ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Ingegneria e Architettura
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

ANALISI DELLE STRATEGIE DI MODELLAZIONE DEI DATI SU DATABASE NOSQL

Tesi di laurea in
Big Data

Relatore
Dott. Enrico Gallinucci

Candidato
Riccardo Salvatori

Terza Sessione di Laurea
Anno Accademico 2019-2020

Sommario

Negli ultimi anni, i sistemi NoSQL si sono rivelati una soluzione efficace alle stringenti esigenze del web e dei sistemi di big data, grazie alle caratteristiche di scalabilità e flessibilità che superano le limitazioni imposte dai classici RDBMS. La crescente popolarità è accompagnata però dalla mancanza di metodologie generali ed efficaci per la modellazione di dati, che permettano di sfruttare al meglio le possibilità offerte dal mondo non relazionale. Infatti, a differenza dei database relazionali, per cui esiste una vasta letteratura sul tema, molte tecniche di progettazione in ambito NoSQL sono proposte sotto forma di linee guida e best practice, per lo più dettate dall'esperienza, che spesso vanno adattate al carico di lavoro e alla specifica implementazione. In questa tesi sono state analizzate le diverse strategie di modellazione in ambito NoSQL, evidenziando il ruolo fondamentale che hanno nel determinare le prestazioni finali del database. L'analisi è stata effettuata utilizzando il benchmark TPC-C, su due popolari database NoSQL: MongoDB e Cassandra. Inoltre, è stato progettato un indicatore che permette di avere un'approssimazione a priori delle prestazioni di una de-normalizzazione. L'indicatore si è dimostrato efficace per i database MongoDB e Cassandra nell'individuare le modellazioni dei dati ottimali per il carico di lavoro previsto.

Alla mia famiglia.

Ringraziamenti

In primo luogo vorrei ringraziare il Dott. Enrico Gallinucci per avermi seguito nello sviluppo della tesi, dedicandomi il suo tempo e indicandomi la giusta direzione per portare a compimento questo elaborato.

Grazie ai miei compagni di corso, con cui ho condiviso la passione e la fatica nel raggiungere questo importante obiettivo e con i quali ho passato bellissimi momenti di studio e divertimento.

Ringrazio i miei amici più cari, che hanno riempito le giornate di gioia e spensieratezza, facendomi sentire il loro affetto anche nella lontananza.

Un ringraziamento speciale alla mia fidanzata, Martina, che con dolce entusiasmo mi ha sostenuto, senza mai farmi mancare il suo amore e rendendo speciale ogni momento.

Ringrazio mia madre Cinzia, mio padre Franco e mio fratello Lorenzo, perché con grande amore mi hanno incoraggiato nello studio e con parole di saggezza accompagnato nella vita, permettendomi di essere ciò che sono. Con orgoglio dedico a loro questo traguardo per me così importante, consapevole che non sarebbe stato possibile senza averli avuti al mio fianco.

Infine, voglio ringraziare mio fratello Stefano, che nel condividere con me questo percorso mi ha dato la sicurezza per affrontare le difficoltà e la forza per superarle, rendendo ogni successo straordinario e indimenticabile.

Indice

Elenco delle figure

Introduzione

I database NoSQL hanno guadagnato nell'ultimo periodo un'enorme popolarità, grazie anche alle loro caratteristiche così diverse dai classici RDBMS relazionali. Flessibilità , scalabilità e prestazioni elevate, hanno spinto molte aziende ad aderire al movimento NoSQL, favorendone così un rapido e ampio sviluppo. L'elasticità sulla gestione dei dati si accompagna generalmente ad una rappresentazione limitata di questi, tanto da poter definire alcuni database "schemaless" (senza schema). Un database schemaless è definito tale, poiché permette di memorizzare assieme informazioni eterogenee, senza la necessità di descriverne la struttura a priori. Se il concetto di modellazione è strettamente correlato a quello di schema, tale proprietà potrebbe far pensare che processi di design e modellazione dei dati siano secondari, o addirittura superflui in un contesto NoSQL. Come indicato anche da diversi autori in letteratura [6, 7], la fase di design assume invece un ruolo fondamentale nella progettazione di un database NoSQL e richiede un approccio totalmente diverso da quello utilizzato per i classici RDBMS. A differenza dei database relazionali, per cui esiste una vasta letteratura sul tema, molte tecniche di progettazione in ambito NoSQL sono proposte sotto forma di linee guida e best practice, per lo più dettate dall'esperienza, che spesso vanno adattate al carico di lavoro e alla specifica implementazione [2].

L'obiettivo di questa tesi é quello di analizzare le strategie di modellazione dei dati in ambito NoSQL, prendendo come caso di studio due popolari DBMS NoSQL: MongoDB e Cassandra, principali esponenti delle famiglie di database documentale e wide-column. A questo proposito è stata sviluppata un'implementazione del benchmark TPC-C per valutare come diverse modellazioni non relazionali del dataset influenzassero l'esecuzione del workload. Sulla base dei risultati è stata effettuata un'analisi delle modellazioni, discutendone pregi e difetti nelle implementazioni dei due database. Nello specifico, oltre a vedere quali modellazioni avessero le migliori prestazioni, è stata fatta un'analisi di dettaglio sull'impatto delle singole de-normalizzazioni sulle transazioni. Infine, è stato definito un indicatore che, sulla base del carico di lavoro, permette di avere un'approssimazione a priori dell'efficacia di una de-normalizzazione.

La tesi verrà strutturata nel seguente modo:

- **Capitolo 1**: nella prima parte del capitolo si descriverà lo stato dell'arte dei sistemi NoSQL, con un dettaglio sui due database utilizzati nello sviluppo della tesi: MongoDB e Cassandra. La seconda parte tratterà i concetti di aggregato e di modellazioni orientata agli aggregati, presentando un innovativo strumento di modellazione astratto per database NoSQL: NoAM.

- **Capitolo 2**: in questo capitolo si descriverà il concetto di benchmark e di database benchmarking. Verranno presentati ed analizzati nel dettaglio 3 popolari benchmark per database: UniBench, YCSB e TPC-C, dal punto di vista del modello e del carico di lavoro. Infine, si spiegheranno le motivazioni che hanno portato alla scelta del TPC-C come benchmark di riferimento per la fase di analisi.

- **Capitolo 3**: in questo capitolo si entrerà nel dettaglio del carico di lavoro del TPC-C, proponendo delle varianti non relazionali alla modellazione standard del benchmark. Le transazioni che compongono il workload verranno descritte nel dettaglio e per ognuna si riporteranno le considerazioni specifiche per la loro ottimizzazione.

- **Capitolo 4**: nella prima parte del capitolo verrà descritta l'implementazione del benchmark, quindi la generazione del dataset e l'implementazione del workload. Nella seconda parta si effettuerà l'analisi dei risultati, confrontando i due database e valutando le performance delle modellazioni non relazionali proposte. Infine, si descriverà la progettazione di un indicatore per la valutazione a priori delle performance di una de-normalizzazione, validando l'efficacia attraverso i risultati ottenuti.

Capitolo 1

Stato dell'arte

In questo capitolo verranno descritti i database NoSQL, le loro caratteristiche e le differenze rispetto ai database relazionali. Verranno discusse le loro proprietà fondamentali nel contesto del teorema CAP, con particolare attenzione ai modelli di consistenza ACID e BASE. Si analizzeranno inoltre le principali famiglie di database non relazionali: chiave-valore, documentale, orientati alle colonne (wide-column) e grafo. Questa tesi si concentra sull'analisi di database documentali e orientatati alle colonne. Si è scelto quindi di utilizzare le implementazioni più popolari delle due famiglie, rispettivamente: MongoDB e Cassandra. I due database verranno quindi commentati in maniera più dettagliata.,

Nella seconda parte del capitolo si tratterà il concetto di aggregato. Si identificherà l'aggregato nell'ambito delle modellazioni non relazionali, quindi si analizzeranno pregi e difetti dei database orientati agli aggregati. Verrà mostrato e commentato un esempio di modellazione orientata agli aggregati per un sistema di e-commerce. Infine si descriverà un nuovo strumento di modellazione astratto per la modellazione su database non relazionali: NoAM (NoSQL Abstract Model).

1.1 Database NoSQL

I database *NoSQL* sono database scalabili e flessibili, nati per superare le rigidità imposte dal modello relazionale. Negli ultimi anni si sono rivelati una soluzione efficace alle stringenti esigenze del web e dei sistemi di big data, creando così un vero e proprio movimento promotore di sistemi alternativi al relazionale. Il termine NoSQL (Not Only SQL) fu utilizzato per la prima volta nel '98 da Carlo Strozzi, per indicare un RDBMS che utilizzava un linguaggio di interrogazione diverso da SQL. Il termine fu ripreso in seguito per indicare tutti quei database non relazionali che non garantiscono le classiche proprietà ACID. A tal proposito

lo stesso Carlo Strozzi, dichiarò che i sistemi NoSQL si distaccano completamente dal modello relazionale, dunque, un nome più appropriato sarebbe NoRel.

Oggi, con il termine NoSQL, si intendono database flessibili, scalabili, ad elevate prestazioni, particolarmente adatti ad applicazioni moderne (web 2.0, big data, videogiochi etc.).

- Flessibilità: i database NoSQL permettono la definizione di schemi flessibili, facilitando così lo sviluppo. Possono essere utilizzati su tipi di dati diversi, strutturati o non strutturati.

- Scalabilità: i database NoSQL sono progettati per favorire la *scalabilità orizzontale*. Invece di prediligere l'utilizzo di server potenti e costosi, si utilizzano cluster distribuiti composti da diverse macchine economiche.

- Prestazioni elevate: rinunciando ad alcune proprietà sui dati (e.g. ACID), i database NoSQL riescono a fornire prestazioni elevate, difficilmente raggiungibili con i modelli relazionali. Sfruttando l'architettura distribuita e algoritmi ad-hoc, forniscono la cosiddetta *high availability*, ovvero la capacità di essere sempre disponibili alle richieste, anche a fronte di guasti o malfunzionamenti di una sottoparte del sistema.

Lo sviluppo dei database NoSQL è stato vasto ed è ancora in evoluzione. Le soluzioni disponibili sono numerose e sempre più aziende stanno aderendo al movimento NoSQL, adottando totalmente o parzialmente database non relazionali.

1.1.1 Tipologie di Database NoSQL

Analizzando le diverse tecnologie, possiamo individuare quattro famiglie di database, differenziate dal modello che si utilizza per la memorizzazione del dato: chiave-valore, documentale, orientati alle colonne (wide-column) e grafo.

Chiave-valore

I database chiave-valore possono essere definiti come delle mappe, contenenti oggetti (valori) identificati da stringhe univoche (chiavi) (Figura 1.1). I valori possono contenere diverse informazioni, strutturate o non strutturate e vengono rappresentati attraverso *BLOB* (binary large object). Risultano dall'esterno come delle *black box*, sulle quali non è possibile effettuare alcun tipo di interrogazione; spesso le informazioni sullo schema vengono quindi rappresentate attraverso la nomenclatura delle chiavi. Le interrogazioni si basano su tre operazioni principali:

put Aggiunge un elemento alla collezione. Se la chiave esiste il valore viene rimpiazzato

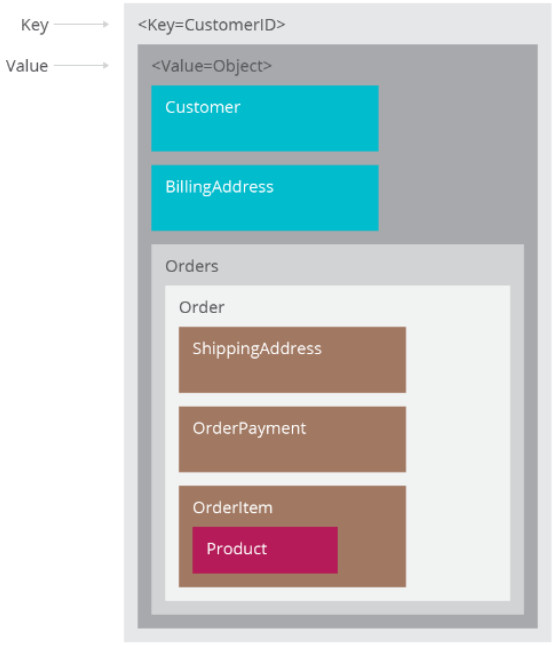


Figura 1.1: Esempio di memorizzazione chiave-valore

get Legge il valore relativo alla chiave specificata (se esiste)

delete Elimina la coppia dalla collezione

La semplicità di memorizzazione e di interrogazione rende questi database estremamente efficienti, tra i più performanti in ambito NoSQL.

Esempi di database chiave-valore:

- Riak: `http://basho.com/riak/`
- Redis: `http://redis.io/`
- Memcached DB: `http://memcached.org/`
- Amazon DynamoDB: `https://aws.amazon.com/dynamodb/`
- Project Voldemort: `http://www.project-voldemort.com/`

Documentale

Nei sistemi documentali i database sono insiemi di collezioni formate da documenti (generalmente XML o JSON). Ogni documento contiene un insieme di campi ed ha associato un id univoco. I campi sono modellati come coppie chiave-valore. La chiave è una stringa univoca nel documento, mentre il valore può essere

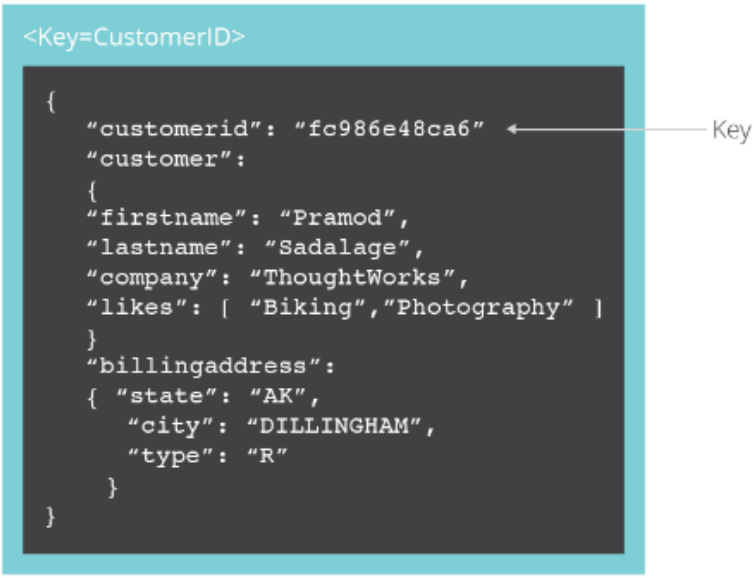


Figura 1.2: Esempio di memorizzazione orientata ai documenti

semplice (stringa, intero, booleano etc.), o complesso (array, blob, object). Un campo complesso può anche contenere campi innestati. Differentemente dal modello chiave-valore, il valore dei campi è interpretabile dal DBMS (Figura 1.2). I database documentali sono caratterizzati da un'elevata espressività di interrogazione, è possibile infatti definire indici, filtrare sui campi, creare proiezioni ed effettuare aggiornamenti a livello di singolo campo. Esempi di database orientati ai documenti:

- MongoDB: `http://www.mongodb.org`
- CouchDB: `http://couchdb.apache.org`
- Terrastore: `https://code.google.com/p/terrastore`

Wide-Column

Nei database wide-column l'astrazione fondamentale è quella di "tabella", chiamata *column family*. Ogni column family è formata da una lista di righe in formato chiave-valore. La chiave è una stringa univoca all'interno della column family, il valore un insieme di colonne (Figura 1.3). Ogni colonna è essa stessa una coppia chiave-valore. Come chiave si ha una stringa univoca nella riga, il valore può essere semplice o composto (*supercolumn*). Le column family hanno una rappresentazione dei dati molto simile a quella relazionale. La differenza principale è nella gestione

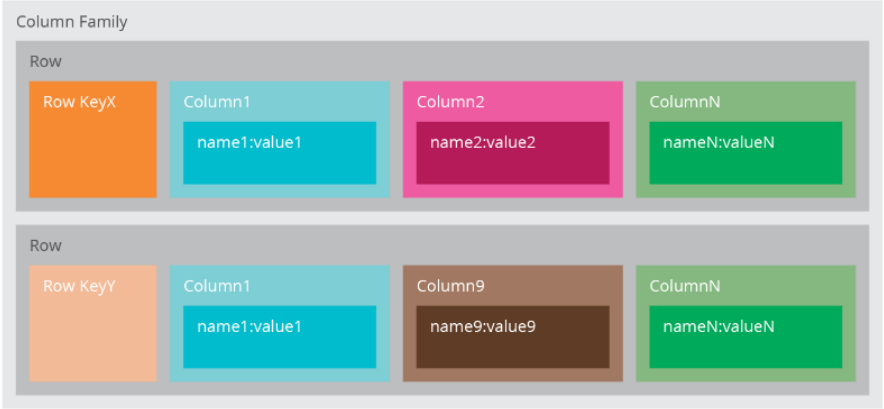


Figura 1.3: Esempio di memorizzazione column-family

dei valori mancanti. Se consideriamo un database relazionale con diversi tipi di attributi, verrà utilizzato un valore *null* per ogni colonna che non ha valori. Le righe di un database wide-column invece specificano solamente le colonne per le quali esiste un valore. Questa caratteristica rende i database wide-column ottimi per gestire matrici sparse e grandi quantità di dati con attributi diversi. L'espressività si può collocare tra quella chiave-valore e quella documentale. Solitamente, data la somiglianza con il modello relazionale, per le interrogazioni, si utilizzano linguaggi *SQL-like*.

Esempi di database wide-column:

- Cassandra: `http://cassandra.apache.org`
- HBase: `https://hbase.apache.org`
- Hypertable: `http://hypertable.org`

Grafo

I database a grafo utilizzano il grafo come principale struttura per modellare i dati. Ogni grafo contiene *vertici* e *archi* (Figura 1.4). Generalmente i vertici rappresentato entità del dominio, mentre gli archi le relazioni che intercorrono tra esse. Archi e vertici sono caratterizzati da proprietà. Per definire interrogazioni sui database a grafo si utilizzano concetti specifici come: pattern matching, adiacenza, raggiungibilità; questo li differenzia molto dai classici linguaggi di interrogazione, ma li rende più espressivi in domini in cui le relazioni hanno un ruolo più importante delle entità.

Esempi di database a grafo:

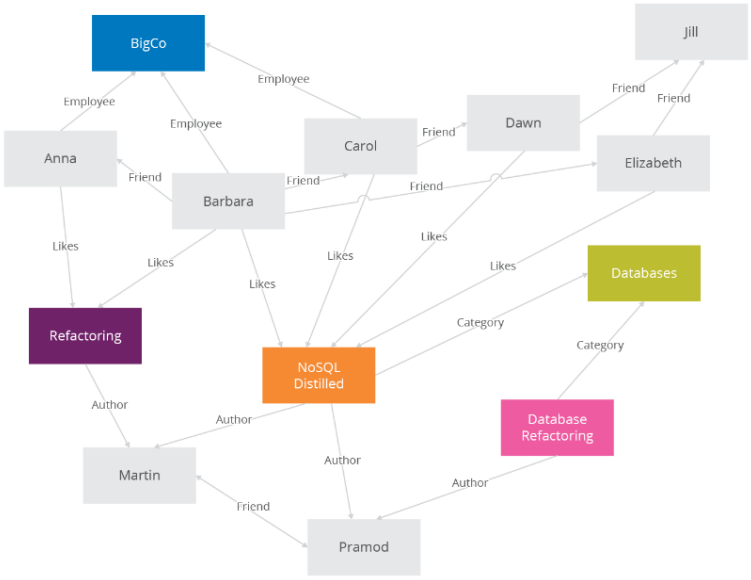


Figura 1.4: Esempio di database a grafo

- Neo4J: `https://neo4j.com/`
- InfiniteGraph: `http://www.objectivity.com/products/infinitegraph/`
- OrientDB: `http://orientdb.com/orientdb/`

1.1.2 ACID vs BASE

Le proprietà fondamentali dei database relazionali, o NoSQL, possono essere descritte utilizzando il *Teorema CAP*. Secondo questo teorema, le seguenti proprietà di un sistema distribuito, non possono essere soddisfatte simultaneamente:

- Consistenza (C) - tutti i nodi del sistema mantengono gli stessi dati nello stesso momento
- Disponibilità (A) - ogni richiesta riceve una risposta
- Tolleranza ai Partizionamenti (P) - il sistema continua a funzionare nonostante due o più nodi non possano più comunicare

Considerando il teorema CAP, ci sono quindi tre possibili soluzioni che un sistema distribuito può adottare (Figura 1.5):

- Consistenza e Disponibilità (CA)

- Consistenza e Tolleranza ai Partizionamenti (CP)
- Disponibilità e Tolleranza ai Partizionamenti (AP)

Solitamente vengono utilizzate queste osservazioni per giustificare l'utilizzo di modelli di consistenza più "deboli" rispetto al classico modello *ACID*. Con il termine ACID, si intendono quattro proprietà fondamentali che hanno le transazioni di un database:

- **A**tomicità: garantisce che una transazione venga eseguita in maniera atomica, ovvero che tale operazione non possa avere un risultato parziale.
- **C**onsistenza: capacità di non violare vincoli di integrità all'interno del database a seguito dell'esecuzione di una transazione.
- **I**solamento: garanzia che l'esecuzione di una transazione avvenga in maniera indipendente, senza subire interferenze da altre transazioni concorrenti.
- **D**urabilità (o Persistenza): impone che il risultato di una transazione sia mantenuto in maniera permanente nel database, anche a seguito di guasti o malfunzionamenti.

Le proprietà ACID sono importanti, ma garantirle può risultare complesso. Per farlo, i database relazionali mettono in campo algoritmi che si rivelano inadeguati per soddisfare le esigenze di performance e scalabilità dei sistemi moderni. Per questo motivo, negli ultimi anni, ci si è spostati verso l'utilizzo di modelli più "deboli", ma più flessibili e scalabili, come ad esempio: *BASE*, acronimo di (Basically Available Soft-state services with Eventual consistency).

- **B**asically **A**vailable: il sistema che dovrebbe essere sempre disponibile.
- **S**oft-state: il sistema può trovarsi momentaneamente in stati inconsistenti. del database a seguito dell'esecuzione di una transazione.
- **E**ventual consistency: indica un modello in cui "prima o poi" il sistema raggiungerà uno stato consistente.

Si potrebbe dire che ACID è un modello "pessimista", nel senso che in ogni momento vuole garantire la consistenza, a discapito della latenza, mentre BASE è "ottimista", nel senso che accetta stati di inconsistenza per favorire la disponibilità e le performance. Scegliere quale modello utilizzare dipende dalle proprie necessità e dal sistema che si vuole implementare. La consistenza è cruciale, ma implementarla in sistemi con centinaia di nodi è complesso, ed è raro che un database necessiti che tale proprietà sia verificata costantemente.

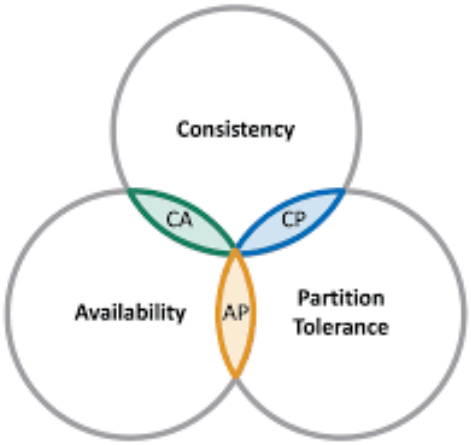


Figura 1.5: Visualizzazione del teorema CAP

1.1.3 MongoDB

MongoDB[1] è un database non relazionale orientato ai documenti. Fu sviluppato nel 2007 dalla società 10gen, che ne nel 2009 lo rende open source. Da quel momento MongoDB viene utilizzato in maniera estensiva in numerosi siti web e da importanti società di servizi come Google ed Ebay. Ad oggi è classificato come uno dei principali database NoSQL, il più popolare tra quelli documentali[2].

Un record in MongoDB è associato a un *documento*, che è una struttura dati composta da coppie chiave valore. La chiave deve essere una stringa univoca (all'interno del documento) mentre il valore può contenere oggetti semplici, array e documenti innestati. Il campo "_id" è obbligatorio. I documenti vengono salvati in formato BSON (Binary Json). Un insieme di documenti viene gestito in *collezioni*. Le collezioni hanno uno schema dinamico, nel senso che possono contenere documenti che hanno strutture diverse. Le collezioni vengono ulteriormente raggruppate in *database*.

MongoDB è un database general-purpose, dunque oltre alle operazioni basilari di read, create, update e delete, fornisce anche le seguenti funzionalità:

Indexing Gli indici in MongoDB vengono gestiti a livello di collezione e permettono l'esecuzione efficiente di query. MongoDB crea un indice di default sul campo "_id", in aggiunta lo sviluppatore può crearne di vari tipi, in base alle esigenze: single field, compound, multikey, geospatial, text, time-to-live, hashed.

[1] `https://www.mongodb.com/`
[2] `https://db-engines.com/en/ranking`

Aggregation Le operazioni di aggregazione in MongoDB permettono di processare i documenti attraverso una *pipeline* e possono essere utilizzate per creare query complesse, che verranno poi ottimizzate dal database. Le operazioni di aggregazione abilitano interrogazioni sui documenti associabili al *join* relazionale tra tabelle.

Capped collections MongoDB supporta la creazione di collezioni a dimensione fissa che una volta raggiunta la capacità massima, si comportano come una coda circolare, rimpiazzando i dati più vecchi con i nuovi aggiunti.

File storage Attraverso 'utilizzo di *GridFS*, MongoDB può essere utilizzato come file system per la gestione di file di grandi dimensioni e dei loro metadati.

Replication MongoDB permette la definizione di *replica set.*, ovvero un insieme di istanze del database che mantengono gli stessi dati. L'utilizzo della replicazione permette la ridondanza e l'elevata disponibilità in MongoDB.

Sharding Con il termine *sharding* si intende la distribuire di dati tra più nodi del cluster. Attraverso questa funzionalità MongoDB supporta la *scalabilità orizzontale*. MongoDB permette infatti di partizionare e distribuire collezioni su più macchine.

Transactions Dalla versione 4.0 MongoDB supporta (seppur con diverse limitazioni) le transazioni multi-documento con proprietà ACID.

1.1.4 Cassandra

Cassandra[3] è un database non relazionale, distribuito, di tipo wide-column. Nasce come progetto interno di Facebook. Nel 2008 viene rilasciato in maniera open-source, e nel 2009 diventa ufficialmente parte dell'Apache Software Foundation. Cassandra è classificato ad oggi tra i primi 10 database più popolari[4]. Viene utilizzato da importanti società come: Facebook, Twitter e Netflix.

Il database è progettato per lavorare con enormi volumi di dati; può infatti gestire miliardi di colonne ed eseguire milioni di operazioni al giorno. Le caratteristiche principali sono l'elevata disponibilità, la tolleranza ai guasti e la scalabilità. I pregi di Cassandra derivano principalmente dalla sua architettura distribuita. I dati vengono salvati sui *nodi*, i nodi correlati vengono raggruppati in *data center*, uno o più data center formano un cluster.

[3] `http://cassandra.apache.org/`
[4] `https://db-engines.com/en/ranking`

- Tutti i nodi in un cluster svolgono lo stesso ruolo (peer-to-peer). Ogni nodo è indipendente e allo stesso tempo interconnesso ad altri nodi.

- Ogni nodo in un cluster può accettare richieste di lettura e scrittura, indipendentemente da dove si trovano effettivamente i dati nel cluster.

- Quando un nodo si interrompe, le richieste di lettura/scrittura possono essere servite da altri nodi nella rete. I nodi difettosi vengono scoperti attraverso un processo di gossip che permette ai nodi di comunicare in background in maniera automatica.

Un'altra caratteristica fondamentale di Cassandra è quella della replicazione. I dati possono essere replicati su più nodi per assicurare disponibilità, tolleranza ai guasti e affidabilità. La *replication strategy* determina in quali nodi vengono posizionate le repliche. Il numero totale di repliche nel cluster viene definito dal *replication factor*.

I dati sono organizzati per tabelle e identificati da una chiave di partizionamento, che determina su quale nodo sono archiviati i dati e una chiave primaria che identifica univocamente la riga nella tabella. Le chiavi primarie e di partizionamento possono essere formate da una o più colonne.

Oltre al livello di replicazione è possibile definire il livello di consistenza che si vuole mantenere. Questo parametro indica il numero di nodi che devono confermare una modifica prima che questa venga resa persistente.

Lo strumento di interrogazione di Cassandra è il Cassandra Query Language (CQL). La sintassi e il modello di interrogazione ricordano il linguaggio SQL, nel senso che i dati vengono gestiti in tabelle contenenti righe di colonne.

1.2 Modellazione orientata agli aggregati

La modellazione nel paradigma relazionale utilizza concetti che si discostano molto dalle strutture dati manipolate dagli sviluppatori. Questo gap di astrazione viene riassunto nel problema dell'*impedence mismatch*. Mentre i database utilizzano il linguaggio dell'algebra relazionale (tabelle, tuple, associazioni etc.), gli sviluppatori generalmente fanno riferimento al paradigma object oriented (classi, interfacce, polimorfismo etc.). In un sistema complesso, che si appoggia su un RDBMS, è necessario quindi effettuare un mapping tra i due modelli, pagando un costo in termini di efficienza e tempo di sviluppo. I database NoSQL vogliono superare il problema dell'impedence mismatch attraverso il concetto di *aggregato*.

1.2.1 Aggregato

Il concetto di "aggregato" è ripreso dal Domain-Drive Design (DDD) ed indica un gruppo di oggetti strettamente correlati che vengono gestiti come un'unica entità. L'approccio ad aggregati riconosce che spesso si vogliono manipolare dati con strutture complesse, difficilmente catturabili utilizzando le tuple del modello relazionale. Propone dunque di rappresentare i dati come insieme di campi, liste ed oggetti innestati, riassumibili in un unica entità (aggregato), con la stessa struttura per lo sviluppatore e per l'RDBMS. L'aggregato diventa l'unità fondamentale sul quale modellare il dominio, comunicare con il database e gestire la consistenza. Molti database infatti garantiscono sugli aggregati transazioni con proprietà ACID.

I database che supportano ed incentivano l'utilizzo di aggregati, come chiave-valore, documentale o wide-column, vengono definiti "orientati agli aggregati". Il concetto di aggregato semplifica la gestione della distribuzione (replicazione e sharding) poiché informazioni correlate saranno memorizzate assieme e, se devono essere replicate, non è necessario preoccuparsi della consistenza, in quanto tutte le informazioni sono contenute all'interno dell'aggregato. L'aggregato è anche un concetto più facile da utilizzare per i programmatori che spesso, nello sviluppo, manipolano dati che mostrano strutture aggregate.

1.2.2 Modellazione degli aggregati

Non ci sono delle linee guida generali sulla modellazione orientati agli aggregati. La definizione di un aggregato è strettamente correlata all'utilizzo dei dati a livello applicativo, aspetto che spesso non si affronta durante modellazione del dominio. La struttura di un aggregato dovrebbe essere definita dai piani d'accesso sui dati che modella. Un approccio utilizzabile è ad esempio quello *query-driven*, ovvero organizzare i dati partendo dalle interrogazioni che si vogliono ottimizzare.

In generale all'interno di un aggregato dovrebbero essere memorizzate informazioni correlate, che vengono accedute assieme. Inoltre, dovrebbe contenere tutte le informazioni necessarie per rispondere alle interrogazioni che lo riguardano. (Da notare che molti database NoSQL non supportano operazioni di "join"). Per supportare consistenza e atomicità, un aggregato dovrebbe contenere tutte le informazioni che sono legate da vincoli di integrità a livello applicativo. D'altra parte, per favorire scalabilità e performance, gli aggregati dovrebbero essere mantenuti di dimensioni contenute.

Nei database orientati agli aggregati risulta più difficile rappresentare relazioni inter-aggregati rispetto a quelle intra-aggregato. Definire relazioni inter-aggregato, o utilizzare le stesse informazioni in diversi contesti, rende il confine dell'aggregato sfumato, difficile da definire. Fissare a priori una modellazione porterebbe alla semplificazione di alcune interrogazioni, ma altre risulterebbero eccessivamente

complesse e poco eleganti nella struttura. Se il confine dell'aggregato non è chiaro quindi, è più utile sfruttare la versatilità dei database non orientati agli aggregati (*aggregate-ignorant*). Fanno parte di questa famiglia i database relazionali, ma anche quelli a grafo. Un modello aggregate-ignorant non costringe a una definizione a priori della struttura dei dati, permette di gestirli in modi differenti a seconda delle necessità.

Esempio di modellazione orientata agli aggregati

La Figura 1.6 mostra il diagramma UML di un dominio di e-commerce in cui gli utenti possono effettuare ordini e pagamenti [17]. La Figura 1.7 mostra una modellazione dello stesso dominio orientata agli aggregati. La notazione UML della composizione è utilizzata per mostrare la de-normalizzazione.

In questo caso vengono definiti due aggregati: customer e order. Il customer contiene una lista di billing address, mentre order contiene una lista di order-item, shipping address e payment. Payment contiene a sua volta un billing address. Uno schema di questo tipo favorisce interrogazioni che lavorano prevalentemente su customer e order, ma che non modificano address. Avendo de-normalizzato la tabella Address infatti, le informazioni dell'indirizzo sono replicate; un'operazione di aggiornamento su address quindi sarebbe inefficiente perché dovrebbe essere propagata su tutti gli aggregati.

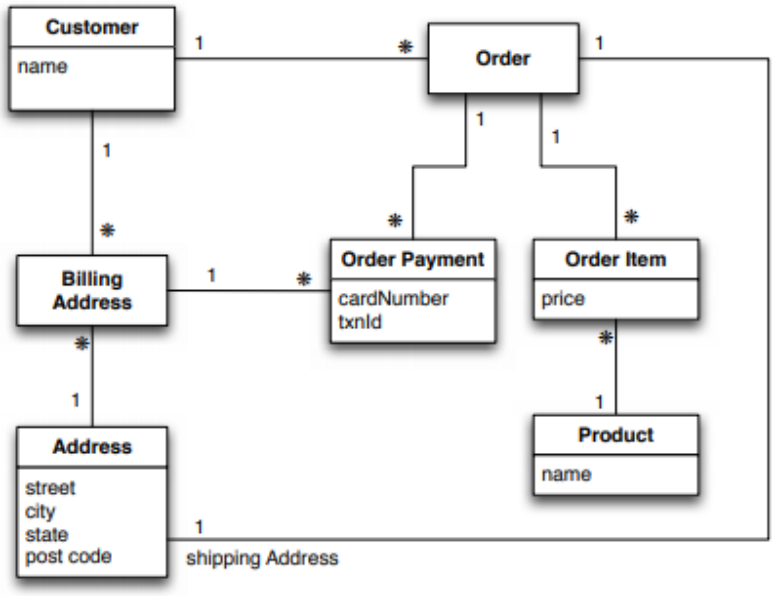


Figura 1.6: Diagramma UML di un sistema di e-commerce

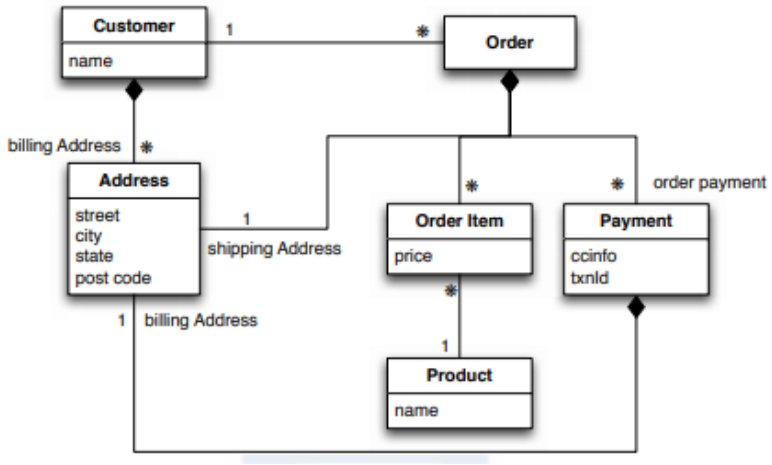


Figura 1.7: Esempio di modellazione orientata agli aggregati

Una soluzione alternativa è quella mostrata in Figura 1.8. In questo caso tutte le informazioni sugli ordini sono inserite dentro customer creando un unico grande aggregato. Questa soluzione è più efficiente se si pensa a una applicazione dove non è necessario accedere alle informazioni dei singoli ordini, ma interessano solamente le informazioni dei clienti.

Le modellazioni possibili sono varie e la scelta su quale implementare dipende interamente da come i dati vengono manipolati a livello applicativo. Se l'applicazione è focalizzata solamente sul customer, la secondo soluzione è da preferire. Se si accede alle informazioni sui singoli ordini, potremmo implementare la prima modellazione. Al contrario, se si volessero analizzare le vendite di specifici prodotti entrambe le modellazioni porterebbero a dei problemi. Per ricavare lo storico delle vendite sarebbe infatti necessario accedere a tutti gli aggregati del database. Una modellazione orientata agli aggregati favorisce quindi alcune query, rendendone altre più complesse. Se non c'è una struttura primaria dettata dall'applicazione su come manipolare i dati, la soluzione più corretta potrebbe essere quella di appoggiarsi su database aggregate-ignorant. Oltre a poter gestire i dati in maniera versatile, si otterrebbero anche tutti i vantaggi della normalizzazione.

1.2.3 NoSQL Abstract Model

Anche se i database NoSQL vengono definiti "schemaless", la modellazione dei dati copre un ruolo importante nell'ambito NoSQL e dovrebbe essere progettata per sfruttare al meglio gli strumenti offerti dai sistemi non relazionali (documenti, collezioni, coppie chiave-valore etc.). I database NoSQL offrono infatti diverse

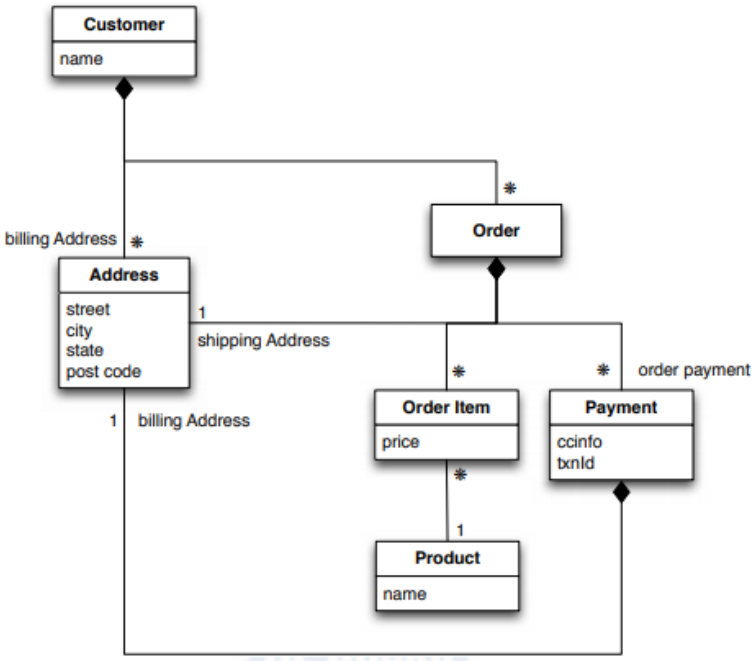


Figura 1.8: Esempio di modellazione orientata agli aggregati

possibilità per modellare le stesse informazioni, ma non tutte sono equivalenti in termini di efficienza, scalabilità e consistenza.

In questo contesto *NoAM* (NoSQL Abstract Model) è uno strumento di modellazione astratto per database NoSQL [6]. Utilizzando il modello NoAM, e la sua metodologia di progettazione, è possibile definire modellazioni di dati indipendenti da uno specifico sistema NoSQL, che siano efficienti, scalabili e consistenti.

Il modello NoAM

Il modello NoAM utilizza gli elementi comuni ai diversi sistemi NoSQL e introduce astrazioni che ne bilanciano le differenze e le variazioni. Come prima cose si osserva che tutti i database NoSQL hanno un elemento che mostra le seguenti proprietà:

- può essere acceduto e manipolato in maniera atomica, efficiente e scalabile (*data access unit*)
- può essere memorizzato su una singola macchina del cluster *data distribution unit*)

Facendo riferimento alle tipologie di database, questo elemento si traduce in: un documento, nei database documentali, un gruppo di coppie chiave-valore, nei database chiave-valore, una riga nei database wide-column. In NoAM un elemento con

queste proprietà è modellato dal concetto di *blocco*. Un *blocco* rappresenta l'entità più grande che può essere manipolata in maniera atomica, scalabile ed efficiente, e memorizzata su una singola macchina. Dunque, mentre l'accesso al singolo blocco è ottimizzato dal sistema, accedere a un gruppo di blocchi può risultare inefficiente.

Una seconda funzionalità comune a sistemi NoSQL è quella di poter manipolare i singoli componenti di un *blocco*. Per componente si intende: un campo, nei database documentali, una colonna, nei database wide-column, una coppia chiave valore, nei database chiave-valore. In NoAM questo componente di chiama *entry*.

In ultimo, la maggior parte dei database NoSQL gestisce insieme di "data access unit". In NoAM un inseme di blocchi definisce una *collezione*.

Il modello NoAM è definito dunque nel seguente modo:

- Un *database* è un insieme di *collezioni*. Ogni collezione ha un nome.
- Una *collezione* è formata da un insieme di *blocchi*. I blocchi sono identificati da una chiave.
- Il *blocco* è un insieme di *entry*. Ogni entry è definita da una coppa chiave valore.

Un esempio di modellazione NoAM è visibile in Figura 1.11b.

La metodologia NoAM

La metodologia NoAM si compone di tre attività principali:

- *Modellazione concettuale e design degli aggregati*: Per identificare le entità e le relazioni del dominio applicativo, e in seguito raggruppare le entità in aggregati.
- *Partizionamento degli aggregati e modellazione NoSQL*: In cui si raffinano gli aggregati e si mappano nei concetti di NoAM.
- *Implementazione*: Si concretizza la rappresentazione intermedia in uno specifico database NoSQL.

Modellazione concettuale e design degli aggregati La prima fase della metodologia consiste nel definire una rappresentazione concettuale del dominio. Si può ad esempio seguire l'approccio del Domain-Driven Design (DDD) e produrre come risultato uno schema UML delle classi (Figura 1.9).

Le entità principali definite nella prima fase devono essere poi raggruppate in aggregati (Figura 1.10). La scelta sulla definizione degli aggregati deve essere fatta sulla base dei piani d'accesso ai dati e sulle esigenze di consistenza. Nell'esempio in Figura 1.10 sono stati specificati due aggregati: Player e Game.

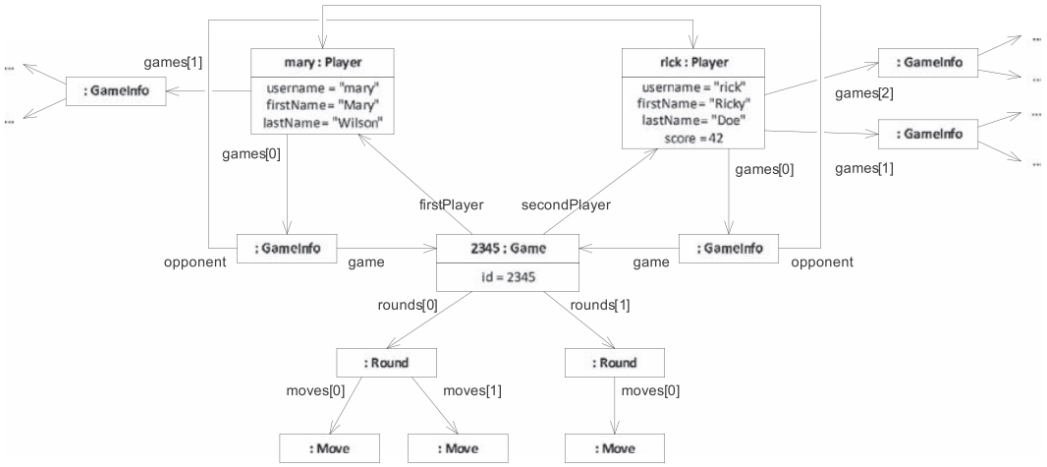


Figura 1.9: Diagramma UML delle classi per un dominio di videogiochi online.

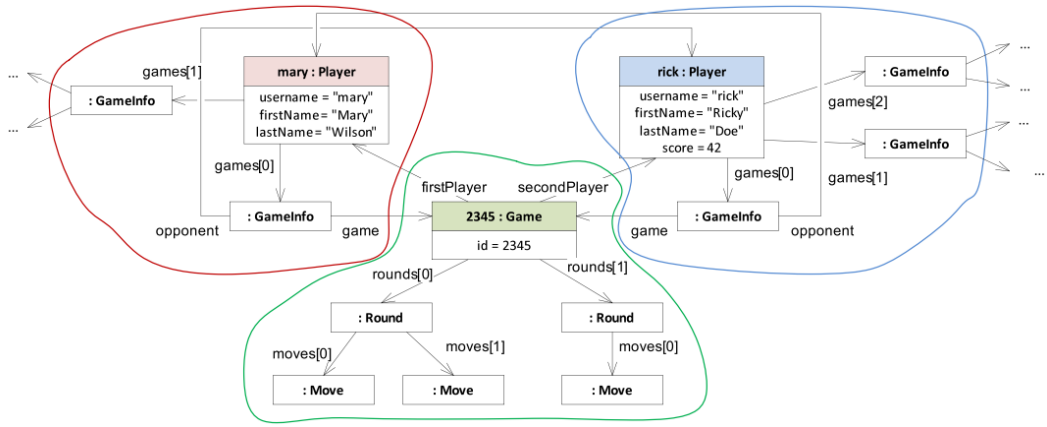
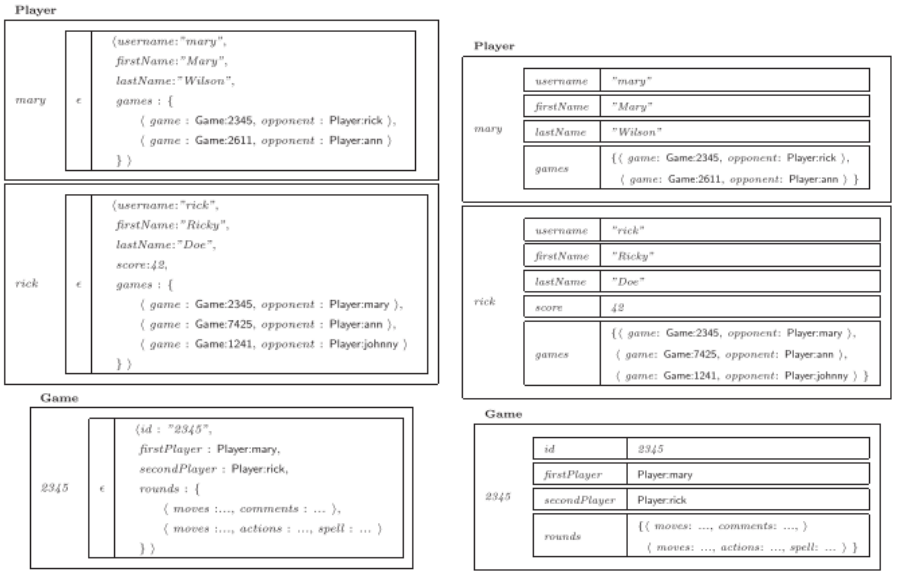


Figura 1.10: Design degli aggregati.

Partizionamento degli aggregati e modellazione NoSQL In questa fase si utilizza NoAM come modello intermedio tra gli aggregati e i database NoSQL. Ogni classe di aggregati viene rappresentato da una *collezione*, e ogni singolo aggregato con un *blocco*. La rappresentazione degli aggregati attraverso i blocchi è motivata dal fatto che entrambi supportano le stesse proprietà di accesso e distribuzione. Infatti, i database NoSQL garantiscono atomicità ed efficienza sui blocchi, riportando così queste proprietà anche sugli aggregati.

Due modalità naturali per rappresentare gli aggregati sono le seguenti:

– *Entry per Aggregate Object (EAO)*: In cui ogni aggregato viene rappresentato con una singola *entry*. La chiave della *entry* e vuota, il valore contiene l'intero aggregato (Figura 1.11a).



(a) EAO (b) ETF

Figura 1.11: Rappresentazione degli aggregati in NoAM.

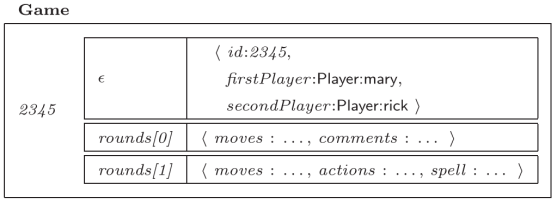


Figura 1.12: Partizionamento dell'aggregato Game.

– *Entry per Top-level Field (ETF)*: Ogni aggregato è rappresentato da diverse *entry*. Per ogni campo non innestato dell' aggregato si definisce una *entry* che ha: come chiave il nome del campo, come valore il valore del campo (Figura 1.11b).

Se queste rappresentazioni dovessero rivelarsi troppo rigide e limitanti, è possibile effettuare il partizionamento degli aggregati. L'idea consiste nel definire nuove *entry* che esplicitino il valore complesso dell'aggregato, oppure raggruppare *entry* che vengono accedute assieme. Un esempio di partizionamento dell'aggregato Game è visibile in Figura 1.12. Considerando la situazione di partenza in fig. 1.11b, si è deciso di accorpare i campi *id*, *firstPlayer* e *secondPlayer* e di partizionare il campo *rounds* nei singoli elementi dell'array. Alcune linee guida per il partizionamento sono:

– Se un aggregato è di dimensioni ridotte, o la maggior parte dei sui dati viene acceduta o modificata assieme, dovrebbe essere rappresentato da una singola *entry*

– Due elementi dovrebbero appartenere alla stessa *entry* se vengono acceduti o modificati assieme.

Implementazione La fase di implementazione risulta abbastanza lineare e consiste nel tradurre le modellazioni NoAM nei concetti messi a disposizione dai database.

Per i database chiave-valore si utilizza una coppia chiave-valore per ogni *entry* del dataset. La chiave viene divisa in chiave primaria e chiave secondaria. La sotto-chiave primaria contiene il nome della collezione e l'id del blocco. La sotto-chiave secondaria contiene il nome della *entry*. Un esempio è mostrato in Figura 1.13.

key (/major/key/-/minor/key)	*value*
Player/mary/-/username	*"mary"*
Player/mary/-/firstName	*"Mary"*
Player/mary/-/lastName	*"Wilson"*
Player/mary/-/games[0]	{*game*: "Game:2345", *opponent*: "Player:rick"}
Player/mary/-/games[1]	{*game*: "Game:2611", *opponent*: "Player:ann"}
Player/rick/-/username	*"rick"*
Player/rick/-/firstName	*"Ricky"*
Player/rick/-/lastName	*"Doe"*
Player/rick/-/score	*42*
Player/rick/-/games[0]	{*game*: "Game:2345", *opponent*: "Player:mary"}
Player/rick/-/games[1]	{*game*: "Game:7425", *opponent*: "Player:ann"}
Player/rick/-/games[2]	{*game*: "Game:1241", *opponent*: "Player:johnny"}
Game/2345/-/id	*2345*
Game/2345/-/firstPlayer	"Player:mary"
Game/2345/-/secondPlayer	"Player:rick"
Game/2345/-/rounds[0]	{*moves*: ..., *comments*: ...}
Game/2345/-/rounds[1]	{*moves*: ..., *actions*: ..., *spell*: ...}

Figura 1.13: Implementazione chiave-valore.

L'implementazione per i database wide-column si ottiene creando una tabella per ogni *collezione*, ed un singolo item per ogni *blocco*. Il nome della collezione diventa il nome della tabella, l'*id* del blocco diventa la chiave primaria, le *entry* del blocco si traducono in attributi dell'item. Un esempio è mostrato in Figura 1.14.

table **Player**

username	*firstName*	*lastName*	*score*	*games[0]*	*games[1]*	*games[2]*
"mary"	*"Mary"*	*"Wilson"*		{ *game*: ..., *opponent*: ... }	{ ... }	
"rick"	*"Ricky"*	*"Doe"*	*42*	{ *game*: ..., *opponent*: ... }	{ ... }	{ ... }

table **Game**

id	*firstPlayer*	*secondPlayer*	*rounds[0]*	*rounds[1]*	*rounds[2]*
2345	Player:mary	Player:rick	{ *moves*: ..., *comments*: ... }	{ ... }	

Figura 1.14: Implementazione wide-column.

collection **Player**

id	*document*
mary	{ *_id:"mary"*, *username:"mary"*, *firstName:"Mary"*, *lastName:"Wilson"*, *games:* [{ *game:"Game:2345"*, *opponent:"Player:rick"*}, { *game:"Game:2611"*, *opponent:"Player:ann"*}] }

Figura 1.15: Implementazione documentale.

Per i database documentali è possibile definire una collezione per ogni *collezione* NoAM e un documento per ogni *blocco*. Il nome della collezione coincide con quello della *collezione* NoAM e per ogni *entry* si definisce un campo chiave-valore nel documento. Un esempio è mostrato in Figura 1.15.

Limiti di NoAM

Attualmente, la progettazione di database per sistemi NoSQL è effettuata sulla base di best practice, spesso specifiche del sistema che si sta utilizzando. NoAM é un approccio innovativo che tenta di risolvere questo problema proponendo una metodologia dettagliata basata su un modello astratto, indipendente dal sistema specifico.

La metodologia non presenta però linee guida sulla parte di modellazione orientata agli aggregati. Questa fase iniziale, che ha ripercussioni su tutto il processo di modellazione, non viene trattata in maniera esaustiva; ci si concentra invece su operazioni di dettaglio sugli aggregati, definendo tecniche di raffinamento e partizionamento che, nell'ottica di questa tesi, non sono di primaria importanza. Inoltre, NoAM, non definisce una notazione che permetta di descrivere le scelte effettuate per il partizionamento e il raffinamento degli aggregati. Questa mancanza rende di difficile utilizzo il modello astratto, non permettendo di descrivere in maniera agevole le modellazioni. Per questi motivi, si è deciso di non utilizzare la metodologia in questa tesi.

Capitolo 2

Benchmark di riferimento

Nel seguente capitolo verrà introdotto il concetto di benchmark e in particolare quello di database benchmarking, spiegandone l'importanza e gli scopi. Si descriveranno in maniera dettagliata tre benchmark per database: UniBench, YCSB e TPC-C. Verranno analizzati i pro e i contro dei singoli benchmark, spiegando le motivazioni che hanno portato alla scelta del benchmark TPC-C.

2.1 Benchmarking

L'atto di misurare le performance di un sistema, testandolo in maniera riproducibile su scenari reali o simulati, viene definito *benchmarking*. Il benchmarking non solo permette di valutare le performance di un singolo sistema, ma può essere utilizzato per confrontare le performance di sistemi diversi, o per analizzare singole componenti di un sistema. Generalmente i benchmark prevedono l'esecuzione di uno o più *workload*, ovvero una serie di operazioni che il sistema deve eseguire in maniera monitorata, generando dati per la successiva analisi. Si distinguono a tal proposito due categorie:

- *Synthetic benchmark*: utilizzano programmi ad-hoc per l'esecuzione del workload
- *Application benchmark*: eseguono il programma che sarà effettivamente messo in produzione

I primi sono utili per testare singoli componenti, mentre gli altri vengono utilizzati per valutare come si comporta effettivamente il sistema in un contesto reale.

Un benchmark, per risultare utile ed efficace dovrebbe avere le seguenti sette proprietà [10]:

- Rilevanza: i benchmark dovrebbero misurare proprietà importanti.

- Rappresentatività: le metriche delle prestazioni dovrebbero essere ampiamente accettate dall'industria e dal mondo accademico.
- Imparzialità: tutti i sistemi dovrebbero essere comparati in maniera imparziale.
- Ripetibilità: i risultati di un benchmark dovrebbero essere verificabili e riproducibili.
- Economicità: eseguire i benchmark non dovrebbe essere troppo costoso.
- Scalabilità: i benchmark dovrebbero poter essere eseguiti su diversi sistemi con diverse risorse e capacità computazionali.
- Trasparenza: i risultati di un benchmark dovrebbero essere facilmente interpretabili.

Nell'ambito dei benchmark il *database benchmark* è un framework di test riproducibile per caratterizzare e confrontare le prestazioni (tempo, memoria o qualità) di sistemi di database o algoritmi su tali sistemi. Il database benchmark definisce il sistema sotto test, il carico di lavoro, le metriche e i modi di esecuzione [8]. I sistemi sotto test comprendono il database e il suo ambiente di sviluppo, ovvero il sistema operativo e l'hardware su cui viene eseguito. E' possibile che un benchmark ponga delle limitazioni sulle risorse hardware, così da rendere i risultati riproducibili e imparziali su tutti i sistemi.

Tipicamente, un database benchmark è costituito da due elementi principali: un modello dei dati e un modello del carico di lavoro (insieme di operazioni di lettura e scrittura) da applicare sul database, seguendo un protocollo predefinito. La maggior parte dei benchmark include anche una serie di metriche delle prestazioni semplici o composite come tempo di risposta, velocità effettiva, numero di input / output, utilizzo del disco o della memoria, ecc. La popolazione iniziale del database può essere generata, in maniera esplicita attraverso un file, oppure utilizzando algoritmi per la generazione dei dati.

2.2 UniBench

UniBench[1] è un benchmark per *Multi-Model Database* (MMDB) [18]. Mentre i database tradizionali sono organizzati su un unico modello, i MMDB supportano modelli diversi in un singolo sistema. Questo per sfruttare al meglio tutte le funzionalità e i punti di forza messi a disposizione dalle varie tipologie di database:

[1] `https://www.helsinki.fi/en/researchgroups/unified-database-management-systems-udbms`

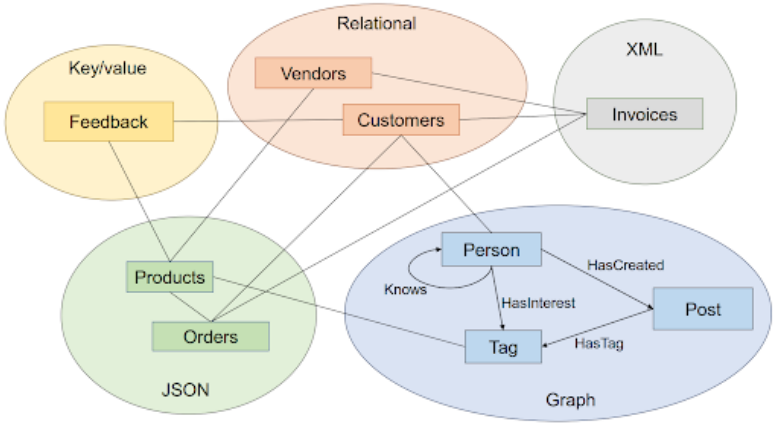


Figura 2.1: Modello di UniBench

grafo, documentale, chiave-valore etc. I classici benchmark per database relazioni o NoSQL, non sono progettati per i MMDB, per questo è nato UniBench.

UniBench è costituito da un modello di dati misto, un generatore di dati multi-modello e un set di carichi di lavoro [16]. I dati generati coprono XML, JSON, tabulare, chiave-valore e grafo; i workload sono composti da interrogazioni e transazioni multi-modello che comprendono diversi aspetti della gestione di dati multi-modello.

Modello Il modello di UniBench simula una web-application che combina e-commerce e social media (Figura 2.1). Il modello relazionale include *clienti* e *fornitori* strutturati, il modello JSON contiene *ordini* e *prodotti* semi-strutturati. Il social network è modellato come un grafo, che include tre entità e quattro relazioni, ovvero *persona*, *post*, *tag*, una persona *conosce* una persona, una persona *è interessata* ha un tag, una persona *crea* un post, il post *ha un tag*. I *feedback* sono modellati come chiave-valore, infine le *fatture* come XML. Come si può notare dal modello (Figura 2.1) esistono delle relazioni extra-modello. Ad esempio, un *cliente* conosce una *persona* (relazionale-grafo), un *cliente* effettua *ordini* (relazionale-JSON).

Workload Il workload di UniBench si compone di un insieme di 10 query read-only e 2 transazioni read-write che lavorano su due o più modelli e coprono interrogazioni OLAP e OLTP. Sono progettate con la tecnica del *choke-point*, con l'intento quindi di testare diversi aspetti del database (query optimizer, storge system etc.) e valutare nuove tecniche per le interrogazioni multi-modello. L'insieme dettagliato è visibile in Tabella A.1.

2.3 YCSB

Yahoo! Cloud Serving Benchmark (YCSB) [10] è un framework open-source sviluppato da Yahoo! con l'intento di facilitare il processo di comparazione di diversi sistemi cloud basati su database NoSQL. Il framework si compone di un client (*YCSB Client*) scritto in Java, per la generazione del dataset e l'esecuzione del workload e di package con dei workload standard che coprono i casi più importanti per valutare le performance di database (read heavy, write heavy etc.). La caratteristica fondamentale di YCSB è la possibilità di estendere il framework con carichi di lavoro personalizzati, contribuendo così alla creazione di nuovi package che modellino applicazioni interessanti. YCSB supporta già diversi database (per la maggior parte NoSQL) e permette la definizione di nuovi moduli per i database non ancora supportati (se necessario).

Modello Il modello di YCSB è minimale, contiene infatti una singola tabella che definisce utenti con 10 proprietà. Il framework è stato comunque progettato per supportare l'utilizzo di modelli diversi, definibili dall'utente.

Workload Il client di YCSB definisce 6 workload standard che si possono adattare a varie applicazioni.

- **Workload A**: Update heavy workload. 50% letture 50% scritture.
- **Workload B**: Read mostly workload. 95% letture 5% scritture.
- **Workload C**: Read only. 100% letture.
- **Workload D**: Read latest workload. 95% letture 5% inserimenti. Gli ultimi dati inseriti sono i primi che vengono letti.
- **Workload E**: Short ranges. 95% scans 5% inserimenti. Si effettuano interrogazioni su range di dati.
- **Workload F**: Read-modify-write. 50% letture 50% scritture. Ogni record viene letto, modificato e poi inserito nel database.

La caratteristica fondamentale di YCSB è comunque la possibilità di definire carichi di lavoro personalizzati, adattabili alle esigenze dello sviluppatore.

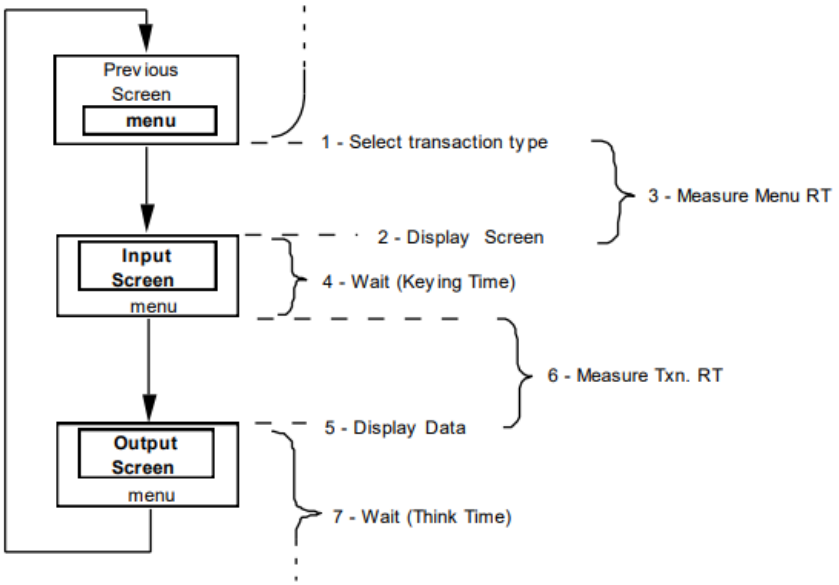


Figura 2.2: Ciclo di vita di un terminale simulato in TPC-C.

2.4 TPC-C

TPC[2] è un'organizzazione no-profit che nasce con l'intento di definire dei buoni benchmark assieme a delle specifiche dettagliate per monitorarne e interpretarne i risultati. TPC Benchmark C (TPC-C) [15] è un on-line transaction processing (OLTP) benchmark di TPC. Si compone di cinque transazioni di diversa complessità, alcune eseguite on-line, altre in modalità differita. La tipologia di transazioni modella un dominio commerciale in cui diversi clienti possono effettuare ordini sulla merce venduta da una compagnia. In particolare le transazioni sono: inserimento ed evasione di un ordine, controllo dello stato di un ordine, pagamento di un ordine e monitoraggio della merce disponibile nel magazzino.

TPC-C prevede l'utilizzo di terminali simulati che eseguono operazioni su un singolo database. Il ciclo di vita di un terminale prevede l'alternarsi di momenti di attività, nei quali si interagisce con il database, e momenti di attesa. La scelta delle transazioni da eseguire da parte di un terminale deve seguire una distribuzione definita in modo da prediligere l'esecuzione di alcune transazioni rispetto ad altre. Infatti, si vuole dare priorità alle transazioni che sono reputate più comuni in un applicazione di gestione ordini, ovvero inserimento e pagamento di un ordine. I terminali simulati prevedono anche la definizione di schermate che permettono di selezionare transazioni (menu), inserire input (input screen) e visualizzare gli out-

[2] `http://www.tpc.org/tpcc/`

Transazione	**Percentuale**	**Keying Time**	**Mean Think Time (μ)**
New Order	45%	18 s	12 s
Payment	43%	3 s	12 s
Order Status	4%	2 s	10 s
Delivery	4%	2 s	5 s
Stock Level	4%	2 s	5 s

Tabella 2.1: Valori per l'esecuzione del workload relativi alla tipologia di transazione.

put (output screen). Ogni terminale quindi, esegue ciclicamente queste 7 attività (Figura 2.2):

1. *Selezionare dal menu la tipologia di transazione*: in questa fase il terminale sceglie quale transazione eseguire. Ogni tipologia di transazione deve partecipare all'insieme delle transazioni eseguite secondo le percentuali mostrate in Tabella 2.1.

2. *Visualizzare la schermata di input*: viene mostrata la schermata relativa all'inserimento dell'input per la transazione.

3. *Misurare il Menu RT (Menu Response Time)*: si misura il tempo trascorso tra la selezione della transazione e la visualizzazione della schermata successiva.

4. *Simulare un tempo di inserimento dei dati (Keying Time)*: si attende un tempo che dipende dal tipo di transazione (Tabella 2.1), simulando un inserimento di dati. Terminata la fase di inserimento, si esegue la transazione.

5. *Visualizzare l'output*: si visualizza l'output derivante dall'esecuzione della transazione.

6. *Misurare il Txn. RT (Transaction Response Time)*: si misura il tempo trascorso dall'invio dei dati di input alla visualizzazione completa dell'output.

7. *Simulare un tempo di attesa (Think Time)*: si attende per un tempo variabile. Per ogni tipologia di transazione, il tempo di attesa T_t è calcolato attraverso la seguente equazione:

$$T_t = -\ln(r) \cdot \mu$$

   Dove r è un numero casuale tra 0 e 1, mentre μ è il tempo di attesa medio, costante per ogni transazione (Tabella 2.1).

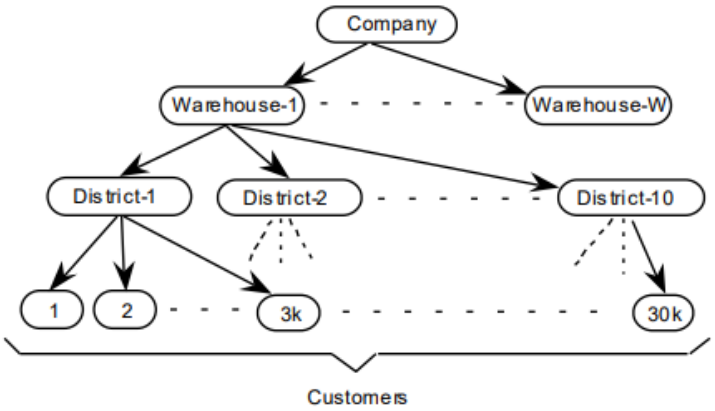


Figura 2.3: Gerarchie di entità del modello TPC-C.

La metrica principale riporta dal TPC-C misura il numero di ordini al minuto che è stato in grado di gestire il database. Oltre ai dati sulle performance vanno riportati anche i costi totali del sistema che si sta testando, che includono: il costo di tutti i componenti hardware e software, i costi di manutenzione su 5 anni e i costi per gestire l'archiviazione dei dati generati per un periodo di 180 giorni (8 ore al giorno), alla velocità segnalata.

Modello Il modello del TPC-C rappresenta un fornitore all'ingrosso con una serie di distretti di vendita e magazzini associati. Ogni magazzino regionale copre 10 distretti. Ogni distretto serve 3.000 clienti. Tutti i magazzini mantengono scorte per i 100.000 articoli venduti dalla società (Figura 2.3).

I clienti chiamano la società per effettuare un nuovo ordine o richiedere lo stato di un ordine esistente. Gli ordini sono composti da una media di 10 dettagli d'ordine. L'uno per cento di tutti i dettagli d'ordine è per articoli non presenti nel magazzino regionale e devono essere forniti da un altro magazzino. Il sistema viene utilizzato anche per inserire pagamenti, elaborare ordini per la consegna e monitorare livelli delle scorte per identificare potenziali carenze di prodotti. Il database di TPC-C si compone quindi di 9 tabelle e 7 relazioni visibili in Figura 2.4.

Workload Il carico di lavoro del TPC-C consiste nell'esecuzione di 10 terminali per ogni warehouse. Ogni terminale esegue il suo ciclo di vita per una durata che deve essere di almeno 120 minuti. Durante il suo ciclo di vita, il terminale può eseguire le seguenti transazioni:

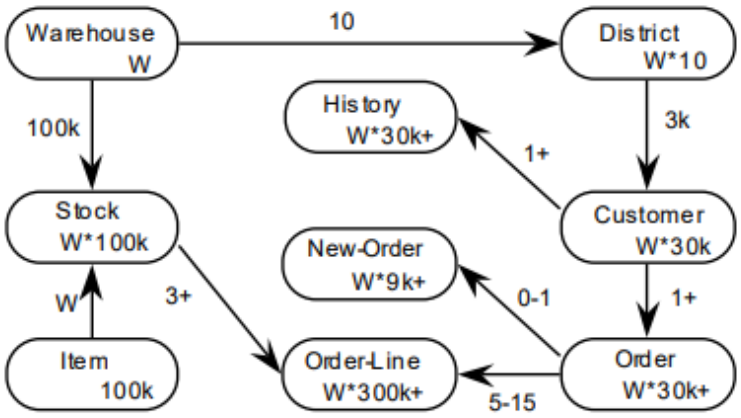


Figura 2.4: Modello dati del TPC-C. "W" rappresenta il numero di warehouse, ed è il fattore di scala del data set.

- **New Order Transaction**: consiste nell'inserimento di un ordine completo nel database.
- **Payment Transaction**: aggiorna il saldo del cliente e riflette il pagamento sul distretto e sulle statistiche del magazzino.
- **Order-Status Transaction**: viene letto l'ultimo ordine effettuato dal cliente.
- **Delivery Transaction**: si richiede di processare un gruppo di 10 ordini non ancora evasi. La transazione viene eseguita in differita, senza che il terminale debba aspettare il risultato.
- **Stock-Level Transaction**: determina il numero di articoli venduti di recente con un livello di scorta inferiore a una soglia specificata.

2.5 Scelta del benchmark

Il benchmark di UniBench offre un modello di dati vario ed eterogeneo, con un workload che si adatta bene sia a database relazionali che NoSQL. Non è stato utilizzato nel contesto di questa tesi poiché le informazioni sul database sono frammentarie, a tratti contraddittorie a causa delle diverse versioni che sono state pubblicate.

YCSB è utilizzato spesso in letteratura come benchmark per database NoSQL [3, 4, 5]. Il carico di lavoro è composto per la maggior parte da letture e scritture

su singoli record e il modello dati è minimale. Un workload di questo tipo non si presta a considerazioni complesse sulle modellazioni, ma piuttosto a valutazioni quantitative sulle performance dei database. Per questo, anche se YCSB è considerato una standard per il benchmarking di database NoSQL, non è stato utilizzato in questa tesi.

Il benchmark TPC-C offre un modello semplice, che in combinazione con la complessità delle transazioni, si presta a varie considerazioni sulle modellazioni, in ottica relazionale e non. Inoltre, anche se propone un workload con particolari riferimenti a uno schema normalizzato, può essere utilizzato, con diverse modifiche, anche su database NoSQL [14]. In questa tesi si è deciso quindi di utilizzare questo benchmark. Lo scopo non era quello di produrre dei risultati "certificabili" secondo le specifiche del TPC-C, quindi, ci si è concentrati in particolare sulle specifiche inerenti al modello e al workload, con particolare attenzione a quegli aspetti che avrebbero influenzato le performance delle modellazioni.

Una prima valutazione è stata fatta sul tempo di esecuzione del benchmark per ciascuna modellazione. Eseguendo un limitato sottoinsieme dei test, si è visto che il tempo di esecuzione del workload poteva essere ridotto a 5 minuti (rispetto ai 120 previsti dal TPC-C), senza inficiare le valutazioni sulle diverse modellazioni. Anche i tempi di attesa dei terminali (key time, think time) non avevano particolari riscontri sulla comparazione delle performance, quindi sono stati eliminati. Queste modifiche hanno permesso di diminuire drasticamente la durata complessiva dei test, semplificando lo sviluppo e permettendo di valutare un insieme maggiore di modellazioni. Aspetti come la generazione dei dati e l'implementazione delle transazioni (generazione degli input ed esecuzione delle query), influenti nella definizione delle diverse modellazioni, sono stati trattati seguendo puntualmente le specifiche.

Capitolo 3

Modellazioni non relazionali del TPC-C

L'obiettivo di questo capitolo è quello di descrivere delle varianti non relazionali alla modellazione standard del TPC-C. Per fare questo, in assenza di una metodologia, si è deciso di definire le modellazioni che meglio si adattano al carico di lavoro previsto. A tal proposito, si procede analizzando quest'ultimo e, transazione per transazione, si definiscono le varianti non-relazionali che potrebbero ottimizzare l'esecuzione. Per ogni transazione verranno elencate le operazioni che la compongono, accompagnando ogni fase da una breve descrizione testuale e uno script SQL che offre un'intuizione della sua esecuzione. Per evitare una spiegazione prolissa, non verranno riportate tutte le informazioni necessarie per un'implementazione completa del workload, ma solamente quei dettagli ritenuti più importanti, utili a giustificare le scelte di modellazione. Le tecniche di progettazione utilizzate sono diverse, la maggior parte delle quali descritte e spesso consigliate, nelle documentazioni ufficiali[1] [2] e nei libri di testo [9, 13]. Le scelte di modellazione verranno presentate utilizzando la notazione UML; nello specifico si utilizzeranno i diagrammi delle classi. Il costrutto dell'aggregazione verrà utilizzato per indicare la de-normalizzazione. Il diagramma UML di riferimento è mostrato in Figura 3.1.

[1] `https://docs.mongodb.com/manual/core/data-modeling-introduction/`
[2] `https://cassandra.apache.org/doc/latest/data_modeling/`

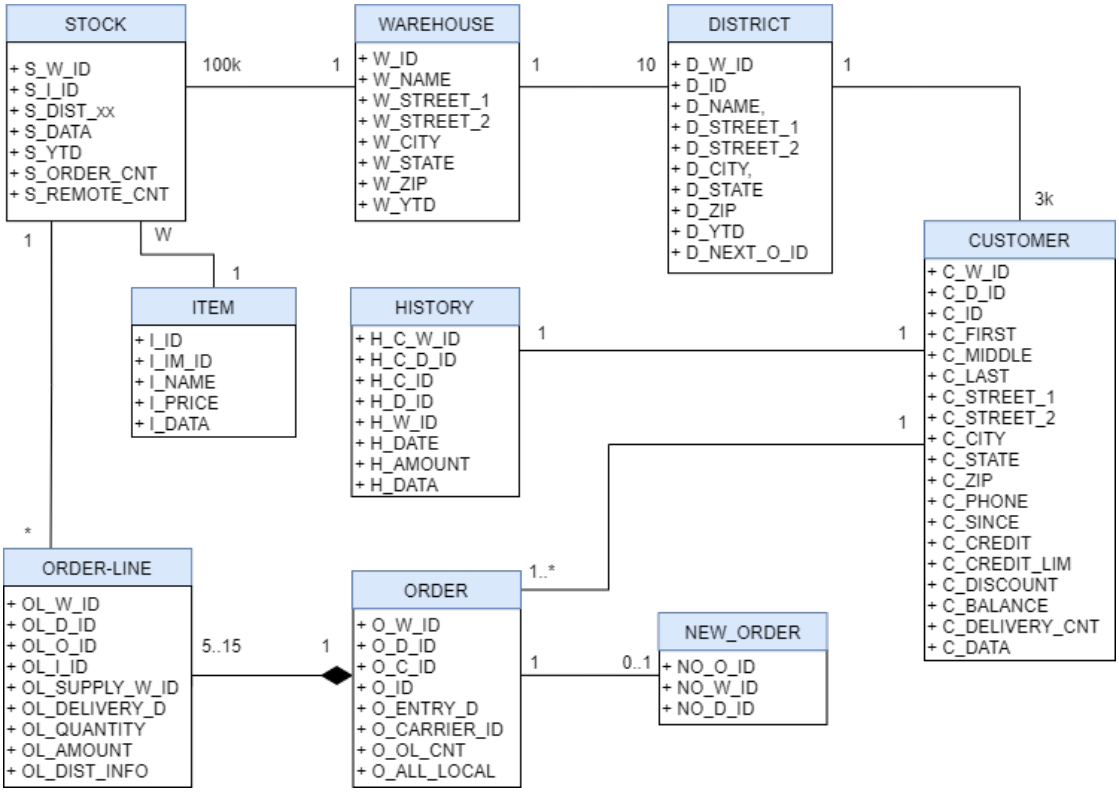


Figura 3.1: Diagramma delle classi del modello TPC-C.

3.1 New Order

La transazione New-Order consiste nell'inserimento di un ordine completo nel database. Si compone di operazioni di lettura e scrittura, viene eseguita con un'elevata frequenza e si richiedono brevi tempi di risposta.

3.1.1 Input

- `warehouse-id`: chiave primaria del warehouse associato al cliente che effettua l'ordine.
- `district-id`: chiave del distretto, che assieme al `warehouse-id` lo identifica univocamente.
- `customer-id`: la tupla `(warehouse-id, district-id, customer-id)` identifica univocamente un cliente.
- `o_ol_cnt`: numero di linee d'ordine.

- per ogni item:
  - `item-id`: id del prodotto acquistato.
  - `quantity`
  - `ol_supply_w_id`: id del warehouse che fornisce il prodotto.

3.1.2 Operazioni

1. Lettura da Warehouse: si legge il valore di `w_tax` da Warehouse.

```
select w_tax from warehouse
where w_id = warehouse-id
```

2. Lettura da District: si legge il valore di `d_tax` da District.

```
select d_tax from district
where d_w_id = warehouse-id
and d_id = district-id
```

3. Scrittura su District: si incrementa `d_next_o_id` di 1.

```
update district
where d_w_id = warehouse-id
and d_id = district-id
set d_next_o_id = d_next_o_id + 1
```

4. Lettura da Customer: si leggono `c_discount`, `c_last`, `c_credit` da Customer.

```
select c_discount,c_last,c_credit
from customer
where c_w_id = warehouse-id
and c_d_id = district-id
and c_id = customer-id
```

5. Inserimento in Order: si inserisce un ordine nella tabella Order.
6. Inserimento in NewOrder: si inserisce un ordine nella tabella New-Order.
7. Per ogni prodotto nell'ordine:
   (a) Lettura da Item: si leggono `i_price`, `i_name`, `i_data` da Item

```
select i_price,i_name,i_data from item
where i_id = item-id
```

(b) Lettura da Stock: si leggono `s_dist_xx`, `s_data` da Stock.

```
select s_dist_xx, s_data from stock
where s_i_id = item-id
and s_w_id = ol_supply_w_id
```

(c) Scrittura su Stock: si aggiornano `s_quantity`, `s_ytd`, `s_remote_cnt` in Stock.

```
update stock
where s_i_id = ol_i_id
and s_w_id = ol_supply_w_id
set s_quantity, s_ytd, s_remote_cnt
```

(d) Inserimento in Order-Line: si inserisce una linea d'ordine in Order-Line.

Varianti per MongoDB

[C-ext] Informazioni di Warehouse e District dentro Customer Considerando le voci 1, 2 e 4, visto che ogni cliente è associato a un singolo distretto (e quindo un singolo warehouse), si possono inserire i valori di `w_tax` e `d_tax` all'interno della tabella Customer (Customer-Extended, Figura 3.2), risparmiando le letture da Warehouse e District. Per mantenere consistente il database, ogni volta che `w_tax` o `d_tax` vengono modificati, si dovrebbero aggiornare tutti i 30000 clienti. Il carico di lavoro del TPC-C non prevede però query di aggiornamento sulle tasse di distretto e warehouse; non bisogna quindi preoccuparsi di mantenere aggiornati i documenti in Customer-Extended . L'overhead in termini di spazio equivale alla replicazione dei due campi su tutti i customer (30000 per ogni warehouse) $30k * (size(w\_tax) + size(d\_tax))$, assumendo di memorizzare `w_tax` e `d_tax` come tipo di dato decimal (16 byte), avremo 910Kb per ogni warehouse.

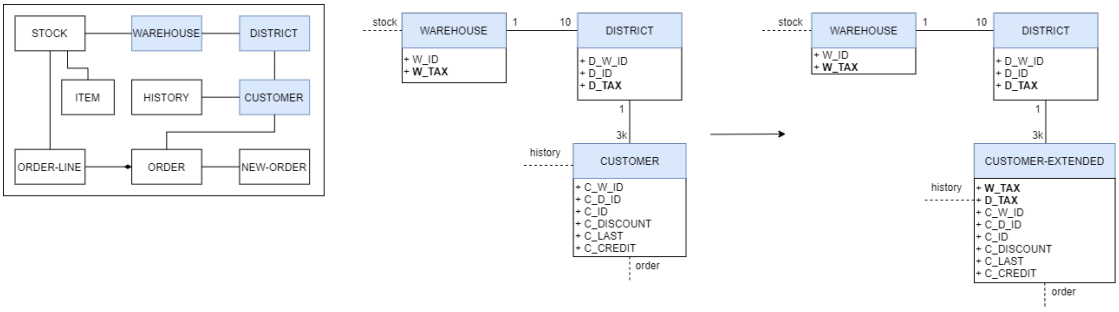


Figura 3.2: Modellazione di Customer con l'aggiunta dei campi `w_tax` e `d_tax` da Warehouse e District.

Informazioni di Order e Order-Line dentro Customer Proseguendo con questa idea, si potrebbe pensare di estendere ulteriormente il documento di Customer con la lista dei suoi ordini, così da avere un documento contenente tutte le informazioni relative al singolo customer. Questa soluzione porta con se due problemi: (a) si dovrebbe mantenere aggiornato l'array di ordini a ogni transazione (voce 5), (b) si andrebbe a definire un array *unbounded*, in quanto un customer può effettuare un numero di ordini potenzialmente illimitato. Come anche indicato nella documentazione di MongoDB[3], collezioni unbounded sono sconsigliate in quanto generano documenti con dimensioni imprevedibili. Con il crescere dell'array, le prestazioni di lettura e creazione di indici su quell'array diminuiscono gradualmente. Inoltre, un array di questo tipo può incrementare la dimensione del documento a valori non più gestibili dal database.

New-Order La voce 6 prevede l'inserimento di un elemento nella tabella New-Order. Questa tabella non contiene delle informazioni aggiuntive rispetto all'ordine, ma indica semplicemente quali ordini non sono ancora stati processati. Si potrebbe sostituire con un campo all'interno del documento Order (e.g isDelivered) che indica se l'ordine è stato spedito. Questo permetterebbe di evitare un inserimento nella transazione (voce 6) e porterebbe a un risparmio in termini di spazio, eliminando le informazioni ridondanti. Si è deciso comunque di mantenere la tabella New-Order, per rimanere consistenti con il modello del TPC-C.

[SWI] Informazioni di Item dentro Stock L'ultima parte della transazione voce 7a, voce 7b e voce 7c, utilizza le tabelle stock e item. Per gestire questa relazione in un ottica non relazionale, abbiamo due possibilità: (a) replicare le informazioni dell'item all'interno degli stock (Figura 3.3a), (b) innestare le informazioni degli stock in un array dentro item (Figura 3.3b).

[3] `https://docs.atlas.mongodb.com/schema-suggestions/avoid-unbounded-arrays/`

La prima soluzione rende più efficienti le letture, ma nel momento in cui si modifica un'informazione su item, si deve propagare l'aggiornamento su tutti gli stock relativi a quell'item. Siccome ogni warehouse mantiene un singolo stock per ogni item, un aggiornamento su Item si traduce in un ulteriore update sulla tabella Stock per ogni warehouse. Anche in questo caso però, il carico di lavoro del TPC-C non prevede operazioni di aggiornamento sulla tabella Item.

In termini di spazio, considerando 100k item per warehouse, si avrà un incremento per warehouse di $100k * size(ITEM)$[4]. Un documento della collezione Item ha una dimensione di circa 140 byte, portando quindi a un overhead di 14Mb per warehouse.

[IWS] Lista degli Stock dentro Item dentro Stock La seconda soluzione, che prevede la definizione di un array contenente le informazioni degli stock dentro Item, non si discosta molto rispetto alla prima. Permette infatti di avere un unico documento nel quale leggere tutte le informazioni necessarie per soddisfare la transazione. In questo caso, supponendo di eliminare la tabella Stock originale, non c'è replicazione di dati, quindi nessun overhead in termini di memoria.

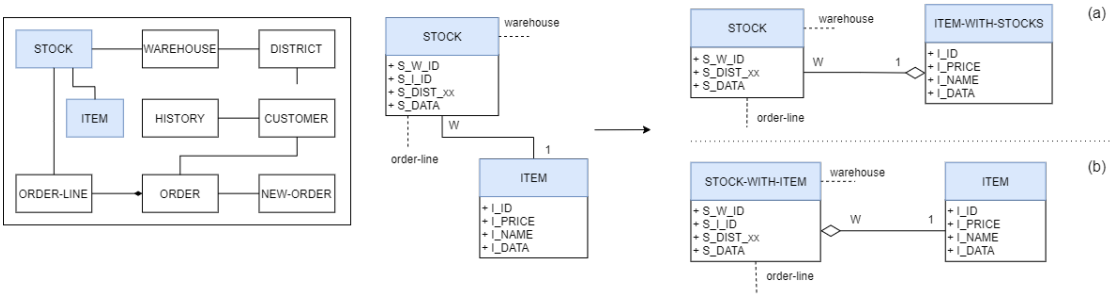


Figura 3.3: Modellazioni per la relazione Stock-Item.

Varianti per Cassandra

Per quanto riguarda Cassandra, non sono state fatte particolari considerazioni per questa transazione, in quanto si compone solamente di interrogazioni in lettura e scrittura con chiave primaria. Anche se poteva essere interessante riportare le modellazioni pensate per MongoDB, su Cassandra, si è preferito lasciare spazio a modellazioni proprie dei database orientati alle colonne, emerse dall'analisi delle altre transazioni.

[4]Dimensione di un documento nella collezione Item

3.2 Payment

La transazione Payment aggiorna il conto del cliente e riflette il pagamento sui dati di vendita del warehouse e del distretto. Si compone di operazioni di letture e scrittura, viene eseguita con un' elevata frequenza e si richiedono brevi tempi di risposta.

3.2.1 Input

- `warehouse-id`: chiave primaria del warehouse che riceve il pagamento.
- `district-id`: la coppia `(warehouse-id, district-id)` identifica univocamente un distretto.
- `customer-warehouse-id`: l'85% delle volte il customer effettua il pagamento nel proprio warehouse, quindi `customer-warehouse-id` = `warehouse-id`, mentre il 15% delle volte il customer effettua il pagamento su un warehouse casuale, diverso dal suo, quindi `customer-warehouse-id` $\neq$ `warehouse-id`
- `customer-id` o `customer-lastname`: Nel primo caso, che si verifica il 40% delle volte, il customer viene selezionato in maniera univoca attraverso il suo customer-id. Nel secondo caso, che si verifica il 60% delle volte, il customer viene selezionato attraverso il cognome. Per come viene popolato il database (ogni distretto ha 3000 customer, ma vengono generati solamente 1000 cognomi), in media 3 customer avranno lo stesso cognome. Il singolo customer viene selezionato ordinandoli per nome e prendendo quello centrale.
- `payment-amount`: Somma che il cliente deve pagare.

3.2.2 Operazioni

1. Lettura da Warehouse: si leggono `w_name`, `w_street_1`, `w_street_2`, `w_city`, `w_state`, `w_zip` da Warehouse.

```
select w_name, w_street_1, w_street_2,
w_state, w_zip from warehouse
where w_id = warehouse-id
```

2. Lettura da District: si leggono `d_name`, `d_street_1`, `d_street_2`, `d_city`, `d_state`, `d_zip` da District.

```
select d_name, d_street_1, street_2, d_city,
d_state, d_zip from district
where d_w_id = warehouse-id
and d_id = district-id
```

3. Lettura da Customer: si leggono `c_first`, `c_middle`, `c_last`, `c_street_1`, `c_street_2`, `c_city`, `c_state`, `c_zip`, `c_phone`, `c_since`, `c_credit`, `c_credit_lim`, `c_discount`, `c_balance` da Customer.

   (a) Lettura da Customer su chiave primaria.

```
select c_first, c_middle, c_last, ...
from customer
where c_w_id = warehouse-id
and c_d_id = district-id
and c_id = customer-id
```

   (b) Lettura da Customer su `c_last`.

```
select c_first, c_middle, c_last, ...
from customer
where c_w_id = warehouse-id
and c_d_id = district-id
and c_last = customer-lastname
order by c_first
```

4. Scrittura su Warehouse: si aggiorna il valore di `w_ytd` su Warehouse.

```
update warehouse
where w_id = warehouse-id
set w_ytd = w_ytd + payment-amount
```

5. Scrittura su District: si aggiorna il valore di `d_ytd` su District.

```
update district
where d_w_id = warehouse-id
and d_id = district-id
set d_ytd = d_ytd + payment-amount
```

6. Scrittura su Customer: si aggiorna il conto del cliente.

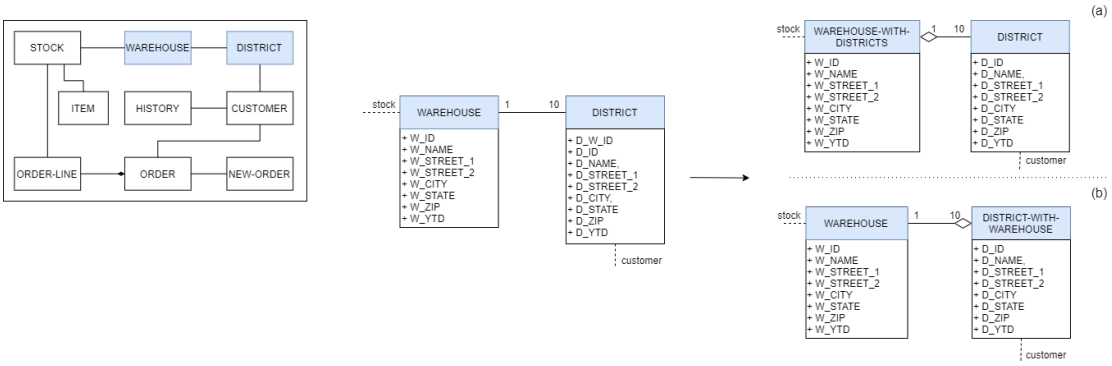


Figura 3.4: Possibili modellazioni della relazione Warehouse-District.

```
update customer
where c_w_id = warehouse-id
and c_d_id = district-id
and c_id = customer-id
set
c_balance = c_balance - payment-amount,
c_ytd_payment = c_ytd_payment + payment-amount,
c_payment_cnt = c_payment_cnt + 1
```

7. Inserimento in History: si inserisce un elemento nella tabella History.

Varianti per MongoDB

[WD] Lista di distretti dentro Warehouse Considerando le voci 1 e 2, si potrebbero accorpare le due letture da Warehouse e District, in una selezione di un unico documento che mantenga le informazioni relative al warehouse e all'insieme dei distretti associati a quel warehouse (Figura 3.4a). Si risparmierebbe la lettura della tabella District, che in questo caso potrebbe essere del tutto eliminata, risparmiando spazio sul campo ridondante `d_w_id`. Anche gli aggiornamenti per la voce 4 e la voce 5 potrebbero essere eseguiti sullo stesso documento.

Informazioni di Warehouse in District Una de-normalizzazione alternativa potrebbe essere quella della replicazione delle informazioni del warehouse in ognuno dei 10 distretti (Figura 3.4b). L'aggiornamento di `w_ytd` (voce 4) richiederebbe però di mantenere aggiornati tutti e 10 i distretti a ogni esecuzione della transazione. Il vantaggio in fase di lettura (voci 1 e 2) dunque, non è tale da giustificare una simile modellazione. Inoltre, la replicazione porterebbe ad overhead di memoria

pari a $10 * size(WAREHOUSE)$, che corrispondono a 2,16Kb (considerano che un documento occupa circa 216 byte).

Il problema della consistenza di `w_ytd`, si potrebbe risolvere supponendo di non mantenerlo aggiornato, ma ricalcolarlo ogni volta che è necessario leggerlo. Come indicato nella documentazione infatti, il campo `w_ytd` di un warehouse (con chiave `w_id`), è ricavabile dalla tabella History:

```
select sum(h_amount) as w_ytd from history
where h_w_id = w_id
```

Lo stesso ragionamento lo si potrebbe fare per `d_ytd`, ricalcolandolo da History:

```
select sum(h_amount) as d_ytd from history
where h_w_id = w_id
and h_d_id = d_id
```

Per rimanere il più possibile fedeli alle specifiche di esecuzione della transazioni, si è comunque deciso di mantenere i due campi `w_ytd` e `d_ytd`, aggiornandoli a ogni esecuzione.

Lista di Customer all'interno di District In riferimento alla voce 2 e alla voce 3, un'idea potrebbe essere quella di inserire la lista dei clienti all'interno del loro distretto, indicizzati per `c_id` (Figura 3.5). Per come è strutturato l'input, l'85% delle volte il distretto del cliente coincide con il distretto che riceve il pagamento, quindi, accedendo al documento in District, trovo le informazioni del cliente all'interno della lista. Questo mi permetterebbe in una singola operazione, di leggere e aggiornare le informazioni del distretto e del cliente. Nel 15% delle esecuzioni invece, la modellazione non aiuterebbe perché si dovrebbe comunque accedere a documenti diversi.

Se si decidesse di mantenere anche la tabella Customer originaria, si dovrebbe mantenere aggiornata (voce 6), inoltre si pagherebbe il costo in termini di memoria dovuto alla replicazione. Sapendo infatti che ogni distretto dovrebbe mantenere la lista dei suoi 3k clienti, si avrebbe un costo (per ogni warehouse) di: $10 * 3k * size(CUSTOMER)$. Considerando che un documento della tabella Customer ha una dimensione di circa 840 byte, risulterebbero circa 25Mb per ogni warehouse.

Informazioni di Warehouse e District dentro Customer Una modellazione alternativa è quella che inserisce all'interno del Customer le informazioni del proprio warehouse e distretto (fig. 3.6). Decidendo di replicare solo le informazioni non accedute in scrittura, si risparmierebbero i costi di aggiornamento di `w_ytd` e `d_ytd`. Anche in questo caso, la modellazione sarebbe utile solamente nel 85%

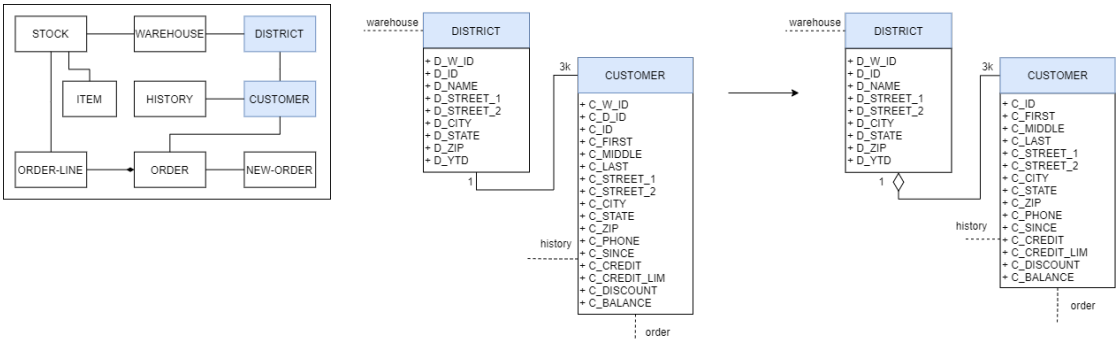


Figura 3.5: Modellazione della relazione District-Customer. Le informazioni dei clienti di ogni distretto vengono inserite all'interno del distretto.

dei casi, quelli in cui il warehouse che deve riceve il pagamento coincide con il warehouse del cliente.

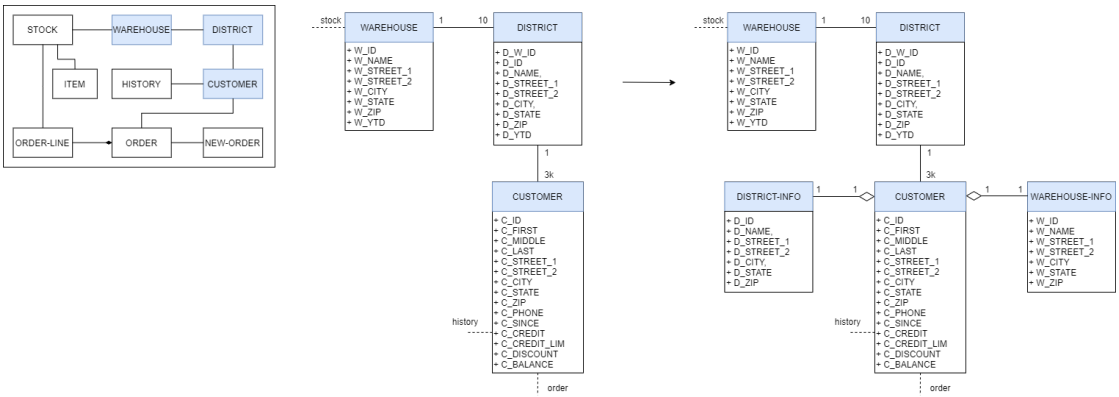


Figura 3.6: Modellazione della relazione Warehouse-District-Customer. Le informazioni del warehouse e del distretto vengono inserite all'interno del documento Customer.

History In ultimo, la transazione viene completata con l'inserimento di un elemento nella tabella History. Quest'ultima non ha un ruolo fondamentale nel carico di lavoro, non viene mai letta, ma viene usata come storico, per effettuare controlli di consistenza sul database. Si è deciso, visto il ruolo marginale della tabella, di non inserirla in particolari modellazioni, ma di mantenerla nella versione prevista dal benchmark.

Varianti per Cassandra

[C-by-last]Customer indicizzato per c_last Per quanto riguarda Cassandra la voce 3b prevede una lettura di Customer su chiave secondaria (`c_last`). Si è deciso di gestire questa lettura attraverso una tabella secondaria (Customer-By-Lastname) che coincide con la tabella Customer ed è indicizzata per `c_last`. Database distribuiti come Cassandra, sconsigliano interrogazioni che contengono filtri su campi non appartenenti alla chiave primaria, proprio per l'imprevedibilità sui tempi di esecuzione della query. Per questo motivo è necessaria la presenza di una tabella secondaria Customer-By-Lastname che permetta l'esecuzione efficiente dell'interrogazione. Un ulteriore miglioramento si avrebbe mantenendo la tabella ordinata per `c_first`, così da evitare l'ordinamento previsto dalla voce 3b. La tabella va mantenuta aggiornata, quindi, la voce 6 si tradurrà nell'esecuzione di due aggiornamenti, uno su Customer e uno su Customer-By-Lastname. Dal punto di vista della memoria, si replicano tutte le informazioni di ciascun cliente. Si ha dunque un costo di circa 15Mb (spazio occupato dalla tabella Customer nel database).

```
CREATE TABLE tpcc.customer(
    c_w_id int,
    c_d_id int,
    c_id int,
    c_last varchar,
    ...
PRIMARY KEY (c_w_id, c_d_id, c_id))
```

```
CREATE TABLE tpcc.customer-by-lastname(
    c_w_id int,
    c_d_id int,
    c_id int,
    c_last varchar,
    ...
PRIMARY KEY ((c_w_id, c_d_id, c_last), c_id))
WITH CLUSTERING ORDER BY (c_first ASC)
```

Figura 3.7: Schema delle tabelle Customer (a sinistra) e Customer-By-Lastname (a destra).

3.3 Order-Status

La transazione di Order-Status legge le informazioni dell'ultimo ordine effettuato da un cliente. Prevede operazioni di sola lettura, viene eseguita con bassa frequenza e non ha vincoli di tempo stringenti.

3.3.1 Input

- `warehouse-id`: chiave primaria del warehouse..
- `district-id`: chiave del distretto, la coppia `(warehouse-id, district-id)` identifica univocamente un distretto
- `customer-warehouse-id`: l'85% delle volte il customer effettua il pagamento nel proprio warehouse, quindi `customer-warehouse-id` = `warehouse-id`,

mentre il 15% delle volte il customer effettua il pagamento su un warehouse casuale, diverso dal suo, quindi `customer-warehouse-id` $\neq$ `warehouse-id`

- `customer-id` o `customer-lastname`: Nel primo caso, che si verifica il 40% delle volte, il customer viene selezionato in maniera univoca attraverso il suo customer-id. Nel secondo caso, che si verifica il 60% delle volte, il customer viene selezionato attraverso il cognome. Per come viene popolato il database (ogni distretto ha 3000 customer, ma vengono generati solamente 1000 cognomi), in media 3 customer avranno lo stesso cognome. Il singolo customer viene selezionato ordinandoli per nome e prendendo quello centrale.

3.3.2 Operazioni

1. lettura da Customer: si leggono `c_balance`, `c_first`, `c_middle`, `c_last`

   (a) lettura da Customer su chiave primaria.

```
select c_balance, c_first, c_middle, c_last
from customer
where c_w_id = warehouse-id
and c_d_id = district-id
and c_id = customer-id
```

   (b) lettura da Customer su `c_last`.

```
select c_balance, c_first, c_middle, c_last
from customer
where c_w_id = warehouse-id
and c_d_id = district-id
and c_last = customer-lastname
order by c_first
```

2. lettura da Order: si legge l'ultimo ordine effettuato da un cliente (corrisponde a l'ordine con id massimo).

```
select o_id, o_entry_d, o_carrier_id,
max(o_id) as maxid from order
where o_w_id = customer-warehouse-id
and o_d_id = district-id
and o_c_id = customer-id
and o_id = maxid
```

3. lettura da Order-Line: si leggono tutte le order-line di quell'ordine.

```
select ol_i_id, ol_supply_w_id, ol_quantity,
ol_amount from order_line
where ol_w_id = customer-warehouse-id
and ol_d_id =  district-id
and ol_o_id = maxid
```

Varianti per MongoDB

Informazioni di Order e Order-Line dentro Customer La transazione di Order-Status prevede la lettura dell'ultimo ordine effettuato da un customer. Considerando la voce 1, la voce 2 e la voce 3, un approccio potrebbe essere quello di mantenere gli ordini e le linee d'ordine dentro la tabella Customer. Si avrebbe un documento contenente le informazioni del cliente con aggiunte due liste: (a) gli ordini effettuati, (b) le liste di linee d'ordine per ogni ordine (Figura 3.8b). Invece di mantenere le due liste separate (ordini, linee d'ordine), si potrebbe mantenere una unica lista di documenti "complessi" che rappresentano gli ordini contenenti la lista delle proprie linee d'ordine (Figura 3.8a).

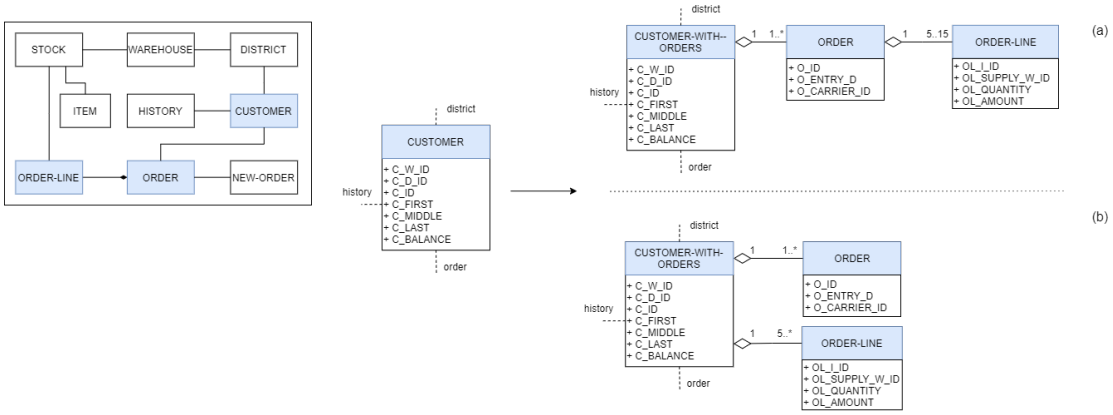


Figura 3.8: Modellazioni della relazione Customer, Order e Order-line.

In entrambi i casi si potrebbe pensare di eliminare le due tabelle Order e Order-Line, mantenendo solamente la tabella Customer. Soluzioni di questo tipo però, porterebbero alla definizione di array unbounded.

[C-last-O] Informazioni dell'ultimo ordine dentro Customer Per evitare questa problematica si possono aggiungere alla tabella Customer solamente l'informazione relativa all'ultimo ordine effettuato (Figura 3.9). In questo modo si eviterebbe la creazione di collezioni di dimensione indeterminata, riuscendo comunque a

soddisfare in maniera efficiente la transazione, che potrebbe essere risolta in un'unica lettura nel documento di Customer-With-Last-Order. La tabella strutturata in questo modo dovrebbe essere mantenuta aggiornata (ad esempio durante l'esecuzione di una transazione di tipo New-Order) e porterebbe a un overhead di memoria, per ogni warehouse, di $3k*(size(ORDER)+10*size(ORDER\_LINE))$ (considerando che in media ogni ordine si compone 10 linee d'ordine e ogni warehouse ha 3k clienti) Assumendo che size(ORDER) = 143 byte e size(ORDER_LINE) = 210 byte, la replicazione ha un costo totale di circa 7Mb.

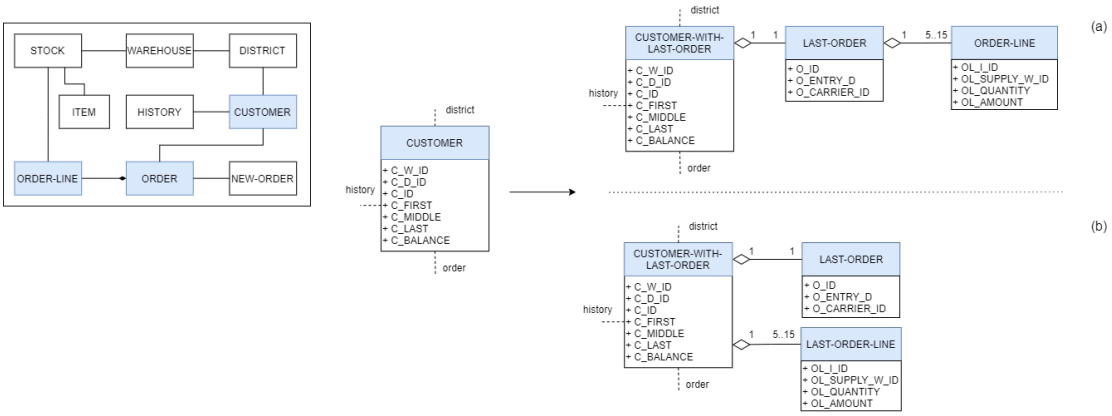


Figura 3.9: Inserimento nella tabella Customer dell'ultimo ordine effettuato.

[OOL] Lista di Order-Line dentro Order Analizzando la voce 2 e la voce 3, si nota che la lettura dell'ordine è affiancata dalle letture di tutte le sue linee d'ordine. Questo suggerisce di inserire le linee d'ordine all'interno del documento Order (Figura 3.10). Oltre ad avere un guadagno in performance, la possibilità di eliminare i campi ridondanti da Order-Line si traduce in un risparmio di memoria. Su un database scalato a un singolo warehouse, si passa ad esempio da 64Mb della versione normalizzata, a 52Mb della versione de-normalizzata.

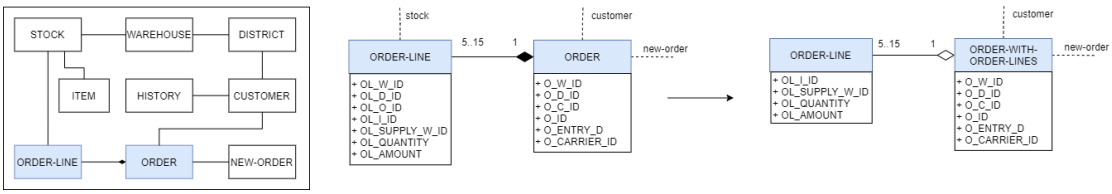


Figura 3.10: Inserimento delle linee d'ordine all'interno del documento Order.

Varianti per Cassandra

Customer indicizzato per c_last Anche in questa transazione abbiamo una lettura di Customer su chiave non primaria (`c_last`). Come già descritto nella transazione Payment (Sezione 3.2.2), è necessario in questi casi, definire una tabella Customer-By-Last, che abbia in chiave primaria il campo `c_last` (Figura 3.7).

[O-by-C] Order indicizzato per o_c_id In questa transazione si aggiunge anche la lettura degli ordini effettuati da un cliente. In maniera analoga a quanto fatto per Customer-By-Last, si definisce una tabella aggiuntiva Order-By-Customer che mantiene gli ordini indicizzati per `customer-id`, inserendo il campo in chiave. Considerando che si vuole recuperare l'ordine con il valore di `o_id` massimo, è utile mantenere la tabella ordinata per `o_id`. Per quanto riguarda l'overhead di memoria, si replica l'intera tabella Order (circa 0,4Mb). La tabella dovrà essere mantenuta aggiornata nel momento in cui si modifica la tabella Order (ad esempio durante la transazione New-Order).

```
CREATE TABLE tpcc.order(
    o_w_id int,
    o_d_id int,
    o_id int,
    o_c_id int,
    ...
PRIMARY KEY (o_w_id, o_d_id, o_id))
```

```
CREATE TABLE tpcc.order-by-customer(
    o_w_id int,
    o_d_id int,
    o_id int,
    o_c_id int,
    ...
PRIMARY KEY ((o_w_id, o_d_id, o_c_id), o_id))
WITH CLUSTERING ORDER BY (o_id DESC)
```

Figura 3.11: Schema delle tabelle Order (a sinistra) e Order-By-Customer (a destra).

Considerando nel complesso la voce 2 e la voce 3, valgono considerazioni simili a quelle fatte per MongoDB. Le informazioni di ordini e linee d'ordine vengono spesso accedute insieme, dunque è ragionevole pensare di accorpare le due tabelle.

[OL-w-O] Informazioni di Order dentro Order-Line Un primo modo potrebbe essere quello di replicare le informazioni dell'ordine su ogni linea d'ordine (Figura 3.12). Si avrebbe quindi un overhead di spazio, per ogni warehouse, relativo alla replicazione delle informazioni dell'ordine su tutte le linee d'ordine: $300k * size(ORDER)$[5]. Considerando che size(ORDER) corrisponde a circa 15 byte, risultano 4,5Mb. Inoltre, sarebbe comunque necessaria la tabella per mantenere le linee d'ordine indicizzate per cliente, raddoppiando quindi lo spazio a 9Mb.

[5]Dimensione di una riga nella tabella Order.

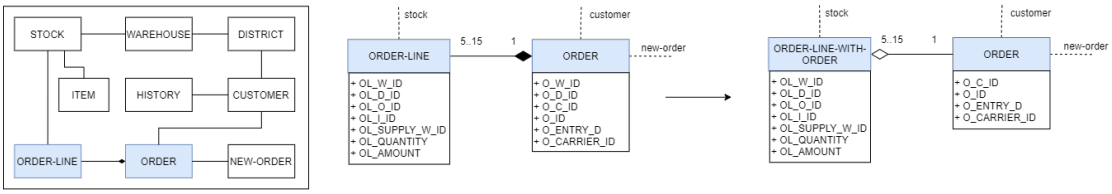


Figura 3.12: Inserimento delle informazioni di Order nella tabella Order-Line.

[OOL] Lista di Order-Line dentro Order Alternativamente, database colonnari come Cassandra, permettono di definire colonne con collezioni di elementi[6]. Dunque, si potrebbe definire una colonna nella tabella Order come collezione di linee d'ordine (Figura 3.10). In questo caso non si avrebbe più la necessità di mantenere la tabella Order-Line, risparmiando in termini di spazio, per l'eliminazione dei campi ridondanti e guadagnando in performance, per l'efficienza delle letture. Nella versione corrente di Cassandra però, collezioni di elementi con tipi definiti dall'utente (come sarebbe quello per order-line) sono gestiti come blob, e possono essere letti ed aggiornati solamente in blocco. Quindi, non sarebbe più possibile modificare o leggere singole linee d'ordine, ma solo l'intera lista.

3.4 Delivery

La transazione Delivery consiste nell'elaborazione di 10 ordini non ancora evasi. Viene processato in maniera indipendente, l'ultimo ordine non evaso di ogni distretto. La transazione ha una bassa frequenza di esecuzione, viene eseguita in modalità differita e non prevede nessun output per l'utente.

3.4.1 Input

- `warehouse-id`: chiave primaria del warehouse.
- per ogni distretto del warehouse:
  - `district-id`: chiave del distretto.
- `carrier_number`: id del corriere che spedisce l'ordine.

3.4.2 Operazioni

Per ogni distretto nel warehouse:

[6] `https://docs.datastax.com/en/cql-oss/3.3/cql/cql_using/useCollections.html`

1. Lettura su New-Order: si legge l'id dell'ordine meno recente, non ancora evaso.

```
select min(no_o_id) as last_order_id
from new_order
where no_w_id = warehouse-id
and no_d_id = district-id
```

2. Eliminazione su New-Order: si elimina dalla tabella New-Order l'ordine meno recente, non ancora evaso.

```
delete from new_order
where no_w_id = warehouse-id
and no_d_id = district-id
and no_o_id = last_order_id
```

3. Lettura su Order: si legge l'id del cliente che ha effettuato l'ordine.

```
select o_c_id as customer-id from order
where o_w_id = warehouse-id
and o_d_id = district-id
and o_id = last_order_id
```

4. Scrittura su Order: si aggiorna il valore di `o_carrier_id`.

```
update order
where o_w_id = warehouse-id
and o_d_id = district-id
and o_id = last_order_id
set o_carrier_id = carrier_number
```

5. Per ogni item nell'ordine:

   (a) Lettura su Order-Line: si calcola la somma del campo `ol_amount`.

```
select sum(ol_amount) as sum_ol_amount
from order_line
where ol_w_id = warehouse-id
and ol_d_id = district-id
and ol_o_id = last_order_id
```

   (b) Scrittura su Order-Line: si aggiorna la data di spedizione.

```
update order_line
where ol_w_id = warehouse-id
and ol_d_id = district-id
and ol_o_id = last_order_id
set ol_delivery_d = getdate()
```

6. Scrittura su Customer: si aggiorna il conto del cliente.

```
update customer
where c_w_id = warehouse-id
and c_d_id = district-id
and c_id = customer-id
set
c_balance = c_balance + sum_ol_amount,
c_delivery_cnt = c_delivery_cnt + 1
```

Varianti per MongoDB

Nella voce 3 si vuole ricavare l'id del cliente che ha effettuato uno specifico ordine. Questo suggerisce di prediligere modellazioni che mantengano integra la tabella Order. Nel caso in cui non fosse presente, l'interrogazione risulterebbe alquanto costosa. Infatti, assumendo una modellazione in cui gli ordini sono inseriti all'interno del cliente (Sezione 3.3.2), si dovrebbero potenzialmente leggere le liste degli ordini di tutti i clienti.

Anche in questa transazione si utilizza l'ordine assieme alle relative linee d'ordine (voce 4, voce 5a, voce 5b). Modellazioni che accorpano le due informazioni in unico documento (Sezione 3.3.2), avrebbero sicuramente effetti positivi sulle performance di questa transazione. Considerando che in questo caso sono richiesti anche aggiornamenti sia su Order che Order-Line, avere un documento che contiene entrambe le informazioni, permette di sfruttare metodi per la lettura e l'aggiornamento in una singola operazione di uno o più documenti (e.g findAndModify[7]).

Varianti per Cassandra

Per quanto riguarda Cassandra valgono le considerazioni fatte per la transazione Order-Status sulle modellazioni di Order e Order-Line. In questo caso, avendo anche operazioni di aggiornamento, la modellazione in (Sezione 3.3.2) risulterebbe

[7] https://docs.mongodb.com/manual/reference/command/findAndModify/

poco efficiente. Infatti, l'update dell'ordine nella voce 5a, si tradurrebbe in un aggiornamento di tutte le relative linee d'ordine (in media 10).

La soluzione che prevede la collezione di linee d'ordine dentro l'ordine (Sezione 3.3.2), risulta quindi più adeguata. Le operazioni di aggiornamento sulle singole linee d'ordine (voce 5b) andrebbero effettuate però leggendo e scrivendo l'intera lista.

Tabella New-Order ordinata per no_o_id In ultimo, si potrebbe ottimizzare la voce 1 mantenendo la tabella New-Order ordinata per `no_o_id` in maniera ascendente. Questo permetterebbe di evitare la ricerca del minimo, limitando l'interrogazione a una lettura su New-Order, selezionando poi il primo risultato.

3.5 Stock Level

La transazione Stock-Level determina il numero di articoli venduti di recente con un livello di scorta inferiore a una soglia specificata. Prevede operazioni di sola lettura ed ha una bassa frequenza di esecuzione.

3.5.1 Input

- `warehouse-id`: chiave primaria del warehouse.
- `district-id`: chiave del distretto, assieme alla chiave del warehouse identifica univocamente il distretto.
- `threshold`: soglia per il livello di scorta degli articoli.

3.5.2 Operazioni

1. Lettura su District: si legge l'id dell'ultimo ordine.

```
1 select d_next_o_id from district
2 where d_w_id = warehouse-id
3 and d_id = district-id
```

2. Lettura da Order-Line: si leggono gli id dei prodotti ordinati negli ultimi 20 ordini.

```
1 select ol_i_id from order_line
2 where ol_w_id = warehouse-id
3 and ol_d_id = district-id
4 and ol_o_id between d_next_o_id - 20 and d_next_o_id
```

3. Lettura da Stock: si conta il numero di stock sotto la soglia in input per ogni item-id.

```
select count(distinct(s_i_id)) from stock
where s_i_id = item-id
and s_w_id = warehouse-id
and s_quantity < threshold
```

Varianti per MongoDB

[D-last-I] Lista di ol_i_id dentro District Considerando la voce 1 e la voce 2, si potrebbero evitare le due letture su District e Order-Line, mantenendo dentro la tabella District la lista delle linee d'ordine relative agli ultimi 20 ordini di quel distretto. In particolare, si vuole replicare solamente il campo `ol_o_id` (senza ripetizioni), necessario per soddisfare la transazione (Figura 3.13). In media, si avrà un insieme di 200 elementi diversi. Lo spazio necessario di la replicazione, per ogni warehouse, è dunque $10*200*size(ol\_i\_id)$. Se l'id del prodotto è memorizzato come intero (8 byte), il costo di replicazione sarà equivalente a 16Kb. La tabella District-With-Last-Items dovrà essere mantenuta aggiornata a ogni aggiunta di un nuovo ordine.

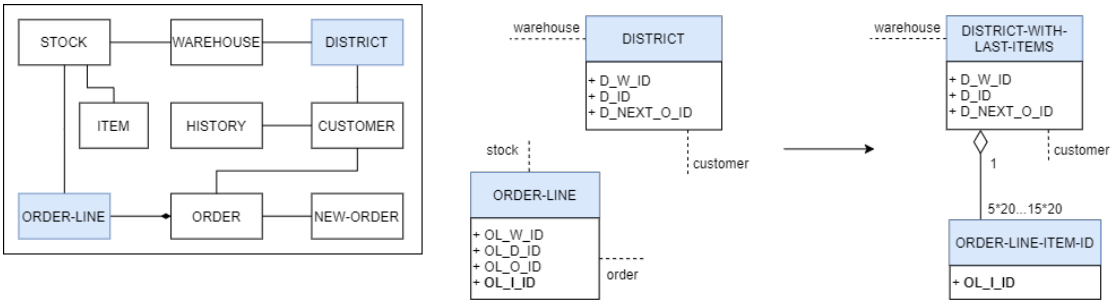


Figura 3.13: Modellazione che inserisce in District la lista dei prodotti venduti negli ultimi 20 ordini.

Lista di Stock dentro District Concentrandosi invece sulla voce 1 e la voce 3, un'idea di modellazione potrebbe essere quella di inserire all'interno del documento District una lista contente gli stock per ogni prodotto acquistato negli ultimi 20 ordini (Figura 3.14). In questo modo si potrebbe soddisfare la transazione effettuando una singola lettura sul distretto. Anche in questo caso si dovrebbe mantenere aggiornato il documento a ogni nuovo ordine. Dal punto di vista della memoria si avrebbe un overhead per ogni warehouse pari a $10 * 200 * (size(s\_i\_id) + size(s\_quantity))$. Assumendo che l'id del prodotto è

memorizzato come intero (8 byte) e la quantità come decimal (16 byte), risultano 48Kb per ogni warehouse.

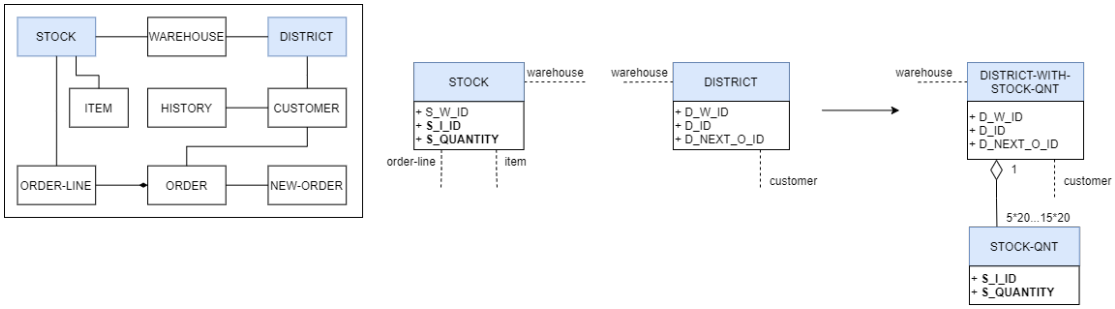


Figura 3.14: Modellazione che inserisce dentro District le informazioni degli stock per i prodotti venduti negli ultimi 20 ordini.

Varianti per Cassandra

La voce 2 prevede una lettura sulla tabella Order-Line. Valgono quindi le considerazioni fatte nelle precedenti transazioni per le tabelle Order e Order-Line (Sezione 3.3.2). In questo caso, considerando che si è interessati solamente al campo `ol_i_id` della linea d'ordine, modellazioni come quella in Sezione 3.3.2, potrebbero risultare meno efficienti, in quanto costringono la lettura dell'intera lista delle linee d'ordine con tutte le relative informazioni.

[S-by-qnt] Tabella Stock indicizzata per s_quantity Per risolvere l'interrogazione indicata dalla voce 3 è necessario definire una tabella Stock-By-Quantity, che mantenga le informazioni degli stock indicizzato per `s_quantity` (Figura 3.15). La maggiore efficienza in lettura si contrappone a un overhead di memoria dovuto alla replicazione dell'intera tabella Stock (circa 30Mb). Inoltre, ogni aggiornamento sulla tabella originale dovrà essere riportato anche sulla tabella Stock-By-Quantity.

```
CREATE TABLE tpcc.stock(
    s_w_id int,
    s_i_id int,
    s_quantity decimal,
    ...
PRIMARY KEY (s_w_id, s_i_id))
```

```
CREATE TABLE tpcc.stock-by-quantity(
    s_w_id int,
    s_i_id int,
    s_quantity decimal,
    ...
PRIMARY KEY ((s_w_id, s_i_id), s_quantity))
```

Figura 3.15: Schema delle tabelle Stock (a sinistra) e Stock-By-Quantity (a destra).

3.6 Scelta delle modellazioni

L'analisi delle transazioni appena descritta ha prodotto come risultato diverse de-normalizzazioni possibili, ognuna relativa a una specifica parte dello schema del TPC-C. Per quanto riguarda l'ambito documentale, le modellazioni complessive vengono definite combinando tutte le de-normalizzazioni. Partendo da una modellazione infatti, è possibile crearne una nuova, sostituendo una qualsiasi tabella con una de-normalizzazione che la modifica. Ad esempio, partendo dalla versione normalizzata dello schema, una modellazione possibile è quella dove si sostituisce la tabella Customer con la sua versione de-normalizzata CustomerExtended, oppure quella in cui si utilizza la tabella OrderWithOrderLines invece delle tabelle Order e Order-Line. Ovviamente, si deve evitare di combinare tra loro de-normalizzazioni "complementari", ovvero che modellano in maniera diversa la stessa relazione (e.g StockWithItem e ItemWithStocks che modellano la relazione Stock-Item). Si è subito notato che, procedendo in questo modo, l'insieme delle possibili modellazioni sarebbe stato troppo elevato per poter essere testato nella sua totalità. Si è deciso quindi di effettuare una selezione delle de-normalizzazioni ritenute più interessanti, arrivando al seguente insieme (Tabella 3.1):

- ItemWithStocks (IWS): inserisce la lista degli stock relativi ad un item all'interno del documento Item.
- StockWithItem (SWI): inserisce le informazioni dell'item nella tabella Stock.
- CustomerExtended (C-ext): estenda le informazioni del documento Customer con i campi `w_tax` e `d_tax`.
- WarehouseWithDistricts (WD): inserisce la liste dei distretti relativi ad un warehouse nel documento Warehouse.
- CustomerWithLastOrder (C-last-O): inserisce dentro il documento Customer le informazioni sull'ultimo ordine effettuato dal cliente.
- OrderWithOrderLines (OOL): inserisce dentro il documento Order la liste di tutte le sue linee d'ordine.
- DistrictWithLastItems (D-last-I): inserisce dentro il documento District gli id dei prodotto comprati negli ultimi 20 ordini.

Per ridurre ulteriormente lo spazio delle modellazioni, si è deciso di non considerare le combinazioni di de-normalizzazioni relative a una stessa tabella (e.g. CustomerExtended e CustomerWithLastOrder).

Per quanto riguarda Cassandra invece, si è scelto di lavorare sulle seguenti de-normalizzazioni (Tabella 3.2):

Stock, Item	**Customer**	**Order, Order-Line**	**Warehouse, District**
SWI	C-ext	OOL	WD
IWS	C-last-O		D-last-I

Tabella 3.1: De-normalizzazioni alterative alle tabelle normalizzate di MongoDB.

- CustomerByLast (C-by-last): mantiene la tabella Customer indicizzata per cognome (`c_last`) e ordinata per nome (`c_first`).
- OrderByCustomer (O-by-C): mantiene la tabella Order indicizzata per id del cliente (`o_c_id`).
- OrderWithOrderLines (OOL): inserisce dentro la tabella Order la liste di tutte le sue linee d'ordine.
- OrderWithOrderLinesByCustomerId (OOL-by-C): mantiene la tabella OrderWithOrderLines indicizzata per id del cliente (`o_c_id`).
- OrderLineWithOrder (OL-w-O): replica le informazioni dell'ordine su tutte le linee d'ordine.
- OrderLineWithOrderByCustomerId (OL-w-O-by-C): mantiene la tabella OrderLineWithOrder indicizzata per id del cliente (`o_c_id`).
- StockByQuantity (S-by-qnt): mantiene la tabella Stock indicizzata per quantità (`s_quantity`).

In questo caso, a differenze del documentale, non tutte le de-normalizzazioni si sostituiscono alla tabella normalizzata. Alcune di queste, come CustomerByLast o StockByQuantity, si aggiungono con l'intento rendere più efficiente l'esecuzione di determinate interrogazioni.

Customer	**Stock**	**Order, Order-Line**	
		Order	**OrderByCustomer**
C-by-last	S-by-qnt	OOL	OOL-by-C
		OL-w-O	OL-w-O-by-C

Tabella 3.2: De-normalizzazioni di Cassandra.

L'insieme delle de-normalizzazioni considerate porta alla definizione di 54 modellazioni per MongoDB e 25 per Cassandra.

Capitolo 4

Valutazione delle modellazioni

In questo capitolo verrà riportata la fase di valutazione delle modellazioni sui due database: MongoDB e Cassandra. Prima di mostrare i risultati finali, si descriverà come è stato generato il dataset del TPC-C, come si sono gestite le denormalizzazioni e quali tOOL e framework sono stati utilizzati per implementare il benchmark. In seguito, si descriverà l'implementazione del workload, spiegando alcune scelte fatte in fase di design e l'approccio utilizzato per l'implementazione. Verrà discussa la fase di esecuzione del benchmark, discutendo alcuni parametri di configurazione, come ad esempio la durata dei test e il numero di terminali. Infine, verranno commentati i risultati per i due database, dapprima singolarmente, poi mettendoli a confronto.

4.1 Generazione del dataset

Il primo passo necessario per eseguire il workload del TPC-C è la generazione del dataset. In merito al procedimento, le specifiche del TPC-C sono estremamente dettagliate, non solo nella definizione delle tabelle, ma anche nella generazione dei singoli campi. Implementare da zero uno script che generasse il dataset in maniera conforme alle specifiche, sarebbe stato decisamente troppo complesso e dispendioso in termini di tempo; per questo motivo si è deciso di utilizzare il tool OLTPBenchmark[12].

OLTPBenchmark è un framework per l'esecuzione di benchmark su database JDBC-based. Oltre a fornire funzionalità per la raccolta dei risultati, contiene l'implementazione per il workload e la generazione del dataset di alcuni tra i benchmark più famosi, compreso TPC-C. In rete è possibile trovare diversi tool per il benchmark in questione, alcuni di questi con un supporto nativo per Cassandra o MongoDB (e.g py-tpcc[1]). Si è scelto comunque di utilizzare OLTPBenchmark

[1] `https://github.com/apavlo/py-tpcc`

per la sua popolarità, chiarezza e semplicità di utilizzo. Non supportando direttamente MongoDB e Cassandra, si è dovuto generare il dataset per il database PostgreSQL, poi lo si è esportato in formato CSV, in modo da poter essere importato successivamente nei due database. Chiaramente il tool non supporta la generazione di tabelle de-normalizzate, ma gestisce solamente quelle standard previste dal benchmark. Per i dati relativi alle de-normalizzazioni dunque, si sono utilizzati script ad-hoc per i singoli database.

Per gestire alcune de-normalizzazioni di Cassandra sono state utilizzate le *viste materializzate*. Una vista materializzata è una tabella di Cassandra che viene utilizzata per gestire la de-normalizzazione. Viene creata a partire da una tabella del database e viene mantenuta aggiornata in maniera automatica da Cassandra. Ad esempio, a fronte di una modifica della tabella originaria, invece di dover manualmente aggiornare la tabella correlata, ci penserà Cassandra a gestire la consistenza. Lo strumento delle viste materializzate è ancora in fase di perfezionamento e presenta diverse limitazioni ma, in generale, la gestione automatica della de-normalizzazione con viste materializzate è più efficiente di quella manuale [1].

Il benchmark TPC-C definisce un fattore di scala per il dataset, configurabile con OLTPBenchmark, che coincide con il numero di warehouse. Per gli scopi di questa tesi si è deciso di utilizzare un fattore di scala pari a 1; questo per evitare di dovere lavorare con dataset di dimensioni troppo elevate, che avrebbero potuto creare delle difficoltà in fase di test. Supponendo infatti di dover ricreare il database ex novo ad ogni modellazione, un dataset troppo grande avrebbe comportato tempi di caricamento troppo elevati, allungando ulteriormente i tempi complessivi dei test.

Per la gestione ed il monitoraggio del database MongoDB si è utilizzato il software MongoDBCompass[2], in combinazione con l'interfaccia a riga di comando: mongoshell[3]. Per Cassandra si è utilizzato il software DBeaver[4] e l'interfaccia a riga di comando: cqlsh[5].

4.2 Implementazione del workload

I tool disponibili per l'esecuzione del benchmark TPC-C si limitano a eseguire il workload su una modellazione normalizzata del database. Considerando però modellazioni multiple, come nel caso di questa tesi, ogni transazione necessità di una implementazione ad-hoc. Per questo motivo, non è stato possibile utilizzare i tool già esistenti, ma si è dovuto implementare il workload manualmente.

[2] `https://www.mongodb.com/products/compass`
[3] `https://docs.mongodb.com/manual/mongo/`
[4] `https://dbeaver.io/`
[5] `https://cassandra.apache.org/doc/latest/tools/cqlsh.html`

Operazione	Modalità di esecuzione	Letture	Scritture
Lettura da Customer, Warehouse e District	Leggo Customer, Warehouse e District	3	
	Leggo CustomerExtended	1	
	Leggo Customer e WarehouseWithDistricts	2	
Scrittura su District	Scrivo su District		1
	Scrivo su DistrictWithLastItems		1
	Scrivo su WarehouseWithDistrict		1
Lettura da Item, Stock e aggiornare Stock	Leggo da Item e Stock, aggiorno Stock	2	1
	Leggo e aggiorno ItemWithStocks	1	1
	Leggo e aggiorno StockWithItem	1	1
Inserimento in NewOrder	Inserisco in NewOrder		1
Inserimento in Order e Order-Line	Inserisco in Order e Order-Line		∼11
	Inserisco in OrderWithOrderLines		1
Aggiornamento delle tabelle de-normalizzate (solo se previste dalla modellazione)	Aggiorno CustomerWithLastOrder		1
	Aggiorno DistrictWithLastItems		1
	Aggiorno CustomerWithLastOrder e DistrictWithLastItems		2

Tabella 4.1: Operazioni della transazione New-Order con le diverse modalità di esecuzione per MongoDB.

Il primo passo è stato quello di suddividere ogni transazione nelle operazioni di lettura e scrittura che la compongono, specificando le modalità di esecuzione sulla base delle tabelle disponibili. La Tabella 4.1 mostra un esempio di approccio sulla transazione New-Order. In questo modo, si evita una nuova implementazione della transazione per ogni modellazione, andando a modificare solamente quelle operazioni inerenti alla de-normalizzazione. Può essere fatta un'ulteriore semplificazione assumendo che alcune operazioni hanno performance simili, indipendentemente da quale tabella si scelga per la loro esecuzione. Ad esempio, nell'operazione di scrittura su District (Tabella 4.1), è ragionevole pensare che modificando il campo `d_next_o_id` in District o in DistrictWithLastItems, si abbia lo stesso costo di esecuzione. Quindi, in fase d'implementazione, è possibile evitare la gestione del caso specifico di DistrictWithLastItems in quanto riconducibile al caso della scrittura su District. Utilizzano questo espediente, si è evitato di implementare casistiche che non avrebbero apportato nessuna variazione al valore finale delle performance.

Terminata la fase di analisi sulle transazioni, si è passati all'effettiva implementazione dei workload. Si è scelto di utilizzare Scala come linguaggio, in combinazione con i driver di MongoDB[6] e Cassandra[7]. Per la gestione dei terminali simulati si è scelto un approccio ad attori utilizzando la libreria Akka[8].

Il TPC-C prevede l'utilizzo specifico di transazioni. Una transazione (nel contesto delle basi di dati), è una sequenze di operazioni che se eseguita correttamente porta una modifica del database (commit), mentre in caso di fallimento può es-

[6] `https://mongodb.github.io/mongo-scala-driver/`

[7] `https://docs.datastax.com/en/developer/java-driver/4.9/manual/`

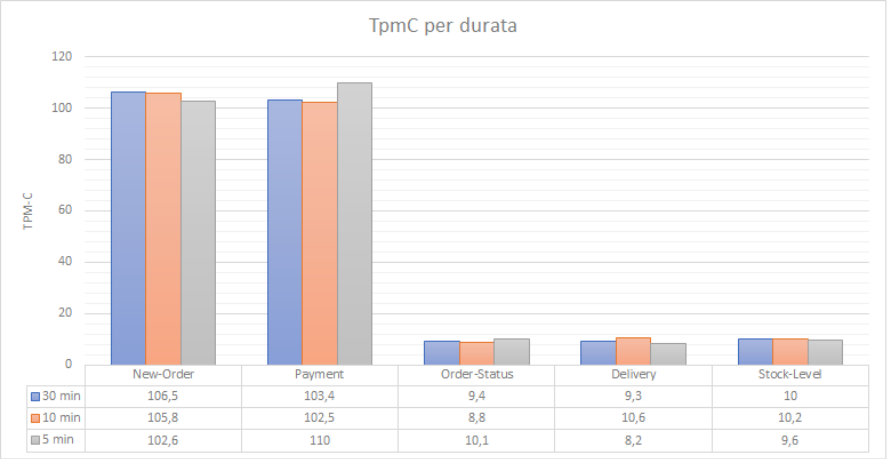
[8] `https://akka.io/`

sere completamente annullata; ovvero è possibile ritornare allo stato del sistema precedente all'esecuzione (rollback). Una transazione non può terminare in uno stato intermedio e deve godere delle cosiddette proprietà ACID. Mentre nei DBMS relazionali è comune l'implementazione delle transazioni, non tutti i database NoSQL forniscono questo supporto, come ad esempio Cassandra (MongoDB supporta le transazioni solamente dalla versione 4.0). Non potendo garantire le proprietà ACID per le transazioni in entrambi i database, si è deciso di non utilizzarle, gestendo fallimenti e rollback a livello applicativo. Questo è importante perché nel caso della transazione New-Order è prevista in maniera specifica la casistica di fallimento. Quindi, per avere dei risultati validi sulle performance, si deve tener conto anche delle operazioni necessarie a riportare il sistema in uno stato consistente, a seguito di errori.

4.3 Esecuzione del benchmark

L'esecuzione del benchmark comprende lo svolgimento del workload e la raccolta dei risultati. A tal proposito, si è realizzato un applicativo che permettesse di lanciare il benchmark su modellazioni multiple e raccogliere i dati dettagliati delle performance. Per avere una maggiore flessibilità sull'esecuzione dei test, si è fatto in modo che l'applicativo permettesse di definire configurazioni personalizzate, con parametri diversi da quelli imposti dal TPC-C. In particolare, è possibile specificare il numero di warehouse, il numero di terminali, la durata dei test e se abilitare i tempi di attesa: key time e think time. Le modellazioni da testare vengono specificate attraverso un file di testo che contiene, in ogni riga, la lista delle tabelle per la modellazione. Vengono prodotti due tipi di file in output: (a) un file per ogni modellazione che contiene i metadati del test e la lista in formato CSV dei dati relativi all'esecuzione di ogni singola transazione (b) un file unico in formato CSV che per ogni riga memorizza le informazioni "aggregate" relative alle performance e le tabelle presenti nella modellazione.

Vengono riportati come valori di performance (per ogni tipologia di transazione): il numero di esecuzioni al minuto e i tempi di risposta medi. La documentazione del TPC-C prevede di riportare, come metrica principale, il numero di transazioni di tipo New-Order (tpmC) eseguite per minuto. Si è deciso di utilizzare tempi di risposta medi poiché la metrica della transazioni-per-minuto ha il difetto di essere dipendente dalla distribuzione di probabilità con cui vengono scelte le transazioni da ciascun terminale, ad esempio, la New-Order ha una probabilità del 45%, mentre la Payment del 43%. In media si avrà sempre che su 100 transazioni eseguite, circa 45 sono New-Order e circa 43 sono Payment. Configurazioni particolarmente buone aumentano il numero di transazioni totale che si riescono a eseguire, e così il numero di transazioni completate al minuto per ogni tipologia.



	New-Order	Payment	Order-Status	Delivery	Stock-Level
30 min	106,5	103,4	9,4	9,3	10
10 min	105,8	102,5	8,8	10,6	10,2
5 min	102,6	110	10,1	8,2	9,6

Figura 4.1: Numero di transazioni al minuto per le durate: 30, 10 e 5 minuti.

In questo modo risulta difficile valutare come, una certa de-normalizzazione, impatti sulle performance di una transazione. Volendo invece analizzare nel dettaglio gli effetti di una particolare de-normalizzazione, è più utile valutare la media dei tempi di esecuzione delle singole transazioni.

4.3.1 Scelta dei parametri del TPC-C

Prima di eseguire i test su tutte le modellazioni, si è dovuto decidere quali parametri utilizzare. In particolare: la durata, il numero di terminali e l'abilitazione dei tempi di attesa previsti dal benchmark. Le specifiche del TPC-C imporrebbero una durata di 120 minuti e 10 terminali per ogni warehouse. Già ad una prima valutazione, si è notato che una durata di 120 minuti sarebbe stata difficilmente gestibile su tutte le 79 modellazioni. Dunque, si è provato a eseguire il benchmark (su una singola modellazione) per 30, 10 e 5 minuti, per poi valutare i risultati. Come mostrato in Figura 4.1, si è visto che, indipendentemente dalla durata, il numero di transazioni al minuto rimane costante. A questo punto, per limitare il tempo complessivo dei test, si è scelto di proseguire con una durata di 5 minuti.

Anche per il numero di terminali si sono effettuati diversi test per valutare le differenze di performance su una singola modellazione, modificando il parametro. I risultati sono mostrati in Figura 4.2. Come si può notare, fino a 10 terminali il numero di transazioni eseguite aumenta, poi l'aumentare dei terminali porta un peggioramento delle performance del database. Per questo motivo, e per rimanere consistenti con le specifiche del TPC-C, si è deciso di proseguire i test con 10 terminali.

In ultimo, si è valutato se mantenere oppure no i tempi di attesa previsti dal benchmark: key time e think time. La Figura 4.3 mostra come, in un test di 30

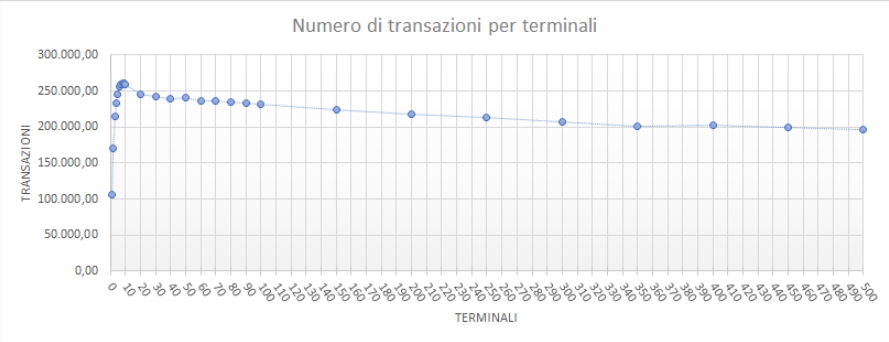


Figura 4.2: Numero di transazioni eseguite al variare del numero di terminali.

minuti, i tempi di attesa riducano drasticamente il numero transazioni eseguite al minuto. Questo è un problema poiché, se si eseguono poche transazioni, diventa difficile valutarne le performance; va considerato inoltre, che, per limitare i tempi complessivi, si vorrebbe ridurre al minimo la durata dei test. A questo punto, considerando anche che altri tool (OLTPBenchmark, py-tpcc) non implementano questa funzionalità, si è deciso di eliminare le attese.

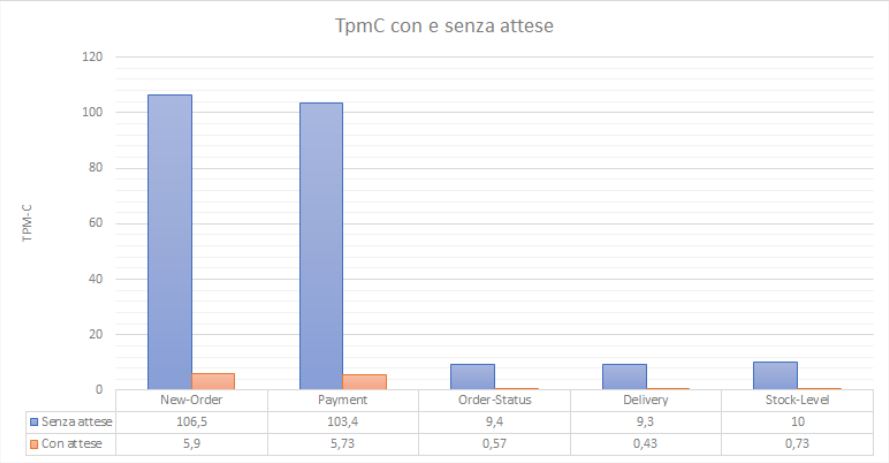


Figura 4.3: Numero di transazioni al minuto con e senza tempi di attesa: key time e think time.

4.3.2 Utilizzo di Docker

Docker[9] è una tecnologia di containerizzazione che consente la creazione e l'utilizzo di *container*. Un container è un'unità software che impacchetta il codice e tutte le

[9] `https://www.docker.com/`

sue dipendenze, in modo che un'applicazione possa girare indipendentemente su qualsiasi piattaforma. Il deploy di container viene effettuato a partire da *immagini*. Un'immagine è un pacchetto software autonomo, che include tutti i componenti necessari per eseguire un'applicazione: codice, librerie, variabili di ambiente e file di configurazione.

Docker è stato utilizzato nello sviluppo della tesi per garantire l'isolamento dei test e per favorire la portabilità dell'applicazione. Sono state utilizzate le immagini ufficiali di MongoDB[10] e Cassandra[11] appositamente modificate in modo che il dataset venisse caricato in maniera automatica all'esecuzione del container. In fase di test, per ogni modellazione viene lanciato un nuovo container Docker con l'immagine del database, e su questo si esegue il workload. Su MongoDB, tutte le tabelle, normalizzate e de-normalizzate, vengono caricate all'avvio. La creazione delle viste materializzate in Cassandra viene invece gestita a livello applicativo; é importante infatti che vengano generate solamente le viste materializzate presenti nelle configurazione che si vuole testare, poiché non è possibile disabilitarne la gestione automatica da parte di Cassandra.

Il container viene eliminato al termine dell'esecuzione, per poi essere ricreato alla modellazione successiva. In questo modo, ogni esecuzione è indipendente ed isolata dalle altre.

4.3.3 Indici

Per fare un'analisi esaustiva dei due database, si è deciso di effettuare i test con implementati degli indici. MongoDB supporta la definizione di indici primari e secondari, può velocizzare notevolmente l'esecuzione delle query evitando di eseguire la lettura di un'intera collezione e ritornando anche il risultato completo utilizzando solamente i valori indicizzati (covered index queries[12]). Una *covered query* è una query che può essere soddisfatta interamente utilizzando un indice e non deve esaminare nessun documento. Quindi, oltre a indici su chiavi primarie, sono stati implementati indici per avere interrogazioni di tipo "covered". In particolare sono stati utilizzati indici di tipo *compound* e di tipo *multikey*.

In Cassandra è possibile utilizzare gli indici per eseguire interrogazioni su attributi non in chiave primaria. Gli indici possono essere utilizzati per collezioni, colonne di collezioni e qualsiasi altra colonna tranne le colonne contatore, le colonne statiche e le colonne in chiave primaria (le colonne in chiave primaria sono automaticamente indicizzate). L'utilizzo di viste materializzate, de-normalizzazioni e replicazione dei dati è comunque una soluzione preferibile alla definizione di indici,

[10] `https://hub.docker.com/_/mongo`
[11] `https://hub.docker.com/_/cassandra`
[12] `https://docs.mongodb.com/manual/indexes/#covered-queries`

Tabella	**Indici**
Warehouse	{w_id: 1, w_tax: 1}
District	{d_w_id: 1, d_id: 1, d_next_o_id: 1, d_tax: 1}
Customer	{c_w_id: 1, c_d_id: 1, c_id: 1}
	{c_w_id: 1, c_d_id: 1, c_last: 1}
Order	{o_w_id: 1, o_d_id: 1, o_id: 1, o_c_id: 1}
	{o_w_id: 1, o_d_id: 1, o_id: 1, o_c_id: 1, o_carrier_id: 1, o_entry_d: 1}
Order-Line	{ol_o_id: 1, ol_d_id: 1, ol_w_id: 1, ol_number: 1}
	{ol_o_id: 1, ol_d_id: 1, ol_w_id: 1, ol_i_id: 1, ol_amount: 1}
New-Order	{no_w_id: 1, no_d_id: 1, no_o_id: 1}
Item	{i_id: 1}
Stock	{s_w_id: 1, s_i_id: 1, s_quantity: 1}

Tabella 4.2: Indici implementati in MongoDB. I campi tra graffe indicano indici di tipo *compound*, mentre il numero "1" indica l'ordinamento ascendente. Gli stessi indici sono stati riportati nelle tabelle de-normalizzate, usando indici di tipo *multikey* dove necessario.

Tabella	**Indici**
Customer	c_last
Order	o_c_id
OrderWithOrderLines	
OrderLineWithOrders	
Order-Line	ol_o_id
Stock	s_quantity

Tabella 4.3: Indici implementati in Cassandra.

soprattutto in un contesto distribuito. In questo caso, sono stati implementati tutti gli indici che permettessero di eseguire le interrogazioni senza l'utilizzo di tabelle ausiliarie.

4.4 Analisi dei risultati

L'analisi dei risultati si basa sui dati aggregati relativi all'esecuzione del workload su tutte le modellazioni. In particolare, oltre al numero di transazioni totale, che fornisce una metrica per valutare la bontà di una modellazione, si considera anche la media dei tempi di risposta misurati dai terminali per ogni transazione. In questo modo, è possibile valutare l'impatto di una de-normalizzazione sia nelle performance complessive che nel dettaglio delle singole transazioni.

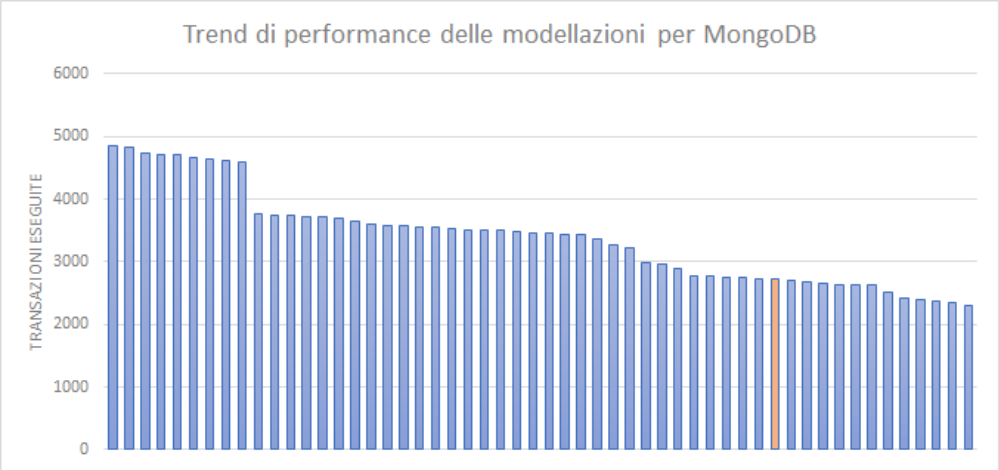


Figura 4.4: Numero di totale di transazioni eseguite per ogni modellazione di MongoDB. In arancione è indicato il valore della modellazione normalizzata.

4.4.1 MongoDB

Per quanto riguarda MongoDB, il trend di performance delle modellazioni è visibile in Figura 4.4. In riferimento al numero di transazioni, si può notare come la scelta di una modellazione possa modificare in maniera radicale le performance, passando da 2310 a 4864 transazioni. Di seguito si mostrano le migliori 5 modellazioni per numero di transazioni eseguite:

1. History, `C-ext`, `OOL`, New-Order, `SWI`, Warehouse, District
2. History, `C-ext`, `OOL`, New-Order, `SWI`, `WD`
3. History, `C-last-O`, `OOL`, New-Order, `SWI`, `WD`
4. History, Customer, `OOL`, New-Order, `SWI`, Warehouse, `D-last-I`
5. History, `C-last-O`, `OOL`, New-Order, `SWI`, Warehouse, District

Da queste, si può già intuire quali de-normalizzazioni siano più efficaci per il workload del TPC-C. Ad esempio `OOL` e `SWI`, che compaiono in tutte le 5 modellazioni, risultano essere particolarmente buone, anche osservando il dettaglio della Tabella 4.4. Come si può notare, `OOL` porta un evidente incremento di performance, visibile su tutte le transazioni. La de-normalizzazione `SWI` invece, anche se non apporta un miglioramento della stessa portata, si concentra in particolare sulla New-Order, permettendo un'esecuzione circa il 25% più veloce. Essendo la transazione New-Order quella eseguita con maggiore frequenza, un miglioramento su questa transazione si traduce in un incremento importante nelle performance

	Customer			**Stock, Item**			**Warehouse, District**			**Order, OrderLine**	
		C-ext	C-last-O		SWI	IWS		WD	D-last-I		OOL
New Order	1478	-0,14%	15,36%	1723	-24,96%	-4,64%	1534	-1,04%	4,82%	1738	-21,29%
Payment	205	0,49%	11,71%	198	-1,01%	24,24%	212	-0,94%	2,83%	242	-23,55%
Order-Status	1271	0,71%	-80,88%	873	-2,75%	22,79%	921	-3,15%	6,51%	1557	-80,35%
Delivery	3013	-0,23%	76,00%	3575	-2,04%	18,74%	3696	-0,81%	7,14%	6673	-86,89%
Stock-Level	1190	1,51%	23,78%	1204	-4,82%	26,33%	1656	-1,45%	-64,79%	1970	-68,98%

Tabella 4.5: Effetti delle de-normalizzazioni sulle transazioni. Il miglioramento (o peggioramento) è espresso come differenza percentuale rispetto alla versione normalizzata. I valori si riferisco ai tempi medi di esecuzione espressi in millisecondi.

complessive. La de-normalizzazione alternativa `IWS` risulta invece poco efficace. Le basse prestazioni potrebbero essere dovute al fatto che la collezione di stock dentro il documento Item, considerando un fattore di scala pari ad 1 warehouse, contiene sempre un solo elemento, rendendo di fatto inutile la gestione attraverso un array.

I miglioramenti delle de-normalizzazioni `WD` e `C-ext` sono di fatto trascurabili. Dai dati si evince comunque una tendenza positiva che potrebbe essere accentuata ripetendo i test scalando il dataset a valori più elevati.

Le de-normalizzazioni `C-last-O` e `D-last-I` portano un miglioramento "mirato" alle transazioni Order-Status e Stock-Level. In questo caso però, abbiamo anche la necessità di mantenere aggiornata e consistente la replicazione, pagando il costo nell'esecuzione delle altre transazioni. Questo è evidente soprattutto nella `C-last-O` che mostra un miglioramento dell'80% nella Order-Status e un peggioramento del 76% nella Delivery, dove viene mantenuta aggiornata.

L'aggiunta di indici porta a un miglioramento notevole da punto di vista delle performance ma, con un dataset di queste dimensioni, diventa più difficile percepire i miglioramenti delle singole de-normalizzazioni. Il numero di transazioni eseguite passa da 4864 (migliore modellazione senza indici) a 287163 (migliore modellazione con indici) transazioni. Analizzando le differenze percentuali, come per la Tabella 4.4, si confermano, in maniera meno evidente, le considerazioni fatte in precedenza per le modellazioni non indicizzate. La differenza principale è sulla de-normalizzazione WD, che nel caso con indici, risulta essere particolarmente inefficiente se confrontata con la versione normalizzata.

4.4.2 Cassandra

Per Cassandra, il trend di performance è mostrato in Figura 4.5. In questo caso si passa dalle 229898 transazioni della migliore modellazione, alle 54778 della peggiore.

Le migliori 5 sono elencate di seguito:

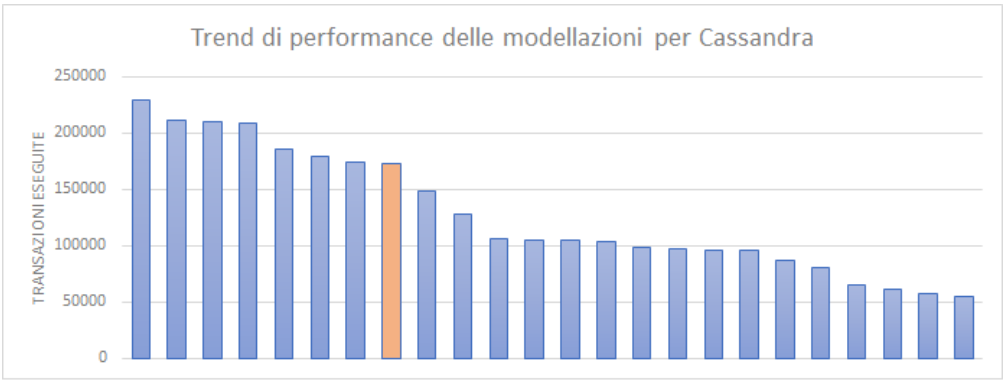


Figura 4.5: Numero di totale di transazioni eseguite per ogni modellazione di Cassandra. In arancione è indicato il valore della modellazione normalizzata.

1. Stock, History, `OOL-by-C`, `OOL`, New-Order, `C-by-last`, Customer, District, Warehouse, Item

2. Stock, Order-Line, Order, History, New-Order, `C-by-last`, Customer, District, Warehouse, Item

3. Stock, History, `OOL`, New-Order, `C-by-last`, Customer, District, Warehouse, Item

4. Stock, Order-Line, Order, History, New-Order, `C-by-last`, `O-by-C`, Customer, District, Warehouse, Item

5. Stock, History, `OOL-cass`, `OOL`, New-Order, Customer, District, Warehouse, Item

La Tabella 4.6 mostra le performance dettagliate delle de-normalizzazioni. Si riportano anche i valori per le tabelle implementate con indici, in modo che sia possibile confrontare le due alternative. Si può notare come, anche in questo caso, la de-normalizzazione `OOL` porti dei miglioramenti, ancora più evidenti se supportata dalla tabella secondaria `OOL-by-C`, al contrario di `OL-w-O` che si rivela invece una scelta del tutto inefficiente, soprattutto nella Order-Status. La de-normalizzazione `S-by-qnt` apporta dei miglioramenti ma i costi di aggiornamento nell'esecuzione della New-Order non giustificano il suo utilizzo.

A differenza di MongoDB, l'implementazione con indici non porta dei miglioramenti evidenti rispetto alla versione senza indici. Da notare il fatto che la migliore modellazione senza indici comprenda diverse de-normalizzazioni (`OOL`, `OOL-by-C`, `C-by-last`), mentre quella con indici sia quella completamente normalizzata. Il

	Customer			Stock			Order, OrderLine			OrderByCustomer, OrderLine			Order, OrderLine index		
		C-by-last	C-index		S-by-qnt	S-index		OL-w-O	OOL		OL-w-O-by-C	OOL-by-C		OL-w-O	OOL
New Order	39,3	-4,33%	9,92%	19,8	188,38%	5,20%	36,75	-14,83%	-4,08%	37,75	37,09%	-1,99%	40,75	-1,23%	-2,45%
Payment	12,7	-38,58%	-44,25%	13,3	-46,62%	-17,89%	10,75	14,42%	-4,65%	10	-7,50%	-2,50%	8,25	0,00%	-3,03%
Order-Status	133,3	3,53%	-97,06%	150,2	-19,37%	-96,62%	15	4922,00%	100,00%	6,25	-20,00%	-24,00%	4,5	16,67%	-27,78%
Delivery	35,7	33,89%	-1,04%	48,8	-28,89%	2,46%	36,25	4,28%	-8,28%	46,75	13,90%	-6,42%	36,75	-6,80%	-8,84%
Stock-Level	16,7	-1,20%	7,31%	14,9	22,82%	23,62%	15,25	47,54%	-13,11%	14,75	20,68%	-6,78%	18,5	13,51%	-14,86%

Tabella 4.7: Performance delle de-normalizzazioni e delle tabelle indicizzate sulle transazioni. Il miglioramento (o peggioramento) è espresso come differenza percentuale rispetto alla versione normalizzata senza indici. I valori si riferisco ai tempi medi di esecuzione espressi in millisecondi.

motivo è che, utilizzando gli indici, si evitano i costi necessari a mantenere aggiornate le tabelle de-normalizzate. In ogni caso, a parità di performance, è consigliabile optare per una soluzione de-normalizzata, poiché la gestione degli indici e legata al singolo nodo, e non avrebbe la stessa efficacia in un contesto distribuito.

4.5 Database a confronto

Confrontando i due database nella versione senza indici, si notano immediatamente le performance superiori di Cassandra rispetto a MongoDB. Infatti, anche la migliore modellazione di MongoDB (4864 transazioni), non riesce a eseguire un numero di transazioni paragonabile a quello di Cassandra (54778 transazioni nella peggiore modellazione). Considerando invece la versione con indici le performance si avvicinano, con un vantaggio di MongoDB. Infatti, con quest'ultimo si sono raggiunte 287163 transazioni, mentre con Cassandra 228377.

Un altro confronto che si può fare è relativo alle dimensioni del database. Infatti, considerando il dataset normalizzato, scalato ad 1 warehouse, su MongoDB si ha un'occupazione di memoria pari a 160MB, mentre su Cassandra 71MB.

4.6 Indicatore di performance

Sulla base dell'analisi delle transazioni, come quella mostrata in Tabella 4.1, si è costruito un indicatore che potesse valutare a priori la bontà delle de-normalizzazioni sulle transazioni.

Per prima cosa si considerano il numero totale di operazioni (letture e scritture) che compongono una transazione. Poi si conta il numero di operazioni che si aggiungono o tolgono all'esecuzione utilizzando una specifica de-normalizzazione; si vuole infatti utilizzare il numero di operazioni risparmiate come approssimazione della bontà della de-normalizzazione. Ad esempio, considerano la transazione New-Order Tabella 4.1, la presenza della tabella `D-last-I` permette di risparmiare una lettura nella Stock-Level, ma richiede una scrittura aggiuntiva nella New-Order.

Si associa quindi il valore +1 per la New-Order e -1 per la Stock-Level. I valori vengono poi scalati in base alla dimensione della tabella de-normalizzata. In particolare, si considera il peso della tabella nell'occupazione di memoria complessiva della modellazione (in valore percentuale). In questo modo avremo un valore più elevato per le tabelle più grandi, che generalmente hanno un'influenza maggiore nelle performance della modellazione. In ultimo, si calcola una media pesata sulla base della distribuzione di probabilità delle transazioni nel carico di lavoro. Infatti, una modellazione che porta un risparmio nelle transazioni più frequenti (e.g New-Order e Payment) è preferibile, poiché si sfrutteranno più spesso i suoi vantaggi. Allo stesso modo, si vogliono penalizzare quelle de-normalizzazioni che richiedono operazioni aggiuntive in transazioni frequenti.

Di seguito una descrizione formale per il calcolo dell'indicatore. Si assume un carico di lavoro con N transazioni ed un vettore $p_i \in [0,1]^N$ che associa all'i-esima transazione un valore compreso tra 0 e 1, relativo alla sua frequenza di esecuzione. Nel caso del TPC-C:

$$N = 5$$

$$p_i = (0.45, 0.43, 0.04, 0.04, 0.04)$$

Si considera una modellazione con un'occupazione totale di memoria M e una tabella de-normalizzata, appartenente alla modellazione, con occupazione di memoria m. Quindi, si indica con α il contributo della tabella de-normalizzata al peso complessivo della modellazione, in valore percentuale. Inoltre, si indica con $w_i \in \mathbb{Z}^N$ il vettore che associa all'i-esima transazione le operazioni risparmiate (o aggiunte), utilizzando la tabella de-normalizzata durante l'esecuzione. A questo punto, il valore dell'indicatore per la de-normalizzazione si può calcolare come:

$$\alpha = \frac{m}{M}$$

$$KPI = \frac{\sum_{i=1}^{N}(\alpha \cdot w_i \cdot p_i)}{\sum_{i=1}^{N} p_i}$$

Considerando che $\sum_{i=1}^{N} p_i = 1$ il calcolo si semplifica in:

$$KPI = \sum_{i=1}^{N}(\alpha \cdot w_i \cdot p_i)$$

In Tabella 4.8 si mostrano i valori ottenuti per le de-normalizzazioni di MongoDB. Più piccolo il valore, maggiore è l'efficacia della de-normalizzazione. Confrontando questa tabella con la Tabella 4.4, si confermano diverse considerazioni già discusse, come ad esempio la validità della de-normalizzazione `OOL`.

		Customer		**Stock, Item**		**Warehouse, District**		**Order, Order-Line**
		C-ext	C-last-O	SWI	IWS	WD	D-last-I	OOL
New Order	45%	-11.617	16.953	-14.769	-14.926	-0.001	0.004	-116.620
Payment	43%					-0.001		
Order-Status	4%		-50.858					-11.662
Delivery	4%		508.581					-1166.203
Stock-Level	4%						-0.004	
KPI		-5.228	25.938	-6.646	-6.717	-0.001	0.002	-99.594

Tabella 4.9: Indicatori di performance per le de-normalizzazioni di MongoDB.

		Customer	**Stock**	**Order, Order-Line**		**Order, OrderLineByCustomer**		
		C-by-last	S-by-qnt	OL-w-O	OOL	O-by-C	OL-w-O-by-C	OOL-by-C
New Order	45%		2.249	-9.206	-79.834	1.504	119.833	9.080
Payment	43%	10.647						
Order-Status	4%			-9.206	-7.983			
Delivery	4%	106.467		-92.063		15.043	1198.332	90.799
Stock-Level	4%							
KPI		8.837	1.012	-8.194	-36.245	1.279	101.858	7.718

Tabella 4.11: Indicatori di performance per le de-normalizzazioni di Cassandra

La stessa analisi è stata riportata su Cassandra (Tabella 4.10). In questo caso però, alcune de-normalizzazioni, come ad esempio `S-by-qnt` e `C-by-last`, pur migliorando l'esecuzione, non diminuiscono il numero di operazioni necessarie all'esecuzione della transazione. Di fatto, richiedono solamente operazioni di scrittura aggiuntive per essere mantenuta aggiornate. In questi casi, un valore basso può essere interpretato come un basso costo di mantenimento della de-normalizzazione, mentre un valore alto significherebbe un costo di mantenimento maggiore.

Considerando invece le tabelle Order e Order-Line, si nota come `OOL` sia migliore di `OL-w-O`, come già evidente dai risultati del benchmark in Tabella 4.6.

Conclusioni

L'obiettivo della tesi era quello di analizzare le strategie di modellazione dei dati in database NoSQL. È stata sviluppata un'implementazione del benchmark TPC-C sui i database MongoDB e Cassandra, per valutare come diverse modellazioni non relazionali influenzassero le performance nell'esecuzione del workload. Si sono analizzati i risultati con e senza l'implementazione di indici, valutando i database singolarmente e poi mettendoli a confronto. Si è dimostrato come la scelta della modellazione sia cruciale nel determinare le performance del sistema, sottolineando la necessità di sviluppare metodologie adeguate per la progettazione di database non relazionali, che permettano di sfruttare al meglio le tecniche di modellazione NoSQL. Analizzando i due database in esame: MongoDB e Cassandra, si è visto come le performance dei database, senza l'implementazione di indici, non siano paragonabili, con un netto vantaggio di Cassandra. L'implementazione di indici invece, avvicina le performance, portando MongoDB a superare Cassandra. Infine, è stato progettato un indicatore che permette di avere una valutazione a priori sull'efficacia di una determinata modellazione de-normalizzata: si utilizzano il numero di operazioni di letture e scrittura risparmiate utilizzando la tabella de-normalizzata, come approssimazione della sua efficacia, tenendo conto della dimensione della tabella e del carico di lavoro previsto. Dunque, si è mostrata la correlazione dell'indicatore con i dati ottenuti dall'esecuzione del benchmark, dimostrando quindi la sua efficacia.

L'analisi proposta si potrebbe estendere lavorando su aspetti non approfonditi in questa tesi.

- **Dimensione del dataset**: l'analisi si è concentrata solamente sul dataset scalata a un warehouse; seppur sufficiente per l'obiettivo di questa tesi, si potrebbe approfondire lo studio osservando come le diverse modellazioni si comportano in dataset di dimensioni maggiori, così da apprezzare pregi e difetti non osservabili a scale ridotte. Inoltre, l'implementazione di indici in dataset di dimensioni limitate, rende l'esecuzione estremamente veloce, appiattendo le differenze di una modellazione rispetto a un'altra.

- **Database distribuiti**: entrambi i database utilizzati nella tesi, supportano la distribuzione come meccanismo per garantire scalabilità e prestazioni. Potrebbe essere interessante valutare quali de-normalizzazioni si adattano meglio all'implementazione su un'infrastruttura distribuita, anche in relazione al modello di consistenza che si vuole mantenere.

- **Valutazione dell'indicatore**: un ulteriore sviluppo potrebbe essere quello di effettuare lo stesso studio su un diverso database o benchmark. In questo modo, si potrebbe valutare la validità dell'indicatore proposto anche in contesti diversi da quelli mostrati. Nello specifico, sarebbe interessante analizzare le famiglie di database key-value e a grafo, non considerate in questa tesi.

Appendice A

Workload di UniBench

Tabella A.1: Workload di UniBench [18]

Nome	Categoria di Business	Dimensione Tecnica	Descrizione
Q1	Individuale	Esegue una query su i dati di un customer, presi da tutti i modelli	Dato un customer, trova profilo, ordini, feedback, e post
Q2	Dialogo	Esegue un join tra Relazionale, Grafo, e JSON	Dato un prodotto, trova le persone che lo hanno comprato e i post che lo riguardano
Q3	Dialogo	Esegue un join tra Relazionale, Grafo, e Chiave-valore, filtra dati strutturati e non strutturati	Dato un prodotto, trova le persone che hanno commentato e postato informazioni che lo riguardano, e trova opinioni negative
Q4	Collettiva	Aggrega e ordina gli ordini JSON, Esegue il 3-hop graph traversal nel sotto-grafo, ritorna l'intersezione tra i due insiemi	Trova le prime 2 persone che hanno speso più soldi in ordini. Poi, per ogni persona, attraversa il "knows-graph" (3-hop) per trovare gli amici, e infine trova gli amici che le due persone hanno in comune

Tabella A.1: Workload di UniBench [18]

Nome	Categoria di Business	Dimensione Tecnica	Descrizione
Q5	Collettiva	Esegue un join tra Relazionale, Grafo, e Chiave-valore con due predicati, ricerca ricorsiva sul Grafo, array innestato sul JSON, e ricerca per chiave composta per il Chiave-valore	Dato un customer ed una categoria di prodotto, trova le persone che sono sue amiche nel grafo delle amicizie (3-hop), e che hanno comprato prodotti nella stessa categoria. Infine, ritorna i feedback con recensione a 5 stelle per i prodotti comprati
Q6	Collettiva	Trova il percorso minimo calcolato tra due nodi, trova gli ordini JSON correlati ai nodi nel percorso, esegue un'aggregazione sugli ordini JSON trovati	Dati customer 1 e customer 2, trova le persone nel percorso minimo tra il loro nel sotto-grafo delle amicizie, e trova i primi 3 prodotti più venduti tra i loro acquisti
Q7	Commerciale	Esegue un join tra Relazionale, JSON e Chiave-valore, compara i risultati di aggregazione tra due periodi, identifica le recensioni con opinioni negative	Dati i prodotti di un venditore con vendite in calo compara l'ultimo quadrimestre, analizza le recensioni per quei prodotti per vedere se ci sono opinioni negative
Q8	Commerciale	Esegue operazioni di filtro e aggregazione sull' array innestato degli ordini JSON, aggrega i dati del grafo correlati per ogni elemento	Per tutti i prodotti di una data categoria durante una anno, calcola la vendita totale, e misura la popolarità nei social media

Tabella A.1: Workload di UniBench [18]

Nome	Categoria di Business	Dimensione Tecnica	Descrizione
Q9	Commerciale	Esegue operazioni di filtro, aggregazione ed ordinamento sull'array innestato degli ordini JSON, poi trova i dati correlati nel grafo	Trova le prime 3 aziende che hanno le vendite maggiori nel paese, per ogni azienda, compara il numero di customer maschi e femmine, e ritorna il loro post più recenti
Q10	Commerciale	Esegue l'aggregazione e l'ordinamento dei dati su grafo, poi trova i dati correlati in Chiave-valore e JSON	Trova le prime 10 persone più attive aggregando i post dell'ultimo anno, poi calcola la loro valore RFM (Recency, Frequency, Monetary) nello stesso periodo, e ritorna le recensioni e i tag più recenti di loro interesse
T1	New order transaction	Controlla le proprietà ACID e valuta l'efficienza su transazioni read-heavy multi-model che comprendono JSON e XML	(i) Crea ed inserisce l'ordine, (ii) aggiorna la quantità dei prodotti relativi, (iii) inserisce la fattura
T2	Payment transaction	Controlla le proprietà ACID e valuta l'efficienza su transazioni write-heavy multi-model che comprendono Relazionale, JSON e XML	(i) Recupera un ordine non pagato, (ii) aggiorna il conto del compratore e del venditore, (iii) aggiorna lo stato d'ordine a pagato, (iv) aggiorna la relativa fattura

Bibliografia

[1] Materialized view performance in cassandra 3.x. https://www.datastax.com/blog/materialized-view-performance-cassandra-3x.

[2] Nosql data modeling techniques. https://highlyscalable.wordpress.com/2012/03/01/nosql-data-modeling-techniques/.

[3] Veronika Abramova and Jorge Bernardino. Nosql databases: Mongodb vs cassandra. C3S2E '13, page 14–22, New York, NY, USA, 2013. Association for Computing Machinery.

[4] Veronika Abramova, Jorge Bernardino, and Pedro Furtado. Evaluating cassandra scalability with ycsb. In Hendrik Decker, Lenka Lhotská, Sebastian Link, Marcus Spies, and Roland R. Wagner, editors, *Database and Expert Systems Applications*, pages 199–207, Cham, 2014. Springer International Publishing.

[5] Yusuf Abubakar, Thankgod Sani Adeyi, and Ibrahim Gambo Auta. Performance evaluation of nosql systems using ycsb in a resource austere environment. *Performance Evaluation*, 7(8):23–27, 2014.

[6] Paolo Atzeni, Francesca Bugiotti, Luca Cabibbo, and Riccardo Torlone. Data modeling in the nosql world. *Computer Standards Interfaces*, 67:103149, 2020.

[7] Antonio Badia and Daniel Lemire. A call to arms: Revisiting database design. *Sigmod Record*, 40, 05 2011.

[8] Philippe Bonnet and Dennis Shasha. *Database Benchmarks*, pages 936–938. Springer New York, New York, NY, 2018.

[9] Kristina Chodorow and Michael Dirolf. *MongoDB: The Definitive Guide*. O'Reilly Media, Inc., 2010.

[10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[11] Wei Dai and Daniel Berleant. Benchmarking contemporary deep learning hardware and frameworks: A survey of qualitative metrics. *2019 IEEE First International Conference on Cognitive Machine Intelligence (CogMI)*, Dec 2019.

[12] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proc. VLDB Endow.*, 7(4):277–288, December 2013.

[13] Eben Hewitt. *Cassandra: The Definitive Guide*. O'Reilly Media, Inc., 2010.

[14] Asya Kamsky. Adapting tpc-c benchmark to measure performance of multi-document transactions in mongodb. *Proc. VLDB Endow.*, 12(12):2254–2262, August 2019.

[15] Scott T Leutenegger and Daniel Dias. A modeling study of the tpc-c benchmark. *ACM Sigmod Record*, 22(2):22–31, 1993.

[16] D. Rolls, C. Joslin, and S. Scholz. Unibench: A tool for automated and collaborative benchmarking. In *2010 IEEE 18th International Conference on Program Comprehension*, pages 50–51, 2010.

[17] Pramod J. Sadalage and Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional, 1st edition, 2012.

[18] Chao Zhang, Jiaheng Lu, Pengfei Xu, and Yuxing Chen. Unibench: A benchmark for multi-model database management systems. In Raghunath Nambiar and Meikel Poess, editors, *Performance Evaluation and Benchmarking for the Era of Artificial Intelligence*, pages 7–23, Cham, 2019. Springer International Publishing.