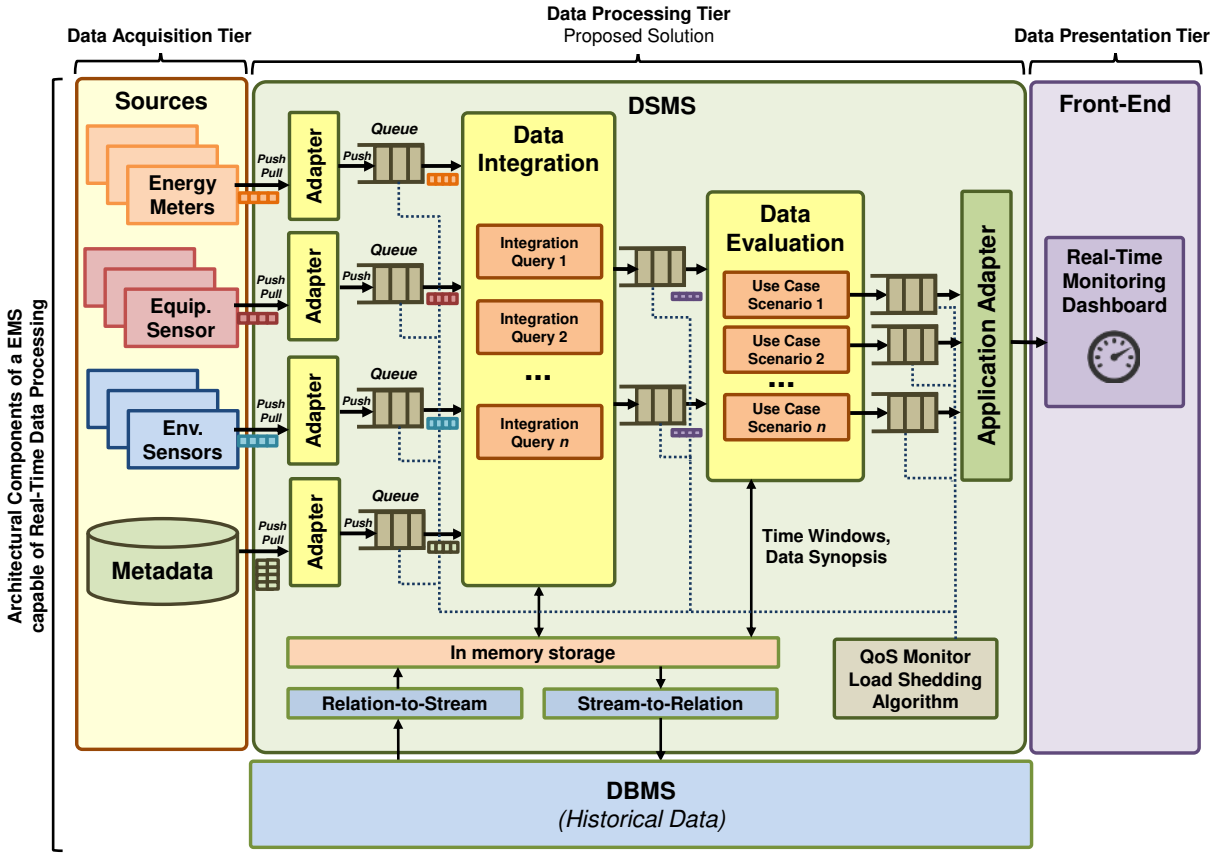

Figure 4.1: Proposed architecture for the Data Processing Tier. From left to right, the data processing components of an EMS: the Data Acquisition Tier composed by the data stream sources, the Proposed Architecture for the Data Processing Tier, that is responsible for the integration and evaluation of gathered data streams, and the Data Presentation Tier where the computed results will be presented, typically in the EMS's real-time monitoring dashboard.

1. The proposed architecture is supported by a DSMS, known for being the most appropriate type of query engine to timely process sensor data streams.

2. The loose coupling of architectural components, allows to deploying them in fully distributed settings (such as cloud environments), by deploying each component in a cluster node, highly improving the systems scalability on huge workloads scenarios.

The proposed Architecture for the Data Processing Tier, as well the Data Acquisition and Presentation Tiers that compose an EMS are detailed below.

4.1.1 Data Processing Tier

Conceptually, the Data Processing Tier is the core component of the solution, it works like a *pipeline of data transformations* supported by queries, where data received by Data Acquisition Tier is continuously manipulated to produce the information required to feed the Data Presentation Tier. The data transformation flow is structured in several stages using the types of components detailed below.

Adapters mediate the extraction of data from several sources into the data transformation process. Adapters understand the sources data delivering model—push or pull based—and push data into

remaining components of the architecture. Adapters may perform a set of data validation steps, such as identify and discard faulty tuples produced by faulty equipment that may hamper the process, and normalize into a common schema distinct data stream schemas that come from different types of sensors. The adapters role is critical to the effectiveness of all data transformation process: they bring to the *pipeline* only the strictly necessary data, pre-processed in the most convenient way, for the remaining data transformation process.

Data Integration is the core functionality of the data transformation process, which consists of Data Integration and Cleaning steps. The main purpose is to combine and analyze several data streams, in order to compute a new set of data flows, adopting schemas that better fit the problem domain, and that will be used as input for domain specific queries. Note that, the integration of several streams are far from being a trivial process, raising several data quality issues. For instance, some data cleaning may be required in order to ensure data consolidation and consistency. These issues must be solved in this components, that must be able to merge data from multiple sources (e.g. sensor networks and databases), transform data under different schemes, recalculate and synthesize attributes, specify default values, calculate new attributes, etc.

Data Evaluation supports the evaluation of application queries including those that represent energy monitoring use-case scenarios. These queries has as input the previous integrated data streams, which represent the available data sources for these application queries. From the evaluation of these queries will result the essential Key Performance Indicators (KPIs) used to support the decision making process. The timely computation of such KPIs depends on how suitable are the data streams produced by the Data Integration component.

Application Adapter converts the output streams into a format that can be understood by the Data Stream Application (e.g. the Real-Time Monitoring Dashboard).

Data Queues holds excess of data when the arrival rate of data stream tuples becomes higher than the processing capability of the receiver component, otherwise there would be loss of data. Queues will be placed at the entrance of the Data Processing Tier and between the most critical components (e.g. Data Integration and Data Evaluation), the ones that due their different data transformation complexity may yield data at different rates. Besides their major purpose, queues may also, if necessary, perform some additional computation, for instance to impose some priority order on the delivery of tuples or even to prevent its infinite growth through the usage of *Load Shedding* techniques, which carefully select tuples that may be discarded without largely affect the accuracy of produced results.

4.1.2 Data Acquisition Tier

Data Acquisition Tier covers all data sources that may interact with Data Processing Tier. Sources may be splitted into two major types according to the nature of its produced data: dynamic sensor data and persisted static data. The former is produced by the building energy metering network, used to

monitor their energy consumption performance, which leads to a *continuous* production of sensor data streams. These data streams may be produced by three different types of sensors: energy meters, environmental, and equipment sensors. The later consists on building metadata that rarely undergoes changes (such as room areas, equipments by room, energy tariffs, etc.) and that is typically available through a database. Although less transient, metadata is highly useful when integrated with volatile data streams, contributing to improve the data stream processing process.

4.1.3 Data Presentation Tier

Data Presentation Tier is the client of Data Processing Tier, consuming the information that is continuously produced through the evaluation of acquired data streams. From all EMS's data presentation applications, the real-time monitoring dashboard is the one that will benefit the most from the timely computation of produced results, thus its reference in the solution architecture. However, provided the proper adapter, any data stream client application could consume data from the Data Processing Tier.

4.2 Requirements Analysis

To identify which data transformations must be preformed by the Data Processing Tier, we develop a *case study* to guide our design decisions on the development of the Data Integration and Evaluation architectural components. The case study comprises a set of use-case scenarios—data transformation queries—, that are related with building energy management domain and require near real-time evaluation. Thus, the case study aims at clarifying the following **requirements** regarding these queries:

1. **Class of Queries.** What is the class of queries used to monitor energy metering data in real-time.
2. **Main Operators.** Which are the main operators and data transformations of these queries.
3. **Data Sources.** What are the distinctive features of the data that has to be processed.
4. **Computed Results.** What kind of information should be computed by the queries.

From the literature surveyed during the development of the case study (summarized in Table A.1), we conclude that: (i) there is not a clear definition of the queries that should be used to timely monitor a building energy consumption, (ii) all the surveyed case studies were conceived without following any formal methodology or framework. In most cases, the authors do not go much further than draw a simple case study of about five queries that they consider to be useful on the domain of their specific research. Aiming at completeness, the case study we present was conceived through a more structured method.

4.2.1 Survey Methodology

As depicted by Figure 4.2, our case study surveys the features that result from the intersection of the two application domains composing the scope of this work:

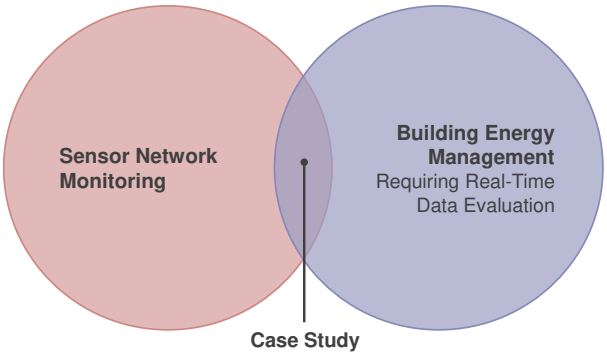

Figure 4.2: Scope coverage of case study queries. The scope of the queries composing the case study is defined through the overlapping of two different, but yet related, domains: (i) the class of queries typically used on real-time monitoring of sensor networks (left), and (ii) the class of queries related with the techniques used to timely manage buildings energy consumption (right).

Author	Domain	
	Sensor Network Monitoring	Building Energy Management
Granderson et al., 2011	○	●
Akhtar and Siddiqui, 2011	●	○
Babcock et al., 2002	●	○
Bizer et al., 2009	●	○
Bonnet et al., 2001	●	○
Chandramouli et al., 2010	●	○
Cranor et al., 2003	●	○
Madden and Franklin, 2002	●	○
Mukherjee et al., 2010	●	○
Zhang et al., 2010	●	○

Table 4.1: Literature references with case studies. Authors analyse the requirements and properties of: Sensor Network Monitoring and Building Energy Management Domains. ●: covered domain. ○: uncovered domain.

Sensor Network Monitoring. The class of queries used to timely evaluate sensor data streams (sensor time-series), which enable the real-time monitoring of a sensor network, such as a building energy metering network.

Building Energy Management Techniques. The class of building energy management techniques that has to timely evaluate huge quantities of sensor data, in order to effectively support the decision making process of building energy managers.

That is, to identify the requirements stated in the beginning of this section, we survey the requirements which result from the intersection of these application domains. To achieve this, our research methodology was to review the literature presented in Table 4.1.

4.2.2 Sensor Network Monitoring Queries

To profile the class of queries used in the sensor network domain, we conduct a survey through a literature review of nearly fifty monitoring queries on several monitoring applications (see Table A.1)

allowing us to conclude the following about the **requirements**: (i) class of queries, (ii) main operators, and (iii) data sources.

Continuous Queries. Monitoring queries are continuous queries, a *class* of long-running queries used to process time-series, that are continuously consuming, evaluating, and producing new data in the form of notifications (alarms) or new data streams (time-series). Note that, a time-series is always changing—at least its temporal dimension is—and therefore the need for a type of query that run continuously to evaluate those changes.

Temporal Correlations. The main *operators* are those which evaluate data through complex time correlations, by performing sensor *data aggregates* over *time windows*.

Correlation of Different Data Sources. Data from different data *sources*, such as *data streams* (as volatile time variant sensor data) and *stored data* (as static relational data from databases), have to be correlated.

Regarding the **requirement** about the *computed results* of queries, namely the one related with the queries produced information, we propose to classify the queries according to the four classes depicted by Figure 4.3: (i) Detection of Abnormal Values, (ii) Data Stream Instant Metrics Summarization, (iii) Database Integration, and (iv) Data Stream Metadata Evaluation. We relate these four classes of queries with the proposed Data Processing Architecture of Figure 4.1 as follows: the first class belongs to the type of queries used on **Data Evaluation** component and the last three classes are the type of queries used on **Data Integration** component. These four classes summarize our judgement on the type of data manipulations performed by the sensor network monitoring queries. More specifically:

Detection of Abnormal Values is a class of high level queries, narrowly related with the application domain, designed to detect abnormal conditions that are taking place on the network and produce the required alarm notifications. It would be hard and error prone to implement these "large-grained" queries on top of the "fine-grained" data data streams that arise directly from the network. Thus, they are implemented on top of the data streams produced by *Data Stream Instant Summarization Metrics* class of queries, those that are structured to facilitate the implementation of the former.

Example: *Produce a notification for each energy meter that are reporting consumptions 10% above the average of the current hour, computed along last month*.

Data Stream Instant Summarization Metrics is a class of data cleaning and integration queries used to transform "finer-grained" sensor data streams into a set of more useful data streams (from the application domain perspective) that will be used by other classes of queries. These queries perform complex data transformations to preform data analytics, such as data aggregation over time windows, detection of patterns, correlate data with different schemas and from different type of sources, data fusion between streams, re-calculate attributes as well create new ones and delete some others, ensure data consistency, etc. The rational behind these queries is to deliver a set of domain bounded data streams which facilitate the implementation of the following ones.

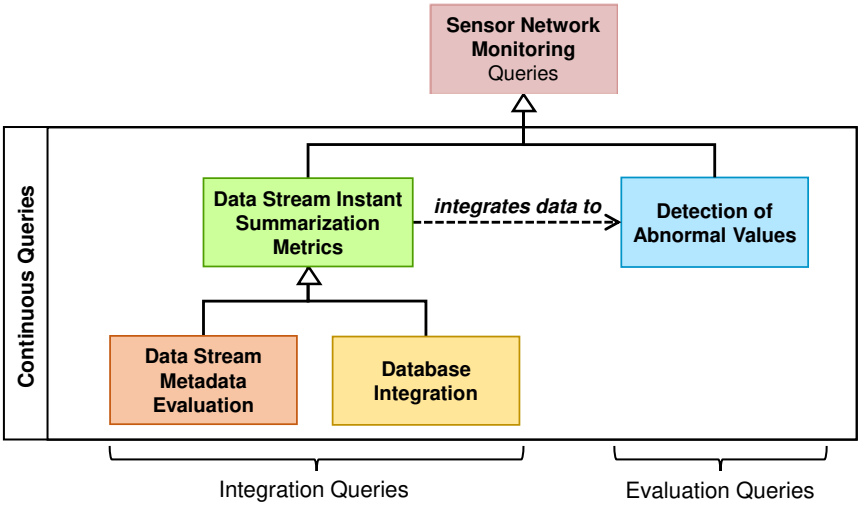

Figure 4.3: UML diagram depicting the classes of sensor network monitoring queries. Sensor Network Monitoring queries classified accordingly to four proposed classes of continuous queries: (i) Detection of Abnormal Values, (ii) Data Stream Instant Metrics Summarization, (iii) Database Integration, and (iv) Data Stream Metadata Evaluation. Being the classes (ii), (iii), and (iv) processed by the solution's Integration component, and (ii) processed by the Evaluation component.

Example: *In order to discard non-representative spike measurements that may spoilt the monitoring process, continuously smooth the energy metering data stream with a 5-minutes moving average.*

Database Integration Queries correlate sensor data streams with persisted data. Thus, these queries main feature is the ability to preform database interactions, such as the retrieval of relational metadata to be joined with sensor data streams, together with database insertions and updates of results computed from the data stream evaluation process, which may be required in the future for further evaluation.

Example: *Normalize the energy consumption readings according the area that is covered by each meter (i.e. normalize to $Watt/meter^2$). Being the area of each meter location made available through a database of sensors metadata.*

Data Stream Metadata Evaluation aims at detecting data stream quality issues that, although beyond the application domain, may interfere with the data analytics process. For instance, to detect data streams that, by being produced by fault energy meters, are presenting data stream tuples with an abnormal periodicity.

Example: *Identify the energy meters producing measurements with a periodicity out of 60$\pm$5 range of seconds.*

4.2.3 Building Energy Management Techniques

It is essential to identify the Building Energy Management Techniques that require a timely data evaluation process, in order to assess the type of data transformations that must be computed by the use-case

queries. Therefore, we have surveyed the literature and concluded that, regarding the scope of this work, those techniques can be classified according to: (i) input data, (ii) data resolution, and (iii) frequency of use [Granderson et al., 2011]. In particular:

Input Data is the type of data required by the technique so it can actually be employed. This data assumes several forms depending on the kind of information required by the technique itself. Still, in most cases, the type of data can be classified as a type of sensor data such as *Energy Meter* and *Weather's Station* measurements, or persisted data such as *Costs and Tariffs*, *Cash Flows*, *Environmental Conversion Factors*, *Usage of Luminaries*, as well as *Historical Baselines*.

Data Resolution is the resolution of the sampling period of input data. That is, the span of time granularity covered by each instance of received data. Building energy management techniques will be arranged according to the values: *Less than one hour*, *Monthly*, and *Annual*.

Frequency of Use is about how often the technique is used in the building energy management process, it determines how often the produced information must be re-computed in order to be kept up-to-date. We will distinguish between the frequencies: *Continuous*, *Monthly*, and *Annual*

A summary of the surveyed techniques according to these requirements is depicted by Table 4.2. The techniques in which we are interested are the ones that *strictly* impose the following conditions: (i) consume input data delivered by a network of **energy meters**, (ii) working data sets with **less than one hour** data resolution, and (iii) that are **continuously** being used to support building energy management process. These conditions highly overlap the three *V's Dimensions of Big Data*, namely: high **volume** of data that has to be processed at highest **velocity**, to continuously deliver up to date information, and that come from an high **variety** on the number of sources. Which will eventually lead us to Big Data scenarios, that are known to be difficult to assess with conventional DBMSs [Beyer and Laney, 2012]. Therefore, the two techniques that narrowly lie in the scope of this work are:

Load Profiling. Evaluates how energy consumption varies within given period of time such as days, weeks, months, or even years, allowing to determine the building energy consumption patterns. Abnormal energy consumption can be easily identified by detecting deviations to the load profiling plot. Moreover saving opportunities can be unveiled by comparing building current load profiling with last profiles of that same building—vertical benchmarking—or by comparing it with the load profiling of other buildings with same characteristics—horizontal benchmarking.

Example: *Identify the energy meters for which the reported energy consumption measurements, along the last 5 minutes, has increased more than 20%*.

Peak Load Analysis. Evaluates the relationship between the minimum and maximum energy consumption along the day. In particular, this technique allows to find potential saving opportunities by shifting some of the load of the day's peak period to periods of lower energy consumption. Shifting consumption does not reduce the total amount of energy consumed during the day, but reduces the maximum load of energy that the building demands from the grid during overload periods.

Building Energy Management Technique	Requirements												
	Input Data							Data Resolution			Frequency of Use		
	Meter Data	Cost	Cash Flow	Conversion Factors	Lighting	Baseline	Weather	**< 1 Hour**	Monthly	Annual	**Continuous**	Monthly	Annual
Simple Tracking	●	○	○	○	○	○	○	●	●	●	○	●	●
Utility Cost Accounting	●	●	○	○	○	○	○	●	●	○	○	●	●
Internal Rate of Return	●	○	●	○	○	○	○	●	●	●	○	○	●
Carbon Accouting	●	○	○	●	○	○	○	●	●	●	○	○	●
Longitudinal Benchmark	●	○	○	○	○	○	○	●	●	●	○	○	●
Cross-Sectional Benchmark	●	○	○	○	○	○	○	●	●	●	○	○	●
Load Profiling	●	○	○	○	○	○	○	●	○	○	●	○	○
Peak Load Analysis	●	○	○	○	○	○	○	●	○	○	●	○	○
Photovoltaic Monitoring	●	○	○	○	○	○	○	●	●	○	●	○	○
Loading Histograms	●	○	○	○	○	○	○	●	○	○	○	○	●
Simple Baseline	○	○	○	○	○	○	○	●	●	●	○	●	●
Model Baselines	●	○	○	○	○	○	●	●	○	○	●	●	○
Lighting Efficiency	●	○	○	○	●	○	○	●	○	○	●	●	○
Heating and Cooling Efficiency	●	○	○	○	○	○	○	●	○	○	●	●	○
Energy Signature	○	○	○	○	○	○	●	●	●	○	○	●	●
Energy Savings	●	○	○	○	○	●	○	●	●	○	○	●	●
Cumulative Sum	●	○	○	○	○	●	○	●	●	○	●	●	○

Table 4.2: Building energy management techniques requiring real-time data evaluation. Summary of the surveyed techniques according to three types of requirements. *Load Profiling* and *Peak Load Analysis* techniques require: (i) input data delivered by a network of energy meters, (ii) by the minute input data resolution, and (iii) are continuously used to support the building energy management plan. ●: covered requirement. ○: uncovered requirement.

Example: *Compute the Min./Max. energy consumption ratio of the building, along last hour.*

4.2.4 Final Use-Case Queries

The above analysis allows us to identify the requirements of the queries used by an EMS to evaluate energy metering data in real-time. It is essential to clearly understand them in order to conceive a representative case study on the building energy management domain. As a result, such requirements were taken into account in the conception of the case study presented in this section, which aims to guide the implementation of the proposed Data Processing Architecture.

The case study consists of 9 Use-Case Scenarios (Q1–9) that are supported, for ease of implementation, by a backbone of 7 Integration Queries (Q10–16), as depicted by Table 4.3 and Figure 4.4. The Graph of Queries presented exemplifies the type of data transformations that must be performed by the proposed Data Processing Architecture. Each query node is a data transformation step that takes part on the implementation of a pipeline of data transformations—capable of streaming data processing—by continuously consuming, processing, and producing new data streams. The graph topology determines how queries interact with each other, by determining with which queries a given query should share its output data stream (that is, its computed results).

The queries of Table 4.3 meet the requirements listed in the beginning of this section as follows: (i) they are *continuous queries*, (ii) most of them performing *temporal data correlations*, (iii) they consume *sensor data*, which in some cases has to be integrated with *persisted data*, being (iv) the type of information computed by each query motivated by the: Sensor Network Monitoring (see Section 4.2.2) and Building Energy Management (see Section 4.2.3) applicational domains.

Q#	Continuous Query Statement	Class of Queries				Technique	
		Detection of Abnormal Values	Data Stream Instant Summary Mettrics	Data Stream Metadata Evaluation	Database Integration	Load Profiling	Peak Load Analysis
	Use Case Scenarios						
1	Identify the energy meters for which the reported energy consumption variation, along the last 5 minutes, has increased more than 20%.	●	○	○	○	●	○
2	Identify the energy meters producing measurements with a periodicity out of 60+-5 range of seconds.	●	○	○	○	—	—
3†	Identify the energy meters that are reporting energy consumption measurements 20% above than the respective average of last 24 hours.	●	○	○	○	○	●
4	For each energy meter, computes the fraction of its reported measurements relative to the total of energy that is being consumed by the building.	○	●	○	○	●	○
5	Sort in decreasing order the energy meters by its current energy consumption measurements.	○	●	○	○	●	○
6†	For each energy meter and building as a whole, compute the Minimum/Maximum energy consumption ratio along last hour.	○	●	○	○	○	●
7	Identify the energy meters that are reporting measurements above a respective threshold.	●	○	○	○	○	●
8†	Identify the energy meters for which the number of reported measurements above its respective expected value, along last hour, lies between 5 and 10.	●	○	○	○	●	○
9	Identify the energy meters that are reporting energy consumption measurements 10% above the average for the current hour, computed along last month.	●	○	○	○	●	○
	Integration Queries to support Use Case Scenarios						
10	For each data stream, aggregate its three-phase current measurements into the total amount of energy being consumed, and then adorn the result stream with the location properties of the respective energy meter, which is available through a database of metadata.	○	●	○	●	—	—
11†	For each data stream, compute the variation of its current measurement value from the moving average of last 5 minutes.	○	●	○	○	●	○
12†	For each data stream, compute the period between its two last measurements.	○	○	●	○	—	—
13†	For each data stream, smooth its measurement values (for noise removal purposes) by applying a 5 minutes moving average.	○	●	○	○	●	●
14	For each data stream, normalizes its energy consumption measurement according to the area that is covered by the respective energy meter (Watt/m^2), and normalizes all building current energy consumption by its total area.	○	●	○	○	●	●
15	For each energy meter measurement, return its current and expected energy consumption value. With expected value given by an User Defined Function.	○	●	○	○	●	○
16†	For each energy meter measurement, return its current and expected energy consumption value. Being each meter expected value the average measurements of the current hour computed along last month.	○	●	○	○	●	○

Table 4.3: Coverage of the 16 Queries used to validate the Data Processing Architecture. 9 Use-Case Scenario Queries (Q1–9) supported by a backbone of 7 Integration Queries (Q10–16). Each query is classified according to the Class of Sensor Network Monitoring Queries and Building Energy Management Technique to which it belongs. ●: matched feature. ○: unmatched feature. —: does not apply. †: Time Window Query.

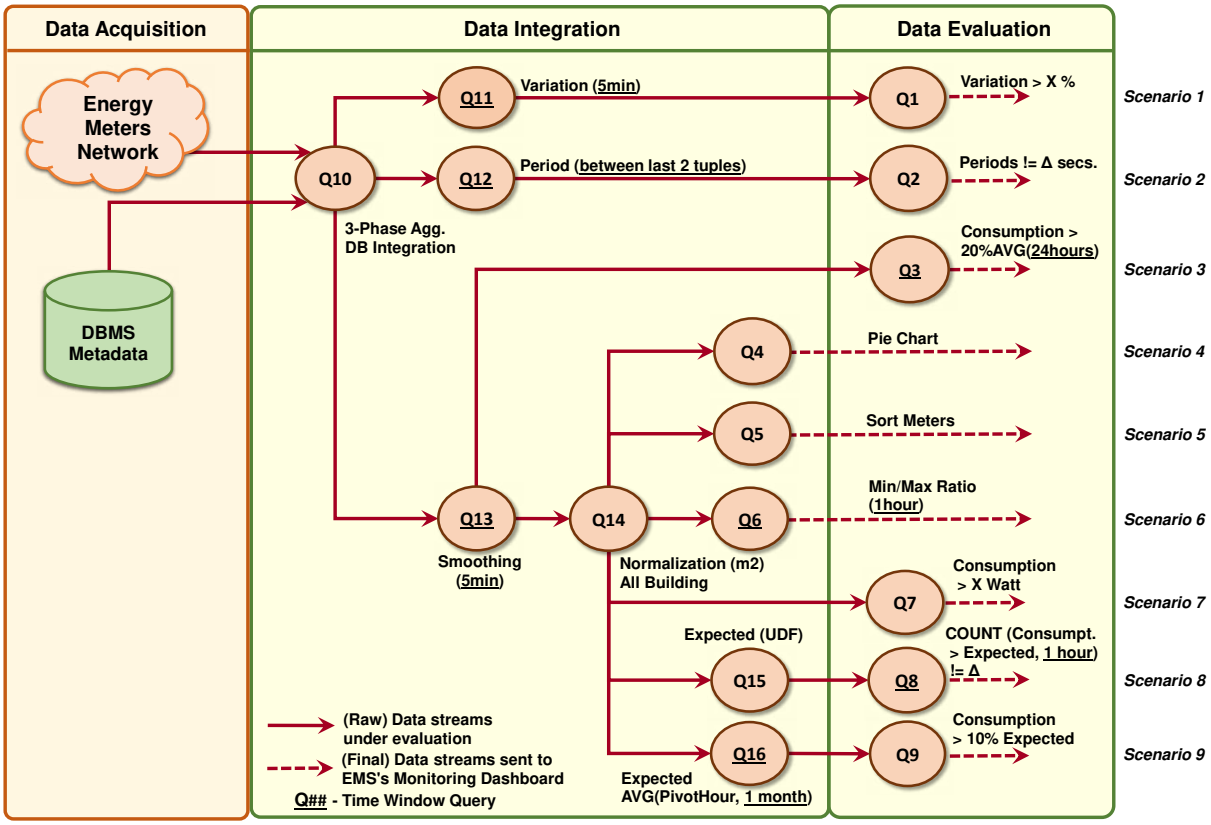

Figure 4.4: Graph of Queries used in the Case Study. In a pipeline of data transformation queries, each query/node represents a data transformation step. Queries Q10–16 form the backbone of data integration queries used to convert raw data into a set of improved data streams that are able to simplify the implementation of queries Q1–9, the use-case scenario queries. Time Window Queries are flagged with an underscore (Q##).

4.3 Case Study

According to the proposed solution, the Data Acquisition Tier is the component that is in charge of provide input data to the Data Processing Tier. In this work, the Data Acquisition Tier was supported by a real network of energy meters deployed on a large scale-building, allowing us to conduct our experiments on a dataset produced by a real scenario.

In this section we describe the energy metering network that was used to support the Data Acquisition Tier, as well the properties of its produced data. The Data Acquisition Simulator that we had to conceive to properly conduct this work experiments is also introduced, we discuss the motivations that led to its conception, together with its architecture and main capabilities.

4.3.1 Building Energy Metering Network

The source data streams used by the Data Acquisition Tier were taken from the energy metering network deployed in the Taguspark[1] University Campus of Instituto Superior Técnico, in the scope of the Smart Campus Project [2]. The network consists of 8 energy meters that are continuously monitoring the energy that is consumed in different types of rooms. The 8 building locations being monitored are depicted in Figure 4.5, being the type of each location, as well its covered area [3], depicted by table Table 4.4.

The energy meters send their measurements periodically to a data ascquisition server, using the Modbus network protocol. Modbus is a data transmission protocol widely used in the industrial automation domain to connect electronic devices being supported by two main Modbus entities, the master and the slaves. In the present context, the Data Acquisition Server, assumes the role of Modbus Master, periodically pulling new measurements from the network of meters, which assumes the role of slaves sending their measurements when requested by the server. Moreover, the server make those measurements available through a RESTful API, which enables any client over HTTP to access the data stream measurements that are being produced by the energy metering network, for instance, to store the measurements in a database. It's worth nothing that, Data Processing Tier of our proposed solution can be easy integrated with this Data Acquisition Server, given the proper Adapter. The building system architecture to collect data from the deployed energy metering network is depicted by Figure 4.6.

Properties of Data Stream Measurements

Each meter data stream comprises **three time-series**, each one for the energy that is being consumed by each phase of the **three-phase current** that supplies the building. A sensor data stream sample produced by an energy meter of the network is depicted in Figure 4.7. It shows, for each phase, the energy consumption of the campus library along one week period. Note that, to know the total amount of energy that is being consumed by a given location, the three phases have to be subsequently aggregated on the Data Processing Tier, as depicted by the fourth data stream (SUM(P1,P2,P3)) of the figure.

[1] `http://tecnico.ulisboa.pt/pt/sobre-IST/localizacao/#tagus`
[2] `http://greensmartcampus.eu/smart-campus-project`
[3] Approximated Values.

Figure 4.5: Building locations monitored by the energy metering network. The deployment location of each energy meter and also its covered area, marked on grounding floor blueprint of the campus building.

Type	Blueprint ID	Area (m^2)
Offices	1	240
	2	60
	3	225
Library	4	500
Lecture Hall	5	110
	6	110
Class Room	7	60
	8	60

Table 4.4: Type and area of each building location being monitored (according to Figure 4.5).

The Data Acquisition Server is configured to fetch new measurements from the energy metering network, every 60 seconds. That is, the **sampling period** of the energy meters is of one measurement per minute, this means that, in theory, the data streams produced by the meters are a sequence of measurements spaced in time by a period of 60 seconds. We denote this concept as the *data stream periodicity*, and a sample of it is depicted by Figure 4.8. The picture illustrates: (i) the *expected stream periodicity* of 60 seconds, the one with which the Data Acquisition server was configured, and (ii) the *observed stream periodicity*, which states the real periodicity of the data stream that was produced by the library's energy meter along one week period. As visible, although the vast majority of measurements acknowledge the expected period of 60 seconds, there is a set of measurements that, due technical setbacks, does not met this period. This observation motivates for the need of data quality assessment queries in the Data Processing Tier, such as the use-case queries Q2 and Q12.

4.3.2 Energy Domain Data Schema

The **Data Schema** specifies the *data structure* that will be manipulated by the *Data Processing Architecture*. More specifically, it describes how the entities of the *Data Acquisition* domain relate each other, in order to efficiently execute queries over the data of this domain. Recall that our domain of interest lies on the data that is produced by the *Sources* of the Data Acquisition Tier (depicted in Figure 4.1), which means that the Data Schema used to support our solution must be capable of manage the entities that

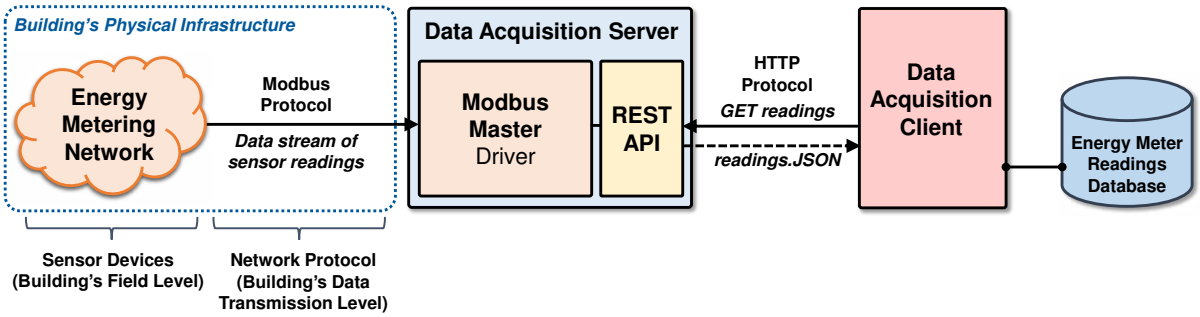

Figure 4.6: Architecture of the system that collects data from energy metering network. Measurements of energy meters are made available to a data acquisition server, through Modbus network protocol. The server (by assuming Modbus Master role) is in charge of periodically pull new data from the meter network and made them available through a RESTful API. Thus, any client running over HTTP may access the building energy metering network.

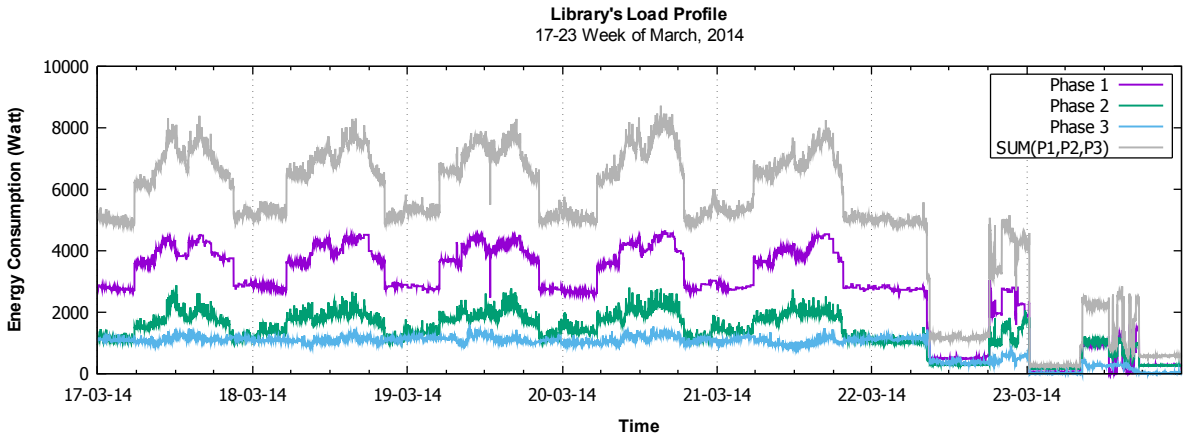

Figure 4.7: Load profile of university campus library along a one week period. Showing the meter measurements for each current phase (Phase 1, 2, 3) produced by the library's energy meter along one week period. For illustrative purposes, since it is not produced by the energy meter, the total amount of energy being consumed is depicted by the sum of these three phases, SUM(P1, P2, P3).

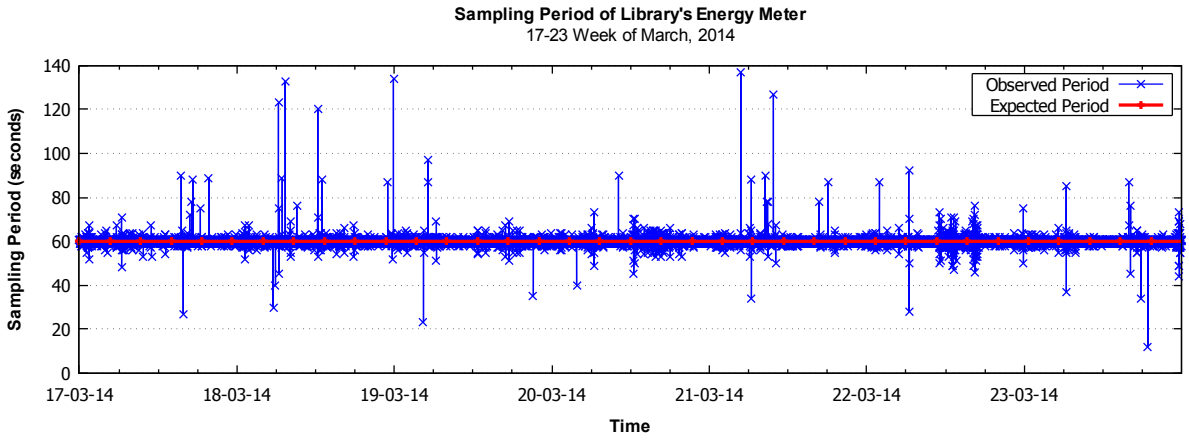

Figure 4.8: Periodicity of energy meter measurements along a one week period. The energy meter is configured to produce a new energy consumption measurement with a period of 60 seconds (red). Yet, the observed values show us that this period is not always fulfilled, producing an energy metering data stream at a non-constant rate (blue).

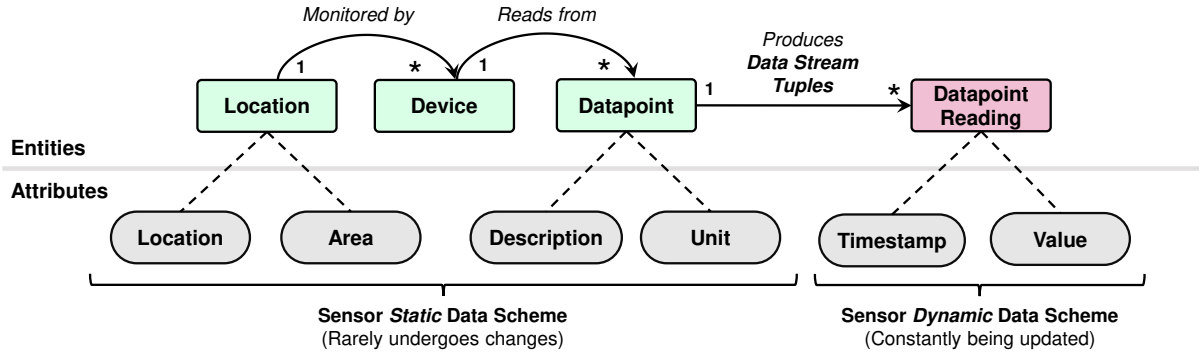

Figure 4.9: Domain model of the energy metering network. The metering network domain is modelled by four entities (and its properties): Location, Device, Datapoint, and Datapoint Reading. The different degrees of data volatility among entities is highlighted: entities on meters metadata (green) hold seldom modified datasets–Static Data; and the entity on meters data stream measurements (red) hold a continuous changing dataset—Dynamic Data.

are related with the building energy metering network.

To assess the entities and relationships that must be taken in account to elaborate such data schema, we review a real EMS database schema (see Figure C.1). This data schema was developed in the scope of **Smart Campus** [4] project with the purpose of modelling the domain of the energy metering network that is deployed at IST Taguspark campus, being used to support several energy management applications. Despite being a fairly complex schema, with many entities and relationships, for the purposes of our work, the original schema can be understood as modelling the domain of the energy metering network as depicted by Figure 4.9. More specifically:

Location. Is the building location that is being monitored by a given *Device*. Which, in the scope of this work, always assumes the form of an Energy Meter. Moreover, the entity is described through its building *location* and covered *area*.

Device. Is the entity that represents all types of sensors deployed at the building, such as: Energy Meters, Equipment Sensors, and Environment Sensors. As said before, in this work, it serves to represent the Energy Meters.

Datapoint. Identifies the measurement points that compose each device. In our case, each device (energy meter) has three datapoints, one to measure each phase of the tree-phase current that power supplies the building. The entity is described by the *description* and the *unit* of the datapoint measurement value.

Datapoint Reading. Are the time-variant values periodically produced by the datapoints of each device. In the case of this work, given the existence of three datapoints per device (three-phase current), each **measurement** that is produced by a device is composed by *three* data stream **tuples**. Each one of the three datapoint readings is described by a *timestamp*, that will be the same for the three tuples belonging to the same measurement, and by the tuple's *value* for each phase.

The most notable aspect of this domain is the different degrees of data volatility among the entities, that is, the different frequency with which the data of the entities is updated. More specifically, entities

[4] `http://greensmartcampus.eu/smart-campus-project`

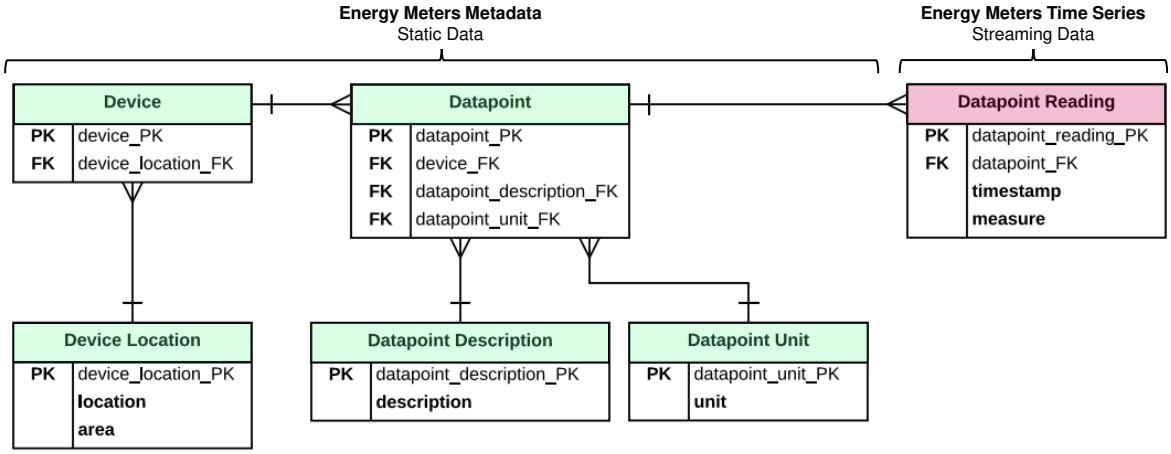

Figure 4.10: Entity-Relationship diagram of the Energy Metering Network domain (using Crow's Foot Notation). There are five entities concerning energy meters metadata containing data that is rarely updated—static data entities. The Datapoint Reading entity is in charge of hold the sensor data streams produced by the energy metering network—streaming data entity.

on meter metadata (Location, Device, and Datapoint) hold datasets that are seldom modified—Static Data. Whereas the entity on energy metering data streams (Datapoint Reading) holds a dataset that is continuously being updated—Dynamic Data. This domain model is converted into the data schema depicted in Figure 4.10. Although it is a simplified version of the original data schema (please refer to Appendix C.1 for a complete version), it preserves the essential aspects of the original ER diagram for this domain. Making clear what we have already concluded in Section 2.2.2, namely: that energy management applications rely on DBMSs to timely process streaming data. The ER diagram of Figure 4.10 is the one used to support the prototype of our proposed solution.

Finally, the manner how the data schema is addressed depends on the type of query engine that supports the Data Processing Architecture. Being the different approaches of addressing this data schema strongly related with **the main concern of this work**:

- In the **DBMS Based Solution**, all six entities are persisted in the database. This is the most suitable approach for the five static data relations. However, streaming data relation **Datapoint Reading** will also be persisted, meaning that the **energy metering data streams will be processed *offline***.

- In the **DSMS Based Solution**, just the five static data relations are persisted in the database. Whereas the relation **Datapoint Reading** is processed on-the-fly by the DSMS, instead of being persisted in the database. Meaning that the **energy metering Data Streams will be processed *online***.

The metadata values that were used to populate the static relations of the data schema are illustrated in Figure D.1. Note that the high level of normalization of this data schema (that comes from the original schema and characterises OLTP databases) will impose an extensive use of *join operations* to integrate the *static* and *streaming* data. However, this is the type of data transformations that should be performed in the Data Integration component of the Data Processing Architecture. Therefore, this integration will be made by the Data Integration Queries.

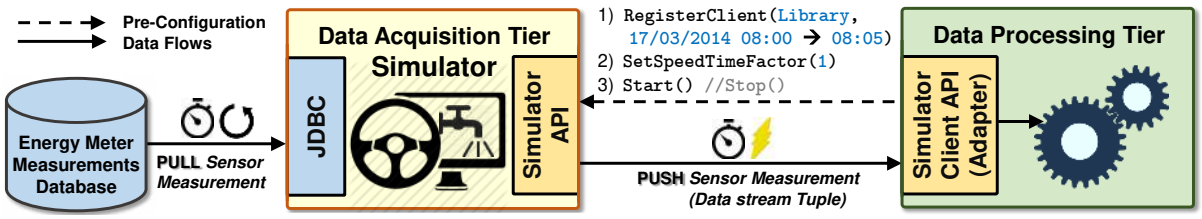

Figure 4.11: Architecture of the Energy Metering Network Simulator. The simulator is built on top of a historical dataset of measurements that were previously retrieved from the network of energy meters. After specifying the target meter and time period, the simulator client initiates the simulation job. As a result, the simulator will periodically push to the client the energy consumption measurements, thus mimicking the sensor network.

4.3.3 The Energy Metering Network Simulator

To properly evaluate our solution, some aspects regarding the data streams produced by the energy metering network must be rigorously controlled and predictable. More specifically:

Deterministic Datasets are required to properly assess the output result of each query, and thus validate their implementations in a repeatable fashion. If we test the queries with data streams that are coming *directly* from the energy metering network, then, to know if the produced output is the correct one, we have first to determine (by ourselves) the expected result of that query for *that* dataset, that will be always different for each time we execute the query. Moreover, to run the final benchmark evaluation, we have to ensure that the same dataset is used in the evaluation of both implemented versions of the proposed solution, otherwise we could not make any assumption about the dataset fairness.

Configurable Time Ranges to properly validate our solution by carefully selecting the time interval of the dataset that will exercises the scenarios under evaluation, to do not have to wait until a given data pattern appear.

Configurable Load is required to control the throughput of data that is sent from the energy metering network to the Data Processing Tier. Allowing to evaluate how the performance of the component is affected periods of overload, such as energy meters producing data with a much greater frequency than one measurement per minute.

To precisely calibrate the above mentioned aspects, we conceived a Simulator that mimics the energy metering network deployed at a large facility, the IST Taguspark university campus. A database of historical measurements, that was retrieved from the energy metering network, is used to support the simulator implementation, as depicted in Figure 4.11. The historical database was populated by a Data Acquisition Client (see Figure 4.6) along nine months (from January to September, 2014), which stored the energy metering data streams that were produced by the eight network meters. A sample of one of those persisted time-series is shown by Figure 4.12.

During the development and evaluation process, Data Processing Tier will receive energy metering data streams from the Simulator instead of the network of energy meters. To do so, Data Processing Tier will connect to the Simulator through its Client API (see Figure B.1) to configure and initiate a simulation job. Is this configuration that will specify the time interval covered by the simulation, the energy meters

	measure_timestamp timestamp(6) without time zone	meter_location unknown	phase1 double precision	phase2 double precision	phase3 double precision
1	2014-03-17 08:00:09	Library	3566.376	1504.8	981.862
2	2014-03-17 08:01:05	Library	3527.964	1683.837	970.445
3	2014-03-17 08:02:05	Library	3527.964	1486.85	961.824
4	2014-03-17 08:03:05	Library	3481.236	1688.96	957.696
5	2014-03-17 08:04:05	Library	3504.6	1505.01	957.696
6	2014-03-17 08:05:05	Library	3504.6	1516.587	957.696

Figure 4.12: Sample of a 5-minutes energy metering data stream stored in the simulator database (pgAdminIII output terminal). Each building energy meter is represented in the database through a table that stores the times series produced along a given period of time.

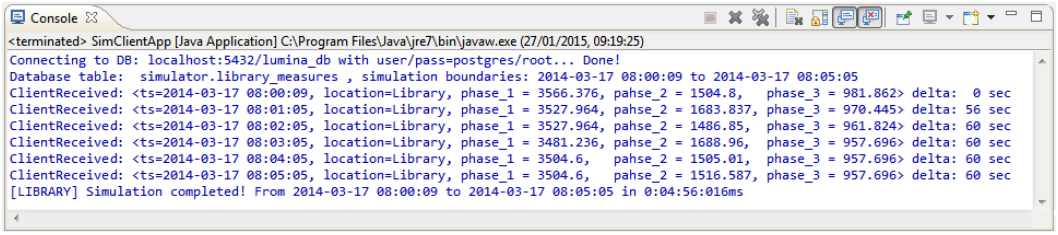

Figure 4.13: Sample of an energy metering data stream sent from the simulator to the client. A sample of five minutes of simulation is depicted (Eclipse IDE's terminal).

under simulation, and each meter production rate of measurements. After that, the Data Processing Architecture (as simulator client) may start, stop, and resume the simulation job as needed. Figure 4.13 presents a sample of the data that is received by Data Processing Tier along a simulation job.

The simulation is consistent with the real production rate of the original data streams. That is, the simulator complies with the period that exists between each data stream measurement, meaning that the simulator will wait the *usual* 60 seconds before pushing a new measurement to the client. Therefore, from the client's perspective there is no difference between the simulator and the original building energy metering network, since the simulator also acknowledge the period of time between each new produced measurement. This property allow us to increase the simulator data production rate, by increasing the frequency of each energy meter being simulated. This is possible through the parameter `SpeedTimeFactor`, that specifies the division factor that must be applied to the regular 60-seconds period of each measurement. For instance, lets consider the simulation of an energy meter that produces a new measurement every 60 seconds with the following configuration, `SpeedTimeFactorParameter = 4`. According to this setup, the simulator will push a new measurement to the Data Processing Tier every 15 seconds (instead of 60 seconds), resulting in a throughput of 4 measurements per minute (instead of 1 measurement per minute).

Development Technologies

The Simulator was developed in Java [5] [6], the Energy Metering Measurements Database is supported by a PostgreSQL [7] server, and the communication between this two components is made through a JDBC. For auditing purposes, the Simulator is made available through a public code repository at GitHub [8].

[5] Java compiler version: 1.7.0_51-b13
[6] JVM version: Java(TM) SE Runtime Environment (build 1.7.0_51-b13),
Java HotSpot(TM) Client VM (build 24.51-b03, mixed mode)
[7] PostgreSQL 9.3.4, 64-bit (build 1600)
[8] `https://github.com/diogo-gsa/building-energy-meters-network-simulator`

Chapter 5

Evaluation

In order to fairly evaluate our proposed solution, we deploy two prototype versions of the proposed Data Processing Tier. One supported by a DSMS, to assess the implementation feasibility of the proposed solution; and the other supported by a DBMS, with the purpose of perform a side-by-side benchmark evaluation between the two architectural approaches. In this manner we assess the performance of the proposed architecture by comparing it with a DBMS based solution. Thus, enabling to validate the hypothesis that: an EMS based on a DSMS performs better, on timely data processing and ease of queries implementation, than common solutions based on a DBMS.

The chapter begins by introducing the methodology used to evaluate the proposed Data Processing Tier, in Section 5.1. Thus Section 5.2 evaluates the query evaluation model of both query engines under analysis. That is, it evaluates the expressivity of each system to support the stream data processing components of Data Processing Tier. To finalize, Section 5.3 evaluates the performance of each solution towards its ability to process energy metering data streams in real-time.

5.1 Methodology

For benchmarking purposes, two independent solution versions of the proposed Data Processing Tier were implemented, each one supported by a *different* query engine. The version supported by a DSMS aims at validating the feasibility of the proposed architecture to process energy metering data streams in real-time, see Figure 5.1 (a). The version supported by a DBMS is based in the state-of-the-art DBMS solutions, and will serve the purpose of properly evaluating the issues that makes it an unsuitable engine to timely process data streams, see Figure 5.1 (b). A great advantage of this setup is the ability to switch between these solutions whenever required. Moreover, this arrangement allows the side-by-side benchmark needed to validate our proposed hypothesis: that an EMS supported by a DSMS performs better than the common solutions supported by a traditional DBMS. By better performance, we mean:

1. Provide a most suitable query language to develop energy management applications.

2. The ability to process energy metering data streams in real-time.

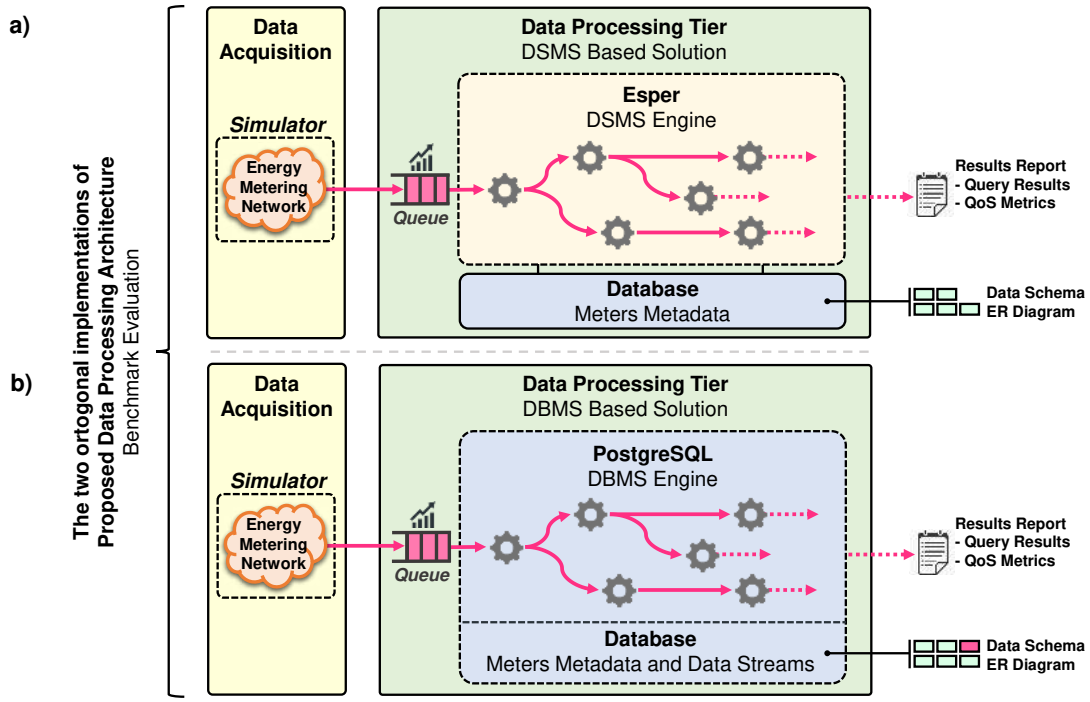

Figure 5.1: Data Processing Architecture supported by two distinct query engines. Two types of query engines were used—DSMS and DBMS—to support the implementation of two orthogonal versions of the proposed Real-Time Data Processing Architecture. DSMS version assess the implementation feasibility of proposed solution (a), and DBMS version is required to run a benchmark validation between the two versions (b).

To validate this hypothesis we have proceeded in the following manner:

1. We implemented the use-case queries identified in Section 4.2.4 in both solutions of the Data Processing Tier. The implementation of these 9 use-case scenarios allows us to assess the ability of each query engine to **implement queries related with this problem domain**.

2. We measured the time it takes for each solution to process energy metering data streams in each use-case scenario, according to the quantity of data already processed. This allows us to assess the capacity of each version of the Data Processing Tier to **process data streams in real-time**.

The relevant components of this setup are detailed below.

5.1.1 Selection of Query Engines

Among the several query engines available (see Chapter 3) to implement both versions of the proposed Data Processing Architecture, we choose the following open-source engines:

1. **Esper** [1] to support the DSMS version of Data Processing Architecture.

2. **PostgreSQL** [2] to support the DBMS version of Data Processing Architecture.

As discussed earlier in Section 3.6, we concluded that the solution implementation would be supported by Esper, a DSMS able to process *Continuous Queries* (CQs) (see Section 2.1.2) over unbounded data

[1] Esper 5.0.0 – `http://www.espertech.com/download`
[2] PostgreSQL 9.3.4, 64-bit (build 1600) – `http://www.postgresql.org/download`

streams. Thus, the architectural data transformation components—Data Integration and Evaluation—were implemented as a composition of CQs, to support the data transformation graph of Figures 4.1 and 4.4. Those queries are expressed through Esper EPL declaratively language, and transparently compiled in an optimal query evaluation plan. Moreover, the Esper SQL-like query language allows a straightforward side-by-side comparison between the respective use-case queries implemented in both solution versions. Therefore, Esper was used as the key building component of Data Processing Tier.

There are several types of DBMSs (see Section 3.1.1). We opted by PostrgreSQL because, as identified in Section 2.2.2, it represents the most widely chosen type of DBMS to support EMSs. Since DBMSs do not support CQs, data transformation graph has to be implemented with One-Time Queries (see Section 2.1.2). Meaning that, to process the arriving sensor data streams in a continuous manner, the use-case queries have to be repeatedly executed whenever a change is made in the persisted set of data streams.

5.1.2 Input Energy Metering Data Streams

The **energy metering data streams** used as input data in the evaluation experiments of the Data Processing Architecture are produced by the Energy Metering Network Simulator introduced in Section 4.3.3. The data produced by this simulator was originated by the energy metering network that is deployed at the IST Taguspark university campus, meaning that the solution evaluation experiments were supported by data from a real world scenario. The usage of the simulator allows us to perform experiments in a more rigorous manner, letting us to: (i) run experiments with expectable results, (ii) repeat the same experiment several times, under the same conditions, in both prototype versions of the solution, and (iii) select the most suitable load profile periods for each experiment.

5.1.3 Input Data Queue

Both solutions were implemented with an Input Data **Queue** (introduced in Section 4.1.1). This queue holds data produced by the energy metering network (Simulator), to then be *consumed* by the query engine—that is, to be processed through the data transformation graph (illustrated back in Figure 4.4). The query engine that is continuously consuming data from the queue (following a FIFO policy) will *block* in the presence of an empty queue and *wait* for new measurements. As explained in Section 4.1.1, the main purpose of the queue is to hold the excess of data that results when the sensor network production rate *exceeds* the engine consumption rate, to avoid loss of data. When it happens, the quantity of queued measurements (waiting to be processed) increases and, therefore, the time it takes to process a *fresh* measurement that has just arrived to the queue will also increase—which may indicate a degradation of the ability of the system to process data in real-time. Therefore, we will monitor the size of the queue during the system operation, to assess if data is being processed in real-time.

5.1.4 Data Schema

The **data schema used in the database** of both versions of the Data Processing Tier is depicted by Figure 4.10, and introduced in Section 4.3.2. As mentioned before, there is a remarkable difference on how each version of the Data Processing Tier uses this schema. DSMS Based Solution only uses the entities related with static data, the ones storing energy meters metadata. The entity "Datapoint Reading" used to store the energy metering data is not used, since the streaming data is processed online by the DSMS. In contrast, DBMS Based Solution uses all entities of the diagram, the ones related with both static and streaming data, meaning that the streaming data will be processed offline by the DBMS. This is a key difference between the two different solution approaches of the Data Processing Tier, they differ on how each one process the data streams produced by the energy metering network.

5.1.5 Produced Output and Query Results

A **Results Report** is maintained by each solution to log the information that is produced in the course of data processing. For each energy meter measurement consumed from the queue and processed through the data transformation graph, a new entry is added to the log, which records the new result value of each use-case scenario. It also records QoS metrics collected during the tuple evaluation process, such as the time that it takes to a measurement to traverse the graph topology (that is, to be evaluated by the use-case scenarios), the current number of measurements in the queue still waiting to be processed, and the total number of tuples that were evaluated so far. This report will be used to asses the ability of each solution to process data in real-time.

5.1.6 Development Technologies

Both prototype versions of the Data Processing Architecture were developed on top of a stack of open source technologies. They were implemented in Java [3][4], being the DBMS solution supported by PostgreSQL [5] and the DSMS solution supported by ESPER [6], as query engines. For auditing purposes, both prototypes were made available through a public code repository at GitHub [7].

5.2 Query Language Evaluation

This section describes the implementation of the case study (recall it from Table 4.3 and Figure 4.4) together with the development of both versions of the Data Processing Architecture. We aim to evaluate how suitable is the query language of each query engine to write and evaluate queries on this application domain. Our discussion will focus on the design decisions taken to address the challenges raised by the

[3] Java compiler version: 1.7.0_51-b13
[4] JVM version: Java(TM) SE Runtime Environment (build 1.7.0_51-b13),
Java HotSpot(TM) Client VM (build 24.51-b03, mixed mode)
[5] PostgreSQL 9.3.4, 64-bit (build 1600)
[6] Esper 5.0.0 – `http://www.espertech.com/download`
[7] `https://github.com/diogo-gsa/data-processing-architecture`

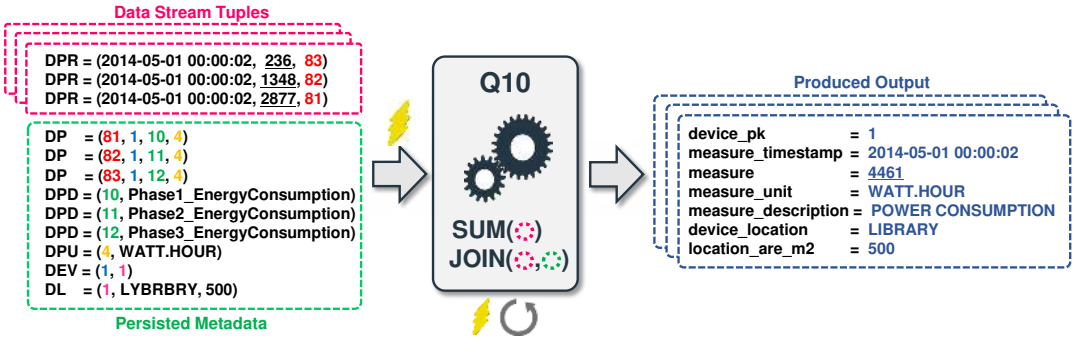

Figure 5.2: Evaluation process of use-case query Q10. The query integrates the arriving data stream tuples (produced by the energy meters) with its persisted metadata, and aggregates the three-phase data point readings. To maintain its output up-to-date, the query must be evaluated continuously in order to process de tuples that arrive from the network of energy meters.

implementation of the case study in both solutions, being the use-case queries used when necessary to illustrate these challenges. The implementation details of each query can be found in Appendix E.

5.2.1 Achieving Continuous Queries Behaviour on a DBMS

As said before in Section 2.1.2, there is a mismatch between the type of queries that are yield by a DBMS and the continues evaluation model required to process this work class of queries. For instance, consider the evaluation of use-case query Q10, depicted in Figure 5.2. The query denormalizes the arriving energy metering data stream according to the data schema of Figure 4.10, and computes the SUM(P1,P2,P3) of Figure 4.7 by aggregating the three phases composing each meter measurement. Since the energy metering network is continuously producing new measurements, the query has to be evaluated in a continuous manner, in order to maintain its produced information up-to-date. Therefore, in order to implement the use-case queries, the DBMS Solution has to be capable of implementing the behaviour of *continuous queries*. In other words, it must react spontaneously to the data streams that are continuously being pushed to, and consumed from, the input data queue. To achieve this, queries must be *explicitly* executed whenever a new energy meter measurement is persisted in the database (in the "Datapoint Reading" relation). In the following manner:

```
1 public class DBMS_VersionImpl implements SimulatorClient{
2  ...
3  private void consumeTupleFromQueue(EnergyMeterMeasurementDTO measurement){
4    databaseEngine.insertIntoDatapointReadingTable(measurement);
5    QueryEvaluationReport logEntry = databaseEngine.executeIntegrationQuery10(); /* Polling */
6    resultsReport.addNew(logEntry);
7 }}
```

It is worth noting that the **DBMS solution**, in order to simulate queries that run periodically, has to submit them to the query engine in a **synchronous** manner. This is required due to the *passive* interaction style of DBMSs, in the sense that they expect the applications to coordinate the operations that they must perform.

Some authors (such as Paton and Díaz [1999]) suggest the usage of **Triggers** to enhance the interaction between applications and (active) databases, still such triggering mechanism is far from being an adequate solution to continuously execute queries and *notify* the client application with the result-

ing dataset. Triggers are useful to enforce database consistency. They can automatically propagate changes across database tables and check for the violation of domain integrity rules. However, there is no obvious way of using triggers to simulate queries that run periodically (whenever a data stream tuple is persisted) and *asynchronously push* the resulting dataset to the application—which would prevent the application to rely on a **polling approach** [Stonebraker et al., 2005]. This requirement, as well the query polling design decision, applies to the implementation of all use-case queries.

In the **DSMS Solution**, continuous queries are natively supported, and therefore the application only has to **push** the arriving data streams into the query engine. The queries that has as input the pushed "Datapoint Reading" tuples (Q10, in our case), will spontaneously react and evaluate those data stream tuples, to then *asynchronously* notify the application, through a notification handler, of the produced results. Therefore, in this solution, the the query execution model differers from previous one, as follows:

```
1  public class DSMS_VersionImpl implements SimulatorClient{
2   ...
3   private void consumeTupleFromQueue(EnergyMeterMeasurementDTO measurement){
4     esperEngine.pushDatapointReadingTuples(measurement); /* Pushing */
5  }}
6  /* Query Results Notification Handler */
7  public class QueryListener_Q10 implements UpdateListener{
8   ...
9   @Override
10  public void update(EventBean[] newEvents, EventBean[] oldEvents) {
11   resultsReport.addNew(newEvents,newEvents);
12 }}
```

5.2.2 Creating a Pipeline of Data Transformations

As explained in Section 4.1.1, the queries (data transformation steps) of our Solution must be interconnected with each other in order to make it possible (i) for each query to push its computed output as input to another queries, which is essential to formulate a chain of data transformation steps, and also (ii) to be able to simplify the elaboration of complex data transformation scenarios through the composition of simpler data transformations steps (integration queries) that can be reused by different scenarios simultaneously. Such pipeline is required to support the graph of queries depicted by Figure 4.4.

In the **DBMS Solution** version, **Materialized Views** were used to implement such pipeline. At first we try to use "standard" Views but, as we will se below, they proved not to be the most suitable solution. The pipeline is implemented by declaring the queries Q10–Q16 on *"top of each other"* (as visible in Section E), and to properly evaluate a use-case scenario, the application must *explicitly* execute each one of its queries *sequentially* from the left to the right part of graph. For instance, to execute *Scenario 1*, the application must execute the sequence of queries Q10, Q11, and Q1.

To speed up query evaluation, we installed **Indexes** on Datapoint Reading (DPR) table of Figure 4.10 schema, which seems a reasonable decision given the amount of data (the energy metering measurements) that it stores. Some experiments were conducted to validate this assumption, and also to understand how indexes may impact the evaluation performance of a given scenario. By performance we mean the time it takes for a scenario to evaluate an energy meter measurement, according to the number of data point reading tuples persisted in the database (that is, the quantity of tuples processed so

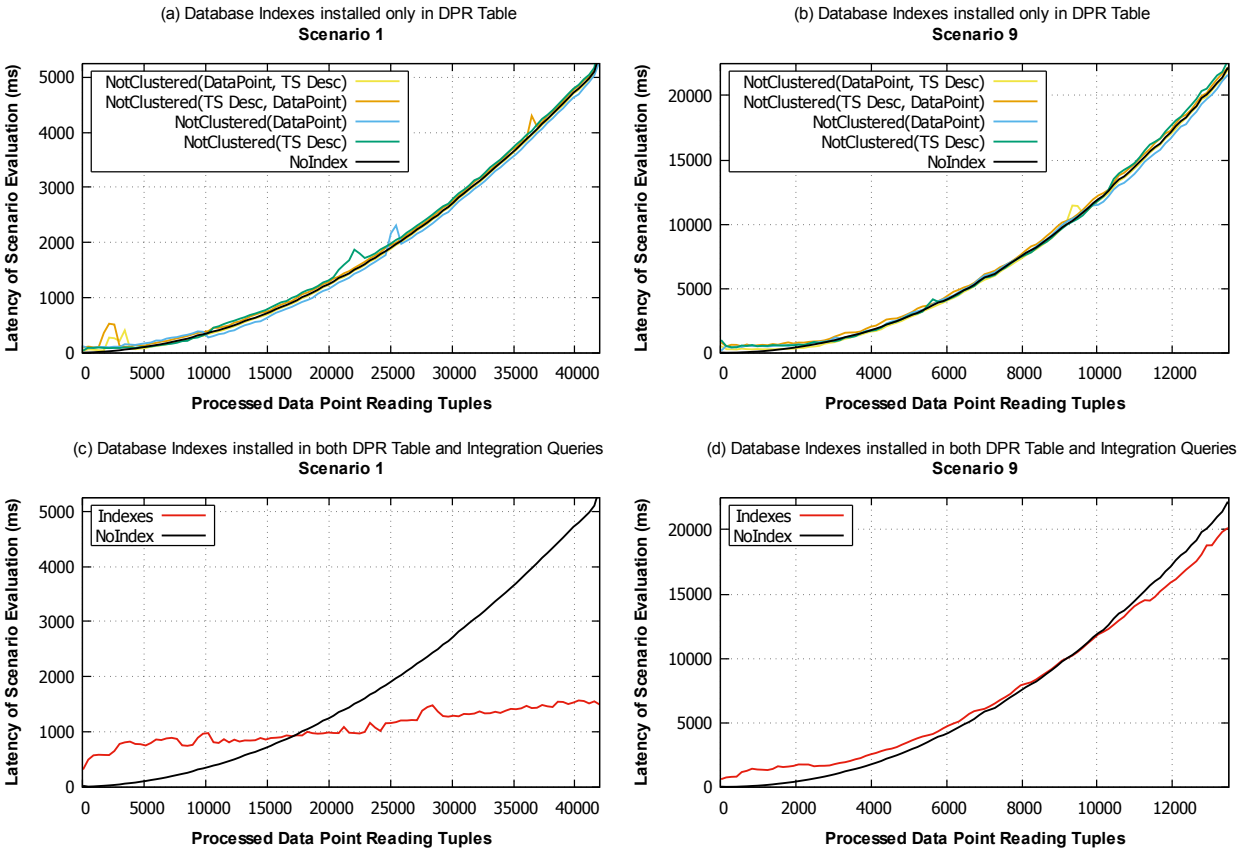

Figure 5.3: Performance of use-case scenarios using distinct types of database indexes. Graphs display the evaluation of the time taken to evaluate Scenarios 1 and 9 as a function of the quantity of data point readings persisted in DPR table, according to different index configurations: four types of indexes independently installed in DPR table versus a setup of no indexes, (a) and (b); indexes installed in both DPR table and Integration Queries, according the configuration of Table 5.1, versus a setup of no indexes, (c) and (d).

far). Our method was to execute two different scenarios (one at a time) for each one of the five different index configurations installed on DPR table. We choose four indexes to deploy in DPR table together with a no index configuration, and test each one through the execution of each scenario over a given set of data stream measurements. *Scenario 1* was used as an example of a less exigent scenario, that requires less demanding data processing operations, while *Scenario 9* was used as the worst case scenario that has to perform a set of quite complex data processing operations. Being the queries of each scenario implemented as “standard” Views in this first row of tests.

The results of the five index configurations are depicted in Figure 5.3 (a) and (b). The main conclusion is that **our initial assumption was wrong**: indexes installed only in DPR table have virtually no impact in the performance of scenarios evaluation, since there is almost no difference between the performance results achieved by the usage of indexes and the results achieved by using no index. This happens because, from all queries that compose each scenario, Q10 is the only one who takes advantage of the Indexes installed on the DPR table, which is not sufficient to import a noticeable impact on the overall performance of scenario evaluation.

To truly enhance performance via indexes, they have to be used not just in the DPR table but also in all the integration queries supporting the backbone of data processing pipeline, that is queries Q10–

Installed Indexes	DPR Table	Integration Queries						
		Q10	Q11	Q12	Q13	Q14	Q15	Q16
NotClustered(DataPoint)	●	○	○	○	○	○	○	○
NotClustered(Device, Timestamp Desc)	○	●	○	○	●	●	●	○
NotClustered(Index Desc)	○	●	●	●	●	●	●	●

Table 5.1: Indexes deployed in both DPR table and Integration Queries. The type of Indexes, and their column attributes, that were installed in both DPR table and Integration Queries (as Materialized Views). There are some queries with more than one installed Index due the different kind of data retrieval operations that are performed over these queries by the other queries.

16. Such queries were initially implemented as "standard" *Views*, which makes them unable to support indexes. Therefore, queries Q10–16 were re-implemented as *Materialized Views* so they can be configured with indexes. To select the most suitable index for each integration query, we have first to understand that the most suitable index depends on the data retrieval operations that are made on this query by the other queries. For instance, the most adequate index (or set of indexes) to be installed in Q0 is driven by the data lookup operations that Q11, Q12, and Q13 perform over Q0. Knowing that there is no a single index that perfectly fits in all Q10–Q16 queries, each query has to be evaluated individually in order to realize which is the most appropriate index (or set of indexes) for it. From this evaluation and taking into account that PostgreSQL only provides **NotClustered** type of indexes—those which sort data *logically* without propagating the sorting order *physically* into disk—, we propose the index configuration depicted in Table 5.1 as the most suitable one to improve the evaluation performance of scenarios. Note that, the attribute *Index* of *NotClustered(Index Desc)* is not related with any database index mechanism and will be discussed below.

To evaluate the performance of this new configuration we conduct a second row of experiments, following the same methodology. The purpose is to understand if the performance of a scenario may be enhanced trough the installation of indexes not only in the DPR table but also in all the integration queries (which force them to be implemented as Materizalized Views). To do so, we benchmark this approach against a configuration of no indexes, allowing queries to be implemented as "standard" Views. The experiment results are depicted in Figure 5.3 (c) and (d). They show that, in fact, the time it takes for a scenario to be evaluated may be improved by the usage of Materialized Views if fine-tuned with Indexes. However, some remarks should be made about these results: Firstly, the performance improvements of an Index based configuration are not visible in the short term, in fact the sizeable maintenance cost of both Materialized Views and its Indexes imposes a considerable performance overhead in the initial phase of the experiment. Nevertheless, despite the initial overhead, the performance of scenarios supported by Indexes can exceed the performance of these same scenarios without Indexes, within a reasonable space of time. In fact, assuming a network of eight energy meters producing (three-phase) measurements within one minute periodicity, this turning point is reached nearly after 12 and 6 hours of network monitoring, respectively for Scenarios 1 and 9. Secondly, yet both scenarios take advantage of this Index configuration, the *scalability* of each performance improvement is strongly related with the complexity of the scenario under evaluation. Thus, Scenario 1, which is by far less complex than Scenario 9, has a much more smoothly performance degradation than the one of Scenario 9. As a result,

in the DBMS Solution the pipeline of data transformations was supported by Materialized Views, being each query fine-tuned according to the indexes of Table 5.1.

In the **DSMS Solution**, the pipeline of data transformations is straightforwardly supported by **Output Streams**. Continuous queries output its computed results as data stream tuples (see `Insert Into` expression in Q10–16 of Section E) that are automatically pushed to the following queries in the pipeline, which highly resembles the desired behaviour for the "edges" of the graph of data transformations (see Figure 4.4). Therefore, to properly evaluate a use-case scenario, the application just has to push the energy metering measurements into the first query of the pipeline, Q10 in our case. For instance, to execute Scenario 1, the application just has to push the data stream measurements into Q10, that in turn will automatically push its computed output to the following queries. This will make the energy metering data streams to traverse the entire graph from the left to the right part of its topology.

Esper also has an indexing mechanism to enhance the performance execution of its queries. However, indexes are created and maintained by the query engine itself (in a transparent manner) without the need of *any* previous configuration [EsperTech, 2014, p.179].

In conclusion, the DBMS Solution consists of a pipeline of queries supported by a "chain" of Materialized Views that must be explicitly "reloaded", one at a time, in order to process the energy metering data streams. Indexes can be used to enhance the performance of scenarios evaluation, yet the query designer has to manually select the most suitable ones and install them on queries. In turn, the DSMS Solution is based on a pipeline of queries supported by a "chain" of Output Streams, which propagate the energy metering data streams through the pipeline of data transformations without external intervention from the application. Moreover, the index mechanism is also managed in a completely transparent manner by the query engine optimizer.

5.2.3 Time Windows and Temporal Data Correlations

Many of the use-case queries identified in Table 4.3 are **Time-Window Queries**. They require performing complex time correlations over sensor data streams by evaluating data aggregation operators over time windows.

In traditional **DBMS Solutions** windowing queries are typically implemented through a **Self-Join** of the table holding the data stream, with a condition on the *Timestamp* attribute to specify the time window boundaries. Figure 5.4 exemplifies the computation of a three minute sliding average. However, there is a remarkable aspect in how DBMS queries produce their output that leads to a quite significant impact on the specification of time windows (self joins). DBMS queries are processed in batch, meaning that from the entire dataset that is evaluated only a single result set will be produced as output. This output is produced from the current state of the database at the time the query is evaluated, and it is called a query **snapshot**. Each query results in a single snapshot at a time and it is replaced for a new one whenever the query is re-evaluated. A snapshot is a materialization of the result of a query. Therefore, at the graph, the snapshot of a query Q_n will be used as input by an upcoming query Q_{n+1}. If Q_{n+1}

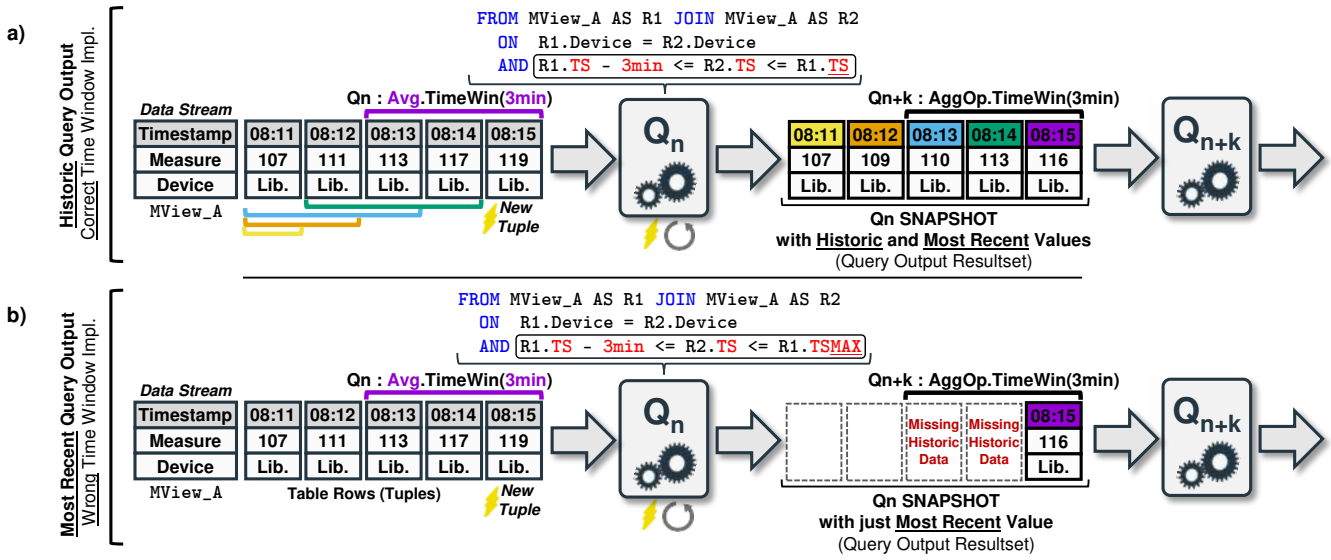

Figure 5.4: Implementation of a Sliding Time-Window query in SQL with a Self-Join operation. The design of the self join condition determines the type of resultset—snapshot—produced by the windowing query. The queries of this work must be designed in order to output the entire data stream processed so far—Historic Snapshot (a). Instead of just output the evaluation results for the most recent measurements of each meter—Most Recent Snapshot (b). Otherwise, would not exist enough data to compute some windowing queries.

is a windowing query, then the snapshot of Q_n has to hold this query evaluation results for all the data stream processed so far, see Figure 5.4 (a). Otherwise, upcoming queries Q_{n+1} will have missing data to properly compute their time windows, as illustrated in Figure 5.4 (b). As a result, the existence of time window queries requires all other queries (even the ones that are not windowing queries) to produce snapshots comprising the results of all data stream processed so far, which we designate as **historic snapshots**, instead of snapshots comprising only the most recent computed value for each device, which we designate as **most recent snapshots**.

The requirement of computing historic snapshots introduces a quite significant overhead in the query evaluation process. Each use-case query has to be (re)evaluated for each new arriving energy meter measurement, meaning that, besides the new measurement, *all* stored data stream will also be (re)evaluated in the process, in order to compute the historic snapshot, otherwise it will be erroneously produced a most recent snapshot. Meaning that, the computations that have already been preformed on the previous executions of the query will be repeated, in order to output the (same) historic snapshot *updated* with the evaluation result of the new tuple that has just been processed. As we will see in Chapter 5, this severely affects the ability of DBMS queries to scale their evaluation latency performance according to the quantity of data stream tuples stored in database. Moreover, this penalty overhead is exacerbated by the time window queries due the expensive self join operations that they have to perform *along the entire* data stream, as depicted by the specification of the self join condition in Figure 5.4.

Workaround Techniques

Some workaround techniques could be considered to enhance the performance and scalability of DBMS queries. For instance, each query could be supported by an auxiliary table storing the output values (the

snapshot) computed in each query evaluation. In this manner, queries would only have to produce results concerning the most recent measurements of each meter—that is, produce the most recent snapshot—and append it to their auxiliary table. The table would behave as an historic snapshot, that would be updated whenever the query is evaluated, and used as input by the another queries. This would avoid the need of compute an expensive historic snapshot every time query is evaluated. Another solution could be the implementation of a *Data Aging Policy* that periodically identifies the maximum range of data that is required by the currently deployed time window queries, and according that deletes the subsets of stored data streams that are no longer necessary—which reduces the amount of data that would be unnecessarily processed by each query (re)evaluation. However, such workarounds are beyond the scope of this work, that aims to assess and compare how suitable is the *query evaluation model* of both DBMS and DSMS engines to efficiently process sensor data streams in a EMS, instead of understanding how a DBMS should be adapted or extended in order to efficiently process data streams.

In the **DSMS Solution**, the windowing behaviour is trivially performed the **Data Window Operators**, provided by EPL for this specific purpose. Those operators, used in the queries `FROM` clause, retain the arriving data stream tuples in a data buffer (i.e. window), dynamically updated according to a given windowing policy (see Section 2.1.6) that defines the data stream sub-part over which aggregate operators will be computed. For instance, see Examples E.13 and E.14, to compare the time window implementation of use-case query Q13 on both solutions. Moreover, several windows can be combined together (chain of windows) in order to achieve complex windowing behaviours [EsperTech, 2014, p.391].

A DSMS query produces its **output as a data stream of tuples**, which greatly differs from the *single* snapshot produced by a DBMS. DSMS queries compute a single data stream tuple for each energy meter measurement that is evaluated, producing an output data stream that will be used as input by the remaining queries, avoiding to produce historic outputs. This happens due the DSMS ability of evaluate its queries in an incremental non-blocking manner, making the output values to be built incrementally through intermediate results. Therefore, the performance scalability of such queries is not affected by the overhead of having to produce historic snapshots, contrary to what happens in DBMSs.

To summarize, in the DBMS queries, time windows must be explicitly implemented with expensive self-join operations that become even more expensive due the need of output historic snapshots. On the other hand, in the DSMS queries, time windows are straightforwardly implemented with a set of window operators specifically provided for this purpose. Moreover, DSMS queries produce their output incrementally as a data stream—a new tuple is outputted each time the query evaluates a meter measurement—, which highly differs from the single historic snapshot produced by DBMS queries, which severely penalizes their performance, as we will see in Section 5.

5.2.4 Incremental Evaluation of Data Queries

The query evaluation process of each query engine is of utmost importance for the implementation and performance of our proposed solution. **DSMS queries evaluation process** is of: one tuple evaluated

at a time, being produced a result for each evaluated tuple. That is, the query evaluation behaviour is **single tuple oriented**, and thus the input data stream is evaluated in a continuous manner, being the results computed incrementally and outputted along the evaluation process. This approach meets the requirements of a continuous query evaluation behaviour identified in Section 4.2.2. On the other hand, **DBMS queries evaluation process** evaluates all dataset "at once", in a batch manner, producing a single output (snapshot) with the evaluation results computed over all dataset. Thus, the query evaluation behaviour is **all dataset oriented**, which defines the One-Time Queries approach of Section 2.1.2

In order to understand the impact of these two query evaluation models in the implementation of use-case queries, consider the following query: *"For each energy meter, return its current measurement and timestamp, together with the respective average of measurements received so far".* Although its simplicity, it holds a not so simple detail: this is a grouped/aggregated query that requires the projection of attributes that are neither aggregated nor grouped, the: `TS` (timestamp) and `Measure` attributes. The query is trivially implemented in the DSMS, the individual evaluation of each tuple (one at a time) makes it easy to solve the previous issue: the values of non-aggregated/grouped attributes (`TS` and `Measure`) that have to be projected are the ones belonging to the—single—tuple under evaluation, and the aggregation value (`AVG(measure)`) that has to be projected is the one that matches the `Device` (energy meter) of the *single* tuple that is being evaluated, as depicted in Figure 5.5 (a). Conversely, in the DBMS such query is somehow cumbersome to implement. Batch evaluation approach makes it impossible to write this query in SQL-92 as it was written in the DSMS EPL's language, since it would lead to the computation of an inconsistent output, as illustrated by Figure 5.5 (b). It depicts why the issue identified above could not be addressed with such query design that would cause a mismatch between the number of rows that are computed for the grouped/aggregated attributes (`Device` and `AVG(Measure)`) and the number of rows computed for the attributes that are neither aggregated nor grouped (`TS` and `Measure`). To overcome this kind of issues SQL-99 introduced the `WINDOW` clause, which like `GROUP BY` allows to specify a set of rows over which we could compute an aggregate operation. Yet, the `WINDOW` clause produces an output row for each evaluated *row* of the input dataset, differing from `GROUP BY` that outputs a single row for each *dataset partition* under evaluation. Figure 5.5 (c) shows how query under consideration could be implemented[8] in the DBMS using this novel operator. However, although the former issue has been solved, the query implementation is quite more complex in the DBMS than in the DSMS [9].

The impact of the this requirement—project attributes that are neither aggregated nor grouped in grouped/aggregated queries—in the ease of solution implementation is exacerbated by the use-case queries Q3 and Q16 (see Table 4.3 and Figure 4.4). As you can see by Sections E.2.3 and E.1.10, the implementation of these queries is *by far* much more complex in a DBMS than in a DSMS. Specifically, query Q3 needs to project the same attribute both aggregated and non-aggregated (Example E.13, lines 10 and 15), and Q16 needs to project two attributes that are neither aggregated nor grouped in a aggregated/grouped query (Example E.19, lines 12 and 13).

[8] By resorting to Self Join operations this query could be written without using the `Window` clause, yet the query implementation would become even more complex.

[9] `Window` clause and `rank()` function were also used with in order to sequentially enumerate the tuples of each stream (to project `Index` attribute), which allows upcoming queries to easily fetch the "head" of their input data streams. Although not strictly necessary, simplifies the implementation of use-case queries in DBMS.

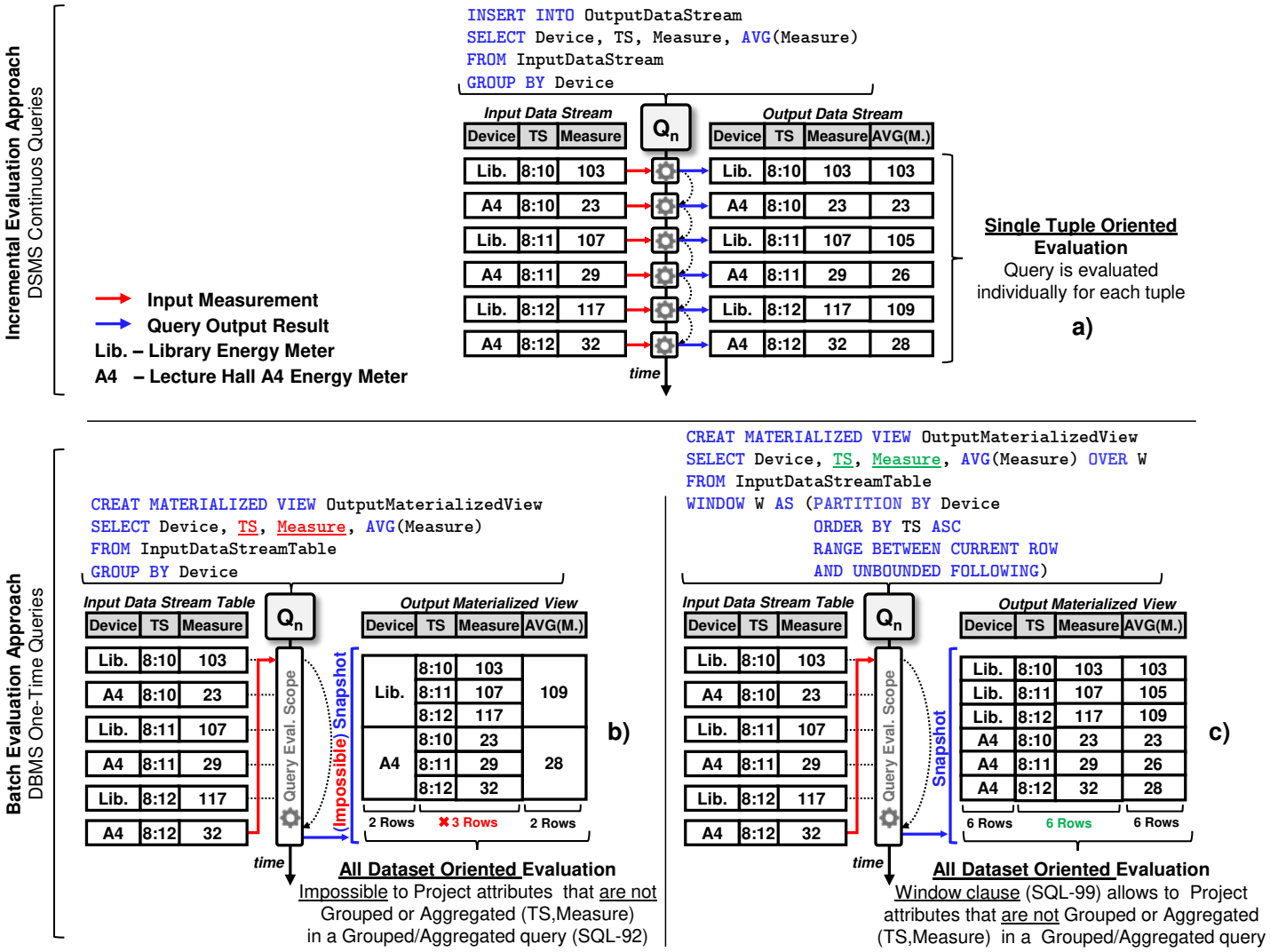

Figure 5.5: Query evaluation model of DSMS and DBMS engines. DSMS Queries are evaluated incrementally, following a tuple oriented approach, where each tupled is evaluated individually by the query (a). For DBMS queries implemented in SQL-92 it is impossible for aggregated/grouped queries to project attributes that are neither aggregated nor grouped (b). If implemented in SQL-99, Window Clause makes it possible for aggregated/grouped queries to project attributes that are neither aggregated nor grouped (c).

A special attention should be given to use-case query Q16, since it's a *very good example* of a query which is very hard to implement in the DBMS, whereas quite straightforward in DSMS, as ilustrated by Figure 5.6. Such query evaluation becomes tremendously expensive in the DBMS, since it computes the entire *data cube*, together with its historical versions, *every time* the query is executed. DSMS's overhead is by far less demanding, the cube is maintained incrementally, being each cell computed on demand. And besides that, the query evaluation overhead is also by far greater in DBMS than in DSMS (see Section 5). This happens due the different *query evaluation model* discussed above that features each query engine. DBMS queries has to specify how *all* dataset tuples should be evaluated at once, whereas DSMS queries just has to specify how the tuple that is *currently* under evaluation should be processed, moreover DBMSs has to output the entire historic of results computed so far, while DSMSs only has to output the most recent computed value.

To conclude, the incremental evaluation approach that features the evaluation process of DSMS

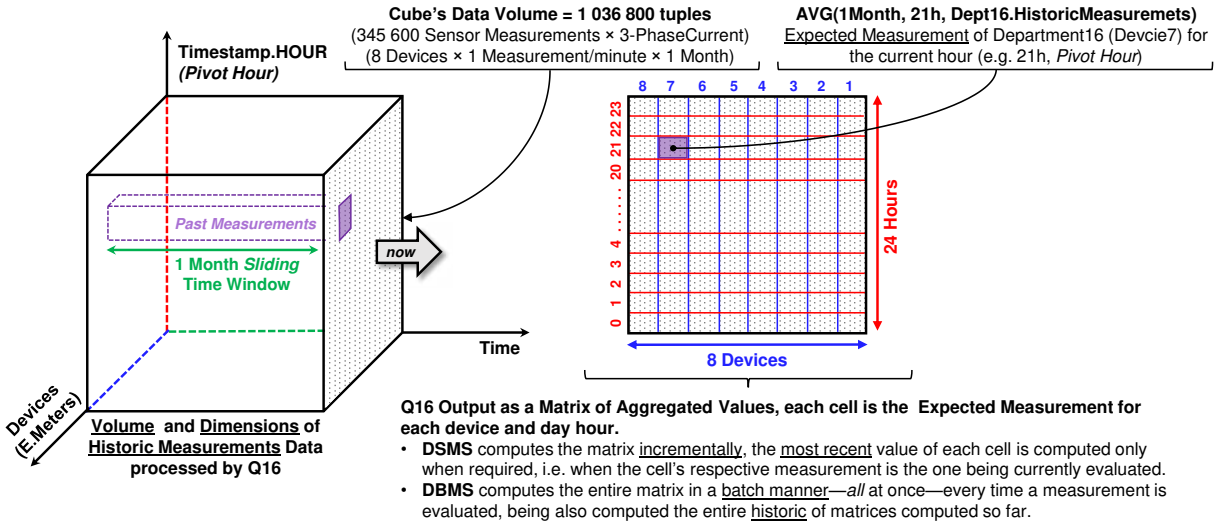

Figure 5.6: Volume and dimensions of the data manipulated by use-case query Q16. Q16 evaluates a sizable amount of data ($\approx 10^6$ tuples) according to 3 dimensions: the *energy meter* of each measurement, the *timestamp's hour* of each measurement, and the *age* of each measurement, .

queries appears the most suitable one to implement and execute our use-case queries on monitoring energy metering networks.

5.2.5 Conclusions and Lessons Learned

The implementation of the two versions of the Data Processing Tier, according to the use-case queries identified in Section 4.2, allows us to achieve the following conclusions:

1. It is easier to implement the use-case queries in the DSMS than in the DBMS. The query language presented by the DSMS is the most suitable one to express the requirements of this domain.

2. The query evaluation model of the DBMS force us to write the queries in a cumbersome manner that leads to performance scalability issues.

The **lesson learned** that comes form these two results is that: the EPL query language provided by the DSMS is more effective and efficient on writing and evaluating queries on the domain of real-time monitoring of energy metering networks, than the SQL query language provided by the DBMS. Therefore, EMSs requiring a continuous evaluation of energy metering data must be supported by a DSMS.

The implementation results that lead to the earlier statement are summarized in Table 5.2. It compares the ease of implementation of use-case queries on both query engines, identifying also the query requirements that hinder the query implementation in the DBMS. More precisely: (i) the fact of time window queries are difficult to implement in the DBMS, and the requirement that they impose to other queries to compute an output containing the entire data stream processed until that point (Historic Snapshot), which increases the overhead of the query evaluation process, and (ii) the requirement of project attributes that are not grouped or aggregated in queries with group or aggregate operators that make these queries cumbersome to implement in the DBMS, together with the fact that the batch evaluation model has to evaluate the entire dataset even when just a sub part of the dataset was modified, which brings great penalty overhead. In DBMS, these problems are transversal to all graph of queries, meaning

Query #	Ease of Query Implementation						Features Hindering Query Implementation in the DBMS	
	DSMS			DBMS			Windowing Query	Grouped/Agg. Query Projecting Not Grouped/Agg. Attributes
	Easy	Hard	Very Hard	Easy	Hard	Very Hard		
Evaluation								
1	●	○	○	●	○	○	○	○
2	●	○	○	●	○	○	○	○
3	●	○	○	○	○	●	●	●
7	●	○	○	○	○	○	○	○
8	●	○	○	○	●	○	●	○
9	●	○	○	●	○	○	○	○
Integration								
4	●	○	○	●	○	○	○	○
5	●	○	○	●	○	○	○	○
6	●	○	○	○	●	○	●	○
10	●	○	○	●	○	○	○	○
11	●	○	○	○	●	○	●	○
12	●	○	○	○	●	○	●	○
13	●	○	○	○	●	○	●	○
14	●	○	○	●	○	○	○	○
15	●	○	○	●	○	○	○	○
16	●	○	○	○	○	●	●	●

Table 5.2: Relative difficulty of implementing the use-case queries on DSMSs and DBMSs. Relative difficulty of each query implementation (left). The features that hinder the implementation in the DBMS Solution (right).

that such problems could not be solved in advance by a set of Integration queries used with the purpose of facilitate the implementation of a a set of Evaluation queries. That is, the DBMS SQL language was not designed to deal in a suitable manner with the requirements of this query domain. While the DSMS EPL language was.

5.3 Performance Evaluation

This section describes the evaluation of our proposed solution to process energy metering data streams in real-time. The purpose is to validate the main claim of this work, that an EMS supported by a DSMS is able to perform better on timely data evaluation than the common state of the art solutions based on a DBMS. More specifically, we intend to discuss the performance results of the benchmark evaluation made between the two prototype versions of the Data Processing Tier. We aim to demonstrate the ability of the DSMS solution to monitor an energy metering network in real-time, and the failure of the DBMS solution on trying to do so.

To validate this statement, the use-case scenarios were executed on both versions of the Data Processing Tier, and the following performance metrics were tracked along each test (see Figure 5.7):

1. **Queue Size and Waiting Time.** The quantity of queued measurements waiting to be evaluated by the scenario and the time each measurement has to wait in the queue to be evaluated.

2. **Latency of Scenario Evaluation.** The time the scenario is taking to evaluate a measurement taken from the queue. That is, the time a measurement is taking to traverse the pipeline of queries composing the scenario, and produce the result set.

3. **Quantity of Processed Tuples.** The amount of tuples that were processed so far by the scenario. Recall that each energy meter measurement is composed by three tuples (the three data-

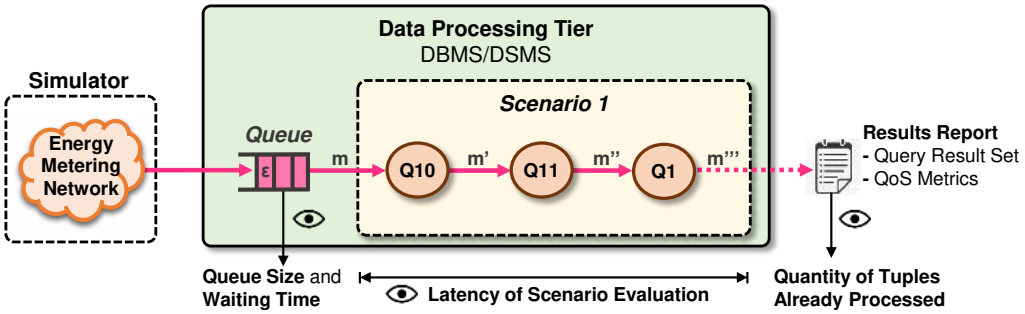

Figure 5.7: Performance Evaluation Metrics. The three evaluation metrics that are tracked along the execution of each test: (i) quantity of queued measurements waiting to be processed and their waiting times, (ii) scenario latency to process a measurement, and (iii) quantity of processed tuples along the test. ε denotes the last queued measurement, the one that has just arrived to the queue, and m denotes the measurement that has just left the queue.

point readings representing the three-phase current), meaning that each processed measurement counts as three processed tuples.

These three metrics tend to be dependent upon each other, meaning that along the experiments the evolution of their values may be correlated, Figure 5.8 shows how this correlation may work. The *scenario evaluation latency* will grow as a function of the *quantity of tuples processed along the test*. At least in the DBMS solution, that has to persist the arriving measurements in order to process them, continuously growing the dataset over which the queries must be executed. Regarding the *queue size*, it will remain around zero as long as the *scenario latency* remains lower than the *average inter arrival period of new measurements*, which we denote by P. However, if this inequality is reversed (i.e. $ScenarioLatency > P$) the amount of measurements waiting in the queue to be processed will start to grow infinitely, since the quantity of measurements that is being received is bigger than the one the system can process per unit of time [Bolch et al., 2006, Chapter 6]. By measuring these three metrics in each solution, we aim to evaluating how the *scenario latency* varies according to the *quantity of tuples that were already processed*, and determine the impact of this in the *size and waiting time of the queue*.

The assessment of these three metrics will enable to verify if the conceived prototype solutions are capable of process data in real-time. Such conclusion must be achieved by interpreting the results in the following manner. We will denote S as the point (of Stress) from which the system becomes unstable, by verifying $ScenarioLatency > P$. After S has been reached, the queue leaves its steady state and its *size* and *waiting time* begin to grow infinitely. At the end of the test, the amount of measurement waiting in the queue is denoted by Q, and the scenario latency by C (see Figure 5.8). Through these values we can conclude two important results:

1. Let $T_{Process(\varepsilon)}$ be the time it takes to process a Measurement ε that has just arrived to the non-empty queue at the end of the test, then $T_{Process(\varepsilon)} \geq C(1+Q)$. That is, the system capability to process energy metering data in real-time is not only related to the *scenario latency* but also to the *amount of measurements that are waiting* in the queue to be processed. Note that we say $\geq$ because we assume that, until ε be evaluated, the *scenario latency* will either continue to grow or, at best, stabilize by achieving its steady state. If the queue is empty when ε arrives, then $T_{Process(\varepsilon)} = C$. Therefore, if $min(T_{Process(\varepsilon)}) = C(1+Q) > Threshold$, being $Threshold$ the least

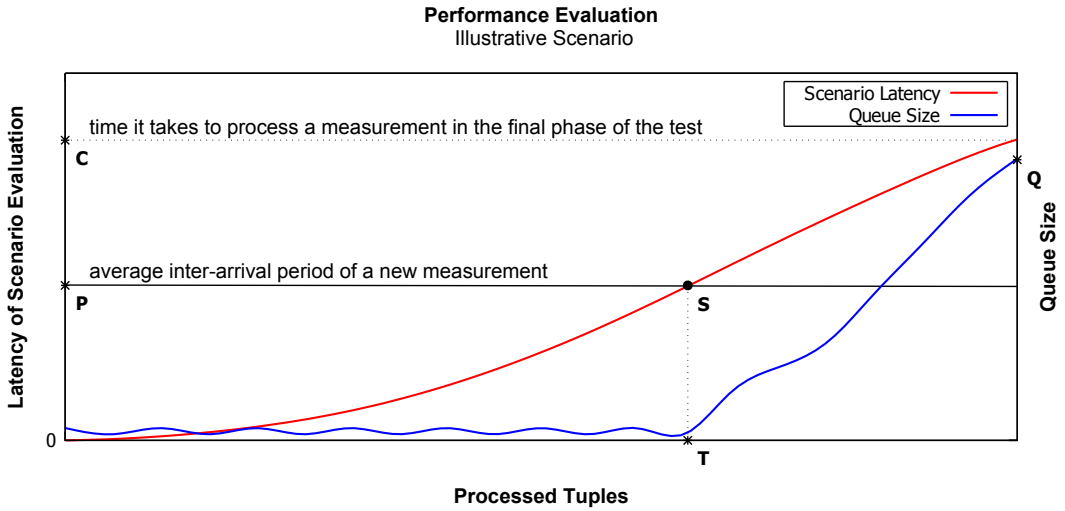

Figure 5.8: Interplay between Performance Evaluation Metrics. Relating the quantity of tuples processed along the test with the scenario latency (left axis) and the queue size (right axis). S identifies the limit conditions ($ScenarioLatency > P$) from which the system starts to be unstable, making the queue leave its stationary state by starting to grow infinitely. Q and C denote, respectively, the amount of queued measurements and the scenario latency at the end of the test.

demanding deadline that a system must met to be said capable of respond in real-time, then we can say that, at the end of the test, and for the scenario under evaluation, the data processing tier is no longer being capable of processing in real-time the energy metering measurements that are arriving. Moreover, this same conclusion can be achieved if, during the test, the average time a measurement has to wait in the queue becomes greater than $Threshold$.

2. After reaching S, system's processing rate became lower than the average arrival rate for new measurements (that is, $ScenarioLatency > P$), which leads to an unlimited growth of the queue. Since that, due memory limitations, a queue cannot grow infinitely, then the system will eventually start to loose data, by discarding the measurements that are still arriving after the maximum size of the queue has been reached—making the system not feasible.

This was the methodology used to analyse the gathered results in order to validate the claim which opened this section, that energy management applications must be supported by a DSMS, instead of a DBMS, so they can process energy metering data data in a timely manner.

5.3.1 Methodology of the Experiments

The evaluation was conducted by running all the **nine use-case scenarios** in both prototypes of the Data Processing Tier, performing a total of **eighteen tests**. Each test was executed individually, being each scenario deployed and evaluated one at a time. The Simulator introduced in Section 4.3.3 was used to produce the energy metering data streams required to feed the system with the data that has been processed by the scenarios. This configuration is illustrated by Figure 5.7.

Each test ran for about **10 hours**, along which the simulator continuously delivered energy metering data produced by the **8 energy meters**, each one with a frequency of **4 measurements per minute**.

Along these 10 hour test, the simulator gradually pushed into the Data Processing Tier a total of **19200 measurements**[10], i.e. **57600 tuples**[11], to be processed by the respective scenario under evaluation. Thus, each new measurement was pushed according to an average period of **1.875 seconds**[12], being this the P value of Figure 5.8. To ensure fairness, the same dataset was used for all eighteen tests.

As pointed out in Section 4.3.3, the simulated data streams are supported by a set of historical measurements taken from the IST Taguspark energy metering network, where each meter is configured with a frequency of one measurement per minute. However, in our experiments we multiply these frequencies by four in order to evaluate the performance of solutions under a situation of increased workload. We prefer to increase the workload by increasing the meters frequency, instead of increase the quantity of meters. The reason for this is that it is more challenging for the query engine to scale through an increased amount of measurements that, by belonging to the same meter, are temporally interdependent and cannot be easily processed in parallel; than to scale trough an increased amount of measurements that, by belonging to different meters, are fairly independent of each other and so easier to process in parallel chunks of data.

We can ensure about the completeness of these experiments, both in its methodology as in the parameterized values, since they were able to show the DBMS's inability to process data in real-time (through the overtaking of S in the majority of the tests), together with the DSMS's ability to successfully achieve this same goal.

5.3.2 Resource Allocation Fairness

For the sake of fairness in the benchmarking process, we had to ensure that both prototypes of the Data Processing Tier were evaluated with the same computational resources, namely: the same amount of memory and CPU capacity. Therefore, a limit of 512MB was defined as the maximum amount of memory available for each solution prototype, being this value the one that maximizes the performance of the DBMS solution [13]. According to the CPU usage, each test was executed in a machine solely dedicated to this purpose, meaning that both prototypes were equally limited by the maximum capacity of the CPU.

In conclusion, this setup allows us to state that: (i) the performance of both systems was measured with a fair resources allocation, (ii) the DBMS results were not compromised by the resources made available, since the system was configured with its most effective amount of memory, and finally (iii) the DSMS results were not dependent on unaffordable amounts of memory.

5.3.3 Experimental Environment

The experiments were conducted in a PC equipped with off-the-shelf hardware. The specifications of the machine are an Intel Core i5-3317U [14] processor running at 2.6GHz (3MB cache), 8GB of RAM (DDR3), and a 500GB HDD (5400rpm, S-ATAII). Running Windows 7 Ultimate 64-bit (kernel version

[10] $19200\,measurements = 10\,hours \times 8\,meters \times 4\,measurements/minute.meter$
[11] $56700\,datapoint\,reading\,tuples = 19200\,measurements \times 3\,datapoint\,reading\,tuples/measurement$
[12] $1.875\,seconds/measurement = (4\,measurements/minute.meter \times 8\,meters)^{-1}$
[13] http://www.postgresql.org/docs/9.3/static/runtime-config-resource.html (see `shared_buffers`)
[14] 2 cores, 4 threads.

6.1.7601.18409) as operating system. The database was placed in the HDD and the Java components of the Data Processing Tier prototypes were executed in the Java HotSpot(TM) Client VM (build 24.51-b03, mixed mode).

5.3.4 Results of the Experiments

This section presents and discusses the results of the eighteen tests that were conducted. The assessment of the performance metrics identified above—*Scenario Latency*, *Queue Size*, and *Processed Tuples*—are depicted by Figures 5.9 and 5.10. They show, for each one of the nine evaluated scenarios, the type of analysis earlier illustrated by Figure 5.8. More specifically, the variation of the *scenario latency* (left axis) and *queue size* (right axis) according to the quantity of *tuples that were processed* [15] along the 10 hours duration of each test (horizontal axis). It is important to analyse these metrics since the total amount of time it takes to process a measurement ε is given by:

$$T_{Process(\varepsilon)} = T_{Queue(\varepsilon)} + T_{Scenario(\varepsilon)} \tag{5.1}$$

Being $T_{Queue(\varepsilon)}$ the time that ε has to wait in the queue to be evaluated, and $T_{Scenario(\varepsilon)}$ the time it takes to evaluate ε according to the respective scenario (that is, the *scenario latency*). Therefore, the Data Processing Tier is able to process energy metering data in real-time if and only if:

$$\forall \varepsilon \in QueuedMeasurements : T_{Queue(\varepsilon)} + T_{Scenario(\varepsilon)} \leq Real\text{-}TimeThreshold \tag{5.2}$$

We denote $Real\text{-}TimeThreshold$ as the least demanding deadline that a system has to meet to be capable of real-time data processing, and it assumes the value of **5 minutes** (according to Section 2.2.1).

As discussed before, the Data Processing Tier becomes unstable if at any point of the experiments we verify the condition $T_{Scenario(\varepsilon)} > 1,875\, secs.$ (that is, the point S of Figure 5.8). In the DBMS solution such state of instability was reached during the evaluation of scenarios 1 and 3–9; while the DSMS solution was able to evaluate all nine scenarios without spoil the system stability (see Figures 5.9 and 5.10). The consequence of such instability for the DBMS solution was its incapacity to process all the 19200 measurements (57600 tuples) produced along each test, within the 10 hours period. Having been left in the queue the measurements that were not timely processed, see Figure 5.11 (left). In contrast, the DSMS solution was capable of process all the data produced by the simulator, since there was no measurements left in the queue at the end of each test, see Figure 5.11 (right). These results tell us that, in the DBMS solution the $T_{Scenario(\varepsilon)}$ at instant t is directly affected by the amount of tuples that were processed until t (i.e. persisted in the database); whereas in the DSMS solution such relationship does not exist. This means that eventually, as the DBMS solution makes progress in the quantity of processed data, the $T_{Scenario(\varepsilon)}$ will increase until become greater than $1,875 secs.$, which will make the queue grow infinitely. Hence the sudden growth of the queue in the evaluation of scenarios 1 and 3–9; by contrast, the queue of the DSMS solution never leaves its steady state, which tends to zero.

[15] The quantity of processed measurements is related with the amount of processed tuples as follows: $ProcessedMeasurements = ProcessedTuples/3$

The ability to manage the size of the queue is a critical issue because of the $T_{Queue(\varepsilon)}$ value, which is essential for a system to respond in a timely manner (see Equation 5.2), that will naturally increase as the queue increases. Therefore, if along an experiment the growth of the queue leads to $T_{Queue(\varepsilon)} > Real\text{-}Time\,Threshold$, then we may conclude that, from this point of the test, the system is no longer capable of process data in real-time. Such results are depicted by Figure 5.12, they show how long each measurement had to wait in the queue to be processed. DBMS solution was not capable of process in real-time all the energy metering data that was produced along the evaluation of scenarios 1 and 3–9, since at a given point of each experiment the condition $T_{Queue(\varepsilon)} > 5\,mins.$ becomes true. On the other hand, the DSMS solution was capable of process all the data in real-time, since the condition $T_{Queue(\varepsilon)} + T_{Scenario(\varepsilon)} \ll 5\,mins.$ was verified along the execution of all nine scenarios.

As discussed at the beginning of this section, we can also infer about the capacity of each solution to process data in real-time by estimating the minimum time it will take to process the last measurement of the test. That is, the $19200th$ simulated measurement, which we denote by ε_{19200}. Such estimative is given by $C(1 + Q)$, being C the *scenario latency* and Q the *size of the queue* at the end of the test, when ε_{19200} arrives to the queue. This estimative assumes that the *scenario latency* will either continue to grow or, at best, stabilize, if a state of plateau has been achieved, which is a reasonable assumption given the results depicted by Figures 5.9 and 5.10. These estimated values are depicted by Figure 5.13, and again they show that the DBMS solution is far from being capable of process energy metering data in real-time for scenarios 1 and 3–9, while the DSMS solution is able to process this same data in a timely manner for all the nine scenarios.

To conclude, the DBMS solution fails to timely process eight of the nine scenarios under evaluation, the performance of the system does not scale with the increased amount of processed data (i.e. stored in database), and therefore the **DBMS solution is not capable of process energy metering data in real-time**. Regarding the DSMS solution, the system successfully processed all the data of the nine scenarios under evaluation, its performance was not affected—at all—by the increased amount of processed data, and therefore the **DSMS solution is capable of process energy metering data in real-time**.

5.3.5 Conclusions and Lessons Learned

The side-by-side benchmark evaluation of the two versions of the Data Processing Tier allows us to conclude the following about the performance of each solution. In the DBMS solution, the query evaluation latency (i.e. scenario latency) does not scale with the quantity of energy metering data already processed; while in the DSMS solution it scales. Such **scalability** relationship between the time it takes to evaluate a query and the quantity of data already processed is a crucial factor for a system that aims to be capable of process streaming data in real-time. In detail, if the query evaluation latency does not scale with the quantity of data already processed, then the time it takes to evaluate the data stream tuples that arrive to the system will eventually become greater than the average arrival period of these tuples—that is, the system will start to receive more data than the one it can process. As a consequence,

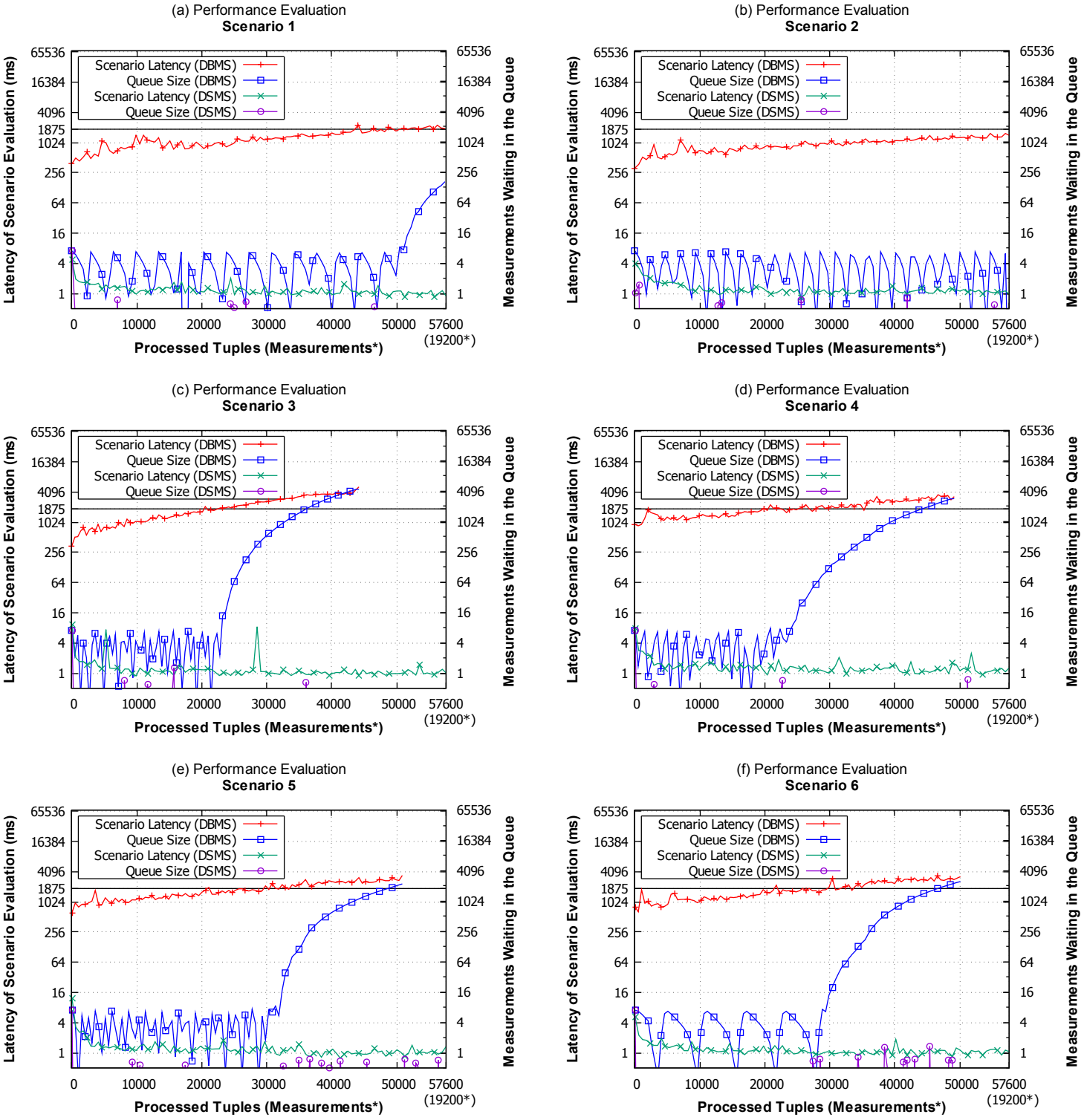

Figure 5.9: Performance evaluation results for use-case scenarios 1–6 (part 1 of 2). Each plot shows, for both solutions, how the scenario latency $T_{Scenario(\varepsilon)}$ (left axis) and queue size (right axis) varies according to the quantity of tuples processed along the test (horizontal axis). 1875ms is the average arrival period of a new measurement to the system, and (*) the quantity of processed measurements is 1/3 of the quantity of processed tuples (i.e. 19200 measurements = 57600 tuples). To improve readability, the plotted values were smoothed with a Spline function.

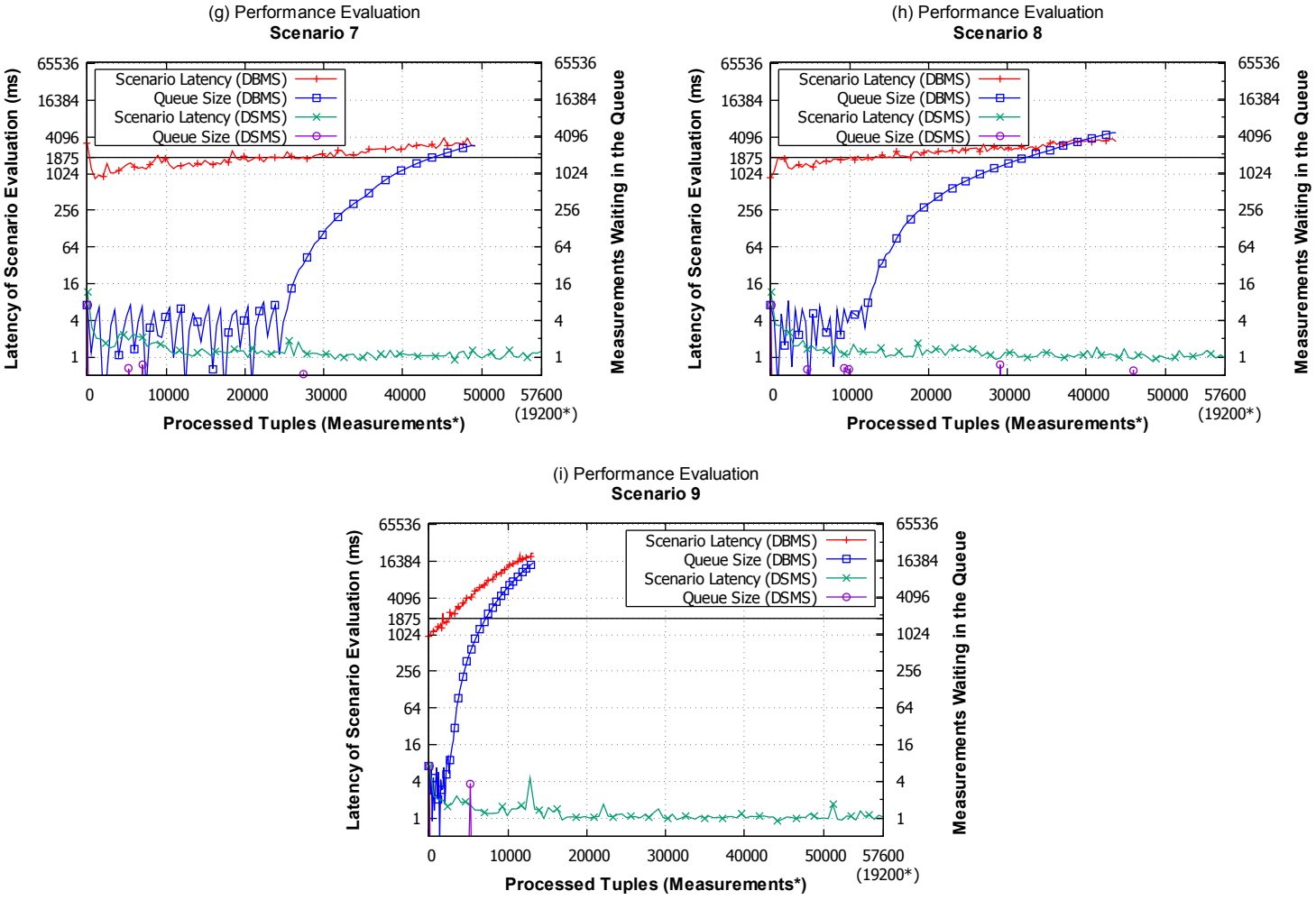

Figure 5.10: Performance evaluation results for use-case scenarios 7–9 (part 2 of 2). Each plot shows, for both solutions, how the scenario latency $T_{Scenario(\varepsilon)}$ (left axis) and queue size (right axis) varies according to the quantity of tuples processed along the test (horizontal axis). 1875ms is the average arrival period of a new measurement to the system, and (*) the quantity of processed measurements is 1/3 of the quantity of processed tuples (i.e. 19200 measurements = 57600 tuples). To improve readability, the plotted values were smoothed with a Spline function.

Figure 5.11: Quantity of processed measurements at the end of each test. Percentage of the test dataset successfully processed (from a total of 19.2k measurements) at the end of each tested scenario (i.e. along 10 hours). In DBMS solution (left), there was a significant quantity of tests were was not possible to process all the measurements within the period of 10 hours, see scenarios 1 and 3–9. In the DSMS solution (right), all the data, of all nine scenarios under evaluation, was successfully processed within the 10 hours period of each test.

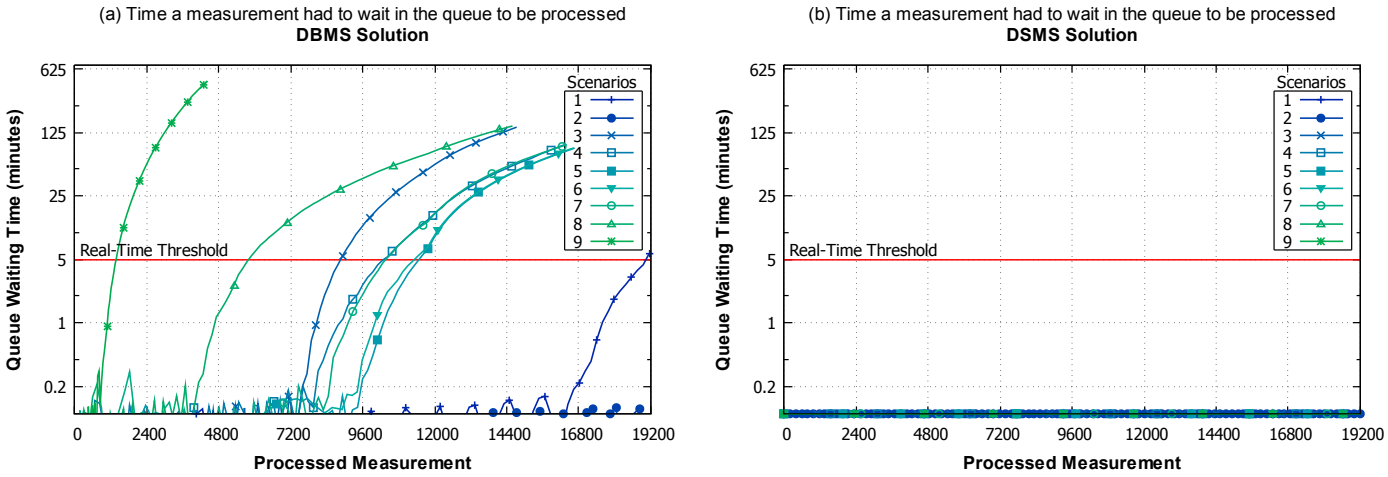

Figure 5.12: Time that a measurement had to wait in the queue to be processed ($T_{Queue(\varepsilon)}$). "Real-Time Threshold" (of 5 minutes) is the least demanding deadline that a system has to met to be responding in real-time. In the DBMS solution (left), it was not possible to maintain the queue waiting time bellow 5 minutes along the evaluation of scenarios 1 and 3–9; while in the DSMS solution (right), all tested scenarios were capable of maintain the queue waiting time always below the threshold.

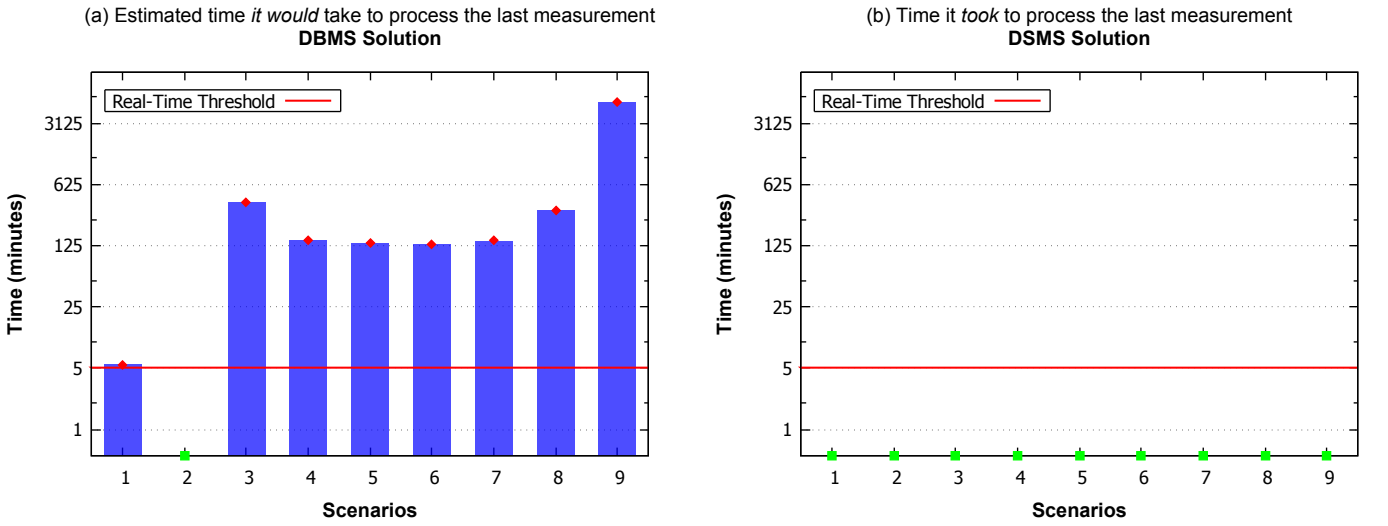

Figure 5.13: Time taken to process the last measurement of the test ($T_{Process(\varepsilon_{19200})}$). "Real-Time Threshold" (of 5 minutes) is the least demanding deadline that a system has to met to be responding in real-time. The estimated time the DBMS solution would take to process the last queued measurement exceeds the threshold for the scenarios 1 and 3–9 (left). The time DSMS solution took to process the last measurement was below the threshold in all tested scenarios (right).

the queue of the system will start to grow endlessly (to accommodate the excess of data) together with the queue waiting time. As a result, the time each energy metering measurement has to wait to be processed will eventually exceed the real-time deadline of the system. This allows us to conclude that:

1. If the query evaluation latency of the system does not scale with the quantity of tuples already processed, then the system will not be capable of process data streams in real-time. Therefore, the incapacity of the DBMS solution to timely process energy metering data.

2. In order to be able to process data streams in real-time, a system must be capable of scale its query evaluation latency with the quantity of tuples already processed. Therefore, the ability of the DSMS solution to timely process energy metering data.

Regarding the Equation 5.2, although the **major performance overhead** becomes visible at the values of $T_{Queue(\varepsilon)}$, the true bottleneck of the DBMS solution lies on the $T_{Scenario(\varepsilon)}$. That is, is the inability of the system to maintain $T_{Scenario(\varepsilon)} \leq P$ [16], along its working time, that breaks its performance. This is worth noting because, at a first glance, we may be tempted to believe that $T_{Scenario(\varepsilon)} \leq Real\text{-}Time\,Threshold$ is the unique condition that the system must verify to work properly, however this is not enough. In the long run, for the system to be capable of process data streams in real-time (i.e. to verify Equation 5.2) we conclude that it must meet the following **stability condition**:

$$\forall \varepsilon \in QueuedMeasurements : T_{Scenario(\varepsilon)} \leq P \tag{5.3}$$

Otherwise, the queue size (and $T_{Queue(\varepsilon)}$) will grow endlessly, breaking the performance of the Data Processing System.

The experiments that were conducted required a Data Processing Tier capable of deliver a minimum **throughput** of 32 measurements/minute [17], in order to make it possible to timely process the arriving energy metering data streams. The DBMS solution failed to timely evaluate eight of the nine tested scenarios, meaning that after 10 hours of working time the system is no longer capable of deliver a throughput of 32 measurements/minute. By assessing the DBMS solution results, we conclude that the scenario features that directly penalized their evaluation performance, were: (i) the quantity and range of time window operations, and the (ii) the quantity of queries (graph nodes) composing the scenario. For instance, compare the performance of scenarios 2 and 9. In the other hand, the DSMS solution is capable of provide a throughput that is by far greater than the minimum one. In fact, taking into account that $T_{Scenario(\varepsilon)} \approx 1ms$ (with maximums of $16ms$) for all evaluated scenarios, a maximum throughput for this solution could be estimated as follows: let the measurements arrive with an average period of $P = 20ms$ (which is cautiously greater than the maximums of $16ms$, required to ensure $T_{Scenario(\varepsilon)} \leq P$), then the system provides a maximum throughput of 3000 measurements/minute [18]. That is, a throughput capable of timely process the data streams produced by 750 meters, each one producing 4 measurements/minute. At the end of a 10 hours test, the quantity of successfully processed measurements

[16] P, denotes the average arrival period of a new measurement to the system (1,875 seconds/measurement in our experiments).
[17] $32\,measurements/minute = (0.03125\,minute/measurement)^{-1} = (1.875\,seconds/measurement)^{-1}$
[18] $3000\,measurements/minute \approx (3.33(3) \times 10^{-4}\,minutes/measurement)^{-1} = (20\,milliseconds/measurement)^{-1}$
In fact, this value can be slightly greater if we are willing to admit a P value lower than $20ms$ (however, still greater than $16ms$).

Data Processing Architecture	Benchmark Dimensions		Verdict
	Query Language	Query Engine Performance	
DSMS supported (Proposed solution)	Easy to implement EMS specific domain queries	Capable of Real-Time Data Stream Processing	DSMS is a suitable solution to support a Real-Time EMS
DBMS supported (State of the art solution)	Hard to implement EMS specific domain queries	Not capable of Real-Time Data Stream Processing	DBMS is not a suitable solution to support a Real-Time EMS

Table 5.3: Summary of the evaluation results. Main conclusions of the benchmark evaluation performed between the two prototypes of the proposed Data Processing Architecture: one supported by a DSMS (our proposed approach) and the other by a DBMS (state of the art approach). Solutions were evaluated according to: its query language suitability and query execution performance.

would be of 1.8×10^6 (that is, 5.4×10^6 DPR tuples). Moreover, the performance of the DSMS solution is nearly the same across all use-case scenarios, not seeming to be affected by the range of time window operations neither by the number of queries composing each scenario.

By comparing the results of the DSMS solution with the ones of the DBMS solution, we are benchmarking the performance of our proposed Data Processing Architecture (that is supported by a DSMS) with the results of a solution that, by being supported by a DBMS, represents the **state of the art**. Therefore, the previous results tell us how the solution we propose to build an EMS capable of process energy metering data in real-time benchmarks with a solution that is based in the state of the art. Such results let us to conclude the following:

1. The proposed Data Processing Architecture, by being supported by a DSMS, achieves a much better performance than the one achieved by the state of the art based solution. Being the proposed Data Processing Architecture capable of process energy metering data streams in real-time.

2. A Data Processing Architecture that, by following the state of the art approach, is supported by a DBMS is not capable of process energy metering data streams in real-time.

The achievements mentioned above lead us to the following **lesson learned**: An EMS that aims to be capable of process energy metering data in real-time must be supported by a DSMS, instead of rely the fulfilment of such requirement to a DBMS. Being the claim of this work validated by this conclusion.

5.4 Final Remarks

Two prototypes of the proposed Data Processing Architecture were implemented. One supported by a DSMS, which represents our proposed solution; and the other supported by a DBMS, which represents a state of the art based solution. In order to compare the performance of both solutions, a benchmark evaluation was conducted between the two prototypes. The evaluated features were: (i) the suitability of the query language to express queries tightly related with the EMS domain, and (ii) the ability of the system to timely evaluate such queries. As a result, we conclude that our proposed solution is superior to the state of the art based solution in both dimensions of the benchmark (see Table 5.3), and therefore the validation of the hypothesis proposed by this work—that, an EMS should be supported by a DSMS in order to be capable of process energy metering data streams in real-time.

Chapter 6

Conclusions

EMSs are used to support the decision making process of energy building managers, helping them to actuate in order to use energy in a more efficient way. To achieve this, those systems monitor energy consumption of buildings to identify potential problems and assess how taken actions affect energy efficiency. Effective problem solving requires early intervention, only possible with an early detection of problems. Typically, a problem takes days or weeks to be detected, reducing this time to hours, or even minutes, would be a major contribution. However, to achieve this EMSs should be able to detect volatile and ephemeral situations, which, in a real scenario, requires the continuous gathering of energy related data, that must be evaluated in a timely manner. Hence the importance of pointing out how to implement an EMS capable of evaluate huge amounts of data in real-time, collected from several buildings or even from large urban areas.

The manner how an EMS interacts with its environment is being dramatically changed by the advent of the Internet of Things (IoT). The cheap and wide available sensor technology, that have emerged from the IoT movement, is making energy metering networks a major and pervasive source of data for energy management applications. The continuous gathering of sensor data leads to datasets that, by being so large and complex, are impossible to process in a useful manner by the traditional data processing systems—which take us to the Big Data challenges known to be imposing a paradigm shift. This topic is requiring a lot of attention from the community, that is struggling to contribute with new types of systems and disruptive techniques capable of to address these new challenges. This work is a step in this direction, by pointing out how an EMS should be adapted to face this emerging paradigm.

Since as we discuss, an EMS supported by a DBMS is not the best solution to timely monitor a network of energy meters, which led us to propose an EMS supported by a DSMS as a more appropriate solution. That is, this work validated the hypothesis that: an EMS based on a DSMS performs better than the state of the art solutions based on a DBMS, by being capable of process energy metering data streams in real-time and by providing a more suitable query language to cope with the requirements of this application domain. To achieve that, we introduced a new EMS's Data Processing Architecture, that has the novelty of being supported by a DSMS, and which we proved to have a superior performance by proceeding as follows. Two prototypes of the proposed architecture were implemented: one supported

by a DSMS, representing our proposed solution; and another supported by a DBMS, representing a state of the art based solution. The performance of both solutions was assessed through a benchmark evaluation, that demonstrated both the implementation feasibility of the proposed architecture, as well its superior performance over to the state of the art based solutions. By superior performance, we mean the ability to: (i) process energy metering data streams in real-time and (ii) provide a most suitable query language to pose queries on the domain of energy monitoring applications.

6.1 Contributions

The main achievements of this work are as follows:

1. **Showing that an EMS must be supported by a DSMS in order to timely process sensor data.** We validate this work hypothesis, that an EMS must be supported by a DSMS, instead of a DBMS, in order to be able to process energy metering data streams in real-time. Since this query engine provides a superior performance and a more suitable query language for this domain (see Sections 5.2 and 5.3).

2. **Introduce an EMS's Real-Time Data Processing Architecture.** We propose an architecture for implementing an EMS capable of monitor a network of energy meters in real-time. We also provide a functional prototype [1] of the proposed architecture, in order to validate its implementation feasibility, together with an evaluation of its performance results (see Section 4.1).

3. **Perform a comprehensive study of why a DBMS fails to timely process sensor data.** We contribute with a detailed explanation of the query language properties and performance bottlenecks that are responsible for making a DBMS an ineffective solution to timely monitor energy metering data streams (see Sections 5.2 and 5.3).

4. **A classification of the type of queries used to monitor sensor networks.** We propose a class of queries that summarizes the features and the requirements of the queries used to monitor sensor networks. That is, a requirement analysis essential to understand the type of data transformations that a monitoring data processing system must efficiently support (see Section 4.2).

5. **An energy metering network simulator.** We described the implementation of an energy metering network simulator [2] that, by producing deterministic datasets with an adjustable workload, streamlines the testing and validation phases of an energy management application along its development process (see Section 4.3.3).

The achievements of this work were well received by the community, **our results were published** as a relevant contribution in the proceedings of *"2014 IEEE International Congress on Big Data, Anchorage, Alaska (July 2014)"* [3], as a full paper entitled *"Real-Time Integration of Building Energy Data"* [4], which

[1] https://github.com/diogo-gsa/data-processing-architecture
[2] https://github.com/diogo-gsa/building-energy-meters-network-simulator
[3] http://dblp.uni-trier.de/db/hy/conf/bigdata/bigdata2014
[4] http://dx.doi.org/10.1109/BigData.Congress.2014.44

introduces and discuss our approach for a Real-Time EMS's Data Processing Architecture. We also **contributed with a second paper** to the *"32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland (May 2016)"* [5], entitled *"Real-Time Monitoring of Building Energy Metering Networks"*, which discuss how the proposed architecture benchmarks with the state of the art based solutions. The publication is following the peer review process.

6.2 Future Work

Although the main purpose of this work has been reached, along its execution we identified a set of topics that in the future are worthy of further consideration, namely:

- **Orchestrate the cooperation between DSMS and DBMS.** Our work claims that, in order to obtain timely results from sensor data, an EMS must be supported by a DSMS, and not by a DBMS. Yet, this does not mean that a DBMS must be discarded from the scope of an EMS's Data Processing Architecture. In fact, they are required to store metadata, historical data (e.g. samples of sensor data), and derived and aggregated data, that despite being computed by the DSMS, has to be stored in the DBMS, to be available when required. Therefore, it would be a great contribution to explore the opportunities that arise from the cooperation between a DBMS and a DSMS, and to understand how they could be orchestrated in order to take the most of batch and stream data processing. Note that, we already make some progress in this direction, in this work the DSMS cooperates with the DBMS by retrieving some metadata from it, still we believe that we can go further by also putting the DSMS writing their computed results in the DBMS. Actually, this concept is not new, the Lambda Architecture proposed by Marz and Warren [2015] points out how to combine batch and stream data processing in a single solution, in order to efficiently process massive quantities of data. Yet, this is a general purpose architecture, finely shape it to fit in the domain of energy management applications would be a great contribution.

- **Heterogeneous Data Sources and Faulty Sensor Data.** Assess the challenges of deal with more than one type of data sources simultaneously. Putting together different types of sensor data, likewise energy metering, environmental and equipment data, will raise several issues at the data integration level. Such as, conflicting data structures, inconsistent semantics, structured vs. unstructured data, as well faulty data produced by faulty equipment. The need to prevent these issues are already foreseen by the data processing phases of the proposed architecture. However, a set of continuous queries capable of solve these issues in a *systematic* manner must be developed, otherwise it will be difficult to timely ensure the quality of the produced results.

- **Mining Sensor Data and Pattern Detection.** In our work, the data operations performed by the DSMS's continuous queries were mainly related with the evaluation of data aggregates over time windows. The assessment of the DSMS's CEP (complex event processing) capabilities to perform complex data analytics over sensor data, such as the identification of patterns and other causality

[5] `http://icde2016.fi`

relationships, have been left out. Understand the issues which arise from mining energy metering data in real-time, as well the kind of data analytics that must be performed, would be a great contribution to better understand how energy management applications could benefit from such mining techniques.

This list is not intended to be an exhaustive enumeration of all the topics that must be addressed in order to properly develop a Real-Time Data Analytics EMS, in fact, there is a lot more to consider. Instead, they are a set of issues that have emerged along the execution of this research, and for which we already have some ideas to work on.

Bibliography

D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal—The International Journal on Very Large Data Bases*, 12(2):120–139, 2003.

N. Akhtar and F. H. Siddiqui. UDP packet monitoring with stanford data stream manager. In *2011 International Conference on Recent Trends in Information Technology (ICRTIT)*, pages 533–537. Ieee, June 2011. ISBN 978-1-4577-0588-5. doi: 10.1109/ICRTIT.2011.5972403.

D. Anjos, P. Carreira, and A. P. Francisco. Real-time integration of building energy data. In *Big Data (BigData Congress), 2014 IEEE International Congress on*, pages 250–257. IEEE, 2014. doi: 10.1109/BigData.Congress.2014.44.

S. F. Apache. Apache storm documentation (apache software foundation). `http://storm.apache.org/documentation.html`, 2014.

A. Arasu, S. Babu, and J. Widom. An abstract semantics and concrete language for continuous queries over streams and relations. 2002.

A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. Stream: The stanford data stream management system. *Stanford InfoLab*, 2004.

A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, July 2005. doi: 10.1007/s00778-004-0147-z.

W. G. Aref, A. K. Elmagarmid, M. H. Ali, A. C. Catlin, M. G. Elfeky, M. Eltabak, T. Ghanem, M. A. Hammad, I. F. Ilyas, M. Lu, M. Marzouk, M. F. Mokbel, X. Xiong, and W. Lafayette. Nile : A Query Processing Engine for Data Streams. In *Proceedings of the 20th International Conference on Data Engineering, ICDE*, page 851, 2004.

B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16. ACM, 2002.

S. Babu and J. Widom. Continuous queries over data streams. *ACM Sigmod Record*, 30(3):109–120, 2001.

R. S. Barga, J. Goldstein, M. H. Ali, and M. Hong. Consistent streaming through time: A vision for event stream processing. In *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*, pages 363–374, 2007.

T. Bass. Mythbusters : Event Stream Processing Versus Complex Event Processing. In *Proceedings of the 2007 Inaugural International Conference on Distributed Event-based Systems*, pages 1–1. ACM, 2007. ISBN 9781595936653.

M. A. Beyer and D. Laney. The importance of 'big data': a definition. *Stamford, CT: Gartner*, 2012.

C. Bizer and A. Schultz. The berlin sparql benchmark. 2009.

G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi. *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. John Wiley & Sons, 2006.

P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. In *Mobile Data Management*, pages 3–14. Springer, 2001.

R. M. Bruckner, B. List, and J. Schiefer. Striving towards Near Real-Time Data Integration for Data Warehouses. In *DaWak*, pages 317–326, 2002.

H.-l. Bui. Survey and Comparison of Event Query Languages Using Practical Examples. 2009.

T. Cardoso. *A Framework towards Efficient Integration of Energy Data*. Instituto Superior Técnico, 2013.

U. Cetintemel, D. Abadi, Y. Ahmad, H. Balakrishnan, M. Balazinska, M. Cherniack, J. Hwang, W. Lindner, S. Madden, A. Maskey, et al. The aurora and borealis stream processing engines. *Data Stream Management: Processing High-Speed Data Streams, Springer-Verlag*, pages 1–23, 2006.

S. Chakravarthy and Q. Jiang. *Stream Data Processing: A Quality of Service Perspective Modeling, Scheduling, Load Shedding, and Complex Event Processing*. Springer Publishing Company, Incorporated, 1st edition, 2009. ISBN 0387710027, 9780387710020.

B. Chandramouli, M. Ali, J. Goldstein, B. Sezgin, and B. S. Raman. Data stream management systems for computational finance. *Computer*, (12):45–52, 2010.

S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. Telegraphcq: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 668–668. ACM, 2003.

J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In *ACM SIGMOD Record*, volume 29, pages 379–390. ACM, 2000.

M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. B. Zdonik. Scalable distributed stream processing. In *CIDR*, volume 3, pages 257–268, 2003.

D. Chwieduk. Towards sustainable-energy buildings. *Applied Energy*, 76(1-3):211–217, Sept. 2003. doi: 10.1016/S0306-2619(03)00059-X.

E. Commission. Proposal for a directive of the european parliament and of the council on energy efficiency and repealing directives 2004/8/ec and 2006/32/ec., 2011.

E. Commission. The european union explained: Sustainable, secure and affordable energy for europeans., 2012.

L. Copin, H. Rey, X. Vasques, A. Laurent, and M. Teisseire. Intelligent Energy Data Warehouse: What Challenges? *2010 22nd IEEE International Conference on Tools with Artificial Intelligence*, pages 337–342, Oct. 2010. doi: 10.1109/ICTAI.2010.120.

C. Cranor, T. Johnson, and O. Spataschek. Gigascope : A Stream Database for Network Applications. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 647–651. ACM, 2003. ISBN 158113634X.

G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, 44(3):15, 2012.

A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. M. White. Cayuga: A general purpose event monitoring system. In *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January, 2007, Online Proceedings*, volume 7, pages 412–422, 2007.

Y. Diao, N. Immerman, and D. Gyllstrom. Sase+: An agile language for kleene closure over event streams, 2007.

W. Eckerson. *Performance Dashboards Measuring, Monitoring, and Managing your Business*. John Wiley & Sons, Inc, second edi edition, 2010. ISBN 9780470589830.

Enerwise. Enerwise — Energy Manager 3.0. `http://www.enerwise.com/energymanager.php`. (Accessed: November 2014).

EsperTech. *Esper Reference, Version 5.0.0*. EsperTech Inc., 2014.

P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, 2003.

Eurostat. Eu27 energy dependence rate at 54% in 2011. Technical report, Eurostat, 2013.

L. Fulop, G. Toth, R. Racz, J. Panczel, T. Gergely, and A. Beszedes. Survey on Complex Event Processing and Predictive Analytics. Technical report, University of Szeged, Department of Software Engineering, 2010.

J. Gama and P. P. Rodrigues. Data stream processing. In *Learning from Data Streams*, pages 25–39. Springer, 2007.

H. Garcia-Molina and K. Salem. Main memory database systems: an overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, 1992. doi: 10.1109/69.180602.

L. Golab and M. T. Özsu. Issues in data stream management. *ACM Sigmod Record*, 32(2):5–14, 2003.

J. Granderson, M. Piette, G. Ghatikar, and P. Price. *Building Energy Information Systems: State of the Technology and User Case Studies*. Number November. Lawrence Berkeley National Laboratory, LBNL-2899E., 2009.

J. Granderson, M. Piette, B. Rosenblum, and et al. L. Hu. *Energy Information Handbook: Applications for Energy-Efficient Building Operations*. Lawrence Berkeley National Laboratory, LBNL-5272E., 2011.

J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7):1645–1660, 2013.

V. Gulisano. *StreamCloud: An Elastic Parallel-Distributed Stream Processing Engine*. PhD thesis, Universidad Politécnica de Madrid, 2012.

D. Gyllstrom, Y. Diao, E. Wu, P. Stahlberg, G. Anderson, and H.-J. Chae. SASE : Complex Event Processing over Streams. *CIDR Conference*, 2007.

I. Interval Data Systems. Energy Witness. `http://www.intdatsys.com/app-data.htm`. (Accessed: November 2014).

X. Jiang, S. Yoo, and J. Choi. Dsms in ubiquitous-healthcare: A borealis-based heart rate variability monitor. In *Biomedical Engineering and Informatics (BMEI), 2011 4th International Conference on*, volume 4, pages 2144–2147. IEEE, 2011.

A. Karakasidis, P. Vassiliadis, and E. Pitoura. ETL queues for active data warehousing. *Proceedings of the 2nd international workshop on Information quality in information systems - IQIS '05*, page 28, 2005. doi: 10.1145/1077501.1077509.

A. H. Kazmi, M. J. O'grady, D. T. Delaney, A. G. Ruzzelli, and G. M. P. O'hare. A review of wireless-sensor-network-enabled building energy management systems. *ACM Trans. Sen. Netw.*, 10(4):66:1–66:43, 2014. doi: 10.1145/2532644.

N. L. Lawrence Berkeley. OpenEIS — Energy Information System. `http://eis.lbl.gov/openeis.html`. (Accessed: November 2014).

X. Li, B. Plale, N. Vijayakumar, R. Ramachandran, S. Graves, and H. Conover. Real-time storm detection and weather forecast activation through data mining and events processing. *Earth Science Informatics*, 1(2):49–57, 2008.

X. Ma, R. Cui, Y. Sun, C. Peng, and Z. Wu. Supervisory and Energy Management System of large public buildings. *2010 IEEE International Conference on Mechatronics and Automation*, pages 928–933, Aug. 2010. doi: 10.1109/ICMA.2010.5589969.

S. Madden and M. J. Franklin. Fjording the Stream : An Architecture for Queries over Streaming Sensor Data. In *Data Engineering, 2002. Proceedings. 18th IEEE International Conference on*, pages 555—-566, 2002.

N. Marz and J. Warren. *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2015. ISBN 1617290343, 9781617290343.

McKinstry. Enterprise Energy Management Suite (EEM SuiteTM). `http://www.mckinstryeem.com/EEM%20Suite%20Overview.pdf`. (Accessed: November 2014).

N. Motegi, A. Piette, S. Kinney, and K. Herter. Introduction to web-based energy information systems for energy management and demand response in commercial buildings. *Information technology for energy managers*, pages 55–74, 2004.

A. Mukherjee, P. Diwan, P. Bhattacharjee, D. Mukherjee, and P. Misra. Capital market surveillance using stream processing. In *Computer Technology and Development (ICCTD), 2010 2nd International Conference on*, pages 577–582. IEEE, 2010.

L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 170–177. IEEE, 2010.

M. Nguyen and A. Min. Zero-Latency Dara Warehousing for Heterogeneous Data Sources and Continuous Data Streams.

N. W. Paton and O. Díaz. Active database systems. *ACM Computing Surveys (CSUR)*, 31(1):63–103, 1999.

L. Pérez-Lombard, J. Ortiz, and C. Pout. A review on buildings energy consumption information. *Energy and Buildings*, 40(3):394–398, Jan. 2008. doi: 10.1016/j.enbuild.2007.03.007.

S. Sathe, T. G. Papaioannou, H. Jeung, and K. Aberer. A survey of model-based sensor data acquisition and management. In *Managing and Mining Sensor Data*, pages 9–50. Springer, 2013.

R. J. Stewart, P. W. Trinder, and H.-w. Loidl. Comparing High Level MapReduce Query Languages. (Section 6):58–72, 2011.

M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, 34(4):42–47, 2005.

P. Vassiliadis. A Survey of Extract – Transform – Load Technology. *International Journal of Data Warehousing & Mining*, 5(September):1–27, 2009.

P. Vassiliadis and A. Simitsis. Near Real Time ETL. *Springer journal Annals of Information Systems*, 3: 1–38, 2008.

H. Wang and C. Zaniolo. Atlas: A native extension of sql for data mining. In *SDM*, pages 130–141. SIAM, 2003.

E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 407–418. ACM, 2006.

Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. *ACM Sigmod Record*, 31(3):9–18, 2002.

M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, page 10, 2010.

M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: A fault-tolerant model for scalable stream processing. Technical report, DTIC Document, 2012.

Q. Zhang, C. Pang, S. Mcbride, D. Hansen, C. Cheung, and M. Steyn. Towards health data stream analytics. In *Complex Medical Engineering (CME), 2010 IEEE/ICME International Conference on*, pages 282–287. IEEE, 2010.

Appendix A

Survey on Sensor Networks Monitoring Queries

#	Query Statement (case study)	Main Operators†								Main Features†						Classes of Queries			
		Selection	Ext. Proj.	Agg.Ops	Group By	Order By	Having	Join	Time Win.	UDF	Data Stream as Input	Database as Input	Output Ctrl.	Pattern Detection		Detection of Abnormal Values	Data Stream Instant Summary Metrics	Data Stream Metadata Evaluation	Database Integration
	Towards Sensor Database Systems [Bonnet et al., 2001]																		
1	Return abnormal temperatures repeatedly measured by all sensors	●	●	○	○	○	○	○	○	●	●	●	○	○		●	○	○	○
2	Every minute, return the temperature measured on 3rd floor	●	○	○	○	○	○	○	●	○	●	○	●	○		○	●	○	○
3	Generate a notification whenever two sensors within 5 yards of each other simultaneously measure an abnormal temperature	●	○	○	●	○	○	●	○	●	●	●	○	○		●	○	○	○
4	Every five minutes retrieve the maximum temperature measured over the last 5 minutes	○	●	●	●	○	○	○	●	○	●	○	●	○		○	●	○	○
5	Return the average temperature measured on each floor over the last 10 minutes	○	●	●	●	○	○	○	●	○	●	○	○	○		○	●	○	○
	Capital Market Surveillance using Stream Processing [Mukherjee et al., 2010]																		
6	By each sensor, identify large time periods between each data stream tuple	○	●	●	●	○	○	○	●	○	●	○	○	○		●	●	●	○
7	Identify large deviation in average trade price and quantity between today and yesterday quotations	○	●	●	●	○	●		●	○	●	●	○	○		○	●	○	●
8	Identify large deviation in trade and price by comparation with normal values Normal values can be derived statistacly by past readings	●	○	○	○	○	○	●	○	○	●	●	○	○		○	●	○	●
9	High-low price variation when compared with close price yeasterday's	●	○	○	○	○	○	○	●	○	●	○	○	○		●	●	○	●
10	Identify price variations above a given delta between yestarday's close value and currently value	●	●	○	○	○	○	○	●	○	●	○	○	○		●	●	○	●
11	Identify Consecutive trade price variation above a given delta	●	○	○	○	○	○	○	●	○	●	○	○	●		●	○	○	○
	Data Stream Management Systems for Computational Finance [Chandramouli et al., 2010]																		
12	Select all stock quotes for the ticker symbol "MSFT", removing the unnecessary ticker symbol information in the output	●	●	○	○	○	○	○	○	○	●	○	○	○		○	●	●	○
13	Take the output signal produced by FilteredStream, and smooth it by reporting a 3 second trailing average every second	●	●	●	○	○	○	○	●	○	●	○	●	○		○	●	○	○
14	Detect the head & shoulders chart pattern over the stock stream for MSFT, over a ten-minute window	○	○	○	○	○	○	○	●	○	●	○	○	●		○	●	○	○
15	For each unique ticket symbol, detect the occurrence of the head & shoulders pattern of Q3	○	○	●	●	○	○	○	○	○	●	○	○	●		○	●	○	○
16	For each stock on a dynamic "white list", determine all occurrences of the head & shoulders pattern	●	○	○	○	○	○	●	●	○	●	●	○	●		○	●	○	●
17	For each stock not present in a dynamic "black list", determine all occurrences of the head & shoulders chart pattern	●	○	○	○	○	○	●	●	○	●	●	○	●		○	●	○	●
	The Berlin SPARQL Benchmark [Bizer et al., 2009]‡																		
18	Find products for a given set of generic features	●	●	○	○	●	○	○	○	○	○	●	○	○		○	○	○	○
19	Retrieve basic information about a specific product for display purposes	●	○	○	○	○	○	○	○	○	○	●	○	○		○	○	○	○
20	Find products having some specific features and not having one feature	●	●	○	○	●	○	○	○	○	○	●	○	○		○	○	○	○
21	Find products matching two different sets of features	●	○	○	○	●	○	○	○	○	○	●	○	○		○	○	○	○
22	Find products that are similar to a given product	●	●	○	○	●	○	○	○	○	○	●	○	○		○	○	○	○
23	Find products having a label that contains a specific string	●	○	○	○	○	○	○	○	○	○	●	○	○		○	○	○	○
24	Retrieve in-depth information about a product including offers and reviews	●	○	○	○	○	○	○	○	○	○	●	○	○		○	○	○	○
25	Give me recent English language reviews for a specific product	●	○	○	○	●	○	○	○	○	○	●	○	○		○	○	○	○
26	Get information about a reviewer	●	○	○	○	○	○	○	○	○	○	●	○	○		○	○	○	○
27	Get cheap offers which fulfill the consumer's delivery requirements	●	●	○	○	●	○	○	○	○	○	●	○	○		○	○	○	○
28	Get all information about an offer	●	●	○	○	○	○	○	○	○	○	●	○	○		○	○	○	○
29	Export information about an offer into another schema	●	○	○	○	○	○	○	○	○	○	●	○	○		○	○	○	○
	Towards Health Data Stream Analytics [Zhang et al., 2010]																		
30	Send me an alarm whenever HR goes up and BP goes down during past 15 minutes	●	○	○	○	○	○	●	●	○	●	○	○	●		●	●	○	○
31	Find all operations with patients experiencing oxygen desaturation and rising of ETC02	●	○	○	○	○	○	●	●	○	●	○	○	●		○	●	○	○
	UDP Packet Monitoring with Stanford Data Stream Manager [Akhtar and Siddiqui, 2011]																		
32	Evaluate the total packet flow in the network for each varying packet speed	○	●	●	●	○	○	○	●	○	●	○	○	○		○	●	●	○
33	Display the network usage by specific IP address	●	●	●	○	○	○	○	●	○	●	○	○	○		○	●	○	○
34	Evaluate network usage service from DNS request ports in the traffic (identify wich services/website consume more network)	○	●	●	●	○	○	○	●	○	●	○	○	○		○	●	○	○
35	Evaluate network usage by 102.66.17.202 over an average period of 10 sec.	●	●	●	●	○	○	○	●	○	●	○	○	○		○	●	○	○
	Gigascope: A Stream Database for Network Applications [Cranor et al., 2003]																		
36	Identify fraction of trafic in the backbone B which can be attributed to a customer network C	●	●	●	○	○	○	●	●	○	●	○	○	○		○	●	○	○
37	Reports the destination IP and port, and a timestamp from TCP packets with IPVersion = 4 and Protocol = 6	●	●	○	○	○	○	○	●	○	●	○	○	○		○	●	○	○
38	Merge the two streams tcpDest1 which matches tcpDest0 except that it reads from Interface eth1	●	○	○	○	○	○	●	●	○	●	○	○	○		○	●	○	○
	Models and Issues in Data Stream Systems [Babcock et al., 2002]																		
39	Computes load on the link B averaged over one-minute intervals, and output notification when the load crosses a specified threshold	●	○	●	●	○	●	○	●	○	●	○	○	○		●	●	○	○
40	Isolates flows in the backbone link and determines the amount of traffic generated by each flow	○	●	●	●	○	○	○	●	○	●	○	○	○		○	●	●	○
41	During periods of congestion determine which network's customer is the likely cause	○	○	●	●	○	●	○	●	○	●	○	○	○		○	●	○	○
42	Monitoring the source-destination pairs in the top 5 percent in terms of backbone traffic	●	○	●	●	●	○	○	●	○	●	○	○	○		●	●	○	○
	Fjording the Stream : An Architecture for Queries over Streaming Sensor Data [Madden and Franklin, 2002]																		
43	Identify the average speed over segments of the road using a time window n	●	○	●	●	○	○	○	●	○	●	○	○	○		○	●	●	○
44	Identify slow segments in wich the driver are interested in	●	○	●	●	○	●	○	●	○	●	○	○	○		○	●	●	○

Table A.1: Survey on Sensor Network Monitoring Queries. Literature survey on the type of queries used to monitor sensor networks, according to several application domains. Each query was: (i) discriminated by its main operators and features, and (ii) classified according a set of classes proposed by us. ●: required. ○: not required.
(†) In most cases, publications does not discriminate this properties, neither the underling data schemas. Therefore, some properties have to be fulfilled by us based on ours domains' knowledge.
(‡) This publication is not related with data stream processing or even sensors network monitoring, but instead with the benchmarking of two relational query engines. Thus, given the benchmark nature of our work, the reviewing of its methodology (and case study) was of great importance to our work, and therefore its presence of this survey.

Appendix B

Simulator API of IST Taguspark Energy Meters Network

```java
package msc_thesis.diogo_anjos.simulator;

/*
 * Simulator of the Energy Meters Network deployed at IST Taguspark campus.
 * @author Diogo Anjos (diogo.silva.anjos@tecnico.ulisboa.pt)
 */

public interface Simulator {

 /*
  * Configures the simulation job by specifying the Energy Meter and period of time
  * to be simulated. This method must be executed before any other simulator method.
  */
 public void initialSetup(EnergyMeter em, String startTimestmap, String stopTimestmap);

 /*
  * Register a new simulator's client. Those clients will be notified
  * (push mechanism) whenever a new measure event is produced by the simulator.
  */
 public void registerNewClient(SimulatorClient client);

 /*
  * Start the simulation process, the production of energy event measures.
  * This method has to be used at least one time during the simulation life cycle process
  */
 public void start();

 /*
  * Stop the simulation process. Stop the production of new event measures.
  * start() method should be used to resume the simulation process, the database iterator
  * index will continue from the last produced event measure.
  * stop() can not be used without the previous usage of start()
  */
 public void stop();

 /*
  * Specify the time scale that should be used between each produced even
  * measurement along simulation.
  * Default value is 1, meaning that  the time interval between event measures
  * will not be shrunk. For instance, for an interval of 60 sec. (60k ms) between
  * tuple T1 and T2, the simulator will produce T1 and wait 6000 ms until produce
  * T2 (1:1 time-scale). setSpeedTimeFactor(int newFactor) readjusts the
  * time-scale as follows: 1:newFactor, i.e. a 60 seconds simulation period with
  * a time factor of 2, will be complete within 30 seconds. The simulator
  * internal unit time is the millisecond (ms). For instance:
  * Time factor = 24 : 24 hour time-series simulation  will take 1 hour to complete.
  *             = 720: 1 mounth (720h) time-series simulation will take 1 hour.
  */
 public boolean setSpeedTimeFactor(int newFactor);

 /*
  * Get the current scale time factor (time divison factor).
  */
 public int getSpeedTimeFactor();

 /*
  * Release all gathered resources during the object construction.
  * In this case, releases the setlled database connection.
  */
 public void destroy();

 /*
  * An overrided toString() to dump the table (device) and time period
  * simulated by this simulator instance.
  */
  public String toString();
}
```

Figure B.1: Simulator API of IST Taguspark Energy Metering Network. The Simulator API (Java based), that was developed under the scope of this work, to mimic the Energy Meters Network deployed at IST Taguspark campus.

Appendix C

Database Schema of IST Taguspark EMS

This data schema aims to model the domain that is related with the building energy metering network, and other energy relevant sensors, that is deployed at IST Taguspark campus. The database schema was developed within the research project **Smart Campus** [1], being used in production environment to support both the EMS and other energy management experiments that are being developed under the scope of this same project.

[1] http://greensmartcampus.eu/smart-campus-project

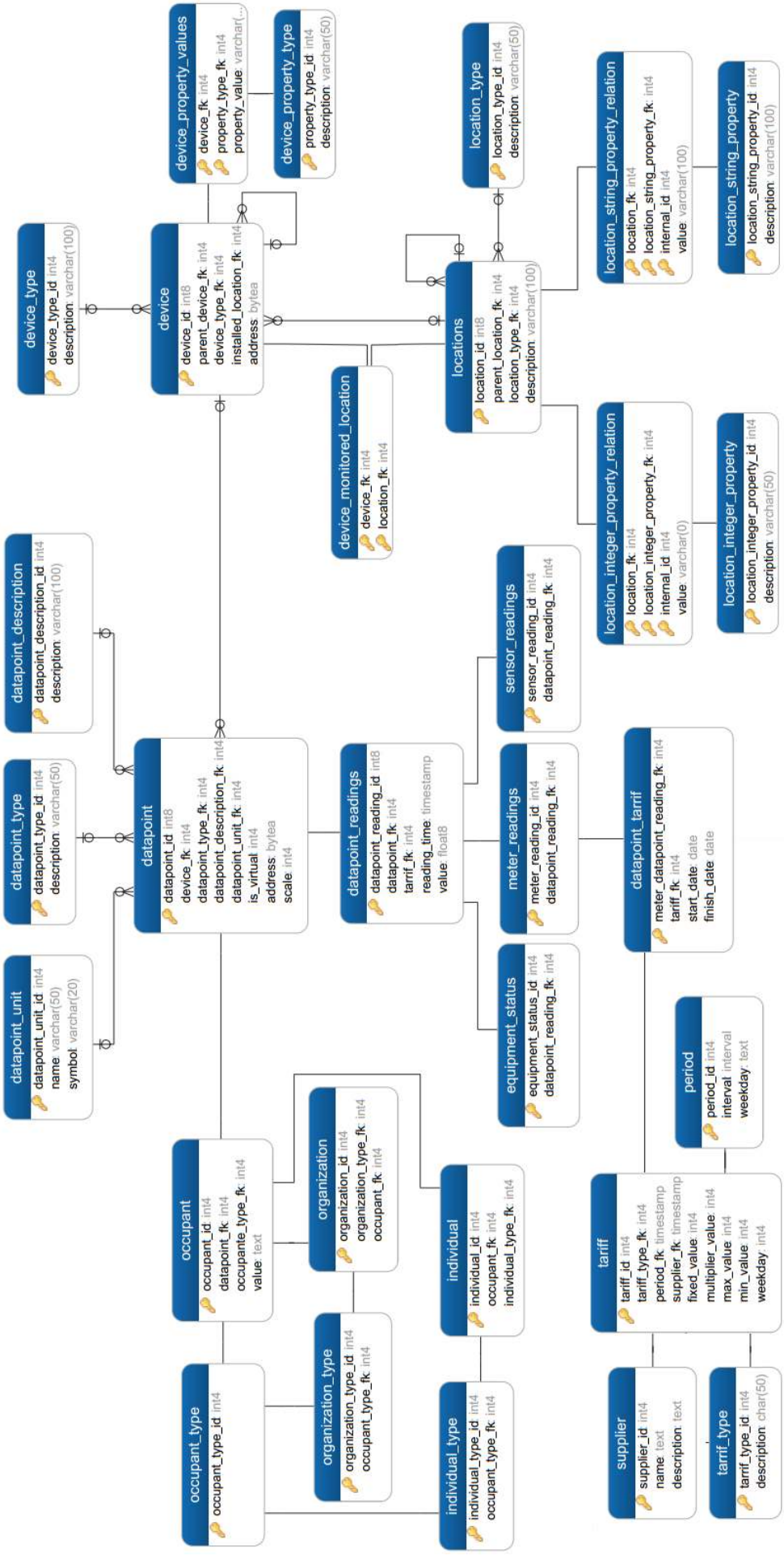

Figure C.1: Entity-Relationship diagram of the EMS database that is in production at IST Taguspark (Crow's Foot Notation). ER diagram models the domain that is related with the energy meters, and other relevant sensors, deployed at campus. The data schema was developed under project Smart Campus, and is in production to support the EMS that is being developed by this same project.

Appendix D

Population of the Solution Database Schema

The values used to populate the database schema of both solution versions of the Data Processing Architecture are listed bellow. Note that, the values of Datapoint Reading table are just a small sample of the entire table.

Persisted Energy Meters Metadata
These relations exist in both DBMS and DSMS Solutions

Persisted Energy Meters Data Streams
These relation only exists in DBMS Solution

Device

	device_pk bigint	device_location_fk bigint
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7
8	8	8

Device Location

	device_location_pk bigint	location character varying(100)	area_m2 integer
1	1	LIBRARY	500
2	2	LECTUREHALL A4	110
3	3	LECTUREHALL A5	110
4	4	CLASSROOM 1 17	60
5	5	CLASSROOM 1 19	60
6	6	DEPARTMENT 14	225
7	7	DEPARTMENT 16	240
8	8	LAB 1 58 MIT	60

Datapoint Description

	datapoint_description_pk bigint	description character varying(100)
1	10	Phase1 EnergyConsumption
2	11	Phase2 EnergyConsumption
3	12	Phase3 EnergyConsumption

Datapoint Unit

	datapoint_unit_pk bigint	unit character varying(100)
1	4	WATT.HOUR

Datapoint

	datapoint_pk bigint	device_fk bigint	datapoint_description_fk bigint	datapoint_unit_fk bigint
1	81	1	10	4
2	82	1	11	4
3	83	1	12	4
4	85	2	10	4
5	86	2	11	4
6	87	2	12	4
7	89	3	10	4
8	90	3	11	4
9	91	3	12	4
10	93	4	10	4
11	94	4	11	4
12	95	4	12	4
13	97	5	10	4
14	98	5	11	4
15	99	5	12	4
16	101	6	10	4
17	102	6	11	4
18	103	6	12	4
19	105	7	10	4
20	106	7	11	4
21	107	7	12	4
22	109	8	10	4
23	110	8	11	4
24	111	8	12	4

Datapoint Reading

	datapoint_reading_pk bigint	measure_timestamp timestamp(6) without time zone	measure double precision	datapoint_fk bigint
1	4071301	2014-05-01 00:00:02	100.38	103
2	4071300	2014-05-01 00:00:02	78.54	102
3	4071299	2014-05-01 00:00:02	194.7	101
4	4071292	2014-05-01 00:00:02	2.33	91
5	4071291	2014-05-01 00:00:02	42.624	90
6	4071290	2014-05-01 00:00:02	0	89
7	4071304	2014-05-01 00:00:02	90.342	87
8	4071303	2014-05-01 00:00:02	0	86
9	4071302	2014-05-01 00:00:02	439.92	85
10	4071298	2014-05-01 00:00:03	10.704	111
11	4071297	2014-05-01 00:00:03	134.838	110
12	4071296	2014-05-01 00:00:03	40.296	109
13	4071286	2014-05-01 00:00:03	0	99
14	4071285	2014-05-01 00:00:03	0	98
15	4071284	2014-05-01 00:00:03	0	97
16	4071283	2014-05-01 00:00:03	64.86	95
17	4071282	2014-05-01 00:00:03	0	94
18	4071281	2014-05-01 00:00:03	63.48	93
19	4071289	2014-05-01 00:00:04	0	107
20	4071288	2014-05-01 00:00:04	413.478	106
21	4071287	2014-05-01 00:00:04	0	105
22	4071295	2014-05-01 00:00:04	236.572	83
23	4071294	2014-05-01 00:00:04	1348.9	82
24	4071293	2014-05-01 00:00:04	2977.38	81
25	4071310	2014-05-01 00:01:03	100.38	103
26	4071309	2014-05-01 00:01:03	77.35	102
27	4071308	2014-05-01 00:01:03	205.32	101
28	4071313	2014-05-01 00:01:03	90.342	87
29	4071312	2014-05-01 00:01:03	0	86
30	...	...	...	...
31	...	...	...	...
32	...	...	...	...

Figure D.1: Population of the database schema that supports the proposed solution. The complete dataset of metadata (left) is depicted, being this five tables persisted in the databases of both solution versions. Whereas, the dataset depicted in the table that holds the energy meters data stream measurements (right) are just a small sample of the entire dataset—that is constantly growing—, and that are only database persisted in the DBMS version of the solution.

Appendix E

Implementation of Use-Case Queries

Implementation of **Use-Case Queries** presented in Table 4.3 and depicted in Figure 4.4.

E.1 Integration Queries

E.1.1 Q4 Implementation

```
SELECT device_pk          AS device_pk
       measure_timestamp AS measure_timestamp
       measure/sum(measure)
       OVER wintotal*100 AS measure,
       'Percentage%'     AS measure_unit,
       'Pie chart of energy consumptions
       per device'       AS measure_description,
       device_location   AS device_location
FROM "Q14_Normalization"
WHERE device_pk <> 0
  AND index=1
WINDOW wintotal AS (PARTITION BY NULL)
```

Example E.1: Q4 implementation in SQL (PostrgreSQL)

```
SELECT device_pk          AS device_pk
       measure_timestamp  AS measure_timestamp,
       (measure/SUM(measure))*100 AS measure,
       "Percentage%"      AS measure_unit,
       "Pie chart of energy consumptions
        per device"       AS measure_description,
       device_location    AS device_location
FROM Q14_Normalization.std:unique(device_pk)
WHERE device_pk != 0
OUTPUT SNAPSHOT EVERY 1 EVENTS
```

Example E.2: Q4 implementation in EPL (Esper)

E.1.2 Q5 Implementation

```
SELECT device_pk          AS device_pk,
       measure_timestamp AS measure_timestamp,
       rank() OVER sortedwindow AS measrure,
       measure         AS current_power_consumption,
       'Ranking List Position' AS measure_unit,
       'Ranked list of energy consumption
       per device'       AS measure_description,
       device_location  AS measure_location
FROM "Q14_Normalization"
WHERE index = 1
WINDOW sortedwindow AS (PARTITION BY NULL
                        ORDER BY measure DESC)
```

Example E.3: Q5 implementation in SQL (PostrgreSQL)

```
SELECT device_pk          AS device_pk,
       measure_timestamp AS measure_timestamp,
       measure           AS measure,
       "WATT"            AS measure_unit,
       "Ranked list of energy consumption
        per device"      AS measure_description,
       device_location   AS device_location
FROM Q14_Normalization.std:unique(device_pk)
OUTPUT SNAPSHOT EVERY 1 EVENTS
ORDER BY measure DESC
```

Example E.4: Q5 implementation in EPL (Esper)

E.1.3 Q6 Implementation

```
SELECT r1.device_pk           AS device_pk,
       r1.measure_timestamp   AS measure_timestamp,
       (MIN(r2.measure)
       /MAX(r2.measure))      AS measure,
       'Ratio = [0,1]'        AS measure_unit,
       'Min/Max Power Consumption Ratio
        during last hour'     AS measure_description,
       r1.device_location     AS device_location,
       MIN(r2.measure)        AS min_hourly,
       MAX(r2.measure)        AS max_hourly
FROM "Q14_Normalization" AS r1
     JOIN
     "Q14_Normalization" AS r2
     ON  r1.index      = 1
     AND r2.device_pk = r1.device_pk
     AND r2.measure_timestamp
       > r1.measure_timestamp-(interval'60 minutes')
GROUP BY r1.device_pk,
         r1.measure_timestamp,
         r1.measure_unit,
         r1.measure_description,
         r1.device_location
```

Example E.5: Q6 implementation in SQL (PostrgreSQL)

```
SELECT device_pk          AS device_pk,
       measure_timestamp AS measure_timestamp,
       (MIN(measure)
       /MAX(measure))     AS measure,
       "Ratio = [0,1]"    AS measure_unit,
   "Min/Max Power Consumption
   Ratio during last hour" AS measure_description,
       device_location     AS device_location,
       MIN(measure)        AS min_hourly,
       MAX(measure)        AS max_hourly,
FROM Q14_Normalization.win:ext_timed
         (measure_timestamp, 60 min)
GROUP BY device_pk
```

Example E.6: Q6 implementation in EPL (Esper)

E.1.4 Q10 Implementation

```
CREATE MATERIALIZED VIEW "Q10_3PhAgg_DBIntegr" AS
SELECT dev.device_pk          AS device_pk,
       dpr.measure_timestamp AS measure_timestamp,
       SUM(dpr.measure)       AS measure,
       'WATT'                 AS measure_unit,
       'Power Consumption'    AS measure_description,
       dl.location            AS device_location,
       dl.area_m2             AS location_area_m2,
       rank()       OVER w    AS index
FROM   "DataPoint" dp
       JOIN "DataPointReading" dpr
       ON dpr.datapoint_fk = dp.datapoint_pk
       JOIN "Device" dev
       ON dp.device_fk = dev.device_pk
       JOIN "DeviceLoceation" dl
       ON dev.device_location_fk
       =   dl.device_location_pk
       JOIN "DataPointDescription" dpd
       ON dp.datapoint_description_fk
       = dpd.datapoint_description_pk
       JOIN "DataPointUnit" dpu
       ON dp.datapoint_unit_fk
       = dpu.datapoint_unit_pk
WHERE  dpd.description = "Phase1_EnergyConsumption"
   OR  dpd.description = "Phase2_EnergyConsumption"
   OR  dpd.description = "Phase3_EnergyConsumption"
GROUP BY dev.device_pk,
         dpr.measure_timestamp,
         dl.location,
         dl.area_m2
HAVING COUNT(dpr.measure) = 3
WINDOW w AS (PARTITION BY dev.device_pk
             ORDER BY dpr.measure_timestamp DESC);
```

Example E.7: Q10 implementation in SQL (PostrgreSQL)

```
INSERT INTO Q10_3PhAgg_DBIntegr
SELECT bd.device_pk          AS device_pk,
       stream.measureTS      AS measure_timestamp,
       SUM(stream.measure)   AS measure,
       "WATT"                AS measure_unit,
       "Power Consumption"   AS measure_description,
       bd.device_location    AS device_location,
       bd.location_area_m2   AS location_area_m2
FROM Datastream.Measure      AS stream,
     sql:database[
     SELECT dev.device_pk    AS device_pk,
            dpu.unit         AS measure_unit,
           dpd.description   AS measure_description,
            dl.location      AS device_location,
            dl.area_m2       AS location_area_m2
     FROM   "DataPoint"            dp
            JOIN "Device"          dev
            ON dp.device_fk = dev.device_pk
            JOIN "DeviceLocation" dl
            ON dev.device_location_fk
            =   dl.device_location_pk
            JOIN "DataPointDescriptiondpd" dpd
            ON dp.datapoint_description_fk
            = dpd.datapoint_description_pk
            JOIN "DataPointUnit"  dpu
            ON dp.datapoint_unit_fk
            = dpu.datapoint_unit_pk
     WHERE ${stream.datapointFk} = dp.datapoint_pk
       AND  (dpd.description
                = "Phase1_EnergyConsumption"
            OR dpd.description
                = "Phase2_EnergyConsumption"
            OR dpd.description
                = "Phase3_EnergyConsumption")
     ] AS bd
GROUP BY bd.device_pk,
         stream.measureTS
HAVING COUNT(stream.measureTS) = 3;
```

Example E.8: Q10 implementation in EPL (Esper)

E.1.5 Q11 Implementation

```
CREATE MATERIALIZED VIEW "Q11_Variation" AS
SELECT r1.device_pk          AS device_pk,
       r1.measure_timestamp AS measure_timestamp,
       (r1.measure/AVG(r2.measure)-1)*100 AS measure,
       r1.measure    AS current_power_consumption,
       'Percentage%'         AS measure_unit,
       'Power Consumption variation
        over 5 minutes'      AS measure_description,
       r1.device_location    AS device_location,
       r1.location_area_m2   AS location_area_m2,
       rank() OVER w         AS index
FROM   "Q10_3PhAgg_DBIntegr" r1
       JOIN
       "Q10_3PhAgg_DBIntegr" r2
       ON  r1.device_pk = r2.device_pk
       AND r2.measure_timestamp
           > (r1.measure_timestamp - '00:05:00')
       AND r2.measure_timestamp
           <= r1.measure_timestamp
GROUP BY r1.device_pk,
         r1.measure_timestamp,
         r1.measure,
         r1.measure_unit,
         r1.measure_description,
         r1.device_location,
         r1.location_area_m2
WINDOW w AS (PARTITION BY r1.device_pk
             ORDER BY r1.measure_timestamp DESC
             RANGE BETWEEN CURRENT ROW
             AND UNBOUNDED FOLLOWING)
```

Example E.9: Q11 implementation in SQL (PostrgreSQL)

```
INSERT INTO Q11_Variation
SELECT device_pk          AS device_pk,
       measure_timestamp  AS measure_timestamp,
       (measure/AVG(measure) - 1)*100 AS measure,
       measure      AS current_power_consumption,
       "Percentage%"      AS measure_unit,
       "Power Consumption variation
        over 5 minutes"   AS measure_description,
       device_location    AS device_location,
       location_area_m2   AS location_area_m2
FROM   Q10_3PhAgg_DBIntegr.win:ext_timed
           (measure_timestamp, 5 min)
GROUP BY device_pk
```

Example E.10: Q11 implementation in EPL (Esper)

E.1.6 Q12 Implementation

```
CREATE MATERIALIZED VIEW "Q12_Period" AS
SELECT r1.device_pk           AS device_pk,
       r1.measure_timestamp   AS measure_timestamp,
       (r1.measure_timestamp
       -r2.measure_timestamp) AS measure,
      'Time Seconds'          AS measure_unit,
   'Last measurement period' AS measure_description,
      r1.device_location      AS device_location,
      r1.location_area_m2     AS location_area_m2,
      r1.index                AS index
FROM "Q10_3PhAgg_DBIntegr" r1
     JOIN
     "Q10_3PhAgg_DBIntegr" r2
     ON   r1.device_pk = r2.device_pk
     AND  r1.index + 1 = r2.index
```

Example E.11: Q12 implementation in SQL (PostrgreSQL)

```
INSERT INTO Q12_Period
SELECT device_pk             AS device_pk
   last(measure_timestamp,0) AS measure_timestamp,
   (last(measure_timestamp,0)
   -last(measure_timestamp,1)) AS measure,
   "Time Seconds"              AS measure_unit,
 "Last measurement period" AS measure_description,
   device_location         AS device_location,
   location_area_m2        AS location_area_m2
FROM Q10_3PhAgg_DBIntegr
     .std:groupwin(device_pk).win:length(2)
GROUP BY device_pk
HAVING COUNT(*) > 1
```

Example E.12: Q12 implementation in EPL (Esper)

E.1.7 Q13 Implementation

```
CREATE MATERIALIZED VIEW "Q13_Smoothing" AS
SELECT r1.device_pk            AS device_pk,
       r1.measure_timestamp    AS measure_timestamp,
       AVG(r2.measure)         AS measure,
       'WATT'                  AS measure_unit,
       'Smoothing measurements with
       10minAVG slide window'  AS measure_description,
       r1.device_location      AS device_location,
       r1.location_area_m2     AS location_area_m2,
       rank() OVER w           AS index
FROM "Q10_3PhAgg_DBIntegr" r1
     JOIN
     "Q10_3PhAgg_DBIntegr" r2
     ON  r1.device_pk = r2.device_pk
     AND r2.measure_timestamp
         >= (r1.measure_timestamp - '00:05:00')
     AND r2.measure_timestamp
         <= r1.measure_timestamp
GROUP BY r1.device_pk,
         r1.measure_timestamp,
         r1.device_location,
         r1.location_area_m2
WINDOW w AS (PARTITION BY r1.device_pk
             ORDER BY r1.measure_timestamp DESC)
```

Example E.13: Q13 implementation in SQL (PostrgreSQL)

```
INSERT INTO Q13_Smoothing
SELECT device_pk            AS device_pk,
       measure_timestamp    AS measure_timestamp,
       AVG(measure)         AS measure,
       "WATT"               AS measure_unit,
   "Smoothing measurements with
   10min AVG slide window"  AS measure_description,
       device_location      AS device_location,
       location_area_m2     AS location_area_m2
FROM Q10_3PhAgg_DBIntegr
     .win:ext_timed(measure_timestamp, 5 min)
GROUP BY device_pk
```

Example E.14: Q13 implementation in EPL (Esper)

E.1.8 Q14 Implementation

```
CREATE MATERIALIZED VIEW "Q14_Normalization" AS
SELECT device_pk               AS device_pk,
       measure_timestamp       AS measure_timestamp,
       measure/location_area_m2 AS measure,
       'WATT/m^2'              AS measure_unit,
       'Each device square
       meter normalization'    AS measure_description,
       device_location         AS device_location,
       index                   AS index,
FROM   "Q13_Smoothing"
UNION
SELECT rel.device_pk           AS device_pk,
       rel.measure_timestamp   AS measure_timestamp,
       rel.measure             AS measure,
       'WATT/m^2'              AS measure_unit,
       measure_description     AS measure_description,
       rel.device_location     AS device_location,
       rank() OVER w           AS rank
FROM  (SELECT 0 AS device_pk,
      to_timestamp(((((((
      date_part('year', measure_timestamp) ||'-')||
      date_part('month',measure_timestamp))||'-')||
      date_part('day',  measure_timestamp))||' ')||
      date_part('hour', measure_timestamp))||':')||
      date_part('minute',measure_timestamp),
      'YYYY-MM-DD HH24:MI:SS') AS measure_timestamp,
      SUM(measure)/SUM(location_area_m2) AS measure,
      'WATT/m^2'               AS measure_unit,
      'All Building square meter
      normalized consumption' AS measure_description,
      'ALL_BUILDING'           AS device_location
    FROM "Q13_Smoothing"
    GROUP BY date_part('year', measure_timestamp),
             date_part('month', measure_timestamp),
             date_part('day',   measure_timestamp),
             date_part('hour',  measure_timestamp),
             date_part('minute',measure_timestamp)
     HAVING count(device_pk) = 8) rel
WINDOW w AS (ORDER BY rel.measure_timestamp DESC)
```

Example E.15: Q14 implementation in SQL (PostrgreSQL)

```
INSERT INTO AuxStream_SquareMeterNormalization
SELECT device_pk              AS device_pk,
       measure_timestamp      AS measure_timestamp,
       measure/location_area_m2 AS measure,
       "WATT/m^2"             AS measure_unit,
     "Each device square
      meter normalization"    AS measure_description,
      device_location         AS device_location
FROM   Q13_Smoothing


INSERT INTO AuxStream_SquareMeterNormalization
SELECT 0                      AS device_pk,
      MIN(measure_timestamp)  AS measure_timestamp,
     (SUM(measure)
     /SUM(location_area_m2))  AS measure,
     "WATT/m^2"               AS measure_unit,
  "All Building square meter
   normalized consumption"    AS measure_description,
     "ALL_BUILDING"           AS device_location
FROM   Q13_Smoothing.std:unique(device_pk)
HAVING COUNT(device_pk) = 8
OUTPUT LAST EVERY 8 EVENTS


INSERT INTO Q14_Normalization
SELECT device_pk,
       measure_timestamp_long,
       measure_timestamp,
       measure,
       measure_unit,
       measure_description,
       device_location
FROM  AuxStream_SquareMeterNormalization
```

Example E.16: Q14 implementation in EPL (Esper)

E.1.9 Q15 Implementation

```sql
CREATE MATERIALIZED VIEW "Q15_ExpectedUDF" AS
SELECT  device_pk           AS device_pk,
        measure_timestamp   AS measure_timestamp,
        measure             AS current_measure,
        getExpectedMeasureUDF(device_pk,
        measure_timestamp) AS expected_measure,
        'WATT/m^2'          AS measure_unit,
    'Current and Expected Power
     consumption given by UDF' AS measure_description,
        device_location     AS device_location,
        index               AS index
FROM "Q14_Normalization"
```

Example E.17: Q15 implementation in SQL (PostrgreSQL)

```
INSERT INTO Q15_ExpectedUDF
SELECT device_pk            AS device_pk,
       measure_timestamp    AS measure_timestamp,
       measure              AS current_measure,
       getExpectedMeasureUDF(device_pk,
       measure_timestamp)   AS expected_measure,
       "WATT/m^2"           AS measure_unit,
"Current and Expected Power
consumption given by UDF." AS measure_description,
       device_location      AS device_location
FROM Q14_Normalization
```

Example E.18: Q15 implementation in EPL (Esper)

E.1.10 Q16 Implementation

```sql
CREATE MATERIALIZED VIEW "Q16_ExpectedLastMonthPivotHourAVG" AS
SELECT rel.device_pk                AS device_pk,
       rel.measure_timestamp_pivot  AS measure_timestamp,
       rel.measure                  AS current_measure,
       rel.expecetd_measure         AS expecetd_measure,
       'WATT/m^2'                   AS measure_unit,
       'Current and Expected Power consumption given by the
        AVG of the current hour computed along last month' AS measure_description,
       rel.device_location          AS device_location,
       rank() OVER w2               AS index
FROM   (SELECT  pivot_measures.device_pk              AS device_pk,
                pivot_measures.measure_timestamp      AS measure_timestamp_pivot,
                cluster_measures.measure_timestamp    AS measure_timestamp_cluster,
                pivot_measures.measure                AS measure,
                AVG(cluster_measures.measure) OVER w1 AS expecetd_measure,
                pivot_measures.measure_description    AS measure_description,
                pivot_measures.measure_unit           AS measure_unit,
                pivot_measures.device_location        AS device_location,
                rank()                          OVER w1 AS rank
        FROM "Q14_Normalization" cluster_measures
             JOIN
             "Q14_Normalization" pivot_measures
             ON  cluster_measures.device_pk = pivot_measures.device_pk
             AND cluster_measures.measure_timestamp <= pivot_measures.measure_timestamp
             AND cluster_measures.measure_timestamp > (pivot_measures.measure_timestamp - '1 month')
         WINDOW w1 AS (PARTITION BY cluster_measures.device_pk,
                                    date_part('hour', cluster_measures.measure_timestamp),
                                    pivot_measures.index
                       ORDER BY cluster_measures.measure_timestamp
                       DESC RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING)
       ) rel
WHERE  rel.rank = 1
  AND  date_part('hour', rel.measure_timestamp_pivot) = date_part('hour', rel.measure_timestamp_cluster)
WINDOW w2 AS (PARTITION BY rel.device_pk
              ORDER BY rel.measure_timestamp_pivot DESC)
```

Example E.19: Q16 implementation in SQL (PostrgreSQL)

```
INSERT INTO Q16_ExpectedLastMonthPivotHourAVG
SELECT device_pk          AS device_pk,
       measure_timestamp  AS measure_timestamp,
       measure            AS current_measure,
       AVG(measure)       AS expected_measure,
       "WATT/m^2"         AS measure_unit,
       "Current and Expected Power consumption given by the
        AVG of the current hour computed along last month"  AS measure_description,
       device_location    AS device_location
FROM Q14_Normalization.win:ext_timed(measure_timestamp, 1 month)
GROUP BY device_pk,
         DateTime.toDate(measure_timestamp, "yyyy-MM-dd HH:mm:ss").getHours()
```

Example E.20: Q16 implementation in EPL (Esper)

E.2 Evaluation Queries

E.2.1 Q1 Implementation

```sql
SELECT device_pk,
       measure_timestamp,
       measure,
       current_power_consumption,
       'Percentage%'          AS measure_unit,
   'Power Consumption variation over
    5 min exceeds threshold' AS measure_description,
       device_location,
       location_area_m2
FROM   "Q11_Variation"
WHERE index = 1    /* Threshold Limit Values */
  AND((device_pk = 1 AND measure >= ThrVal1)
   OR (device_pk = 2 AND measure >= ThrVal2)
   OR (device_pk = 3 AND measure >= ThrVal3)
   OR (device_pk = 4 AND measure >= ThrVal4)
   OR (device_pk = 5 AND measure >= ThrVal5)
   OR (device_pk = 6 AND measure >= ThrVal6)
   OR (device_pk = 7 AND measure >= ThrVal7)
   OR (device_pk = 8 AND measure >= ThrVal8))
```

Example E.21: Q1 implementation in SQL (PostrgreSQL)

```sql
SELECT device_pk,
       measure_timestamp,
       measure,
       current_power_consumption,
       "Percentage%"          AS measure_unit,
  "Power Consumption variation over
  5 min exceeds threshold" AS measure_description,
       device_location
       location_area_m2
FROM  Q11_Variation
                  /* Threshold Limit Values */
WHERE ((device_pk = 1 AND measure >= ThrVal1)
   OR  (device_pk = 2 AND measure >= ThrVal2)
   OR  (device_pk = 3 AND measure >= ThrVal3)
   OR  (device_pk = 4 AND measure >= ThrVal4)
   OR  (device_pk = 5 AND measure >= ThrVal5)
   OR  (device_pk = 6 AND measure >= ThrVal6)
   OR  (device_pk = 7 AND measure >= ThrVal7)
   OR  (device_pk = 8 AND measure >= ThrVal8))
```

Example E.22: Q1 implementation in EPL (Esper)

E.2.2 Q2 Implementation

```sql
SELECT device_pk,
       measure_timestamp,
       measure,
       'Time Seconds'         AS measure_unit,
    'Last measurement period is out
    of range [55, 65] secs.' AS meausre_description,
      device_location,
      location_area_m2
FROM  "Q12_Period"
WHERE index = 1 AND
  NOT('00:00:55'<=measure AND measure<='00:01:05')
```

Example E.23: Q2 implementation in SQL (PostrgreSQL)

```sql
SELECT device_pk,
       measure_timestamp,
       measure,
       "Time Seconds"         AS measure_unit,
 "Last measurement period is out
  of range [55, 65] secs." AS meausre_description,
       device_location,
       location_area_m2
FROM   Q12_Period
WHERE  NOT(55 <= measure AND measure <= 65)
```

Example E.24: Q2 implementation in EPL (Esper)

E.2.3 Q3 Implementation

```
SELECT device_pk,
       measure_timestamp,
       current_measure              AS measure,
       measure_sliding24h_avg*1.20  AS measure_threshold,
       measure_unit,
       'Power consumption 20% above the AVG of last 24 hours' AS measure_description,
       device_location
FROM (SELECT all_measures.device_pk,
             all_measures.measure_timestamp,
             all_measures.measure             AS current_measure,
             all_measures.measure_unit,
             all_measures.measure_description,
             all_measures.device_location,
             all_measures.location_area_m2,
             AVG(all_measures.measure) over w AS measure_sliding24h_avg,
             rank()                    over w AS index
      FROM  "Q13_Smoothing" AS all_measures
            JOIN
            "Q13_Smoothing" AS most_recent_measure
            ON  most_recent_measure.index = 1
            AND most_recent_measure.device_pk = all_measures.device_pk
            AND all_measures.measure_timestamp >=
                most_recent_measure.measure_timestamp-'24 hours'
      WINDOW w AS (PARTITION BY all_measures.device_pk
                   ORDER BY all_measures.measure_timestamp DESC
                   RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING)
      ) AS rel
WHERE index = 1
  AND current_measure >= measure_sliding24h_avg*1.2 /* Threshold */
```

Example E.25: Q3 implementation in SQL (PostrgreSQL)

```
SELECT device_pk,
       measure_timestamp,
       measure,
       AVG(measure)*1.20 AS measure_threshold,
       measure_unit,
       "Power consumption 20% > last 24 hours" AS measure_description,
       device_location
FROM Q13_Smoothing.win:ext_timed(measure_timestamp_long, 24 hours)
GROUP BY device_pk
HAVING measure >= AVG(measure)*1.20/* Threshold */
```

Example E.26: Q3 implementation in EPL (Esper)

E.2.4 Q7 Implementation

```
SELECT device_pk,
       measure_timestamp,
       measure,
       measure_unit,
       'Power consumption
       above threshold' AS measure_description,
       device_location,
FROM  "Q14_Normalization"
WHERE index = 1    /* Threshold Limit Values */
  AND ((device_pk = 0 AND measure >= ThrVal0)
    OR (device_pk = 1 AND measure >= ThrVal1)
    OR (device_pk = 2 AND measure >= ThrVal2)
    OR (device_pk = 3 AND measure >= ThrVal3)
    OR (device_pk = 4 AND measure >= ThrVal4)
    OR (device_pk = 5 AND measure >= ThrVal5)
    OR (device_pk = 6 AND measure >= ThrVal6)
    OR (device_pk = 7 AND measure >= ThrVal7)
    OR (device_pk = 8 AND measure >= ThrVal8))
```

Example E.27: Q7 implementation in SQL (PostrgreSQL)

```
SELECT device_pk,
       measure_timestamp,
       measure,
       measure_unit,
       "Power consumption
       above threshold" AS measure_description,
       device_location
FROM  Q14_Normalization
                  /* Threshold Limit Values */
WHERE (device_pk = 0 AND measure >= ThrVal0)
   OR (device_pk = 1 AND measure >= ThrVal1)
   OR (device_pk = 2 AND measure >= ThrVal2)
   OR (device_pk = 3 AND measure >= ThrVal3)
   OR (device_pk = 4 AND measure >= ThrVal4)
   OR (device_pk = 5 AND measure >= ThrVal5)
   OR (device_pk = 6 AND measure >= ThrVal6)
   OR (device_pk = 7 AND measure >= ThrVal7)
   OR (device_pk = 8 AND measure >= ThrVal8)
```

Example E.28: Q7 implementation in EPL (Esper)

E.2.5 Q8 Implementation

```
SELECT r2.device_pk,
     MAX(r2.measure_timestamp) AS measure_timestamp,
     COUNT(r2.current_measure) AS measure,
     'Positive Integer'        AS measure_unit
 'Number of times that current consumption
 was greater than expected' AS measure_description,
     r2.device_location
FROM "Q15_ExpectedUDF" r1
     JOIN
     "Q15_ExpectedUDF" r2
     ON  r1.device_pk = r2.device_pk
     AND r1.index = 1
     AND r2.measure_timestamp
      > (r1.measure_timestamp-'01:00:00')
WHERE r2.current_measure > r2.expected_measure
GROUP BY r2.device_pk,
         r2.expected_measure,
         r2.device_location
HAVING 5 <= COUNT(r2.current_measure)
        AND COUNT(r2.current_measure) <= 10
```

Example E.29: Q8 implementation in SQL (PostrgreSQL)

```
SELECT device_pk,
       measure_timestamp,
       COUNT(current_measure) AS measure,
       "Positive Integer"     AS measure_unit,
 "Number of times that current consumption
 was greater than expected" AS measure_description,
       device_location
FROM   Q15_ExpectedUDF.win:ext_timed
       (measure_timestamp, 60 min)
WHERE  current_measure > expected_measure
GROUP BY device_pk
HAVING 5 <= COUNT(current_measure)
        AND COUNT(current_measure) <= 10
```

Example E.30: Q8 implementation in EPL (Esper)

E.2.6 Q9 Implementation

```
SELECT device_pk,
       measure_timestamp,
       (current_measure
       /expecetd_measure-1)*100 AS measure,
       'Percentage%'            AS measure_unit,
 'Delta between current and expecetd consumption
 exceeded a given threshold' AS measure_description,
       device_location,
       current_measure,
       expecetd_measure
FROM "Q16_ExpectedLastMonthPivotHourAVG"
WHERE index = 1                      /* Threshold */
  AND (current_measure/expecetd_measure-1)*100 > 10
```

Example E.31: Q9 implementation in SQL (PostrgreSQL)

```
SELECT device_pk,
       measure_timestamp,
       (current_measure
       /expected_measure-1)*100 AS measure,
       "Percentage%"            AS measure_unit,
"Delta between current and expecetd consumption
exceeded a given threshold" AS measure_description,
       current_measure
       expected_measure
       device_location
FROM Q16_ExpectedLastMonthPivotHourAVG
                                     /* Threshold */
WHERE (current_measure/expected_measure-1)*100 > 10
```

Example E.32: Q9 implementation in EPL (Esper)