ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA

DIPARTIMENTO DI INFORMATICA — SCIENZA E INGEGNERIA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA E SCIENZE INFORMATICHE

UN FRAMEWORK PER L'ANALISI DI BIG DATA CON ELEVATA ETEROGENEITÀ ALL'INTERNO DI MULTISTORE

Tesi in

Sistemi informativi

Relatore:
Prof. Matteo Golfarelli

Presentata da:
Chiara Forresi

Correlatore:
Dott. Enrico Gallinucci

Sessione II
Anno Accademico 2019/2020

Indice

Elenco delle figure

Introduzione

È ormai noto che ci troviamo nell'era dei big data: miliardi di dati vengono prodotti quotidianamente da fonti diverse quali i social network, i sensori, i dispositivi IoT. Tali dati sono caratterizzati dalle ben note *4v* [42]: *volume* per quanto riguarda la quantità di dati, *velocità* da un lato per quanto riguarda la velocità con cui essi vengono prodotti e dall'altro quella con cui sono richiesti, *veracità* per sottolineare quanto possano essere accurati o veritieri i dati, *varietà* per indicare quanto possano essere differenti i dati tra loro. Quest'ultimo aspetto spesso viene sottovalutato, pur essendo molto importante. Un ambito in cui esso si manifesta è nelle strutture dati *schema-less*, ovvero strutture dati dove il concetto di schema non è rigido. In questo contesto rientrano i database NoSQL [19], i quali offrono diversi modelli dati: *documentale*, *wide-column*, *grafo* e *key-value*. A differenza del classico modello relazionale, essi non hanno un concetto di schema definito in modo rigido e da far rispettare a tutti i record di una collezione. Data l'assenza di uno schema rigido, il paradigma che abilita l'accesso ai dati viene chiamato *schema-on-read*, differente da quello tipico del modello relazionale (*schema-on-write*): il controllo dello schema non avviene cioè in fase di scrittura del dato nella collezione, ma solo in fase lettura. Il vantaggio di questo paradigma è dato dalla libertà di poter memorizzare le informazioni senza seguire uno schema fisso, ma è controbilanciato dalla maggior difficoltà a leggere ed analizzare i dati a causa dell'eterogeneità degli stessi. La tecnica di modellazione utilizzata nell'ambito dei database NoSQL è definita *orientata agli aggregati*, ovvero a un concetto incapsulato e atomico; l'idea consiste nell'incapsulare all'interno

di un unico oggetto tutti i dati ad esso correlati (ad esempio, un cliente e tutti suoi ordini). Il vantaggio principale di questa tecnica risiede nella semplificazione delle operazioni di lettura e scrittura in un contesto operazionale (si evita infatti di dover eseguire join tra tabelle diverse e la distribuzione dei dati è semplificata); questa *denormalizzazione* dei dati espone però anche al rischio di generare incosistenze.

In tale ambito si parla di *multistore* quando ci si riferisce all'uso di database con modelli dati diversi [66] che vengono esposti con un'unica interfaccia di interrogazione, sia per sfruttare caratteristiche di un modello dati piuttosto che un altro che per le maggiori performance che hanno i database NoSQL in contesti distribuiti. Considerando un multistore, tra le sorgenti che lo compongono – e anche all'interno di una singola sorgente schema-less – si possono possono manifestare le seguenti problematiche: (i) record con campi mancanti: due rappresentanti uno stesso concetto potrebbero avere attributi diversi; (ii) attributi con diverso nome o tipo che descrivono lo stesso concetto; (iii) presenza di dati denormalizzati o replicati: dato che l'impiego di database NoSQL spesso avviene in contesti in cui si tende a prediligere alte prestazioni anche a discapito della consistenza, spesso si tende a denormalizzare o replicare i dati per questo motivo.

Gli approcci tradizionali prevedono una riconciliazione degli schemi sorgenti sotto un'unica "visione concettuale"; tuttavia, essi non risultano adatti in contesti come quello in esame. Infatti, in ambito big data: (i) gestire un contesto di integrazione tradizionale, in un contesto presumibilmente distribuito, risulta molto difficile e costoso; (ii) l'evoluzione degli schemi nei dati potrebbe essere tale da dove aggiornare in continuazione la visione globale dei dati considerata. A questo scopo nasce la necessità di approcci più leggeri delle tradizionali ETL e, per gestire l'eterogeneità che emerge e supportare le analisi aziendali, di costruire sistemi potenti e intelligenti che sfruttino la conoscenza presente nelle sorgenti dati. Gli approcci usati sono chiamati *pay-as-you-go* [38]: l'integrazione avviene progressivamente man mano che i dati vengono esplorati dall'utente, aggirando la complessità di approcci di

integrazione tradizionali e restituendo risposte *best-effort* (migliori sforzi) o approssimative. Per cui, nasce il concetto di *dataspace* [23] come rappresentazione logica e di alto livello dei dataset disponibili che viene creato attraverso il processo di integrazione integrato, iterativo, incrementale e leggero, seguendo la filosofia pay-as-you-go.

Obiettivo di questo lavoro tesi è studiare, progettare e realizzare un sistema per interrogare sorgenti dati eterogenee in un contesto multistore con l'intento di fare analisi situazionali, considerando le possibili problematiche di varietà sovraesposte e appoggiandosi all'integrazione fornita dal dataspace. Lo scopo finale è quello di sviluppare un prototipo che esponga un'interfaccia per interrogare il dataspace con la semantica *GPSJ* (Generalized Projection / Selection / Join), ovvero la classe di query più comune nelle applicazioni OLAP. Un'interrogazione nel dataspace dovrà essere tradotta opportunamente in una serie di interrogazioni nelle sorgenti e successivamente, attraverso un livello middleware, i risultati parziali dovranno essere integrati tra loro. Il lavoro di tesi è organizzato nei seguenti capitoli.

- Nel Capitolo 1 si analizzano i contesti di dati eterogenei, introducendo termini e concetti usati nell'ambito di memorizzazione di dati eterogenei e la letteratura correlata.

- Nel Capitolo 2 si formalizzano le informazioni per la costruzione del dataspace e le possibili tecniche per automatizzare la loro estrazione.

- Nel Capitolo 3 si descrivono gli aspetti che riguardano la formulazione di interrogazioni nel dataspace, focalizzandosi in particolare sullo stack algoritmico.

- Nel Capitolo 4 viene presentata l'architettura del prototipo e le tecnologie utilizzate al suo interno.

- Nel Capitolo 5 vengono descritti i dati utilizzati, i test effettuati e i relativi risultati ottenuti.

- Infine, nelle Conclusioni, vengono tirate le somme su quanto sviluppato e i possibili sviluppi futuri.

Capitolo 1

Memorizzazione di dati eterogenei

Per addentrarci all'interno del contesto della tesi, in questo capitolo vengono introdotti alcuni termini e concetti usati nell'ambito di memorizzazione di dati eterogenei e viene analizzata la letteratura correlata.

1.1 Modelli dati

Innanzitutto, per poter comprendere quanto verrà successivamente espresso è necessario precisare che negli ultimi anni sono nate esigenze diverse per quanto riguarda la memorizzazione dei dati che hanno portato alla nascita di nuove metodologie, tecniche e, di conseguenza, tecnologie per raccogliere i dati. La motivazione di questo cambiamento si colloca all'interno del contesto moderno di necessità di scalabilità e performance, del quale fanno parte anche i Big Data. Questa transizione è possibile notarla nell'evoluzione e nelle sfumature che si insidiano all'interno del concetto di **modello dati**, il quale è considerato un'entità fondamentale di astrazione all'interno di un DBMS (*Database Management System*). Un modello dati è considerato come un insieme di strumenti concettuali per descrivere i dati, la loro semantica, i vincoli e le relazioni tra essi [61]. È importante sottolineare che c'è una

forte differenza tra modello dati e schema, infatti, mentre il modello dati è uno strumento di astrazione, uno schema è il risultato prodotto dallo strumento utilizzato [67]. Ogni DBMS segue una propria logica di modellazione, implementando uno o più modello dati.

Il modello dati usato tradizionalmente è il modello dati relazionale: da un lato è noto come in questo modello ci sia molta rigidità nella definizione dei dati, del tipo di dato da utilizzare e delle relazioni tra i dati, nel pieno rispetto delle proprietà ACID (*Atomicity, Consistency, Isolation e Durability*), dall'altro le nuove esigenze e tecnologie hanno messo in dubbio l'utilità di questa rigidità in molti contesti.

> Il tema del design del database relazionale si concentra sulle risposte, mentre nei database NoSQL si concentra sulle domande [40].

Questa frase ben sintetizza il concetto: rispetto ai database relazionali in cui la modellazione è decisa dalla struttura dati, nei nuovi database la modellazione tiene presente il tipo di carico di lavoro che verrà eseguito sui dati. Database NoSQL (*Not Only SQL*) [19] sta ad indicare la famiglia di database con modelli dati diversi da quello relazionale. Una delle sfide principali è, dunque, comprendere quale modello dati sia il più adatto per una tale applicazione. Di seguito verranno presentati i principali modelli dati usati per la memorizzazione di dati dai principali DBMS e le loro caratteristiche principali, in particolare si partirà dal riassumere i concetti fondamentali del modello relazionale e successivamente verranno descritti i modelli dati NoSQL principali (chiave-valore, documentale, wide-column, a grafo).

1.1.1 Modello relazionale

Il modello relazionale è modello dati tradizionale utilizzato negli RDBMS (*DBMS relazionali*), lo schema è un concetto rigido da definire prima di poter popolare il database e i dati inseriti dovranno seguire tale definizione di schema, per quanto riguarda nome e tipologia di attributi. I dati sono,

per cui, memorizzati all'interno di una tabella e ciascun record che compone la tabella è memorizzato come una singola entità su disco.

1.1.2 Modello chiave-valore

Nel modello dati chiave-valore ogni database contiene una o più collezioni ed ogni collezione contiene una lista di coppie chiave-valore, dove: (i) la chiave è una stringa unica; (ii) il valore è un BLOB (*oggetto binario di grandi dimensioni*). In questo contesto la raccolta dei dati può essere vista come un dizionario, infatti essa è indicizzata per chiave e il valore può contenere svariate informazioni.

- Il *livello di atomicità* considerato è, per l'appunto, la coppia chiave-valore, per cui un'operazione di aggiornamento che coinvolge tante righe non è per definizione atomica.
- Per quanto riguarda *l'espressività delle interrogazioni*, il valore risulta essere una "scatola nera": non è possibile interrogare i dati basandosi sul valore, i valori non possono essere indicizzati e, spesso per questo motivo, le informazioni che riguardano lo schema dei dati sono indicate nella chiave.

1.1.3 Modello documentale

Nel modello documentale ogni database contiene una o più collezioni e ogni collezione contiene una lista di documenti strutturati gerarchicamente (di solito in formato JSON). Ogni documento, a sua volta, contiene un set di campi che corrispondono a coppie chiave-valore dove: (i) la chiave è una stringa unica nel documento; (ii) il valore può essere sia semplice (ad esempio una stringa, un numero o un booleano) che complesso (oggetti, array, BLOB).

- Il *livello di atomicità* considerato è il documento.
- A differenza del modello chiave-valore, il valore è visibile dal DBMS e, pertanto, i *linguaggi di interrogazione* risultano piuttosto espressivi.

Infatti, è possibile creare indici sui campi, filtrare sui campi, restituire eventualmente più documenti con un'interrogazione, aggiornare campi specifici e decidere quali campi proiettare.

Per questo modello esistono diverse implementazioni con svariate funzionalità, la principale da sottolineare è che sono presenti tecnologie con connettori ai principali strumenti Big Data.

1.1.4 Modello wide-column

Nel modello dati *wide-column* ogni database contiene una o più *column-family* (famiglia di colonne), ognuna di esse contiene un elenco di righe sottoforma di coppie chiave-valore, dove: (i) la chiave è una stringa unica nella column-family; (ii) il valore è un insieme di colonne. Ogni colonna è, a sua volta, una coppia chiave-valore in cui: (i) la chiave è una stringa unica nella riga; (ii) il valore è di tipo semplice o complesso (supercolonna), il livello di innestamento è però limitato rispetto al modello documentale. Rispetto al modello relazionale, è bene sottolineare che le righe specificano solo le colonne per le quali esiste un valore, per cui questo modello dati risulta particolarmente adatto per matrici sparse. Considerata la somiglianza con il modello relazionale, spesso per le interrogazioni vengono usati linguaggi SQL-like.

- Il *livello di atomicità* è la riga.

- L'*espressività delle interrogazioni* risulta, in questo caso, una via di mezzo tra il modello chiave-valore e quello documentale infatti non in tutti i casi c'è una possibilità di filtrare i valori per colonna o aggiornare colonne specifiche, è sempre possibile proiettare determinate colonne e proiettare più righe ma gli indici a livello di colonna sono scoraggiati.

Inoltre, per concludere, è bene sottolineare che il modello wide-column è diverso dall'archiviazione dei dati in forma colonnare utilizzata per le applicazioni di tipo analitico.

1.1.5 Modello a grafo

Nel modello a grafo ogni database contiene uno più grafi, che sono un insieme di vertici ed archi dove: (i) il vertice solitamente rappresenta un'entità del mondo reale; (ii) l'arco rappresenta le relazioni dirette tra i vertici. Vertici e archi sono solitamente descritti da proprietà, gli archi sono memorizzati come puntatori fisici.

- Il *livello di atomicità* è la transazione, infatti la garanzia è molto più forte rispetto agli altri modelli NoSQL sopra citati.

- Per il motivo del punto sopra, i database a grafo si usano per modellare contesti completamente diversi e, pertanto, il *linguaggio e il meccanismo delle interrogazioni* sono abbastanza diversi. Vi è supporto alle transazioni, a indici, selezioni e proiezioni e il linguaggio d'interrogazione è basato sul modello stesso.

1.1.6 Modellazione basata sugli aggregati

I modelli dati key-value, documentale e wide-column sono chiamati *orientati agli aggregati*. In ciascuno di essi rispettivamente l'aggregato è la coppia key-value, il documento e la riga. Si tratta di una *modellazione basata su query* e l'aggregato è da considerarsi un blocco atomico, questo rientra nel concetto di incapsulamento e, di conseguenza, vengono evitati il più possibile i join a discapito di possibili denormalizzazioni e incongruenze dei dati. In generale, non esistono delle strategie di progettazione per gli aggragati: dipende dal carico di lavoro che si intende ottimizzare. Per questo scopo è necessaria un'analisi preliminare per comprendere quale modello dati risulta essere più adatto per supportare determinate applicazioni.

Il modello dati a grafo è, invece, intrinsecamente diverso dagli altri, infatti si parla di una *modellazione data-driven*, dove ci si incentra sulle relazioni tra le entità piuttosto che sulle entità in sè. Per questo motivo la scalabilità, caratteristica peculiare dei modelli orientati agli aggregati, è limitata:

la frammentazione di un grafo risulta difficile senza dover eliminare delle relazioni.

1.2 Database NoSQL e persistenza poliglotta

I database NoSQL, come sottolineato in precedenza, sono database che supportano modelli dati diversi dal modello relazionale, usato nei database relazionali tradizionali.

La prima differenza che emerge tra i database NoSQL e quelli tradizionali riguarda l'eterogeneità a livello di schema. Nei database relazionali l'approccio usato è *"schema-first, data later"* (prima lo schema, poi i dati): il paradigma di accesso ai dati è *schema-on-write* ed è richiesto che tutti i record di una tabella soddisfino uno schema predefinito. Nei database NoSQL, invece, il paradigma di accesso ai dati è *schema-on-read* e, dunque, in questo caso il controllo dello schema non viene fatto in fase di lettura, ma delegato all'utente finale che andrà ad accedere al dato. Questo deriva dal fatto che in contesti NoSQL si parla di *schema-less*, che non significa che lo schema nei dati non esiste ma che, rispetto al modello relazionale, non c'è un concetto di schema rigido e l'approccio utilizzato è di tipo *"soft-schema"*: ogni record incorpora la propria definizione di schema. Infatti, due record appartenenti alla stessa collezione possono differire per quanto riguarda il numero (attributi mancanti), il nome o il tipo di attributi (ad esempio, utilizzando convenzioni differenti per i nomi e/o i tipi degli attributi). Questo aspetto è dovuto principalmente all'evoluzione degli schemi dei dati nel tempo e all'acquisizione di dati da sorgenti che adottano rappresentazioni di schemi differenti per le stesse entità.

Un'ulteriore considerazione da precisare è che i database NoSQL, oltre a memorizzare i dati in maniera diversa, a differenza degli RDBMS, possono essere distribuiti facilmente in un cluster di computer. Per questo motivo, i database NoSQL vengono spesso utilizzati in contesti in cui si desiderano alte performance, anche a discapito della consistenza, per cui vengono

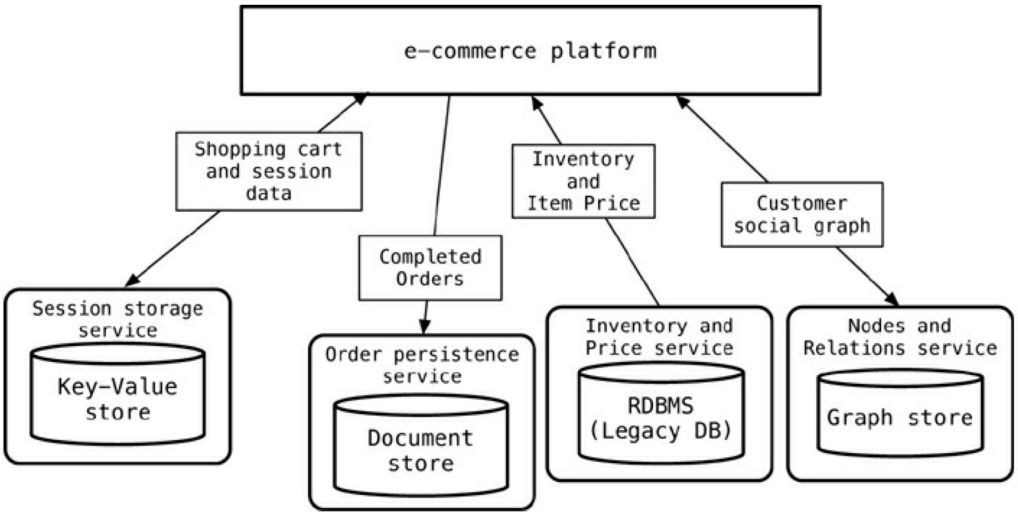

Figura 1.1: Esempio di persistenza poliglotta.

Tratta da [59]

adottate tecniche di modellazione quali *replicazione* e *denormalizzazione* dei dati. Dunque, ci possono essere, rispettivamente, record contenenti le stesse informazioni replicati più volte o record memorizzati a livelli di dettaglio non congruenti rispetto ai dati: ad esempio, si potrebbe pensare di memorizzare i dati dei prodotti allo stesso livello di dettaglio di quelli delle righe d'ordine. Questo porta, inevitabilmente, a possibili incongruenze dei dati, infatti l'utilizzo di queste tecniche va ad ottimizzare carichi di lavoro operazionali (OLTP – *Online Transaction Processing*), ma rende più difficili quelli analitici (OLAP – *On-Line Analytical Processing*).

Sulla stessa linea d'onda, in contesti aziendali, è nata la necessità di voler sfruttare le caratteristiche migliori di ogni tecnologia e modello dati, usandone più di una, in modo da poterne trarre il meglio in termini di performance e ottimizzare il carico di lavoro operazionale, nello specifico si parla di *persistenza poliglotta* [59, 66]. Un esempio è mostrato in Figura 1.1, in cui l'accento è posto su una piattaforma e-commerce che utilizza, sotto forma di servizi messi a sua disposizione, i seguenti database: (i) chiave-valore per memorizzare i dati del carrello e delle sessioni; (ii) documentale per la memorizzazione degli ordini completati; (iii) relazionale per memorizzare il listino

dei prodotti; (iv) a grafo per memorizzare i dati social relativi alla clientela.

In letteratura esistono diversi approcci utilizzati per eseguire analisi e interrogazioni in contesti di persistenza poliglotta. Una tassonomia di soluzioni allo stato dell'arte è descritta in [66]:

(i) *federated database system* (sistema di database federato) e *polyglot system* (sistema poliglotta): supportano archivi di dati omogenei esponendo, rispettivamente, un'interfaccia di interrogazione singola e multipla; per questo motivo, è necessario uno sforzo in termini di trasformazioni di dati e spazio per archiviarli nel nuovo database integrato;

(ii) *multistore system* (sistema multistore): supporta archivi di dati eterogenei ed espone un'unica interfaccia d'interrogazioni, molte soluzioni sono focalizzate sull'integrazione di database NoSQL e relazionali;

(iii) *polystore system* (sistema polystore): supporta archivi di dati eterogenei ed espone più interfacce di interrogazione, combinando così l'espressività sia dei sistemi poliglotti che di quelli sistemi multistore.

Per quanto riguarda le voci (ii) e (iii), è necessario un livello middleware e, anche in questi casi, è necessario definire una vista globale o un database o specificare specificare una particolare origine dati da utilizzare. In questo elaborato di tesi ci si concentra sui *sistemi multistore*.

Di seguito verranno trattate due problematiche correlate che entrano in gioco in questo contesto. In prima istanza, nella Sezione 1.2.1, viene descritta una possibile soluzione attuale per integrare più sorgenti dati. Di seguito, nella Sezione 1.2.2, vengono discusse le problematiche di varietà che si presentano in un sistema multistore e, più in generale, quando si parla di persistenza poliglotta.

1.2.1 Integrazione pay-as-you-go

Innanzitutto bisogna considerare che i processi tradizionalmente utilizzati per il passaggio da contesti operazionali a contesti analitici, come l'ETL *(Extract,*

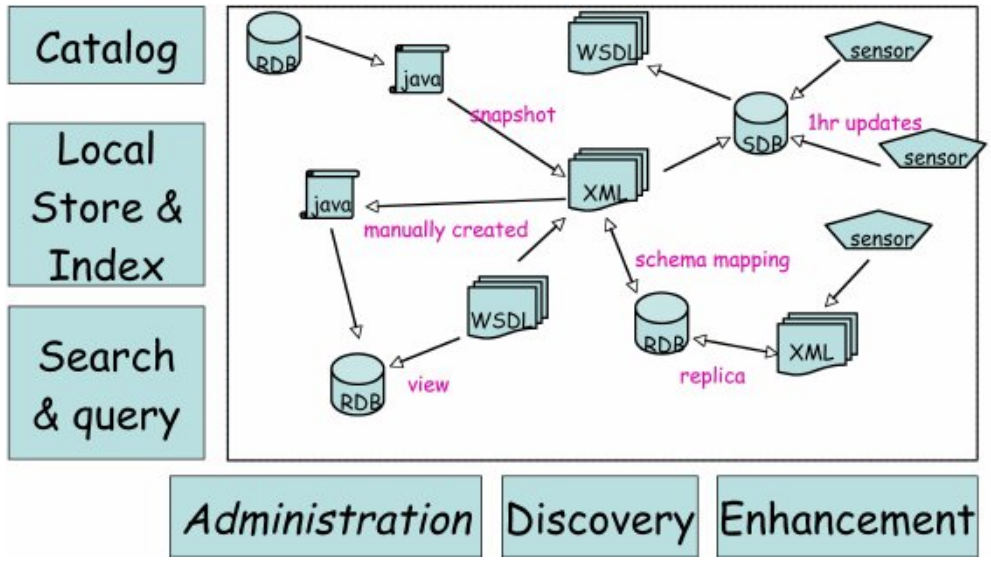

Figura 1.2: Un esempio di dataspace e dei componenti di un sistema dataspace.

Tratta da [23]

Transform, Load), risultano troppo invasivi e rigidi. I nuovi approcci nati sono definiti **pay-as-you-go** [38] (con pagamento a consumo): essi aggirano la complessità degli approcci di integrazione tradizionali, fornendo risposte *best-effort* o approssimative, l'integrazione non è materializzata e viene progressivamente eseguita dall'utente man mano che vengono esplorati i dati disponibili. In quest'ottica nasce il concetto di **dataspace** [23] come rappresentazione logica ed ad alto livello dei dataset disponibili. Il dataspace non è da considerarsi un approccio per l'integrazione dei dati, ma è molto di più: è garantito che ciascuna sorgente appartenente al dataspace fornisca determinate funzionalità, indipendentemente da quanto le sorgenti siano integrate. Come mostrato in Figura 1.2, un dataspace dovrebbe contenere tutte le informazioni rilevanti per una particolare organizzazione, indipendentemente dal loro formato e posizione, e modellare una ricca raccolta di relazioni tra le sorgenti. L'interazione con il dataspace, nell'esempio, è consentita in varie modalità. Uno dei servizi basilari è il *catalogo* delle risorse contenente informazioni basilari come: origine, nome, posizione in origine, dimensioni, data

di creazione e proprietario di ciascuna sorgente. Il catalogo è un'infrastruttura per la maggior parte degli altri servizi del dataspace e potrebbe anche supportare l'esplorazione dei dati. I due servizi fondamentali forniti da un sistema dataspace sono la ricerca e le query. Mentre i DBMS si sono distinti per quanto riguarda il fornire un supporto di query, la ricerca risulta più tollerante, basandosi su somiglianze, e ha un supporto interattivo da parte dell'utente. L'obiettivo di un sistema dataspace dovrebbe essere quello di consentire all'utente di formulare una query di ricerca e perfezionarla in modo iterativo, eventualmente arricchendo e modificando le relazioni presenti tra i dati, in modo automatico o manuale, man mano che l'utente procede nell'esplorazione degli stessi.

1.2.2 Gestione della varietà in contesti di persistenza poliglotta

È bene evidenziare che la varietà in un contesto di persistenza poliglotta si manifesta in maniera pervasiva e bisogna gestirla, soprattutto nella transazione da contesto operazionale a contesto analitico. In particolare, è necessario un modo trasparente per accedere ai dati che vengono frammentati nelle sorgenti e memorizzati in forme differenti, per questo si parla di *cross-database querying* (interrogazioni tra più database). Le sfide principali da affrontare riguardano l'eterogeneità dei dati in termini di modello dati e di schema.

Per affrontare l'eterogeneità a livello di modello dati, alla base di contesti di persistenza poliglotta, è necessario attuare un processo di *query rewriting* (riscrittura delle interrogazioni) dal dataspace (presentato nella Sezione 1.2.1), secondo una semantica ben definita, alle sorgenti. È importante sottolineare che le sorgenti sono i diversi database che operano con linguaggi di interrogazione differenti e, dunque, probabilmente supportano livelli di espressività diversi. Successivamente, è necessario far riferimento ad un livello middleware che vada a combinare e elaborare i risultati provenienti da ciascuna sorgente, decidendo l'ordine e le modalità con cui tali dati verranno integrati tra loro. In questo processo è necessario considerare che le inter-

rogazioni che verranno effettuate sulle sorgenti dovranno tener conto, oltre che al tipo di interrogazione effettuata, degli aspetti di varietà dello schema e quindi saranno a un livello di dettaglio diverso rispetto all'interrogazione originale.

Per quanto riguarda l'eterogeneità a livello di schema essa si presenta sia, come già detto in precedenza, per l'utilizzo di database NoSQL (ovvero database schema-less), sia per la presenza di più database che seguono modelli dati differenti. La varietà presente può essere a vari livelli, infatti, potrebbero manifestarsi le seguenti situazioni: (i) record appartenenti allo stesso concetto ma rappresentati in modi diversi in una o più collezioni: attributi mancanti, attributi che differiscono per tipo e/o per nome a causa dell'utilizzo di convenzioni differenti; (ii) dati memorizzati a livelli di dettaglio differente a causa dell'utilizzo di tecniche di denormalizzazione; (iii) record sovrapposti appartenenti a una o più collezioni, per cui bisognerà definire quali dati mantenere e quali no e in quali modalità, cercando un modo di mantenere più informazioni possibili e corrette. In generale, dunque, il problema comune è quello di avere una grande difficoltà nel definire uno *schema globale* che vada ad integrare quelli presenti in tutte le collezioni del multistore. La soluzione a questo problema è il *dataspace*: esso è lo strumento concettuale usato, grazie al quale viene creata dell'omogeneità sopra alle sorgenti che compongono il sistema multistore. Attraverso il supporto dell'utente, il dataspace crea un livello logico che risulta trasparente e integra le collezioni presenti nel multistore. È bene sottolinearare che, oltre alla conoscenza dell'utente, a supporto dell'integrazione pay-as-you-go entrano in gioco i metadati, essi assicurano che i dati possano essere trovati, attendibili e usati [29]. La meta-conoscenza che può essere estratta a partire dai dati e le informazioni che si possono raccogliere possono essere svariate, tra queste sottolineamo: (i) *informazioni sullo schema* (ad esempio, formato e fonte dei dati); (ii) *informazioni semantiche* (ad esempio, parole chiave e categorizzazione). Infatti, nella transazione verso sistemi sofisticati per la gestione della varietà i metadati sono considerati molto importanti [68].

1.3 Letteratura correlata

La problematica di interrogare dataset distribuiti viene discussa dalla comunità a partire dall'introduzione del concetto di federated database [60] (presentati nella tassonomia descritta nella Sezione 1.2). La varietà in termini dei modelli dati disponibili [34] (ad esempio relazionale, wide-column, documentale) risponde ai diversi requisiti delle moderne applicazioni data-intensive, ma la possibilità di fornire meccanismi di interrogazione trasparenti per interrogare collezioni su larga scala su archivi di dati eterogenei è un'area di ricerca attiva [66]. Di seguito, verranno rispettivamente trattate tre classi principali di soluzioni per risolvere problematiche relative all'interrogazione di dati ad alta varietà. Il primo gruppo di lavori (discusso nella Sezione 1.3.1) si riferisce a soluzioni che mirano ad affrontare l'eterogeneità in contesti con lo stesso modello dati, il secondo (discusso nella Sezione 1.3.2) – considerando le problematiche di eterogeneità descritte a livello di schema nella Sezione 1.2.2 – scende più nel dettaglio e punta ad analizzare tecniche che consentono di fondere e risolvere conflitti su record eterogenei e sovrapposti. Il terzo gruppo di lavori (discusso nella Sezione 1.3.3), infine, si riferisce al problema di interrogare dati distribuiti su sorgenti che seguono modelli dati differenti.

1.3.1 Eterogeneità a livello di schema

Trasformazione del modello di dati. Questo gruppo di lavori suggerisce di attuare una trasformazione a livello di modello dati per facilitare l'accesso a dati con strutture eterogenee. La strategia comune consiste nel cambiare il modello dati sottostante, di solito si passa da un modello dati non-relazionale a un modello dati relazionale. Per cui, questo tipo di soluzioni da un lato porta a una perdita per quanto riguarda la flessibilità dal punto di vista di schema fornita dalla maggior parte dei database NoSQL, dall'altro garantiscono l'utilizzo delle convenzionali tecniche di interrogazione e memorizzazione dei dati relazionali. In particolare, tipicamente, per definire come spostare i dati da un modello dati all'altro vengono definite trasformazioni

personalizzate e mapping [64, 20]. Un approccio mainstream ampiamente utilizzato quando si tratta di database documentali eterogenei consiste nel trasformare i documenti facendo in modo che afferiscano ad un modello dati relazionale [2, 13, 22]. Delle alternative, invece, suggeriscono di memorizzare i documenti seguendo il modello dati wide-column. A questo proposito, MonetDB [37] utilizza una codifica dei dati specializzata, metodi di join e archiviazione per la gestione dei documenti sul modello di dati wide-column. In [2] gli autori usano la definizione del tipo di documento (DTD) per rendere flat i documenti e mapparli in tabelle relazionali. Tuttavia, nonostante i vantaggi portati dall'utilizzo di schemi relazionali e il potere espressivo degli operatori relazionali, il partizionamento per attributi dei dati in tabelle [22] influisce sulle prestazioni del sistema relazionale. Ciò è dovuto alla necessità di eseguire più join per ricostruire i dati iniziali. Inoltre, gli utenti di questi sistemi devono apprendere nuovi schemi ogni volta che vengono inseriti (o aggiornati) nuovi dati perché è necessario rigenerare le viste relazionali.

Interrogazione indipendente dallo schema. Tale classe di lavori comprende soluzioni che superano l'eterogeneità a livello di schema abilitando l'interrogazione indipendente dallo schema. La strategia usata tipicamente si basa su tecniche di query rewriting [56] per riformulare una query in input in una serie di derivazioni. La maggior parte dei lavori si limitano a considerare i database relazionali, in cui la varietà solitamente si manifesta in maniera ridotta e solo a livello lessicale. Quando si parla di dati con una struttura gerarchica (modello dati documentale), viene adottato il *keyword querying* (interrogazione per parole chiave) [45]. Il processo di risposta a query di keyword inizia con l'identificazione dell'esistenza delle parole chiave all'interno dei documenti senza la necessità di conoscere gli schemi sottostanti. Il problema è che i risultati non considerano l'eterogeneità in termini di nodi, ma assumono che se la parola chiave viene trovata, indipendentemente dal nodo che la contiene, il documento è adeguato e deve essere restituito all'utente. In [7], gli autori hanno introdotto un nuovo meccanismo di interrogazione trasparente di dati eterogenei basato su tecniche di query rewriting [50], su-

perando il problema dell'eterogeneità strutturale dei database documentali. Questo è reso possibile grazie all'utilizzo di un dizionario per stabilire mapping tra gli attributi che rappresentano lo stesso concetto, quindi una query formulata dall'utente viene arricchita grazie alla conoscenza presente nel dizionario e con tutti i possibili percorsi ivi presenti. Su questa linea d'onda, esistono altri approcci che risolvono i problemi relativi all'uso di stessi attributi con nomi o tipi diversi facendo affidamento su mapping e funzioni di transcodifica che riconoscono e collegano la variazione degli attributi e i diversi formati. In [70] vengono indicati i link tra attributi appartenenti a schemi diversi nel momento in cui si vanno a raggruppare record con schemi considerati equivalenti. Similmente, in [28, 6, 30] vengono utilizzati i mapping e la loro semantica viene arricchita utilizzando funzioni di transcodifica, abilitando il supporto anche all'eterogeneità in termini di attributi di diverso tipo che rappresentano lo stesso concetto. Per concludere, è bene sottolineare che: (i) la maggior parte degli approcci presentati considerano il problema dell'eterogeneità all'interno di una collezione per volta e solo per un particolare modello di dati; (ii) ognuno di questi approcci ha dei limiti in termini di supporto alla varietà dei dati e/o all'espressività concessa dal punto di vista di interrogazione ai dati.

Inferenza dello schema. La gestione della varietà nei database NoSQL parte dal riconoscimento degli schemi che adottano i dati. Questa attività viene definita come *schema inference* (inferenza dello schema) e può essere svolta in svariati modi in base al livello di dettaglio che l'utente desidera acquisire, in letteratura sono presenti una moltitudine di lavori che vanno in questa direzione. Ad esempio, in [4, 58] viene estratto lo schema a partire da documenti. Questo gruppo di lavori inizialmente fu introdotto per inferire strutture a partire da documenti semi-strutturati codificati in formato XML. Molte volte schema viene dedotto tramite regole fornite sotto forma di espressioni regolari [24]. C'è da sottolineare che la maggior parte dei documenti sono sono codificati mediante i formati JSON e XML, ma non tutte le soluzioni che lavorano su documenti codificati in XML possono essere

applicate a documenti in formato JSON. Altri lavori, invece, si focalizzano sull'estrazione dello schema a partire da dati RDF [17]. Il problema di questa classe di lavori è che nessuno di questi approcci è progettato per gestire enormi dataset come quelli resi disponibili dalle attuali applicazioni, spesso in formato JSON.

In [70] viene presentato un approccio efficiente per l'estrazione dello schema da database con modelli dati relazionali e co-relazionali (che nel loro caso corrispondono ai modelli key-value, documentale e column-wide). L'approccio proposto si basa su una nozione di schema JSON chiamato scheletro, ovvero una rappresentazione ad albero che descrive le strutture che appaiono frequentemente in una collezione di documenti eterogenei, considerando e rendendo flat gli eventuali sottoschemi innestati. Pertanto, nello scheletro potrebbero essere esclusi percorsi che compaiono in un piccolo insieme di documenti. Un approccio simile è usato in [36], dove però l'attenzione è posta, in particolare, sul modello dati documentale, la raccolta dei differenti schemi distinti di un documento avviene attraverso confronti efficienti con un approccio basato su confronti tra alberi ed è abilitata la modalità di fornire i vari schemi distinti agli sviluppatori. In modo simile, in [27] viene proposto un approccio per l'estrazione di schemi a partire da documenti, dove l'obiettivo è quello di esporre le regole che guidano l'uso di schemi differenti per rappresentare gli stessi dati. Tuttavia, questo approccio si concentra sul fornire più informazioni agli utenti, ma non fornisce alcun meccanismo di query. In [26] viene definita una nuova tecnica per ottenere le varianti dello schema presenti all'interno di una collezione documentale. Questa tecnica ricorre all'utilizzo di mapping per scoprire le diverse variazioni per un dato attributo (ad esempio, tipi di dato o posizione diverse all'interno del documento), per poi costruire una vista integrata multidimensionale dei dati per supportare le query OLAP. I limiti principali di questo approccio sono che si concentra su una raccolta alla volta e che il meccanismo di query rewriting crea una query per ogni variazione dello schema rilevata in tale raccolta.

Nel complesso, tutti i lavori presentati permettono di dedurre le strutture

implicite da dati eterogenei e fornire all'utente un'illustrazione di alto livello per quanto riguarda l'interezza o una parte delle strutture presenti all'interno di collezioni di dati eterogenei. Le tecniche di inferenza dello schema potrebbero aiutare gli utenti a comprendere meglio le diverse strutture sottostanti e ad adottare le misure e le decisioni necessarie durante la fase di progettazione dell'applicazione. Il limite di tale visualizzazione logica è che richiede un processo manuale per costruire le interrogazioni desiderate includendo gli attributi desiderati e tutti i loro possibili percorsi di navigazione. Per cui, in questi approcci l'utente ha la conoscenza sulle strutture dei dati e la gestione dell'eterogeneità è rimandata ad egli stesso. Inoltre, alcuni dei lavori presentati non restituiscono una visione completata dei dati, ma costruiscono gli schemi sopra agli attributi più utilizzati, utilizzando, ad esempio, misure probabilistiche; ciò può comportare un'inesattezza a livello di risultati delle interrogazioni. Per concludere, la maggior parte delle proposte non offre supporto automatico per le valutazioni delle strutture ed è necessario, in caso di cambiamenti, replicare il processo di inferenza, ciò potrebbe influenzare i carichi di lavoro e le applicazioni associati ai dati.

1.3.2 Fusione di record eterogenei e sovrapposti

Per quanto riguarda la risoluzione di conflitti su record sovrapposti con schemi differenti, in letteratura è presente una famiglia di tecniche [52] che ricorrono all'operatore di full-outer join che consente di ottenere un risultato completo a seguito del join di sorgenti con record sovrapposti e, a valle di ciò, applicano funzioni per risolvere i conflitti tra attributi che rappresentano lo stesso concetto. In particolare, in [12] si parla delle seguenti fasi: (i) full outer join; (ii) rimozione dei duplicati; (iii) applicazione della risoluzione conflitti e delle inconsistenze: tra due tuple con gli stessi attributi si sceglie quella che porta maggiori informazioni; (iv) modellazione dei risultati. Il lavoro descritto fa parte di un ambito identificato come *data fusion* (fusione dei dati) [11]. In [5] viene presentata e discussa una panoramica completa di queste tecniche. È stata, inoltre, introdotta una nuova problematica definita *knowledge*

fusion (fusione della conoscenza) [21] che sposta la tematica di data fusion verso triple RDF. La problematica di risolvere conflitti tra record eterogenei e sovrapposti rientra, in aggiunta, nella famiglia di tecniche chiamate di *entity resolution* (risoluzione delle entità) [65], ovvero determinare quando i riferimenti a entità del mondo reale sono equivalenti o non. In [46] gli autori propongono una tecnica di entity resolution che lavora con record eterogenei. Viene applicato un approccio iterativo che come primo passo da effettuare fonde record simili – attraverso una funzione di similarità che lavora con una struttura ad indice e una soglia di tale funzione da considerare per fermare le iterazioni – e, solo successivamente, si parla di comparare le informazioni. In [47] viene descritto un approccio strutturato che considera data fusion ed entity resolution in un processo a tre step che consente di risolvere la problematica in esame: (i) mapping a livello di schema: per risolvere l'eterogeneità dal punto di vista strutturale; (ii) entity resolution: per risolvere l'eterogeneità a livello di valore, raggruppando le descrizioni differenti della stessa entità del mondo reale; (iii) data fusion: per risolvere l'eterogeneità a livello di valore fondendo tali rappresentazioni differenti in una singola rappresentazione. Anche in [8] viene sottolineato che le attività di data fusion e entity resolution non sono da considerare indipendenti, ma facenti parte di un processo comune che mira ad integrare le sorgenti. È bene sottolineare che tra gli approcci menzionati, alcuni applicano tecniche di data/text mining, valutando anche aspetti che riguardano l'utilizzo di grandi volumi di dati [9, 55].

1.3.3 Eterogeneità a livello di modello dati

Multistore e polystore. Come già detto nella Sezione 1.2, i sistemi multistore e polystore sono quei sistemi che supportano più di un modello dati fornendo, rispettivamente, una o più interfacce per l'interrogazione, essi per poter fare ciò necessitano di un livello mediatore. Sistemi come Teradata [72] o HadoopDB [1] propongono di partizionare i dati tra gli archivi. Inoltre consentono alle query di accedere ai dati frammentanti su archivi differenti e di

spostare l'elaborazione e/o i dati tra gli archivi. Tali soluzioni richiedono di memorizzare gli archivi all'interno degli stessi nodi fisici al fine di ridurre il traffico nella rete, sia dal punto di vista dell'esecuzione della query sia della condivisione delle strategie di partizionamento tra i diversi sistemi.

Proposte più recenti garantiscono l'accesso ai dati utilizzando un nuovo linguaggio di query unificato (sistemi multistore). Ad esempio, in [54] resa possibile l'interrogazione di sorgenti relazionali e non, attraverso un linguaggio retro-compatibile con l'SQL con interrogazioni formulate su un database virtuale che integra le sorgenti. In [73], invece, viene proposto un meccanismo estensibile per interrogare sorgenti relazionali con dati memorizzati su database cloud (nello specifico su BigTable[1]), nell'andare a riscrivere le interrogazioni sulle sorgenti il sistema tiene conto dell'espressività limitata del modello a grafo. In [43] ci si concentra sul design fisico e si punta ad ottimizzare le performance e gli spostamenti di dati nella rete in un sistema multistore, con una tendenza di sfruttare caratteristiche proprie di ogni modello dati. Sulla stessa direzione, in [44] gli autori sfruttano diversi database e piattaforme di elaborazione e definiscono un'interfaccia di elaborazione dichiarativa unificata per accedere e interrogare dati eterogenei. È interessante l'approccio utilizzato in [14], dove si ricorre all'uso di ontologie per mediare fonti di dati relazionali e non relazionali. Anche in [41] viene svolta un'integrazione tra i vari modelli dati che punta ad esporre un'interfaccia "matematica" delle sorgenti, utilizzando tecniche di algebra lineare. In [15] vi è un'alta attenzione per quanto riguarda le prestazioni e vengono supportati vari modelli dati: chiave-valore, documentali, relazionali e relazionali nidificati.

Altre proposte supportano più interfacce di interrogazione (sistemi polystore). Ad esempio, Spark SQL [3] supporta sia i linguaggi di interrogazione degli archivi sottostanti che un linguaggio di query unificato. Un'alternativa è presentata in [25], dove viene richiesto all'utente di utilizzare il linguaggio di query adeguato per ciascun modello di dati e viene impiegato un livello

[1] https://cloud.google.com/bigtable

middleware per unire e restituire i risultati finali. Apache Drill [35], invece, è nato come livello di interrogazione per lavorare con un numero considerevole di sorgenti, è progettato principalmente per eseguire scansioni complete dei dati rilevanti anziché, ad esempio, mantenere gli indici. Alla base di un'interrogazione formulata in un certo linguaggio dall'utente, viene creato il piano di esecuzione e vengono interrogate le sorgenti.

Considerando gli approcci sopra esposti emerge un problema: la maggior parte di essi si focalizza principalmente sul risolvere l'eterogeneità in termini di modelli dati (ad esempio, stimando il costo di esecuzione trasferendo i dati da un database all'altro) piuttosto che sulla gestione della varietà dei dati a livello di schema (infatti, essi di solito assumono raccolte di dati con uno schema fisso). A questo scopo si ricorre alle tecniche proposte nella Sezione 1.3.1.

***Multimodel systems* (sistemi multi-modello)**. In relazione ai sistemi multistore e polystore, ci sono i sistemi multimodel: un singolo database che supporta diversi modelli di dati; offre garanzie in termini di governance, gestione e accesso ai dati, ma ha un limite in termine di modelli di dati supportati ed estensione degli stessi. Tali sistemi sostengono l'idea di ridurre il compito di combinare risultati parziali da diversi archivi e quindi suggeriscono di avere un database integrato, che nasconde l'eterogeneità in termini di modelli di dati fornendo un approccio dichiarativo di interrogazione di dati multi-modello. Pertanto, la trasformazione del modello di dati può essere eseguita solo quando è necessaria. Ad esempio OrientDB[2] è il primo DBMS NoSQL open source multimodel che combina la potenza del modello a grafo con più modelli, tra cui documentale e key-value, in un unico database operativo, scalabile e ad alte prestazioni. Si sottolinea che il concetto di sistema multimodel è stato introdotto in precedenza con i sistemi ORDBMS (Object-Relational DBMS) che offrono supporto alla programmazione orientata agli oggetti con database relazionali [33].

[2]https://orientdb.com

Capitolo 2

Metadati per la costruzione del dataspace

In questo capitolo verranno formalizzate e presentate le meta–informazioni estratte dai dati e le possibili tecniche algoritmiche utilizzabili per raccoglierle, rendendo possibile la costruzione del dataspace e, dunque, un'integrazione pay-as-you-go come presentata nella Sezione 1.2.1. Nel dettaglio: (i) nella Sezione 2.1 vengono presentati i concetti di base che consentono di rappresentare i dati; (ii) nella Sezione 2.2 si descrivono e formalizzano i concetti che rappresentano il dataspace, a partire da quelli esposti nella Sezione 2.1; (iii) nella Sezione 2.3 vengono presentate possibili tecniche per estrarre le meta–informazioni formalizzate nelle Sezioni 2.1 e 2.2.

2.1 Concetti di base per rappresentare i dati

Il contesto in esame è quello presentato nella Sezione 1.3 come *multistore*: esso consente interrogare sorgenti che afferiscono a modelli dati differenti attraverso un'unica interfaccia di interrogazione. In questo contesto è, dunque, necessario definire i concetti di base dei database in modo da avere una visione astratta rispetto ai differenti modelli dati. La Figura 2.1 fornisce una descrizione UML dei concetti utilizzati.

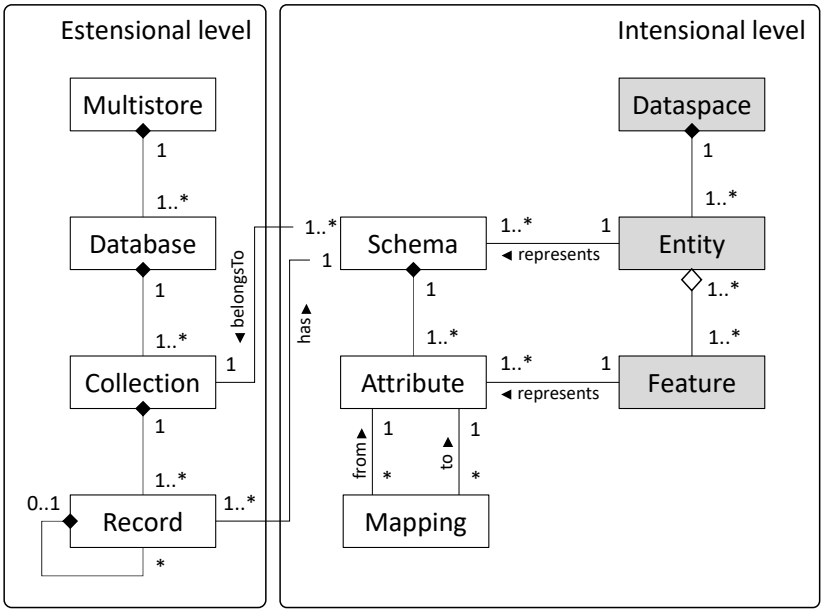

Figura 2.1: Descrizione UML della terminologia.

I concetti sotto esposti si focalizzano su un sistema multistore che supporta i seguenti modelli dati: relazionale, wide-column e documentale. Nonostante ciò, quanto descritto può essere facilmente esteso ad altri modelli dati.

Definizione 1 (Multistore, Database, *Collection* (collezione)). Un multistore è una collezione database; ogni database D è un set di collezioni; ogni collezione C è un set di record.

Il termine *collezione* comprende un qualsiasi contenitore di dati (definito in modo diverso per ciascun modello dati: *tabella* per il modello dati relazionale, *column family* per il modello dati wide-column e *collezione* per il modello dati documentale). Un *record* rappresenta un'istanza di una collezione, coincide esattamente con le tuple appartenenti a un database relazionale, ma nel caso di database documentale e wide-column, rispettivamente, il documento e la riga potrebbero corrispondere a più record a causa dell'innestamento dei dati proprio di questi modelli dati e della progettazione basata sugli aggregati descritta nella Sezione 1.1.6. Per cui questa definizione non

considera documenti e righe nel loro insieme, ma i record disponibili vengono modellati separatamente a ciascun livello di innestamento.

Definizione 2 (Record, *Attribute* (attributo)). Un record $r = \{v_1, \ldots, v_n\}$ è un set di valori, ogni valore v_i è associato a un certo attributo a_i. Sia $r[a_i] = v_i$ dove a_i è un attributo; v_i può essere sia un valore di tipo primitivo (ad esempio numerico o stringa) o un array di record. Un attributo a è definito da un nome e da un tipo (primitivo o array).

Inoltre, dato un attributo a, con $name(a)$ e $type(a)$ ci si riferisce rispettivamente al suo nome e al suo tipo. Se un attributo è annidato all'interno di uno o più attributi di un array, il suo nome include la concatenazione di punti dei nomi di quegli attributi di array. Per semplicità non vengono considerati array di tipi primitivo.

In Figura 2.2 è mostrato un esempio di database documentale. I rettangoli sottolineano la presenza di quattro record: uno contente i dati riguardanti il cliente, uno i dati dell'ordine e due i dati dei dettagli dell'ordine.

Definizione 3 (Schema, *Key* (chiave)). Uno schema S si applica a uno o più record di una collezione ed è definito come un set di attributi primitivi. L'attributo che identifica in modo univoco i record di uno schema S è la *chiave*, definita come $key(S)$. Se i record che fanno riferimento a S sono nidificati, S^{μ} denota la sequenza opzionale di attributi dell'array nella collezione che devono essere "estratti" in modo sequenziale per ottenere i record di S.

Poiché tale nozione di schema si applica ai record piuttosto che all'intera collezione, è possibile trovare più schemi per una determinata collezione a causa della possibile presenza sia di variabilità dello schema che di record annidati. Per semplicità, infatti, dato un record r, il suo schema (indicato con S_r) è l'insieme di attributi direttamente disponibili in r (cioè senza estrarre alcun array). Se un record r' è annidato all'interno di r $S_{r'}$ include anche $key(S_r)$; cioè è necessario per mantenere la relazione tra lo schema di un record nidificato e quello del record padre. Per cui la definizione dello schema

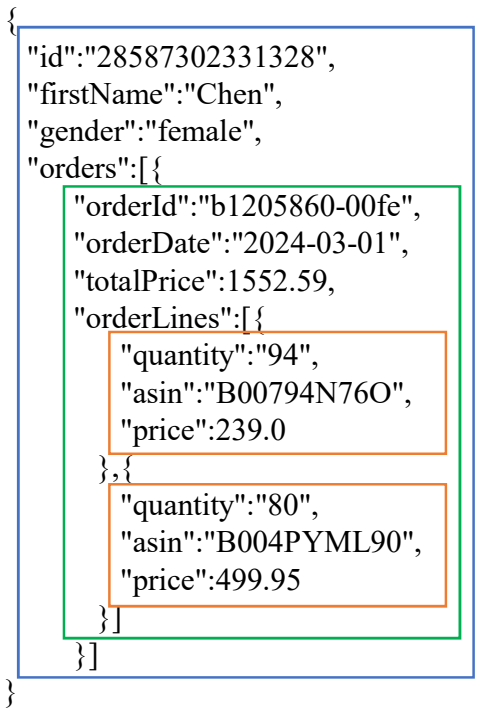

Figura 2.2: Un documento di esempio; sono mostrati quattro *record* nei rettangoli e ogni colore (blu, verde e arancione) corrisponde a uno *schema* diverso.

fornisce una vista dei record nella prima forma normale dal punto di vista della nidificazione dei record, poiché nasconde la denormalizzazione dovuta alla nidificazione stessa ed espone le relazioni tra gli schemi a diversi livelli di nidificazione. Questa definizione dello schema ci consentirà di comprendere le relazioni tra gli schemi nei passaggi seguenti.

Per semplicità, si assume che tutte le chiavi siano semplici (ovvero non composte). Inoltre, è ragionevole presumere che tutti gli schemi (inclusi quelli annidati negli array) abbiano una chiave. $\mathcal{S}$ è l'insieme di tutti gli schemi all'interno del multistore. Si sottolinea che, rispetto alla proprietà schema-less dei database non relazionali, viene presa in considerazione ogni variazione di schema in una collezione. Infatti, se due record differiscono anche per un singolo attributo, vengono estratti due schemi separati: ciascuno con

il proprio insieme di attributi. È importante precisare che un attributo è contenuto in uno e un solo schema.

Il documento di esempio mostrato in Figura 2.2, secondo tale definizione, contiene tre schemi, evidenziati dai diversi colori presenti.

1. $S_{blu} = \{\mathsf{id}, \mathsf{firstName}, \mathsf{gender}\}$.
2. $S_{verde} = \{\mathsf{id}, \mathsf{orders.orderId}, \mathsf{orders.orderDate}, \mathsf{orders.totalprice}\}$.
3. $S_{arancio} = \{\mathsf{orders.orderId}, \mathsf{orders.orderLines.quantity}, \mathsf{orders.orderLines.asin}, \mathsf{orders.orderLines.price}\}$.

Ciascuno di essi ha i seguenti attributi da estrarre per poter arrivare al livello di dettaglio dello schema: $S^{\mu}_{blu} = []$, $S^{\mu}_{verde} = [\mathsf{orders}]$, and $S^{\mu}_{arancio} = [\mathsf{orders}, \mathsf{orders.orderLines}]$.

2.2 Rappresentazione del dataspace

A causa della variabilità e della denormalizzazione dello schema, due o più attributi appartenenti a schemi differenti potrebbero rappresentare la stessa proprietà con nomi e/o tipi diversi. Per risolvere le diverse classi di eterogeneità e modellare l'equivalenza tra i diversi attributi dello spazio dati si utilizzano i *mapping*, come già analizzato nella Sezione 1.3.1.

Definizione 4 (Mapping). Un mapping m è una tripla $m = (a_i, a_j, \varphi)$ che esprime un'equivalenza semantica tra due attributi primitivi a_i e a_j; φ è una funzione di transcodifica per esprimere i valori di a_j nel formato di a_i (se necessario; altrimenti, $\varphi = I()$ dove $I()$ è la funzione identità). L'esistenza di un mapping tra a_i e a_j è indicata con $a_i \equiv a_j$. I mapping sono transitivi, infatti se $a_i \equiv a_j$ e $a_j \equiv a_k$, allora $a_i \equiv a_k$.

Considerando due schemi S_i e S_j, a partire dai mapping è possibile inferire delle relazioni tra gli schemi:

- se $key(S_i) \equiv key(S_j)$, viene inferita una relazione uno-a-uno, rappresentata come $S_i \leftrightarrow S_j$;

- se $a_k \equiv key(S_j) : a_k \in \{S_i \setminus key(S_i)\}$, viene inferita una relazione *molti-a-uno* da S_i a S_j, rappresentata come $S_i \xrightarrow{a_k} S_j$;

- se $a_k \equiv a_l : (a_k, a_l) \in (\{S_i \setminus key(S_i)\}, \{S_j \setminus key(S_j)\})$, non esiste una relazione diretta tra i due schemi.

I mapping, dunque, riconoscono che esiste un'equivalenza semantica tra due attributi di schemi diversi, quindi è necessario affrontarli tutti attraverso un riferimento univoco. Questo è lo scopo delle *feature*.

Definizione 5 (*Feature* (caratteristica)). Una feature rappresenta un singolo attributo o un gruppo di attributi mappati tra loro. Una feature è definita come $f = (a, name, M, \divideontimes)$, dove a è l'attributo *rappresentativo* della feature; $name$ è il nome della feature (possibilmente diverso da $name(a)$); M è l'insieme di mapping che collegano tutti gli attributi della feature al rappresentativo a (le funzioni di transcodifica nel mapping sono tutte dirette verso a); $\divideontimes : (v_i, v_j) \to v_k$ è una funzione associativa e commutativa che risolve il possibile conflitto tra i valori di due attributi qualsiasi (a_i, a_j) appartenenti f e restituisce un unico valore v_k. La funzione $\divideontimes$ può scegliere un valore tra v_i e v_j o calcolare un nuovo valore v_k. $M = \varnothing$ quando un concetto è modellato da un singolo attributo.

Sia $attr(f)$ l'insieme di attributi che caratterizzano f (ovvero l'attributo rappresentativo e tutti quelli derivati dai mapping). Dato un record r, la funzione di risoluzione dei conflitti $\divideontimes$ può essere applicata a $r[a_i]$ e $r[a_j]$ se $\{a_i, a_j\} \subseteq attr(f)$. In [12] vengono forniti vari metodi per la definizione di funzioni di risoluzione dei conflitti, uno tra questi potrebbe essere l'adozione della funzione coalesce che consente di prendere il primo elemento non nullo in un'insieme di dati. È, in aggiunta, importante sottolineare che un attributo è sempre rappresentato da una e una sola feature: per due qualsiasi feature f_i e f_j, allora $attr(f_i) \cap attr(f_j) = \varnothing$. Inoltre, $feat(a)$ viene usato per far riferimento alla feature di un attributo a, $name(f)$ al nome della feature, $rep(f)$ per ottenere l'attributo rappresentativo di f e $rep(a)$

è un'abbreviazione di $rep(feat(a))$. È importante, inoltre, evidenziare che ogni schema contiene al più un attributo per una tale feature.

Analogamente agli attributi, è possibile trovare diversi schemi che rappresentano lo stesso concetto semantico. Per nascondere tale complessità strutturale viene introdotto il concetto di entità.

Definizione 6 (*Entity* (entità)). Un'entità è una rappresentazione di un insieme di schemi nel multistore che modellano semanticamente lo stesso concetto. Un'entità è definita come $E = (name, \mathcal{S}_E)$, dove $\mathcal{S}_E \subseteq \mathcal{S}$ è l'insieme di schemi rappresentato da E. Gli schemi in $\mathcal{S}_E$ devono essere in una relazione uno-a-uno tra loro, cioè $key(S_i) \equiv key(S_j)\ \forall\ (S_i, S_j) \in \mathcal{S}_E$.

È bene sottolineare che una feature può comparire in più entità. Per evidenziare le relazioni tra le entità, esse vengono organizzate in un grafo.

Definizione 7 (*Entity graph* (grafo delle entità)). Il grafo delle entità è un grafo diretto aciclico $G^{\mathcal{E}} = (\mathcal{E}, L^{\mathcal{E}})$ dove $\mathcal{E}$ è l'insieme delle entità nel dataspace e $L^{\mathcal{E}}$ è il set delle relazioni molti-a-uno tra le entità.

Si dice che $E_i \xrightarrow{f} E_j$ se $\exists\ f \in E_i : \forall\ a \in attr(f), a \in S_{E_i}$ è $S_{E_i} \xrightarrow{a} S_{E_j}, (S_{E_i}, S_{E_j}) \in (\mathcal{S}_{E_i}, \mathcal{S}_{E_j})$. In altre parole, c'è una relazione molti-a-uno E_i da E_j su f se $attr(f) \cap key(S_{E_j}) \neq \varnothing\ \forall\ S_{E_j} \in \mathcal{S}_{E_j}$ (ovvero, gli attributi di f sono chiavi negli schemi di E_j) e $attr(f) \cap key(S_{E_i}) = \varnothing\ \forall\ S_{E_i} \in \mathcal{S}_{E_i}$ (cioè gli attributi di f non sono chiavi negli schemi di E_i). Si noti che non si considerano le relazioni molti-a-molti perché gli schemi sono dedotti da implementazioni fisiche, dove le tali associazioni vengono inglobate da più relazioni molti-a-uno.

Infine, il dataspace è definito come segue.

Definizione 8 (Dataspace). Un dataspace $\mathcal{D}$ è un insieme di entità e feature.

2.3 Tecniche per l'estrazione dei metadati per la costruzione del dataspace

Di seguito verranno descritte, per ciascuno dei concetti sopra definiti per la costruzione del dataspace, delle possibili modalità con cui estrarli a partire dai dati ricorrendo a tecniche automatizzabili, ove possibile.

Innanzitutto, i concetti di **database** e **collezione** sono estraibili in modo automatico considerando appunto una connessione verso un database e una collezione definiti.

Per quanto riguarda gli **attributi**, molti degli strumenti big data offrono connettori ai principali database e la possibilità di accedere ai dati in forma strutturata e dunque ottenere informazioni basilari riguardo gli attributi risulta molto semplice e automatizzabile (ad esempio, mediante Spark SQL [3]). A partire da tali informazioni è necessario estrarre e comprimere gli **schemi** in modo da ottenere un insieme distinto di schemi. Queste operazioni sono facilmente parallelizzabili, ma hanno un costo elevato perché si tratta in ogni caso di dover scansionare tutti i record di una collezione per ottenere una panoramica precisa e completa degli schemi presenti. Potrebbe essere interessante svolgere un'estrazione degli schemi solo su un campionamento dei record e non sulla totalità di essi. L'estrazione della **chiave** di uno schema potrebbe essere svolta in modo automatico e attraverso delle euristiche come descritto in [39] e con i limiti in termini di accuratezza e precisione di tali approcci.

Per quanto riguarda l'estrazione dei **mapping**, essa può avvenire in modo automatico ricorrendo ad algoritmi di *schema matching* (corrispondenza dello schema) [10] o incorporando strumenti esistenti (ad esempio Coma 3.0 [48]). Questa operazione, senza il supporto dell'utente, potrebbe non essere completa. Infatti, si tratta delle informazioni fondamentali che l'utente dovrebbe fornire per avere un dataspace che sia preciso e abbia una conoscenza completa dei dati.

L'estrazione delle **feature** a partire dai mapping risulta, effettivamente,

automatizzabile. Basti considerare l'insieme dei mapping tra gli attributi come un grafo non diretto, dove gli attributi sono i nodi e i mapping gli archi, una feature corrisponde a una *componente connessa* di tale grafo. Una componente connessa è infatti un sotto-grafo in cui ogni coppia di nodi è connessa tra loro da un percorso, per cui ogni nodo appartenente a una componente connessa può raggiungere un qualsiasi nodo appartenente alla stessa. Per semplicità, da definizione, l'attributo rappresentativo è quell'attributo verso il quale sono direzionati tutti i mapping, per cui caso basterebbe banalmente effettuare una semplice ricerca sintattica tra i mapping direzionati verso uno stesso attributo, ma in tal caso si rischierebbe di perdere componenti connesse dall'utente in modo errato. Per cui, per completezza, si potrebbe utilizzare l'algoritmo di estrazione componenti connesse segnalando all'utente eventuali inconsistenze rispetto alla definizione di feature.

Un'**entità**, a questo punto, è ottenibile in modo automatico: non è altro che l'insieme degli schemi per cui esiste una feature comune che è chiave primaria all'interno degli schemi, per cui esiste una relazione uno-a-uno tra loro.

Infine, estrarre l'**entity graph** significa analizzare le feature e le entità ottenute e costruire un grafo a partire dal fatto che per ogni combinazione delle entità estratte esiste un link se e solo se: (i) le entità sono diverse; (ii) esiste una feature f che in un'entità svolge il ruolo di chiave primaria e nell'altra no: in un'entità E_1 per ogni schema S_1 l'attributo appartenente alla feature f non svolge il ruolo di chiave e, invece, nell'entità E_2 per ogni schema svolge tale ruolo. Per finire, è necessario verificare che il grafo ottenuto sia aciclico, ovvero che non esistano cicli diretti: scelto qualsiasi vertice del grafo non deve essere possibile tornare ad esso percorrendo un qualsiasi percorso.

Per concludere, si ricorda, come già detto nella Sezione 1.2.1, che nell'approccio pay-as-you-go (adottato dal dataspace) la conoscenza dell'utente è ritenuta fondamentale. Per questo motivo, in qualsiasi momento è fornita la possibilità di far intervenire l'utente per modificare e applicare la sua conoscenza a quanto ricavato mediante le automazioni sopra esposte.

Capitolo 3

Stack algoritmico per interrogare il dataspace

In questo capitolo, partendo dalle informazioni definite nel Capitolo 2, viene descritto il processo di traduzione di una query formulata sul dataspace in un piano di esecuzione che a partire dalle singole sorgenti riconcilia i risultati parziali ottenuti seguendo le informazioni presenti nel dataspace. Nello specifico: (i) nella Sezione 3.1 vengono formalizzate le modalità di interrogazione del dataspace; (ii) nella Sezione 3.2 si descrivono le proprietà sintattiche del piano di esecuzione che si intende costruire e si formalizza l'operatore di merge definito per fondere i record; (iii) nella Sezione 3.3 si presentano gli algoritmi per la costruzione, vera e propria, del piano di esecuzione a partire un'interrogazione; (iv) infine, nella Sezione 3.4, vengono descritte le tecniche di ottimizzazione applicate nella costruzione del piano di esecuzione.

3.1 Interrogare il dataspace

L'aspetto fondamentale da considerare è come interrogare il dataspace, quindi come sfruttare i metadati raccolti e la conoscenza fornita dall'utente per tradurre una query nel dataspace in una serie di query nelle sorgenti, come eseguire l'interrogazione nelle stesse e come riconciliare i risultati. Il contesto

principale che si va a considerare in questo elaborato di tesi è quello analitico, infatti lo scopo principale dell'integrazione lightweight, fornita dal concetto di dataspace, è abilitare un'analisi di alto livello e riassuntiva sui dati di tutte le sorgenti riconciliati all'interno del dataspace stesso. L'espressività considerata è quella delle interrogazioni GPSJ. Le **query GPSJ** (*Generalized Projection, Selection, Join*) [32] sono la classe di query più comune in applicazioni OLAP, sono la combinazione di tre operatori SQL di base: join, predicati di selezione e aggregazioni. Un'interrogazione GPSJ risulta composta dai seguenti elementi.

1. *Proiezioni*: sono clausole che riguardano le misure che si andranno ad aggregare nell'interrogazione.

2. *Selezioni*: un insieme di clausole booleane sul valore degli attributi, può essere vuoto.

3. *Aggregazioni*: si tratta dei group-by-set da considerare nell'interrogazione, se non presente i dati vengono analizzati al massimo livello di dettaglio.

Per considerare una query valida almeno uno tra l'insieme di aggregazioni e proiezioni deve essere non vuoto.

Definizione 9 (Query). Dato un dataspace $\mathcal{D}$, una query è definita come $q = (q_\pi, q_\gamma, q_\sigma)$, dove: (i) $q_\pi \subseteq \mathcal{D}$ specifica l'insieme opzionale di feature da proiettare; (ii) q_γ specifica l'insieme opzionale di aggregazioni come un'insieme di coppie (f, op), dove $f \in \mathcal{D}$ e op è una funzione di aggregazione (ad esempio, $max()$); (iii) q_σ è un insieme facoltativo di predicati di selezione congiuntivi ($\wedge$) sotto forma di triple (f, ω, v), dove $f \in \mathcal{D}, \omega \in \{=; >; <; \neq; \geq; \leq\}$ e v è il valore. Chiaramente, almeno uno tra gli insiemi q_π e q_γ deve essere non vuoto.

Tabella 3.1: Operatori NRA.

Operatore	Descrizione
CA_{col}	Indica l'accesso ai record della collezione *col*.
$\mu_a(C)$	Denota l'*unnesting* (estrazione) di un attributo array a in una collezione C.
$\sigma_x(C)$	Indica un'operazione di selezione su una collezione C, dove $x = \bigwedge_T$ è una congiunzione di predicati di selezione; ogni predicato di selezione $t \in T$ è nella forma (a, ω, v), dove a è un attributo primitivo, $\omega \in \{=; >; <; \neq; \geq; \leq\}$ e v è un valore.
$\pi_Y(C)$	Indica un'operazione di proiezione su una collezione C, dove Y è un insieme di predicati di proiezione; ogni predicato di proiezione $y \in Y$ è nella forma $y = \bigvee_A / f$ dove A è un insieme di attributi primitivi (di cui vengono presi i primi valori non nulli) e $/f$ indica che l'attributo risultante prende il nome dalla feature f. È $attr(f) \supseteq A$.
$\gamma_{(F,Z)}(C)$	Indica un'operazione di aggregazione su una collezione C, dove F è il group-by set (ovvere un insieme di feature) e Z è l'insieme di aggregazioni; ogni aggregazione è nella forma (f, op) dove f è una feature e op è una funzione di aggregazione.
$(C_1) \sqcup_{(a_i,a_j)} (C_2)$	Indica un'operazione di merge tra le collezioni C_1 e C_2 basato sull'equivalenza $a_i = a_j$, con $(a_i, a_j) \in (C_1, C_2)$. Guarda la definizione 10.

3.2 NRA e operatore di merge

Per definire il piano di esecuzione della query si ricorre all'algebra relazionale annidata (NRA). Nella Tabella 3.1 vengono descritti brevemente tutti gli operatori. Rispetto all'algebra tradizionale viene introdotto un nuovo operatore chiamato *merge* ($\sqcup$): una versione per tale scenario dell'operatore *full outerjoin-merge* introdotto in [52] che considera quanto discusso nella Sezione 1.3.2. L'obiettivo di tale operatore è quello di sostituire l'operatore di join ($\bowtie$) andando a considerare la sovrapposizioni tra gli schemi a livello intensionale ed estensionale. Questo viene fatto con lo scopo principale di mantenere più informazione possibile nel momento in cui avviene il join di record appartenenti a schemi diversi. Le condizioni da soddisfare sono, infatti, le seguenti: (i) evitare la perdita di record; (ii) risolvere i mapping fornendo un output in termini di feauture invece che di attributi; (iii) risolvere i conflitti, ove necessario. Infatti, l'operatore considera una possibile sovrapposizione parziale di record appartenenti allo stesso concetto semantico/entità (ad esempio il Cliente, sia in termini di istanza (ad esempio un'istanza dello stesso cliente

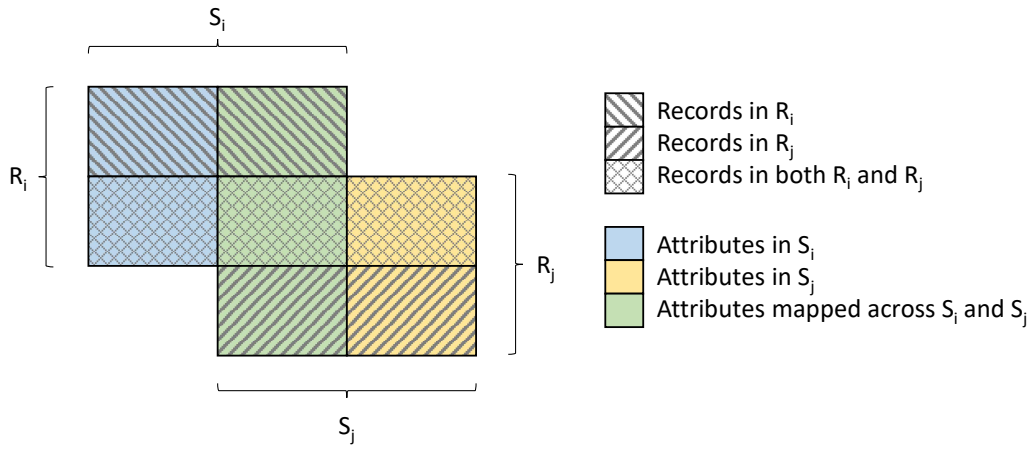

Figura 3.1: Rappresentazione grafica dell'operatore di merge.

si può ripetere in schemi differenti) che in termini di schema (ad esempio il nome del cliente può essere un attributo di due schemi diversi). La stessa considerazione viene fatta anche quando si uniscono record di entità diverse (ad esempio Cliente e Ordine): i record possono essere parzialmente sovrapposti (ad esempio, un cliente potrebbe non avere ordini o un ordine non avere il cliente) e anche i loro schemi (ad esempio il nome del cliente potrebbe essere utilizzato anche nello schema dell'ordine). Un esempio di questo scenario è mostrato in Figura 3.1, dove sono mostrati due schemi sovrapposti S_i e S_j con i rispettivi insiemi di record sovrapposti R_i e R_j. L'area verde verticale rappresenta l'intersezione degli schemi e l'area centrale orizzontale, invece, rappresenta l'intersezione tra i record dettata dalla condizione di join.

Definizione 10 (Operatore di merge). Siano R_i e R_j l'insieme di record di due schemi S_i e S_j e considerando $(a_k, a_l) \in (S_i, S_j)$ tale che $a_k \equiv a_l$, ovvero $\exists f : \{a_k, a_l\} \subseteq attr(f)$. Il merge dei due schemi $S_i \sqcup_f S_j$ produce un insieme di record R_{ij} con schema $S_{ij} = S_i^* \cup S_j^* \cup S_{ij}^{\cap}$ tale per cui:

- $S_i^* = \{a \in S_i : \nexists\ a' \in S_j, a \equiv a'\}$
- $S_j^* = \{a' \in S_j : \nexists\ a \in S_i, a \equiv a'\}$
- $S_{ij}^{\cap} = \{rep(a)\ \forall\ (a, a') \in (S_i, S_j) : a \equiv a'\}$

R_{ij} si traduce in un full-outer join tra R_i e R_j dove le coppie di attributi collegate da un mapping vengono unite tramite la funzione $\mathbb{M}$. In particolare, dato un record $r \in R_{ij}$ ottenuto facendo un join tra $s \in R_i$ e $t \in R_j$ (ovvero $s[a_i] = t[a_j]$), si ottiene $r[rep(a)] = \mathbb{M}(s[a], t[a']) \; \forall \; (a, a') \in (S_i, S_j) : a \equiv a'$.

3.3 Il piano di esecuzione

Costruire il piano di esecuzione richiede come prima cosa di identificare le entità a cui bisogna accedere per eseguire l'interrogazione stessa, potrebbero non essere solo le entità a cui fanno riferimento le feature richieste dalla query: esse potrebbero non essere direttamente collegate all'interno del grafo delle entità.

Definizione 11 (*Query graph* (grafo della query)). Il query graph $G_q^{\mathcal{E}}$ è un sotto-grafo di $G^{\mathcal{E}}$ (cioè $G_q^{\mathcal{E}} = (\mathcal{E}_q \subseteq \mathcal{E}, L_q^{\mathcal{E}} \subseteq L^{\mathcal{E}})$) tale per cui:

(i) $G_q^{\mathcal{E}}$ è minimamente connesso;

(ii) $\mathcal{E}_q \supseteq attr(q)$;

(iii) $\exists \; E^* \subseteq \mathcal{E}_q : E^* \supseteq q_\gamma, E^* \Rightarrow E^{'} \; \forall \; E' \in \mathcal{E}_q$.

La condizione espressa dalla Voce (i) assicura che non viene fatto l'accesso a entità non necessarie per l'esecuzione dell'interrogazione. La condizione della Voce (ii) assicura che tutti gli attributi appartenenti alle feature coinvolte nella query sono coperti dalle entità presenti in $\mathcal{E}_q$. La condizione evidenziata nella Voce (iii) implica la *conformità* della query q con la semantica GPSJ, cioè l'esistenza di un'entità che rappresenta gli eventi al massimo livello di dettaglio (ovvero, esiste un percorso diretto da E^* a ogni altra entità in $\mathcal{E}_q$). Potrebbero esistere molti sotto-grafi per una data query poiché potrebbero esistere molti percorsi *-a-uno*, ciascuno associato a una semantica diversa (ad esempio un'entità di vendite potrebbe essere associata a un'entità di date attraverso i mapping sia delle *date di vendita* che delle *date di spedizione*). In

Algoritmo 1 Definizione del query graph $G_q^{\mathcal{E}}$ per la query q.

Input $q = (q_\pi, q_\gamma, q_\sigma)$: la query; $G^{\mathcal{E}}$: l'entity graph.
Output $Set(G_q^{\mathcal{E}})$: l'insieme dei query graph per la query q.

```
1:  Es ← entities(q)                          ▷ ottieni l'insieme delle entità direttamente coinvolte in q
2:  if Es.size() == 1 then
3:      return new Graph(Es)                  ▷ restituisci il grafo formato da Es
4:  else
5:      Rs_{G^E} = roots(G^E)                 ▷ estrai i nodi root dell'entity graph
6:      VRs = Rs_{G^E}.filter(x => x.canReach(Es))   ▷ cerca i nodi root validi
7:      R = new Set()
8:      for all r ∈ VRs do
9:          AG = pruneLeaves(G^E, Es)         ▷ rimuovi foglie da G^E finchè esse non sono tutte in Es
10:         AG = pruneRoot(r, AG)             ▷ pota AG dall'alto
11:         R.addAll(getCorrectQueryGraphs(AG, Es))   ▷ aggiungi i possibili query graph in AG a R
12:     return min(R)                         ▷ restituisci solo i query graph con lunghezza minima
```

questo caso è richiesta un'interazione dell'utente per identificare il sotto-grafo adeguato.

Il query graph $G_q^{\mathcal{E}}$ è, dunque, il punto di partenza per definire il piano di esecuzione della query q (detto P_q).

Definizione 12 (*Query plan* (piano della query)). Un query plan è un albero NRA dove le foglie denotano l'accesso a una collezione (CA) e la radice è o un'aggregazione (γ) o una proiezione (π).

3.3.1 Costruzione del query graph

Per quanto riguarda l'estrazione algoritmica dei possibili query graph a partire da l'interrogazione in esame q e dall'entity graph $G^{\mathcal{E}}$, una delle possibili soluzioni potrebbe essere quella mostrata nel Algoritmo 1. L'algoritmo sostanzialmente parte dall'identificare le entità direttamente coinvolte nell'interrogazione (Es): tutte le entità a cui appartengono le feature richieste nella query (ovvero, le feature presenti in q_π, q_γ e q_σ). Se la query coinvolge direttamente una sola entità l'algoritmo ritornerà un grafo composto dalla sola entità coinvolta (Linea 3), altrimenti si procede con l'identificare i possibili sotto-grafi minimi che contengano almeno le entità coinvolte nella query (da Linea 5 a Linea 12). Per fare questo, vengono innanzitutto iden-

Algoritmo 2 getCorrectQueryGraphs(G, xs).

Input $G = (V, E)$: il grafo di partenza; xs: le entità coinvolte sicuramente nell'interrogazione.
Output $Set(G_q^{\mathcal{E}})$: l'insieme dei query graph ricavabili da G, assicurando che esso rimanga connesso e contenga almeno xs.

```
1: if overLinkedEdges(G).isEmpty then        ▷ se in G ∃ al più un arco per ogni coppia di nodi
2:     return new Set(G)                      ▷ restituisci il set composto solo da G
3: else
4:     R = new Set()
5:     edges = first(overLinkedEdges(G))      ▷ ottieni il primo insieme di archi presente in overLinkedEdges(G)
6:     for all e ∈ edges do
7:         NG = G \ (edges \ e)               ▷ rimuovi da G gli archi che sono in edges \ e
8:         NG = pruneLeaves(NG, xs)           ▷ rimuovi da NG le foglie che non sono in xs
9:         R.addAll(getCorrectQueryGraphs(NG, xs))
10:    return R
```

tificate le *root* (radici) dell'entity graph che sono quei nodi per cui esistono solo archi in uscita (Linea 5) e tra questo insieme vengono trovati solo quelli validi (Linea 6), ovvero quelli che possono raggiungere, anche non direttamente, tutte le entità Es. Si itera per ciascun elemento r di questo insieme (da Linea 8 a Linea 11) e ogni volta: (i) si trova, in modo ricorsivo, il grafo ridotto AG che consiste nella porzione del grafo $G^{\mathcal{E}}$ che ha come foglie solo elementi appartenenti a Es (Linea 9); (ii) AG viene potato dalla radice ricorsivamente, finchè i vicini validi della radice sono uno e le radici non sono in Es (Linea 10); (iii) nella Linea 11, a partire da AG, vengono estratti (e aggiunti al risultato finale) tutti i possibili sotto-grafi validi per cui per ogni coppia di nodi esiste al più un arco e vengono potate eventuali foglie non in Es (richiamando quanto espresso in Linea 9), più dettagli sono esposti nell'Algoritmo 2. Infine, vengono estratti i grafi presenti nel risultato con lunghezza minima, ogni arco è considerato con peso unitario, e tale insieme viene restituito dall'algoritmo (Linea 12).

Nel caso in cui non venga trovata nessuna radice dell'entity graph che soddisfi l'interrogazione, il risultato restituito dall'algoritmo sarà un insieme vuoto e, per cui, l'interrogazione non risulterà risolvibile.

Algoritmo 3 Definizione del query plan P_q per la query q.

Input $q = (q_\pi, q_\gamma, q_\sigma)$: una query; $G_q^{\mathcal{E}} = (\mathcal{E}_q, L_q^{\mathcal{E}})$: il query graph.
Output P_q: il query plan di q.

1: $entityList \leftarrow sortEntities(G_q^{\mathcal{E}}, q_\sigma)$ ▷ applica l'euristica di selettività minima per ordinare le entità
2: $E \leftarrow pop(entityList)$ ▷ $pop()$ estrae il primo elemento della lista
3: $P_q \leftarrow createEntityPlan(q, E)$
4: $mergedEntities \leftarrow E$
5: **while** $entityList \neq \varnothing$ **do**
6: $E = pop(entityList)$
7: $rightPlan \leftarrow createEntityPlan(q, E)$
8: $l \leftarrow getLink(G_q^{\mathcal{E}}, mergedEntities, E)$
9: $P_q \leftarrow extendPlansWithMergeOp(P_q, rightPlan, feat(l))$
10: $mergedEntities \leftarrow mergedEntities \cup E$
11: **if** $q_\gamma \neq \varnothing$ **then** ▷ reggruppa q_γ per q_π
12: $predicate \leftarrow (q_\pi, q_\gamma)$
13: $P_q \leftarrow extendPlanWithUnaryOp(P_q, \gamma, predicate)$
14: **else** ▷ proietta su q_π
15: $predicate \leftarrow q_\pi$
16: $P_q \leftarrow extendPlanWithUnaryOp(P_q, \pi, predicate)$
17: **return** P_q

3.3.2 Costruzione di un query plan

La costruzione del piano d'esecuzione di un'interrogazione parte dalla riconciliazione dei record appartenenti alla stessa entità e, successivamente, si procede all'unione di essi con record appartenenti alle altre entità. Dunque, un piano di esecuzione non è altro che una composizione, attraverso l'operatore di merge $\sqcup$, di uno o più *entity plan* (piano di esecuzione a livello di entità). Tale approccio risulta già utilizzato in letteratura [52, 12, 31].

Il query plan P_q è organizzato come un albero *left-deep* (profondo a sinistra) di entity plan, dove l'ordine delle operazioni di merge è ottimizzato attraverso l'*euristica di selettività minima* [63]. Tale euristica restituisce, per l'appunto, un albero letf-deep cercando di mantenere le relazioni intermedie il più piccole possibili e, quindi, limitare il più possibile lo spostamento di dati non necessari, in un determinato punto della computazione.

L'Algoritmo 3 crea incrementalmente Pq. Le entità che sono state identificate dal query graph ($\mathcal{E}_q$) vengono ordinate basandosi sull'euristica utilizzata (Linea 3) e, successivamente, in modo progressivo vengono fusi i vari

Algoritmo 4 extendPlanWithUnaryOp

Input *plan*: il piano da estendere; *type*: il tipo di operazione; *predicate*: il predicato dell'operazione.
Output *extPlan*: il piano esteso.

```
1: extPlan ← new PlanNode()                ▷ inizializza il piano esteso come nuovo nodo
2: extPlan.type ← type
3: extPlan.predicate ← predicate
4: extPlan.child ← plan                    ▷ imposta il vecchio piano come figlio del piano esteso
5: return extPlan
```

Algoritmo 5 extendPlansWithMergeOp

Input $leftPlan$, $rightPlan$: i piani da fondere; f: la feature su cui l'operazione di merge si basa.
Output *extPlan*: il piano esteso.

```
1: extPlan ← new PlanNode()                ▷ inizializza il piano esteso come nuovo nodo
2: extPlan.type ← ⊔
3: extPlan.predicate ← f
4: extPlan.leftChild ← leftPlan
5: extPlan.rightChild ← rightPlan
6: return extPlan
```

entity plan (da Linea 5 a Linea 10). La funzione *createEntityPlan* (Linee 3 e 7) è definita nell'Algoritmo 6. La funzione *getLink* (Linea 8) recupera dal query graph $G_q^{\mathcal{E}}$ il collegamento l che connette l'entità corrente E con quelle fuse precedentemente, ovvero *mergedEntities*; questo è necessario per identicare la feature su cui basarsi nel momento in cui viene applicato l'operatore di merge, indicata con $feat(l)$ (Linea 9). Una volta che ciascun entity plan è stato fuso in un singolo albero NRA, gli opearatori finali (da aggiungere come radice del query plan) dipendono dalla query formulata. Se la query specifica un'aggregazione ($q_\gamma \neq \varnothing$), un'operazione di aggregazione viene aggiunta come radice del query plan (da Linea 11 a Linea 13); altrimenti, viene aggiunta una semplice proiezione (da Linea 14 a Linea 16). Le due funzioni *extendPlanWithUnaryOp* e *extendPlansWithMergeOp* (Linee 9, 13 e 16 dell'Algoritmo 3) sono descritte, rispettivamente, dagli Algoritmi 4 e 5. Il primo semplicemente estende il piano esistente con una nuova operazione unaria da aggiungere come radice dell'albero NRA (gli operatori unari supportati sono quelli elencati nella Tabella 3.1, fatta eccezione per l'unione binaria). Il secondo prende due piani (a livello di collezione o di entità) e crea un nuovo piano che unisce quelli esistenti tramite un'operazione di fusione

Algoritmo 6 createEntityPlan

Input $q = (q_\pi, q_\gamma, q_\sigma)$: una query; E: un'entità; $\mathcal{S}_E^q$.
Output P_E: l' entity plan per E.
1: $\mathcal{S}_E^q = \bigcup_{S \in \mathcal{S}_E} S \cap feat_E(q_\sigma) \neq \varnothing$ ▷ gli schemi di E a cui è necessario accedere
2: $collections \leftarrow \bigcup_{S \in \mathcal{S}_E^q} col(S)$
3: $collectionList \leftarrow sortCollection(collections, q_\sigma)$ ▷ applica euristica di selettività minima
4: $col \leftarrow pop(collectionList)$
5: $P_E \leftarrow createCollectionPlan(q, \mathcal{S}_{col}^q)$
6: $f \leftarrow feat(S_{col}^q)$ ▷ ottieni la feature f che rappresenta le chiavi degli schemi
7: **while** $collectionList \neq \varnothing$ **do**
8: $col \leftarrow pop(collectionList)$
9: $right \leftarrow createCollectionPlan(q, \mathcal{S}_{col}^q)$
10: $P_E \leftarrow extendPlanWithMergeOp(P_E, right, f)$
11: **return** P_E

sulla feature fornita. Per semplicità al suo interno non viene presentato il fatto che nel caso in cui la query abbia almeno una selezione (cioè $q_\sigma \neq \emptyset$), dopo aver effettuato la fusione tra due entity plan, verrà applicata una selezione sulla non nullità di ogni feature presente in q_σ (di conseguenza, questa feature dovrà essere proiettata fino a quando sarà necessaria l'applicazione di tale selezione). Questo consente di evitare di restituire record inutili e restituire un risultato non corretto della query richiesta ma, di contro, comporta che in caso di filtro "*is null*" il comportamento non sarà quello atteso. Per questo motivo la soluzione proposta non funziona in presenza di selezioni del tipo $(f, \neq, null)$ con f una feature qualsiasi.

3.3.3 Costruzione di un entity plan

Similmente al query plan, un entity plan è un albero left-deep dove le foglie sono *collection plan* (piani di esecuzione a livello di collezione). L'obiettivo dell'entity plan è di fondere i record ottenuti dagli schemi che appartengono a una certa entità. Tuttavia, le attuali tecnologie NoSQL non consentono l'accesso ai record della raccolta in base a un determinato schema (le collezione sono schema-less da definizione); questa è la motivazione per cui anziché definire piani a livelli di schema vengono definiti piani a livello di collezione. L'ordine delle operazioni di merge tra i collection plan è determinato

Algoritmo 7 createCollectionPlan

Input $q = (q_\pi, q_\gamma, q_\sigma)$: un query; $\mathcal{S}^q_{col}$: l'insieme degli schemi di una certa collezione *col* coinvolti in q.
Output P_{col}: il collection plan di *col*.

1: $P_{col} \leftarrow$ **new** $PlanNode()$
2: $P_{col} \leftarrow extendPlanWithUnaryOp(P_{col}, \mathsf{CA}, col)$ ▷ inizia con l'accesso alla collezione
3: $predicateSet \leftarrow \varnothing$
4: **for all** $S \in \mathcal{S}^q_{col}$ **do**
5: **for** $i = 1$ **to** $|S^\mu|$ **step** 1 **do**
6: **if** $predicateSet \cap S^\mu[i] = \varnothing$ **then** ▷ controllo dei duplicati
7: $predicateSet \leftarrow predicateSet \cup S^\mu[i]$
8: $P_{col} \leftarrow extendPlanWithUnaryOp(P_{col}, \mu, S^\mu[i])$ ▷ aggiungi le op di unnesting opzionali
9: $predicateSet \leftarrow \varnothing$
10: **for all** $f \in q_\sigma$ **do**
11: $A \leftarrow \mathcal{S}^q_{col} \cap f$
12: $predicateSet \leftarrow predicateSet \cup (\bigvee_{a \in A}(\varphi(a), \omega, v))$
13: **if** $predicateSet \neq \varnothing$ **then**
14: $P_{col} \leftarrow extendPlanWithUnaryOp(P_{col}, \sigma, \bigwedge_{p \in predicateSet})$ ▷ aggiungi l'op di selezione
15: $predicateSet \leftarrow \emptyset$
16: **for all** $f \in F_\pi = \{feat(q_\pi) \cup feat(q_\gamma) \cup F_\sqcup\}$ **do** ▷ l'insieme di feature de proiettare
17: $A \leftarrow \mathcal{S}^q_{col} \cap attr(f)$
18: **if** $A \neq \varnothing$ **then**
19: $predicateSet \leftarrow (\bigvee_{a \in A} \varphi(a))$ / $rep(f)$
20: $P_{col} \leftarrow extendPlanWithUnaryOp(P_{col}, \pi, predicateSet)$ ▷ aggiungi l'operazione di selezione
21: **return** P_{col}

adottando la stessa euristica utilizzata per gli entity plan.

L'Algoritmo 6 genera in modo incrementale l'entity plan P_E per una data entità E. Sia $\mathcal{S}^q_E = \bigcup_{S \in \mathcal{S}_E} S \cap feat_E(q_\sigma) \neq \varnothing$ l'insieme degli schemi che afferiscono ad E ai quali è necessario accedere: in particolare, possiamo escludere gli schemi che non contengono un attributo per le feature di q_σ che sono esclusivamente in E, perché il filtro eliminerebbe automaticamente ogni record. Per definire i collection plan, identifichiamo il set distinto di collezioni a cui è necessario accedere nella Linea 2, essi poi vengono ordinati sulla base dell'euristica adottata nella Linea 3. L'entity plan è costruito come come un albero left-deep definendo prima il collection plan della prima collezione (Linee 4 e 5) e poi, progressivamente, fondendo i collection plan delle collezioni successive (da Linea 7 a Linea 10); la funzione *createCollectionPlan* è definita nell'Algoritmo 7.

3.3.4 Costruzione di un collection plan

Infine, ogni collection plan descrive la sequenza delle operazioni NRA unarie per raccogliere i record di una determinata entità E in una collezione *col*. Poiché la collezione può contenere più schemi appartenenti alla stessa entità, il collection plan prende in considerazione le variazioni di schema intrinseche: dato $\mathcal{S}_E^q$ l'insieme degli schemi appartenenti a E che devono essere acceduti, ci riferiamo a $\mathcal{S}_{col}^q \subseteq \mathcal{S}_E^q$ come sottoinsieme di schemi da considerare per *col*.

L'Algoritmo 7 genera il collection plan P_{col} prendendo in considerazione la varietà in termini di schemi all'interno *col*. Il collection plan è definito come una sequenza ordinata di operazioni NRA unarie, nel seguente ordine: operazioni di unsesting opzionali, un'operazione di selezione facoltativa e un'operazione di proiezione finale. Si sottolinea che tale ordine è il più ovvio, poiché (i) è necessario estrarre i record annidati prima di poter accedere ad essi e (ii) di solito è una buona pratica applicare i predicati di selezione il prima possibile [69]. Il piano P_{col} è costruito in modo bottom-up come segue.

- La prima operazione è l'accesso alla collezione CA a *col* (Linea 2).

- Gli operatori di unnesting vengono eventualmente aggiunti (da Linea 3 a Linea 8) nel caso in cui uno o più schemi siano annidati all'interno di array ($|S^\mu| \geq 1$). Viene eseguito nella Linea 6 un semplice controllo per i duplicati nel caso $\exists\ (S_1, S_2) \in \mathcal{S}_{col}^q : S_1^\mu \cap S_2^\mu \neq \varnothing$; le operazioni di unnesting vengono aggiunte a P_{col} nella Linea 8.

- L'operazione di selezione facoltativa viene costruita da Linea 9 a Linea 14. Per ogni feature per cui è necessaria una selezione, costruiamo una disgiunzione di predicato che considera ogni variazione dello schema di f (Linea 12), ovvero in una congiunzione di predicati di selezione sugli attributi appartenenti alla feature. Successivamente, il predicato di selezione finale è la congiunzione ($\wedge$) dei predicati costruiti per ciascuna feature (Linea 14).

- Infine, l'operazione di proiezione viene costruita da Linea 15 a Linea 20. Sia $F_\pi = \{feat(q_\pi) \cup feat(q_\gamma) \cup F_{\sqcup}\}$ (usato in Linea 10) l'insieme delle feature da proiettare, dove $F_{\sqcup} = feat(l)\ \forall\ l \in L_q^{\mathcal{E}}$ è l'insieme delle feature i cui attributi sono necessari per l'operazione di fusione. Per ogni feature $f \in F_\pi$ rappresentante gli attributi in $\mathcal{S}_{col}^q$ viene proiettato un singolo attributo (prende il nome di $rep(f)$) che contiene l'unico valore non nullo tra le sue variazioni di schema (semplificato in Linea 19 come una disgiunzione su ciascun $a \in A$). È importante sottolineare che, in questa fase, vengono anche applicate le funzioni di transcodifica φ per confrontare in modo consistente i valori dei record nelle operazioni di merge che seguiranno.

Per semplicità nell'algoritmo non viene espressa la casistica in cui esita uno schema allo stesso livello di dettaglio di quello considerato (cioè nella stessa collezione e con lo stesso insieme di attributi S^μ) appartenente ad un'altra entità, diversa da quella considerata. In questo caso deve essere applicato un filtro di non nullità sulla chiave primaria dello schema.

3.4 Ottimizzazioni applicate nella costruzione del piano di esecuzione

La natura distribuita e multi-modello dell'ambiente multistore e il contesto di eterogeneità considerata ci aprono verso una serie di opportunità in termini di ottimizzazione del query plan. L'approccio algoritmico sopra definito attua, intrinsecamente, una serie di tecniche di ottimizzazione per la costruzione del piano di esecuzione.

- ***Push-down* (spinta verso il basso) dei predicati.** Si tratta di una delle tecniche di ottimizzazione più basilari, consiste nell'applicare i predicati di selezione più vicino possibile alle sorgenti. Esso viene applicato nella costruzione del collection plan (da Linea 9 a Linea 14 dell'Algoritmo 7) ma, per garantire la correttezza del risultato e per

quanto spiegato nella Sezione 3.3.2, una successiva selezione sulla non nullità delle feature appartenenti all'insieme dei predicati di selezione viene applicata dopo ciascun passo di fusione a livello di entity plan (Algoritmo 5).

- **Raggruppamento dei piani di esecuzione a livello di schema.** Poiché in una collezione, da definizione, potrebbero essere definiti più schemi, la soluzione naif andrebbe ad accedere ad accedere ad una collezione tanti quanti sono gli schemi definiti per quella collezione. Nella soluzione proposta nella sezione 3.3.4, questo viene limitato poiché per tutti gli schemi della stessa collezione riferiti alla stessa entità viene effettuato un solo accesso. Per questo motivo già nella crezione del collection plan (Algoritmo 7) si ricorre all'uso dei mapping e alla risoluzione di potenziali conflitti dovuti alla possibile presenza di più schemi.

- **Riordinamento delle sequenze di fusione.** Quando l'interrogazione coinvolge tre o più collezioni l'ordine secondo il quale esse vengono fuse tra loro ha un impatto sulle performance. Come già detto, viene utilizzata l'euristica di selettività minima [63]. Questa tecnica è utilizzata sia effettuare la fusione dei collection plan in un unico entity plan (Linea 3 dell'Algoritmo 6) che per effettuare la fusione degli entity plan nel query plan (Linea 1 dell'Algoritmo 3). L'idea di base è di partire dalla collezione (o dall'entità) con cardinalità più bassa e man mano fondere le collezioni (o le entità) con cardinalità crescente. Nella versione utilizzata, per cui, viene solo considerata la cardinalità dei record e non il fattore di selettività del join, poiché si ipotizza un'indipendenza dei predicati, per costruzione l'operatore di merge va ad effettuare un full-outer join e nel dataspace creato non ci sono informazioni riguardanti statistiche degli attributi. La cardinalità di ciascuno schema è memorizzata all'interno del dataspace e viene utilizzata per calcolare, di conseguenza, le cardinalità a livello di collezione e di entità, sem-

plicemente come somma delle cardinalità a livello di dettagli più basso (ovvero, per le collezioni vi è una somma delle cardinalità a livello di schema che appartengono a tale collezione e, invece, a livello di entità vi è una somma delle somme delle cardinalità calcolate per le collezioni che appartengono alla entità). È bene sottolineare che vi è una sovrastima nel calcolo di tali cardinalità: in caso di record sovrapposti, un record viene contato in entrambi gli schemi a cui appartiene.

- ***Column pruning*** **(potatura delle colonne).** Questa tecnica consiste nell'estrarre da ogni collezione i soli attributi corrispondenti alle feature necessarie nell'interrogazione richiesta e all'esecuzione della stessa. Minimizzando il set di attributi da proiettare, di conseguenza viene ridotto il quantitativo di dati che si spostano nella rete. Questo aspetto parte dalla creazione del collection plan (Linea 10 dell'Algoritmo 7), dove queste feature vengono considerate come F_π. Tale insieme comprende le feature da proiettare (appartenti a $feat(q_\pi) \cup feat(q_\gamma)$) e quelle necessarie nelle operazioni di merge. Successivamente, una volta creato il piano, viene analizzato l'albero creato e viene valutato in modo top-down, dopo ogni passo di merge, quali feature non è necessario proiettare.

Capitolo 4

Il prototipo realizzato

In questo capitolo viene presentato il prototipo realizzato, descrivendo la sua architettura e le tecnologie utilizzate, rispettivamente nelle Sezioni 4.1 e 4.2.

4.1 Architettura del prototipo

L'architettura funzionale e tecnologica del prototipo è mostrata in Figura 4.1. L'ApplicationLogic (logica dell'applicazione) è sviluppata in Scala. Essa include le funzionalità per creare, gestire e visualizzare i dataspace. Si sottolinea, infatti, che i dataspace utilizzabili dal prototipo sono più di uno, sia per quanto riguarda le diverse versioni di multistore generate (maggiori dettagli su questo vengono descritti nella Sezione 5.1.1) sia per le possibili versioni di dataspace per lo stesso multistore. L'utente può interagire, modificare e interrogare ciascuna di queste istanze e crearne delle nuove. In particolare, un dataspace viene generato attraverso le seguenti tre fasi.

1. Estrazione degli schemi e degli attribuiti: questo processo avviene parallelamente su ogni DBMS, vengono raccolte anche statistiche relative al numero di record per schema e, in questa fase, potrebbero venire raccolte altre informazioni rilevanti per la generazione del dataspace. La chiave primaria di ciascuno schema viene fornita manualmente. Si

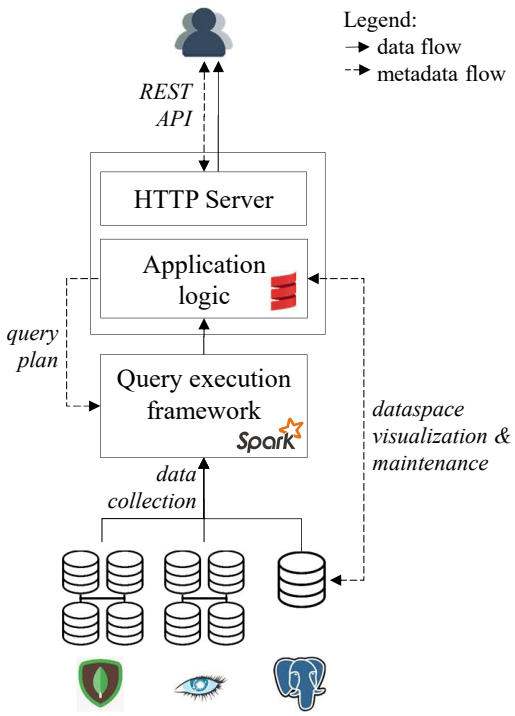

Figura 4.1: Architettura funzionale e tecnologica del prototipo.

ricorda che questa è la parte più onerosa a livello di tempistiche perché è necessario analizzare ciascun record di una collezione.

2. Definizione manuale dei mapping tra gli attributi.

3. Estrazione automatica di feature, entità e entity graph, seguendo le automazioni presentate nella Sezione 2.3. Spark viene sfruttato anche in questa fase per l'estrazione delle feature e per verificare l'aciclità dell'entity graph. Questa fase risulta pressocchè immediata.

Le istanze di dataspace sono memorizzate sotto forma di tabelle all'interno della stessa istanza di PostgreSQL usata per i dati del multistore. Esse seguono il diagramma UML delle classi mostrato in Figura 4.2, si noti che vengono create (e gestite) una serie di denormalizzazioni per favorire un accesso più rapido alle informazioni (ad esempio, all'interno di una feature si ha sia il riferimento agli attributi che la compongono, esso potrebbe essere ricavato

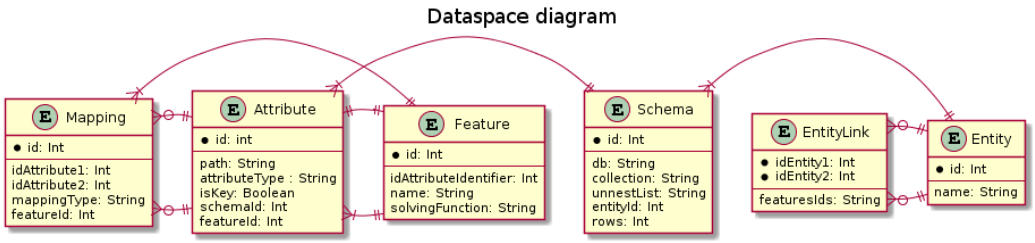

Figura 4.2: Diagramma UML del dataspace utilizzato all'interno del prototipo.

a partire dai mapping). Inoltre, oltre alle tabelle visibili dal diagramma, è presente una tabella con i suggerimenti che l'utente vuole fornire nell'utilizzo del dataspace, al momento è possibile: suggerire la chiave di partizionamento di una collezione, i valori minimo e massimo che essa assume e il numero di partizioni. L'utente, in ogni caso, può intervenire in qualsiasi momento sulle informazioni di un dataspace, modificandole in maniera consistente rispetto alle loro definizioni.

Per l'interrogazione dei dataspace sono stati implementati gli Algoritmi 1, 3, 4, 5, 6 e 7. La possibilità di interrogare i dataspace avviene mediante un HTTP Server che espone all'utente una REST API per poter formulare l'interrogazione nei dataspace disponibili. L'interrogazione viene, attraverso una serie di passaggi, tradotta in un piano di esecuzione eseguibile in Spark grazie all'astrazione fornita dal DataFrame. In particolare, il Query execution framework (framework di esecuzione delle query) consiste in 4 executor, ognuno con 4 core di CPU e 8GB di RAM[1]. Nell'eseguire l'interrogazione i dati vengono recuperati dai DBMS sottostanti, facendo in modo che la maggior parte della computazione venga spostata verso gli stessi e forzando Spark a non attuare ottimizzazioni. Infatti, alcune delle ottimizzazioni di Spark potrebbero andare a inficiare nel piano formulato tramite l'implementazione degli algoritmi presentati, anticipando operazioni che non dovrebbero essere anticipate per ottenere un risultato corretto. Successivamente, il Query execution

[1]Il numero di tali risorse è limitato per garantire abbastanza risorse ai DBMS sottostanti e alle altre applicazioni che vengono eseguite all'interno del cluster.

framework esegue il calcolo in memoria per completare l'esecuzione dell'interrogazione e ottenere il risultato effettivo della query, da restituire all'utente in risposta alla richiesta HTTP fornita. La richiesta oltre che semplicemente richiedere di formulare una determinata interrogazione in un tale dataspace, può specificare dei parametri: (i) l'indice del query graph da utilizzare, nel caso in cui la query abbia più di un grafo; (ii) decidere se coinvolgere un'entità nell'interrogazione anche se di essa è coinvolta solo la feature che svolge il ruolo di chiave esterna; (iii) quali ottimizzazioni si intendono non sfruttare, ovvero: disabilitare l'uso delle euristica di selettività minima e/o il controllo finale del piano di esecuzione per attuare il column pruning completo; (iv) il tipo di esecuzione che si richiede: ad esempio, è possibile esprimere se si vuole lanciare un'interrogazione senza ricevere il risultato o soltanto visualizzare il possibile piano di esecuzione ricavato dalla query. I parametri espressi dalle Voci (iii) e (iv) sono molto utili per fare valutazioni sulle tempistiche e sulle ottimizzazioni utilizzate e sono, infatti, sfruttati nella valutazione del prototipo presentata nel Capitolo 5.

I DBMS sottostanti sono: PostgreSQL per quanto riguarda il modello dati relazionale, MongoDB per quanto riguarda quello documentale, Cassandra per quanto riguarda il wide-column. Si sottolinea, inoltre, che mentre l'istanza di PostgreSQL è installata su una sola macchina, le istanze di MongoDB e Cassandra sono distribuite su 15 macchine. Nell'interrogazione delle sorgenti si cerca di sfruttare gli indici e le partizioni presenti nelle sorgenti stesse. L'uso degli indici da parte di Spark risulta poco trasparente e, per ottimizzazioni interne del framework, a volte l'uso degli indici viene trascurato e scoraggiato.

4.2 Tecnologie utilizzate

Le tecnologie utilizzate nella creazione del prototipo, come mostrato in Figura 4.1, sono molteplici. Tutte le tecnologie lavorano all'interno di un cluster big data a due rack formato da 18 macchine con una configurazione mini-

ma di CPU i7 8-core @3.2GHz, 32GB di RAM e 6TB di hard disk. Ogni macchina esegue la distribuzione Cloudera per Apache Hadoop (CDH) 6.2.0.

Apache Hadoop. Hadoop [71] è il principale framework open source usato in una piattaforma di big data. Si tratta di un progetto Apache (nato da Yahoo! e ispirato da Google) che fornisce una libreria software con un insieme di strumenti di base per lavorare su un cluster di computer di *commodity hardware* (hardware di base e pronto all'utilizzo, da non confondere con hardware a basso costo e di scarsa qualità). Le caratteristiche principali sono le seguenti.

- Permette l'elaborazione distribuita di grandi quantità di dati.
- È predisposto per scalare su migliaia di macchine.
- È progettato per garantire affidabilità e continuità di servizio (high-availability): la libreria stessa è progettata per rilevare e gestire i guasti a livello di applicazione, in modo da fornire un servizio a disponibilità elevata su un cluster di computer.
- Viene utilizzato usato in produzione da Google, Facebook, Yahoo! e tanti altri.

Apache Hadoop è composto dai seguenti moduli.

- *Common*: un set di librerie e utility (servizi essenziali, JARs, etc.).
- *HDFS* (Hadoop Distributed File System): file system distribuito.
- *MapReduce*: il primo paradigma di programmazione per l'elaborazione di dati su larga scala [49]. Per programmare in MapReduce è necessario definire una funzione di *map* (mappa) che va a trasformare i record in forma chiave-valore e una di *reduce* (riduzione/aggregazione) che va ad aggregare l'insieme di record con la stessa chiave. Tutte le operazioni di map possono essere svolte in parallelo da un insieme di *mapper* (mappatori). Le operazioni di reduce, invece, vengono svolte dai *reducer* (riduttori) e in questo caso, per poter aggregare tutti i record con

la stessa chiave, può venire meno il *principio di data locality* (principio di località dei dati). Infatti secondo tale principio bisognerebbe cercare non spostare dati nella rete e, piuttosto, spostare l'elaborazione verso i dati. Ma in tal caso non è sempre possibile, infatti, tutti i record con la stessa chiave devono essere spostati nel nodo che si occupa di svolgere tale operazione.

- *YARN* (Yet Another Resource Manager): una piattaforma di gestione delle risorse, allocare processi su singole macchine e distribuire carico di lavoro.

Cloudera per Apache Hadoop (CDH). CDH è la piattaforma di distribuzione open source di Cloudera, include Apache Hadoop ed è costruita appositamente per soddisfare le esigenze aziendali. Essa semplifica la gestione di installazioni dello stack tecnologico, integrando Hadoop con più di una dozzina di altri progetti open source. CDH può essere definito come un sistema funzionalmente avanzato che aiuta a eseguire flussi di lavoro Big Data end-to-end.

È possibile dividere le altre tecnologie usate in due gruppi: (i) tecnologie a livello di raccolta, mantenimento e memorizzazione di dati, ovvero tutti i DBMS impiegati per formare il multistore e per poter mostrare effettivamente il funzionamento del prototipo; (ii) tecnologie al supporto della gestione del concetti descritti nei Capitoli 2 e 3 e, quindi, principalmente della creazione del dataspace e del query plan. I punti sopra esposti verranno sviscerati di seguito, rispettivamente, nelle Sezioni 4.2.1 e 4.2.2.

4.2.1 Tecnologie a livello di database

Nella creazione del multistore sono stati utilizzati database afferenti ai modelli dati relazionale, doumentale e wide-column, utilizzando le tecnologie di seguito descritte.

PostgreSQL. Il DBMS relazionale utilizzato è PostgreSQL [51]. Esso è uno tra i DBMS relazionali ed open-source più utilizzati ed enfatizza l'esten-

sibilità e la conformità SQL. L'istanza di tale RDBMS è installata su una singola macchina del cluster. Tale istanza è anche utilizzata per memorizzare e manipolare i dati relativi al dataspace costruito sopra al multistore.

MongoDB. Il database afferente al modello dati documentale utilizzato è MongoDB [18]. Si tratta di un database orientato ai documenti e open-source. Esso è considerato generico, potente, flessibile e scalabile; combina la capacità di scalabilità orizzontale con funzionalità quali indici secondari, query di range, ordinamento, aggregazioni e indici geo-spaziali. L'istanza utilizzata è distribuita su 15 macchine del cluster.

Cassandra. Apache Cassandra [16] è il database usato per quanto concerne il modello dati wide-column. Esso è un DBMS non relazionale che supporta il modello dati wide-column. È caratterizzato dalle seguenti caratteristiche: open-source, distribuito, decentralizzato, scalabile, a tolleranza di errore. L'istanza utilizzata è distribuita su 15 macchine del cluster.

4.2.2 Tecnologie a livello di dataspace

Le tecnologie utilizzate per creare e interrogare il dataspace sono Scala e Spark.

Scala. Scala [53] è un linguaggio di programmazione di tipo general-purpose e multi-paradigma, nasce per integrare aspetti di programmazione orientata agli oggetti con aspetti di programmazione funzionale. L'intero prototipo è sviluppato in Scala e molto spesso si ricorre all'uso di Spark.

Spark. Apache Spark [62] è un famoso framework Scala per big-data e cluster computing. Si tratta di un motore di esecuzione general porpuse (adatto per analisi batch, streaming, machine learning, etc.) con le seguenti caratteristiche principali.

1. *In-memory data caching*: i dati che vengono letti dal disco possono essere memorizzati in RAM.

2. *Lazy computation*: la computazione viene eseguita solo quando se ne ha effettivamente bisogno.

3. *Efficient pipeling*: la scrittura sul disco rigido è evitata il più possibile.

Spark è uno dei framework di esecuzione open-source più utilizzati per i cluster Apache Hadoop. Supera i limiti di MapReduce tradizionale, rendendo la programmazione secondo il paradigma MapReduce molto più semplice e al passo con le nuove caratteristiche hardware. Offre connettori ai principali database commerciali, tra cui quelli sopra menzionati, per cui connettersi ad un database risulta molto semplice.

Tra i moduli offerti da Spark, quelli utilizzati all'interno dell'elaborato sono Spark SQL e Graphx. **Spark SQL** [3] è il modulo che fornisce un'astrazione strutturata sui dati che vengono letti dalle sorgenti (ad esempio, database di varia natura o file). I dati, rispetto alle funzionalità offerte dal modulo base di Spark, vengono memorizzati sotto forma di tabelle, grazie alle estrazioni di Spark chiamate DataFrame e Dataset. Esprimere le interrogazioni su tali strutture risulta piuttosto naturale, sia attraverso un linguaggio relazionale che procedurale (ad esempio mediante il linguaggio SQL o la libreria Scala). All'interno di SparkSQL vi è un ottimizzatore, chiamato *Catalyst*, che crea piani di esecuzione ottimizzati ma viene, allo stesso tempo, lasciato spazio al programmatore per cambiare le regole su cui esso si basa. Nel nostro caso utilizzare SparkSQL è stato fondamentale per poter trasformare il piano di esecuzione costruito nel risultato vero e proprio dell'interrogazione, lavorando ricorsivamente su DataFrame. Inoltre, esso è stato utilizzato per estrarre gli attributi e, di conseguenza, gli schemi dai record di ciascuna collezione in modo parallelo. **Graphx** è il componente di Spark che supporta l'elaborazione su grafi, fornendo una serie di algoritmi per gli stessi. Tale componente è stato utilizzato sia nella parte di creazione del dataspace per estrarre le feature e verificare l'aciclicità dell'entity graph (come presentato nella Sezione 2.3), sia nella parte di esecuzione dell'interrogazione per creare il query graph (come discusso nella Sezione 3.3.1).

Capitolo 5

Test sperimentali

Per concludere e valutare quanto proposto, in questo capitolo vengono descritti i dati utilizzati, i test svolti e i risultati ottenuti. Nel dettaglio, nella Sezione 5.1 vengono descritti gli aspetti chiave dei dati utilizzati all'interno del prototipo e, nella Sezione 5.2, vengono presentati i test effettuati e i risultati ottenuti.

5.1 I dati utilizzati

La descrizione dei dati utilizzati parte dalla descrizione del multistore (nella Sezione 5.1.1) su cui va a tirare le fondamenta il dataspace estratto e procede con l'estrazione del dataspace stesso (nella Sezione 5.1.2).

5.1.1 Il multistore utilizzato

In questo lavoro viene utilizzato un multistore che comprende database che afferiscono a tre modelli dati: relazionale, documentale e wide-column. I dati appartenenti al multistore sono una variazione di Unibench [57], ovvero un dataset di riferimento per database multi-modello basato su un'applicazione di e-commerce che memorizza i dettagli sui prodotti ordinati e acquistati dai clienti. Lo schema concettuale è mostrato in Figura 5.1, la quale mostra il diagramma UML delle classi (sulla sinistra) e l'implementazione fisica sui

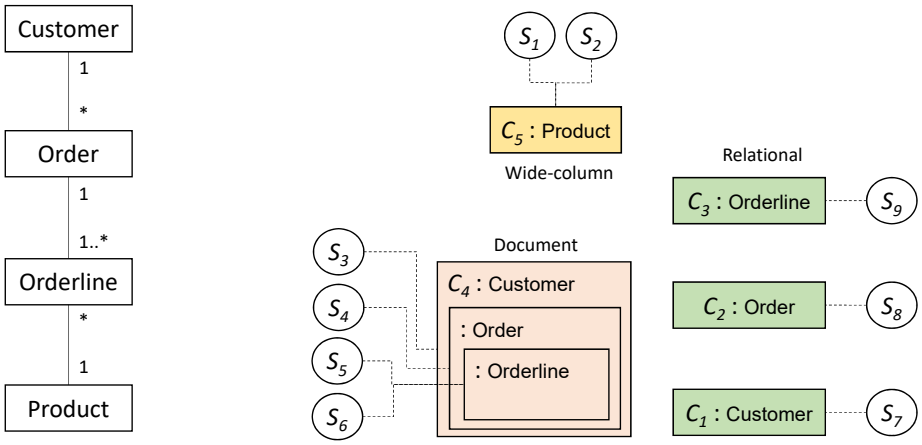

Figura 5.1: Il diagramma UML delle classi (a sinistra) e la rappresentazione grafica dell'implementazione fisica (a destra) del multistore utilizzato nel prototipo. I diversi colori rappresentano i diversi DBMS con diverso modello dati.

differenti DBMS (sulla destra). In quest'ultimo, i simboli da C_1 a C_5 rappresentano le collezioni/tabelle di dati all'interno del multistore, mentre la notazione ":" sta ad indicare il livello di granularità del dato che contiene tale una singola collezione. I simboli da S_1 a S_9, invece, rappresentano i diversi schemi presenti in una collezione. È bene sottolineare che il database documentale contiene una singola collezione (cioè C_4) che usa una struttura innestata per memorizzare dentro ai dati del Customer (Cliente) i dati dell'Order (ordine) e, dentro a quest'ultimo, i dati delle Orderline (righe d'ordine). Inoltre, i colori rappresentano, per l'appunto, l'utilizzo di database differenti. Nello specifico, i dati relativi al Product (prodotto) sono memorizzati all'interno di una database wide-column e i dati relativi agli al cliente, agli ordini e ai dettagli d'ordine sono divisi tra i database con modello dati relazionale e documentale. Nella versione dei dati utilizzati ci sono le seguenti differenze rispetto a Unibench.

- Non vengono utilizzati database per i modelli dato a grafo e chiave-valore.

- Viene inserita dell'eterogeneità a livello di schema, nei dati memorizzati nei database con modello dati documentale e wide-column (ovviamente, nel database relazionale questo non è possibile per quanto detto nella Sezione 1.1.1). Nel dettaglio vengono inserite le seguenti caratteristiche: (i) *attributi mancanti*: nella collezione C_5 vi è un attributo mancante nel 10% dei record (imgUrl); (ii) *attributi con tipo di dato diverso*: ad esempio, la quantità della riga d'ordine è memorizzata come un intero in S_5 e come una stringa in S_6 e la data dell'ordine (orderDate) è memorizzata come una data in C_2 e come una stringa in C_4; (iii) *equivalenze semantiche*: in C_4 vengono utilizzate due convenzioni diverse per rappresentare le righe degli ordini. Si evidenzia, inoltre, che in C_4 vengono memorizzati anche i dati del prodotto denormalizzati con i dati della riga d'ordine.

- Sono stati introdotti record sovrapposti in DBMS differenti. In particolare, per simulare un scenario con record sovrapposti, i dati relativi al cliente, agli ordini e ai dettagli d'ordine sono divisi tra i database con modello dati relazionale e documentale come segue: (i) i record relativi al cliente sono divisi tra le collezioni C_1 e C_4 in modo sovrapposto: entrambe le collezioni contengono il 60% dei record originali, questo significa che il 20% dei clienti viene replicato; (ii) i record relativi agli ordini e alle righe d'ordine sono divisi tra i due DBMS, ma non sono sovrapposti e, quindi, gli ordini che appartengono a clienti sovrapposti vengono memorizzati in uno e solo uno dei due DBMS.

Per poter valutare l'approccio in termini di scalabilità, sono state generate tre versioni del multistore con tre *fattori di scala* (o ridimensionamento) differenti: 1, 10 e 100. In particolare, i dati originali sono stati manipolati in Scala e si è creato un generatore di multistore che consente, attraverso una serie di parametri (tra cui il fattore di ridimensionamento, la percentuale di sovrapposizione di record e la percentuale di record con attributo mancante), di creare le sorgenti del proprio multistore e installarle nei DBMS di riferimento. La dimensione di ogni collezione nelle differenti versioni del

Tabella 5.1: Numero di record e chiave di partizionamento per ogni collezione nei diversi fattori di ridimensionamento.

Entity	Collection	Cardinalità SF1	Cardinalità SF10	Cardinalità SF100	Chiave di partizionamento
Customer	C_1	5,970	59,700	597,000	FirstName
	C_4	5,968	59,680	596,800	FirstName
	Total	**10,000**	**100,000**	**1,000,000**	-
Order	C_2	71,048	742,745	7,459,715	OrderDate
	C_4	71,209	742,627	7,456,807	-
	Total	**142,257**	**1,485,372**	**14,916,522**	-
Orderline	C_3	346,649	3,705,130	37,289,874	OrderId
	C_4	347,262	3,704,355	37,275,361	-
	Total	**693,911**	**7,409,485**	**74,565,235**	-
Product	C_5	**9,691**	**9,691**	**9,691**	ProductName

multistore è mostrata nella Tabella 5.1. Nel ridimensionamento, dunque, si scala sul numero di clienti e, con essi, aumenta il numero di ordini e di righe d'ordine. Il numero dei prodotti rimane fisso, così come la media di ordini per cliente (in media 15) e di righe d'ordine per ordine (in media 5).

La Tabella 5.1, inoltre, evidenzia la chiave di partizionamento per ogni collezione. Il partizionamento, oltre essere necessario nei DBMS distribuiti per distribuire i dati, aiuta ad aumentare l'efficienza delle interrogazioni su ogni DBMS in presenza di un certo predicato di selezione. Si noti che nel database con modello dati documentale la collezione C_4 è partizionata esclusivamente sull'attributo FirstName (nome) del cliente, questo avviene perché non sono possibili ulteriori partizionamenti ad altri livelli di dettaglio considerando la struttura innestata sopra descritta. Ogni database, oltre al

Tabella 5.2: Indici presenti per ogni collezione, raggruppati per entità.

Entity	Collection	Indici
Customer	C_1	FirstName, Gender
	C_4	FirstName, Gender
Order	C_2	OrderDate, TotalPrice
	C_4	OrderDate, TotalPrice
Orderline	C_3	-
	C_4	-
Product	C_5	-

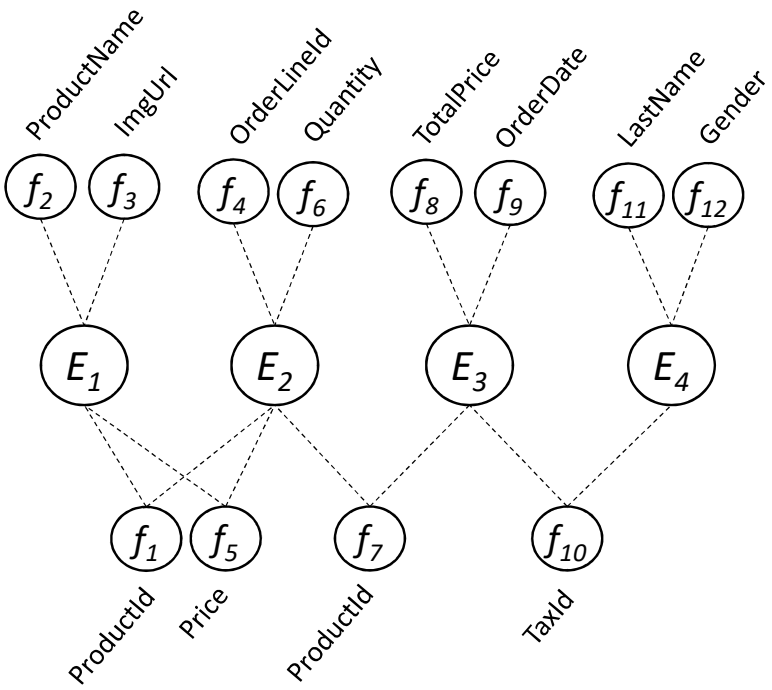

Figura 5.2: Il dataspace utilizzato nel prototipo.

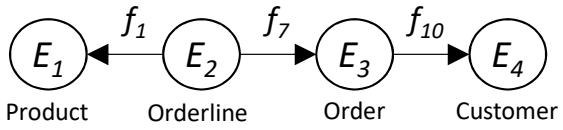

Figura 5.3: L'entity graph del dataspace utilizzato nel prototipo.

partizionamento appena descritto, ha degli indici per cercare di ottimizzare ulteriori tipi di carichi di lavoro. La tabella Tabella 5.2 mostra gli indici presenti in ogni collezione del multistore, raggruppati per entità.

5.1.2 Il dataspace utilizzato

Il dataspace costruito sopra i dati presentati nella sezione 5.1.1 viene mostrato nella Figura 5.2 e nella Tabella 5.3, la quale mostra – in modo dettagliato su ogni attributo (da a_1 a a_{35}) – la distribuzione delle feature (da f_1 a f_{12}) e delle entità (da E_1 a E_4) nel dataspace. Si ricorda che gli schemi (da S_1 a S_9)

Tabella 5.3: La tabella mostra la corrispondenza tra attributi e schemi; la cella $[i, j]$ ha un segno di spunta (✓) se $a_i \in S_j$, o la lettera "K" se $a_i = key(S_j)$. Gli attributi sono organizzati per feature f_k e indicano la collezione C_l a cui essi appartengono, mentre gli schemi sono organizzati per entità E_m; le collezioni e gli schemi corrispondo a quelli mostrati in Figura 5.1. Tutti questi concetti sono stati presentati nel Capitolo 2.

			$name(E)$	Product		Orderline			Order		Customer	
			E	E_1		E_2			E_3		E_4	
			S	S_1	S_2	S_5	S_6	S_9	S_4	S_8	S_3	S_7
$name(f)$	f	a	C									
ProductId	f_1	a_1	C_3					✓				
		a_2	C_4				✓					
		a_3	C_4			✓						
		a_4	C_5	K								
		a_5	C_5		K							
ProductName	f_2	a_6	C_5	✓								
		a_7	C_5		✓							
ImgUrl	f_3	a_8	C_5	✓								
OrderLineId	f_4	a_9	C_3					K				
		a_{10}	C_4			K						
		a_{11}	C_4				K					
Price	f_5	a_{12}	C_4			✓						
		a_{13}	C_4				✓					
		a_{14}	C_5	✓								
		a_{15}	C_5		✓							
Quantity	f_6	a_{16}	C_3					✓				
		a_{17}	C_4			✓						
		a_{18}	C_4				✓					
OrderId	f_7	a_{19}	C_3					✓				
		a_{20}	C_3							K		
		a_{21}	C_4			✓						
		a_{22}	C_4				✓					
		a_{23}	C_4						K			
TotalPrice	f_8	a_{24}	C_2							✓		
		a_{25}	C_4						✓			
OrderDate	f_9	a_{26}	C_2							✓		
		a_{27}	C_4						✓			
TaxId	f_{10}	a_{28}	C_1									K
		a_{29}	C_2							✓		
		a_{30}	C_4								K	
		a_{31}	C_4						✓			
LastName	f_{11}	a_{32}	C_1									✓
		a_{33}	C_4								✓	
Gender	f_{12}	a_{34}	C_1									✓
		a_{35}	C_4								✓	

e le collezioni (da C_1 a C_5) presenti nei database che formano il multistore sono quelli mostrati in Figura 5.1 e, inoltre, che un attributo è contenuto in uno ed un solo schema e ogni schema contiene un attributo che svolge il ruolo di chiave. Si sottolinea che S_1 è diverso da S_2 per la presenza di un attributo rappresentante f_3 e che S_5 differisce da S_6 per il tipo f_6 (a_{17} è una modellato come una stringa e a_{18} come un numero).

Gli schemi nella Tabella 5.3 sono organizzati per *entità*. Si noti che esistono molte feature che si sovrappongono su più entità (ProductId, Price, OrderId e TaxId). Questo si nota molto bene nella Figura 5.2, che mostra le entità e le feature presenti nel dataspace, evidenziando tale sovrapposizione. Alcune di queste feature, nello specifico quelle che in un'entità svolgono il ruolo di chiave e in un'altra no (ProductId, OrderId e TaxId), saranno il punto chiave per la creazione dell'entity graph mostrato in Figura 5.3.

Gli attributi mostrati nella Tabella 5.3 sono, invece, organizzati per *feature*, in tale rappresentazione, per semplicità grafica, i mapping sono impliciti all'interno della stessa feature. Ad esempio, poiché $feat(a_6) = feat(a_7) = f_2$ allora $a_6 \equiv a_7$. Inoltre, si ricorda che attraverso i mapping è possibile ricavare implicitamente le relazioni tra gli schemi. Ad esempio, poichè $key(S_1) \equiv key(S_2)$ allora $S_1 \leftrightarrow S_2$; similmente si ha $S_5 \leftrightarrow S_6 \leftrightarrow S_9$. Invece, il mapping $a_3 \equiv a_4$ indica $S_5 \xrightarrow{a_4} S_1$ poiché $a_3 \neq key(S_5)$ e $a_4 = key(S_1)$. Grazie a queste inferenze viene ricavato l'entity graph mostrato in Figura 5.3. Si fa notare, inoltre, che ogni schema contiene al più un attributo rappresentante una tale feature.

Esempio 1. Per quanto riguarda l'**interrogazione di tale dataspace**, un esempio di possibile query q da formulare sopra lo stesso è la seguente: per ogni ProductName (nome di prodotto), calcolare la media delle Quantity (quantità) acquistata clienti di Gender (genere) femminile ordinati a partire dal 2009 (attributo OrderDate). Questa interrogazione, seguendo la definizione di query presentata nella Sezione 3.1, viene tradotta come segue: l'insieme di feature da proiettare (il *group-by set*) di q è $q_\pi = \{f_2\}$, l'insieme di aggregazioni è $q_\gamma = \{(f_6, avg())\}$ e l'insieme di predicati di selezione è

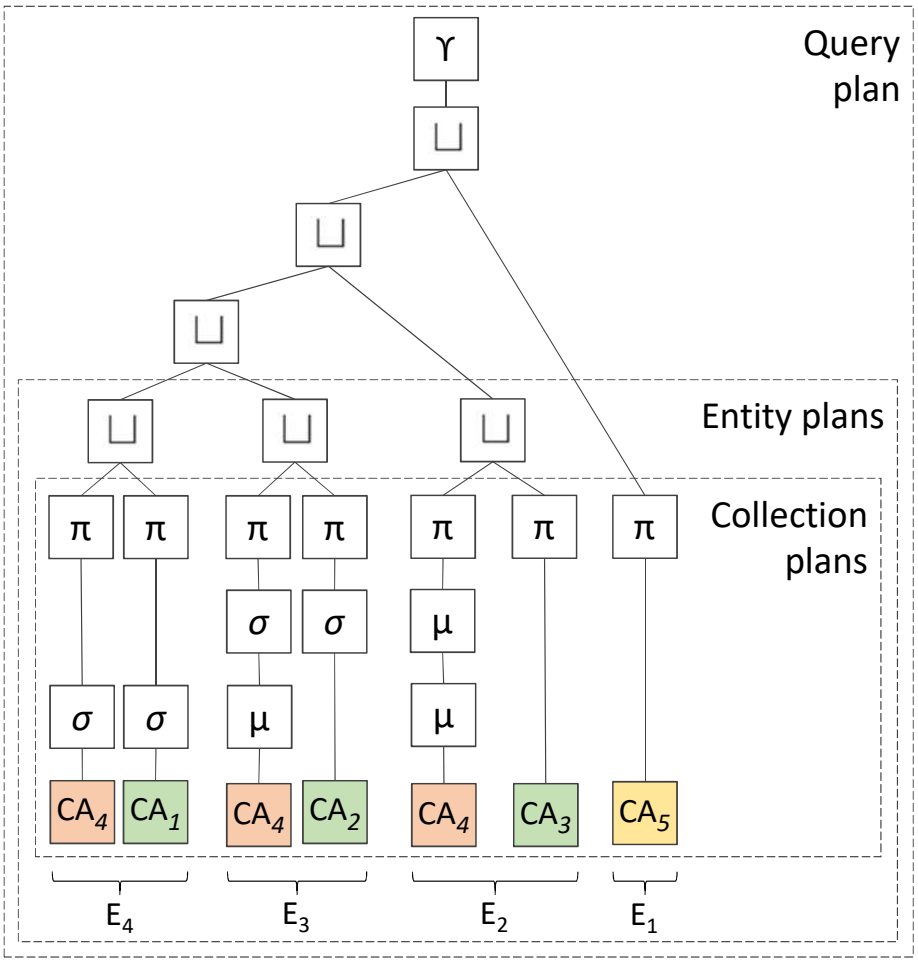

Figura 5.4: Il piano di esecuzione per la query dell'esempio 1.

$q_\sigma = \{(f_9,\ \geq,\ \text{“2019/01/01”}), (f_{13},\ =,\ \text{“}F\text{”})\}$.

Il piano di esecuzione di q è mostrato in Figura 5.4, per semplicità non vengono mostrati i predicati degli operatori. Come descritto nel Capitolo 3, le foglie del *query plan* possono essere organizzate in *entity plan* e le foglie di ogni entity plan possono essere organizzate in *collection plan*. Il piano, dunque, parte dal basso con la riconciliazione a livello di entità e, solo dopo, viene fatta l'unione tra concetti diversi. Ogni passo di fusione è svolto con l'uso dell'operatore di merge presentato nella Sezione 3.2. Ad esempio, considerando la risoluzione a livello dell'entità E_4, sia $S_3 \sqcup_{f_{10}} S_7$. Considerando due record s e t con $s \in C_4$ con schema S_3, $t \in C_1$ con schema S_7; sia $s[a_{28}] = t[a_{30}]$ dove $attr(f_{10}) \supset \{a_{28}, a_{30}\}$. Siano i valori dell'attributo LastName (cognome) $s[a_{33}]$ = “Rossi” e $t[a_{32}]$ = “Rosi”. La fusione tra s e t produce un record r dove $r[f_{10}] = \mathbb{M}_1(s[a_{33}], t[a_{32}])$ e $\mathbb{M}_1$ è una funzione di risoluzione dei conflitti che decide quale valore tra “Rossi” e “Rosi”, o una combinazione dei due, deve essere considerato in $r[f_{10}]$. Ad esempio, in questo caso una possibile soluzione potrebbe essere prendere il valore massimo tra i due (seguendo l'ordine alfabetico). Potrebbero essere interessanti anche funzioni di risoluzione dei conflitti più complesse, come discusso nella Sezione 2.3.

5.2 Test e risultati ottenuti

I test effettuati nel prototipo presentato nel Capitolo 4 e nella Sezione 5.1 mirano ad analizzare i seguenti aspetti fondamentali: (i) l'*efficienza* dell'approccio proposto (discussa nella Sezione 5.2.1), attraverso 18 interrogazioni che mirano a valutare il comportamento del prototipo in un'ampia serie di circostanze; (ii) l'*efficacia* dell'approccio proposto (discussa nella Sezione 5.2.2), valutando come cambiano i risultati delle interrogazioni considerando una serie di variazioni applicate alla conoscenza fornita nel dataspace.

Tabella 5.4: Le query del carico di lavoro usato per valutare l'efficienza del prototipo.

Q	Group-by set	Pred. di sel.	q_π	q_γ	q_σ
Q1.1	Assente	Assente	{FirstName, LastName}	{}	{}
Q1.2		Debole	{FirstName, LastName}	{}	{(Gender, =, "female")}
Q1.3		Forte	{FirstName, LastName}	{}	{(FirstName. = ."Erik")}
Q1.4	Debole	Assente	{LastName}	{TotalPrice, $sum()$}	{}
Q1.5		Debole	{LastName}	{TotalPrice, $sum()$}	{(Gender, =, "female")}
Q1.6		Forte	{LastName}	{TotalPrice, $sum()$}	{(FirstName, =, "Erik")}
Q1.7	Forte	Assente	{BrowserUsed}	{TotalPrice, $sum()$}	{}
Q1.8		Debole	{BrowserUsed}	{TotalPrice, $sum()$}	{(Gender, =, "female")}
Q1.9		Forte	{BrowserUsed}	{TotalPrice, $sum()$}	{(FirstName, =, "Erik")}
Q2.1	Assente	Assente	{OrderDate, TotalPrice}	{}	{}
Q2.2		Debole	{OrderDate, TotalPrice}	{}	{(OrderDate, <, "2020-01-01")}
Q2.3		Forte	{OrderDate, TotalPrice}	{}	{(TotalPrice, <, 100)}
Q2.4	Debole	Assente	{OrderDate, Brand}	{Quantity, $sum()$}	{}
Q2.5		Debole	{OrderDate, Brand}	{Quantity, $sum()$}	{(OrderDate, <, "2020-01-01")}
Q2.6		Forte	{OrderDate, Brand}	{Quantity, $sum()$}	{(TotalPrice, <, 100)}
Q2.7	Forte	Assente	{Brand}	{Quantity, $sum()$}	{}
Q2.8		Debole	{Brand}	{Quantity, $sum()$}	{(OrderDate, <, "2020-01-01")}
Q2.9		Forte	{Brand}	{Quantity, $sum()$}	{(TotalPrice. < .100)}

5.2.1 Valutazione dell'efficienza

Per poter valutare l'efficienza di quanto sviluppato si è creato un gruppo di 18 query GPSJ che variano in termini di forza dei group-by set e di selettività del predicato di selezione. Il livello dei group-by set considerato è sviluppato in tre livelli: (i) *assente*: viene fatta solo una proiezione senza nessuna aggregazione; (ii) *debole*: l'aggregazione coinvolge feature con un'elevata cardinalità, restituendo molti gruppi (righe) nel risultato; (iii) *forte*: l'aggregazione viene fatta su feature con una bassa cardinalità e, di conseguenza, il risultato sarà formato da un numero ridotto di gruppi. Allo stesso modo, viene fatta una suddivisione simile sui livelli di selettività del predicato di selezione: (i) *assente*: non viene effettuata alcuna selezione; (ii) *debole*: la selettività del predicato di selezione è bassa e, dunque, tra i record restituiti compaiono un numero elevato di record di quelli risultanti dalla stessa interrogazione senza filtro; (iii) *forte*: la selettività del predicato di selezione è alta e, perciò, i record che passano il filtro sono un numero limitato. Per

effettuare la valutazione sono stati identificati due gruppi di query (Q1 e Q2) mostrate nella Tabella 5.4. Tali query coprono ognuna delle combinazioni tra i group-by set e le selettività del predicato di selezione descritti, mantenendo lo stesso predicato di selezione quando si cambia il group-by set e viceversa. Le interrogazioni fatte su un gruppo simile di feature hanno la stessa numerazione nei due insiemi di query, ovvero la query Q1.3 è simile alla Q2.3.

Tabella 5.5: Tempi di esecuzione (in secondi) delle interrogazioni presentate nella Tabella 5.4.

Group-by set	Pred. di sel.	Q	Q1.y			Q2.y		
			SF 1	SF 10	SF 100	SF 1	SF 10	SF 100
Assente	Assente	Qx.1	3,7	3,8	5,2	4,4	11,9	102,7
	Debole	Qx.2	0,5	0,5	1,8	4,6	10,4	95,2
	Forte	Qx.3	0,3	0,3	0,4	4,3	10,5	92,4
Debole	Assente	Qx.4	5,1	12,4	97,5	7,5	23,7	212,3
	Debole	Qx.5	5,1	11,9	95	6,3	22,4	203,9
	Forte	Qx.6	5	11,4	95,5	5,7	22,4	203,7
Forte	Assente	Qx.7	4,9	11,2	95,1	5,4	13,6	121,3
	Debole	Qx.8	4,9	11,4	97,2	5,9	21,9	193,4
	Forte	Qx.9	4,7	11,2	95	5,7	21,1	203,4

Le query del carico di lavoro sono state eseguite nel multistore su ogni fattore di ridimensionamento (SF), ovvero 1, 10, 100, come descritto nella Sezione 5.1.1. Nella Tabella 5.5 sono mostrati i tempi di esecuzione (in secondi) delle interrogazioni mostrate nella Tabella 5.4. Il tempo di esecuzione è stato calcolato su una media di 5 esecuzioni di ogni interrogazione, in modo da ottenere un risultato abbastanza "pulito" dai possibili ritardi che potrebbero accadere in un'esecuzione distribuita tramite un framework Big Data. Da un'analisi dei tempi emerge che il tempo di esecuzione dipende principalmente sia dalla complessità dell'interrogazione che del dataset, ma è anche condizionato dalle di risorse computazionali allocate nel cluster. Infatti, i framework Big Data come Spark tendono a cercare di applicare il più possibile il principio di data locality, ma non garantiscono che la computazione avvenga negli stessi nodi. Per questo motivo il tempo di esecuzione della

stessa computazione potrebbe variare in base alla quantità di *data shuffling* (spostamento dei dati) richieste quando non è possibile applicare il principio di data locality. Al contrario, per quanto riguarda le tempistiche per la costruzione del piano di esecuzione, esso non risulta influenzato né dalla complessità dell'interrogazione né da quella del dataset. Questo avviene poiché gli algoritmi presentati nella Sezione 3.3 non sono influenzati da tali aspetti, ma si basano esclusivamente sull'integrazione leggera svolta a livello da dataspace. Inoltre, neanche il framework Big Data influenza le tempistiche di costruzione del piano di esecuzione, perché tale elaborazione è svolta in modo centralizzato ed il tempo è, in tal caso, costante nell'intervallo di 1-2 secondi.

Da un'analisi dei tempi di esecuzione mostrati nella Tabella 5.5 è possibile notare i seguenti punti.

- Le query nel gruppo Q2 sono generalmente più costose di quelle del gruppo Q1. Questo avviene poiché nelle query Q2 è coinvolta anche l'entità OrderLine che è quella con cardinalità maggiore rispetto a tutte le altre (ad eccezione per la query Q2.7, la quale è molto semplice perché non coinvolge l'entità Order).

- I tempi di esecuzione crescono quasi linearmente con il fattore di ridimensionamento, questo risulta abbastanza evidente quando si passa da fattore di replicazione 10 a 100.

- I predicati di selezione sembrano avere un effetto limitato. Il sistema, infatti, può sfruttare l'indicizzazione e/o il partizionamento locale solo sulle collezioni su cui vengono applicati i predicati di selezione. Per ulteriori dettagli si rimanda a quanto mostrato nelle Tabelle 5.1 e 5.2. In particolare, per quanto riguarda il partizionamento locale, si ricorda che il database documentale, considerata la struttura innestata, è partizionato solo a livello di clienti.

- Il comportamento in diverse condizioni di group by è diverso: i tempi di esecuzione sono più bassi in assenza di raggruppamento perché in tal

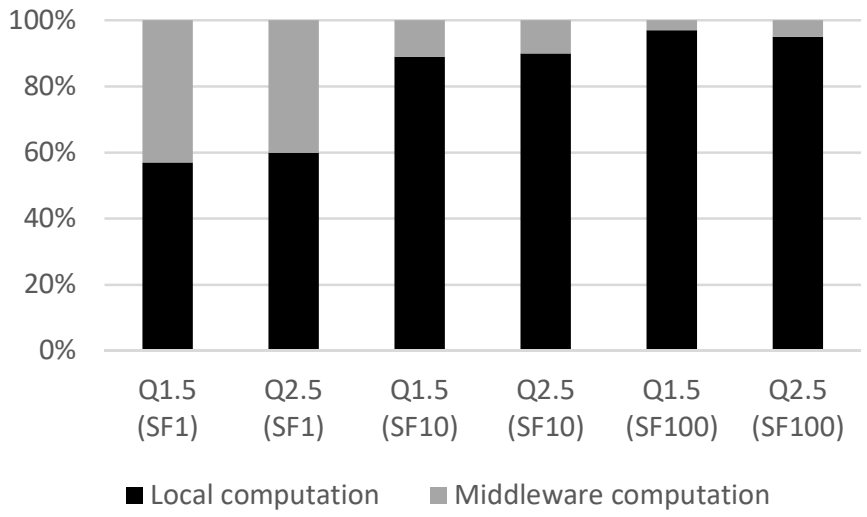

Figura 5.5: Confronto della computazione locale e sul middleware per le query Q1.5 e Q2.5 con diversi fattori di ridimensionamento.

caso non è richiesto lo *shuffling* (spostamento) dei dati per effettuare un'aggregazione, infatti nel momento in cui bisogna effettuare un'aggregazione è necessario trasferire dati nella rete per fare in modo che dati appartenenti allo stesso gruppo si trovino nello stesso nodo. Quando l'aggregazione è necessaria, il sistema funziona leggermente meglio con un group-by set più forte, in cui vengono generati e spostati meno record. Ciò è dovuto, nel contesto MapReduce, all'utilizzo del *combiner* (combinatore) che esegue una pre-aggregazione lato mapper, spostando così meno record per l'aggregazione lato reducer.

Una seconda valutazione è stata fatta per comparare quanta computazione è assegnata alle sorgenti rispetto a quella assegnata al middleware. In questo caso, vista la complessità di ottenere manualmente tali misure, si considerano le query Q1.5 e Q2.5 come riferimento per i diversi fattori di

ridimensionamento. Per ogni esecuzione dell'interrogazione: (i) la computazione locale è considerata come la somma dei tempi di esecuzione dei task di Spark responsabili di leggere dalle sorgenti; (ii) la computazione a livello di middleware è, invece, considerata come la somma dei tempi di esecuzione dei task di Spark rimanenti. Tale confronto è reso possibile perché si considerano le tempistiche a livello di task e, di conseguenza, si vanno ad "appiattire" i tempi di esecuzione non considerando tali esecuzioni come parallele. I risultati della valutazione sono mostrati in Figura 5.5, dove si nota in modo evidente che la percentuale di computazione richiesta a livello di middleware diminuisce con l'aumento del fattore di ridimensionamento. In termini assoluti, la computazione locale richiesta dalle sorgenti scala linearmente con il fattore di ridimensionamento, mentre la computazione lato middleware scala sublinearmente (circa 2x da SF 1 a SF 10, circa 5x da SF 10 a SF 100). Questo è probabilmente dovuto al fatto che il middleware subisce l'overhead del framework distribuito nella gestione di quantità ridotte di dati con un fattore di ridimensionamento inferiore. Pertanto, si deduce che fare affidamento sul middleware per fondere i record (che comporta lo scambio di dati sulla rete tra diversi strumenti software) non ha un impatto significativo, soprattutto quando la quantità di dati da considerare diventa maggiore.

Tabella 5.6: Perdita media dei tempi di esecuzione delle query disattivando le ottimizzazioni che riguardano il column pruning (CP) o l'euristica di selettività minima (MSR).

Q	Perdita senza CP			Perdita senza MSR		
	SF1	SF10	SF100	SF1	SF10	SF100
Q1.y	2.7%	8.2%	2.9%	-	-	-
Q2.y	0.4%	1.2%	0.5%	19.4%	3.8%	0.6%

Infine, una terza ed ultima valutazione è stata fatta per misurare l'impatto delle tecniche di ottimizzazione utilizzate, disattivandole selettivamente e verificando i tempi di esecuzione. In particolare, si sono considerate le ottimizzazioni che riguardano il column pruning (CP) e l'euristica di selettività minima (MSR). Le altre ottimizzazioni presenti e descritte nella Sezione 3.4

non sono state valutate perché: (i) considerare un piano di esecuzione a livello di schema e non di collezione, non risulta possibile nei database non relazionali; (ii) per quanto riguarda il push-down dei predicati, esso non avrebbe portato risultati interessanti da quelli che ci saremmo aspettati e, per questo motivom non si è considerato necessario valutarlo.

I risultati sono mostrati nella Tabella 5.6. In questo caso, vengono prese misure sulla media 5 esecuzioni di ogni interrogazione, su ogni fattore di ridimensionamento, e poi si misura la perdita media in percentuale rispetto all'esecuzione con tutte le ottimizzazioni in funzione. Tra le due ottimizzazioni, si nota che: MSR è quella che produce il vantaggio più significativo, mentre il contributo di CP è limitato e irregolare. Nella comprensione dei risultati ottenuti bisogna, inoltre, considerare che MSR è applicabile solo in Q2.2, Q5.2, Q6.2, Q8.2 e Q9.2 poiché tali interrogazioni sono le uniche che coinvolgono più di due entità o collezioni quando si applica l'operatore di merge. A differenza della valutazione precedente, i risultati mostrano che l'impatto di entrambe le ottimizzazioni diminuisce con l'aumento del fattore di ridimensionamento. Infatti, ciò può essere spiegato dalla stessa valutazione precedente: poiché le ottimizzazioni hanno un impatto sulla computazione a livello di middleware, è naturale che minore sarà la quantità di computazione lato middleware, minore sarà l'impatto delle ottimizzazioni.

5.2.2 Valutazione dell'efficacia

Considerando il dataspace costruito con i concetti presentati nel Capitolo 2, è evidente che la qualità delle risposte alle interrogazioni dipende dal numero di mapping definiti. A tal proposito, si sono svolti dei test per analizzare come il risultato di un'interrogazione varia rimuovendo dei determinati mapping. L'obiettivo è da un lato quantificare l'impatto dal punto di vista di riduzione dei mapping e, dall'altro, fornire esempi di degradazione della qualità dei risultati.

Sia $\mathcal{D}^*$ il dataspace definito nella Sezione 5.1.2, ovvero il dataspace che inquadra in maniera completa i dati disponibili nelle sorgenti (dato che viene

Tabella 5.7: Valutazione della degradazione della qualità in scenari con determinati mapping rimossi dal dataspace di riferimento $\mathcal{D}^*$ presentato nella Tabella 5.3. La densità e la copertura delle query vengono misurate solo sulle query del carico di lavoro effettivamente interessate dalle rimozioni delle mappature.

Dataspace	Mapping rimossi	Densità delle query	Copertura delle query	Affidabilità dell'aggregazione	Supporto alla selezione
$\mathcal{D}_1$	$a_{26} \not\equiv a_{27}$	70.3%	100%	si	parziale
$\mathcal{D}_2$	$a_{32} \not\equiv a_{33}$	80.0%	100%	no	parziale
$\mathcal{D}_3$	$a_{28} \not\equiv a_{29}$, $a_{28} \not\equiv a_{30}$, $a_{28} \not\equiv a_{31}$	91.1%	64.3%	si	parziale

fatta una valutazione dal punto di vista di qualità del risultato, e non di performance, si considera il dataspace con fattore di ridimensionamento 1). A partire da $\mathcal{D}^*$ vengono considerati tre scenari diversi, ognuno rappresentante un dataspace differente ($\mathcal{D}_1$, $\mathcal{D}_2$, $\mathcal{D}_3$), dove diverse tipologie di mapping vengono rimosse. La Tabella 5.7 mostra le caratteristiche di ogni scenario e misura la degradazione della qualità media delle query che compongono il carico di lavoro (definite nella Tabella 5.4) che sono influenzate dalla rimozione dei mapping. Ispirandosi a [52], la Tabella 5.7 valuta le seguenti caratteristiche.

- *Densità delle query* come percentuale di celle non nulle nei risultati della query.
- *Copertura delle query* come percentuale di record considerati dalla query rispetto a quelli considerati in $\mathcal{D}^*$.
- *Affidabilità dell'aggregazione*, ovvero se l'aggregazione dei risultati parziali in cui mancano i mapping è coerente con i risultati ottenuti in $\mathcal{D}^*$.
- *Supporto alla selezione*, cioè se l'assenza di mappatura ostacola la capacità di applicare predicati di selezione.

Di seguito verranno descritte le valutazioni effettuate per ogni variazione di dataspace considerata.

Eliminazione di mapping tra attributi semplici. Nel dataspace $\mathcal{D}_1$ viene considerata una mancanza di mapping tra attributi della stessa entità. Nello specifico si considera la mancanza del mapping tra le date degli ordini. L'assenza di questo mapping comporta la creazione due feature che rappresentano la data dell'ordine, una per ognuno dei due attributi ($f_9^{'}$ in C_2 e $f_9^{''}$ in C_4). Per interrogare questo dataspace e ricavare le date degli ordini è, quindi, necessario richiedere di proiettare ambedue le feature create. Lo scenario descritto ha un impatto in termini di densità delle query.

- In caso di proiezioni, ognuna delle due feature restituirà valori nulli per ogni record in cui l'attributo a cui essa si riferisce non è definito (in modo approssimativo questo accadrà al 50% dei record, in modo simmetrico, restituiti da un'interrogazione su una delle due feature generate). Le densità delle query nella Tabella 5.7 sono maggiori a causa della proiezione di altre feature senza valori nulli.

- In caso di selezioni, bisognerà formulare manualmente la disgiunzione dei predicati che sarebbe generata automaticamente nel caso in cui il mapping sia presente. Questo nella implementazione corrente non è possibile, poiché i predicati di selezione all'interno di un'interrogazione vengono considerati in congiunzione e la disgiunzione di predicati di selezione non è supportata.

Eliminazione di mapping tra attributi semplici con record sovrapposti. Il dataspace $\mathcal{D}_2$ considera lo stesso scenario proposto in $\mathcal{D}_1$ ma il mapping che viene rimosso, in questo caso, riguarda attributi di collezioni con record sovrapposti. In tal caso, infatti, viene rimosso il mapping che riconcilia gli attributi che rappresentano il cognome del cliente (LastName). Oltre la considerazione fatta per $\mathcal{D}_1$, ovvero la generazione di due feature distinte che rappresentano il cognome del cliente ($f_{11}^{'}$ in C_1 e $f_{11}^{''}$ in C_4), viene introdotto un problema in termini di affidabilità del risultato delle interro-

Tabella 5.8: L'assenza di un mapping tra i due attributi che rappresentano il cognome del cliente (LastName) in $\mathcal{D}_2$ non rende possibile l'applicazione della funzione di risoluzione dei conflitti tra record sovrapposti e porta a risultati incoerenti, in quanto le somme dei due risultati parziali in $\mathcal{D}_2$ non sempre corrispondono ai risultati reali in $\mathcal{D}^*$.

$\mathcal{D}_2$				$\mathcal{D}^*$	
$f_{11}^{'}$	$\{f_8, sum()\}$	$f_{11}^{''}$	$\{f_8, sum()\}$	f_{11}	$\{f_8, sum()\}$
Faye	201542,1	Faye	194213,3	Faye	366440,2
Baloch	178805,2	Baloch	197430,7	Baloch	372510,7
Francois	54354,3			Francois	54354,3
Alschitz	11082,4	Alschitz	9030,1	Alschitz	20523,0
		Guelleh	67471,7	Guelleh	67471,7
		Akongo	186595,7	Akongo	186595,7
Nagy	118644,1	Nagy	136006,7	Nagy	289375,9

gazioni. Infatti, mentre in $\mathcal{D}^*$ quando ci sono record sovrapposti la funzione di risoluzione dei conflitti ($\mathbb{M}$) entra in gioco, in questo caso, avendo due feature distinte, le interrogazioni dovranno essere effettuate su queste feature in modo separato. In tal caso non ci sarà la possibilità di unire facilmente i risultati ottenuti ed applicare la risoluzione dei conflitti. Ad esempio, un estratto delle interrogazioni che raccolgono la somma dei TotalPrice (prezzo totale dell'ordine) raggruppando per LastName è mostrato nella Tabella 5.8. Vengono formulate due interrogazioni in $\mathcal{D}_2$, considerando le due feature generate, e una sola query considerando la feature LastName in $\mathcal{D}^*$. Nel caso in cui non ci fosse stata sovrapposizione tra i record, il risultato dell'interrogazione in $\mathcal{D}^*$ sarebbe stato uguale alla somma dei risultati per ogni gruppo ottenuto nelle due query formulate in $\mathcal{D}_2$. Nel caso di record sovrapposti questo non è sempre vero: la funzione $\mathbb{M}$ risolve prima i conflitti tra gli attributi nella feature LastName e, solo successivamente, viene effettuata l'aggregazione. In particolare, in questo caso sommando i risultati delle query su $\mathcal{D}_2$ si otterrebbe una variazione del $\pm 111\%$ rispetto ai risultati in $\mathcal{D}^*$.

Eliminazione di mapping tra attributi chiave. Il dataspace $\mathcal{D}_3$ considera una mancanza di mapping che coinvolgono attributi chiave. In particolare, non viene esplicitata la corrispondenza tra le due chiavi primarie

delle collezioni che rappresentano i clienti ($a_{28} \not\equiv a_{29}$) e una delle due chiavi non è in alcun modo collegata all'entità degli ordini ($a_{28} \not\equiv a_{30}$ e $a_{28} \not\equiv a_{31}$). La mancanza di questi mapping comporta: oltre che come in $\mathcal{D}_1$ e $\mathcal{D}_2$, vengono create due feature rappresentanti la chiave primaria del cliente (f'_{10} e f''_{10}), la creazione di due entità che rappresentano il cliente e solo una di queste due entità è collegata all'entità degli ordini. L'impatto principale di questo scenario sui risultati delle interrogazioni è in termini di copertura, ciò significa che i record del cliente non possono essere interrogati nella loro interezza. In particolare:

- Una query che coinvolge le feature di una delle due entità restituirà solo un numero selezionato di record: le query avranno per lo più la densità completa, ma la copertura diminuirà in modo significativo. Infatti, selezionando una tra le due entità dei clienti create, verrà recuperato il 60% di clienti (ricordando che il 20% di clienti è sovrapposto tra le sorgenti).

- Non è possibile rispondere a una query che coinvolge feature appartenenti ad entrambe le entità perché esse non sono collegate in alcun modo nell'entity graph di $\mathcal{D}_3$.

Conclusioni e sviluppi futuri

In questa tesi sono stati sviluppati un modello e un prototipo per l'interrogazione di multistore che considera l'eterogeneità a livello di modello dati (supportando i modelli dati documentale, relazionale e wide-column) e a livello di schema (supportando attributi mancanti, attributi che rappresentano lo stesso concetto diverso nome e/o tipo, record sovrapposti). I dati presenti nei database sorgenti sono stati integrati attraverso un approccio leggero e incrementale (chiamato anche *pay-as-you-go*) che permette di superare la rigidità dei tradizionali metodi di integrazione e fornire risultati best-effort o approssimativi. Il risultato dell'integrazione è costituito da un *dataspace* [23], ovvero una rappresentazione logica ed ad alto livello dei dataset disponibili che viene creata attraverso il processo di integrazione integrato, iterativo, incrementale e leggero, seguendo la filosofia pay-as-you-go. Si è quindi formalizzata la modalità di interrogazione del dataspace offrendo l'espressività della classe di query GSPJ, ovvero quella più utilizzata nelle applicazioni OLAP. Insieme a ciò, si è definito il processo di traduzione di un'interrogazione nel dataspace in una serie di interrogazioni nelle sorgenti, ricorrendo all'uso di un middleware per poter integrare ed elaborare i risultati parziali. Per poter mettere in azione quanto formalizzato, è stato sviluppato un prototipo che lavora su un multistore che considera tutti gli aspetti di eterogeneità considerati nella formalizzazione, sia a livello di modello dati che di schema. I test svolti sopra al prototipo mirano a valutare l'approccio sviluppato, sia in termini di efficienza che di efficacia. In particolare, per quanto riguarda l'efficienza si sono valutate caratteristiche di performance

dell'approccio in uno scenario di interrogazioni variegato, considerando le tempistiche per ottenere i risultati, rapportando quanta computazione viene eseguita lato sorgenti e quanta lato middleware e come le ottimizzazioni del piano di esecuzione inferiscono sulle tempistiche. Per quanto riguarda l'efficacia, invece, si è mostrato come aumentando le conoscenze vengono fornite nel dataspace incrementa la correttezza e l'affidabilità dei risultati. In generale, risultati ottenuti giustificano e danno spiegazioni coerenti rispetto all'approccio utilizzato.

Allo stato attuale, l'elaborato di tesi offre diversi spunti per ulteriori sviluppi. In termini di espressività si potrebbe valutare di supportare una semantica per le interrogazioni più espressiva di quella attuale e, inoltre, di estendere il multistore ai modelli dati a grafo e chiave-valore. In termini di ottimizzazione, sarebbe interessante costruire piani di esecuzione ottimizzati e, mediante un modello di costo, valutarli. Si potrebbe pensare di estrarre ed utilizzare statistiche relative agli attributi e sfruttare tali informazioni per costruire piani di esecuzione ottimizzati. Oltre alle ottimizzazioni intrinseche del piano di esecuzione che viene attualmente creato (push down dei predicati, raggruppamento dei piani di esecuzione a livello di schema, riordinamento delle sequenze di fusione e column pruning), sarebbe interessante evitare di accedere alla stessa collezione più volte nel caso essa abbia entità differenti al suo interno. In tal caso, la logica attuale, che si propone di integrare i dati prima a livello di collezione e poi di entità, verrebbe sostituita da una logica differente. Il piano di esecuzione attuale garantisce una correttezza dei risultati, bisognerà capire, valutare e quantificare quanto saranno corretti i risultati ottenuti con un piano di esecuzione del genere. Banalmente, si potrebbe considerare di svolgere una serie di computazioni locali sulle sorgenti che verranno fuse e ordinate alla fine, ma è evidente che non si otterrebbe un risultato uguale a quello attuale: i record sovrapposti tra più sorgenti non verranno considerati in modo corretto. Si sottolinea che l'architettura del prototipo sviluppato risulta facilmente estendibile e migliorabile, per cui si ritiene che sarà possibile integrare agilmente quanto sopra menzionato.

Bibliografia

[1] Azza Abouzeid et al. «HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads». In: *Proceedings of the VLDB Endowment* 2.1 (2009), pp. 922–933.

[2] Sihem Amer-Yahia, Fang Du e Juliana Freire. «A comprehensive solution to the XML-to-relational mapping problem». In: *Proceedings of the 6th annual ACM international workshop on Web information and data management*. ACM. 2004, pp. 31–38.

[3] Michael Armbrust et al. «Spark sql: Relational data processing in spark». In: *Proceedings of the 2015 ACM SIGMOD*. ACM. 2015, pp. 1383–1394.

[4] Mohamed-Amine Baazizi et al. «Schema inference for massive json datasets». In: *(EDBT)*. 2017, pp. 222–233.

[5] Abdelghani Bakhtouchi. «Data reconciliation and fusion methods: A survey». In: *Applied Computing and Informatics* (2019). ISSN: 2210-8327. DOI: https://doi.org/10.1016/j.aci.2019.07.001. URL: http://www.sciencedirect.com/science/article/pii/S2210832718303168.

[6] Hamdi Ben Hamadou, Enrico Gallinucci e Matteo Golfarelli. «Answering GPSJ Queries in a Polystore: A Dataspace-Based Approach». In: *Conceptual Modeling*. A cura di Alberto H. F. Laender et al. Cham: Springer International Publishing, 2019, pp. 189–203. ISBN: 978-3-030-33223-5.

[7] Hamdi Ben Hamadou et al. «Towards Schema-independent Querying on Document Data Stores». In: *20th Int. Workshop on Design, Optimization, Languages and Analytical Processing of Big Data co-located with EDBT/ICDT*. CEUR-WS.org, 2018.

[8] Domenico Beneventano et al. «Entity Resolution and Data Fusion: An Integrated Approach». In: *Proceedings of the 27th Italian Symposium on Advanced Database Systems, Castiglione della Pescaia (Grosseto), Italy, June 16-19, 2019*. A cura di Massimo Mecella, Giuseppe Amato e Claudio Gennaro. Vol. 2400. CEUR Workshop Proceedings. CEUR-WS.org, 2019. URL: http://ceur-ws.org/Vol-2400/paper-17.pdf.

[9] Sonia Bergamaschi et al. «From Data Integration to Big Data Integration». In: *A Comprehensive Guide Through the Italian Database Research Over the Last 25 Years*. A cura di Sergio Flesca et al. Cham: Springer International Publishing, 2018, pp. 43–59. ISBN: 978-3-319-61893-7. DOI: 10.1007/978-3-319-61893-7_3. URL: https://doi.org/10.1007/978-3-319-61893-7_3.

[10] Philip A Bernstein, Jayant Madhavan e Erhard Rahm. «Generic schema matching, ten years later». In: *Proc. VLDB Endowment* 4.11 (2011), pp. 695–701.

[11] Jens Bleiholder e Felix Naumann. «Data Fusion». In: 41.1 (2009). ISSN: 0360-0300. DOI: 10.1145/1456650.1456651. URL: https://doi.org/10.1145/1456650.1456651.

[12] Jens Bleiholder e Felix Naumann. «Declarative Data Fusion - Syntax, Semantics, and Implementation». In: *Advances in Databases and Information Systems, 9th East European Conference, ADBIS 2005, Tallinn, Estonia, September 12-15, 2005, Proceedings*. A cura di Johann Eder et al. Vol. 3631. Lecture Notes in Computer Science. Springer, 2005, pp. 58–73.

[13] Timo Böhme e Erhard Rahm. «Supporting Efficient Streaming and Insertion of XML Data in RDBMS.» In: *DIWeb*. 2004, pp. 70–81.

[14] R. Bonaque et al. «Mixed-instance querying: a lightweight integration architecture for data journalism». In: *Proceedings of the VLDB Endowment* 9 (set. 2016), pp. 1513–1516. DOI: 10.14778/3007263.3007297.

[15] Francesca Bugiotti e et al. «Invisible Glue: Scalable Self-Tunning Multi-Stores». In: *7th Biennial Conf. on Innovative Data Systems Research*. www.cidrdb.org, 2015.

[16] Jeff Carpenter e Eben Hewitt. *Cassandra: the definitive guide*. O'Reilly, 2016.

[17] Šejla Čebirić, François Goasdoué e Ioana Manolescu. «Query-oriented summarization of RDF graphs». In: *Proceedings of the VLDB Endowment* 8.12 (2015), pp. 2012–2015.

[18] Kristina Chodorow. *MongoDB: The Definitive Guide*. O'Reilly Media, Inc., 2013. ISBN: 1449344682.

[19] Alejandro Corbellini et al. «Persisting big-data: The NoSQL landscape». In: *Inf. Syst.* 63 (2017), pp. 1–23. DOI: 10.1016/j.is.2016.07.009.

[20] Michael DiScala e Daniel J. Abadi. «Automatic Generation of Normalized Relational Schemas from Nested Key-Value Data». In: *2016 ACM SIGMOD Int. Conf. on Management of Data*. ACM, 2016, pp. 295–310. DOI: 10.1145/2882903.2882924.

[21] Xin Dong et al. «From Data Fusion to Knowledge Fusion». In: *Proceedings of the VLDB Endowment* 7 (mar. 2015). DOI: 10.14778/2732951.2732962.

[22] Daniela Florescu e Donald Kossmann. «Storing and querying XML data using an RDMBS». In: *IEEE data engineering bulletin* 22 (1999), p. 3.

[23] Michael J. Franklin, Alon Y. Halevy e David Maier. «From databases to dataspaces: a new abstraction for information management». In: *SIGMOD Record* 34.4 (2005), pp. 27–33. DOI: 10.1145/1107499.1107502.

[24] Dominik D Freydenberger e Timo Kötzing. «Fast learning of restricted regular expressions and DTDs». In: *Theory of Computing Systems* 57.4 (2015), pp. 1114–1158.

[25] Vijay Gadepally e et al. «The BigDAWG polystore system and architecture». In: *2016 IEEE High Performance Extreme Computing Conf.* IEEE, 2016, pp. 1–6. DOI: 10.1109/HPEC.2016.7761636.

[26] Enrico Gallinucci, Matteo Golfarelli e Stefano Rizzi. «Approximate OLAP of document-oriented databases: A variety-aware approach». In: *Inf. Syst.* (2019). In press.

[27] Enrico Gallinucci, Matteo Golfarelli e Stefano Rizzi. «Schema profiling of document-oriented databases». In: *Inf. Syst.* 75 (2018), pp. 13–25.

[28] Enrico Gallinucci, Matteo Golfarelli e Stefano Rizzi. «Variety-Aware OLAP of Document-Oriented Databases». In: *20th Int. Workshop on Design, Optimization, Languages and Analytical Processing of Big Data co-located with EDBT/ICDT*. CEUR-WS.org, 2018.

[29] Corinna Giebler et al. «Leveraging the Data Lake: Current State and Challenges». In: *Proc. DaWaK*. Linz, Austria, 2019, pp. 179–188.

[30] Matteo Golfarelli e et al. «OLAP query reformulation in peer-to-peer data warehousing». In: *Inf. Syst.* 37.5 (2012), pp. 393–411. DOI: 10.1016/j.is.2011.06.003.

[31] Sergio Greco, Luigi Pontieri e Ester Zumpano. «Integrating and Managing Conflicting Data». In: *Perspectives of System Informatics, 4th International Andrei Ershov Memorial Conference, PSI 2001, Akademgorodok, Novosibirsk, Russia, July 2-6, 2001, Revised Papers*. A cura di Dines Bjørner, Manfred Broy e Alexandre V. Zamulin. Vol. 2244. Lecture Notes in Computer Science. Springer, 2001, pp. 349–362.

[32] Ashish Gupta, Venky Harinarayan e Dallan Quass. «Aggregate-Query Processing in Data Warehousing Environments». In: *21th Int. Conf. on Very Large Data Bases*. Morgan Kaufmann, 1995, pp. 358–369.

[33] Bradley Hall e Mark Lunetta. *Object relational database management system*. US Patent App. 10/122,088. Nov. 2003.

[34] Jing Han et al. «Survey on NoSQL database». In: *2011 6th international conference on pervasive computing and applications*. IEEE. 2011, pp. 363–366.

[35] Michael Hausenblas e Jacques Nadeau. «Apache Drill: Interactive Ad-Hoc Analysis at Scale». In: *Big Data* 1 (giu. 2013), pp. 100–104. DOI: 10.1089/big.2013.0011.

[36] Victor Herrero, Alberto Abelló e Oscar Romero. «NOSQL Design for Analytical Workloads: Variability Matters». In: *35th Int. Conf. on Conceptual Modeling*. 2016, pp. 50–64. DOI: 10.1007/978-3-319-46397-1_4.

[37] Stratos Idreos et al. «MonetDB: Two Decades of Research in Column-oriented Database Architectures». In: *IEEE Data Eng. Bull.* 35.1 (2012), pp. 40–45.

[38] Shawn Jeffery, Michael Franklin e Alon Halevy. «Pay-as-you-go User Feedback for Dataspace Systems». In: gen. 2008, pp. 847–860. DOI: 10.1145/1376616.1376701.

[39] Lan Jiang e Felix Naumann. «Holistic primary key and foreign key detection». In: *Journal of Intelligent Information Systems* 54.3 (giu. 2020), pp. 439–461. ISSN: 1573-7675. DOI: 10.1007/s10844-019-00562-z. URL: https://doi.org/10.1007/s10844-019-00562-z.

[40] K. Kaur e R. Rani. «Modeling and querying data in NoSQL databases». In: *2013 IEEE International Conference on Big Data*. 2013, pp. 1–7.

[41] J. Kepner et al. «Dynamic distributed dimensional data model (D4M) database and computation system». In: *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2012, pp. 5349–5352.

[42] Rob Kitchin e Gavin McArdle. «What makes Big Data, Big Data? Exploring the ontological characteristics of 26 datasets». In: *Big Data & Society* 3.1 (2016), p. 2053951716631130. DOI: 10.1177/2053951716631130. eprint: https://doi.org/10.1177/2053951716631130. URL: https://doi.org/10.1177/2053951716631130.

[43] Jeff LeFevre e et al. «MISO: souping up big data query processing with a multistore system». In: *2014 ACM SIGMOD Int. Conf. on Management of Data*. ACM, 2014, pp. 1591–1602. DOI: 10.1145/2588555.2588568.

[44] Harold Lim, Yuzhang Han e Shivnath Babu. «How to Fit when No One Size Fits.» In: *CIDR*. Vol. 4. Citeseer. 2013, p. 35.

[45] Chunbin Lin, Jianguo Wang e Chuitian Rong. «Towards heterogeneous keyword search». In: *Proceedings of the ACM Turing 50th Celebration Conference-China*. ACM. 2017, p. 46.

[46] Y. Lin et al. «Efficient Entity Resolution on Heterogeneous Records». In: *IEEE Transactions on Knowledge and Data Engineering* 32.5 (2020), pp. 912–926.

[47] Federica Mandreoli e Manuela Montangero. «Chapter 9 - Dealing With Data Heterogeneity in a Data Fusion Perspective: Models, Methodologies, and Algorithms». In: *Data Fusion Methodology and Applications*. A cura di Marina Cocchi. Vol. 31. Data Handling in Science and Technology. Elsevier, 2019, pp. 235–270. DOI: https://doi.org/10.1016/B978-0-444-63984-4.00009-0. URL: http://www.sciencedirect.com/science/article/pii/B9780444639844000090.

[48] Sabine Maßmann et al. «Evolution of the COMA match system». In: *Proceedings of the 6th International Workshop on Ontology Matching, Bonn, Germany, October 24, 2011*. 2011.

[49] Donald Miner e Adam Shook. *MapReduce Design Patterns: Building Effective Algorithms and Analytics for Hadoop and Other Systems*. 1st. O'Reilly Media, Inc., 2012. ISBN: 1449327176.

[50] Mark Lukas Möller et al. «Query Rewriting for Continuously Evolving NoSQL Databases». In: *International Conference on Conceptual Modeling*. Springer. 2019, pp. 213–221.

[51] Bruce Momjian. *PostgreSQL: introduction and concepts*. Vol. 192. Addison-Wesley New York, 2001.

[52] Felix Naumann, Johann Christoph Freytag e Ulf Leser. «Completeness of integrated information sources». In: *Inf. Syst.* 29.7 (2004), pp. 583–615.

[53] Martin Odersky, Lex Spoon e Bill Venners. *Programming in Scala*. Artima, 2008.

[54] Kian Win Ong, Yannis Papakonstantinou e Romain Vernoux. «The SQL++ Semi-structured Data Model and Query Language: A Capabilities Survey of SQL-on-Hadoop, NoSQL and NewSQL Databases». In: *CoRR* abs/1405.3631 (2014).

[55] M. Tamer Özsu e Patrick Valduriez. «Big Data Processing». In: *Principles of Distributed Database Systems*. Cham: Springer International Publishing, 2020, pp. 449–518. ISBN: 978-3-030-26253-2. DOI: 10.1007/978-3-030-26253-2_10. URL: https://doi.org/10.1007/978-3-030-26253-2_10.

[56] Yannis Papakonstantinou e Vasilis Vassalos. «Query rewriting for semistructured data». In: *ACM SIGMOD Record*. Vol. 28. ACM. 1999, pp. 455–466.

[57] Daniel Rolls, Carl Joslin e Sven-Bodo Scholz. «Unibench: A Tool for Automated and Collaborative Benchmarking». In: *18th IEEE Int. Conf. on Program Comprehension*. IEEE Computer Society, 2010, pp. 50–51.

[58] Diego Sevilla Ruiz, Severino Feliciano Morales e Jesús Garcıéa Molina. «Inferring Versioned Schemas from NoSQL Databases and Its Applications». In: *Proc. ER*. 2015, pp. 467–480.

[59] Pramod J Sadalage e Martin Fowler. *NoSQL distilled: a brief guide to the emerging world of polyglot persistence*. Pearson Education, 2013.

[60] Amit P. Sheth. «Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases». In: *17th Int. Conf. on Very Large Data Bases*. Morgan Kaufmann, 1991, p. 489.

[61] Abraham Silberschatz, Henry F. Korth e S. Sudarshan. *Database system concepts*. 6ª ed. New York: McGraw-Hill, 2010. URL: http://www.db-book.com/.

[62] Apache Spark. *Spark Overview*. URL: https://spark.apache.org/.

[63] Michael Steinbrunn, Guido Moerkotte e Alfons Kemper. «Heuristic and Randomized Optimization for the Join Ordering Problem». In: *VLDB J.* 6.3 (1997), pp. 191–208.

[64] Daniel Tahara, Thaddeus Diamond e Daniel J. Abadi. «Sinew: a SQL system for multi-structured data». In: *2014 ACM SIGMOD Int. Conf. on Management of Data*. ACM, 2014, pp. 815–826. DOI: 10.1145/2588555.2612183.

[65] John R. Talburt. *Entity Resolution and Information Quality*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010. ISBN: 0123819725.

[66] Ran Tan e et al. «Enabling query processing across heterogeneous data models: A survey». In: *2017 IEEE Int. Conf. on Big Data*. IEEE Computer Society, 2017, pp. 3211–3220. DOI: 10.1109/BigData.2017.8258302.

[67] Serguei Tarassov. «Classification of data models in DBMS». In: (mag. 2017).

[68] Ignacio G. Terrizzano et al. «Data Wrangling: The Challenging Journey from the Wild to the Lake». In: *Proc. CIDR*. Asilomar, CA, USA, 2015.

[69] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I*. Vol. 14. Principles of computer science series. Computer Science Press, 1988. ISBN: 0-7167-8069-0. URL: http://www.worldcat.org/oclc/310956623.

[70] Lanjun Wang e et al. «Schema Management for Document Stores». In: *PVLDB* 8.9 (2015), pp. 922–933. DOI: 10.14778/2777598.2777601.

[71] Tom White. *Hadoop: The Definitive Guide*. 1st. O'Reilly Media, Inc., 2009. ISBN: 0596521979.

[72] Yu Xu, Pekka Kostamaa e Like Gao. «Integrating hadoop and parallel dbms». In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 2010, pp. 969–974.

[73] M. Zhu e T. Risch. «Querying combined cloud-based and relational databases». In: *2011 International Conference on Cloud and Service Computing*. 2011, pp. 330–335.