http://lib.ulg.ac.be http://matheo.ulg.ac.be

Towards a distributed implementation of a data extraction system for monitoring the processing of video clips

Auteur : Vasbinder, Thomas
Promoteur(s) : Wolper, Pierre
Faculté : Faculté des Sciences appliquées
Diplôme : Master en ingénieur civil en informatique, à finalité approfondie
Année académique : 2015-2016
URI/URL : http://hdl.handle.net/2268.2/1306

Avertissement à l'attention des usagers :

Tous les documents placés en accès ouvert sur le site le site MatheO sont protégés par le droit d'auteur. Conformément aux principes énoncés par la "Budapest Open Access Initiative"(BOAI, 2002), l'utilisateur du site peut lire, télécharger, copier, transmettre, imprimer, chercher ou faire un lien vers le texte intégral de ces documents, les disséquer pour les indexer, s'en servir de données pour un logiciel, ou s'en servir à toute autre fin légale (ou prévue par la réglementation relative au droit d'auteur). Toute utilisation du document à des fins commerciales est strictement interdite.

Par ailleurs, l'utilisateur s'engage à respecter les droits moraux de l'auteur, principalement le droit à l'intégrité de l'oeuvre et le droit de paternité et ce dans toute utilisation que l'utilisateur entreprend. Ainsi, à titre d'exemple, lorsqu'il reproduira un document par extrait ou dans son intégralité, l'utilisateur citera de manière complète les sources telles que mentionnées ci-dessus. Toute utilisation non explicitement autorisée ci-avant (telle que par exemple, la modification du document ou son résumé) nécessite l'autorisation préalable et expresse des auteurs ou de leurs ayants droit.

Towards a distributed implementation of a data extraction system for monitoring the processing of video clips

Vasbinder Thomas

Faculty of Applied Sciences

University of Liège

Final project submitted for obtaining the Master's degree in computer science and engineering.

Academic year 2015-2016

Acknowledgements

I would like to express my gratitude to my supervisors Louis Latour and Prof. Pierre Wolper for the useful advice, remarks and engagement through the learning process of this master thesis. Also, I would like to thank all the Xsquare team members for their hospitality and their help.

Abstract

Handling distributed systems which must be highly available is a challenge for many companies. One of the issues is to maintain data consistency while maintaining the service highly available. In this work, we review the data handling of a service in order to be able to run multiple instances of that service, enabling load balancing and redundancy,

This work has been carried out at EVS Broadcast Equipment, which is specialised in live video broadcast, providing among others the XT server family of products. With the aim to provide a better experience when managing remote production, EVS has recently developed an application, called Multi-Monitoring, displaying the real-time status of the operations happening on remote sites.

The starting point for this work was to review the data handling of the Multi-Monitoring application and to remove any barrier preventing the concurrent execution of multiple instances of the service while maintaining or improving the overall performance, and in particular, to review the caching infrastructure.

Table of contents

List of figures **xi**

List of tables **xiii**

Abbreviations **xv**

1 Introduction **1**

1.1 Motivation 1

1.2 Contributions 2

1.3 Outline of the Thesis 3

2 Description of the MultiMonitoring application **5**

2.1 Context 5

2.1.1 Video processing jobs 5

2.1.2 Job orchestrator: Xsquare 6

2.1.3 Outside broadcast trucks 6

2.1.4 MultiMonitoring application 7

2.2 Representational State Transfer (REST) 11

2.2.1 Introduction 11

2.2.2 MultiMonitoring resources 13

2.2.3 Query Language 13

2.3 MultiMonitoring Architecture 15

2.3.1 Server architecture 15

2.3.2 Agent architecture 17

2.3.3 Database 18

2.3.4 Cache 19

2.3.5 Object-Relational Mapper: Entity Framework 19

2.4 Typical MultiMonitoring setup 20

2.5 Summary 21

3 Performance analysis of the cache 23
3.1 Testing environment 23
3.2 Database loading 23
3.3 RAM usage 24
3.4 Querying resources: LINQ and LINQ to Objects 24
3.5 Performance analysis 25
  3.5.1 Query execution time 25
  3.5.2 Number of supported clients 28
3.6 Conclusions 29

4 Performance analysis of the database 31
4.1 Querying resources: LINQ to Entities 31
4.2 Page caching 31
4.3 Performance analysis 33
  4.3.1 Sorting performance 35
  4.3.2 Filtering performance 37
4.4 Comparison with the cache 40
  4.4.1 Removing the cache 41
4.5 Conclusions 41

5 Open Data Protocol 43
5.1 Presentation 43
  5.1.1 OData .NET library 45
5.2 Using OData with the MultiMonitoring application 45
  5.2.1 OData and MultiMonitoring server 45
  5.2.2 OData and MultiMonitoring client 49
5.3 Performance analysis 49
  5.3.1 Filtering 49
  5.3.2 Count query option 50
  5.3.3 Sorting 51
5.4 Use of database indexes 52
  5.4.1 Introduction 52
  5.4.2 Filtering 53
  5.4.3 Count query option 53
  5.4.4 Sorting 53
  5.4.5 Sorting and filtering 54
  5.4.6 Full-text index 54

5.5 Indexing strategy 56
5.5.1 Query performance summary 56
5.5.2 Indexed columns 56
5.6 OData limitations 57
5.7 Number of supported clients 58
5.7.1 Count queries and pagination system 59
5.8 Conclusions 59

6 Conclusion **61**

References **63**

List of figures

2.1 Video processing job sent by a client application to a processing device. . . 6
2.2 Xsquare, a job orchestrator distributing jobs initiated by client applications to processing devices according to available resources. 7
2.3 Xsquares present in broadcast centre and OB trucks. 8
2.4 MultiMonitoring high-level architecture. 9
2.5 Job monitoring window. 9
2.6 Architecture of the MultiMonitoring server. 16
2.7 Data models used at the MultiMonitoring server. 17
2.8 Architecture of the MultiMonitoring agent. 18
2.9 Entity Data Model: entities and relationships [3]. 20
2.10 Entity Data Model: entity container, entity sets and properties [3]. 21

3.1 Server RAM usage for several cache sizes. 24
3.2 Query execution time for increasing number of sorts. 27

4.1 Evolution of SQL Server performance counters during read queries. 32
4.2 Simplified query execution plan generated for a query on `Jobs` and `Destinations` tables, linked by a join. 34
4.3 SQL Server sort operators 35

5.1 Components of the OData technology [3]. 44
5.2 OData query translation from URI to SQL. 45
5.3 MultiMonitoring server: link between CRUD operations and data models. . 46

List of tables

2.1 Job information displayed at the client and allowed operations. 10

3.1 Mean times spent on the cache, in the middleware and between client and server, in ms. 26
3.2 Maximum number of clients for several queries. 29

4.1 Mean times spent on the database, in the controller, in the middleware and between client and server, in ms. 33
4.2 Execution time (ms) at the database for several sort combinations (LINQ to Entities). 37
4.3 Execution time (ms) at the database for several filtering queries (LINQ to Entities), type of filtered columns and number of jobs (thousands) satisfying the filtering condition. 38
4.4 Execution time (ms) at the database for queries containing several filter combinations followed by a sort on MeanFrameRate, type of filtered columns and number of jobs (thousands) satisfying the filtering condition. 39

5.1 Execution time (ms) at the database for several count queries, type of filtered columns and number of jobs (thousands) satisfying the filtering condition. . 50
5.2 Execution time (ms) at the database for several sort combinations (LINQ to Entities and OData). 51
5.3 Execution time (ms) at the database (with indexes) for several count queries, type of filtered columns and number of jobs (thousands) satisfying the filtering condition. 53
5.4 Execution time (ms) at the database (with indexes) for several sort queries. . 54
5.5 Execution time (ms) at the database (with indexes) for queries containing several filter combinations followed by a sort on MeanFrameRate, type of filtered columns and number of jobs (thousands) satisfying the filtering condition. 55

5.6 Maximum number of clients for several queries using OData. 58
5.7 Maximum number of clients for several queries using OData, with reduced counts frequency. 59

Abbreviations

Roman Symbols

EDM	Entity Data Model
EF	Entity Framework
HATEOAS	Hypermedia As The Engine Of Application State
JSON	JavaScript Object Notation
LINQ	Language Integrated Query
OData	Open Data Protocol
ORM	Object-Relational Mapper
RDBMS	Relational Database Management System
REST	Representational State Transfer
URI	Uniform Resource Identifier

Chapter 1

Introduction

This work has been carried out at EVS Broadcast Equipment, which is specialised in live video broadcast, providing among others the XT server family of products. In addition to their production servers, EVS provides also production and content management tools (scheduler, logger, clipper, playlist management, media manager and browser) and live editing tools (slow motion, graphic insertions, etc). Those products are used in several fields such as sport events, TV shows and news production.

Some EVS products process video clips (rewrap, transcoding) and transfer them from one location to another. An application allows the user to monitor the real-time status and the progress of those operations (called jobs). However, the job monitoring is limited to one production site. With the aim to provide a better experience when managing remote production, EVS has recently developed an application, called Multi-Monitoring, displaying the jobs happening on several remote sites.

In this work, we review the data handling of the Multi-Monitoring application in order to be able to run multiple instances of that service, enabling load balancing and redundancy. However, handling distributed systems which must be highly available is a challenge for many companies. One of the issue is to maintain data consistency while maintaining the service highly available.

1.1 Motivation

The starting point for this work was to review the data handling of the Multi-Monitoring application and to remove any barrier preventing the concurrent execution of multiple instances of the service while maintaining or improving the overall performance, and in particular, to review the caching infrastructure.

The caching infrastructure was not really sophisticated and consisted in a set of in-memory lists corresponding to the most recent jobs as well as the information related to the remote site application.

One of the issues was that there was no mechanism designed for keeping the database and the cache synchronised, the service always modified both of them. So, if another service were to rely on the same database, the other services would not be informed of the changes. Another area to investigate was the data structure used for the cache. This implementation led to poor performance. Indeed, requests were executed by travelling entirely and several times the list of jobs. This involved a high CPU consumption that limited the number of requests that could be handled and thus the number of clients that could interact with the server. Moreover, the lists used offered no mechanism for improving performance of the queries performed, like indexes. Finally, storing the database content in-memory imposed a burden on the RAM usage.

While the above limitations were known to the team, the team had never really assessed the benefit of the caching infrastructure, relying on the intuition that keeping data in-memory could only improve the query performance.

The caching strategy is inherited from a cache used in another application developed by EVS. Though, this strategy isn't really suited to the studied application. The first flaw of this caching strategy is that some jobs are stored in the database but are never queried. Indeed, the cache contains only the most recent jobs and its capacity is constant. Since new jobs are constantly added (in the cache and in the database), a purge mechanism is used to delete older jobs from the cache. Those jobs are thus present only in the database after the purge. Since requests are executed on the cache, the older jobs that are only in the database are useless because they are never sent to clients anymore. Moreover, the Multi-Monitoring server receives up to 20 times more job insertions than the other application from which the caching strategy is inherited. Some recently processed jobs could be not present in the cache if its capacity isn't large enough. However, choosing a fixed capacity is hard because the jobs insertion rate is not constant. Thus, a solution to this problem should be found.

1.2 Contributions

This work first provides an analysis of the query-handling performance of the existing Multi-Monitoring service. This analysis identifies the cache as a bottleneck and shows the performance heavily depends on the query being performed. The study also compares the performance of the system when the complete caching infrastructure is removed. This shows that accessing directly the database is both more efficient and more predictable. Based on this, we implemented and tested a new version of the service not relying on an explicit caching mechanism. Thus, the concurrent execution of multiple instances of this new version is now possible. Removing the cache also solves the caching strategy problem linked to the fixed cache capacity. With the removal of the cache and the use of OData (tool allowing the creation and consumption of queryable RESTful APIs), the code of the server is now shorter and simplified. Finally, we also list some limitations encountered when using the OData library for Microsoft .NET. Since we could not find those mentioned in the existing literature, this list will help people to use OData in an adapted way.

1.3 Outline of the Thesis

Chapter 2 presents the Multi-Monitoring application. It first explains the context in which the application is used. Then, the REST architectural style is briefly presented. Next, the architecture and the components of the Multi-Monitoring application are explained. Finally, we define a typical configuration used for the tests.

Chapter 3 analyses the performance of the initial application. It first explains how the database is filled with test data and measures the memory used by the Multi-Monitoring server. Then, it presents how the queries contained in URIs are translated to be executed on the cache. A performance analysis is conducted to discover the cache behaviour and obtain results to compare with alternative solutions.

Chapter 4 studies the performance of the database. It first explains how the database can be queried using the ORM. Then, it analyses the caching mechanism provided by the database and measures its performance when executing queries. The obtained results are then compared with those of the cache.

Chapter 5 presents a tool called OData used for its query capabilities. It explains how using this tool in the Multi-Monitoring server to query the database, It explains also how the cache is removed and the simplifications that are obtained thanks to that removal. The performance of OData are then measured and improved using database indexes.

Chapter 2

Description of the MultiMonitoring application

This chapter aims to explain the context of use of the MultiMonitoring application, its features, its architecture, its components and provides a typical usage configuration.

2.1 Context

In this section, we first describe the data on which the MultiMonitoring application works and explain the origin of those data. Then, we explain the goal of the MultiMonitoring application and present its components.

2.1.1 Video processing jobs

Video clips are stored in video servers (XT servers). For example, video clips can come from HD cameras located in a football stadium. EVS provides production and content management applications that work on those video clips: IPDirector and Xedio. Using those client applications, users can perform operations on video clips (rewrap, transcoding, copy,...). The video processing operations are not directly performed by the client applications but by processing devices called XTAccess, as illustrated in Figure 2.1. Indeed, a client application must send to an XTAccess the information describing the operation it want to apply on a video clip. That information is represented by a *video processing job*, which consists of

- a *source material* (video clip) selected by the client application.
- an *operation* (copy, rewrap, transcoding, restore, grab, etc) executed on the source material.

- a *destination* that stores the result of the operation on the source material. The destination corresponds to an XT server.

Jobs are thus initiated by client applications which provide the source video clip that will undergo the operation. The jobs are then sent to processing devices that process the jobs that they receive.

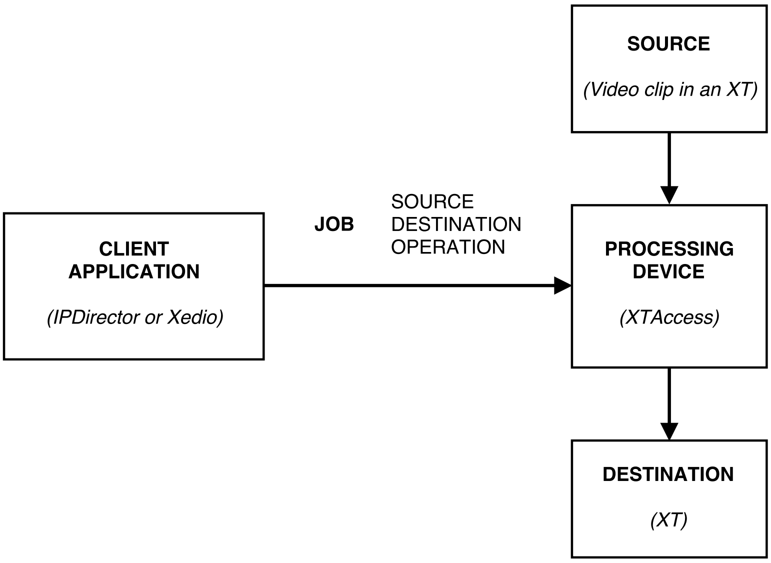


Fig. 2.1 Video processing job sent by a client application to a processing device.

2.1.2 Job orchestrator: Xsquare

A configuration can contain several client applications that initiate a lot of jobs and several processing devices (XTAccess). A job orchestrator, called Xsquare [7], is used to centrally manage resources (CPU/RAM/network) used by the processing devices. It distributes the jobs to processing devices according to available resources, as illustrated in Figure 2.2.

In addition of being a central job orchestrator, an Xsquare provides also a monitoring tool used to monitor and manage all the jobs handled by that Xsquare. While the jobs are processed, an Xsquare sends to client applications notifications about the progress of their jobs.

2.1.3 Outside broadcast trucks

Outside broadcast (OB) *trucks* are used for the production of television programmes that cover sport and news events. An OB truck is a kind of mobile studio equipped with several XT

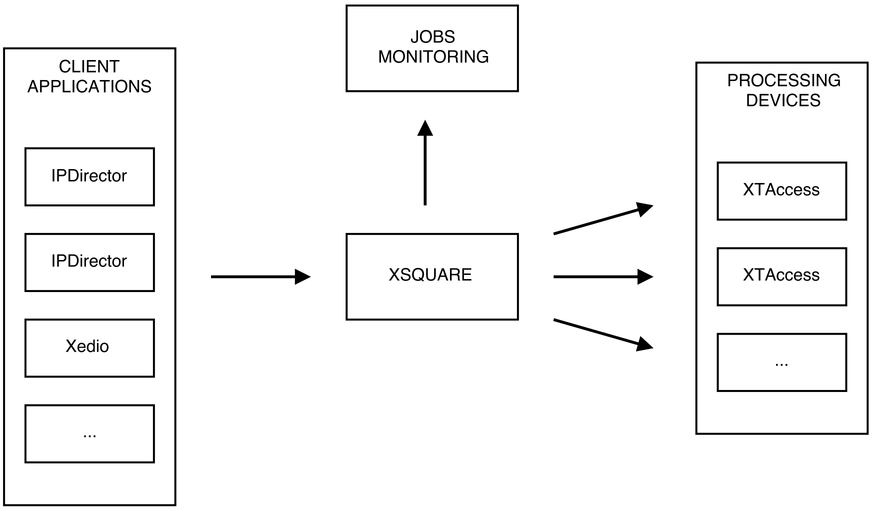


Fig. 2.2 Xsquare, a job orchestrator distributing jobs initiated by client applications to processing devices according to available resources.

servers, one or two Xsquares, an XFile (file archiving tool) and sometimes an IPDirector [5, 6].

2.1.4 MultiMonitoring application

A broadcast centre is usually connected to several OB trucks through a VPN, as represented in Figure 2.3. Like the trucks, the broadcast centre also contains one or several Xsquares.

Each Xsquare provides a monitoring tool that displays jobs managed by that Xsquare. However, there is no way to monitor remotely the jobs managed by the Xsquares present in all the OB trucks from the broadcast centre. As its name suggests, the goal of the MultiMonitoring application is to centralise the jobs handled by Xsquares from all OB trucks on the same monitoring screen. The notion of *job monitor* is used to represent the job monitoring activity of a remote Xsquare.

MultiMonitoring agent and server

The MultiMonitoring application relies on two entities, represented in grey in Figure 2.3:

- The *MultiMonitoring server*, located in the broadcast centre, that stores information about the jobs managed by all Xsquares of the setup.
- The *MultiMonitoring agent* located in each truck.

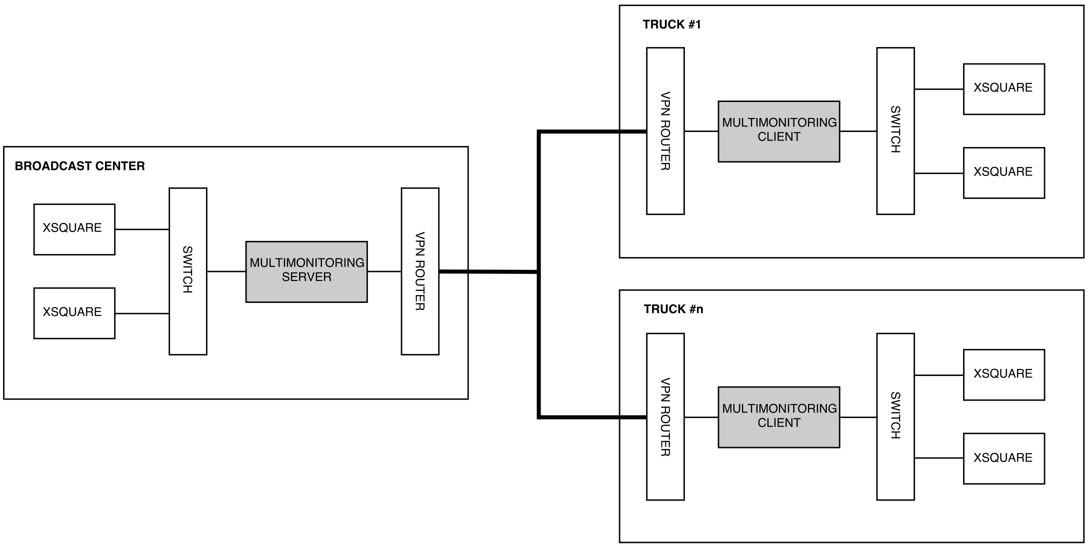

BROADCAST CENTER
XSQUARE
XSQUARE
SWITCH
MULTIMONITORING SERVER
VPN ROUTER
TRUCK #1
VPN ROUTER
MULTIMONITORING CLIENT
SWITCH
XSQUARE
XSQUARE
TRUCK #n
VPN ROUTER
MULTIMONITORING CLIENT
SWITCH
XSQUARE
XSQUARE

Fig. 2.3 Xsquares present in broadcast centre and OB trucks.

MultiMonitoring agents are primarily used as proxies. In that way, the Xsquares are not visible from outside the trucks. The agents are also used to avoid disturbing the Xsquares by the remote monitoring. Indeed, an agent is responsible for managing and sending notifications instead of the Xsquare.

MultiMonitoring high-level architecture

MultiMonitoring agents send notifications about job progression to the MultiMonitoring server that stores this information to expose it to clients. This high-level architecture is illustrated in Figure 2.4. The remainder of this text is based on this organisation in agents, server and clients.

MultiMonitoring client

The MultiMonitoring client consists in a web browser application that queries each second the MultiMonitoring server to obtain the jobs corresponding to a particular filter. The web page returned by the MultiMonitoring server to the web client is displayed at Figure 2.5.

The main element of the job monitoring window is the job grid that displays all the operations processed by the processing devices, i.e. the jobs. The columns of the grid correspond to metadata about the jobs and each line of the grid corresponds to a job.

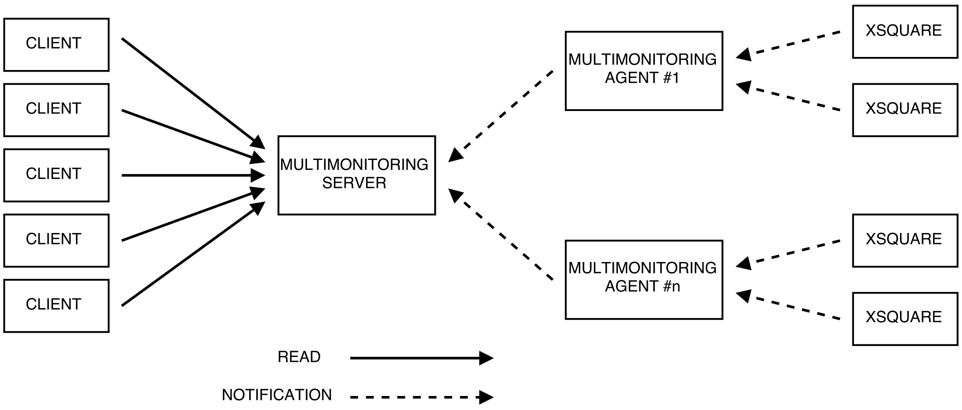
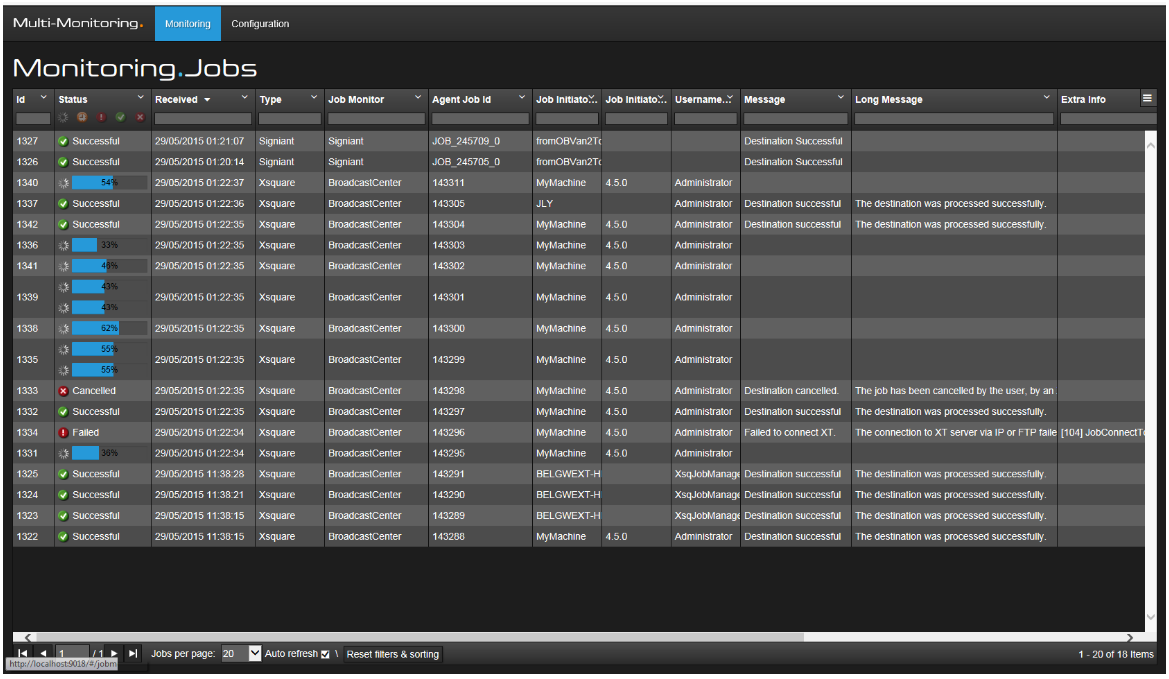


Fig. 2.4 MultiMonitoring high-level architecture.



Id	Status	Received	Type	Job Monitor	Agent Job Id	Job Initiato...	Job Initiato...	Username...	Message	Long Message	Extra Info
1327	✓ Successful	29/05/2015 01:21:07	Signiant	Signiant	JOB_245709_0	fromOBVan2Tc			Destination Successful		
1326	✓ Successful	29/05/2015 01:20:14	Signiant	Signiant	JOB_245705_0	fromOBVan2Tc			Destination Successful		
1340	54%	29/05/2015 01:22:37	Xsquare	BroadcastCenter	143311	MyMachine	4.5.0	Administrator			
1337	✓ Successful	29/05/2015 01:22:36	Xsquare	BroadcastCenter	143305	JLY		Administrator	Destination successful	The destination was processed successfully.	
1342	✓ Successful	29/05/2015 01:22:35	Xsquare	BroadcastCenter	143304	MyMachine	4.5.0	Administrator	Destination successful	The destination was processed successfully.	
1336	33%	29/05/2015 01:22:35	Xsquare	BroadcastCenter	143303	MyMachine	4.5.0	Administrator			
1341	46%	29/05/2015 01:22:35	Xsquare	BroadcastCenter	143302	MyMachine	4.5.0	Administrator			
1339	43% 43%	29/05/2015 01:22:35	Xsquare	BroadcastCenter	143301	MyMachine	4.5.0	Administrator			
1338	62%	29/05/2015 01:22:35	Xsquare	BroadcastCenter	143300	MyMachine	4.5.0	Administrator			
1335	55% 55%	29/05/2015 01:22:35	Xsquare	BroadcastCenter	143299	MyMachine	4.5.0	Administrator			
1333	✗ Cancelled	29/05/2015 01:22:35	Xsquare	BroadcastCenter	143298	MyMachine	4.5.0	Administrator	Destination cancelled.	The job has been cancelled by the user, by an	
1332	✓ Successful	29/05/2015 01:22:35	Xsquare	BroadcastCenter	143297	MyMachine	4.5.0	Administrator	Destination successful	The destination was processed successfully.	
1334	! Failed	29/05/2015 01:22:34	Xsquare	BroadcastCenter	143296	MyMachine	4.5.0	Administrator	Failed to connect XT.	The connection to XT server via IP or FTP faile	[104] JobConnectTc
1331	36%	29/05/2015 01:22:34	Xsquare	BroadcastCenter	143295	MyMachine	4.5.0	Administrator			
1325	✓ Successful	29/05/2015 11:38:28	Xsquare	BroadcastCenter	143291	BELGWEXT-H		XsqJobManage	Destination successful	The destination was processed successfully.	
1324	✓ Successful	29/05/2015 11:38:21	Xsquare	BroadcastCenter	143290	BELGWEXT-H		XsqJobManage	Destination successful	The destination was processed successfully.	
1323	✓ Successful	29/05/2015 11:38:15	Xsquare	BroadcastCenter	143289	BELGWEXT-H		XsqJobManage	Destination successful	The destination was processed successfully.	
1322	✓ Successful	29/05/2015 11:38:15	Xsquare	BroadcastCenter	143288	MyMachine	4.5.0	Administrator	Destination successful	The destination was processed successfully.	

1 / 1 Jobs per page: 20 Auto refresh ☑ Reset filters & sorting 1 - 20 of 18 Items

http://localhost:9018/#/jobm

Fig. 2.5 Job monitoring window.

The grid displays filters that can be used by the user to select the jobs that are displayed in the job monitoring window. Rows can be sorted based on the values of a column of the grid and filtered by selecting a predefined filter or by typing a text to search.

The second part of the job monitoring window allows to:

- Choose the number of jobs displayed on the screen.
- Navigate between pages of jobs (pagination system).

- Enable/disable the one second auto refresh of the job grid.
- Reset all the filters and sortings.

Table 2.1 describes the information displayed for each job and the corresponding allowed operations (filter or sort). We can notice that destination fields can't be sorted. Indeed, there can be several destinations and thus several sort keys for one job, which is ambiguous.

Field name	**Description**	**Sort**	**Filter**
Id	Unique job identifier	yes	yes
Agent Job Id	Unique job identifier for the corresponding agent	no	yes
Status	Status of the job (in progress, scheduled, failed, successful or cancelled	yes	yes
Type	Job monitor type (Xsquare)	yes	yes
Reception Time	Date and time when Xsquare has received the job request	yes	yes
Start Time	Date and time when Xsquare has started the job request	yes	yes
End Time	Date and time when Xsquare has finished the job request	yes	yes
Initiator	Xsquare initiator	yes	yes
Username	Xsquare username	yes	yes
Source Agent	Source agent	yes	yes
Source Path	Location of the source material (path or IP address)	yes	yes
Source Filename	Name of the source material	yes	yes
Source Clipname	Name of the source material	yes	yes
MBytes/s	Transfer rate in megabytes per second	yes	yes
Frames/s	Transfer rate in frames per second	yes	yes
Job Monitor Name	Job monitor nickname	no	yes
Destination Agent	Destination agent	no	yes
Destination Path	Location where the job output will be stored	no	yes
Destination Filename	Name of the destination material	no	yes
Destination Clipname	Name of the destination material	no	yes
Message	Message giving information on the outcome of the job	no	yes

Table 2.1 Job information displayed at the client and allowed operations.

2.2 Representational State Transfer (REST)

In the previous section, we explained that the MultiMonitoring server exposes jobs that can be queried by web clients. The server defines an API that conforms to the constraints defined by REST. In this section, we describe what is REST and how it can be used in the MultiMonitoring application.

2.2.1 Introduction

Representational State Transfer (REST) is an architectural style for distributed hypermedia systems defined by Roy Thomas Fielding in chapter 5 of his PhD dissertation "Architectural Styles and the Design of Network-based Software Architectures" [8].

An architectural style consists of a set of architectural constraints. Those constraints induce properties for the resulting architecture. Some properties induced by REST constraints are desirable such as performance, scalability, simplicity, visibility, interoperability and reliability. However, other induced properties lead to design trade-offs like reduction of efficiency.

An application that conforms to REST constraints is called «RESTful». REST is independent of a particular communication protocol. However, most RESTful architectures use HTTP as the communication protocol.

The six constraints that define the REST architectural style are:

1. **Client-Server**
   The client-server constraint consists in separating the user interface (client) from the data storage (server). Client and server are thus distinct components.

2. **Statelessness**
   Statelessness is about the server. The server must not store information about the state of the client. Thus, each request sent by the client to the server must contain all of the information necessary to understand and service the request. If a session state is needed, it must be stored on the client and given to the server when needed.

3. **Cacheability**
   The data in a response to a request must be implicitly or explicitly labelled as cacheable or non-cacheable. The client cache can then reuse the data of a cacheable response for new requests.

4. **Uniform interface**
   This constraint is about the interface between clients and servers. Its goal is to decouple

clients and servers and allow them to evolve independently. This constraint consists of four sub-constraints:

A. Identification of resources

   The *resource* is the key abstraction of information in REST and corresponds to any information that can be named such as a document, an image, a collection of resources, etc. This first sub-constraint states that each resource must be given a unique and stable identifier. *Uniform Resource Identifier* (URI) are used to identify resources when using HTTP.

B. Manipulation of resources through representations

   Clients and servers can store data in several ways. A *representation* is a sequence of bytes that captures the current or intended state of a resource. Representations are used to exchange information between the client and the server and is thus contained in requests and responses.

   In HTTP, resources are manipulated using a set of HTTP verbs. The number of verbs is limited and describes the uniform interface. HTTP verbs map to CRUD operations in the following way:

   - Create: `PUT`, `POST`
   - Read: `GET`
   - Update: `PUT`
   - Partial update: `PATCH`
   - Delete: `DELETE`

C. Self-descriptive messages

   Messages exchanged in requests and responses must be self-descriptive. To do so, a message contains

   - *data* : representation of the resource
   - *metadata*: description of the resource

   HTTP provides the `Content-Type` response header to describe the type of the representation (JSON, XML, etc). *JavaScript Object Notation* (JSON) is used for the representation of resources between client and server of the MultiMonitoring application. Metadata can be contained in the body of the response.

D. Hypermedia As The Engine Of Application State (HATEOAS)

   HATEOAS provides the client a way to control the application state. A state transition can be done through actions identified in an hypermedia. A resource representation corresponds to an hypermedia. This one can contains hyperlinks representing the actions. The client can thus change the application state by following the hyperlinks.

In the MultiMonitoring application, the client can navigate through the paging system by following hyperlinks points, for example, to pages before and after the current page.

HTTP `Link` header contains a link to another resource and allows thus hypermedia control in HTTP responses. Hypermedia control can also be added directly in the resource representation.

5. **Layered system**
   The layered system constraint states that an architecture can be composed of hierarchical layers where each component can't see beyond the layer to which it is directly connected. For a given component, knowledge of the system is thus restricted to a single layer.

6. **Code on demand (optional)**
   This final constraint allows to extend client functionality by downloading and executing code in the form of applets (e.g., Java applets) or client-side scripts (e.g., JavaScript).

2.2.2 MultiMonitoring resources

As said previously, resources are identified by URIs. It is recommended to use human-readable URIs. In particular, URIs should be hierarchical and predictable.

In the MultiMonitoring application, the resources correspond to jobs. The following URI identifies a collection of jobs:

```
http://localhost:9018/multimonitoring/jobs
```

A job can also be accessed by its identifier. For instance, to get the job whose identifier is 5, the URI to use is:

```
http://localhost:9018/multimonitoring/jobs/5
```

2.2.3 Query Language

The job monitoring window displayed at the client allows him to select the jobs he is interested in. Indeed, the window permits the client to filter and sort the jobs and to use a pagination system. All those operations are applied on the jobs and so on the corresponding resources. Thus, a query language is used in the URI and is interpreted at the server side to return the jobs corresponding to the query.

The MultiMonitoring application defines a query language based on *query parameters*. A query on the resource `jobs` has the following form:

```
.../jobs/?param1=val1&param2=val2&...&paramN=valN
```

It begins with a question mark followed by a list of query parameter-value pairs separated by an ampersand.

Filtering

The first operation consists in filtering a resource representing a list of other resources to obtain a sublist corresponding to the filter. Filters are based on fields of the resources. We can use query parameters corresponding to those fields in the URI. For instance, the following URI identifies the list of jobs whose agent job identifier is 1, the mean byte rate is lower or equal to 40 megabytes per second and the job monitor identifier is either 2, 3 or 4:

```
.../jobs/?AgentJobId=1&MeanByteRate:lte=40&JobMonitorId=2,3,4
```

Filter conditions on the same field form a disjunction and conditions on different fields form a conjunction.

Sorting

Then, we want to sort a list of resources. We want to sort on several fields in ascending or descending order. The `_sort` query parameter is used in the URI. The value of this parameter is a comma-separated list of values corresponding to fields of the resources. Each field is optionally followed by a colon and the sort direction. The following URI identifies the list of jobs sorted by identifiers in ascending order, then by mean byte rates in descending order:

```
.../jobs/?_sort=Id:asc,MeanByteRate:desc
```

Pagination

To construct a pagination system, we need to separate the list of resources in pages of a given number of resources and access a page using its number. To do this, two query parameters are used:

- `_max`: number of resources per page, i.e. the maximum number of resources returned to the client.

- `_offset`: the number of resources to skip to reach the wanted page.

For instance, to access the 5th page of 20 resources, we use:

- `_max` = 20
- `_offset` = page number * `_max` = 5 * 20 = 100

The 100 first resources are skipped and resources from 101 to 120 are returned.

2.3 MultiMonitoring Architecture

In this section, we present the architecture of the MultiMonitoring server and agent. The different data models used are explained. Then, the cache implementation and mechanisms are presented. Finally, we give a description of Entity Framework, the ORM used at the server.

2.3.1 Server architecture

As illustrated in Figure 2.6, the server architecture consists in three main layers:

1. HTTP Api / Hosting
2. Application
3. Infrastructure / Domain.

The first layer exposes a REST API represented by the *Job API* (that allows to do CRUD operations on jobs) and *Job Monitor API* (that allows to do CRUD operations on job monitors). It also hosts html and javascript files used in the GUI.

The second layer is composed of 4 modules:

- *Job Service*: service exposing jobs to other modules, mostly HTTP controllers.
- *Job Monitor Service*: service exposing job monitors to other modules, mostly HTTP controllers.
- *Data Purge Controller*: removes older jobs from the database to keep its size below a given limit.
- *Job Monitor Observer*: collects the jobs from the job monitors and insert them using the Job Service module.

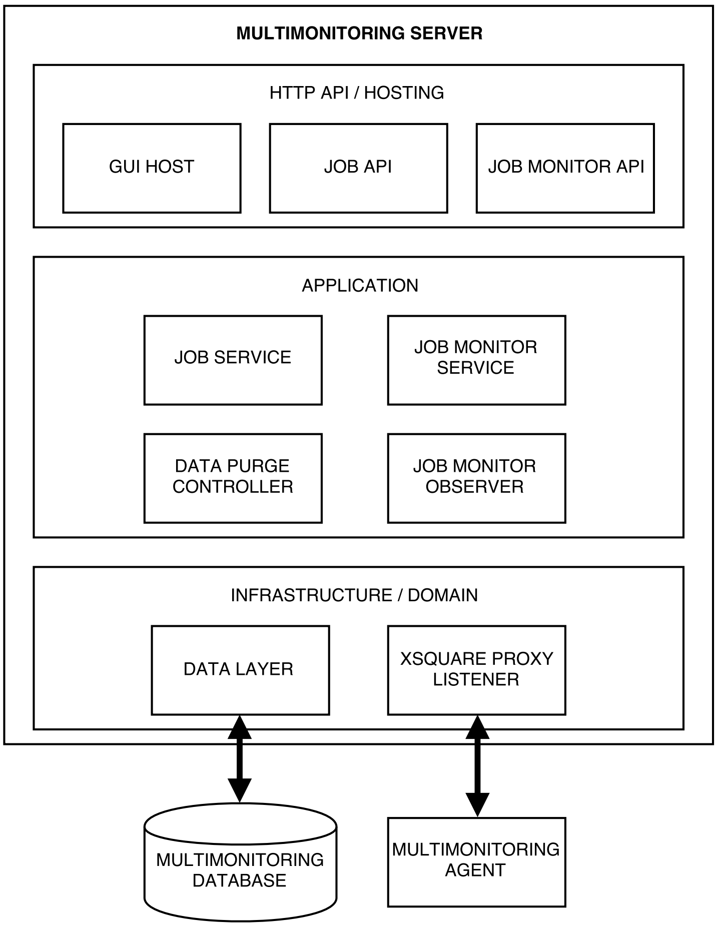

MULTIMONITORING SERVER
HTTP API / HOSTING
GUI HOST
JOB API
JOB MONITOR API
APPLICATION
JOB SERVICE
JOB MONITOR SERVICE
DATA PURGE CONTROLLER
JOB MONITOR OBSERVER
INFRASTRUCTURE / DOMAIN
DATA LAYER
XSQUARE PROXY LISTENER
MULTIMONITORING DATABASE
MULTIMONITORING AGENT

Fig. 2.6 Architecture of the MultiMonitoring server.

The third layer manages the data of the application and provides access to the database. It also contains a listener that subscribes to job notifications (job creation, job update,...) of a remote Xsquare accessed via a MultiMonitoring agent.

As already mentioned, the server contains an in-memory cache of jobs. Mechanisms of the cache are presented in Section 2.3.4.

An *Object-Relational Mapper* (ORM) is used by the server to convert easily domain objects from the server to scalar values organised within tables in the database. More information about the ORM is given in Section 2.3.5.

The server manages three representations of jobs and job monitors. Each representation corresponds to a different class. Those classes contain in general the same properties (`Id`, `Status`,...) with small differences:

- Type of properties

- Name of properties
- Addition/removal of properties

We will call those representations *data models*. Three data models are used, as illustrated at Figure 2.7:

- The *API model* is the model that is serialised in JSON and sent to the client.
- The *domain model* is the model used to store jobs in the cache.
- The *core model* is used by the ORM and corresponds to columns in database tables.

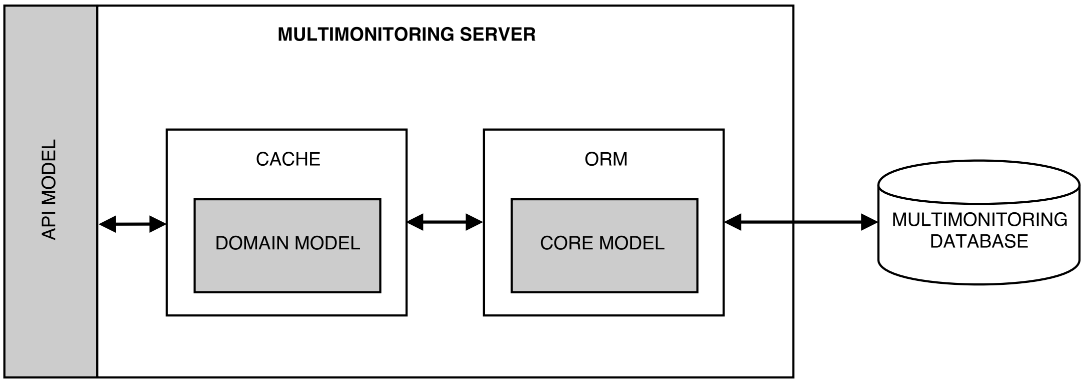


Fig. 2.7 Data models used at the MultiMonitoring server.

Having three different models requires to convert jobs between them several times in the server code and increases its complexity. Moreover, the models contain only small differences between them and are thus maybe not necessary.

2.3.2 Agent architecture

The agent architecture is a simplified version of the server architecture, as illustrated in Figure 2.8. Indeed, GUI, jobs and database are not needed here.

This architecture contains 6 modules:

- The *Job Monitor API* and the *Job Monitor Service* have the same roles as in the server.
- The *Job Monitor Observer* collects job monitors and forward them to the server.
- *Data Layer* here doesn't provide access to a database but to an XML file used to persist job monitors.
- The *Xsquare Listener* subscribes to job notifications of the local Xsquare.

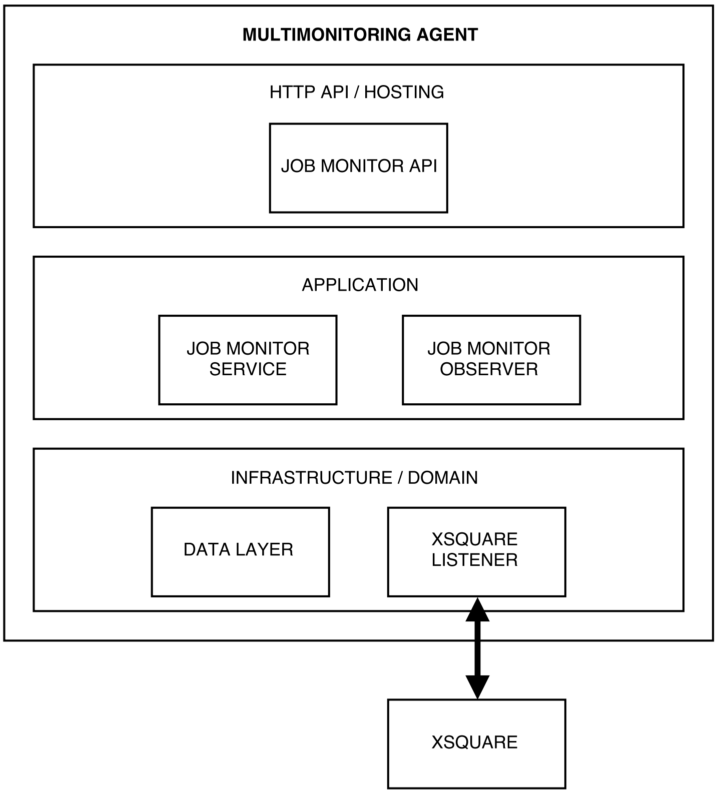

MULTIMONITORING AGENT
HTTP API / HOSTING
JOB MONITOR API
APPLICATION
JOB MONITOR SERVICE
JOB MONITOR OBSERVER
INFRASTRUCTURE / DOMAIN
DATA LAYER
XSQUARE LISTENER
XSQUARE

Fig. 2.8 Architecture of the MultiMonitoring agent.

2.3.3 Database

A *relational database management system* (RDBMS) is used by the MultiMonitoring server to store information about jobs and job monitors. The RDBMS used is MICROSOFT SQL SERVER. The MultiMonitoring database is composed of three tables: `Jobs`, `Destinations` and `JobMonitors`. Destinations of jobs are separated from jobs information because a job can have multiple destinations. We can use a join operation between tables `Jobs` and `Destinations` to obtain jobs information with their destinations.

The only indexes present in the database are:

- The primary key `Id` in the `JobMonitors` table.
- The primary key `Id` and foreign key `JobMonitorId` in the `Jobs` table.
- The primary key `Id` and foreign key `Job_Id` in the `Destinations` table.

The number of jobs in the database is limited to 300,000. The *Data Purge Controller* module is responsible to remove older jobs when this limit is passed. A purge removes 1000 jobs at once.

2.3.4 Cache

An in-memory cache is used to store the 100,000 out of 300,000 newer jobs from the database. When the server is started, those jobs are copied in the cache. The cache is then used in two distincts ways:

- Read operations are performed only on the cache.
- Create, update and delete operations are performed on both the cache and the database. In this way, the cache and the database are in a consistent state.

Like the database, the cache has a limited size. The *Data Purge Controller* module is also responsible to remove older jobs from the cache when jobs are added into it.

Read and modification operations are performed at the same time on the cache. To avoid reading inconsistent data, a read-only copy of the cache is used for reading. This copy is completely overwritten after that the original cache is modified. The update of the read-only copy is protected with a lock.

As explained in the introduction, the fact that the cache contains only a subset of the jobs in the database is not appropriate for the MultiMonitoring application. So, we'll study the performance of the cache containing all the jobs from the database.

2.3.5 Object-Relational Mapper: Entity Framework

Difficulties appear when a program written in an object-oriented language uses a RDBMS. Indeed, data is not represented and accessed in the same way. This problem is called the object-relational impedance mismatch.

An *Object-Relational Mapper* (ORM) is a tool used to convert data between domain objects and relational database. For instance, *Entity Framework* (EF) is an Object-Relational Mapping framework [10]. It is used in the MultiMonitoring application to link domain objects to SQL SERVER tables. EF provides several additional features like query translation and caching.

Entity Data Model

Data can be stored in many forms (relational databases, XML files, text files, etc). An abstract data model is needed to have a general way to access all those forms of data. The *Entity*

Data Model (EDM) is such a model that describes the structure of data independently of the storage type [3, 9]. It is an extended version of the entity-relationship model. Indeed, the structure of data is described in terms of entities and relationships (see Figure 2.9).

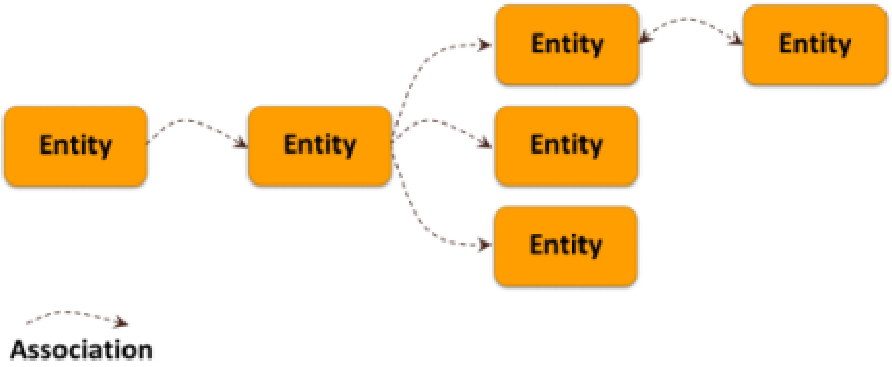


Fig. 2.9 Entity Data Model: entities and relationships [3].

The EDM must be mapped to the data storage schema. For example, if the data store is a relational database, each entity could correspond to a table and the relations could be represented by the foreign keys contained in those tables.

The *entity type* is the key concept used by EDM to describe the structure of data. An entity type contains properties and navigation properties:

- A *property* contains data.
- A *navigation property* represents a relationship between two entities.

An *entity* is an instance of a specific entity type. A collection of entities is called an *entity set*. Then, entity sets are grouped in an *entity container*. All those concepts are illustrated in Figure 2.10.

EF uses the EDM to map domain objects to SQL SERVER tables. Indeed, it instantiates entities as domain objects. Those objects are then used to manipulate the corresponding EDM entities. The EDM is also used by OData, as described in Section 5.1.

2.4 Typical MultiMonitoring setup

In this work, a typical setup is simulated. The server interacts with 20 Xsquares. Each of them sends two types of notifications to the server:

- a job creation every 5 seconds.

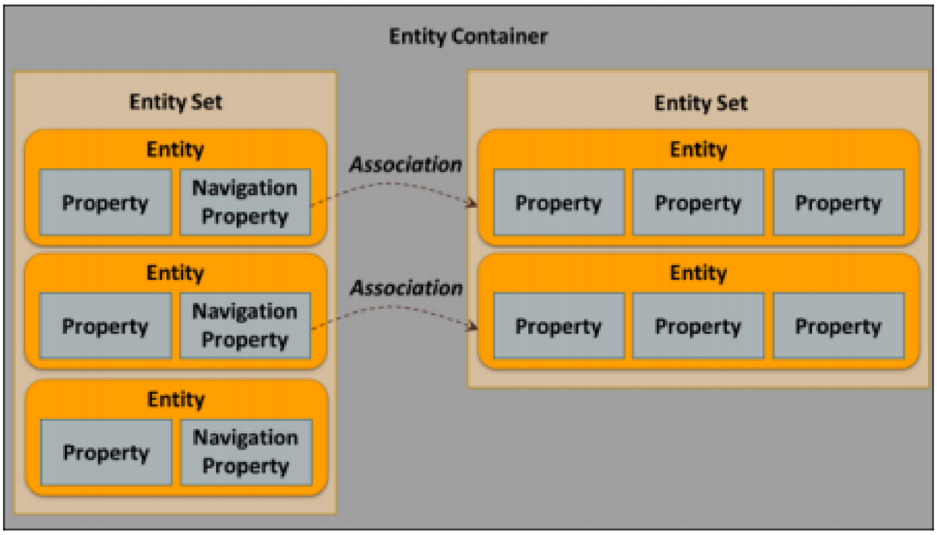

Entity Container
Entity Set
Entity
Property
Navigation Property
Entity
Property
Navigation Property
Entity
Property
Navigation Property
Association
Association
Entity Set
Entity
Property
Property
Property
Entity
Property
Property
Property

Fig. 2.10 Entity Data Model: entity container, entity sets and properties [3].

- for each job, an update every 7 seconds.

The application should support around 100 clients. Each of them can open up to three tabs in its web browser application (each tab corresponding to a particular query). Thus, the server could receives up to 300 read requests per second. So, the ratio read / modify requests is high.

2.5 Summary

The MultiMonitoring application is composed of three components:

- A server
- Agents
- Web clients

An Xsquare, which corresponds to one job monitor, emits notifications about job creations and progression of jobs processing. Those notifications are caught by the agent to which the Xsquare is linked. Then, the agent pre-processes those notifications and sends them to the server to fill its database. The server has an in-memory cache that contains only the newer jobs contained in the database. Jobs extraction requests are performed on the cache, not on the database. The server exposes a RESTful API for jobs which also provides query features like filtering, sorting and pagination. Finally, a web client requests every second the web server to obtain jobs corresponding to a filter that he defined in the job monitoring window displayed in its web browser application. Since there are much more client requests than Xsquare notifications, the read/modification requests ratio is high.

Chapter 3

Performance analysis of the cache

The goal of the cache used in the MultiMonitoring server is to improve performance by querying data in memory. In this chapter, we study the performance of that cache to compare it later to alternative solutions.

3.1 Testing environment

All the tests in this work are performed on a computer running a Windows 7 Enterprise operating system. The computer has 16 Go of RAM and an Intel Core i7-4790 (3.6 GHz). The installed .NET Framework version is 4.5 and SQL Server 2012 is used as the database.

3.2 Database loading

The first step to analyse the cache is to fill the database. To be consistent with the setup from Section 2.4, we inserted 20 job monitors named "Xsquare01", "Xsquare02", ..., "Xsquare20" in the database. To reach the 300,000 jobs database limit fixed by the application, we added 15,000 jobs for each job monitor. Each job has two destinations: a high and a low quality file.

Jobs are inserted by notifications sent by an agent to the server. A tool generating job creation and deletion notifications is available in the project solution (`Tools.Utils`). However, it only supported jobs with one destination. Thus, the tool has been modified to allow a job to have two destinations. We also coded a Power Shell script that uses this tool to simulate the creation of the 300,000 jobs by the 20 job monitors (`fill_db.ps1`).

3.3 RAM usage

Since the number of jobs stored in cache increased from 100,000 to 300,000, we measured the RAM usage when increasing the number of jobs to check if it was not too high. To do so, we used performance counters. Performance counters are used to monitor system components such as memory, CPU, disk I/O and network I/O. One of them is `Process\Private Bytes` and represents the memory allocated exclusively to a process. Here, the RAM usage corresponds to the RAM used by the whole server instance, not only the cache. For a given number of jobs in cache, that usage varies slightly over time. Thus, we took the median of values observed over a little period.

Figure 3.1 displays the RAM usage corresponding to several cache capacities. As expected, the RAM usage increases linearly with the number of jobs. For 300,000 jobs, the server instance uses around 600 Mo. Since servers usually contain several gigabytes of RAM, this value is acceptable.

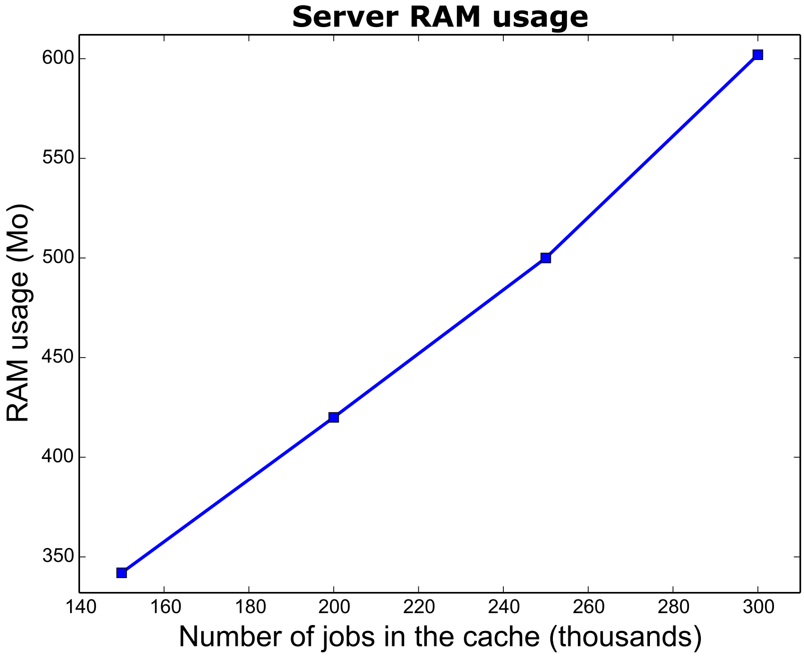


Fig. 3.1 Server RAM usage for several cache sizes.

3.4 Querying resources: LINQ and LINQ to Objects

The *Language Integrated Query* (LINQ) is a component from the MICROSOFT .NET framework that provides data querying capabilities to .NET languages such as C# [15]. Several query operators are available to construct query expressions. For example, operators can be filtering, sorting, aggregation or projection actions.

LINQ can be used to query several kinds of data sources like in-memory collections (arrays, lists, etc), relational databases, XML documents and so on. A LINQ provider translates LINQ expressions to queries understandable by the target data source.

LINQ to Objects is one of the available LINQ providers. It is used to query in-memory collections. In the MultiMonitoring server, the query contained in the URI is translated to a LINQ expression that is then used to query the cache using LINQ to Objects.

3.5 Performance analysis

We will represent the performance of a request by its execution time. More generally, the performance of the server can be represented by the number of clients it can support. There is a direct link between the query execution time and the number of clients supported by the server: faster requests result in a bigger number of clients supported. However, another factor can limit the number of supported clients: bottlenecks (resource contention, lack of memory, high CPU usage, etc). There may be contention for shared resources either in the application itself (e.g., lock on data structures) or by the OS (number of simultaneous TCP connections, etc) or in the network (bandwidth). Here, network is not an issue since the client and the server are on the same computer.

3.5.1 Query execution time

We measure the execution time of a client query at three locations:

- At the client: time between the sending of the HTTP request and the receiving of the corresponding response.
- At the middleware (web server).
- At the controller (MVC pattern): time taken by the cache to apply the query and return the corresponding data.

The most interesting measure here is the one taken at the controller. The two others are useful to determine where the request spends most of its execution time.

The duration are measured using the `Stopwatch` class from the .NET framework. This class allows to measure the elapsed time in milliseconds. The three measured times are logged in separate log files and linked together thanks to identifiers. Since the client execution time contains the middleware time that contains the controller time, we can subtract them to obtain the time spent only in the cache, spent only in the middleware and spent only between the client and the server.

A program has been developed to execute sequentially a specific request a given number of time (`ClientMM` directory in the annexes). Another program computes the mean values of the three execution times from the three corresponding log files (`Parsing` directory in the annexes).

We determined five typical requests that will be measured according to the presented methodology:

1. 20 first jobs
2. 20 first jobs, sorted by ascending start time
3. 50 jobs (100th to 150th jobs), sorted by ascending start time
4. 20 first jobs, sorted by ascending start time, where the job status is "failed" (a little portion of jobs has this status)
5. 20 first jobs, sorted by ascending start time, where the job status is "failed" and where at least one destination contains "file5" substring in its destination file name.

Table 3.1 presents the mean of the measured execution times when the 5 typical requests are executed 1000 times.

Request	**Controller (cache)**	**Middleware**	**Client**
#1	15	9	3
#2	125	8	5
#3	115	25	5
#4	452	9	6
#5	452	8	5

Table 3.1 Mean times spent on the cache, in the middleware and between client and server, in ms.

Most of the execution time is spent when querying the cache. Reducing the time taken to query the cache will thus reduce greatly the whole execution time of the query.

The middleware time of request 3 is about three times greater than request 2. Indeed, the middleware is responsible for instance to serialise data and sending it on the network. Thus, since request 3 has almost three times more jobs to process, execution time is around three times greater than request 2.

From request 1 to request 2, a sort operation is added to the request and the execution time increased by about 100 ms. Requests 2 and 3 show that taking 20 or 50 jobs results in about the same querying time at the cache. Then, adding a filter to request 3 to obtain request 4 increased the execution time by around 300 ms, which is a lot. Finally, adding one more filter has nearly no impact on the measured time. Indeed, the resulting set of jobs from the previous filter is small because it is a good filter. Thus, the second filter has only a few jobs to consider and is executed very quickly.

This little analysis showed that sorts and filters can lead to very different execution times on the cache. So, it is interesting to study in more details the behaviour of sort and filter operations.

Sorting performance

We measured the execution time at the cache for incremental combination of sorts. For each sort combination, we performed 1000 measures and we took the mean of the execution times. Those measures are presented in Figure 3.2. We can notice that the execution time increases nearly linearly. This could mean that the list is travelled once for each sort operation, which is not very efficient. It is hard to check this supposition since LINQ to Objects can be implemented in various way.

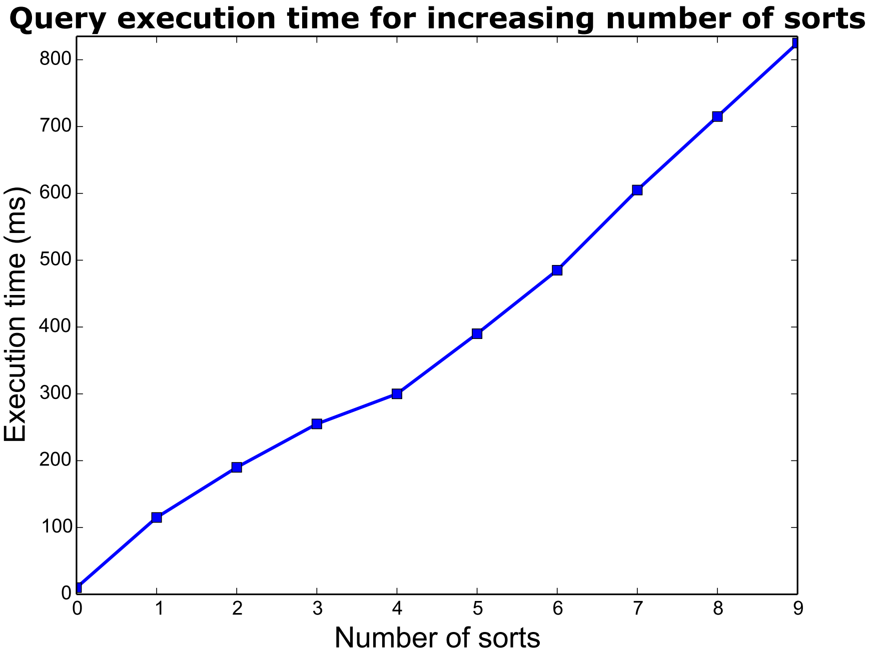


Fig. 3.2 Query execution time for increasing number of sorts.

Filtering performance

The execution time varies greatly depending on the filtered field. The execution time of a query containing one filter can vary from 100 to 500 ms. When multiple filters are applied, the execution time stops to vary significantly once the current result contains a little number of jobs. Indeed, the remaining filters are applied to few jobs.

3.5.2 Number of supported clients

We now try to find the number of clients that the server can handle at the same time. A client requests the sever each second. The response must thus be received within one second from the sending of the request. Therefore, we search the maximum number of clients for which the server can respond within one second for almost all the requests. Indeed, we can tolerate a small portion of responses to arrive a bit later than one second. We set the threshold to 95% of responses received within 1200 ms.

We use GATLING [4], a load testing framework, to simulate the clients. It allows to define scenarios that are executed by clients. In our case, the scenario is quite simple: executing a request every second during, for instance, 5 minutes. We defined several scenarios that differ by the query contained in the URI. Then, a given number of clients is started uniformly during the first second of the test. At the end of the test, a report of statistics is generated and contains distribution of responses times. The scenarios are defined in the file `cache_max_clients.scala`.

We measured seven queries with several sorting and filter combinations, covering nearly all operations possible at the client. Those queries are defined as follows:

1. No sorting, no filter.
2. Sort jobs in decreasing `MeanFrameRate` order.
3. Take the jobs with failed `ShortStatus`.
4. Sort jobs in decreasing `ReceptionTime` order and take the jobs with `SourceFileName` containing the "mp4" string.
5. Take jobs whose at least one destination `FileName` contains "file1".
6. Sort jobs in decreasing `MeanByteRate` order, then in increasing `StartTime` order.
7. Sort jobs in increasing `ShortStatus` order, then in increasing `EndTime` order and take only jobs whose `MeanByteRate` is greater than 30.

Job notifications sent by the agents to the server increases the number of requests that the latter has to handle. To determine the impact of job notifications, we measured the maximum number of clients with and without job notifications sent by the agents. Table 3.2 displays for each request its cache execution time and the maximum number of clients reached (with and without the 20 Xsquares sending job notifications).

Query	**Query execution time (ms)**	**Clients**	**Clients (with notifications)**
#1	17	60	40
#2	170	17	9
#3	400	6	5
#4	130	30	23
#5	125	25	13
#6	255	5	5
#7	350	4	2

Table 3.2 Maximum number of clients for several queries.

We first analyse the results obtained without job notifications. We can observe that the higher maximum number of clients are reached for queries that have smaller cache execution times. The higher number of clients is reached with the query that applies no sort and no filter (query 1).

The number of clients decreases for all queries when the job notifications are enabled. Indeed, the cache is locked when a job is inserted/updated. Read queries must thus wait that the ongoing insertion/update finishes.

3.6 Conclusions

We first showed that loading 300,000 jobs in the cache instead of 100,000 was not a problem for RAM consumption. We then analysed the performance of some queries to understand the impact of query operators on the execution time. We discovered that the list was travelled for each sort operation. Filtering operations are quite uncertain, which is not wanted. We also measured the maximum number of clients that the server can support at the same time.

The cache is used to perform query in memory. However, databases also use caching. Moreover, databases are optimised to execute complex queries. For example, they contain a query optimiser and provide indexes to speed up operations. The performance of the database is studied in the next chapter.

Chapter 4

Performance analysis of the database

This chapter provides an analysis of the database performances. It studies the page caching of SQL SERVER and its query execution performance. The results obtained in this chapter are then compared to those obtained in the previous chapter. Finally, the consequences of a cache removal are discussed.

4.1 Querying resources: LINQ to Entities

We don't query directly the database but we query entities defined by the ORM, as presented in Section 2.3.5. Those entities correspond to database tables. The *LINQ to Entities* [15] provider is used. This provider translates LINQ expressions to SQL queries that are executed on the database.

In the MultiMonitoring server, queries contained in URIs are translated to LINQ expressions that are used with the LINQ to Objects provider. However, LINQ expressions for the LINQ to Objects and LINQ to Entities providers are not completely compatible. Thus, in the following test, LINQ query will be hard coded directly in the server.

4.2 Page caching

When executing a query, the database must access data. However, accessing the disk is very slow. That is why SQL SERVER caches the data and always reads data from the cache and never from the disk when executing a query. SQL SERVER caches pages of 8Kb. When the requested data is not in the cache, the query must wait until data is retrieved from the disk and inserted in the cache. When a query must modify data, it modifies pages in the cache. Those pages are then written to the disk [17].

Since disk I/O are very slow, we want to avoid them. We first try to understand what data is cached in the MultiMonitoring database. To do so, we use three performance counters about the cache of SQL Server: `Page reads/sec`, `Page writes/sec` and `Buffer cache hit ratio`. The first two track the number of physical database page reads and writes. The third corresponds to the percentage of pages found in cache without reading the disk.

To study the SQL Server disk I/O, we proceed as follows. We start the MultiMonitoring server and the corresponding SQL Server instance. We start to monitor the three performance counters. Then, we execute a read request returning all the jobs and we observe that a lot of pages are read from disk while the request is processed, as shown in Figure 4.1. After that, we execute several types of read requests. We observe no disk reads during those requests. We can also notice that while the pages are read from the disk, the `Buffer cache hit ratio` decreases, since requested pages are not in the cache. Its value rises to 100 while new queries are processed. The `Page writes/sec` counter is always zero, which means that the database doesn't write any pages from cache to disk. Indeed, read queries don't modify the jobs.

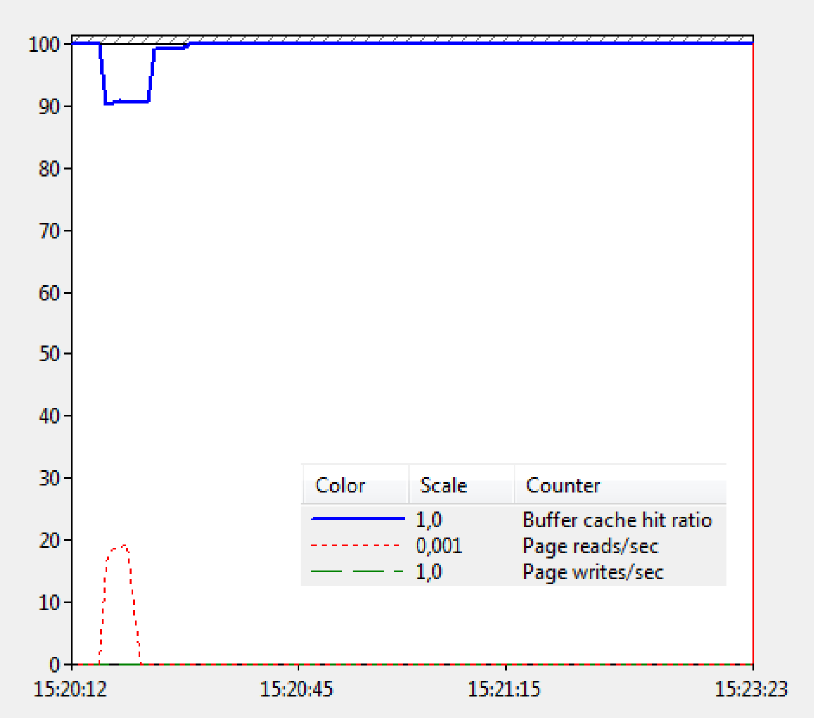


Fig. 4.1 Evolution of SQL Server performance counters during read queries.

With this experiment, we showed that SQL Server loaded all the pages corresponding to the jobs in the cache during the first request. After that, no disk I/O were needed since the following read requests read data in the cache and because there were no modification requests that would modify pages in the cache and copy them to disk.

We then performed modification requests and observed disk writes thanks to the `Page writes/sec` performance counter. Those writes correspond to copy of modified cached pages to disk.

Now, we know that all the jobs can be stored in the cache and thus in memory, like the in-memory cache used at the server. However, a drawback of the database is that pages must be written to disk when modified. Since the read/modification ratio of job requests is high, we can hope that it won't reduce too much the performance.

Another point is the concurrent access protection of pages. In the cache contained at the server, a lock on the list provides that protection. In SQL SERVER, each page in the cache has a latch to protect it.

4.3 Performance analysis

As for the cache, we measure the execution times of the five typical queries defined in Section 3.5.1. In addition to measuring the time at the client, the middleware and the controller, we also measure it at the database. To do so, we use SQL SERVER PROFILER [11] that is able to log every request sent to the database.

All execution times of this chapter are obtained by executing the corresponding query 1000 times and taking the mean of those 1000 measured execution times. Table 4.1 presents the execution times for the 5 typical requests.

Request	**Database**	**Controller**	**Middleware**	**Client**
#1	3	3	8	3
#2	183	2	8	5
#3	188	2	16	5
#4	26	3	8	5
#5	26	3	8	5

Table 4.1 Mean times spent on the database, in the controller, in the middleware and between client and server, in ms.

Most of the time is spent at the database, where the query is actually executed. The controller time is low because the controller just sends the query to the database and receives the results back. As for the cache, the midlleware time is greater when the query returns 50 jobs.

Database time of request 2 increased by 180 ms compared to request 1 because of the additional sorting operation. Requests 2 and 3 times are fairly the same. Pagination has thus little consequence on the execution time. The additional filtering operation in request 4 greatly reduces the execution time in database. Adding a second filter in request 5 has almost no impact on the execution time because the resulting set of jobs from the previous filter is small.

There are several ways to execute a query on a database. Some are more efficient than others. SQL SERVER uses a *query optimiser* to find the better alternative and construct the corresponding query execution plan. The query optimiser used by SQL SERVER is a cost based one. It means that it assigns cost to each alternative and chooses the one that has the lowest cost. Costs are computed according to the size of data that must be read, the CPU consumption, the memory usage and so on. To compute those costs, the optimiser uses statistics about the table sizes and the distribution of columns values [17].

Figure 4.2 shows the typical query execution plan generated for most of queries on jobs. The first thing to mention is that the sorting operation is the most expensive. The sorting is allowed on the fields of jobs, but not on the fields of destinations. Operations are performed from right to left in the diagram.

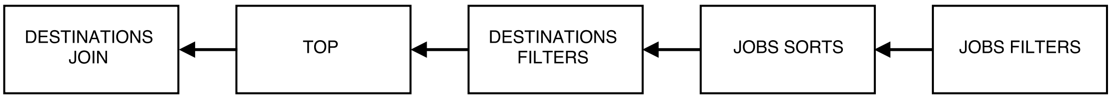
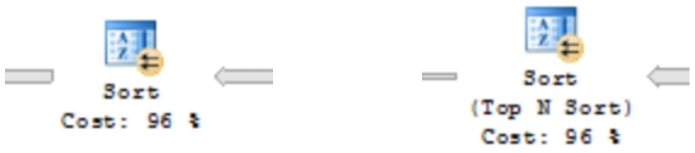


Fig. 4.2 Simplified query execution plan generated for a query on `Jobs` and `Destinations` tables, linked by a join.

The first operation performed is jobs filtering. Indeed, the sorting execution time increases with the number of jobs to sort. This is why it is advantageous to first reduce this number of jobs by filtering them. The second operation is the sorting of jobs. Using statistics computed by the database, the sort operator outputs less jobs that it receives in input. With this optimisation, the next operation, the filtering of destinations, will have to examine less data and will be faster. So, the following operation is the destinations filtering. For each filter, the corresponding job identifiers are found in the `Destinations` table using the `Job_Id` column and they are used to filter the jobs. The next operation is the `TOP` operation. This operator keeps only a given number of jobs taken in the sorting order. The final operation is the join between the resulting set of jobs and the corresponding destinations. A join is a time consuming operation. That is why it is done at the end, on a small number of jobs.

We will now study in more details the behaviour of the database when sorting and filtering jobs and destinations.



(a) Usual sort operator (b) Top N Sort operator

Fig. 4.3 SQL Server sort operators

4.3.1 Sorting performance

The execution plan does not contain the usual sort operator but an optimised operator, as shown in Figure 4.3. This optimised sort operator is called `Top N Sort`. It is defined as follows [12]:

> *«Top N Sort is similar to the Sort iterator, except that only the first N rows are needed, and not the entire result set. For small values of N, the SQL Server query execution engine attempts to perform the entire sort operation in memory. For large values of N, the query execution engine resorts to the more generic method of sorting to which N is not a parameter.»*

In our case, the `Top N Sort` operator is used because we take the N first jobs of the sort result. Indeed, the `TOP` and `ORDER BY` SQL operators are present in the same `SELECT` statement from the SQL query generated by LINQ to Entities.

Microsoft doesn't document the sorting algorithms used by SQL SERVER. However, the behaviour of the `Top N Sort` operator can be studied experimentally [18, 20]. We can observe that the threshold value N is equal to 100. For values of N lower than or equal to 100, the operator uses the optimised sort algorithm. Otherwise, it uses the usual sort algorithm. This limit is adapted to the MultiMonitoring application since allowed page sizes are 10, 20, 25 or 50.

The optimised algorithm uses the fact that only N lines must be returned to avoid completely sorting the data set. This allows it to perform the sort entirely in memory, which is faster. The algorithm must still consider all rows in the table but keeps only the N first lines of the ranking progressively.

The experimentally discovered worst case (for performance) of the optimised algorithm appears when keys are long, similar and presented in the reverse sort order.

We want to know the effect of various sorting combinations on the query execution time. The tested queries have the following pattern:

- Join between `Jobs` and `Destination` tables.

- Sort the jobs on a set of given fields, in decreasing order
- Take the 20 first jobs

Since jobs are presented to the user in decreasing identifier order, we complete the sort fields set by the `Id` field. Since the lines of the `Jobs` table are sorted by `Id` in the database pages (a clustered index is defined on this field), the database simply reads the `Jobs` table by decreasing `Id` instead of increasing `Id` and performance are not impacted. The corresponding LINQ to Entities queries have the following pattern:

```
context.Jobs.Include("Destinations")
             .OrderByDescending(field-1)
             .ThenByDescending(field-2)
             ...
             .ThenByDescending(field-n)
             .ThenByDescending(Id)
             .Take(20);
```

Table 4.2 gives the tested sorting combinations and the corresponding execution times measured at the database. For each sorted field, it gives the dispersion of corresponding column values in the database. If the column contains a lot of different values, the dispersion is high. If all the values of the column are the same, the dispersion is null.

First, we can observe that the execution times are fairly constant for queries 1 to 5. Indeed, keys have high dispersion and non similar keys, so comparisons are done quickly.

Queries 6 and 7 are slower than the first five queries. The keys are not very long, but each has a null dispersion. `Type` equals "Xsquare" and `JobInitiatorFull` equals "MyMachine Tools.Utils". So, the database must compare character by character the entire key at each comparison, which takes more time than a simple number comparison for instance. That is why the execution times are higher than others. Then, the execution time of query 7 is higher than that of query 6 because the key is longer.

Queries 8 and 9 contains the three same keys, but in two different orders. In query 8, the high dispersion key is the first and in query 9 it is the third. The two others (`Type` and `JobInitiatorFull`) have constant values. The set of three keys defines in the two configurations a high dispersion key since it contains a high dispersion key. Query 9 is executed slower than query 8. This could come from the fact that it must first compare the 2 null dispersion keys before comparing the third key to determine the sorting order. The comparisons take thus more time and that is why the execution time is a bit greater.

Query	Sorted field(s)	Dispersion	DB time
#1	`MeanByteRate`	High	155
#2	`AgentJobId`	High	166
#3	`MeanByteRate` `AgentJobId`	High High	156
#4	`SourceFileName`	Medium	162
#5	`StartTime`	High	180
#6	`Type`	Null	225
#7	`JobInitiatorFull`	Null	322

#8	`AgentJobId`
	`JobInitiatorFull`
	`Type`
		High
		Null
		Null
			182

#9	`JobInitiatorFull`
	`Type`
	`AgentJobId`
		Null
		Null
		High
			217

#10	`JobMonitorId (FK)`	Low	1.5

Table 4.2 Execution time (ms) at the database for several sort combinations (LINQ to Entities).

Query 10 is executed very quickly because it uses the index defined on the foreign key (FK) `JobMonitorId` and not the `Top N Sort` operator.

Thus, the `Top N Sort` allows to return N jobs in a given sorting order with an execution time that varies slightly with the sorted columns. This operator is slower when sorting column values with low or null dispersion, especially when the keys are long strings, leading to long comparisons. In the case of the MultiMonitoring, the two fields which have constant values would have more diverse values in a real situation.

4.3.2 Filtering performance

We begin by measuring the execution time (at the database) of queries that return the first 20 jobs corresponding to a filter. Once again, the jobs are sorted in decreasing `Id` order, which implies that the database travels the `Jobs` in the reverse order and has no significant impact on performance. The LINQ to Entities queries have the following form:

```
context.Jobs.Include("Destinations")
            .Where(filer-expression)
            .OrderByDescending(Id)
            .Take(20);
```

Table 4.3 displays the queries along with the type of the filtered column, the number of jobs (thousands) obtained after filtering and the execution time at the database in milliseconds. First of all, we notice that all the execution times are quite small. This mean that a filtering operation, which requires to travel at most once the table, is very fast. However, since we need only the first 20 jobs, the table examination can stop once 20 lines satisfying the filter are found. The filter from query 1 is satisfied by 300 thousands jobs, i.e. by all jobs of the table (bad filter). The 20 jobs returned by the query are thus the first 20 jobs examined. Query 2 is a bit slower than query 1. Indeed, only 60 thousands jobs match the filter and thus the database needs to examine more jobs to find 20 jobs satisfying the filter.

Queries 3, 4, 5 and 6 are slower than queries 1 and 2. Indeed, the search of a substring takes more time than a simple comparison between two double values. Once again, query 4 execution time is greater than that of query 3 because it must examine more lines to obtain 20 jobs satisfying the filter. Query 6 execution time is greater than that of query 5 for the same reason.

Queries 3 to 6 are all filters that search a substring in a string column of the database. However, the queries that filter on the destination file name are slower than those filtering on the source file name. Indeed, there are two destinations per job and the corresponding lines in the `Destination` table are stored one after the other since they are inserted at the same time. Thus, the filter operator must examine two destinations to obtain one job and the overall execution time is thus greater.

Query	Type	Filter(s)	Jobs	DB time
#1	double	`MeanByteRate > 15`	300	0.956
#2	double	`MeanByteRate > 35`	60	0.993
#3	string	`SourceFileName contains "jm"`	300	3.522
#4	string	`SourceFileName contains "file5_jm13.flv"`	2.5	5.809
#5	string	`Destination.FileName contains "jm"`	300	3.772
#6	string	`Destination.FileName` contains `"file5_jm13.flv"`	2.5	13.159

Table 4.3 Execution time (ms) at the database for several filtering queries (LINQ to Entities), type of filtered columns and number of jobs (thousands) satisfying the filtering condition.

To force the database to fully filter the table, we add a sorting operation. The sorted column is not important because it will always be the same. So, we decide to sort the result of the filter by decreasing `MeanFrameRate`. LINQ to Entities queries have now the form:

```
context.Jobs.Include("Destinations")
            .Where(filer-expression-1)
```

```
...
.Where(filer-expression-n)
.OrderByDescending(MeanFrameRate)
.ThenByDescending(Id)
.Take(20);
```

Query	Type	Filter(s)	Jobs	DB time
#1	double	`MeanByteRate > 15`	300	149
#2	double	`MeanByteRate > 35`	60	47
#3	double	`MeanFrameRate > 15`	300	151
#4	datetime	`Hour of StartTime > 16`	50	47
#5	int	`ShortStatus = Successful`	300	148
#6	double int	`MeanByteRate > 15` `ShortStatus = Successful`	300	154

#7	double
	int
	datetime
		`MeanByteRate > 15`
		`ShortStatus = Successful`
		`Hour of StartTime > 16`
			50	52

#8	datetime
	double
	int
		`Hour of StartTime > 16`
		`MeanByteRate > 15`
		`ShortStatus = Successful`
			50	52

#9	double
	datetime
	double
	int
		`MeanFrameRate > 15`
		`Hour of StartTime > 16`
		`MeanByteRate > 15`
		`ShortStatus = Successful`
			50	52

#10	string	`SourceFileName contains "jm"`	300	297
#11	string	`SourceFileName contains "mp4"`	65	127
#12	string	`SourcePath contains "source"`	300	301
#13	string string	`SourceFileName contains "mp4"` `SourcePath contains "source"`	65	141
#14	string	`Destination.FileName contains "jm"`	300	402
#15	string	`Destination.FileName contains "mp4"`	65	196

Table 4.4 Execution time (ms) at the database for queries containing several filter combinations followed by a sort on MeanFrameRate, type of filtered columns and number of jobs (thousands) satisfying the filtering condition.

Results are shown in Table 4.4. We first examine the first nine queries, ignoring the filters on string columns. We can observe that the execution time depends on the number of jobs returned by the filter. The more returned jobs the higher the execution time. Indeed, the sort operator takes more time when it sorts more jobs. Then, we observe that several filter combinations returning the same number of jobs have approximately the same execution time. Indeed, the time spent sorting is preponderant and the filter combination (on double/int/datetime) has nearly no impact on the execution time. Queries 1, 3, 5 and 6 sort all the table since each filter returns nearly 300 thousands jobs. The execution time of those queries is around 150 ms, which is approximately the time observed in Table 4.2 displaying only sort queries.

Queries 7 and 8 contains the same filters but in a different order. Yet, the query execution plan shows that the database applies filters in the same order in both cases: `MeanByteRate` filter, `ShortStatus` and then `StartTime` filter. The `StartTime` filter is the more filtering one. However, the database must extract the hour information from the complete date. That is maybe why the database performs this filter in the last position. We then add a new filter on `MeanFrameRate` to query 8 to obtain query 9. We observe in the query execution plan that this new filter took the first place of the filtering order. The query optimiser is quite complex but we saw that it reorganises the filters to obtain the best execution plan, according to him. Then, contrary to the cache, the database applies all filters at the same time when examining a line.

For the same number of jobs satisfying the filters, the execution time of queries filtering on `string` columns (queries 10 to 15) is higher. So, the performance problem comes from the filtering part of the queries. As already mentioned, searching a substring in another string takes more time than a number comparison. That is why the filtering part of the query for queries 10 to 15 has more impact on the execution time than for queries 1 to 9.

For the same filtered number of jobs, the execution time of queries filtering on destination fields (queries 14 and 15) is greater than that of queries filtering on job fields (queries 10 to 13) because the former examine twice more lines since there are two destinations for one job.

The query execution plan of query 13 shows that the database applies first the `SourceFileName` filter (the more filtering one), which is the desired result. We can also notice that, as for previous queries, the execution time is higher when the sort operator has more jobs to sort.

4.4 Comparison with the cache

As for the cache, the database can maintain all the jobs in memory and thus perform operations directly in memory. However, pages cached in memory must be copied to disk each time their are modified. Since the ratio job read/updates is quite high for the MultiMonitoring application, this shouldn't happen too often and performance should stay good.

The database sorts the jobs in a more efficient way than the cache. Indeed, the execution time increases for each additional sorted fields for the version with the cache while this time is quite independent of the number of fields to sort for the version using the database. Then, the query optimiser of the database provides clever filtering features. Contrary to the cache, the table of jobs is travelled only once and the sort conditions are applied in an order that seems efficient to the query optimiser. This results in a filtering operator executed quickly in most cases. Yet, filtering on string columns requires string matching that is handled quite inefficiently. On the other hand, the filtering performance of the cache is quite random and can lead to poor execution times.

The database query-handling performance seem more appropriate for the MultiMonitoring application because the execution time is in most cases reasonable and bounded. Moreover, its performance can be optimised by using indexes.

4.4.1 Removing the cache

We can analyse the consequences of deleting the cache. This would allow to distribute the application, which is the main objective of this work. Then, the intermediate data model (domain model) and some conversions between data models could be deleted, simplifying the application. However, the data model of the cache (domain model) contains two information in addition to the list of jobs:

- the number of jobs matching the filter of the query
- the name of the corresponding job monitor for each job

If we remove this model, we must find another way to add these values to the result.

Finally, remember that we hard-coded LINQ queries in the server. The module that translates queries contained in URIs to LINQ expressions must be deeply modified to be compatible with LINQ to Entities. Since that module is quite complex, a lot of time would be needed to modify it to use LINQ to Entities. Moreover, it should be heavily tested to ensure its reliability before using it in a company product. That is why we searched for an existing tool providing a translation of queries executed on a REST API to LINQ to Entities query. A tool providing this feature is OData.

4.5 Conclusions

The database performances appear to be competitive with those of the cache. All the tables can be kept in memory and queries are generally well executed thanks to the query optimiser. Moreover, the use of indexes may improve query execution times. However, the impact of table modifications (job insertions and updates) on the performances of the database must be studied.

Removing the cache allows to distribute the application and simplifies its code. However, we loose some features and the URI to LINQ query translation. In the next chapter, we will present and study the performance of the tool OData, which provides that translation.

Chapter 5

Open Data Protocol

In this chapter, we present OData and explain how we can create an OData endpoint in the MultiMonitoring application. We then measure the performance of the queries generated by OData and we use indexes to speed up query execution. We finally measure the number of clients the MultiMonitoring server can support.

5.1 Presentation

The *Open Data Protocol* (OData) is a web-based protocol that allows the creation and consumption of queryable RESTful APIs [1]. The goal of OData is to provide a common way to access diverse data sources. Version 4 of OData has been standardised by OASIS in March 2014 [16]. As shown in Figure 5.1, the OData technology can be divided into four components:

1. **OData data model**
   OData needs an abstract data model to describe in a generic way the data coming from several data sources. To do so, it uses the Entity Data Model (EDM) defined in Section 2.3.5.

2. **OData protocol**
   This protocol defines a set of rules for the transmission of information between OData clients and OData services. The OData protocol is based on REST and is built on HTTP, as explained in Section 2.2.1. Resources can be represented in JSON or XML. In addition to the usual CRUD operations, the protocol defines a rich query language. This query language allows a client to issue precise queries on resources when performing a read operation. To do that, a set of options can be appended in the URI of a `GET` request. A subset of those options includes:

- `$top=n`: returns the first n entities of the corresponding entity set.
- `$skip=n`: skips the first n entities in the corresponding entity set.
- `$orderby=order-expression`: orders the resulting entities according to one or several properties, in ascending or descending order.
- `$filter=filter-expression`: returns entities that match the `filter-expression` in the corresponding entity set.
- `$select=select-expression`: returns a subset of the entity properties specified by the `select-expression`.
- `$count=true/false`: if `true`, inserts in the response the number of entities corresponding to the `filter-expression`; does nothing otherwise.
- ...

The complete query language options and the correct syntax can be found in the documentation of OData [1].

3. **OData service**
The OData service implements the OData protocol and exposes the data via the OData data model. It also translates the OData data model to the data store model.

4. **OData client libraries**
Those libraries are used in client applications and provide features to easily perform OData requests and handle the corresponding results. The use of those libraries is optional.

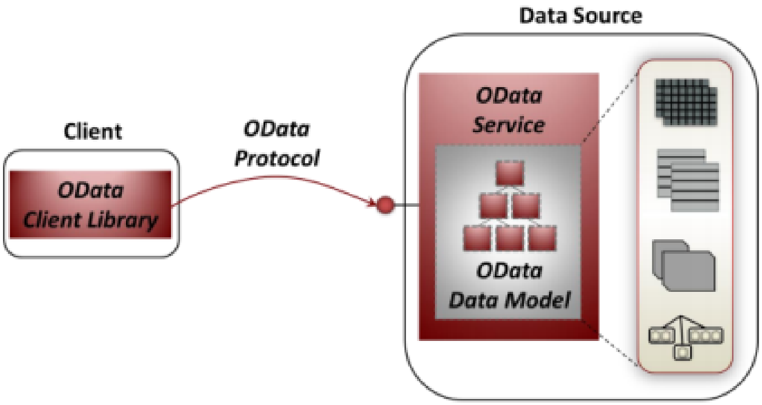


Fig. 5.1 Components of the OData technology [3].

5.1.1 OData .NET library

Several client-server libraries implement the OData protocol. In particular, Microsoft provides an OData .NET library that can be used with Web API to build an OData v4 service [19]. This library includes several features in addition to the protocol implementation: query options checking, EDM security, etc.

We will now explain how OData queries the EDM defined by EF. The process is shown in Figure 5.2. First, a scanner and a parser translates the query contained in the URI in an abstract syntax tree (AST). The OData visitor then visits that AST to create the corresponding LINQ query. Finally, that LINQ query is given to LINQ to Entities which makes the translation to SQL.

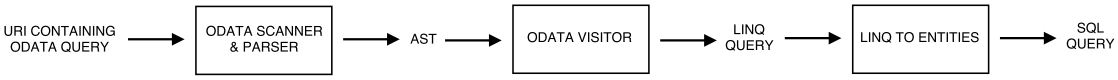


Fig. 5.2 OData query translation from URI to SQL.

5.2 Using OData with the MultiMonitoring application

In this section, we explain how an OData endpoint can be added in the MultiMonitoring server and how we removed the cache. We also detail the code modifications performed. The modified version of the MultiMonitoring application can be found in the directory `MultiMonitoring (modified version)` in the annexes. Comments starting by `// TFE:` are present in the code to show the main modifications.

5.2.1 OData and MultiMonitoring server

As shown in Figure 2.7, the MultiMonitoring server used three data models: the API model, the domain model used by the cache and the core model for the Entity Data Model. The domain model can be deleted since the cache is not used anymore. One goal of the API model is to hide the internal representation of data in the server. However, an OData endpoint exposes the model corresponding to the EDM, so the core model.

A solution is to use AutoMapper [2] that can provide a translation of OData queries on an API model to the core model (corresponding to the EDM). However, our tests showed that this translation mechanism results in very poor performance and restricts the set of allowed query functions.

That is why we decided to expose directly the core model. OData allows to hide some properties of the entity types from the EDM if we don't want them to be part of the interface.

What interests us in OData is its querying functionality. So, we use only OData for read requests and use the initial REST API for create-update-delete operations. Only the jobs must be queried. Thus, the job monitors still use the API model. The resulting configuration is shown in Figure 5.3. The CRUD operations on the job monitors are handled by the initial job monitor controller. Since the cache and thus the domain model has been removed, the API model is directly converted to the core model. The CUD operations on jobs are handled by the initial job controller. It has been modified to work on the core model. Finally, the read operation (that provides the query language) is handled by the OData controller and thus uses the core model.

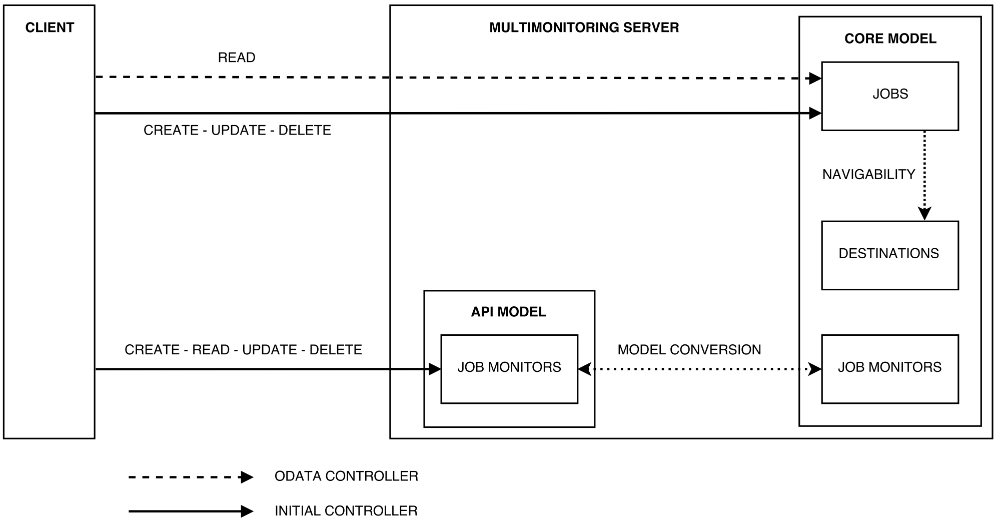

CLIENT
MULTIMONITORING SERVER
CORE MODEL
READ
JOBS
CREATE - UPDATE - DELETE
NAVIGABILITY
DESTINATIONS
API MODEL
CREATE - READ - UPDATE - DELETE
JOB MONITORS
MODEL CONVERSION
JOB MONITORS
ODATA CONTROLLER
INITIAL CONTROLLER

Fig. 5.3 MultiMonitoring server: link between CRUD operations and data models.

Code modification

The code of the MultiMonitoring server has been modified to remove the cache and to install an OData endpoint. Here is a list of the main modifications performed on the code:

- `Http.Api.Controllers.JobController`:
  Deletion of the two methods handling `GET` requests, modification of the methods to use the core model and not the API model.

- `Http.Api.Controllers.JobMonitorController`:
  Modification of the `SaveJobMonitor` method to convert the API model to the core model instead of the domain model.

- `Http.Api.Controllers.JobsController`:
  OData controller class added, addition of a method handling read queries. Notice that OData needs a controller name equals to the plural entity set name (Job**s**).

- `Http.Api.Model.JobMonitor`:
  Modification of the constructor to construct an API model job monitor from the core model instead of the domain model.

- `Application.Services.IJobMonitorService`:
  Modification of the function signatures from the interface to return core model job monitors instead of domain model job monitors.

- `Application.Services.IJobService`:
  Removal of cache-specific function signatures.

- `Application.Services.JobMonitorService`:
  Removal of the job monitor cache, modification of functions to execute operations directly on the database (using the job monitor repository) instead of the cache, domain model has been replaced by the core model.

- `Application.Services.JobService`:
  Removal of the job cache: jobs are accessed directly at the database, updates of the cache have been removed, the cache purge system has been removed, the cache loading at start-up has been removed, the read-only copy of the cache and its refresh mechanism have been removed. Addition of a function overwriting an existing core jobs.

- `Infrastructure.Data.Repositories.IJobRepository`:
  Modification of the function signatures of the interface to work with core jobs instead of domain jobs, removal of the function signature used to fill the cache from the database at start-up.

- `Infrastructure.Data.Repositories.JobRepository`:
  Removal of all conversions from core model to domain model, removal of a function returning the jobs from database to fill the cache at start-up.

- `Infrastructure.Data.Repositories.IJobMonitorRepository`:
  Modification of the function signatures of the interface to work with core job monitors instead of domain job monitors.

- `Infrastructure.Data.Repositories`:
  Removal of all conversions from core model to domain model.

- `Infrastructure.Core.Job`:
  Addition of two properties (`ShortStatus` and `JobInitiatorFull`) and a navigation property to allow joins with job monitors (`JobMonitor`).

- `Infrastructure.Core.Destination`:
  Addition of a property (`ShortStatus`)

- `Infrastructure.Core.Status`:
  Simplification of the `InternalCode` representation, addition of a constructor used by the tool sending job notifications.

- `Host.Startup`:
  Configuration of the OData endpoint: creation of an OData controller based on the entity sets (`Jobs`, `Destinations` and `JobMonitors`) and creation of a route for the OData service,

- `Tools.Utils.Command`:
  Addition of some parameters to the command.

- `Tools.Utils.SendXsquareJobsCommand`:
  Addition of a second destination for job notifications, addition of missing information.

The code has been greatly simplified and shortened. Indeed, the following classes have been removed:

- In `Http.Api.Controllers`:
  `ExpressionBuilder`, `MultimonitoringExpressionVisitor`, `FilterParser` and `SortParser`.

- In `Http.Api.Model`:
  `Job`, `Destination`, `JobCollection`, `IMapper`, `JobMapper`, `DestinationMapper`, `StatusMapper`, `Map` and `ComparisonOperatorMapper`.

- In `Domain`:
  `IDomainToCoreConverter`, `DomainToCoreConverter`, `IDomainUpdater` and `DomainUpdater`.
- In `Domain.Model`:
  `IJob`, `Job`, `IDestination`, `Destination`, `JobType`, `ShortStatus` and `StateType`.

5.2.2 OData and MultiMonitoring client

OData uses a different syntax than the MultiMonitoring for its query language. The MultiMonitoring client must thus be updated to generate queries in URIs that follows the OData query language. To do so, two files have been modified:

- `Http.UI.app.services.jobService.js`
- `Http.UI.app.controllers.jobController.js`

When we removed the cache, we lost an information that was present in the cache but not in the database: the number of jobs corresponding to the filter of the query. This information is used to determine the total number of pages in the pagination system. Hopefully, OData inserts this information in the server response when we set the `$count` query option to true in the URI. This is done in the `jobService.js` file.

5.3 Performance analysis

In this section, we study the performance of OData queries. We can expect to observe similar performance to the performance of the database from Section 4.3 since OData uses LINQ to Entities. However, OData performs the translation of the query parameters contained in the URI to LINQ expressions. Those resulting LINQ expressions are not necessarily optimal. Moreover, the query translation mechanism can't be altered. We recall that queries return only 20 jobs by using the `$top` query option.

All execution times of this chapter are obtained by executing the corresponding query 1000 times and taking the mean of those 1000 measured execution times.

5.3.1 Filtering

We measured the execution times of the same queries as in Section 4.3.2. We obtained the same results as those displayed in Table 4.3. Indeed, the query execution plan obtained with OData is the same as the one generated for LINQ to Entities for filtering queries.

5.3.2 Count query option

When the `$count` query option is set to true, a second query is executed on the database. That query computes the number of jobs corresponding to the filter contained in the principal query. This second query travels all the table and applies the filter on each line. It then counts the number of jobs satisfying the filtering condition. This is a different situation form the previous section since here the table is completely travelled for each query. Table 5.1 displays the execution times of count queries for several filtering conditions.

Query	**Type**	**Filter(s)**	**Jobs**	**DB time**
#1	double	`MeanByteRate > 15`	300	22
#2	double	`MeanByteRate > 35`	60	21
#3	int	`ShortStatus = Successful`	300	22
#4	double int	`MeanByteRate > 15` `ShortStatus = Successful`	300	25
#6	string	`SourceFileName contains "mp4"`	65	97
#7	string	`SourcePath contains "source"`	300	80
#8	string string	`SourceFileName contains "mp4"` `SourcePath contains "source"`	65	112
#9	string	`Destination.FileName contains "jm"`	300	325
#10	string	`Destination.FileName contains "file5_jm13.flv"`	2.5	249

Table 5.1 Execution time (ms) at the database for several count queries, type of filtered columns and number of jobs (thousands) satisfying the filtering condition.

We first analyse queries 1 to 4 that filter on numbers. The execution time is quite the same for the four queries because the filter is tested for all the lines. Query 4 is a bit slower because it must check two conditions.

The execution times of queries 6 to 10 are greater than those of queries 1 to 4 because searching a substring is slower than a number comparison. The execution time of query 8 is greater than those of queries 6 and 7 because the filter contains two conditions.

Finally, we notice that queries 9 and 10 are much slower than the other queries. Indeed, those two queries filter on columns of the `Destinations` table, contrary to the others that filter on the `Jobs` table. However, the query want to count the number of *jobs* corresponding to the filter and not the number of destinations. Thus, queries 9 and 10 must perform an additional operation: determining the jobs corresponding to the destinations matching the filtering condition. This is done by performing a join between the jobs table and the destinations resulting from the filtering operation. Yet, a join is a very costly operation.

Query 9 is slower than query 10 because the join is performed on 600,000 (2 × 300,000) destinations matching the filtering condition against 5,000 (2 × 2,500) for query 10.

5.3.3 Sorting

As explained in Section 4.3.1, the query execution plan corresponding to the SQL query generated by LINQ to Entities used an optimised sort operator called `Top N Sort`. Indeed, this operator was used because the `TOP` and `ORDER BY` keywords were present in the same `SELECT` statement. So, the query optimiser was aware that it should not sort the whole table but only return the N first lines in the given sorting order.

In the SQL query generated by OData, however, the `TOP` and `ORDER BY` keywords are not in the same `SELECT` statement and the optimised sort operator is not used. Indeed, OData inserts some constants (for its internal function) in an inner `SELECT` statement, which contains the `ORDER BY` keyword. The `TOP` keyword is present in an outer `SELECT` statement.

We measured the execution times of queries defined in Section 4.3.1. Those queries contained various sorting combinations. Table 5.2 contains the query execution times for OData and for LINQ to Entities, as a reminder.

Query	Sorted field(s)	Dispersion	DB time	OData time
#1	`MeanByteRate`	High	155	244
#2	`AgentJobId`	High	166	196
#3	`MeanByteRate` `AgentJobId`	High High	156	250
#4	`SourceFileName`	Medium	162	323
#5	`StartTime`	High	180	195
#6	`Type`	Null	225	211
#7	`JobInitiatorFull`	Null	322	264

#8	`AgentJobId`
	`JobInitiatorFull`
	`Type`
		High
		Null
		Null
			182	208

#9	`JobInitiatorFull`
	`Type`
	`AgentJobId`
		Null
		Null
		High
			217	301

#10	`JobMonitorId (FK)`	Low	1.5	3.6

Table 5.2 Execution time (ms) at the database for several sort combinations (LINQ to Entities and OData).

We can observe that the OData execution times are generally higher and more varied than the LINQ to Entities execution times. This is due to the loss of the optimised sort operator.

5.4 Use of database indexes

5.4.1 Introduction

An *index* is a data structure associated with a table used to accelerate the data retrieval on the corresponding table. An index is created using keys built from one or more columns of the table. It can be of two types:

- *Non-clustered*: The index contains keys in a sorted order and row pointers to the row storage location. The rows of the underlying table are not sorted following the keys order. There can be several non-clustered indexes per table.
- *Clustered*: The index sorts the table rows according to the key values. Since the physical order of the rows is the same as the clustered index order, a table can contain only one clustered index.

In SQL SERVER, clustered and non-clustered indexes are both organised as B-trees [14]. The leaf level of the tree contains either data rows (clustered index) or pointers to those rows (non-clustered index).

When creating an index, the sort key order must be defined (ascending or descending). An index can avoid having to use a sort operator to sort the data. Indeed, the table rows can be retrieved from the index directly in the good sort order. The key columns of the index must of course match the columns of the `ORDER BY` clause. However, an index is generally used by the query optimiser to sort a table only if the columns specified in the `ORDER BY` clause are in the same order and in the same sort direction that in the index key.

Without index, all table rows must be examined to check the condition specified in a `WHERE` clause. This is not efficient, particularly when the database contains a lot of data. Since indexes maintain keys in a sorted order, it enables look-up in sub-linear time.

Indexes must be updated when the columns of the corresponding table are updated. Insertion, modification and deletion of rows are thus slower when indexes exist on the table.

The goal of the query optimiser is to find the optimal data access path. Indexes provide new ways to access data. The query optimiser won't necessarily use an index when it is possible. Indeed, a full table scan is sometimes more efficient, depending on data statistics and table sizes for instance.

We added a non-clustered index on each column of the `Jobs` and of the `Destinations` tables, in both sorting orders. In each table, we also added a non-clustered index on each pair of columns, on both sorting orders. In the following sections, we analyse the performance of queries and the effect of indexes.

5.4.2 Filtering

We obtain the same performance as before because indexes are not used. Indeed, the LINQ query given in Section 4.3.2 orders the results by decreasing job `Id`. The database thus travels the table in this order and performs the filtering condition on each line. It doesn't use indexes to filter in this case but we will see how he can use them for filtering in the next section.

5.4.3 Count query option

We perform the same queries as in Table 5.1. The query execution times are shown in Table 5.3. Queries 1 to 4 are executed quicker than previously because the filters use the indexes. They help to find quickly keys satisfying the (in)equality condition since they keep keys ordered.

Query	Type	Filter(s)	Jobs	DB time
#1	double	`MeanByteRate > 15`	300	17
#2	double	`MeanByteRate > 35`	60	3
#3	int	`ShortStatus = Successful`	300	16
#4	double int	`MeanByteRate > 15` `ShortStatus = Successful`	300	17
#6	string	`SourceFileName contains "mp4"`	65	294
#7	string	`SourcePath contains "source"`	300	227
#8	string string	`SourceFileName contains "mp4"` `SourcePath contains "source"`	65	342
#9	string	`Destination.FileName contains "jm"`	300	840
#10	string	`Destination.FileName contains "file5_jm13.flv"`	2.5	787

Table 5.3 Execution time (ms) at the database (with indexes) for several count queries, type of filtered columns and number of jobs (thousands) satisfying the filtering condition.

Queries 6 to 10 are much slower than before. For each job, the database performs an index search to retrieve the value to test and then performs the search of the substring. Performing an index search for each job is clearly much slower than travelling directly the table and examining the values of the filtered columns.

5.4.4 Sorting

Table 5.4 show the execution times of sorting queries. Only the columns of the `Jobs` table can be sorted. Queries 1 to 6 are executed very quickly. Indeed, the keys are already sorted in the indexes. The database can thus travel the index corresponding to the filtered column(s) in the good sorting direction and take the 20 first corresponding jobs. Query 7 is slow because there is no index created on 3 columns and thus a sort operation is needed. The query optimiser didn't use the index used in query 3 to first sort the jobs on the first two columns and then sorting according to the third column.

Query	Type	Sorted column(s)	DB time
#1	double	`MeanByteRate`	3.805
#2	datetime	`StartTime`	3.775
#3	double datetime	`MeanByteRate` `StartTime`	3.769
#4	string	`SourcePath`	3.614
#5	string	`UserName`	3.862
#6	string string	`SourcePath` `UserName`	3.944

#7	double
	datetime
	int
		`MeanByteRate`
		`StartTime`
		`ShortStatus`
			254

Table 5.4 Execution time (ms) at the database (with indexes) for several sort queries.

5.4.5 Sorting and filtering

We measured the same queries as in Table 4.4, where we performed some filtering followed by a sort on `MeanFrameRate` to ensure that the filters examined all the table. The results obtained with OData and indexes are displayed in Table 5.5.

Globally, all execution times are smaller than before. This is because there is no sort operation performed. Indeed, the database first uses the index on `MeanFrameRate` to fetch a set of job identifiers already sorted in the right order. Then, for each returned job identifier, a key look-up is performed on the jobs/destinations table to fetch the corresponding line of the table and to apply the filter. The number of job identifiers fetched in the index is computed based on statistics. Indeed, more identifiers must be returned for filters satisfied by few jobs.

5.4.6 Full-text index

We saw that performing a search on text columns wasn't particularly efficient. Improving that kind of filtering would improve greatly the query-handling database performance. A *Full-text search* is used to search words and phrases in a text [13]. A full-text index defined on a text column cuts the text in words based on rules of a particular language. It is still

Query	Type	Filter(s)	Jobs	DB time
#1	double	`MeanByteRate > 15`	300	3.747
#2	double	`MeanByteRate > 35`	60	3.848
#3	double	`MeanFrameRate > 15`	300	3.889
#4	datetime	`Hour of StartTime > 16`	50	3.980
#5	int	`ShortStatus = Successful`	300	3.482
#6	double int	`MeanByteRate > 15` `ShortStatus = Successful`	300	3.882

#7	double
	int
	datetime
		`MeanByteRate > 15`
		`ShortStatus = Successful`
		`Hour of StartTime > 16`
			50	4.241

#8	datetime
	double
	int
		`Hour of StartTime > 16`
		`MeanByteRate > 15`
		`ShortStatus = Successful`
			50	3.983

#9	double
	datetime
	double
	int
		`MeanFrameRate > 15`
		`Hour of StartTime > 16`
		`MeanByteRate > 15`
		`ShortStatus = Successful`
			50	4.044

#10	string	`SourceFileName contains "jm"`	300	3.926
#11	string	`SourceFileName contains "mp4"`	65	4.130
#12	string	`SourcePath contains "source"`	300	3.911
#13	string string	`SourceFileName contains "mp4"` `SourcePath contains "source"`	65	4.117
#14	string	`Destination.FileName contains "jm"`	300	3.987
#15	string	`Destination.FileName contains "mp4"`	65	6.654

Table 5.5 Execution time (ms) at the database (with indexes) for queries containing several filter combinations followed by a sort on MeanFrameRate, type of filtered columns and number of jobs (thousands) satisfying the filtering condition.

a character-based search but the search begins at the beginning of the words and it is thus quicker to search them.

Full-text indexes could be used to improve the filters on text fields. For instance, the full name of the job initiator (`JobInitiatorFull`) could be cut into words. For fields like `SourceFileName` representing a path and not a phrase, the cut into words is ambiguous and the resulting cut is not satisfactory.

To search words using full-text indexes in SQL Server, the SQL query must use the Transact-SQL predicate `CONTAINS`. However, the SQL statement generated by OData uses the SQL `LIKE` operator. It is thus hard to use full-text index using OData.

The OData protocol defines the `$search` query option that allows to look for terms and phrases and return matching entities [1]. However, the meaning of "matching" depends on the implementation of OData. The .Net OData library doesn't implement the `$search` query option currently [19].

5.5 Indexing strategy

5.5.1 Query performance summary

We can now review the performance obtained using OData and indexes. When we want to obtain a small number of jobs (e.g., 20) corresponding to a filter, the filtering operation is performed quickly, even without indexes.

The `$count` query option requires to filter all the table (`Jobs` or `Destinations`) and to count the resulting number of jobs. This operation is quite fast for all types of columns, excepting when filtering on the `Destinations` table. Indeed, an expensive join operation is needed to count the corresponding number of jobs. This operation takes more time when there is a lot of destinations satisfying the filter condition because the join is bigger. Adding indexes on filtered columns decreases lightly the execution time of `$count` queries filtering on non-text columns. However, the execution time of `$count` queries filtering on text columns becomes very high because indexes are used in an inappropriate way.

When using OData, we lost the optimised sort operator `Top N Sort` and the sorting queries are thus slower and vary more. Adding indexes on sorted columns improves significantly the execution time.

A sort operation is much slower than a filter operation, except when filtering on text columns. Using indexes on queries containing both sorts and filters decreases greatly the execution time. Indeed, the sort is avoided by looking in the index. Then, filters are applied on a small number of jobs.

5.5.2 Indexed columns

Indexes can decrease greatly the query execution time. However, they need to be updated and require storage space. We must thus choose carefully columns on which indexes are created.

We can only filter on destination fields and all those fields are text columns. Filtering on those columns is not too slow. However, if we add indexes on those columns, they are used in a inappropriate way by the `$count` query, so we won't create indexes on those columns.

Sorts and filters are allowed on jobs columns. For non-text columns, indexes improves both filtering and sorting, so we will create indexes on these 6 columns. For text columns, sorts are quicker but we obtain the same performance flaws as for destination columns, so indexes won't be used for non-text job columns.

The MultiMonitoring application allows the client to sort on any combination of columns but we can't create indexes for all possible combinations. Indeed, there would be too much. We will therefore restrict the sort combinations to two columns at most. That won't disturb the user as sorting on more than two columns is not useful in general. So, we can create an index on each non-text `Jobs` table columns, in each sorting direction (ascending or descending). There are $2 \times 6 = 12$ single column indexes. Then, we can create an index for each combination of two columns, taking each column in both sorting orders. The number of permutations of 6 distinct columns taken 2 at a time is given by $P_2^6 = \frac{6!}{(6-2)!} = 30$. For each pair of columns, four sorting orders are possible: (ascending, ascending), (ascending, descending), (descending, ascending), (descending, descending). There are thus $30 \times 4 = 120$ double columns indexes. An SQL statement creating those 132 indexes has been generated by a little script in C# and is available in file `create_indexes.sql`.

Dynamic index creation

Sorting on two jobs may be sometimes not enough. One can imagine creating dynamically indexes for queries sorting on more than two jobs and then deleting those indexes when not used anymore.

We created new indexes in the condition of the setup described in Section 2.4. The creation of an index was quite fast (less than one second) and didn't decrease the number of clients supported by the server. However, this feature has not been implemented.

Job monitor names

When we removed the cache, we lost two information: the number of jobs satisfying the filtering condition (that is now given by the `$count` query option) and the job monitor name of each jobs. We can perform a join with the `JobMonitors` table to obtain this information. This join operation is very fast when performed on a few number of jobs.

5.6 OData limitations

OData is a useful tool because it simplifies the code and provides a rich query language and several interesting features. However, we saw that OData has some limitations. First, it generates SQL statements that don't use the optimised sort operator `Top n Sort`. This results in slower sorting queries whose execution time varies more. Then, OData exposes directly the model corresponding to the EDM. Although we can hide some properties from the entity types, we can't hide completely the EDM. AutoMapper provides this feature but using it decreases strongly the query performances and limit the allowed operations of the query language. Next, we explained that OData defines a `$search` query option that could be used to perform full-text search in order to improve search performance for text fields. However, the .Net OData library doesn't implement this option.

OData became a standard to create, consume and query RESTful APIs and is more and more used. Although OData libraries have currently some limitations, they evolve rapidly.

5.7 Number of supported clients

As for the cache, we can try to find the number of clients that the server can handle at the same time. We measured the same queries as in Section 3.5.2 and defined the scenarios in the file `database_max_clients.scala`. The results are shown in Table 5.6.

Query	Query execution time (ms)	Clients	Clients (with notifications)
#1	30	110	60
#2	30	100	54
#3	18	113	80
#4	190	7	3
#5	260	6	2
#6	30	110	60
#7	25	133	86

Table 5.6 Maximum number of clients for several queries using OData.

The query execution time is the sum of the main query and the `$count` query execution times (because they are executed sequentially). All main queries are executed in less than 10 ms. All `$count` queries are executed in maximum 20 ms, except those of queries 4 and 5. Indeed, they contain filters on text columns and the corresponding `$count` query execution time is high.

As expected, the number of clients reached without notifications is higher than the number of clients reached when notifications are enabled.

We can compare the results with those obtained for the cache. For queries 1, 2, 3, 6 and 7, the obtain numbers of supported clients are higher than those reached with the cache. However, the number of clients reached for queries 4 and 5 are lower than for the cache because of the slow `$count` queries.

5.7.1 Count queries and pagination system

The count queries are used to computes the total number of pages in the pagination system. This number doesn't need to be updated every second. We could update this number every 10 seconds for example and send 10 times less `$count` queries to the server. This will reduce the overall load of the server (especially for `$count` queries filtering on text columns) and thus allow to support more clients. We measure the number of clients handled by the server with this modification. Results are displayed in Table 5.7.

Query	Query execution time (ms)	Clients	Clients (with notifications)
#1	30	181	121
#2	30	179	122
#3	18	184	125
#4	190	115	65
#5	260	102	52
#6	30	182	125
#7	25	194	128

Table 5.7 Maximum number of clients for several queries using OData, with reduced counts frequency.

As expected, reducing the `$count` queries frequency allows to increase the number of clients. The obtained results are much higher than those obtained for the cache.

5.8 Conclusions

OData libraries are useful to create and consume queryable RESTful APIs. We showed how we used the OData .NET library with the MultiMonitoring application. Then, the performance of OData has been measured. The main difference between queries generated by OData and hard-coded LINQ to Entities queries is that OData doesn't use the optimised sort operator. Hopefully, the use of indexes helped to speed-up the queries. By limiting the frequency of count queries, the server can handle more clients than the version of the server using the cache.

Chapter 6

Conclusion

We reviewed the data handling of the Multi-Monitoring application in order to remove any barrier preventing the concurrent execution of multiple instances of the service while improving the performance.

The main obstacle to concurrent execution was that the caching infrastructure didn't provide synchronisation between the cache and the database. Moreover, the cache was implemented by a set of lists. This data structure lead to unpredictable performance when handling various queries.

We showed that the database could contain all the jobs in its cache and thus perform queries in memory like the in-memory cache at the server. The database executed most of the queries in a more efficient and predictable way than the cache. Indeed, the database contains a query optimiser computing the best way to execute a query based on statistics. In particular, the query execution plan contained an optimised sort operator that avoids to sort completely the table of jobs.

Based on those results, we decided to remove the cache and perform queries directly on the database. This removal simplified and shortened the code of the application. However, there was no mechanism of translation of the query contained in the URI to the database. Queries contained in the URI were initially translated to query expressions applied to the list. However, this translation mechanism was not easily adaptable to query the database. We thus used OData, a tool providing a query language for RESTful APIs, that can translate automatically query parameters from the URI to SQL queries. Yet, OData came with some performance flaws. To restore performances, we used indexes to speed up queries execution and we restricted query parameters combinations.

This work resulted in a solution allowing the concurrent execution of multiple instances of the service. Moreover, queries are in most cases faster and thus the service can now support more clients.

The performance of the Multi-Monitoring application could still be improved. Indeed, the filtering of string columns is the weak point when querying the database. Although tested, some features improving performance still need to be implemented, such as the dynamic creation of indexes and the adaptive count of jobs.

References

[1] (2015). Odata - the best way to rest. www.odata.org.

[2] Bogard, J. (2016). Automapper official web site. http://automapper.org/.

[3] Chappell, D. (2011). Introducing odata. https://msdn.microsoft.com/en-us/data/hh237663.aspx.

[4] Corp, G. (2016). Gatling official web site. http://gatling.io.

[5] EVS (2016a). Evs xt3 servers power beijing television's new 3d production truck. https://evs.com/en/news/evs-xt3-servers-power-beijing-televisions-new-3d-production-truck.

[6] EVS (2016b). Kbs launched new hd ob truck fully equipped with evs systems for iaaf world championships in daegu. https://evs.com/en/news/kbs-launched-new-hd-ob-truck-fully-equipped-evs-systems-iaaf-world-championships-daegu.

[7] EVS (2016c). Xsquare 03.06 user's manual. https://evs.com/en/download-area-version/3091.

[8] Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine.

[9] Microsoft (2016a). Entity data model key concepts. https://msdn.microsoft.com/en-us/library/ee382840%28v=vs.100%29.aspx.

[10] Microsoft (2016b). Entity framework. https://msdn.microsoft.com/en-us/data/ef.aspx.

[11] Microsoft (2016c). Sql server profiler. https://msdn.microsoft.com/fr-fr/library/ms181091%28v=sql.120%29.aspx.

[12] Microsoft (2016d). Top n sort showplan operator. https://technet.microsoft.com/en-us/library/ms189054%28v=sql.105%29.aspx.

[13] Mirosoft (2016a). Full-text search. https://msdn.microsoft.com/en-us/library/ms142571.aspx.

[14] Mirosoft (2016b). Guide de conception d'index sql server. https://msdn.microsoft.com/fr-fr/library/jj835095

[15] Mirosoft (2016c). Linq (language-integrated query). https://msdn.microsoft.com/fr-be/library/bb397926.aspx.

[16] OASIS (2014). Oasis open data protocol (odata) tc. www.oasis-open.org/committees/tc_home.php?wg_abbrev=odata.

[17] Rusanu, R. (2013). Understanding how sql server executes a query. http://rusanu.com/2013/08/01/understanding-how-sql-server-executes-a-query/.

[18] sqlity.net (2012). Top n sort – a little bit of sorting. http://sqlity.net/en/908/top-n-sort-a-little-bit-of-sorting/.

[19] Wasson, M. (2014). Odata support in asp.net web api. www.asp.net/web-api/overview/odata-support-in-aspnet-web-api.

[20] White, P. (2010). Sorting, row goals, and the top 100 problem. http://sqlblog.com/blogs/paul_white/archive/2010/08/27/sorting-row-goals-and-the-top-100-problem.aspx.