Università Politecnica delle Marche

Facoltà di Ingegneria

Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

Applicazione di tecnologie Big Data per l'interrogazione di un Data Warehouse aziendale

Application of Big Data technologies for querying an enterprise Data Warehouse

Relatore
Domenico Potena

Candidato
Mattia Scuriatti

Anno Accademico 2022–2023

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Il progetto in questione è stato realizzato presso Esisoftware, una società facente parte del gruppo QS-Group.

Il gruppo QS-Group costruisce macchine industriali automatizzate, impianti e stampi per la lavorazione della lamiera, plastica e poliuretano espanso, sistemi di movimentazione prodotti e magazzini automatizzati; l'azienda sviluppa anche software per l'acquisizione, l'elaborazione e la gestione dei dati di produzione.

L'azienda Esisoftware realizza soluzioni informatiche integrate a supporto della digitalizzazione della Supply Chain. L'insieme dei prodotti offerti comprende le seguenti soluzioni:

- SCADA (Supervisory Control And Data Acquisition): per il controllo dei macchinari attraverso la disponibilità di dati real-time e la progettazione di interfacce su misura;
- MES (Manufacturing Execution System): ha come obiettivo seguire le fasi di produzione per informatizzare i flussi operativi ed eliminare errori dovuti a gestioni inefficaci;
- WMS (Warehouse Management System): è uno strumento integrato per la gestione dei flussi logistici di magazzino.

Questa suite forma un ecosistema digitale progettato in prospettiva dell'Industrial Internet of Things (IIoT) che garantisce l'affidabilità del dato, grazie al supporto di strumenti in real-time. Esisoftware fornisce strumenti che garantiscono un processo di miglioramento continuo per la riduzione degli errori, la pianificazione e il monitoraggio della produzione e per la centralizzazione delle informazioni, così da facilitare la comunicazione tra i vari dipartimenti tramite l'utilizzo di dispositivi mobile.

1.1 Problema dell'azienda

L'azienda ha riscontrato alcune inefficienze all'interno di un proprio applicativo nella fase di interrogazione di un datawarehouse attraverso SQL Server. In sintesi, l'estrazione dei dati in determinate circostanze comporta tempi non accettabili; questo nel momento in cui i dati convergono verso quantità associabili al mondo Big Data (ordine di milioni di record).

1.2 Obiettivi

Lo scopo di questo elaborato è quello di introdurre nell'applicativo il sistema Apache Spark, al fine di ottimizzare il processo di interrogazione dei dati. Quest'integrazione, non deve alterare in alcun modo né il risultato delle richieste né la forma di sottoscrizione. Una volta soddisfatti questi requisiti, l'ultima fase prevede la conduzione di molteplici test. Questi, porranno l'attenzione sui tempi di esecuzione al variare di alcuni fattori, con l'obiettivo di ottenere un confronto finale tra la soluzione in essere ("SQLServer-based") e quella con Spark.

1.3 Struttura dell'elaborato

L'elaborato è stato strutturato come segue:

- Capitolo 2: in cui verrà illustrata la soluzione in essere;
- Capitolo 3: in cui verranno introdotte le tecnologie usate;
- Capitolo 4: in cui verrà descritta l'integrazione delle suddette tecnologie all'interno della soluzione;
- Capitolo 5: in cui verranno mostrati i test effettuati per confrontare le prestazioni pre e post modifica;
- Capitolo 6: in cui verranno raccolti i risultati dei test per fornire un quadro finale.

Capitolo 2

Soluzione aziendale

La soluzione aziendale presa in esame consta di una web application che interagisce con un datawarehouse. Il suo scopo è offrire un servizio di reportistica inerente a delle linee di collaudo di un cliente. Sostanzialmente, queste linee effettuano dei test su dei prodotti e i dati generati vanno a popolare il datawarehouse. Il cliente, interagendo con una pagina web, sarà in grado di interloquire con l'applicativo interrogando tale base di dati, al fine di estrarre informazioni utili all'attività di analisi. Per quanto riguarda la "data ingestion", la base di dati viene aggiornata su base giornaliera con i dati del giorno precedente attraverso un "job" sviluppato internamente.

2.1 Datawarehouse

Il datawarehouse in questione è quello rappresentato dal diagramma ER riportato in Figura 2.1.

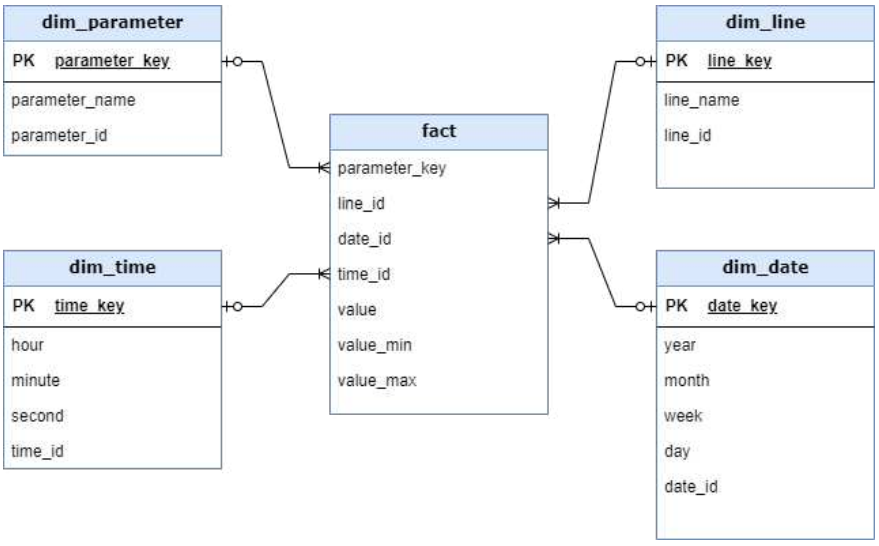

Figura 2.1: Diagramma ER del datawarehouse

La struttura multidimensionale dei dati è rappresentata utilizzando il modello logico ROLAP (Relational On-Line Analytical Processing), il quale utilizza il modello relazionale per implementare questa rappresentazione. Tale modellazione è basata su uno schema a stella così composto:

- un insieme di tabelle (denormalizzate), chiamate "tabelle delle dimensioni". Ciascuna di queste tabelle è caratterizzata da una chiave primaria e da un insieme di attributi che descrivono le dimensioni di analisi a diversi livelli di aggregazione;
- una "tabella dei fatti" in cui sono presenti le chiavi di tutte le tabelle delle dimensioni. La chiave primaria di questa tabella sarà data dall'insieme delle chiavi esterne delle dimensioni. La tabella dei fatti contiene un attributo per ogni misura, ognuno dei quali descrive l'evento di business di interesse per le analisi.

La visione multidimensionale si ottiene eseguendo il "join" tra la tabella dei fatti e le tabelle delle dimensioni. Di seguito verranno descritte tutte le tabelle.

Tabella fact La tabella dei fatti ha 7 colonne, delle quali 4 sono le chiavi delle dimensioni e 3 sono le misure:

- parameter_id: chiave naturale della dimensione parameter;
- line_id: chiave naturale della dimensione line;
- date_id: chiave naturale della dimensione date;
- time_id: chiave naturale della dimensione time;
- value: valore della misurazione effettuata (relativa ad un certo parametro);
- value_min: valore minimo;
- value_max: valore massimo.

Una selezione della fact è riportata in Tabella 2.1.

parameter_id	date_id	line_id	time_id	value	value_min	value_max
PRESS_HM	2023-04-03	1	09:00:27	56	0	0
PRESS_HM	2023-04-03	1	09:00:28	38	0	0

Tabella 2.1: Selezione della tabella fact

Tabella dim_line La dimensione linea ha 3 colonne:

- line_key: chiave primaria surrogata;
- line_id: chiave naturale;
- line_name: nome della linea.

Una selezione della dim_line è riportata in Tabella 2.2.

line_key	line_id	line_name
1003	1	TUB_LINE_1

Tabella 2.2: Selezione della tabella dim_line

Tabella dim_parameter La dimensione parametro ha 3 colonne:

- parameter_key: chiave primaria surrogata;
- parameter_id: chiave naturale;
- parameter_name: nome del parametro acquisito.

Una selezione della dim_parameter è riportata in Tabella 2.3.

parameter_key	parameter_id	parameter_name
1	TEMP_BIT	Temperature Bitumen
2	PRESS_BIT	Pressure Bitumen
3	TEMP_HM	Temperature HotMelt Glue
4	PRESS_HM	Pressure HotMelt Glue

Tabella 2.3: Selezione della tabella dim_line

Tabella dim_date La dimensione data ha 6 colonne:

- date_key: chiave primaria surrogata;
- date_id: chiave naturale;
- year: anno dell'acquisizione;
- month: mese dell'acquisizione;
- week: settimana dell'acquisizione;
- day: giorno dell'acquisizione.

Una selezione della dim_date è riportata in Tabella 2.4.

date_key	date_id	year	month	week	day
101542	2023-01-01	2023	1	1	1
101543	2023-01-01	2023	1	2	2
101544	2023-01-01	2023	1	2	3

Tabella 2.4: Selezione della tabella dim_date

Tabella dim_time La dimensione orario ha 5 colonne:

- time_key: chiave primaria surrogata;
- time_id: chiave naturale (identifica l'orario nel formato HH:MM:SS);
- hour: ora dell'acquisizione;
- minute: minuto dell'acquisizione;
- second: secondo dell'acquisizione.

Una selezione della dim_time è riportata in Tabella 2.5.

time_key	time_id	hour	minute	second
1	0	0	0	00:00:00
2	0	0	1	00:00:01
3	0	0	2	00:00:02

Tabella 2.5: Selezione della tabella dim_time

Tavola dei volumi Il Datawarehouse è contenuto in un'istanza SQL Server all'interno di una macchina aziendale. In origine il numero di righe della fact era di circa 4 milioni di record (relativo ad un periodo di acquisizione di circa quattro mesi). Per operare in un contesto più adatto alle tecnologie Big Data, è stato portato a circa 150 milioni di record (duplicando i dati nel tempo), simulando così un periodo di acquisizione pari a 1 anno e 4 mesi. La cardinalità di righe ottenute sono esposte nella Tabella 2.6.

Tabella	Numero di righe
fact	152.510.624
dim_line	1
dim_parameter	191
dim_date	18.629
dim_time	86.400

Tabella 2.6: Tabella dei volumi

2.2 Web Application

.NET L'applicativo è stato sviluppato all'interno di .NET, un framework creato da Microsoft per il proprio sistema operativo Windows. .NET fa riferimento ad un ecosistema composto da linguaggi, librerie, ambiente di esecuzione e sviluppo:

- Linguaggi: C# (usato nell'applicazione corrente), F# e Visual Basic (VB);
- Ambiente di esecuzione (runtime): Common Language Runtime (CLR);
- Ambiente di sviluppo: Visual Studio;
- Libreria: Base Class Library (BCL), è possibile ampliare le funzionalità grazie a NuGet, il gestore di pacchetti di .NET (l'equivalente di npm per Node.js).

.NET utilizza il concetto di compilazione just-in-time (JIT), che consente di compilare il codice al momento dell'esecuzione. Ciò significa che il codice sorgente viene scritto in un linguaggio ad alto livello come C# e poi compilato in un codice intermedio, chiamato Microsoft Intermediate Language (MSIL), eseguito sulla piattaforma .NET utilizzando il JIT. Questo implica che .NET è in grado di eseguire il codice su qualsiasi sistema operativo, purché sia installata la versione corretta della piattaforma. .NET nasce dall'unione dei seguenti strumenti:

- .NET Framework : fornisce l'accesso alle funzionalità generali di Windows e Windows Server. Usato anche ampiamente per il cloud computing basato su Windows;
- Mono e Xamarin: implementazioni di .NET per contesti mobile;
- .NET Core: è stato il successore multi-piattaforma e open source di .NET Framework.

Inoltre, la piattaforma .NET offre un framework per la creazione di app e servizi web con ASP.NET.

Dependency Injection Un design pattern supportato da .NET è la "dependency injection". L'obiettivo di questa tecnica, è quello di ottenere l'inversione del controllo (IoC) tra le classi e le relative dipendenze. Tale principio ha la seguente accezione: partendo dal concetto di dipendenza della programmazione ad oggetti, è possibile invertirne la direzione a "compile time". Ciò che vuole essere evitato è l'esistenza di dipendenze "hard

coded", le quali generano un grafo delle dipendenze diretto tra le classi (Figura 2.2).

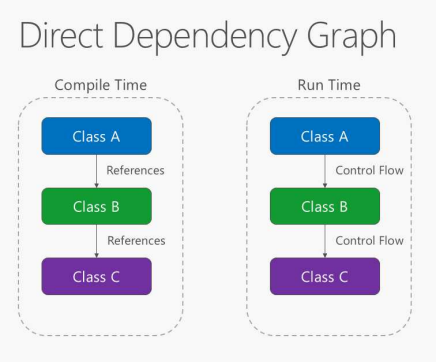

Figura 2.2: Grafo delle dipendenze diretto

L'inversione del controllo risolve questa criticità introducendo delle astrazioni (interfacce) e sostituendole alle classi vere e proprie nei riferimenti (Figura 2.3). A "Run Time", il flusso di esecuzione del programma rimane invariato, ma l'introduzione delle interfacce dà la possibilità di collegare facilmente diverse implementazioni.

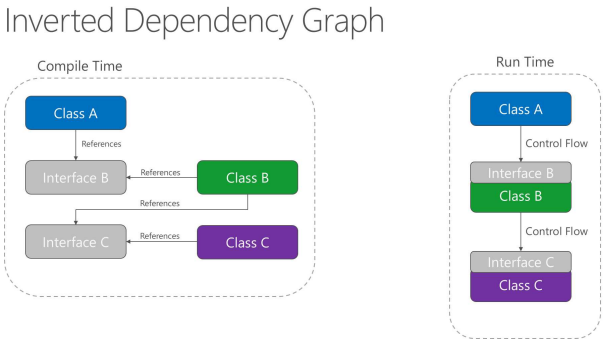

Figura 2.3: Grafo delle dipendenze invertite

La registrazione di queste dipendenze avviene all'interno di un container denominato "IoC container". I tipi (classi) gestiti dal contenitore sono chiamati servizi, questi possono far parte del framework ASP.NET oppure essere creati dal programmatore. L'inversione delle dipendenze è una parte fondamentale della creazione di applicazioni liberamente accoppiate, poiché i dettagli di implementazione possono essere scritti in modo che dipendano e implementino astrazioni di livello superiore, anziché il contrario. Di conseguenza, le applicazioni risultanti sono più testabili, modulari e gestibili.

Visual Studio Per introdurre l'applicativo, è necessario comprendere alcuni aspetti anche dell'ambiente di sviluppo. Ogni progetto di Visual Stu-

dio include un file di progetto MSBuild (un motore per creare applicazioni), con un'estensione che riflette il tipo di progetto; ad esempio un progetto C# (come quello trattato) ha estensione *csproj*. Per compilare un progetto, MSBuild deve elaborare il file di progetto associato. Quest'ultimo è un documento XML contenente tutte le informazioni e le istruzioni richieste da MSBuild per compilare il progetto. La struttura di base di un file di progetto MSBuild è riportata in Figura 2.4.

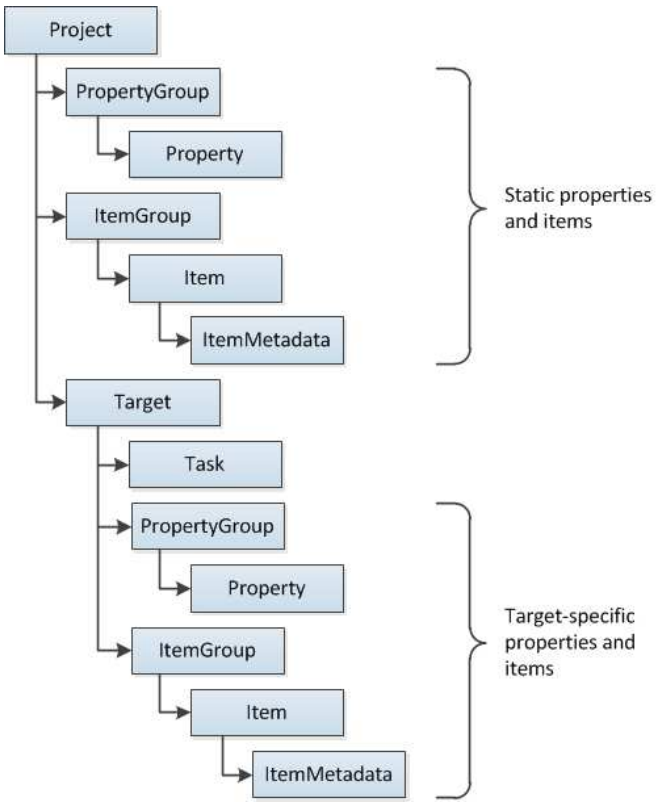

Figura 2.4: Struttura di un file di progetto MSBuild

L'elemento Project è la radice di ogni file di progetto e può includere attributi aggiuntivi, ad esempio per specificare i punti di ingresso per il processo di compilazione. Gli elementi Property sono delle informazioni che possono includere nomi di server, stringhe di connessione, credenziali, configurazioni di compilazione, percorsi di file di origine e di destinazione e qualsiasi altra informazione da includere per supportare la personalizzazione. Sono definite all'interno di un elemento PropertyGroup e sono costituite da coppie chiave-valore: il nome dell'elemento definisce la chiave della proprietà e il contenuto il valore. Le proprietà vengono spesso usate in combinazione con delle condizioni (attraverso l'attributo Condition), che consentono di specificare i criteri in base ai quali MSBuild deve valutare l'elemento. Uno dei ruoli importanti del file di progetto consiste nel definire gli input per il processo di compilazione. In genere, questi input sono file,

file di codice, file di configurazione o file di comandi. Nello schema del progetto MSBuild questi input sono rappresentati dagli elementi Item. In un file di progetto, tali Item devono essere definiti all'interno di un elemento ItemGroup. Un Task invece, rappresenta una singola istruzione di compilazione (o attività). MSBuild include una moltitudine di attività predefinite, ad esempio:

- L'attività Copy copia i file in un nuovo percorso;
- L'attività Csc richiama il compilatore Visual C#;
- L'attività Vbc richiama il compilatore Visual Basic;
- L'attività Exec esegue un programma specificato;
- L'attività Message scrive un messaggio in un logger.

Gli elementi Task devono essere contenuti negli elementi Target.

Swagger Swagger è un insieme di strumenti che semplificano il processo di produzione e documentazione dell'API. La documentazione fornisce istruzioni su come utilizzare e integrare un'API in modo efficace con dettagli su funzioni, classi, tipi restituiti, argomenti e molto altro. Swagger è composto da due parti. La prima è la specifica, un file *swagger.json* che viene creato automaticamente da Swagger (partendo dai percorsi, controller e modelli dell'applicazione) e che contiene le informazioni delle API.

```
{
  "openapi": "3.0.1",
  "info": {
    "title": "EsiHistorian.WebApi",
    "version": "v0.0.1"
  },
  "paths": {
    "/api/Fact/GetGroupedFacts": {
      "post": {
        "tags": [
          "Facts"
        ],
        "summary": "Get Facts data grouped by \r\nYear-Month-Week-Day-Hour-
   Minute-Second\r\nAnd filtered",
        "operationId": "Facts_GetGroupedFacts",
        "requestBody": { .... }
        },
        "responses": {
          "200": {
            "description": "Success"
          }
        }
```

```
      }
    }
}
```

La seconda parte si chiama Swagger UI, un'interfaccia utente generata dalla specifica attraverso cui è possibile verificare il funzionamento dei metodi, il tipo di risposte e le informazioni sui parametri di ingresso. [9] (Figura 2.5).

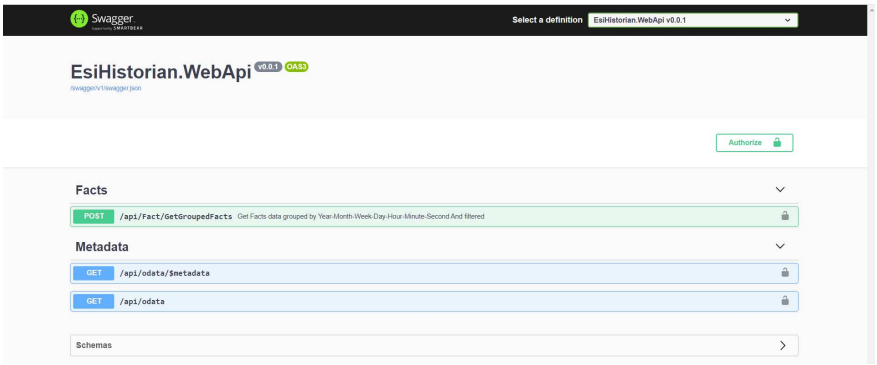

Figura 2.5: SwaggerUI dell'applicazione web

SqlKata Per interrogare il datawarehouse è stata utilizzata la libreria SqlKata, un generatore ed esecutore di query SQL in C# (importata attraverso il gestore dei pacchetti NuGet). Per interagire con la base di dati viene richiesta, inizialmente viene richiesta l'installazione del driver per interfacciarsi con SQL Server. Fatto ciò, per eseguire le interrogazioni, si deve invocare il metodo *Query()* il quale richiede due parametri: l'istanza della connessione al database e il compilatore. É possibile semplificare l'esecuzione del metodo *Query()* attraverso una *QueryFactory*. Questo oggetto ci evita di mantenere ogni volta i riferimenti della connessione (classe *SqlConnection*) e del compilatore (classe*SqlServerCompiler*). Un esempio che illustra tale vantaggio è il seguente.

```
//Senza QueryFactory
using var connection = new SqlConnection(
    "Host=localhost;Port=3306;User=user;Password=secret;Database=Users;
    SslMode=None"
);

var compiler = new SqlServerCompiler();

var users = new Query(connection, compiler).From("Users").Get();
-------------------------------------------------------------------------
//Con QueryFactory
using var connection = new SqlConnection(
    "Host=localhost;Port=3306;User=user;Password=secret;Database=Users;
    SslMode=None"
```

```
);

using var db = new QueryFactory(connection, new SqlCompiler());

//Da adesso si utilizza esclusivamente ‘db.Query()‘
var users = db.Query("Users").Get();
```

Sui progetti Asp.Net, è possibile configurare il contenitore IoC per risolvere le istanze necessarie della QueryFactory (dentro il metodo *ConfigureServices()* di Startup.cs che verrà descritta più avanti).

```
services.AddTransient<QueryFactory>((serviceProvider) => {
        string connectionString = Configuration.GetConnectionString("
    DefaultConnection");

        var connection = new SqlConnection(connectionString);
        var compiler = new SqlServerCompiler();
        var queryFactory = new QueryFactory(connection, compiler);

        return queryFactory;
    });
```

Un servizio "Transient" implica che viene creata una nuova istanza del servizio per ciascun oggetto nella richiesta HTTP. Il servizio aggiunto è di tipo QueryFactory. La "callback" inizia con l’estrazione della stringa di connessione (in *connection*) e l’inizializzazione di un’istanza della classe *SqlConnection(connection)*. Successivamente viene creato il compilatore per SQL Server che avrà la responsabilità di trasformare *Query()* in una stringa SQL che può essere eseguita direttamente dal motore del database. Fatto ciò, viene inizializzato e ritornato l’oggetto della classe.

2.2.1 Struttura

La web application in questione è suddivisa in molteplici progetti. Tale suddivisione è espressa all’interno di un file soluzione denominato *EsiHistorian.sln*: un contenitore per organizzare più progetti correlati. Una volta aperto, il file soluzione permetterà a Visual Studio di caricare automaticamente tutti i progetti espressi all’interno. Il listato sottostante è una porzione del file soluzione dell’applicazione. Questo contiene diverse informazioni per ogni progetto: il GUID (Globally Unique Identifier), il nome, il file di progetto (.csproj) e il GUID relativo al tipo di progetto. Queste informazioni vengono utilizzate dall’ambiente per trovare il file o i file di progetto appartenenti alla soluzione.

```
Project("{9A19103F-16F7-4668-BE54-9A1E7A4F7556}") = "EsiHistorian.
    BusinessLayer", "EsiHistorian.BusinessLayer\EsiHistorian.BusinessLayer.
    csproj", "{6AFA595F-ED18-4451-AED0-2C5D193C9AF2}"
EndProject
```

```
Project("{9A19103F-16F7-4668-BE54-9A1E7A4F7556}") = "EsiHistorian.Dto", "
    EsiHistorian.Dto\EsiHistorian.Dto.csproj", "{31C17D40-78D0-46AC-BE1F-76
    FEC324A236}"
EndProject
Project("{9A19103F-16F7-4668-BE54-9A1E7A4F7556}") = "EsiHistorian.Entity",
    "EsiHistorian.Entity\EsiHistorian.Entity.csproj", "{709FD0DA-285A-4D86-8
    B65-BF27185F376A}"
EndProject
```

La struttura dell'applicazione è stata derivata da un template aziendale. Questo template è organizzato per applicazioni conformi al "repository pattern", un modello di progettazione che ha come scopo quello di realizzare un'astrazione tra l'accesso ai dati e la logica di business (ottenendo interoperabilità nella manutenzione e nello sviluppo). I progetti inclusi sono i seguenti:

- EsiHistorian.BusinessLayer: adibito alla logica di business;
- EsiHistorian.DB: per la gestione del database;
- EsiHistorian.Dto: per la gestione del trasferimento degli oggetti;
- EsiHistorian.Entity: per la definizione delle entità (rappresentano le istanze degli oggetti di dominio contenuti nel database);
- EsiHistorian.EntityFrameworkCore: per la definizione delle interfacce dei repository e il contesto del database, gestisce anche le migrazioni delle tabelle;
- EsiHistorian.LoginManagement: per la gestione degli accessi;
- EsiHistorian.Mocks: per l'esecuzione di test anche in assenza della sorgente dati;
- EsiHistorian.Proxy: generato da AutoRest, un framework di generazione di codice per convertire le specifiche Swagger in librerie client per i servizi descritti da tali specifiche;
- EsiHistorian.Repository: per l'implementazione delle interfacce definite in EsiHistorian.EntityFrameworkCore;
- EsiHistorian.SignalRHelper: permette l'aggiunta di funzionalità Web in tempo reale tra client e server;
- EsiHistorian.WebAPI: contiene l'applicazione web.

Tutti questi progetti registrano le dipendenze secondo un'architettura a cipolla, un modello architetturale in cui il software viene scomposto in livelli concentrici (ognuno contenente uno o più progetti). La principale differenza con la "classica" architettura stratificata è che ogni livello esterno

vede tutti i livelli interni, non solo quello direttamente sottostante (evitando quindi un accoppiamento forte). Inoltre, la direzione di dipendenza va sempre dall'esterno verso l'interno, mai viceversa. Nello specifico, il nome di qualcosa che è dichiarato in un cerchio esterno, non deve essere menzionato in un cerchio più interno. Questo ragionamento vale per le funzioni, le classi, le variabili e qualsiasi altra entità software dotata di un nome [10].

2.2.2 Distribuzione

Per comprendere meglio alcune logiche dell'implementazione (Sezione 2.2.3) è opportuno descrivere il processo di "deploy". L'applicazione viene distribuita attraverso la procedura di pubblicazione fornita da .NET, che può avvenire in due modalità:

- framework-dependent: produce in uscita solo l'applicazione stessa e le relative dipendenze. Dunque è necessario installare separatamente il runtime .NET;

- self-contained: la cartella di pubblicazione include il runtime e le librerie .NET, nonché l'applicazione e le relative dipendenze. Gli utenti possono eseguirla anche su ambienti in cui non è installato il runtime .NET (ovviamente la dimensione della cartella di pubblicazione sarà maggiore).

Una volta pubblicato (nella forma desiderata) e distribuito, l'applicativo viene lanciato attraverso il promt dei comandi oppure installato ed eseguito come un servizio Windows. Questa seconda alternativa è più vantaggiosa, in quanto i servizi Windows consentono di creare applicazioni eseguibili di lunga durata, in esecuzione in sessioni Windows dedicate. Questi servizi possono essere avviati automaticamente all'avvio del computer, possono essere sospesi e riavviati. Queste funzionalità rendono i servizi ideali per l'utilizzo all'intero di server o per fornire funzionalità di lunga durata che non interferiscono con gli altri utenti che lavorano sullo stesso computer (casi tipici dei clienti). Per creare un servizio Windows viene usato Topshelf, un framework di hosting scritto utilizzando il framework .NET. Topshelf semplifica la creazione di servizi, consentendo agli sviluppatori di creare una semplice applicazione console che può essere installata come un servizio Windows. In aggiunta, una volta che l'applicazione è stata testata e pronta per la produzione, Topshelf semplifica l'installazione dell'applicazione come servizio rispetto al classico approccio dei servizi Windows.

2.2.3 Implementazione

Come accennato, l'azienda ha derivato la struttura da un template. All'effettivo però, i progetti nei quali è stata immessa della logica sono:

- EsiHistorian.WebAPI;
- EsiHistorian.Dto;
- EsiHistorian.BusinessLayer.

EsiHistorian.WebAPI Il progetto EsiHistorian.WebAPI è una API Web ASP .NET Core di tipo REST, ossia basata sui seguenti principi:

- Identificazione delle risorse: le risorse (gli oggetti) sono gli elementi fondamentali (nell'approccio SOAP la base sono le chiamate remote);
- Utilizzo esplicito dei metodi HTTP: sfruttiamo una mappatura tra i metodi HTTP (GET, POST, PUT e DELETE) e le operazioni CRUD;
- Risorse autodescrittive: non viene posto nessun vincolo sulle modalità di rappresentazione di una risorsa;
- Comunicazione "stateless": ciascuna richiesta non ha alcuna relazione con le richieste precedenti e successive.

La directory di progetto ha la conformazione illustrata in Figura 2.6.

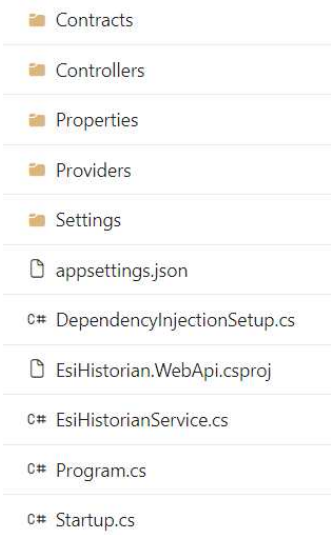

Figura 2.6: Directory del progetto EsiHistorian.WebAPI

Il file Program.cs contiene il *Main()*, il punto d'ingresso del programma. Dentro al metodo vengono eseguite le seguenti istruzioni.

```
config = new ConfigurationBuilder()
                .SetBasePath(pathToContentRoot)
                .AddJsonFile("appsettings.json")
                .Build();

HostFactory.Run(x =>
{
    x.Service<EsiHistorianService>();
});
```

La prima istruzione crea un oggetto *config* nel quale viene mappato il file *appsetting.json*. Questo permette l'estrazione dei parametri di configurazione contenuti nel file. Il secondo metodo è un API fornita da TopShelf (aggiunto tra i pacchetti NuGet) per configurare l'host del servizio. L'argomento è una funzione lambda che definisce e avvia un servizio di tipo *EsiHistorianService*. Il tipo in esame, fa riferimento alla classe omonima (contenuta in EsiHistorianService.cs) che implementa i metodi *Start()* e *Stop()* definiti dall'interfaccia da cui eredita: *ServiceControl* (di Topshelf). Il metodo *Start()* preliminarmente crea un "web host" attraverso il "builder design pattern".

```
WebHost.CreateDefaultBuilder()
                .UseStartup<Startup>()
                .UseConfiguration(config)
                .UseEsiLogger()
                .UseUrls(urls);
```

Il codice sopra definisce:

- il tipo di avvio del "web host": classe *Startup* (definita in Startup.cs);
- l'utilizzo delle impostazioni di configurazione contenute in *config*;
- il provider di logging (*Serilog*);
- l'indirizzo web dove l'applicazione sarà raggiungibile, contenuto nell'oggetto *urls*: `http://localhost:5000` (viene utilizzato il web server predefinito delle app ASP.NET: Kestrel).

Dopodiché, l'applicazione viene eseguita per via del metodo *RunAsync()* dell'oggetto *host*, il quale mappa la configurazione appena illustrata. Il tipo di avvio del "web host" come detto, è definito dalla classe *Startup*. Questa espone due metodi:

- *ConfigureServices()*: consente di registrare i servizi applicativi all'interno del contenitore IoC. Dopo aver registrato la classe dipendente, è possibile utilizzarla includendola come parametro del costruttore di una classe in cui si vuole usare. Il contenitore IoC la inietterà automaticamente (dependency injection);

- *Configure()*: configura la "pipeline" delle richieste HTTP (abilita i middleware per il CORS, per l'autenticazione, per Swagger/SwaggerUI, per il "routing", per l'autorizzazione ecc..).

In fase di esecuzione, il metodo *ConfigureServices()* viene chiamato prima di *Configure()*. In questo modo, vengono preliminarmente registrati i servizi e poi utilizzati all'interno della *Configure()*. Un'ulteriore meccanismo di "iniezione delle dipendenze" viene messo in pratica dalla classe *DependencyInjectionSetup* (definita nell'omonimo file). La differenza tra questo e il precedente è il contenitore IoC utilizzato. In *DependencyInjectionSetup* viene utilizzato Autofac, rispetto al container di default di Microsoft, il quale offre ulteriori funzionalità (non oggetto di questo elaborato). Un altro aspetto da introdurre nella gestione delle API sono i contratti. Un contratto è un accordo tra client e server, per sancire delle regole nello scambio dei dati. L'esempio più lampante è la dichiarazione dei campi obbligatori che una richiesta deve fornire (generando un errore in caso di mancato rispetto). La classe *FactModel* (nella cartella Contracts) gestisce questo aspetto, mappando le chiavi del "body" della richiesta (formato JSON) in omonime proprietà, indicandone l'obbligatorietà attraverso l'attributo *[Required]*.

```
[Required]
public StringDateTime TotalDate { get; set; }
[Required]
public StringDateTime Year { get; set; }
[Required]
public StringDateTime Month { get; set; }
[Required]
public StringDateTime Week { get; set; }
[Required]
public StringDateTime Day { get; set; }
[Required]
public StringDateTime Hour { get; set; }
[Required]
public StringDateTime Minutes { get; set; }
[Required]
public StringDateTime Seconds { get; set; }

[Required]
public StringDimension[] Dimensions { get; set; }

public string FactName { get; set; }
}
```

I tipi personalizzati StringDateTime e StringDimension[] sono dichiarati come classi in *CommonModel.cs* (all'interno della cartella Contracts); tali classi esplicitano le proprietà di ciascuno dei tipi.

```
public class StringDimension
{
    public bool GroupBy { get; set; }
    public string Filter { get; set; }
    public string DimensionName { get; set; }
    public string[] DimensionFields { get; set; }
}

public class StringDateTime
{
    public bool GroupBy { get; set; }

    [JsonConverter(typeof(LocalDataFormatJsonConverter))]
    public DateTime FilterFrom { get; set; }
    [JsonConverter(typeof(LocalDataFormatJsonConverter))]
    public DateTime FilterTo { get; set; }
}
```

StringDimension definisce:

- GroupBy: booleano che indica la volontà o meno di raggruppare la misura per una dimensione (non temporale);
- Filter: stringa che indica il valore del campo "*dimensione*_id" su cui filtrare (es: line_id = 1);
- DimensionName: stringa che indica il nome della dimensione;
- DimensionFields: campi della dimensione che verranno restituiti.

StringDateTime definisce:

- GroupBy: booleano che indica la volontà o meno di raggruppare la misura per una certa granularità della dimensione tempo (es: Anno, Mese ecc...);
- FilterFrom e FilterTo: rappresentano gli istanti temporali (tipo DateTime in formato "YYYY-MM-DD'T'HH:MM:SS.SSS'Z'") all'interno dei quali filtrare i dati. L'attributo *[JsonConverter(typeof(LocalDataFormatJsonConverter))]* converte l'oggetto *DateTime* nell'ora locale associata al sistema operativo.

La gestione delle richieste HTTP è demandata alla classe *FactController* (nella cartella Controllers) al seguente segmento URL: `http://localhost:5000/api/Fact` (producendo un "JSON object" in risposta). La classe possiede un campo *_db* con il modificatore *readonly* (l'assegnazione può essere eseguita solo come parte della dichiarazione o in un costruttore nella stessa classe) e di tipo *QueryFactory*. L'unica rotta definita è di tipo POST e prende il nome di *GetGroupedFacts*.

```
[HttpPost]
[Route("GetGroupedFacts")]
public IActionResult GetGroupedFacts([FromBody] FactModel request)
{
    if (!ModelState.IsValid)
        return BadRequest(ModelState);

    var request_dto = (FactDTO)request;
    var businessQL = new QueryBL();

    var result = businessQL.GetGroupedFactsLogic(_db, request_dto);

    return Ok(result);
}
```

La rotta prende come parametro il "body" della richiesta (di tipo *FactModel*). L'istruzione condizionale che verifica il valore di *ModelState.IsValid* serve per controllare l'associazione con il modello ("bind" con la richiesta) e anche la sua convalida (controlla campi obbligatori). In tal caso, l'applicazione ritornerà al client un "400 bad request error": un codice di stato HTTP che descrive un errore causato da una richiesta non valida. Nel caso in cui venga appurata la correttezza della richiesta, viene effettuata un'operazione di "casting" con un DTO (Data Transfer Object): FactDTO. L'interpretazione di questa operazione verrà fornita nel Paragrafo EsiHistorian.Dto. In conclusione, la rotta chiamerà il metodo *GetGroupedFactsLogic* di *QueryBL()* (spiegato nel Paragrafo EsiHistorian.BusinessLayer) passando i seguenti parametri:

- _db: contiene l'oggetto *QueryFactory*;
- request_dto: DTO della richiesta.

Il risultato del metodo *GetGroupedFactsLogic* sarà l'oggetto *Ok(result)*, il quale annetterà al risultato il codice di stato HTTP 200 ad indicare il successo della richiesta.

EsiHistorian.Dto Il progetto EsiHistorian.Dto ha lo scopo di modificare la forma dei dati durante il trasferimento, tale forma viene definita attraverso un Data Transfer Object. Le ragioni dietro questo modello sono molteplici:

- Rimuovere riferimenti circolari tra le dipendenze (in contrasto con la definizione di architettura a cipolla introdotta).
- Nascondere proprietà specifiche che i client non devono visualizzare.
- Omettere alcune proprietà per ridurre le dimensioni del payload.

L'introduzione del DTO nel progetto è stata necessaria per evitare un riferimento circolare tra i progetti WebApi e BusinessLayer. Difatti, WebApi aveva già una dipendenza da BusinessLayer (in *FactController* richiamiamo la classe *QueryBL*). È per tale motivo che l'oggetto *request* (del metodo *GetGroupedFacts* di *FactController*) è stato convertito in *FactDTO*. Di norma per implementare un DTO si definisce un metodo per la conversione dall'istanza del modello all'istanza del DTO. Un modo alternativo è utilizzare operatori impliciti ed espliciti. Il fine è quello di operare una conversione da un oggetto di *FactModel* ad un oggetto di *FactDTO*. Una volta creata la classe *FactDTO* (identica a *FactModel*) all'interno del progetto EsiHistorian.Dto, è stato definito in *FactModel* l'operatore esplicito di conversione.

```
public static explicit operator FactDTO(FactModel contract) {
    return new FactDTO
    {
        TotalDate = contract.TotalDate,
        Year = contract.Year,
        Month = contract.Month,
        Week = contract.Week,
        Day = contract.Day,
        Hour = contract.Hour,
        Minutes = contract.Minutes,
        Seconds = contract.Seconds,

        Dimensions = contract.Dimensions,

        FactName = contract.FactName

    };
}
```

EsiHistorian.BusinessLayer Nel progetto EsiHistorian.BusinessLayer è stata immagazzinata la logica per costruire la query SQL sulla base dell'oggetto JSON. Tutta l'operazione viene svolta dal metodo *GetGroupedFactsLogic,* che come accennato precedentemente, riceve l'oggetto *qf* (QueryFactory) e *dto* (FactDTO). Il processo si instaura con la dichiarazione e inizializzazione delle seguenti variabili: *tablename* e *query*.

```
var tableName = dto.FactName != null ? "fact_" + dto.FactName : "fact";

var query = qf.Query().From(tableName);
```

L'operatore ternario che valorizza *tablename*, assegna "fact" nell'eventualità che il valore associato alla chiave "tablename" sia nullo. Altrimenti, concatena il nome della tabella indicato alla stringa "fact_". La variabile *query* rappresenta un'istanza di *Query()* sulla tabella dei fatti. L'intera logica per la costruzione della query SQL è troppo articolata per essere

illustrata minuziosamente. In sintesi, la procedura analizza i valori delle proprietà di *dto* e su tale base costruisce pezzo per pezzo l'interrogazione. Un esempio è quello mostrato di seguito, in cui prima si verifica la volontà di raggruppare per giorno e successivamente si compongono le clausole di "Join", "GroupBy", "Select", "OrderBy" e infine "Where" per filtrare i dati all'interno dei giorni richiesti.

```
else if (dto.Day.GroupBy)
    {
        query
            .Join("dim_date", tableName + ".date_id", "dim_date.date_id")
            .GroupBy("Year", "Month", "Day")
            .Select("Year", "Month", "Day")
            .OrderBy("Year", "Month", "Day");

        query.Where("dim_date.date_id", ">=", $"{dto.Day.FilterFrom.Year}-{
    dto.Day.FilterFrom.Month}-{dto.Day.FilterFrom.Day}");
        query.Where("dim_date.date_id", "<=", $"{dto.Day.FilterTo.Year}-{
    dto.Day.FilterTo.Month}-{dto.Day.FilterTo.Day}");

    }
```

Applicando tale ragionamento a tutte le granularità delle dimensioni con le dovute accortezze, l'oggetto *query* necessiterà infine di selezionare gli aggregati della misura: media, minimo e massimo.

```
query = query.SelectRaw("AVG(value) as Value, MIN(value) as ValueMin, MAX(
    value) as ValueMax");
```

Terminata quest'istruzione, l'interrogazione è pronta per essere eseguita attraverso il metodo *Get()*. Il risultato sarà di tipo IEnumerable<dynamic> (l'interfaccia di base per tutte le raccolte generiche che possono essere enumerate).

```
var result = query.Get();

return result;
```

2.2.4 Esecuzione

L'applicazione una volta avviata stampa in console le informazioni seguenti.

```
Hosting environment: Development
Content root path: C:\Users\mattia.scuriatti\EsiHistorian\EsiHistorian.
    WebApi\bin\x86\Debug\net6
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
```

A questo punto, tramite un web browser è possibile aprire Swagger all'indirizzo `http:localhost:5000/swagger`. Facendo click sul bottone "Try it out" verrà abilitata l'API: "/api/Fact/GetGroupedFacts". Espandendo la

scheda, sarà possibile introdurre il "request body" in formato JSON (riportato un esempio) e avviare la richiesta attraverso il bottone "Execute".

```
{
  "totalDate": {
    "groupBy": true,
    "filterFrom": "2023-01-01T23:00:00.429Z",
    "filterTo": "2023-04-30T00:00:00.429Z"
  },
  "year": {
    "groupBy": true,
    "filterFrom": "2023-01-01T23:00:00.429Z",
    "filterTo": "2023-04-30T00:00:00.429Z"
  },
  "month": {
    "groupBy": true,
    "filterFrom": "2023-01-01T23:00:00.429Z",
    "filterTo": "2023-04-30T00:00:00.429Z"
  },
  "week": {
    "groupBy": false,
    "filterFrom": "2023-01-01T23:00:00.429Z",
    "filterTo": "2023-04-30T00:00:00.429Z"
  },
  "day": {
    "groupBy": true,
    "filterFrom": "2023-01-01T23:00:00.429Z",
    "filterTo": "2023-04-30T00:00:00.429Z"
  },
  "hour": {
    "groupBy": true,
    "filterFrom": "2023-01-01T23:00:00.429Z",
    "filterTo": "2023-04-30T00:00:00.429Z"
  },
  "minutes": {
    "groupBy": false,
    "filterFrom": "2023-01-01T23:00:00.429Z",
    "filterTo": "2023-04-30T00:00:00.429Z"
  },
  "seconds": {
    "groupBy": false,
    "filterFrom": "2023-01-01T23:00:00.429Z",
    "filterTo": "2023-04-30T00:00:00.429Z"
  },
  "dimensions": [
    {
      "groupBy": true,
      "filter": "1",
      "dimensionName": "line",
      "dimensionFields": [
        "name"
      ]
```

```
    },
       {
      "groupBy": true,
      "filter": "PRESS_HM",
      "dimensionName": "Parameter",
      "dimensionFields": [
        "name"
      ]
    }
  ],
  "factName": null
}
```

Una volta terminata, la risposta verrà visualizzata sotto la voce "Server Response". In essa vengono riportati:

- "Code": code di stato HTML;
- "Details": dettagli della risposta, divisi in "Response body" e "Response headers".

Vengono riportati i "Details" della chiamata antecedente, rispettivamente "Response body" (in parte) e "Response headers".

```
[
  {
    "Year": 2023,
    "Month": 1,
    "Day": 1,
    "Hour": 0,
    "line_name": "TUB_LINE_1",
    "Parameter_name": "Pressure HotMelt Glue",
    "Value": 99.570555,
    "ValueMin": 0,
    "ValueMax": 199
  },
  {
    "Year": 2023,
    "Month": 1,
    "Day": 2,
    "Hour": 1,
    "line_name": "TUB_LINE_1",
    "Parameter_name": "Pressure HotMelt Glue",
    "Value": 98.923333,
    "ValueMin": 0,
    "ValueMax": 199
  },
  ...
]
```

```
content-length: 416144
content-type: application/json; charset=utf-8
```

```
date: Tue, 29 Aug 2023 06:43:32 GMT
server: Kestrel
```

Capitolo 3

Tecnologie utilizzate

3.1 Apache Spark

Apache Spark (Spark) è un motore di elaborazione dati open source per grandi dataset. È stato progettato per offrire velocità di elaborazione, scalabilità e programmabilità per i Big Data; specificamente per dati in streaming, dati grafici e machine learning. Spark elabora i dati da 10 a 100 volte più velocemente rispetto alle alternative. Applica la scalabilità distribuendo il lavoro di elaborazione su grandi cluster di computer, con parallelismo integrato e tolleranza agli errori. Inoltre, include le API per i linguaggi di programmazione: Scala (linguaggio con cui è stato scritto), Java, Python e R. Spark viene spesso paragonato ad Apache Hadoop, o a MapReduce, il paradigma di programmazione nativo di Hadoop. La principale differenza sta nel fatto che Spark elabora e mantiene i dati in memoria, senza scrittura o lettura da disco, il che si traduce in velocità di elaborazione notevolmente più elevate. L'architettura dell'ecosistema Spark è rappresentata in Figura 3.1.

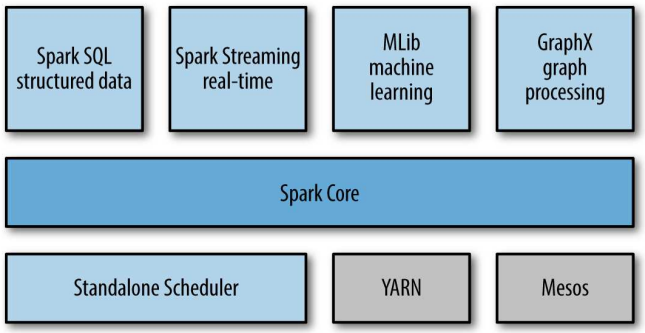

Figura 3.1: Spark Stack

3.1.1 Cluster Manager

Il livello più basso dello stack prende il nome di Cluster Manager. Le applicazioni Spark vengono eseguite come insiemi indipendenti di processi su un cluster, coordinati dall'oggetto SparkContext (o SparkSession per versioni maggiori della 2.0) nel programma principale (chiamato programma driver). Nello specifico, per essere eseguito su un cluster, SparkContext può connettersi a diversi tipi di gestori di cluster, che allocano le risorse tra le applicazioni. I cluster manager attualmente supportati sono:

- Standalone: configura il cluster attraverso degli script (es: si avvia il nodo master attraverso `./sbin/start-master.sh` e si connettono i worker attraverso `./sbin/start-worker.sh <master-spark-URL>`);
- Apache Mesos: può anche eseguire Hadoop MapReduce e applicazioni di servizio (deprecato);
- Hadoop YARN: il gestore delle risorse in Hadoop 2 e 3;
- Kubernetes: un sistema open source per automatizzare la distribuzione, il dimensionamento e la gestione delle applicazioni "containerizzate";

3.1.2 Spark Core

Lo Spark Core fornisce una semplice interfaccia di programmazione per l'elaborazione di grandi dataset: RDD API. È implementato in Scala, ma fornisce API per Java, Python e R, riguardo le seguenti operazioni: "trasformation" e "action". Infine, implementa i principi di "in-memory computing", "fault-recovery" e "job scheduling".

Strutture dati La struttura dati principe in Spark è RDD (Resilient Distributed Dataset): raccolte di elementi con tolleranza agli errori che possono essere distribuiti tra più nodi in un cluster e su cui è possibile lavorare in parallelo. Ciascun dataset in un RDD è suddiviso in partizioni logiche, che possono essere elaborate su nodi differenti del cluster. Oltre agli RDD, Spark gestisce anche un altro tipo di dato: il Dataframe, che rappresenta concettualmente una tabella di dati con righe e colonne. Nel tempo hanno riscosso un ruolo centrale all'interno di Spark poichè forniscono uniformità tra i diversi linguaggi, come Scala, Java, Python ed R.

Livelli di memorizzazione Le strutture dati possono essere immagazzinate secondo diversi "storage level":

- *MEMORY_ONLY*: memorizza RDD come oggetti Java deserializzati nella JVM. Se l'RDD non entra in memoria, alcune partizioni non verranno memorizzate nella cache e verranno ricalcolate al volo ogni volta che saranno necessarie. Questo è il livello predefinito;

- *MEMORY_AND_DISK*: memorizza RDD come oggetti Java deserializzati nella JVM. Se l'RDD non entra in memoria, archivia le partizioni restanti nel disco;

- *MEMORY_ONLY_SER*: memorizza RDD come oggetti Java serializzati (un array di byte per partizione). Questo è generalmente più efficiente in termini di spazio rispetto agli oggetti deserializzati, soprattutto quando si utilizza un serializzatore veloce, ma richiede una lettura più intensiva della CPU;

- *MEMORY_AND_DISK_SER*: simile a MEMORY_ONLY_SER, ma distribuisce le partizioni che non rientrano nella memoria sul disco invece di ricalcolarle al volo ogni volta che sono necessarie;

- *DISK_ONLY*: memorizza le partizioni RDD solo su disco;

- *MEMORY_ONLY_2, MEMORY_AND_DISK_2*: come i livelli precedenti, ma replica ogni partizione su due nodi del cluster.

Questi livelli possono essere indicati esplicitamente tramite il metodo *persist()* degli RDD. D'altro canto il metodo *cache()* indica il default storage level: *MEMORY_ONLY*.

DAG Rispetto al processo di esecuzione, Spark crea un DAG (Directed Acyclic Graph) per pianificare le attività e l'orchestrazione dei nodi di lavoro sul cluster. Questo grafo dunque descrive logicamente il piano di esecuzione; i nodi rappresentano gli RDD mentre gli archi sono le operazioni applicate ad essi. Ogni nodo contiene il puntatore al nodo padre (possono esserci più nodi padre, ad esempio: JOIN). Nel DAG non sono contenuti i valori, ma solo puntatori e operazioni. Questa traccia delle attività rende possibile la tolleranza agli errori, in quanto dà la possibilità di riapplicare le operazioni registrate da uno stato precedente (invece che dall'inizio). Prende il nome di RDD Lineage (o RDD dependency graph) il DAG di tutti gli RDD padre di un dato RDD (Figura 3.2).

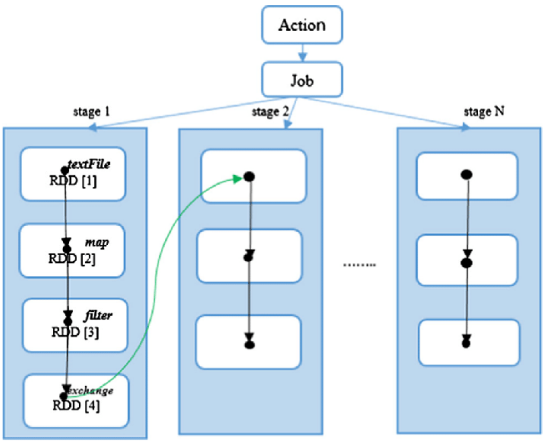

Figura 3.2: Esempio di un DAG

Transformation e Action Come accennato le operazioni sugli RDD sono di due tipologie. Le "transformations" effettuano delle manipolazione in un RDD, producendone uno nuovo. Alcuni esempi di "Transformations" sono:

- *Map* e *Filter*: generano un nuovo RDD operando su un solo RDD padre (*narrow dependencies*);

- *Join*, *GroupByKey* e *ReduceByKey*: generano un nuovo RDD operando su più RDD padri (*wide dependencies*).

In tutti questi casi parliamo di operatori "Lazy", ossia che definiscono un nuovo RDD ma che verrà costruito solo nel momento in cui verrrà elaborata una "action". Una "action", lancia una esecuzione su un RDD e ritorna dei valori (ad esempio: *Count*, *First*, *Take*, *Reduce* e *Collect*).

Esecuzione L'esecuzione di processi indipendenti e paralleli, all'interno di un cluster, viene coordinata dall'oggetto *SparkContext* creato nel Driver Program (il processo che esegue la funzione *Main()* dell'applicazione). Lo *SparkContext* si connette al Cluster Manager per allocare le risorse. In particolare, attiva degli executor sui diversi nodi worker: processi che eseguono i task, memorizzano dati e gestiscono la propria cache. Una volta attivati gli executor, il Driver invia loro prima l'applicazione (esempio: in un JAR) poi i task da eseguire. L'architettura in questione è mostrata in Figura 3.3.

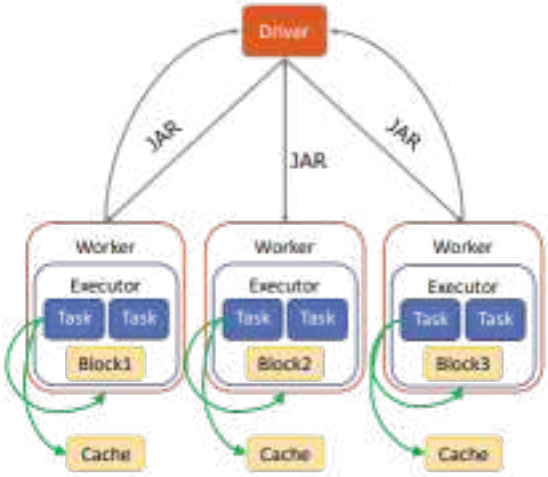

Figura 3.3: Architettura cluster di Spark

L'architettura di Spark è "shared-nothing" ossia non offre una memoria globale condivisa tra driver e executor. Supporta però 2 tipi di variabili condivise:

- Variabile di broadcast: di tipo read-only e copiata su ogni worker. È utile quando task distribuiti devono accedere agli stessi dataset di input in quanto evita l'invio di grandi dataset ai worker insieme al codice dei task;
- Accumulatore: può essere modificata dai worker e letta dal driver. Vengono usate per implementare *Count* e *Sum*.

3.1.3 Spark SQL

Spark SQL è il modulo Spark che consente di eseguire query su dati strutturati all'interno dei programmi Spark, utilizzando la sintassi SQL o tramite un'API familiare. Questo modulo si basa sul Dataframe (visto come una tabella del modello relazionale) e sul Dataset. Il primo può essere creato da un RDD esistente o da una sorgente, il secondo invece è un'estensione dei DataFrame basata su oggetti JVM fortemente tipizzati. Viene riportato un esempio della creazione di un Dataframe e dell'interrogazione attraverso le API di Spark SQL.

```
//crea il dataframe a partire da un file json e ne mostra il contenuto
val df = spark.read.json("dataset/people.json")
df.show()

//query tramite codice
df.filter($"age" > 21).show()
df.groupBy("age").count().show()
```

```
//query in SQL
df.createOrReplaceTempView("people") //definisce una vista
val sqlDF = spark.sql("SELECT * FROM people where age > 21")
sqlDF.show()
```

Tabelle e Viste Spark La web application costruisce le "raw query SQL" per interrogare le tabelle in SQL Server. La medesima query, con alcune modifiche sintattiche (illustrate nel Capitolo 4) verrà utilizzata da Spark. Di conseguenza, all'interno di Spark dovranno essere definite delle tabelle/viste associate alle tabelle in SQL Server. Le tabelle/viste in Spark, vengono costruite "sopra" i Dataframe. Come qualsiasi tabella in un modello relazionale, la tabella Spark è una raccolta di righe e colonne archiviate come file di dati (es: file Parquet o Csv). Le viste Spark invece, sono delle tabelle "virtuali" senza dati fisici disponibili. Le strutture in questione vengono categorizzate nel seguente modo:

- Tabelle: *Managed* (o Internal) e *External*;
- Viste: *Temporary View*, *Global Temporary View* e *Global Permanent View*.

Tabelle Managed Una *Managed Table* è una tabella Spark che gestisce sia i dati che i metadati. I dati vengono in genere archiviati nella directory predefinita (*app-spark-directory/spark-warehouse*). I metadati vengono archiviati in un catalogo che verrà illustrato nel paragrafo seguente. Quando viene eliminata una tabella interna, vengono eliminati sia i dati che i metadati. Viene riportato un esempio di API Spark SQL per la creazione di una *Managed Table*.

```
// df: il Dataframe contente i dati, questo viene salvato come un tabella
    denominata "my_table"
df.write.saveAsTable("my_table")
```

Tabelle External Una *External Table* è una tabella della quale Spark gestisce i metadati (come nel caso *Managed*), mentre la locazione dei dati veri e propri è controllata dall'utente. Dunque, all'utente viene richiesto di specificare la posizione in cui archiviare la tabella o, in alternativa, la directory di origine da cui verranno estratti i dati per creare una tabella. Le tabelle esterne sono accessibili solo dai cluster che hanno accesso al sistema di archiviazione delle tabelle. L'eliminazione di una tabella esterna eliminerà solo i dettagli della tabella dal catalogo, ma i dati sottostanti rimarranno così come sono nella relativa directory. Viene riportato un esempio di API Spark SQL per la creazione di una *External Table*.

```
// df: Dataframe contente i dati, questo viene salvato come un tabella all'
    interno della directory indicata ("<your-storage-path>") denominata "
    my_table"
df.Write.Option('path', "<your-storage-path>").SaveAsTable("my_table")
```

Temporary View Le "Temporary View" sono viste Spark che hanno un ambito relativo alla sessione ("SessionSpark-scoped"), pertanto sono disponibili solo per la durata della sessione che le ha create. Queste viste Spark non sono accessibili da altre sessioni o cluster. Vengono definite come tabelle all'interno del catalogo (con opzione "isTemporary" = true). Viene riportato un esempio di API Spark SQL per la creazione di una "External Table".

```
// df: Dataframe contente i dati, questo vene salvato come vista temporanea
     denominata "my_view"
df.CreateOrReplaceTempView("my_view")
```

Global Temporary View Le "Temporary View" in Spark SQL hanno un ambito di sessione, possiamo cambiare tale ambito tramite le "Global Temporary View" che rendono le viste Spark visibili tra tutte le sessioni. La vista temporanea globale è collegata a un database *global_temp* mantenuto nel catalogo. Viene riportato un esempio di API Spark SQL per la creazione di una "Global Temporary View".

```
// df: Dataframe contente i dati, questo viene salvato come vista globale
    temporanea denominata "my_global_view"
dataframe.createOrReplaceGlobalTempView("my_global_view")
```

Global Permanent View Le "Global Permanent View" vengono create rendendo persistenti i dati. Possono essere create sia su tabelle interne che esterne, ma non sopra viste temporanee e Dataframe. Viene riportato un esempio di API Spark SQL per la creazione di una "Global Temporary View" [12].

```
// table: la tabella a partire dalla quale viene creata una vista globale
    permante di nome "permanent_view"
spark.sql("CREATE VIEW permanent_view AS SELECT * FROM table")
```

Catalogo Per interagire con le tabelle/viste, Spark si avvale di un catalogo, o metastore: un database relazionale che gestisce i metadati del database oggetto. Spark viene fornito con un catalogo predefinito in modalità non persistente (svuotato alla fine della sessione) che è un database Apache Derby. Questa configurazione è consigliata solo per unit test

e uso locale, poiché Apache Derby è in modalità utente singolo (non supporta più di una connessione alla volta). Durante una sessione, si può lavorare con un database definito dall'utente, altrimenti se non viene esplicitata questa necessità Spark utilizza il database preconfigurato denominato *default* (con la directory per immagazzinare i file delle tabelle in "app-spark-directory/spark-warehouse") [7].

```
In [11]: spark.Catalog.CurrentDatabase()
Out[11]: 'default'
In [12]: spark.Catalog.ListDatabases()
Out[12]: [Database(name='default', description='default database',
    locationUri='file:/app/spark-warehouse')]
In [14]: spark.Catalog.ListTables()
Out[14]: []
```

Oltre al database di default, è stato menzionato il database *global_temp*, il quale però non viene visualizzato nel catalogo.

3.1.4 Spark Streaming, GraphX e Mlib

Spark Streaming è un'estensione dell'API Spark per l'elaborazione di dati in streaming. Questi una volta ricevuti vengono divisi in lotti (batches), ognuno dei quali sarà elaborato dallo Spark Core come un qualsiasi RDD (Figura 3.4).

Figura 3.4: Flusso di lavoro della libreria Spark Streaming

GraphX è un componente in Spark per grafici e calcoli paralleli ai grafic. Esso è basato su RDG (resilient distributed graph): un'estensione di RDD per rappresentare grafi e che offre la possibilità di svolgere algoritmi specifici della teoria dei grafi (es: "shortest path"). Infine, Mlib è la libreria per il machine learning che contiene algoritmi di classificazione, regressione, e clustering. Contiene inoltre altre utilità, come la costruzione di pipeline ML, la valutazione dei modelli, l'algebra lineare distribuita e le statistiche.

3.2 Apache Parquet

Le tabelle che verranno elaborate da Spark saranno immagazzinate su disco tramite dei file Apache Parquet. Apache Parquet è un formato di archiviazione con le seguenti caratteristiche:

- column oriented: a differenza dei formati basati su righe come CSV, Apache Parquet è orientato a colonne, ovvero i valori di ciascuna colonna della tabella vengono archiviati in maniera sequenziale;
- open-source;
- self-describing: oltre ai dati, un file Parquet contiene dei metadati includendo lo schema e la struttura. Ogni file memorizza sia i dati che gli standard utilizzati per accedere a ciascun record, semplificando il disaccoppiamento dei servizi che scrivono, archiviano e leggono i file Parquet.

3.2.1 Struttura di un file Parquet

Un file Parquet in generale è diviso in 2 parti: la prima contenente i dati e la seconda contenente i metadati. Nella prima i dati sono organizzati in "gruppi di righe". Questi gruppi di righe a loro volta sono costituiti da uno o più blocchi di colonne. I dati per ogni blocco di colonna vengono scritti sotto forma di pagine. Ogni pagina dunque contiene valori solo per una particolare colonna. L'altra parte del file Parquet prende il nome di footer. In esso abbiamo 3 tipologie di metadati associati al file, alle colonne e alle pagine. Tutte queste informazioni permettono l'ottimizzazione per il recupero dei dati durante il query processing. La struttura viene rappresentata in Figura 3.5.

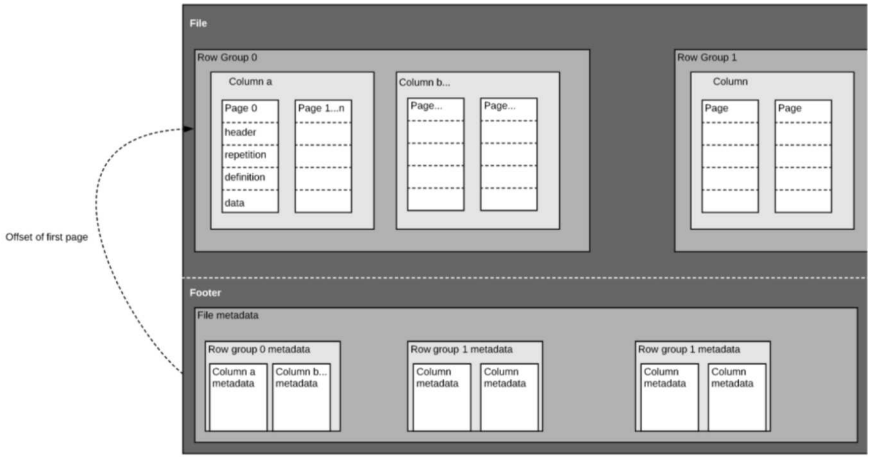

Figura 3.5: Struttura di un file Parquet

3.2.2 Vantaggi

In generale, i vantaggi che si ottengono dal formato Parquet sono: compressione e performance. In Parquet, la compressione viene eseguita colonna per colonna ed è progettata per supportare opzioni di compressione flessibili, e schemi di codifica estensibili per tipo di dati: ad esempio, è possibile utilizzare codifiche diverse per comprimere dati interi e stringhe. Inoltre, la proprietà di essere "column oriented" permette l'ottimizzazione nell'esecuzione delle query di analisi, in quanto permettere alla query di concentrarsi solo sulle colonne di interesse. Pertanto, la quantità di dati scansionati sarà molto inferiore e comporterà un minore utilizzo di operazioni di I/O. La scrittura di contro sarà più svantaggiosa, ma trattandosi di un applicativo OLAP questo non andrà ad incidere.

3.3 Delta Lake

Delta Lake è un "source storage layer" che fornisce affidabilità nella gestione dei dati garantendo le proprietà transazionali. Questo strumento offre la possibilità di integrarsi perfettamente con Spark. Delta Lake nello specifico è un "wrapper" del formato Parquet, il quale aggiunge le seguenti funzionalità

- "transaction log": ogni operazione di modifica sui dati viene inserita in questo log per tenere una registrazione cronologica delle attività eseguite sul sistema;
- proprietà ACID: garantite a livello di tabella;
- gestione scalabile dei metadati: sfrutta la potenza di elaborazione distribuita di Spark per gestire facilmente tutti i metadati;
- "schema enforcement": una volta acquisito lo schema dei dati effettua dei controlli per garantire che i nuovi dati rispettino tale struttura;
- "time travel": sfruttando il transaciton log è possibile "viaggiare" tra le varie versione dei nostri dati.

3.3.1 Transaction Log

Il log delle transazioni di Delta Lake (noto anche come DeltaLog) è un registro ordinato contenente tutte le operazioni di modifica eseguite su una tabella Delta Lake. Queste operazioni vengono registrate come unità atomiche note come commit, scomposte a loro volta in azioni (es: aggiungi file, rimuovi file o aggiorna i metadati ecc.) [1]. Di seguito viene riportato un esempio di commit per l'operazione di *WRITE* in modalità *APPEND*.

```
{"commitInfo":{"timestamp":1689932801208,"operation":"WRITE","
    operationParameters":{"mode":"Append","partitionBy":"[]"},"readVersion"
    :1,"isBlindAppend":true,"operationMetrics":{"numFiles":"1","
    numOutputBytes":"1944","numOutputRows":"1"}}}
{"add":{"path":"part-00000-0383b12c-c91e-4bb7-8543-cdf590214aac-c000.snappy
    .parquet","partitionValues":{},"size":1944,"modificationTime"
    :1689932801152,"dataChange":true}}
```

Transaction Log nel file system Quando un utente crea una tabella Delta Lake, il log delle transazioni associato a tale tabella viene creato nella sotto-directory *_delta_log*. Man mano che vengono apportate modifiche alla tabella, queste vengono registrate come commit atomici ordinati nel registro delle transazioni. Ogni commit viene scritto come file JSON, iniziando con *000000.json* e così via per ulteriori modifiche.

```
my_table/
└── _delta_log/
    ├── 000000.json
    └── 000001.json
```

Checkpoint file Dopo aver registrato diversi commit, Delta Lake salva un file di *checkpoint* nella stessa sotto-directory *_delta_log*. Il checkpoint ha lo scopo di per mantenere buone prestazioni di lettura. Difatti, questi file salvano l'intero stato della tabella in un determinato momento, in formato Parquet nativo, facile e veloce da leggere per Spark. In altre parole, offrono agli utenti una sorta di "scorciatoia" per riprodurre completamente lo stato di una tabella evitando di rielaborare quelli che potrebbero essere migliaia di file JSON minuscoli e inefficienti.

```
my_table/
└── _delta_log/
    ├── 000000.json
    ├── 000001.json
    ├── .
    ├── .
    ├── .
    ├── .
    ├── 000010.json
    └── 000010.checkpoint.parquet
```

3.3.2 Proprietà ACID

Atomicità Il log è il meccanismo attraverso il quale Delta Lake è in grado di offrire la garanzia di atomicità (per le operazioni di INSERT e UPDATE). Senza questa proprietà, un guasto hardware o un bug del software potrebbero provocare la scrittura solo parziale dei dati in una tabella. Nel caso in cui una transazione fallisca dopo una scrittura su una tabella, Delta Lake impedisce ai file scritti di danneggiare lo stato della tabella tramite l'operazione *VACUUM*. Questa, oltre ad eliminare tutti i file di dati non tracciati nel log di tabella, eliminerà i file rimanenti delle transazioni non riuscite.

Consistenza La consistenza garantisce che una transazione porti il database da uno stato valido a un altro. Questa viene garantita dalla proprietà di "schema enforcement" citata precedentemente.

Isolamento L'isolamento si riferisce alla capacità di evitare anomalie in un contesto di transazioni concorrenti. Delta Lake utilizza un algoritmo

denomitato "Optimistic Concurrency Control" per capire se due scritture simultanee sono compatibili o meno. Due scritture simultanee potrebbero essere:

- Compatibili: se non modificano la stessa porzione di dati (stessi file). Ad esempio, se aggiorneranno partizioni diverse. In questo caso entrambe le operazioni potrebbero essere eseguite senza alcun problema.

- Conflitto: se tentano di aggiornare gli stessi file. Delta può capire quali file sarebbero interessati da ciascun commit e la "Reading Version" di ciascuna operazione di scrittura (qual era la versione della tabella quando quell'operazione aveva letto quella tabella). In questo caso uno degli scrittori avrà successo e l'altro fallirà. Se si dispone di una politica di "retry", l'operazione non riuscita verrà eseguita nuovamente.

Delta Lake garantisce l'isolamento attraverso un concetto chiamato "snapshot isolation". Quando viene eseguita un'operazione di lettura o scrittura, Delta Lake acquisisce uno "snapshot" dei dati e applica tutte le operazioni su questo "snapshot". Delta Lake supporta due tipi di livelli di isolamento per gestire la concorrenza delle transazioni:

- *Serializable*: è il livello di isolamento più elevato, il che significa che qualsiasi transazione di scrittura su una tabella è completamente isolata dalle altre transazioni. Se due lavori tentano di scrivere contemporaneamente sulla stessa tabella, un lavoro dovrà attendere il completamento dell'altro, evitando così eventuali conflitti. La sequenza seriale è esattamente la stessa vista nello storico della tabella.

- *WriteSerializable*: è più debole rispetto a *Serializable* ed è il livello di isolamento predefinito in Delta Lake in quanto fornisce un ottimo "trade-off" tra consistenza e disponibilità dei dati. Garantisce solo che le operazioni di scrittura (non le letture) siano serializzabili. In questa modalità il contenuto della tabella Delta potrebbe essere diverso da quello previsto dalla sequenza di operazioni viste nello storico della tabella.

Persistenza In Delta Lake, tutte le operazioni vengono considerate completate nel momento in cui avviene la registrazione nel transaction log e l'eventuale archiviazione dei dati in formato Parquet. Tutto questo viene memorizzato all'interno del disco, ciò è quanto basta per garantire la persistenza.

3.3.3 Time travel

Come anticipato, Delta Lake conserva una cronologia delle modifiche apportate nel tempo, archiviando diverse versioni dei dati. Eseguire il rollback della tabella Delta Lake a una versione precedente, può essere un ottimo modo per invertire inserimenti di dati errati o annullare un'operazione che ha modificato la tabella in una maniera imprevista. La storia cronologica di una tabella Delta Lake è visualizzabile tramite il comando SQL seguente.

```
DESCRIBE HISTORY delta.‘/data/table/‘
```

Ogni versione riportata nella "history" avrà questo schema:

- *version*: numero della versione generata dall'operazione;
- *timestamp*: momento del commit della versione;
- *userId*: ID dell'utente che ha eseguito l'operazione;
- *userName*: nome dell'utente che ha eseguito l'operazione;
- *operation*: nome dell'operazione;
- *operationParameters*: parametri dell'operazione;
- *job*: dettagli del job che ha eseguito l'operazione;
- *notebook*: dettagli del notebook da cui è stata eseguita l'operazione;
- *clusterId*: ID del cluster su cui è stata eseguita l'operazione;
- *readVersion*: versione della tabella letta per eseguire l'operazione di scrittura;
- *isolationLevel*: livello di isolamento utilizzato per questa operazione;
- *isBlindAppend*: se questa operazione ha aggiunto dati;
- *operationMetrics*: metriche dell'operazione (ad esempio, numero di righe e file modificati);
- *userMetadata*: metadati di commit definiti dall'utente, se specificati.

Un esempio di "history" (compressa per ragioni di leggibilità) della la tabella "fact" viene riportato in Tabella 3.1.

version	timestamp	operation	op.Parameters	isBlindAppend
6	2023-08-08 ...	RESTORE	{version -> 4, ..	false
5	2023-08-07 ...	WRITE	{mode -> Append, ..	true
4	2023-07-24 ...	WRITE	{mode -> Overwite, ..	false
3	2023-07-21 ...	WRITE	{mode -> Append, ..	true
2	2023-07-21 ...	WRITE	{mode -> Overwrite, ..	false
1	2023-07-21 ...	WRITE	{mode -> Overwrite, ..	false

Tabella 3.1: "History compressa" della tabella "fact"

È possibile ripristinare una tabella Delta Lake a qualsiasi versione precedente attraverso il comando *RestoreToVersion()*. Alternativamente. è possibile ripristinare la versione indicando il timestamp associato attraverso il comando *RestoreToTimestamp()* [8].

```
//ripristino tramite versione
deltaTable.RestoreToVersion(1)

//ripristino tramite timestamp
deltaTable.RestoreToTimestamp('2021-01-01 01:01:01')
```

In alternativa, è possibile leggere temporaneamente determinate versioni senza effettivamente procedere al ripristino come versione attuale.

```
//lettura temporanea della versione 1 della tabella 'table'
spark.Read().Format("delta").Option("versionAsOf", "1").load("/data/table")
    .show()
```

L'utilizzo del comando "restore" reimposta il contenuto della tabella su una versione precedente, ma non rimuove alcun dato. Sulla base della versione espressa (o del timestamp) utilizza il transaction log per intuire quali file devono essere letti ignorando quelli inutili. Per rimuovere fisicamente i file inutilizzati dalla versione attuale è necessario il comando *vacuum()*.

3.3.4 Integrazione in Spark

Per integrare Delta Lake in Spark è richiesto preliminarmente di aggiungere le dipendenze del pacchetto (verrà mostrato nella Sezione 4.1.3). Supposto di aver completato questo passaggio, per usufruire delle funzionalità di Delta Lake bisogna effettuare il "wrap" delle nostre tabelle salvate in file Parquet. Viene riportato un esempio per comprendere come avviene questo "wrapping".

```
//jdbcDf: Dataframe contenente i dati della tabella
jdbcDf
    .Write()
    .Format("delta") //indica il nuovo formato delta
    .Save();
```

L'istruzione scriverà il Dataframe, contenente i dati della tabella, all'interno di un Parquet nel formato "delta". A livello di file system verrà aggiunta la cartella *_delta _log*.

Capitolo 4

Implementazione

4.1 Integrazione di Spark in .NET

L'ambiente .NET recentemente ha concesso l'integrazione di Apache Spark al suo interno, fornendo così API per l'utilizzo di Spark attraverso il linguaggio C#. Il progetto ".NET for Apache Spark" è stato collocato sopra il livello di interoperabilità di Spark che costruisce questo "bind" tramite le API dei linguaggi "basilari" di Spark quali Scala e Java [3] (Figura 4.1).

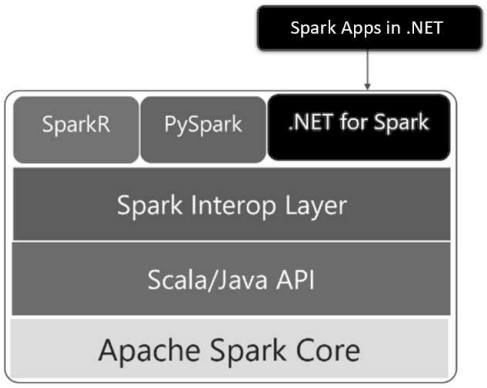

Figura 4.1: Interoperabilità dell'architettura di Spark

4.1.1 Preparazione dell'ambiente

Essendo che le applicazioni "NET for Apache Spark" sfruttano la JVM è essenziale avere installato Java nella macchina, oltre che .NET SDK ovviamente. Accertato ciò si può passare a Spark, nello specifico è stata installata l'ultima versione: 3.2.1; la questione della versione è molto importante perché .NET assicura la compatibilità solo con determinate versioni di

Spark. Una volta estratti i file scaricati è importante impostare le variabili di ambiente (globalmente) per "individuare" Apache Spark.

```
setx /M HADOOP_HOME C:\bin\spark-3.0.1-bin-hadoop2.7\
setx /M SPARK_HOME C:\bin\spark-3.0.1-bin-hadoop2.7\
setx /M PATH "%PATH%;%HADOOP_HOME%;%SPARK_HOME%bin" #Warning: Non eseguirlo
    nel caso di PATH con molti caratteri (altrimenti viene troncato a 1024
   caratteri)
```

Successivamente, viene prima scaricato ".NET for Apache Spark" (contenuto in un archivio denominato *Microsoft.Spark.Worker*) e poi, se ci si trova in ambiente Windows, anche *WinUtils*. Winutils è un file eseguibile che consente a Spark di eseguire alcuni comandi shell e "aiuta" Hadoop ad interagire con l'ambiente Windows. L'eseguibile *winutils.exe*, deve essere posto nella directory `%SPARK_HOME%bin`. *Microsoft.Spark.Worker* è un componente back-end che risiede sui singoli nodi di lavoro del tuo cluster e che permette a Spark di comprendere come avviare .NET CLR per eseguire le istruzioni. L'ultimo passo sarà impostare la variabile di ambiente *DOTNET_WORKER_DIR*, utilizzata per individuare i file binari di ".NET for Apache Spark" (valorizzandola con la directory in cui è stato estratto il *Microsoft.Spark.Worker*) [2].

```
setx /M DOTNET_WORKER_DIR <PATH-DOTNET-WORKER-DIR>
```

Per utilizzare le API di Spark in .NET viene richiesta anche l'istallazione del pacchetto *Microsoft.Spark* attraverso NuGet. Una volta scaricato, all'interno delle dipendenze del progetto verranno visualizzati dei file JAR (del tipo *microsoft-spark-<version>.jar*).

4.1.2 Connettere applicazioni ".NET for Apache Spark" a SQL Server

Per connettersi ad un'istanza SQL Server da un'applicazione ".NET for Apache Spark", è necessario preliminarmente concedere all'utente (che effettuerà la connessione) i permessi:

- *db_datareader*: per leggere tutti i dati dalle tabelle/viste dell'utente;
- *db_datawriter*: garantisce la possibilità di aggiungere, eliminare o modificare i dati delle tabelle dell'utente;
- *db_dataowner*: per eseguire tutte le attività possibili all'interno del database, tra cui anche il "drop" del database.

Tali permessi si possono concedere dal menù *Proprietà* degli utenti dell'istanza SQL Server, all'interno del database usato da Spark. Un altro aspetto da controllare è l'abilitazione del traffico TCP/IP verso l'istanza

SQL Server alla porta 1433 (possono essere configurati anche determinati IP da cui accettare la richiesta, in alternativa si può usare il parametro "Listen All") [5].

Connettersi dall'applicazione L'applicativo per connettersi richiede di aver scaricato il "Microsoft JDBC Driver" per SQL Server. Quest'ultimo fornisce diversi file JAR da usare in base alle versioni del JRE (Java Runtime Environment). Per questo progetto è stato scaricato:

- Microsoft JDBC Driver 12.2 per SQL Server: fornisce file di libreria di classi *mssql-jdbc-12.2.0.jre8.jar* e *mssql-jdbc-12.2.0.jre11.jar*.

I file JAR di Microsoft JDBC Driver non fanno parte di Java SDK e devono essere inclusi nel "classpath" (indica alla JVM la posizione delle classi/pacchetti) dell'applicazione dell'utente. Questa inclusione verrà illustrata nella Sezione 4.1.3. Scaricato il driver, all'interno dell'applicazione sono stati impostati i seguenti parametri di configurazione nel file *appsettings.json*:

```
"SQLServer": {
    "url": "jdbc:sqlserver://<SQL_server_IP_address>;databaseName=<
    SQL_server_Db_name>;encrypt=false;",
    "user": "<SQL_server_User_name>",
    "password": "<SQL_server_Password>",
    "driver": "com.microsoft.sqlserver.jdbc.SQLServerDriver"
  },
"SparkTables": [
    "fact",
    "dim_line",
    "dim_date",
    "dim_parameter",
    "dim_time"
  ]
```

Questi parametri verranno usati a runtime per stabilire la connessione con SQL Server ed estrarre le tabelle. Tralasciando il codice per il "retrieve" di queste informazioni (introdotto più avanti), la lettura di una tabella avverrà con questa modalità:

```
    string connection_url = "<URL to connect to SQL server instance>";
    string dbtable = "<Database table to access>";
    string user = "<Login user name>";
    string password = "<Login user password>";
    string driver = "<Driver>";

    DataFrame jdbcDF = spark.Read()
        .Format("jdbc")
        .Option("driver", driver)
        .Option("url", connection_url)
        .Option("dbtable", dbtable)
        .Option("user", user)
```

```
        .Option("password", password).Load(); //Lettura della tabella

    jdbcDF.Show(); // Visualizza il contenuto della tabella
```

4.1.3 Esecuzione di un'applicazione ".NET for Apache Spark"

Presupponendo di aver scritto la propria applicazione ".NET for Apache Spark", per avviare l'esecuzione è necessario lanciare da un promt dei comandi: *spark-submit* [6]. In ambiente Windows, si tratta di un "command file" contenuto nella cartella `%SPARK_HOME%bin` che si occupa di impostare il classpath con Spark, le sue dipendenze e può supportare diversi gestori di cluster nonché diverse modalità di distribuzione supportate da Spark. La sintassi è la seguente, dove le parentesi quadre individuano gli elementi opzionali.

```
spark-submit [options] <app jar> [app arguments]
```

Le *options* sono:

- --*master*: l' URL del nodo master del cluster (nodo che gestisce i worker e funge da cluster manager nella modalità standalone) ;
- --*deploy-mode*: indica se eseguire il "driver program" sui worker (modalità cluster) o localmente (modalità client) (impostazione predefinita: client);
- --*class*: classe che fungerà da entry point;
- --*jars*: indica i file jar da includere nel classpath del driver e degli executor. Non aggiungerà solo i jar ai classpath (del driver/executor), ma distribuirà anche gli archivi sul cluster. Se un particolare jar viene utilizzato solo dal driver, ciò comporta un sovraccarico non necessario;
- --*conf*: proprietà di configurazione Spark nel formato *chiave* = *valore*. Configurazioni multiple devono essere passate come argomenti separati. (ad esempio –conf <key>=<value> –conf <key2>=<value2>);
- --*driver-class-path*: può essere utilizzato per modificare il classpath solo per il driver. Ciò è utile per le librerie che non sono richieste dagli executor (risolve il problema di sovraccarico delle librerie inutili negli executor introdotta da --*jars*);
- --*driver-memory*: memoria allocata al driver (Default: 1024MB);

- --*packages*: indica le coordinate maven dei jar da includere sui classpath del driver e degli executor (le coordinate dovrebbero essere espresse nel seguente formato groupId:artifactId:version).

Infine, all'interno del comando vanno specificati:

- *app jar*: percorso del file jar che include l'applicazione e tutte le dipendenze. Nel caso di applicazioni ".NET for Apache Spark" questo viene valorizzato con i jar del pacchetto Microsoft.Spark (e.g. <path to your jar>/microsoft-spark-<version>.jar);

- *app arguments* (opzionale): argomenti passati al metodo main della classe principale, se presente.

Opzione *master* L'indirizzo URL del nodo master può essere in uno dei seguenti formati:

- *local*: esegue Spark localmente con un "worker thread" (ovvero senza parallelismo).

- *local[K]*: esegue Spark localmente con un numero K di "worker thread" (idealmente, impostalo sul numero di core della macchina).

- *local[K,F]*: esegue Spark localmente con un numero K di "worker thread" e di F "maxFailure" (variabile di configurazione che imposta il numero di task che possono fallire in continuo all'interno di un job).

- *local[*]*: esegue Spark localmente con con un numero di "worker thread" pari al numero di core logici della macchina.

- *local[*,F]*: esegue Spark localmente con un numero di "worker thread" pari al numero di core logici della macchina e F "maxFailures".

- *local-cluster[N,C,M]*: è una modalità solo per ambiente di test. Emula un cluster distribuito in una singola JVM con N worker, C core per worker e M MiB di memoria per worker.

- *spark://HOST:PORT*: si connette al master di un cluster standalone specificato. La porta deve essere quella su cui il master è configurato che è 7077 di default.

Per semplicità non sono state riportate tutte le configurazioni possibili essendo che l'unica differenza risiede nel tipo di cluster manager utilizzato (yarn, mesos ecc.).

Opzioni risorse per driver ed executor Durante il "submit" di un'applicazione è possibile anche specificare la quantità di memoria e di processori da allocare al driver (e anche agli "executors" in presenza di un cluster).

- --*driver-memory*: memoria allocata al driver;
- --*driver-cores*: numero di core della CPU allocati al driver;
- --*num-executors*: numero di executor da utilizzare;
- --*executor-memory*: quantità di memoria per ogni executor;
- --*executor-cores*: numero di core della CPU allocati ad ogni executor;
- --*total-executor-cores*: numero totale dei core degli executor da utilizzare.

Il comando completo per seguire l'applicativo verrà esposto nella Sezione 4.4.

Spark Shuffle Lo *shuffle* in Spark è un meccanismo di ridistribuire o ripartizionare dei dati in modo vengano raggruppati in modo diverso tra le partizioni. Tale meccanismo si innesca durante le trasformazioni come *groupByKey(), reducebyKey(), join()* e *groupBy()*; ed è un'operazione costosa poiché comporta:

- I/O del disco;
- coinvolge la serializzazione e la de-serializzazione dei dati;
- I/O di rete.

Il numero di partizioni che si vengono a generare può variare solo per le strutture dati di tipo *Dataframe*. Un *DataFrame* aumenta automaticamente il numero di partizioni a 200 quando avviene un'operazione di shuffling. Questo numero di partizioni può essere modificato attraverso l'opzione di configurazione: "spark.sql.shuffle.partitions" [4]. In base alle dimensioni del dataset, al numero di core e alla memoria, lo shuffling di Spark può avvantaggiare o danneggiare i job. Quando si possiede una quantità inferiore di dati, in genere è conveniente ridurre le partizioni altrimenti si verranno a creare molti file partizionati con un numero esiguo di record in ciascuna partizione; ciò si traduce nell'esecuzione di molti task con meno dati da elaborare. D'altro canto, quando si hanno troppi dati e un numero inferiore di partizioni si ottengono meno task ma più onerosi, talvolta si potrebbe anche riscontrare un errore di memoria insufficiente. Ottenere la giusta dimensione è un processo complicato e richiede molte esecuzioni con valori diversi per ottenere una soluzione "sub-ottima" (processo euristico).

4.1.4 Spark Web UI

Apache Spark fornisce una suite di interfacce utente Web (alla porta 4040 del *localhost*) per monitorare lo stato e il consumo di risorse del cluster Spark. Queste informazioni, vengono riportare in una serie di schede che ora verranno descritte.

Scheda Jobs La scheda *Jobs* visualizza una pagina di riepilogo di tutti i job dell'applicazione Spark e per ciascuno, una pagina dei dettagli. La pagina di riepilogo mostra informazioni di alto livello come:

- "User": utente Spark attuale;
- "Total Uptime": tempo trascorso dall'avvio dell'applicazione Spark;
- "Scheduling mode": modalità di "scheduling" dei job;
- Numero di job (raggruppati per stato): Attivi, Completati, Non riusciti;

Figura 4.2: Informazioni generali nella scheda Job

- "Event Timeline": Visualizza in ordine cronologico gli eventi relativi agli executor (aggiunti, rimossi) e ai job;

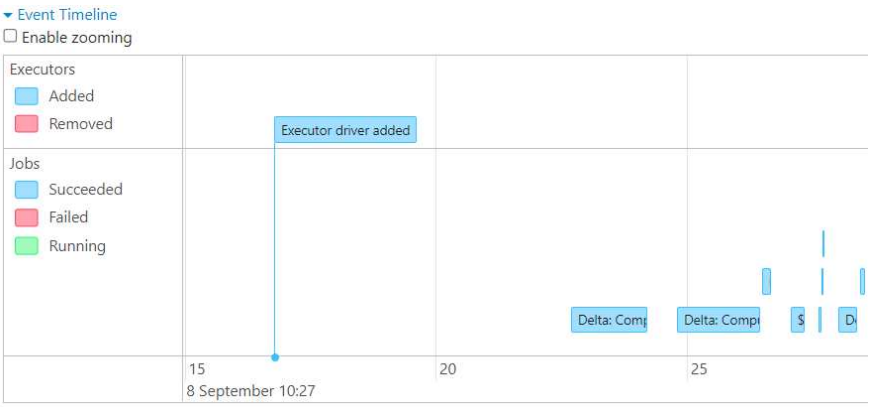

Figura 4.3: Event timeline nella scheda Job

- Dettagli dei job (raggruppati per stato): visualizza informazioni dettagliate sui job tra cui Job ID, descrizione (con un collegamento alla pagina dettagliata del job), orario di invio, durata, riepilogo delle fasi e avanzamento delle attività.

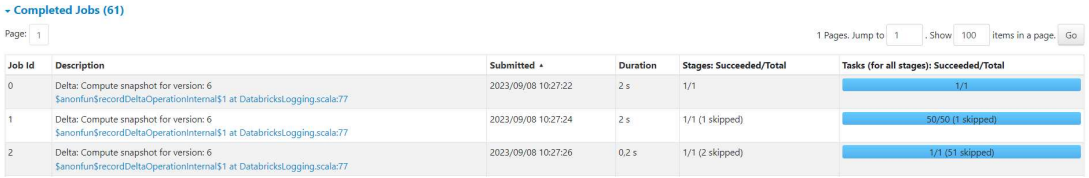

Figura 4.4: Lista dei job completati nella scheda Job

La pagina dei dettagli, accessibile facendo click su un job, mostra inoltre la sequenza temporale del job specifico, la visualizzazione del DAG e di tutte le fasi (stage). Una visione globale di tutti gli stage è riportata in un'ulteriore scheda: Scheda *Stages*.

Scheda Storage Visualizza gli RDD e i DataFrame (solo se materializzati in memoria) presenti nell'applicazione. Di questi vengono mostrati gli "storage level", le dimensioni e le partizioni (Figura 4.5). La pagina dei dettagli mostra anche le dimensioni e l'utilizzo negli executor per tutte le partizioni in un RDD/DataFrame.

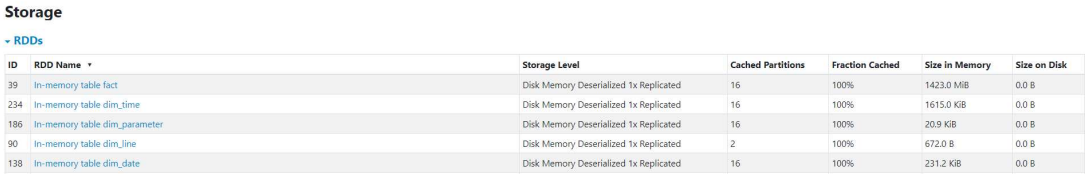

Figura 4.5: Lista degli RDD archiviati (tabelle del DW) nella scheda Storage

Scheda Environment La scheda *Environment* visualizza i valori per le diverse variabili di ambiente e configurazione, tra cui JVM, Spark e proprietà di sistema. La pagina è composta da cinque parti.

- "Runtime Information": contiene semplicemente le proprietà di runtime come le versioni di Java e Scala;
- "Spark Property": elenca le proprietà dell'applicazione (es: "spark.app.name" e "spark.driver.memory");
- "Hadoop Property": vengono visualizzate le proprietà relative a Hadoop e YARN.

- "System Property": mostra maggiori dettagli sulla JVM;
- "Classpath Entries" elenca le classi caricate da diverse fonti, il che è molto utile per risolvere i conflitti di classe.

Scheda Executors Visualizza prima delle informazioni di riepilogo sugli executors come: l'utilizzo della memoria/del disco e varie informazioni inerenti ai task. Da notare la colonna "Storage Memory" la quale mostra la quantità di memoria utilizzata su quella riservata esclusivamente alla memorizzazione nella cache. Inoltre vi sono delle informazioni prestazionali inerenti al "Garbage Collection time" e allo shuffling. (Figura 4.6).

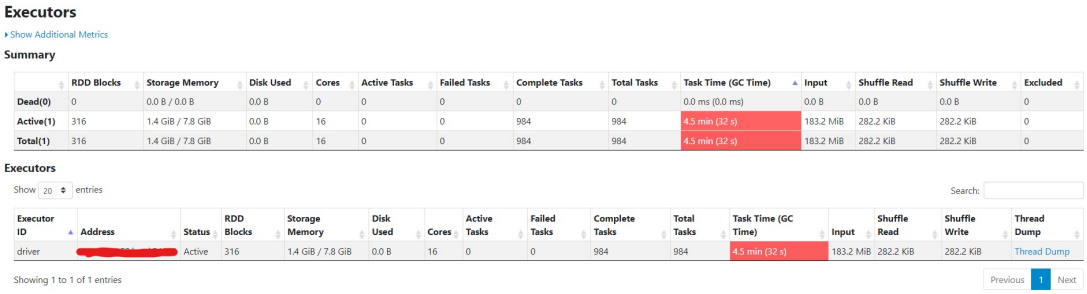

Figura 4.6: Visualizzazione della scheda Executors

Un piccolo appunto deve essere fatto riguardo alla memoria. Di fatti, la quantità totale espressa in "Storage Memory" non coincide con il quantitativo espresso in *–driver-memory* (all'interno dello *spark-submit*). Questo perché la memoria memoria del driver è divisa in quattro parti:

- Storage Memory: spazio riservato per i dati memorizzati nella cache;
- Execution Memory: spazio utilizzato dalle strutture dati durante le operazioni di shuffle;
- User Memory: per memorizzare le strutture dati create e gestite dal codice dell'utente;
- Reserved Memory: riservata da Spark per scopi interni.

La Reserved Memory riceve un quantitativo fisso pari a 300MB. La parte restante viene divisa in due: una per Storage/Execution Memory e l'altra per la User Memory. In genere il rapporto è 60% e 40%, ma può essere variato con il parametro di configurazione *spark.memory.fraction*.

Scheda SQL Se l'applicazione esegue interrogazioni attraverso Spark-SQL, la scheda SQL visualizza informazioni quali durata e job interessati.

Inoltre possono essere esaminati i dettagli sui piani di esecuzione delle singole query (tramite click).

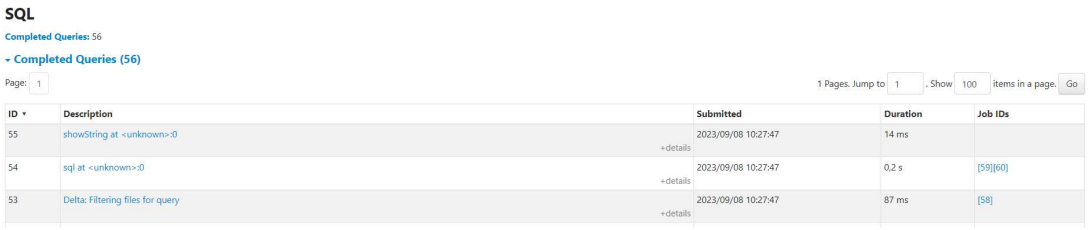

Figura 4.7: Visualizzazione della scheda SQL

4.2 Data ingestion

Il meccanismo di data ingestion serve per gestire l'inserimento dei nuovi record giornalieri all'interno della tabella dei fatti. Tale meccanismo non sarà necessario per le tabelle delle dimensioni, in quanto contengono pochi record e la variazione di questa cardinalità anche nel futuro rimarrà contenuta. La procedura è la seguente:

1. dalla tabella "fact", memorizzata nel formato Delta Table, si effettua un ordinamento dei record sui campi *date_id, time_id, line_id , parameter_id* e *value;*

2. sulla base di tale ordinamento viene aggiunta una colonna "fact_key" come un "numero" incrementale;

3. si individua l'ultima record come quello con il valore maggiore di "fact_key" (Tabella 4.1);

date_id	time_id	line_id	parameter_id	value	fact_key
2023-04-03	23:59:50	1	PRESS_HM	38	1
2023-04-03	23:59:53	1	PRESS_HM	56	2
2023-04-03	23:59:57	1	PRESS_HM	100	3

Tabella 4.1: Tabella "fact" salvata su disco da Spark con in giallo l'ultimo record

4. per immettere i nuovi dati verrà letta la tabella di SQL Server, filtrando i record temporalmente "recenti" diminuendo così i record da elaborare. Effettuando nuovamente lo stesso ordinamento e l'aggiunta della chiave, verranno estratti i record con "fact_key" maggiore della chiave dell'ultimo record individuato precedentemente (Tabella 4.2);

date_id	time_id	line_id	parameter_id	value	fact_key
2023-04-03	23:59:50	1	PRESS_HM	38	3
2023-04-04	00:00:01	1	PRESS_HM	56	4
2023-04-04	00:00:03	1	PRESS_BIT	100	5

Tabella 4.2: Tabella "fact" su SQL Server con in verde i nuovi record da estrarre

5. i record estratti verranno appesi in coda alla tabella Delta Lake (Tabella 4.3).

date_id	time_id	line_id	parameter_id	value	fact_key
2023-04-03	23:59:50	1	PRESS_HM	38	1
2023-04-03	23:59:53	1	PRESS_HM	56	2
2023-04-03	23:59:50	1	PRESS_HM	38	3
2023-04-04	00:00:01	1	PRESS_HM	5	4
2023-04-04	00:00:03	1	PRESS_BIT	23	5

Tabella 4.3: Tabella alla fine del meccanismo di "data ingestion"

4.3 Integrazione di Spark nell'applicativo

4.3.1 Classe SparkClass

Il primo passo di un applicazione ".NET for Apache Spark" è la creazione della sessione attraverso la classe *SparkSession* del pacchetto *Microsoft.Spark*. L'inizializzazione comporta la valorizzazione di due proprietà della classe:

- *Catalog*: interfaccia attraverso la quale gli utenti interagiscono con il catalogo per creare, eliminare, modificare o interrogare database e tabelle;
- *SparkContext*: punto di ingresso principale per la funzionalità Spark. Rappresenta la connessione a un cluster e può essere utilizzato per creare RDD, accumulatori e variabili di broadcast su quel cluster. Dovrebbe essere attivo solo uno *SparkContext* per JVM.

Nell'applicativo la sessione è stata assegnata al campo statico *Session* della classe *SparkClass* creata all'interno del *BusinessLayer*.

```
public static class SparkClass
{
    public static SparkSession Session = SparkSession.Builder()
        .AppName("MySparkApp")
        .Master("local[*]")
        .Config("spark.sql.shuffle.partitions", 16)
        .GetOrCreate();
}
```

Attraverso il metodo *Builder()* sono state impostate le seguenti opzioni:

- nome dell'applicazione: verrà visualizzato nella GUI di Spark;
- URL del master: impostato in locale e con abilitati tutti i core della macchina disponibili ("local[*]");
- opzione di configurazione: "spark.sql.shuffle.partitions" configura il numero di partizioni a 16 per il processo di shuffling (numero ricavato attraverso un'euristica basata sul numero di core).

La sessione verrà interrotta nel momento in cui verrà interrotta a sua volta l'applicazione. Come anticipato, il processo di arresto passa per il metodo *Stop()* della classe *EsiHistorianService*. Dunque, all'interno del metodo è stata aggiunta l'istruzione: *SparkClass.Session.Stop();*.

4.3.2 Classe EsiSparkService

Sempre all'interno del *BusinessLayer* è stata definita la classe *EsiSparkService* per svolgere il seguente "workflow":

- scrivere le tabelle in modo persistente nel disco (per evitare ad ogni avvio di caricare interamente le tabelle da SQL Server). Quest'operazione, viene effettuata solo una volta per l'intero contenuto mentre i nuovi record, che arrivano giornalmente al database SQL Server, verranno prelevati e "appesi" in coda;
- a partire dai dati immagazzinati su disco, creare delle "Temporary View" su cui verranno eseguite le interrogazioni;
- al fine di ottimizzare i tempi verranno trasferite in memoria centrale le viste.

Nella classe vengono definite preliminarmente delle funzioni ausiliare, queste verranno utilizzate all'interno del metodo principale *EsiSparkStartUp()* che implementerà il "workflow" appena descritto.

EsiFindLastRecord() È una funzione che ritorna un DataFrame contenente l'ultimo record della tabella dei fatti presente su disco (come parametri riceve nome e DataFrame associati alla tabella). L'idea è quella di seguire il meccanismo di "data ingestion" della Sezione 4.2; ossia ordinare il DataFrame secondo dei campi e creare una chiave basata su tale ordinamento. L'ultimo record trovato verrà poi ritornato al chiamante. Il processo viene riassunto nei passi seguenti.

1. definizione di 5 oggetti di tipo *Column* in cui sono mappate tutte le colonne (eccetto value_min e value_max) della tabella fact. Viene aggiunto un'ulteriore oggetto *Column* denominato *fact_key* che rappresenterà un identificatore incrementale;
2. a partire dal Dataframe della fact (tableDf), viene preliminarmente effettuato un filtraggio dei record (per alleggerire il carico da elaborare). Il Dataframe filtrato viene poi ordinato per le colonne estratte (in modo ascendente).
3. Per associare la fact_key ad ogni record devono prima essere riunificate le partizioni tramite la funzione *Repartition(1)*. Senza questo, la funzione *MonotonicallyIncreasingId()* (con cui è definita *fact_key*) avrebbe implementato un identificatore non coerente globalmente. Viene quindi aggiunta *fact_key* come nuova colonna;
4. riordinamento decrescente basato sulla *fact_key*. L'ultima riga sarà dunque accessibile attraverso il comando *tableDf.Limit(1)*.

```
//estrazione delle colonne per ordinare la fact
Column date_id = tableDf.Col("date_id");
Column time_id = tableDf.Col("time_id");
Column line_id = tableDf.Col("line_id");
Column parameter_id = tableDf.Col("parameter_id");
Column value = tableDf.Col("value");
Column fact_key = Functions.MonotonicallyIncreasingId();

//ordino per trovare ultimo record
tableDf = tableDf
    .Filter("date_id >= '2023-04-28'") //si filtra da un certo giorno per
    migliorare le prestazioni (tale giorno anche non hard coded)
    .Sort(date_id, time_id, line_id, parameter_id, value) //prima ordino in
     modo ascendente per creare una chiave ascendente
    .Repartition(1) //il ripartizionamento serve per la funzione
    MonotonicallyIncreasingId() (genera chiavi consecutive solo per singola
    partizione)
    .WithColumn("fact_key", fact_key) //inseriamo la chiave
    .Sort(fact_key.Desc()); //ordiniamo in modo discendente per tenere gli
    ultimi record in cima

Console.WriteLine($"Table {table_name} ordered");
//tableDf.Show();

DataFrame last_rowDf = tableDf.Limit(1);
Console.WriteLine($"Last row processed {table_name} (in Spark)");
last_rowDf.Show();

return last_rowDf;
```

EsiLoad() Funzione che riceve come parametri in ingresso:

- *string path*: percorso della directory contenente i file delle tabelle;
- *string table_name*: nome della tabella;
- *Dictionary<string,string> db_options*: dizionario con le opzioni per la connessione al database SQL Server;
- *SaveMode saveMode*: oggetto di tipo SaveMode (enum) che definisce la modalità di salvataggio;
- *DataFrame last_rowDf* [opzionale]: contiene eventualmente l'ultima riga della tabella salvata su disco.

Il metodo *EsiLoad()* inizialmente legge le tabelle da SQL Server e le carica all'interno di un Dataframe.

```
//recupera la sessione
SparkSession spark = SparkClass.Session;
```

```
//carica ogni tabella SQL dal db SQL server e lo salva in un dataframe
DataFrame jdbcDf = spark.Read()
        .Format("jdbc")
        .Options(db_options)
        .Option("dbtable", $"dbo.{table_name}")
        .Load();
```

Caricata la tabella all'interno del DataFrame *jdbcDf* si effettua un controllo. Nel caso in cui la tabella in questione sia "fact" o "dim_time" è necessario riportare la colonna "time_id" nel formato originale: "HH:mm:ss". Questo è stato necessario in quanto Spark automaticamente assegnava alla colonna il suo formato di default per timestamp (es: "1900-01-01 18:31:58" anzichè "18:31:58").

```
 if (table_name.Equals("fact") | table_name.Equals("dim_time"))
    {
      Column time_col = jdbcDf.Col("time_id");
      jdbcDf = jdbcDf.WithColumn("time_id", Functions.DateFormat(time_col,
    "HH:mm:ss"));
    }
```

Terminato questo passaggio viene memorizzata la tabella sul disco. Sono disponibili due procedure di memorizzazione che differiscono dal valore del parametro *savemode*. Questo perchè *EsiLoad()* dovrà immagazzinare sia le tabelle intere che i nuovi record giornalieri. Il parametro è un oggetto della classe SaveMode (di *Microsoft.Spark*) che nell'applicativo può assumere solo i seguenti valori:

- *SaveMode.Overwrite*: la modalità Overwrite indica che nel momento in cui si salva un DataFrame in un'origine dati, se il dato esiste già, è previsto che i dati esistenti vengano sovrascritti dal contenuto del DataFrame;
- *SaveMode.Append*: la modalità Append indica che nel momento in cui si salva un DataFrame in un'origine dati, se il dato esiste già, è previsto che il contenuto di DataFrame venga aggiunto in coda ai dati esistenti.

Nel caso della prima modalità verranno eseguite le istruzioni seguenti.

```
//saveMode = saveMode.OverWrite
jdbcDf
    .Repartition(16)
    .Write()
    .Format("delta")
    .Mode(saveMode)
    .Save(path + table_name);

//metodo per creare la vista temporanea
EsiSparkService.EsiCreateView(jdbcDf, table_name);
```

Nell'istruzione di scrittura riveste un ruolo centrale il metodo *Repartition(16)* il quale esplicitamente crea 16 partizioni di *jdbcDf*. Questo per ottenere quella caratteristica di parallelismo nell'elaborazione dei file. A differenza dell'opzione di configurazione "spark.sql.shuffle .partitions" (della *SparkClass*) qua non necessitiamo di un'operazione di trasformazione per partizionare il dato. Infine, dopo aver specificato il formato "delta" verrà salvata la tabella (Delta Table) nel percorso specificato attraverso una serie di file Parquet ognuno per ciascuna partizione (vengono riportati i file parquet delle partizioni della tabella "fact").

```
spark-warehouse\fact
├─part-00000-1af7f9e1-dec0..8faa065c7992-c000.snappy.parquet
├─part-00001-1af7f9e1-dec0..8faa065c7992-c000.snappy.parquet
├─part-00002-1af7f9e1-dec0..8faa065c7992-c000.snappy.parquet
├─part-00003-1af7f9e1-dec0..8faa065c7992-c000.snappy.parquet
├─part-00004-1af7f9e1-dec0..8faa065c7992-c000.snappy.parquet
├─part-00005-1af7f9e1-dec0..8faa065c7992-c000.snappy.parquet
├─part-00006-1af7f9e1-dec0..8faa065c7992-c000.snappy.parquet
├─part-00007-1af7f9e1-dec0..8faa065c7992-c000.snappy.parquet
├─part-00008-1af7f9e1-dec0..8faa065c7992-c000.snappy.parquet
├─part-00009-1af7f9e1-dec0..8faa065c7992-c000.snappy.parquet
├─part-00010-1af7f9e1-dec0..8faa065c7992-c000.snappy.parquet
├─part-00011-1af7f9e1-dec0..8faa065c7992-c000.snappy.parquet
├─part-00012-1af7f9e1-dec0..8faa065c7992-c000.snappy.parquet
├─part-00013-1af7f9e1-dec0..8faa065c7992-c000.snappy.parquet
├─part-00014-1af7f9e1-dec0..8faa065c7992-c000.snappy.parquet
└─part-00015-1af7f9e1-dec0..8faa065c7992-c000.snappy.parquet
```

L'altra modalità di salvataggio (*SaveMode.Append*) per appendere i nuovi record giornalieri effettua tale procedura:

1. si estrae il valore di chiave (*fact_key*) associato all'ultimo record salvato su disco;
2. si legge la nuova tabella da SQL Server e da questa si costruisce un DataFrame contenente tutti record con chiave maggiore della *fact_key* precedente;
3. se tale insieme non è vuoto si procede con la scrittura in modalità *Append*.

Il codice che concretizza tale meccanismo è quello riportato.

```
//estrazione valore di chiave dell'ultimo elemento
var key = last_rowDf.First().Get("fact_key");
```

```
//creazione dataframe con il delta
Delta = jdbcDf
    .Filter("date_id >= '2023-04-28'")
    .Sort(date_id, time_id, line_id, parameter_id, value) //essendo jdbcDf
    divisa in partizioni la lettura di spark non presenva l'ordinamento
    originale. Dunque viene riordinato in modo ascendente.
    .Repartition(1)
    .WithColumn("fact_key", fact_key) //inseriamo la chiave (fact_key
    crescente)
    .Filter($"fact_key > '{key}'") //teniamo tutti i record con chiave
    maggiore {key}
    .Drop("fact_key") //eliminiamo la colonna per tenere il formato
    originale ;

if (!Delta.IsEmpty())
{
    Delta
        .Write()
        .Format("delta")
        .Mode(saveMode) //saveMode = saveMode.Append
        .Save(path + table_name);
}
```

EsiCreateView() Funzione molto semplice che, a partire dal Dataframe (*tableDf*) e dal nome della tabella (*table_name*), crea la vista temporanea e ne visualizza le prime dieci righe in console.

```
Console.WriteLine($"Temporary view created: {table_name}");
tabledf.CreateOrReplaceTempView(table_name);
tabledf.Show(10);
```

EsiReadDeltaTable() Funzione adibita alla lettura delle tabelle sfruttando le funzioni del pacchetto Delta. Oltre a ritornare il Dataframe associato alla tabella, verrà visualizzata sulla console la storia della tabella (struttura descritta nella Sezione 3.3).

```
var deltaTable = DeltaTable.ForPath(SparkClass.Session, path + table_name);
DataFrame history = SparkClass.Session.Sql($"DESCRIBE HISTORY delta.'{path
    + table_name}'");
Console.WriteLine($"History of {table_name} table");
history.Show();
DataFrame deltaTableDf = deltaTable.ToDF();

return deltaTableDf;
```

EsiSparkStartUp() È la funzione principale che prende in ingresso:

- *string path*: stringa con il percorso della directory che contiene i file delle tabelle;

- *IEnumerable<string> tables_names*: collezione con i nomi delle tabelle;
- *Dictionary<string, bool> spark_options*: dizionario contenente le opzioni di esecuzione "Load" e "CheckDelta". Sono due dati di tipo *bool* che esprimono rispettivamente la volontà di caricare interamente le tabelle e di controllare eventualmente l'integrazione dei nuovi record giornalieri;
- *Dictionary<string, string> db_options*: dizionario con le opzioni per la connessione al database SQL Server.

La funzione inizia iterando la collezione *tables_names*. Pertanto per ogni tabella viene eseguito il codice riportato. Da notare come all'interno del ramo *else if (spark_options["CheckDelta"])* il meccanismo di "data ingestion" per i nuovi record viene realizzato solo per la tabella dei fatti (per le considerazioni fatte nella Sezione 3.3).

```
if (spark_options["Load"])
    {
        //Primo caricamento assoluto delle tabelle
        EsiSparkService.EsiLoad(path, table_name, db_options, SaveMode.
    Overwrite);
    }
else if (spark_options["CheckDelta"])
{
    if (table_name.Equals("fact"))
    {
        DataFrame factDf = EsiSparkService.EsiReadDeltaTable(path,
    table_name);

        //metodo che restituisce un dataframe con l'ultimo record che Spark
     ha scritto su disco
        DataFrame last_rowDf = EsiSparkService.EsiFindLastRecord(factDf,
    table_name);

        //carico il parquet del delta "appeso"
        EsiSparkService.EsiLoad(path, table_name, db_options, SaveMode.
    Append, last_rowDf);

        //lettura della nuova delta table
        DataFrame table_withDeltaDf = EsiSparkService.EsiReadDeltaTable(
    path, table_name);

        //creazione della vista
        EsiSparkService.EsiCreateView(table_withDeltaDf, table_name);
    }
    else
    {
        EsiSparkService.EsiLoad(path, table_name, db_options, SaveMode.
    Overwrite);
```

```
    }
}
else
{
    DataFrame tableDf = EsiSparkService.EsiReadDeltaTable(path, table_name)
    ;
    EsiSparkService.EsiCreateView(tableDf, table_name);
}
```

Ogni tabella verrò poi trasferita in memoria centrale.

```
SparkClass.Session.Sql($"CACHE TABLE {table_name}");
```

In aggiunta, è stata definita anche una vista temporanea (caricata anch'essa in memoria centrale) in cui sono stati realizzati preliminarmente i JOIN tra la "fact" e le dimensioni. Questo è stato fatto con il fine di evitare l'esecuzione dei JOIN durante l'esecuzione delle query. Le migliorie prestazionali verranno osservate nel Capitolo 5.

```
//Cache tabella con i JOIN
DataFrame factJoinDf = spark.Sql("SELECT fact.parameter_id, fact.line_id,
    fact.date_id, fact.time_id, value, value_min, value_max, date_key, year,
     month, week, day, time_key, hour, minute, second, line_key, line_name,
    parameter_key, parameter_name FROM fact INNER JOIN dim_date ON fact.
    date_id = dim_date.date_id INNER JOIN dim_time ON fact.time_id =
    dim_time.time_id INNER JOIN dim_line ON fact.line_id = dim_line.line_id
    INNER JOIN dim_Parameter ON fact.Parameter_id = dim_Parameter.
    Parameter_id");
factJoinDf.CreateOrReplaceTempView("fact_prova");
factJoinDf.Show(10);
spark.Sql($"CACHE TABLE fact_prova");
```

4.3.3 Classe FactController

Per implementare le interrogazioni attraversi Spark anzichè SQL Server è stata aggiunta una nuova rotta "GetGroupedFactsSpark" (POST) all'interno della classe *FactController*. Il controller associato è identico a quello della rotta originale. L'unica differenza risiede nella valorizzazione di una nuova variabile booleana *isSpark,* utile per eseguire l'interrogazione tramite Spark(*isSpark = true*) o SQL Server (*false*).

```
[HttpPost]
[Route("GetGroupedFactsSpark")]
public IActionResult GetGroupedFactsSpark([FromBody] FactModel request)
{
    if (!ModelState.IsValid)
        return BadRequest(ModelState);

    //nuova variabile per differenziare la logica di esecuzione
    IsSpark = true;
    var request_dto = (FactDTO)request;
```

```
    var businessQL = new QueryBL();

    var result = businessQL.GetGroupedFactsLogic(_db, request_dto, IsSpark)
    ;
    return Ok(result);
}
```

4.3.4 Classe QueryBL

Il procedimento per la costruzione della query è il medesimo che è stato descritto nella sezione 2.2.3. Ciò che è stato aggiunto è un blocco condizionale che decide di eseguire l'interrogazione sulle tabelle di SQL Server o sulle viste di Spark sulla base del valore della variabile *isSpark*. Nel secondo caso vengono eseguite queste istruzioni:

1. viene usato il *Compiler* fornito da SQLKata per trasformare un istanza di *Query* in una stringa SQL;

2. vengono risolti due problemi di sintassi che SparkSQL non interpreta correttamente: le parantesi quadre e i parametri introdotti da SQL Kata (es:*@p1*). I primi vengono semplicemente sostituiti con una stringa vuota mentre per i secondi sono stati sostituiti i valori associati (estraibili dal dizionario *NamedBindings* accessibile dalla query compilata);

3. viene eseguita la query tramite la funzione SparkClass.Session.Sql(query_spark). L'argomento sarà appunto la stringa ottenuta dal passo precedente;

4. viene riportato il risultato (DataFrame) in un formato interoperabile con la vecchia logica (una collezione).

L'elaborazione (per entrambe le logiche) viene cronometrata tramite uno *Stopwatch* per prendere i tempi di elaborazione. Il codice che implementa quanto detto finora viene riportato di sotto.

```
Stopwatch spark_query = Stopwatch.StartNew();
//avvio del cronometro
spark_query.Start();

//compilazione della query per ottenere la stringa SQL
var compiler = new SqlServerCompiler();
SqlResult compiled_query = compiler.Compile(query);
string query_sql = compiled_query.Sql;

//ciclo che ad ogni parametro sostituisce il valore
foreach (var p in compiled_query.NamedBindings) //NameBindings dizionario
    di coppie chiave,valore (@p1,value) della query
{
```

```
    if (p.Value is string){
        query_sql = query_sql.Replace(p.Key, $"'{p.Value}'");
    }
    else{
        query_sql = query_sql.Replace(p.Key, p.Value.ToString());
    }
}

//eliminazione delle parentesi quadre
string query_spark = query_sql.Replace("[", string.Empty).Replace("]",
    string.Empty);

//esecuzione della query
result_df = SparkClass.Session.Sql(query_spark);

//riportare il risultato come una lista
var result = result_df.Collect().ToList();

//stop del cronometro
spark_query.Stop();
Console.WriteLine($"Elapsed time query in Spark: {spark_query.
    ElapsedMilliseconds} ms");
```

4.3.5 Classe Program

All'interno della classe Program (contenuta nell'omonimo file) è stato modificato il metodo *Main()*. Prima dell'avvio del servizio è stato aggiunto il codice che viene riportato sotto.

```
 //estrazione della configurazione dal file appsettings.json
Dictionary<string, string> db_options = config.GetSection("SQLServer").
    GetChildren().ToDictionary(x => x.Key, x => x.Value);
Dictionary<string, bool> spark_options = config.GetSection("SparkOptions").
    Get<Dictionary<string, bool>>();
IEnumerable<string> tables_names = config.GetSection("SparkTables").Get<
    IEnumerable<string>>();
string path = spark_options["LocalTest"] ? config.GetSection("SparkPath").
    GetValue<string>("Local") : config.GetSection("SparkPath").GetValue<
    string>("Server");

Stopwatch sw_session = Stopwatch.StartNew();
sw_session.Start();

//avvio di Spark
EsiSparkService.EsiSparkStartUp(path, tables_names, spark_options,
    db_options);

sw_session.Stop();
Console.WriteLine($"Spark startup elapsed time: {sw_session.
    ElapsedMilliseconds}");
```

In sintesi, le prime quattro istruzioni estraggono dal file *appsettings.json* dei parametri di configurazione. Questi saranno argomento della funzione *EsiSparkService.EsiSparkStartUp()* di cui è stata già definita l'implementazione. Il tempo impiegato ad eseguire questa funzione verrà cronometrato tramite uno *Stopwatch*.

4.4 Esecuzione dell'applicativo con Spark

Per eseguire l'applicativo con Spark nell'ambiente di sviluppo (Visual Studio) si deve aprire un promt dei comandi all'interno del progetto EsiHistorian.WebAPI ed eseguire il comando seguente.

```
spark-submit --driver-memory Xg --packages io.delta:delta-core_2.12:2.0.0
    --conf "spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension"
    --conf "spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.
    catalog.DeltaCatalog" --jars <PATH>/mssql-jdbc-12.2.0.jre8.jar --class
    org.apache.spark.deploy.dotnet.DotnetRunner microsoft-spark-3-2_2
    .12-2.1.1.jar debug
```

Da notare l'integrazione del pacchetto io.delta:delta-core_2.12:2.0.0 (libreria Delta Lake) e della sostituzione del catalogo di default con quello del pacchetto citato. Le interrogazioni verranno lanciate sempre da Swagger tramite le rotte *GetGroupedFacts* (per SQL Server) e *GetGroupedFactsSpark* (per Spark). Nel Capitolo 5, verrà cambiata la quantità di memoria associata all'opzione - - *driver-memory* per condurre dei test sulle performance.

Capitolo 5

Valutazione delle performance

5.1 Ambiente di testing

Per condurre i test si è fatto affidamento ad una macchina aziendale (differente da quella di sviluppo) con le seguenti caratteristiche:

- memoria centrale: 32 GB;
- numero di core logici: 16;
- memoria di massa (SSD): 512 GB.

Dentro questa macchina è stato preparato l'ambiente in modo analogo alla Sezione 4.1.1. Volendo però simulare il "deploy" della soluzione lato cliente, gli strumenti legati allo sviluppo, come .NET SDK, non sono necessari. All'effettivo, nella macchina sono stati installati:

- SQL Server con SQL Server Management Studio. Il secondo è un IDE per la configurazione, la gestione e l'amministrazione di tutti i componenti, le istanze e i database all'interno di SQL Server;
- Java;
- il bundle di hosting .NET, un programma di installazione per il runtime di .NET. Il bundle avrà il compito di eseguire l'applicazione una volta pubblicata (in modalità "framework-dependent").

Successivamente dalla macchina di sviluppo a quella di test sono stati trasferiti:

- il datawarehouse all'interno dell'istanza SQL Server (attraverso un file di backup);
- i file parquet delle tabelle che verranno usati da Spark;

- la cartella di pubblicazione dell'applicativo, in cui è stata modificata la stringa di connessione all'istanza di SQL Server con l'indirizzo della macchina di test.

Per lanciare l'applicativo è stato aperto un terminale nella cartella di pubblicazione dell'applicazione; il comando da eseguire è analogo a quello mostrato in Sezione 4.4, con il nome dell'eseguibile generato (EsiHistorianWebApi.exe) al posto del termine "debug".

```
spark-submit --driver-memory Xg --packages io.delta:delta-core_2.12:2.0.0
    --conf "spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension"
    --conf "spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.
    catalog.DeltaCatalog" --jars <PATH>/mssql-jdbc-12.2.0.jre8.jar --class
    org.apache.spark.deploy.dotnet.DotnetRunner microsoft-spark-3-2_2
    .12-2.1.1.jar EsiHistorianWebApi.exe
```

5.2 Conduzione dei Test

I test hanno l'obiettivo di comparare i tempi di esecuzione delle interrogazioni attraverso SQL Server e Spark. Al fine di ottenere dei risultati più veritieri, le query sono state eseguite molteplici volte per poi riportare nei grafici il tempo medio tra le varie esecuzioni. La query tipo è quella riportata di seguito.

```
SELECT [Year], [Month], [Day], [Hour], [dim_line].[line_name],
[dim_Parameter].[Parameter_name], AVG(value) as Value,
MIN(value) as ValueMin, MAX(value) as ValueMax
FROM [fact]
INNER JOIN [dim_date] ON [fact].[date_id] = [dim_date].[date_id]
INNER JOIN [dim_time] ON [fact].[time_id] = [dim_time].[time_id]
INNER JOIN [dim_line] ON [fact].[line_id] = [dim_line].[line_id]
INNER JOIN [dim_Parameter] ON
[fact].[Parameter_id] = [dim_Parameter].[Parameter_id]
WHERE ([dim_date].[date_id] > '2023-1-2'
AND [dim_date].[date_id]
< '2023-4-30' OR ([dim_date].[date_id] = '2023-1-2' AND
[dim_time].[time_id] >= '0:0:0') OR ([dim_date].[date_id] =
'2023-4-30' AND [dim_time].[time_id] <= '2:0:0')) AND
[dim_line].[line_id] = '1'
AND [dim_Parameter].[Parameter_id] = 'PRESS_HM'
GROUP BY [Year], [Month], [Week], [Day], [Hour],
[dim_line].[line_name], [dim_Parameter].[Parameter_name]
ORDER BY [Year], [Month], [Day], [Hour],
[dim_line].[line_name], [dim_Parameter].[Parameter_name]
```

I test verranno riportati all'interno di grafici a linea su uno spazio bidimensionale:

- il numero di righe interessate (asse x): la query presenta una clausola WHERE che altera il numero di righe da elaborare per creare gli aggregati. Cambiando gli intervalli temporali sono state interessate le seguenti quantità di record: 1M, 5M, 10M, 15M, 20M e 25M;
- quantitativo di RAM (asse y): associata sia a Spark sia a SQL Server;

La quantità di core disponibili è stata lasciata al massimo in quanto il numero delle partizioni in Spark era stato scelto sulla base del primo.

5.2.1 Test SQL Server

Per comprendere al meglio i test effettuati, è opportuno descrivere preliminarmente la logica dietro l'esecuzione di una query all'interno di SQL Server. Un volta che il DBMS riceve la query in linguaggio SQL, a valle del processo di analisi (lessicale, sintattica e semantica) cerca di trovare il miglior piano di esecuzione. Il processo di ottimizzazione costruisce un albero, detto albero delle alternative in cui:

- i nodi sono le operazioni (es: JOIN);
- i percorsi radice-foglia rappresentano un piano di esecuzione.

Il DBMS valuterà il costo associato ad ogni piano, in termini di accessi in memoria secondaria/centrale, e alla fine eseguirà il piano con il costo minimo. Ritornando ai parametri di SQL Kata, introdotti in Sezione 4.3.4, questi di norma consentono al motore SQL di memorizzare nella cache e riutilizzare lo stesso piano di query anche se i parametri vengono modificati. Quest'opzione però non è risultata vantaggiosa nelle prove effettuate. A riprova di ciò le interrogazioni con SQL Server sono state effettuate in due varianti:

- aggiungendo un "hint" per ricalcolare il piano ad ogni esecuzione: *OPTION(RECOMPILE)*;
- senza l'hint *OPTION(RECOMPILE)*.

Il primo grafico (Figura 5.1) rispetto al secondo (Figura 5.2), mostra come il ricalcolo del "query plan" comporti un miglioramento nel momento in cui il numero di righe affette è maggiore di 15 milioni. Inoltre, in ogni grafico vi sono quattro serie: ognuna con un quantitativo diverso di memoria massima accessibile a SQL Server [11]. Come è facile osservare, le serie sono pressoché sovrapposte, dunque tale variazione non ha comportato miglioramenti nei tempi. Questo probabilmente è dovuto dall'esiguo uso che SQL Server fa della memoria centrale.

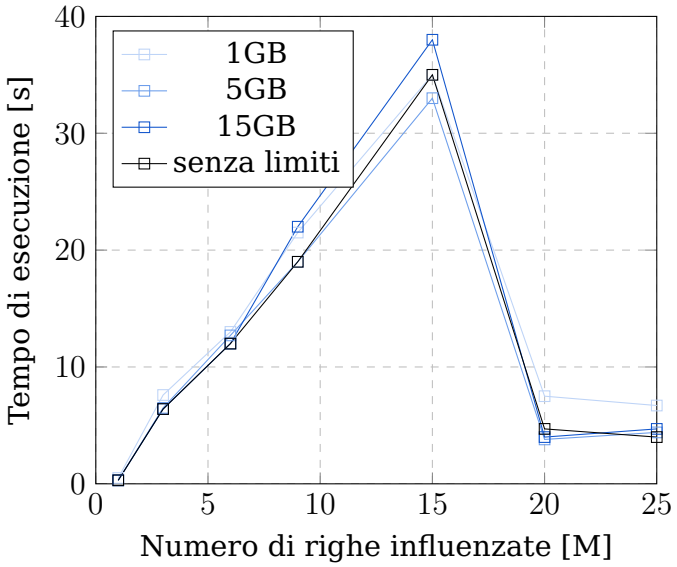

Figura 5.1: Test SQL Server con OPTION(RECOMPILE)

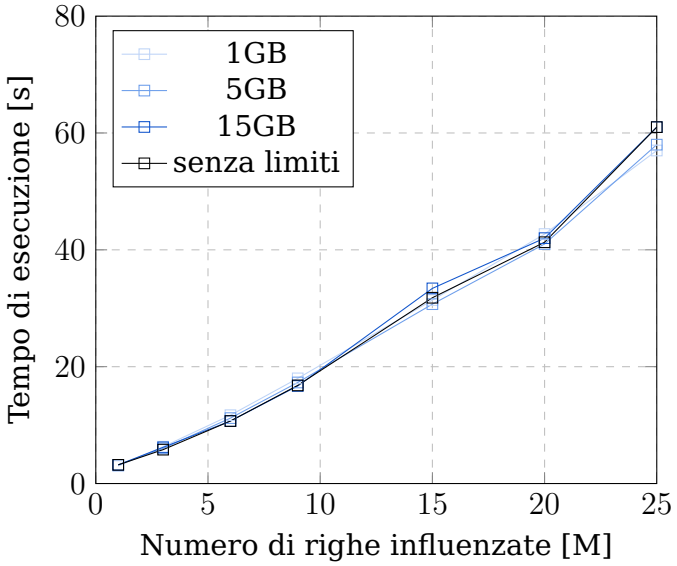

Figura 5.2: Test SQL Server senza OPTION(RECOMPILE)

5.2.2 Test Spark

I test effettuati per Spark sono stati due:

- esecuzione delle query (identiche a quelle di cui sopra) sulle tabelle mantenute "separatamente" in cache;
- esecuzione delle query (con alcune modifiche) sulla tabella definita dai JOIN della "fact" con le dimensioni, anch'essa mantenuta in cache;

La query verso la tabella definita sui JOIN deve subire delle modifiche. Per fare questo all'interno dell'applicazione, più precisamente nella classe *QueryBL* sono state commentate tutte le istruzioni che generavano le clausole di JOIN nella query ed sono state sistemate ulteriori incongruenze nei nomi delle colonne.

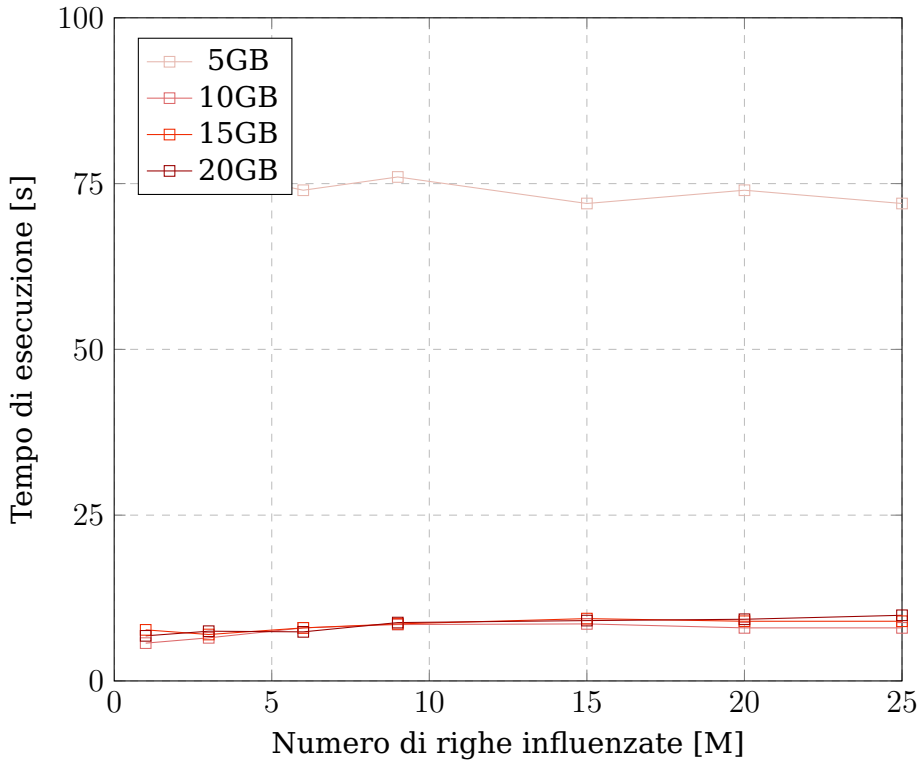

Figura 5.3: Test Spark senza pre-JOIN

Nel test di Figura 5.3, in tutte e quattro le configurazioni, l'evoluzione dei tempi di esecuzione rimane costante all'aumentare delle righe influenzate. In aggiunta si può notare come Spark necessiti di un quantitativo di RAM superiore a 5GB per ottenere delle prestazioni accettabili.

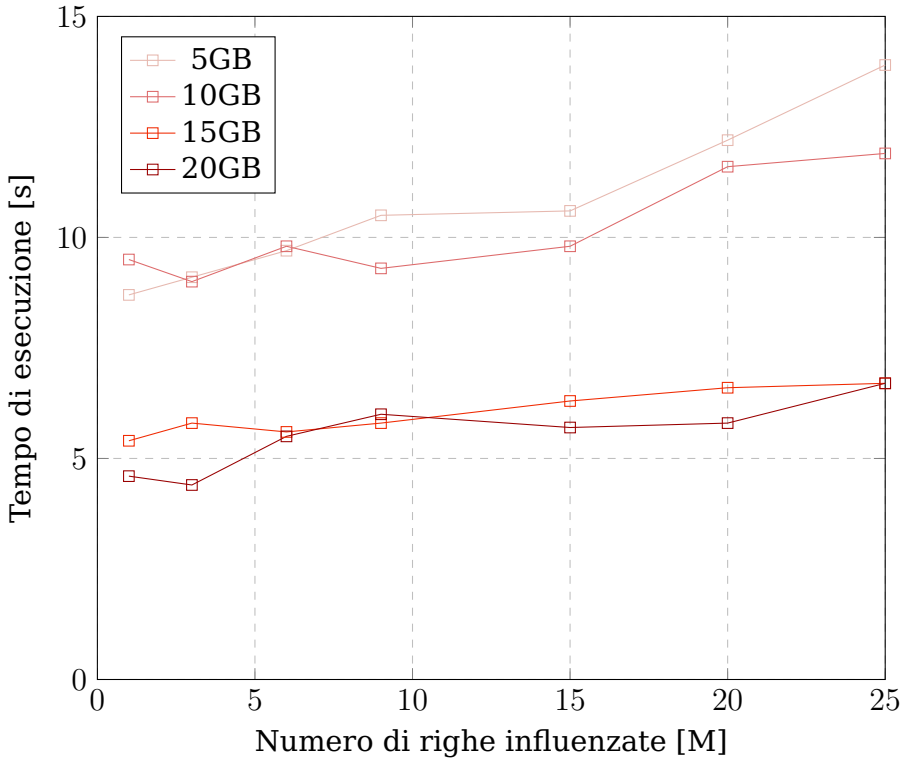

Figura 5.4: Test Spark con pre-JOIN

In Figura 5.4, solo la serie associata a 5GB di RAM indica una significativa diminuzione dei tempi. Negli altri casi invece sembrerebbe vi sia un aumento dei tempi, anche se di poco, rispetto al test precedente.

Capitolo 6

Conclusioni

Per dare una visione complessiva, sono state riportate all'interno di Figura 6.1 le serie di ogni grafico precedente, con il maggiore quantitativo di memoria disponibile.

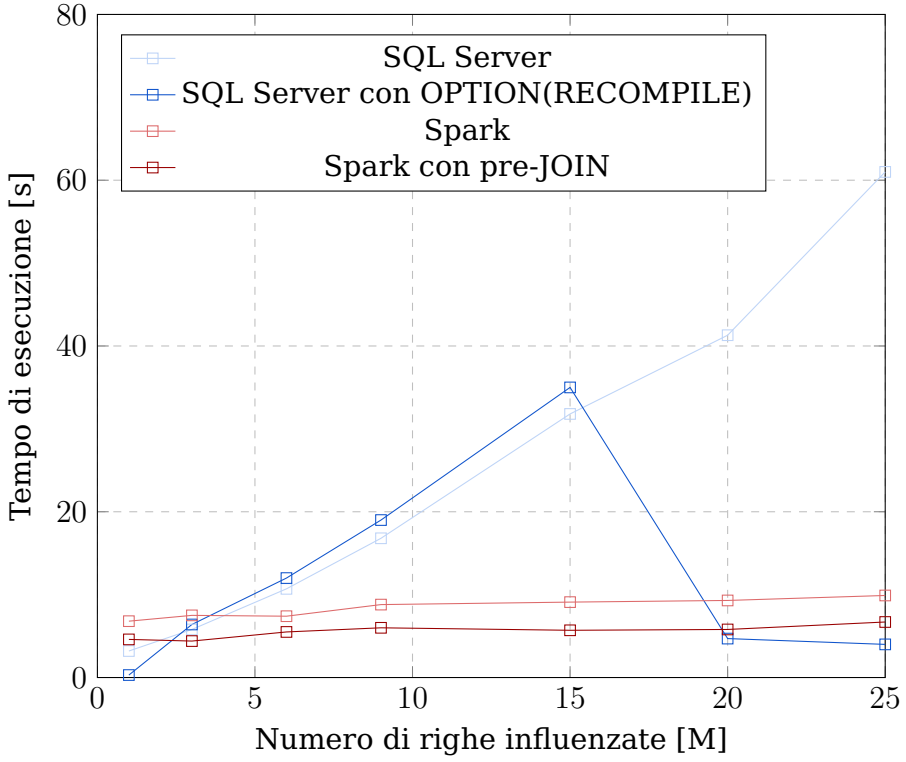

Figura 6.1: Confronto tra SQL Server e Spark

Dai test sembrerebbe che la vecchia soluzione basata su SQL Server sia ancora vantaggiosa in quei casi in cui la selettività della query è molto forte (es: nella clausola WHERE si considera un intervallo di tempo minore di un mese). D'altro canto nel momento in cui il numero delle righe tende a salire SQL Server inizia a "soffrire" mentre Spark mantiene i suoi tempi

costanti. Da tenere in considerazione anche il caso in cui SQL Server con *OPTION(RECOMPILE)* riesca ad ottenere le prestazioni di Spark (anche leggeremente meglio) per quantità di righe pari a 20M e 25M. Nonostante questo, bisogna tenere in considerazione che il processo di ricalcolo del "query plan", ad ogni interrogazione, potrebbe risultare più oneroso in altre casistiche. L'ultima considerazione riguardo al fatto che Spark è stato usato su una singola macchina. Lo step successivo potrebbe essere mettere in piedi un cluster di macchine per valutare un potenziamento ulteriore. Questa valutazione ovviamente deve essere affiancata ad uno studio di "costi-benifici" per constatare la convenienza di questa modalità.

Bibliografia

[1] Michael Armbrust e Brenner Heintz Burak Yavuz. *Diving Into Delta Lake: Unpacking The Transaction Log*. https://www.databricks.com/blog/2019/08/21/diving-into-delta-lake-unpacking-the-transaction-log.html. 21/08/2019.

[2] Luis Quintanilla e David Pine. *Get started with .NET for Apache Spark*. https://learn.microsoft.com/en-us/previous-versions/dotnet/spark/tutorials/get-started?tabs=windows. 10/09/2020.

[3] Mary McCready e David Pine. *What is .NET for Apache Spark*. https://learn.microsoft.com/it-it/dotnet/spark/what-is-apache-spark-dotnet. 16/12/2022.

[4] Niharika Dutta e David Pine. *6 recommendations for optimizing a Spark job*. https://towardsdatascience.com/6-recommendations-for-optimizing-a-spark-job-5899ec269b4b. 16/12/2022.

[5] Niharika Dutta e David Pine. *Connect .NET for Apache Spark to SQL Server*. https://learn.microsoft.com/en-us/dotnet/spark/how-to-guides/connect-to-sql-server. 16/12/2022.

[6] Niharika Dutta e David Pine. *Submitting Applications*. https://spark.apache.org/docs/latest/submitting-applications.html.

[7] Petrica Leuca. *Data processing with Spark: data catalog*. https://medium.com/@petrica.leuca/data-processing-with-spark-data-catalog-45236f590296. 22/06/2022.

[8] Matthew Powers. *How to Rollback a Delta Lake Table to a Previous Version with Restore*.

[9] Christoph Nienaber e Rico Suter. *Documentazione delle API Web ASP.NET Core con Swagger/OpenAPI*. http://aiweb.techfak.uni-bielefeld.de/content/bworld-robot-control-software/. 12/09/2023.

[10] Marco Schaefer. *Onion Architecture explained — Building maintainable software*. https://marcoatschaefer.medium.com/onion-architecture-explained-building-maintainable-software-54996ff8e464. 16/12/2020.

[11] *Server memory configuration options*. https://learn.microsoft.com/en-us/sql/database-engine/configure-windows/server-memory-server-configuration-options?view=sql-server-ver16. 8/21/2023.

[12] Subash Sivaji. *Types of Apache Spark tables and views*. https://medium.com/@subashsivaji/types-of-apache-spark-tables-and-views-f468e2e53af2. 10/05/2019.