Journal of Universal Computer Science, vol. 22, no. 2 (2016), 228-246
submitted: 19/5/15, accepted: 29/1/16, appeared: 1/2/16 © J.UCS

A Data Management Middleware for ITS Services in Smart Cities

Luca Carafoli
(FIM - University of Modena and Reggio Emilia, Italy
luca.carafoli@unimore.it)

Federica Mandreoli
(FIM - University of Modena and Reggio Emilia, Italy
federica.mandreoli@unimore.it)

Riccardo Martoglia
(FIM - University of Modena and Reggio Emilia, Italy
riccardo.martoglia@unimore.it)

Wilma Penzo
(DISI - University of Bologna, Italy
wilma.penzo@unibo.it)

Abstract: A major societal challenge to be tackled in megacities is sustainable urban transportation. Intelligent Transportation Systems (ITSs) are actually data-centric applications that need to store and query real-time as well as historical/static data from various data sources and have to provide timely responses to users' transportation needs.

In this paper we introduce a data management middleware that offers the robustness of a common framework to support the development of smart applications having the above needs. It supports the efficient storage and access to real-time and historical/static data and provides both one-time and continuous query capabilities. While the middleware has been designed to be general and versatile to support data management for any kind of application, in this paper we explore its suitability to ITS smart services also by means of an experimental evaluation conducted on a variety of traffic scenarios.

Key Words: Data management middleware, Intelligent Transportation Systems, Database Management Systems, Data Stream Management Systems, Smart City
Category: H.2, H.2.8, E.2

1 Introduction

By 2025, the upward trend of world urbanization is expected to move around 5 billion people towards megacities. Great efforts are going to be made to make citizens' life-style sustainable in such an overcrowded scenario, thus laying the foundations for the so-called smart cities [European Commission, 2014]. According to the Horizon 2020 EU Programme, a major societal challenge to be tackled

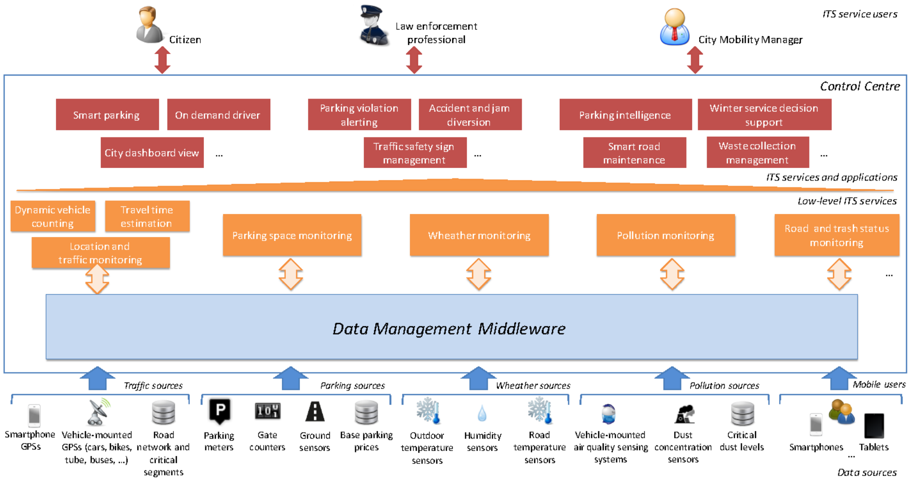

Figure 1: The data management middleware in a smart city ITS scenario

in a smart city is sustainable urban transportation. Decreasing traffic congestions is the first step to be taken to counteract traffic related effects, such as the increase of pollution, the increase of the fuel cost, and the worsening of the public transportation.

In this perspective, smart services that make transportation an easier and more efficient experience are expected to be developed and delivered. The upper part of Figure 1 shows some examples of services that are of interest for different stakeholders. For example, a driver would be encouraged to take an alternative means (e.g. bus, metro, bike sharing) knowing that no parking is available, or that it is not allowed, at the desired destination [Delot et al., 2013]. Similarly, a mobility manager could dynamically reinforce the transportation services on the basis of traffic conditions and of the expectation of large inflows due, for instance, to sporting events. Such services can exploit the recent development of sensing technology to collect a multitude of real-time data from various sources, e.g. vehicles, toll gates, traffic lights, parking meters, weather stations, and others (see the lower part of Figure 1).

Nevertheless, Intelligent Transportation System (ITS) services are also expected to access historical and static data, e.g. traffic statistics, road works and street cleaning programmes, upcoming exhibitions. Several smart ITS services already exist and are made available to citizens by cities on the crest of the technological wave, like London (City Dashboard [UCL, 2013]), San Francisco (Dynamic Parking Pricing [SFMTA, 2014]), and Oslo (Street Lighting [Echelon Corp., 2013]), just to mention a few. The approach adopted so far for the development of ITS smart services has been to carry out independent implementations. However, despite the fact that such services implement different logics depending on their own specific objectives, they actually are data-centric applications that need/have to:

- access heterogeneous data at different sources;
- manage and query high rate data streams;
- provide real-time responses.

The implementation of this kind of applications would greatly benefit of software development facilities like:

- data management capabilities to satisfy the above needs;
- the robustness of a common framework, to give up naïf implementation solutions that start from scratch every time;
- the decoupling between service logics and data management, so that developers can focus on the implementation of the service objectives;
- ease of specification of services' data access needs, to facilitate the developers to make use of various data by disregarding specific data source access details.

This is in line with the well-established design principles of DBMSs, that represent a full-fledged and universally adopted technology for the design and development of data-centric applications. However, standard DBMSs are not suitable for such smart services' purposes, mainly because they are inefficient in storing/retrieving data at the rate that satisfying real-time demands requires [Chandrasekaran and Franklin, 2004].

On the other hand, in order to cope with large volumes of streaming data in use to common software applications, Data Stream Management Systems (DSMSs) were introduced [Abadi et. al., 2003, Arasu et al., 2006, Chen et al., 2000, Liarou et al., 2009]. These systems natively support continuous queries (CQs) over (continuous unbounded) streams of data according to windows where only the most recent data is retained. Once data goes out of the windows it is deleted from the system.

Unfortunately, this model too does not completely fit the needs of a smart city service context, where:

1. past streamed data has to be retained in the system since it represents a valuable knowledge for diverse service purposes, e.g., to provide for statistical data and traffic forecasts;

2. besides CQs, also one-time queries (OTQs) on a variety of recent/historical/static data have to be supported, e.g., to reconstruct the dynamics of an accident.

In order to fulfill these needs, in this paper we present a Data Management Middleware born as a result of our data management experience in the PEGASUS ITS project[1] and featuring the guarantees discussed above. The middleware offers the following facilities:

- it enables the efficient storage of streaming data through the implementation of fast mechanisms for continuous writes on temporary and permanent data stores;
- it manages both static data and real-time/historical data, coming from heterogeneous sources, in a transparent way;
- it provides a wide range of SQL-like query capabilities for the timely delivery of smart city services, i.e. both CQs and OTQs, and supports their execution efficiently.

To this end, we consider the scenario of a Control Centre that offers various smart city services. These services rely on the data access functionalities made available by the data management middleware we propose. Notice that, while the envisioned scenario is conceptually centralized, the offered facilities can be clearly implemented in a distributed fashion on several servers, both at the service level and at the data management level, e.g. by employing cloud computing technologies like Software-as-a-Service (SaaS) and Data-as-a-Service (DaaS) mechanisms, respectively. The proposed middleware has been designed to be general and versatile in efficiently and effectively supporting data management for any kind of service. In this paper we explore its suitability to services for ITSs.

The paper is organized as follows. Section 2 presents the reference scenario and sketches the middleware's architecture, which is detailed in Section 3. Section 4 shows the experimental evaluation we conducted on a variety of traffic scenarios, Section 5 discusses related work, while Section 6 draws conclusions and briefly describes future work.

2 Scenario and Architecture

In this section we consider a scenario where a Control Centre delivers ITS smart services to different users. Looking at the top part of Figure 1, we see some cutting-edge services and applications, among which:

[1] Italian Council Industria 2015 PEGASUS Project, `http://www.wilab.org/content/progetto-pegasus`.

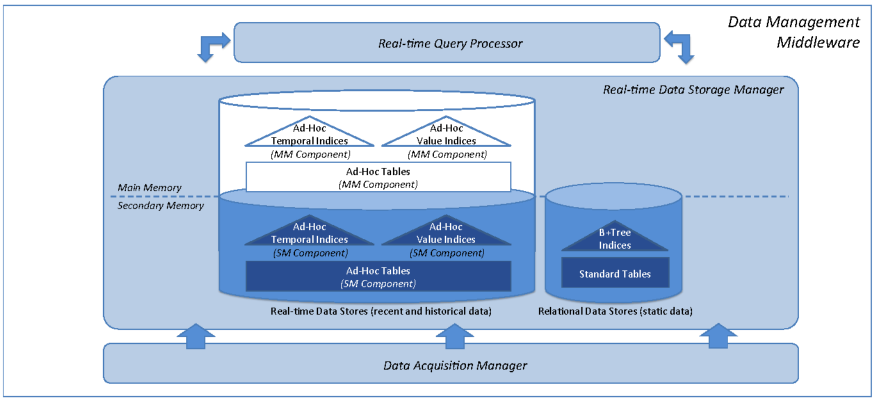

Figure 2: The architecture of the data management middleware

- a "smart parking" service suggesting to drivers the most convenient available space and the best route to reach it, taking into account dynamic weather, traffic and parking space availability information;
- a city "dashboard view" combining observational and official data into a single view of a city, including real time aggregates on bike sharing station status, air quality, river levels, predictions on weather and traffic, tube schedules, etc.
- a "parking violation" service continuously monitoring parking spaces and parking meters to promptly alert law enforcement professional about violations;
- "winter services decision support" and "smart road maintenance" services helping mobility managers to fix road problems and give the best support to drivers based on statistical and real-time weather and road temperature information, smartphone-collected road bumps notifications, etc.

As we can see, these services usually depend on lower level ITS services such as weather, position, and traffic monitoring. The latter, in turn, are based on the prompt availability of data gathered from a very large number of heterogeneous sources (lower part of the figure): highly dynamic data streams coming from vehicle-mounted GPSs, parking meters, weather and pollution sensors, smartphones, but also static data such as road networks, critical road segments that need specific monitoring, critical pollution levels, parking prices, etc.

Although all these services can be developed independently, we state that their high interconnection, their shared use of data sources, and the complexity of their information needs could be much more effectively and efficiently supported by a Data Management Middleware (see central part of Figure 1) that provides a common framework offering data access and query capabilities. In this way, data management would be decoupled from the service logics and the developers would be lightened from the burden of dealing with data management issues.

In our ITS scenario, the data management middleware is located in a Control Centre, and it receives data from different information sources. Figure 2 details the middleware's architecture that consists of three main components:

The *Data Acquisition Manager* has the goal to convert data coming from heterogeneous data sources to common formats to be stored and queried. For example, as to geographic coordinates, one possible format is the Universal Transverse Mercator UTM-NAD83 [NOAA, 2008]. In this case, all data coming in the system in different formats, e.g. Degrees, minutes and seconds - WGS84 [NIMA, 2000], needs to be converted. It is also in charge of implementing data cleansing algorithms to extract useful information from noisy inputs with a satisfactory level of confidence [Kanagal and Deshpande, 2008].

The *Real-time Data Storage Manager (RTDSM)* has the goal to collect, store, and index data, supporting both very large volumes and very high input rates. *Real-time Data Stores* span both main and secondary memory to seamlessly accommodate and retrieve both recent and historical data in a flexible and scalable way, while static data is maintained in standard *Relational Data Stores* and can be accessed and joined when needed.

The *Real-time Query Processor (RTQP)* is in charge of the efficient execution of continuous and one-time queries. For instance, whenever a citizen submits a parking request, the smart parking service could retrieve the vehicle's position through location services, which would perform one-time queries (OTQs) on the position reports acquired from the vehicles. Best routes could be computed thanks to traffic monitoring services, which would perform continuous queries (CQs) returning, every minute, the average traveling speed of each segment. Most importantly, besides "fresh" data, many advanced services could also work on past trends, i.e. historical data, joining them with real-time data. For instance, an "accident and jams diversion" service could forecast the possible jams for the next 30 minutes by comparing the current traffic situation with historical trends, better directing the vehicles and avoiding congestions.

In the following section we will focus on the storage and querying features offered by the RTDSM and RTQP components.

3 Storing and Querying Data

Unlike DSMSs [Abadi et. al., 2003, Arasu et al., 2006, Chen et al., 2000, Liarou et al., 2009] where the storage manager (SM) and the query processor (QP) are tightly coupled, the proposed middleware borrows from traditional

DBMSs a decoupled design principle: the RTDSM is in charge of data storage and indexing and enables the RTQP to solve both OTQs and CQs on the stored data. The decoupling principle preserves change independence between the two components. In this view, the middleware aims at extending traditional DBMS technologies toward stream processing, thus leveraging a considerable amount of long-established query processing and data management techniques and making modifications on one component transparent to the other. Its interface is a declarative language that extends SQL with primitives for real-time data declaration and continuos query specification.

3.1 The Real-time Data Storage Manager

In the RTDSM, static data is kept into the relational data stores that are based on standard relational tables and indices. Real-time data is instead stored into real-time data stores that found on ad-hoc data structures and algorithms meant for supporting very high data write rates and low access latency. While the former kind of data stores is well-established in the literature, the latter one represents a novel contribution and we will focus on it to show how it supports the storage of real-time data flowing from the data sources into the system.

When a real-time data source[2] is added, a real-time data store is directly connected to the source stream and pulls the item from it. To this end, a data definition language statement is put at data and service designers disposal to create data stores and define their schemas. Schemas are represented as a set of attributes and always include a temporal attribute [Snodgrass, 1995] given the temporal nature of the stored data. For instance, the schema of the reports received from the active vehicles can ben defined as `VEHICLES(TIME, VID, SPD, XWAY, LANE, DIR, SEG, POSITION)`, where `TIME` is the temporal attribute, `VID` is the vehicle ID, `SPD` represents the reported speed, while the other attributes are related to the actual position of the vehicle in the reference road map.

Each item is retained in the store for a long period, ideally forever. The middleware allows designers to specify the above period, named historical period, through a specific command. For instance, it is possible to specify that vehicle reports must be retained in its store for two weeks.

Relational data stores exploit standard indices (e.g. B+-trees). Fast access to real-time data stores are instead ensured by two novel kinds of indices that efficiently support the specific workload this kind of data stores are subject to: high write rate together with concurrent continuous reads with real time requirements and one-time reads often involving temporal predicates.

A temporal index on a real-time data store `T` accelerates temporal predicate evaluation as well as temporal window computation. It is implemented at main

[2] Regardless of their nature and aim, we consider real-time data sources as possible infinite sequences of data items having a fixed schema.

memory level through a circular dynamic array that covers `T`'s window and allows for efficient random access to a circular linked list of blocks on a temporal value basis; at secondary memory level, the index exploits the temporal order of `T`'s tuples in the disk blocks and it is implemented as a block-oriented clustered B+-tree built on the lower bound of the time interval covered by each block.

A value index on a real-time data store `T` provides fast access to data values. It extends both kinds of standard secondary memory index, i.e. B+-tree and hash, with a main memory structure for fast access to the most recent data. This component is a linked list of entries where all tuples with the same attribute value are linked through forward and backward pointers, by creating a ring for each attribute value.

More details about indices can be found in [Carafoli et al., 2016]. For instance, it is possible to create an index on the attributes `VID` and `TIME` in the vehicle data store `VEHICLES` to speed up vehicle position requests at specific times, or to index segments (attribute `SEG`) to facilitate the computation of the average traveling speed of each segment.

Data stored in a real-time data store is made available to the RTQP through a data access interface that provides two methods:

- `Scan(store_name[,pred])` that supports one-time reads, where: `store_name` is a real-time data store name; `pred` is an optional selection predicate on `store_name`'s attributes;
- `CScan(store_name,window,sample[,pred])` that implements continuous reads, where: `store_name` is a real-time data store name; `window` is a time-frame that specifies the portion of the most recent items of interest; `sample` is a period of time and represents the delivery period; `pred` is an optional selection predicate on `store_name`'s attributes. Please note that, whereas `CScan` calls are issued only once, updates are delivered at the time rate that is specified in the `sample` parameter.

Both access methods are implemented through sequential scans as well as indexed scans.

3.2 The Real-time Query Processor

The RTQP represents the middleware entry point. It receives CQs and OTQs from the services running in the Control Centre and executes them by accessing the data available in the RTDSM. Results are then delivered to services to be used for the specific service purposes.

For continuous query specification, we draw inspiration from the CQL continuous query language [Arasu et al., 2006]. For instance, the following query could be submitted by the location and traffic monitoring service to calculate the average speed of each road segment:

```
SELECT segment, AVG(speed)
FROM VEHICLES [WINDOW INTERVAL '1' MINUTE]
GROUP BY segment
SAMPLE INTERVAL '1' MINUTE
```

According to the `SAMPLE INTERVAL` statement, the RTQP delivers the query results every minute while the `WINDOW INTERVAL` statement sets the query's domain to the data arrived in the last minute. To process the above query, the RTQP submits the continuous request `CScan(VEHICLES,60,60)`[3] to the RTDSM over the `VEHICLES` real-time data store. Every minute, the RTQP computes the average speed of each road segment within query answering latency specified in the query.

It is worth noting that for the `SAMPLE INTERVAL` specification, the special parameter `REALTIME` can be used in place of a time interval, that means that the query is re-evaluated as new data arrive. For instance the following CQ shows the possibility of querying real-time data together with static data, joining them as needed:

```
SELECT p.stationId, AVG(p.ozone)
FROM POLLUTION p [WINDOW INTERVAL '30' MINUTE]
GROUP BY p.stationId
HAVING AVG(p.ozone) > (SELECT c.qty FROM CRITICAL_LEVELS c
                       WHERE c.type ='ozone')
SAMPLE INTERVAL REALTIME
```

It involves the `POLLUTION` real-time data store that receives pollution reports from the pollution stations at specific time intervals and the relational data store `CRITICAL_LEVELS` that maintains the critical dust levels at the ground level. The query returns the identifiers of the stations whose average ozone values in the reference period are greater than the admitted value (indeed, ozone concentration at the ground level could be harmful to people). In this case the RTQP splits the query into the continuous request `CScan(POLLUTION,1800,REALTIME)` and a standard read request over the `CRITICAL_LEVELS` to the RTDSM. Each time the RTQP receives the required data from the RTDSM, it computes the aggregate values and joins them against the `CRITICAL_LEVELS` values.

Finally, the following query is an OTQ on a a real-time data store:

```
SELECT position
FROM VEHICLES
WHERE time=NOW
AND vid=VID_X
```

[3] Notice that intervals are always expressed in seconds.

returns the current position of a specified vehicle, so it is one base query for the location and traffic monitoring service. In this case, the RTQP submits the one-time request `Scan(VEHICLES,time=NOW AND vid=VID_X)` to the RTDSM and directly delivers the returned result to the calling service.

4 Experimental Evaluation

We implemented a prototype of the data management middleware, where all the real-time data store structures and management code are implemented in Java 1.6. We exploited Oracle BerkeleyDB 11gR2, a lightweight embedded database library, to implement the secondary memory component of the two new kinds of indices, i.e. temporal and value indices, described in Section 3.1; standard relational data stores, i.e. relational tables and their indices, are maintained in PostgreSQL 9.0. Furthermore, we completed the functionalities of our system by implementing in Java the state-of-the-art QP algebraic operators.

In this section we will show the results we obtained from the tests we performed on the prototype. All the experiments are executed on standard PC configuration: an Intel Core2 Quad Q9450 2.66Ghz Win7 Pro 64Bit workstation, equipped with 4GB RAM and a 500GB 7200rpm SATA disk.

4.1 Experimental Setting

Since the tests are mainly focused on the prototype overall performance and scalability, without loss of generality we will consider vehicles (i.e., their position and speed sensors) as the primary source of incoming information. In particular, we will report on the tests we designed to stress the middleware capabilities in reacting to simulated smart city-like workloads. We considered four different city scenarios, reproducing actual traffic conditions of as many cities in different parts of the world:

- Bologna: a portion of the city center of Bologna (Italy), i.e. a typical European urban scenario involving narrow streets;
- Rome: another European portion involving non-central junctions and crossroads in Rome (Italy), with fast and multi-way segments, and high traffic density;
- Toll Plaza: a portion of multi-way road junctions in Camden (New Jersey, USA), including a toll-payment station, with medium traffic density;
- Beijing: a portion of Beijing traffic network, with very intense, congested and heterogenous (i.e. cars, motorbikes and bicycles) traffic conditions.

a)

Scenarios

Scenario	Duration (h)	Size (Km)	#Segments	Vehicles #Vehicles	Vehicles Average life (h)	Vehicles speed (Km/h)	Reports Reports / vehicle	Reports Reports / segment	Reports Reports / minute
Bologna	10	10x10	468	137	7.171	29.19	0	0	7839
Rome	10	2.2x0.7	253	10667	0.22	53.58	166	6994	29489
Toll Plaza	10	2.3x1.4	80	7902	0.43	43.4	307	30358	40477
Beijing	10	1.2x1	100	17184	0.28	15.93	204	35058	58430

b)

Query workload

Query	Data Recent	Data Histor	Data Static	Load #Tot (CQ) / Avg Rate (OTQ) High	Med	Low	Description
CQ1	X			10000	5000	1000	Every minute, the current position of a given vehicle
CQ2	X			10000	5000	1000	Every 5 minutes, the current average speed of the vehicles in a given segment (aggregate)
CQ3	X		X	10000	5000	1000	Every 5 minutes, the current mean number of vehicles for each segment in a list (aggregate)
OTQ1		X		30/sec	15/sec	3/sec	The position of a given vehicle on a given time in the past
OTQ2	X		X	30/sec	15/sec	3/sec	The trajectory (last 50 positions) of each vehicle in a list
OTQ3	X	X		30/sec	15/sec	3/sec	Check if a given vehicle is currently in the same position that it had on a given time in the past

Figure 3: Experimental setting: (a) specifications of the four scenarios and (b) specifications of the query workload

All the scenarios simulate the incoming data exchanges (position reports from vehicles and queries on them) needed to support services such as the ones described in Section 2. The scenarios were created by professional traffic engineers in the PTV Vision VISSIM[4] simulation software, a widely used tool for state-of-art transportation planning and operations analysis. Each of the scenarios is accurately modelled by means of several data defining the characteristics of the roads that need to be implemented: arcs, characterized by the number and form of the lanes; connections between arcs for modeling changes of direction (including turning at intersections) and for reducing or increasing the number of lanes; speed distribution laws for both arcs and connections; specific data to model the resolution of conflict points at intersections, including split levels, precedence rules and traffic lights. The volumes of traffic, the speed, the percentage of violations and all other VISSIM inputs are taken from real data, therefore all the scenarios represent an absolutely realistic traffic behavior.

As we can see from Figure 3(a), which shows the detailed features of the scenarios, all of them share a length of 10 hours, while the rate of incoming position reports varies from nearly 8000 to nearly 60000 reports per minute. The diverse characteristics of the scenarios show the reaction of the middleware in storing, indexing and querying differently distributed data in real-time: for instance, Toll Plaza and Beijing have less and longer road segments and, therefore, a very high concentration of vehicles per segment, Bologna has less vehicles but with a much higher average life, and so on.

[4] http://www.ptv-vision.com

Results				
High load				
Scenario	**RetrTup/min**	**Response Time (msec)**		
		Avg	**Min**	**Max**
Bologna	49432	76.54	12	423
Rome	79937	42.92	6	306
Toll Plaza	69543	69.05	9	623
Beijing	218791	70.73	6	515
Med load				
Scenario	**RetrTup/min**	**Response Time (msec)**		
		Avg	**Min**	**Max**
Bologna	24553	48.22	4	148
Rome	43897	27.04	2	107
Toll Plaza	35269	43.5	3	218
Beijing	112986	44.56	2	180
Low load				
Scenario	**RetrTup/min**	**Response Time (msec)**		
		Avg	**Min**	**Max**
Bologna	4854	6.03	1	19
Rome	8943	3.38	1	13
Toll Plaza	6873	5.44	1	27
Beijing	22861	5.57	1	23

Figure 4: Results: Response time and retrieved tuple rates for the three query workloads on the four city scenarios

On these four scenarios, we considered three different query workloads (high, medium and low load) each composed by a varying number of CQs and OTQs taken from 6 representative types (see Figure 3(b)). CQ1, CQ2 and CQ3 represent possible queries issued by location, traffic monitoring, and dynamic vehicle counting services, respectively. On the other hand, the OTQs could be possibly issued for an insurance check (OTQ1), for analysing a recent accident (OTQ2) or for a parking violation check (OTQ3). All OTQs are generated randomly with the mean frequency shown in table, while all the CQs run continuously over the full simulation time; therefore, we will have from 3000 (low workload) to 30000 (high workload) CQs concurrently running, besides all OTQs. All queries involve the retrieval of a combination of recent, historical and static information from the middleware (see the Description field in the figure); in order to answer most of the queries, further elaborations on the retrieved data are also required and performed, such as aggregates (e.g. CQ2 and CQ3) and joins (e.g. CQ3, OTQ2, OTQ3).

4.2 Middleware Performances

Figure 4 shows the average, minimum and maximum response time the middleware achieves for the three query workloads on the four city scenarios. The retrieved tuple rate, i.e. the number of tuples that have to be accessed and retrieved in order to produce the final answers, is also shown in order to quantify

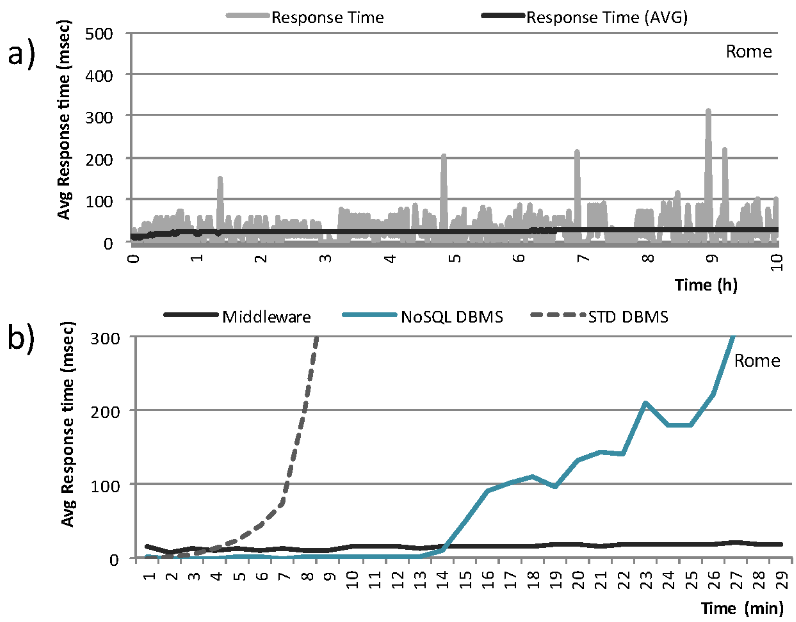

Figure 5: Results: (a) Average response time detail and (b) comparison over the 10-hours simulation time for Rome at high load

the complexity of each scenario at the RTDSM level. As we can see, each scenario is completed with very good results, i.e. all the data incoming from the vehicles is indexed and stored in the real-time data repository and all the query answers are delivered with very low response times. Even at high load, the system is able to cope well with the very demanding query frequency: the average response time is 76 milliseconds at most, with maximum times of 0.6 seconds or less. Also, the stability and scalability trend appears good: for instance, between medium and high (2x) load, response times are less than double, while the average response time keeps steady for all the duration of the simulation (see the trend depicted in Figure 5(a)).

4.3 Comparison with Other Data Management Systems

In Figure 5(b) we compared the middleware performances to those obtainable by a standard DBMS (PostgresSQL 9) and a NoSQL system (Apache Cassandra 1.1.16 [Apache Software Foundation, 2015]). While the DBMS is certainly very powerful in managing frequent and complex one-time queries on large amounts of data, the very high insertion rates coupled with the inability to natively support CQs (which have to be transformed in a sequence of

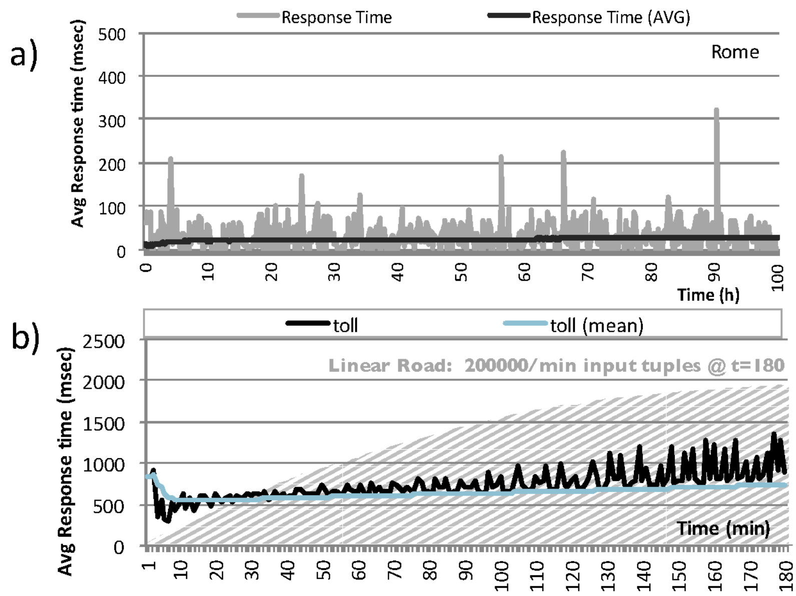

Figure 6: Scalability results: Average response time detail on (a) Extended 100 hours scenario and (b) synthetic Linear Road benchmark

OTQs), ultimately produce unacceptable results. On the other hand, also the NoSQL system in our setting seems to be unable to keep up with the rate required to support the scenarios. Please note that in our tests we focused on single-node implementations for all the considered systems; approaches relying on parallel and/or distributed architectures are orthogonal and would, of course, help in managing heavier workloads and data volumes. We will discuss this aspect in future work. Finally, we purposely not compared the middleware with state-of-the-art DSMSs [Abadi et. al., 2003, Arasu et al., 2006, Chen et al., 2000, Liarou et al., 2009] since these systems are not suitable for the purposes of the considered application scenarios, as discussed in Section 5.

4.4 Scalability Tests

In order to assess the scalability of our system, we performed two kinds of additional "stress" tests. First of all, we instantiated extended versions of the tests discussed in the previous sections, running up to 100 hours and thus with a consequently much larger amount of data to be indexed and queried. Figure 6(a) shows the results obtained for Rome in a 100 hours setting (this is representative of the results we obtained for all the other scenarios): as we can see, the average response time keeps steady, and its trend is completely comparable to the one shown in Figure 5(a), even if the tests take place on a 10 times longer time span. This confirms the general stability and good scalability of the middleware, with low average response time keeping below 100 ms.

The second kind of scalability tests we performed (Figure 6(b)) is based on the synthetic Linear Road benchmark [Arasu et al., 2004], which is a reference in the ITS and stream data management fields. The benchmark simulates a real time traffic management scenario where a multitude of cars move on multiple lanes of a virtual highway and pay dynamically calculated tolls. The position of cars have to be monitored and the tolls each of them needs to pay have to be computed in real-time; this computation should also be performed by dynamically identifying accidents. Four types of requests have to be satisfied in a strict response time deadline of 5 seconds: accident notifications, toll notifications, account balances and daily expenditures. The input data can be generated at varying levels of complexity (i.e., number of simulated expressways); in our tests we simulated two expressways. As it is, the standard benchmark is mainly devised for testing in-memory systems; therefore, we extend its requirements to go beyond recent data: while executing all the CQs and OTQs required for producing the output, we also demand the system to maintain the full stream history of position reports and to make such history always queryable. Figure 6(b) shows the obtained results for average response time for toll notification queries (other queries performed in a similar manner). Please note that the more time goes by, the more data enter the system (see shadowed area in figure), making this kind of test specifically significant for our scalability evaluation purposes: for instance, in the last minutes of the simulation we get to process more than 200000 reports per minute along with complex OTQs and CQs running. Also in this setting, the average response time is kept steadily below 0.7 seconds, well below the benchmark requirements, even with the additional requests on secondary memory storage.

5 Related Work

Building on our data management experiences in actual smart city scenarios [Mandreoli et al., 2010, Carafoli et al., 2012], and from the preliminary ideas presented in [Carafoli et al., 2013], we developed the middleware we presented and tested in detail in this paper. While we experienced its application in the ITS context, the middleware is actually general and can be employed in several areas (e.g., networking, retail industry, sensor networks). Its main advantage is the full decoupling between service logics and data management, supporting the development of any kind of service, and between data storage and data querying, making the management of real-time, historical, and static data an intuitive and unified experience, as is for static-only data in a standard DBMS.

To the authors' knowledge, no data management proposals exist which are explicitly devoted to the issues of ITSs, or which are able to offer all the advantages proposed.

To the state of the art, the core of a typical ITS is usually a Data Stream Management System (DSMS), a powerful architecture in processing huge amounts of real-time data. This is witnessed by DSMSs' successful application in specific real-world ITS contexts such as a real-time route planner in Lucerne [Özal et al., 2011] or a real-time traffic information management system in Stockholm [Biem et al., 2010]. In these systems, data is stored in main memory only, and it is kept there as long as it is needed in order to solve the continuous requests that are currently in execution; then, data flows out of the system. Due to these characteristics, DSMSs (e.g., [Abadi et. al., 2003, Arasu et al., 2006, Liarou et al., 2009, Chandrasekaran and Franklin, 2003, Chen et al., 2000]) as well as NewSQL systems (e.g. [Cetintemel et al., 2014]) do not comply with the context envisioned for the development of complex smart city services, where real-time data must be retained beyond their real-time processing to offer extended knowledge for various service purposes and also need to be joined with static data.

Although some DSMSs have moved towards mechanisms that permanently store part of the data [Abadi et. al., 2003, Arasu et al., 2006], in such systems any flow of stored tuples is actually a stream, and OTQs reduce to CQs. The severe overhead of converting permanently stored data to streams before being able to query it makes these systems unsuitable to support application scenarios like the ones considered in this paper. Moreover, they need to redesign from scratch a core of well-established DBMS functionalities that can not be reused as such in a DSMS architecture.

Other works (e.g., [Botan et al., 2009, Golab et al., 2009, Balazinska et al., 2007, Tufte et al., 2007, Chandrasekaran and Franklin, 2004]) propose two-layered solutions with a DSMS relying on the storage functionalities provided by a DBMS. However, in these dichotomic DSMS-DBMS solutions, persistent storage of streaming data is performed through external databases, thus making queries on historical data highly inefficient and ineffective because of the lack of continuous update capabilities by traditional DBMSs, as also discussed in [Franklin et al., 2009] and shown in section 4.

Other systems, like key-value and column-oriented NoSQL DBs [Han et al., 2011, Apache Software Foundation, 2015], although supporting high-performance read/write workloads, hardly fit complex and general scenarios like the one considered in this paper. They are primarily designed to either efficiently support primary-key-based operations or to accelerate aggregation operations in data warehouse contexts, but in more general and query-intensive scenarios their performance degrades significantly, as shown by our experiments in Sect. 4.

Some commercial DBMS vendors have recently invested in the development of streaming systems [IBM, 2015, PipelineDB, 2015, TIBCO Software, 2015, StreamInsight, 2014, Witkowski et al., 2007]. For instance, InfoSphere Streams [IBM, 2015], StreamBase [TIBCO Software, 2015], StreamInsight [StreamInsight, 2014], and Oracle [Witkowski et al., 2007] offer powerful platforms for the development of complex event processing and real-time analytical applications. These systems enable the access to live and historical data in a streaming fashion, in that databases are considered as event sources and/or targets. Thus, they implement the dichotomic DSMS-DBMS vision having the limitations discussed above. PipelineDB [PipelineDB, 2015] is an open-source relational database that runs SQL queries continuously on streams, incrementally storing results in tables. The main difference with our approach is that in PipelineDB continuous queries are intended to reduce the cardinality of the streaming data to be stored. To this end, queries must be known a priori.

On the other hand, by following an orthogonal approach, we adopt an all-embracing perspective, in that we aim at proving the feasibility of making both real-time, past streamed, and traditional static data coexist under a common framework, by offering transparent data storage and query capabilities. The results obtained by the evaluation of the middleware introduced in this paper show that this perspective is practicable and offers satisfactory results. Further, thanks to its generality and versatility, the proposed architectural approach may lead to many benefits in the design and management of ITSs as well as of various hybrid data-centric applications.

6 Conclusions and Future Work

In this paper we have presented a Data Management Middleware that offers implementation facilities for the development of smart city services that require access to both real-time and historical/static data, and need to execute both continuous queries (CQs) and one-time queries (OTQs). While being employable in various domains, as a proof of its good performances, we have shown very promising results obtained by its application in an ITS scenario on a variety of traffic conditions.

In this application domain, several research directions could be further explored. For instance, leveraging on our previous work on the employment of vehicle-to-infrastructure (V2I) data reduction techniques for guaranteeing sustainable workloads in data-intensive scenarios like ITSs [Carafoli et al., 2012], a possible future direction would be to blend these techniques with vehicle-to-vehicle (V2V) counterparts (e.g. [Ilarri et al., 2015]), within a synergistic combination of techniques.

As a final note, the prototype of the proposed middleware currently runs on a single-node architecture. In the future, we plan to investigate the opportunity of implementing its design principles on parallel and/or distributed architectures such as the recently proposed Big Data Lambda Architecture [Marz and Warren, 2015] in order to support heavier workloads and data volumes.

Also, further investigations will be devoted to other open research issues, such as query optimization and query reuse.

References

[Abadi et. al., 2003] Abadi et. al., D. (2003). Aurora: a new model and architecture for data stream management. *VLDBJ*, 12(2):120–139.

[Apache Software Foundation, 2015] Apache Software Foundation (2015). The Apache Cassandra Project. http://cassandra.apache.org/.

[Arasu et al., 2006] Arasu, A., Babu, S., and Widom, J. (2006). The CQL continuous query language: semantic foundations and query execution". *VLDB J.*, 15(2):121–142.

[Arasu et al., 2004] Arasu, A., Cherniack, M., Galvez, E., Maier, D., Maskey, A., Ryvkina, E., Stonebraker, M., and Tibbetts, R. (2004). Linear Road: A Stream Data Management Benchmark. In *Proc. of VLDB*, pages 480–491.

[Balazinska et al., 2007] Balazinska, M., Kwon, Y., Kuchta, N., and Lee, D. (2007). Moirae: History-Enhanced Monitoring. In *Proc. of CIDR*, pages 375–386.

[Biem et al., 2010] Biem, A., Bouillet, E., Feng, H., Ranganathan, A., Riabov, A., Verscheure, O., Koutsopoulos, H., Rahmani, M., and Güç, B. (2010). Real-time traffic information management using stream computing. *IEEE Data Eng. Bull.*, 33(2):64–68.

[Botan et al., 2009] Botan, I., Alonso, G., Fischer, P., Kossmann, D., and Tatbul, N. (2009). Flexible and scalable storage management for data-intensive stream processing. In *Proc. of EDBT*, pages 934–945.

[Carafoli et al., 2012] Carafoli, L., Mandreoli, F., Martoglia, R., and Penzo, W. (2012). Evaluation of data reduction techniques for vehicle to infrastructure communication saving purposes. In *IDEAS*, pages 61–70.

[Carafoli et al., 2013] Carafoli, L., Mandreoli, F., Martoglia, R., and Penzo, W. (2013). A Framework for ITS Data Management in a Smart City Scenario. In *SMARTGREENS*, pages 215–221.

[Carafoli et al., 2016] Carafoli, L., Mandreoli, F., Martoglia, R., and Penzo, W. (2016). Streaming Tables: Native Support to Streaming Data in DBMSs. Under review.

[Cetintemel et al., 2014] Cetintemel, U., Du, J., Kraska, T., Madden, S., Maier, D., Meehan, J., Pavlo, A., Stonebraker, M., Sutherland, E., Tatbul, N., Tufte, K., Wang, H., and Zdonik, S. (2014). S-Store: A Streaming NewSQL System for Big Velocity Applications. *Proc. VLDB Endow.*, 7(13):1633–1636.

[Chandrasekaran and Franklin, 2003] Chandrasekaran, S. and Franklin, M. (2003). PSoup: a system for streaming queries over streaming data. *VLDB J.*, 12(2):140–156.

[Chandrasekaran and Franklin, 2004] Chandrasekaran, S. and Franklin, M. (2004). Remembrance of Streams Past: Overload-Sensitive Management of Archived Streams. In *Proc. of VLDB*, pages 348–359.

[Chen et al., 2000] Chen, J., DeWitt, D., Tian, F., and Wang, Y. (2000). NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proc. of SIGMOD*, pages 379–390.

[Delot et al., 2013] Delot, T., Ilarri, S., Lecomte, S., and Cenerario, N. (2013). Sharing with Caution: Managing Parking Spaces in Vehicular Networks. *Mobile Information Systems*, 9(1):69–98.

[Echelon Corp., 2013] Echelon Corp. (2013). Smart Street Lighting. https://www.echelon.com/applications/street-lighting/.

[European Commission, 2014] European Commission (2014). The Digital Agenda for Europe. http://ec.europa.eu/digital-agenda/.

[Franklin et al., 2009] Franklin, M., Krishnamurthy, S., Conway, N., Li, A., Russakovsky, A., and Thombre, N. (2009). Continuous Analytics: Rethinking Query Processing in a Network-Effect World. In *Proc. of CIDR*.

[Golab et al., 2009] Golab, L., Johnson, T., Seidel, J., and Shkapenyuk, V. (2009). Stream warehousing with DataDepot. In *Proc. of SIGMOD*, pages 847–854.

[Han et al., 2011] Han, J., E, H., Le, G., and Du, J. (2011). Survey on NoSQL Database. In *Proc. of Int. Conf. on Perv. Comp. and App. (ICPCA)*, pages 363–366.

[IBM, 2015] IBM (2015). InfoSphere Streams. http://www-03.ibm.com/software/products/us/en/infosphere-streams.

[Ilarri et al., 2015] Ilarri, S., Delot, T., and Trillo, R. (2015). A data management perspective on vehicular networks. *Communications Surveys Tutorials, IEEE*, PP(99):1–1.

[Kanagal and Deshpande, 2008] Kanagal, B. and Deshpande, A. (2008). Online filtering, smoothing and probabilistic modeling of streaming data. In *IEEE 24th International Conference on Data Engineering (ICDE'08)*, pages 1160–1169.

[Liarou et al., 2009] Liarou, E., Goncalves, R., and Idreos, S. (2009). Exploiting the power of relational databases for efficient stream processing. In *Proc. of EDBT*, pages 323–334.

[Mandreoli et al., 2010] Mandreoli, F., Martoglia, R., Penzo, W., and Sassatelli, S. (2010). Data management issues for intelligent transportation systems. In *Proc. of SEBD*, pages 198–209.

[Marz and Warren, 2015] Marz, N. and Warren, J. (2015). *Big Data. Principles and best practices of scalable realtime data systems.* Manning Publications.

[NIMA, 2000] NIMA (2000). Department of defense world geodetic system 1984, its definition and relationships with local geodetic systems. Technical report, NIMA Technical Report TR8350.2.

[NOAA, 2008] NOAA (2008). Nad 83 (nsrs2007) national readjustment final report. Technical report, NOAA Technical Report NOS NGS 60.

[Özal et al., 2011] Özal, A., Ranganathan, A., and Tatbul, N. (2011). Real-time route planning with stream processing systems: a case study for the city of lucerne. In *Proceedings of the 2nd ACM SIGSPATIAL International Workshop on GeoStreaming*, IWGS '11, pages 21–28.

[PipelineDB, 2015] PipelineDB (2015). PipelineDB. https://www.pipelinedb.com.

[SFMTA, 2014] SFMTA (2014). San Francisco Park. http://sfpark.org/.

[Snodgrass, 1995] Snodgrass, R., editor (1995). *The TSQL2 Temporal Query Language.* Kluwer.

[StreamInsight, 2014] StreamInsight (2014). StreamInsight. http://msdn.microsoft.com/en-us/sqlserver/ee476990.aspx.

[TIBCO Software, 2015] TIBCO Software (2015). StreamBase. http://www.streambase.com.

[Tufte et al., 2007] Tufte, K., Li, J., Maier, D., Papadimos, V., Bertini, R., and Rucker, J. (2007). Travel time estimation using NiagaraST and latte. In *Proc. of SIGMOD*, pages 1091–1093.

[UCL, 2013] UCL (2013). London City Dashboard. http://citydashboard.org/london/.

[Witkowski et al., 2007] Witkowski, A., Bellamkonda, S., Li, H., Liang, V., Sheng, L., Smith, W., Subramanian, S., Terry, J., and Yu, T. (2007). Continuous Queries in Oracle. In *Proc. of VLDB*, pages 1173–1184.