Gestione e Configurazione di AdventureWorksLT2019

Il database di riferimento per il progetto BikeVille fornito dal docente è il punto di partenza. Si è deciso di concentrarsi inizialmente sulla struttura e la sua ottimizzazione, adottando un approccio di tipo DB-First, che implica avere un database completo e basato su connessioni logiche solide. A tal proposito, è stata studiata la documentazione ed è stata condotta un'analisi approfondita del database fornito.

Modifiche, Idee e Ottimizzazioni

Modifiche alla Tabella Customer

La prima modifica effettuata riguarda la tabella Customer: i dati sensibili, come indirizzo email, password cifrata e numero di telefono, sono stati rimossi dal database in SSMS e aggiunti a un database in MongoDB per motivi di sicurezza. In aggiunta, in MongoDB è stata creata anche una tabella con i dati sensibili degli amministratori della pagina, per permettere il loro log-in per le modifiche ai vari articoli.
Per permettere la comunicazione tra la tabella Customer nel database in SSMS e quella in MongoDB, la colonna 'EmailAddress' è stata impostata come indice unico: in questo modo si riesce a identificare univocamente il cliente da una tabella all'altra. Inoltre, questa operazione permette di evitare che due utenti si registrino con la stessa email. Per implementare questa modifica è stato necessario eliminare i clienti "doppi", ossia quelli registrati due volte, i cui record differivano unicamente per il 'CustomerID'.

Modifiche alla Tabella Product

Successivamente, tramite una inner-join con il database principale dell'e-commerce (AdventureWorks), sono state aggiunte alla tabella Product le colonne che riportano le informazioni per la foto di dimensioni normali da aggiungere al sito per rappresentare i vari articoli (LargePhoto e LargePhotoFileName).

Implementazione dei Trigger

Per tenere conto di tutte le operazioni che avvengono nel database e registrarle nella tabella OperationLog, sono stati implementati i trigger che vengono eseguiti dopo le varie attività di CRUD in ogni tabella; in ogni trigger è presente una gestione degli errori, i quali vengono a loro volta registrati nella tabella ErrorLog.

Stored Procedure

Sono state implementate anche due stored procedure per la visualizzazione dello storico ordini tramite l'ID del cliente e per la visualizzazione dei vari dettagli di ogni ordine tramite l'ID del dettaglio di vendita di ogni ordine.
Per la gestione degli errori nel backend, è stata creata un'ulteriore stored procedure, che prende come parametro l'errore avvenuto e lo registra nell'apposita tabella ErrorLog.

Gestione del Carrello e Wishlist

Per la gestione di un e-commerce, uno degli strumenti fondamentali è il Carrello e la Wishlist dell'utente registrato al sito. In merito a questo, ci si è accorti che il database a disposizione non gestiva questo aspetto, di conseguenza sono state introdotte due tabelle dedicate a questo scopo:

- **WishList** ([CustomerID]PK/FK, [ProductID]PK/FK, Add_Date)
- **Cart** ([CustomerID]PK/FK, [ProductID]PK/FK, Add_Date, Quantity)

Tuttavia, mantenere i dati in due tabelle separate potrebbe comportare difficoltà future per la gestione degli ordini e dei dati stessi, che sono considerabili come dati dinamici, essendo che la loro presenza nelle tabelle non è persistente. Dunque, si è passati a una seconda versione della medesima idea: fondere le due tabelle in una sola che mette in relazione gli utenti e i prodotti (presenti nella wishlist o carrello), in modo tale da centralizzare i dati in un'unica tabella. È stata così creata la tabella:

- **CustomerProduct** ([CustomerID]PK/FK, [ProductID]PK/FK, Add_Date, Quantity, In_Cart, Purchased)

Le novità sono le colonne In_Cart e Purchased, che sono di tipo bit, quindi gli unici due valori consentiti sono 0 e 1. Le due colonne sono dei flag per la gestione dei prodotti:

- Se In_Cart è 0, allora il prodotto è stato aggiunto alla wishlist ma non nel carrello e il valore di Quantity è Null, nonché il valore di default. Inoltre, il valore di Purchased è 0 (valore predefinito).
- Se In_Cart è 1, allora il prodotto non sarà più presente nella wishlist ma nel carrello e il valore di Quantity viene impostato a 1 (default), con la possibilità di variazioni future, mentre Purchased potrà cambiare il valore da 0 a 1.

Analisi del Comportamento dei Flag

- L'utente vede un prodotto e decide di inserirlo nella wishlist, viene quindi inserita una nuova tupla nella tabella CustomerProduct contenente i dati relativi all'utente e al prodotto:
  - In_Cart = 0 → Quantity NULL
  - Purchased = 0

- L'utente decide di spostare il prodotto nel Carrello:
  - In_Cart = 1 → Quantity = 1
  - Purchased = 0

- L'utente decide di:
  - Acquistare i prodotti nel carrello:
    - Purchased = 1
  - Rimuovere i prodotti dal carrello:
    - Conferma cancellazione:
      Cancellazione tupla contenente i dati
    - Sposta nuovamente nella wishlist:
      In_Cart = 0 → Quantity NULL

Questi sono i principali aspetti dei due flag. A tal proposito è stato inserito un vincolo di tipo CHECK (CK_CustomerProduct_Purchased) per assicurarci del fatto che Purchased possa cambiare il proprio valore a 1 solamente se In_Cart = 1. Inoltre, è stato creato un trigger che gestisce l'aggiornamento dei valori di In_Cart e Quantity → WishlistQuantity.
Il flag Purchased è fondamentale per la gestione delle tuple presenti nella tabella CustomerProduct. Il suo cambio di valore coincide con l'attivazione del trigger MoveToSalesOrder. Si tratta di un automatismo che ha lo scopo di mantenere la gestione delle

tuple relative a prodotti acquistati, gestite interamente dal database, facilitando così un'eventuale implementazione di questa funzionalità dal backend. Per la creazione del trigger è stata necessaria la creazione di due nuove stored procedure:

- spAddSalesOrderDetail per l'inserimento di nuove tuple in SalesOrderDetail
- spAddSalesOrderHeader per l'inserimento di nuove tuple in SalesOrderHeader

---

Analisi delle Stored Procedure

spAddSalesOrderHeader:

Si tratta di una stored procedure che ha bisogno di un solo valore in input, ovvero CustomerID e per funzionare correttamente, CustomerID deve essere presente anche in CustomerProduct. Si evince dunque che questa stored procedure ha uno scopo ben preciso: gestire tutte le tuple con lo stesso CustomerID passato nella chiamata.

```
EXEC spAddSalesOrderHeader CustomerID
```

Restituisce la chiave primaria generata in SalesOrderHeader → SalesOrderID. La maggior parte dei dati vengono estratti automaticamente dalla relazione tra CustomerID con altre tabelle, oppure sono calcolati automaticamente o eventualmente viene assegnato il valore predefinito.

spAddSalesOrderDetail:

Si tratta di una stored procedure che ha una funzionalità praticamente identica a quella appena vista, con la differenza che in ingresso viene passata un'altra variabile oltre a CustomerID: la SalesOrderID generata quando viene effettuato un nuovo inserimento in SalesOrderHeader, nonché chiave primaria per SalesOrderID in SalesOrderDetail.

```
EXEC spAddSalesOrderDetail SalesOrderID, CustomerID
```

La tabella in questione ha lo scopo di registrare in modo più dettagliato i prodotti venduti in un unico ordine. Infatti, all'attivazione di questa stored procedure verranno aggiunte tante nuove tuple quante le tuple che hanno subito un aggiornamento sul valore Purchased da 0 a 1.

Trigger MoveToSalesOrder

Torniamo al trigger che si occupa di mettere insieme tutti questi pezzi. MoveToSalesOrder si attiva quando il valore di Purchased cambia da 0 a 1. Prima di procedere con qualsiasi manipolazione delle tuple interessate, si assicura che le tuple salvate nella tabella temporanea INSERTED abbiano effettivamente il valore di Purchased impostato a 1. Dopodiché vengono dichiarate delle variabili necessarie per il funzionamento corretto del trigger e delle stored procedure:

```
DECLARE @CustomerIDs TABLE (CustomerID INT)
DECLARE @Result TABLE (TempSalesId INT)
DECLARE @CustomerID INT, @SalesOrderID INT
```

Tra queste, farei particolare attenzione alla tabella temporanea @CustomerIDs che contiene tutti i CustomerID presenti in INSERTED. La probabilità che due utenti ordinino i propri prodotti nello stesso identico momento, con una precisione 'macchina', è molto bassa, tuttavia se questo scenario si dovesse presentare, il rischio è che i dati relativi agli ordini vengano gestiti in maniera errata. Per ovviare a questo scenario improbabile, ma pur sempre possibile, si è deciso di creare una tabella temporanea che contenga tuple con diversi CustomerID all'interno. Inoltre, per gestire correttamente le tuple con CustomerID diverse, si è optato per un ciclo WHILE come soluzione. All'interno del ciclo vengono chiamate le due stored procedure: prima spAddSalesOrderHeader che ha bisogno solamente del CustomerID e poi spAddSalesOrderDetail che utilizza il

CustomerID in questione e utilizza anche SalesOrderID generato dalla stored procedure precedente. Una volta che il ciclo termina, tutte le tuple con Purchased = 1 vengono eliminate.

Altre Modifiche

Altre modifiche fatte riguardano alcune colonne e dati delle tabelle SalesOrderHeader e SalesOrderDetail, dove sono state trovate alcune inconsistenze e incoerenze strutturali.

SalesOrderHeader

Le colonne SalesOrderNumber, PurchasedOrderNumber e AccountNumber si basano sulla stessa logica, tuttavia solo la prima delle tre è il risultato di un calcolo, le altre due sembrano siano state inserite manualmente, talvolta in modo incoerente: le lunghezze variano senza una vera logica. Quindi la modifica che si è deciso di applicare è di calcolare i dati inseriti anche per queste due colonne allo stesso modo di SalesOrderNumber.

SalesOrderDetail

In questa tabella si è notato che i dati relativi a UnitPrice sono tutti diversi rispetto ai dati contenuti in ListPrice nella tabella Product. Questa discrepanza fra dati non sembrava avesse alcun senso, di conseguenza è stata valutata come errata e allora, è stata apportata una modifica a tutti i dati relativi a UnitPrice per essere eguagliati ai rispettivi dati in ListPrice.

CustomerAddress

Un'altra modifica è stata fatta alla tabella CustomerAddress, dove è risultato che più CustomerID avessero associate più di un AddressID, il che è plausibile in uno scenario reale dove all'utente è data la possibilità di cambiare l'indirizzo di spedizione. Per risolvere questo problema è stata inserita una nuova colonna → IsPrimary con valore di tipo bit, che segue la seguente logica:

- IsPrimary = 1 (default), indirizzo attivo
- IsPrimary = 0 implica che ci sia un altro indirizzo per lo stesso utente e che questo sia inattivo

Avere una colonna che segue questo ragionamento è molto utile per gestire uno scenario in cui un utente registrato volesse decidere di inserire un nuovo indirizzo e mantenere ancora presente ma non attivo l'indirizzo precedente.

Stored Procedure per la Gestione dei Clienti

Inoltre, è stata aggiunta una stored procedure, ovvero spAddCustomer, per l'inserimento di nuovi clienti. Per rendere possibile la registrazione a BikeVille, sono richiesti nome, cognome, email e password, che corrispondono a 'FirstName', 'LastName' e 'EmailAddress' per la tabella Customer nel database principale, mentre la password è necessaria per generare le credenziali da inserire nella CollectionName Customer su MongoDB. La stored procedure è stata pensata per essere utilizzata dal backend. Accetta tre variabili passate tramite frontend e crea il nuovo cliente da inserire nel database, e finisce restituendo in output la nuova CustomerID creata. Questa viene utilizzata dal backend per l'inserimento dei dati sensibili del nuovo cliente su MongoDB. Un dettaglio particolare della stored procedure è:

```
DECLARE @SalesPersonList TABLE (SalesPerson NVARCHAR(256));
INSERT INTO @SalesPersonList (SalesPerson)
VALUES
('adventure-works\linda3'),
('adventure-works\jillian0'),
('adventure-works\shu0'),
('adventure-works\david8'),
('adventure-works\josé1');
```

```
SELECT TOP 1 @SalesPerson = SalesPerson
FROM @SalesPersonList
ORDER BY NEWID();
```

Si è deciso di assegnare in modo random un valore alla colonna relativa a SalesPerson, dato che si può osservare che i valori comunemente utilizzati sono i 5 elencati sopra e vengono assegnati senza una evidente logica all'interno del database di riferimento. Per permettere l'implementazione di questo meccanismo, è necessaria una tabella che contiene i 5 venditori comunemente usati e poi grazie a ORDER BY NEWID() è possibile l'assegnazione in modo casuale di uno fra i valori presenti nella tabella SalesPersonList.

Centralizzazione della Logica nel Database

Per mantenere una separazione tra linguaggi (SQL e C#), si è deciso di centralizzare la logica per la gestione di alcuni metodi direttamente nel database, tramite stored procedure. Il backend a questo punto deve occuparsi solo del passaggio dei parametri e richiamare la stored procedure già progettata. La lista completa di stored procedure, escluse quelle già menzionate, è la seguente:

- **spAddCustomerAddress**: Aggiunge un nuovo indirizzo a un cliente esistente nel database e lo associa al cliente specificato. La procedura gestisce sia l'inserimento dell'indirizzo nella tabella Address che l'associazione del cliente all'indirizzo nella tabella CustomerAddress.

- **spGetCustomerAddress**: Recupera tutti gli indirizzi associati a un cliente specifico. Restituisce i dettagli completi degli indirizzi inclusi il tipo di indirizzo e se è l'indirizzo principale del cliente.

- **spUpdateCustomerAddress**: Aggiorna i dettagli di un indirizzo associato a un cliente specifico. Verifica che l'indirizzo appartenga al cliente prima di eseguire l'aggiornamento, garantendo che solo gli indirizzi corretti vengano modificati.

- **spDeleteCustomerAddress**: Rimuove l'associazione tra un cliente e un indirizzo specifico. Se l'indirizzo non è più associato ad alcun cliente dopo la rimozione, viene eliminato anche dalla tabella degli indirizzi. Questa procedura è utile per gestire la pulizia degli indirizzi non più in utilizzo.

- **spAddCustomerProducts**: Associa un prodotto a un cliente nella tabella CustomerProduct. Se l'associazione esiste già, la procedura inserisce un nuovo record con una quantità predefinita e la data di aggiunta. Utile per la gestione del carrello o della wishlist.

- **spGetCustomerProducts**: Recupera l'elenco dei prodotti associati a un cliente, filtrati in base allo stato del carrello. La procedura restituisce gli ID dei prodotti che sono nel carrello del cliente (In_Cart = 1) o che sono nella wishlist (In_Cart = 0).

- **spRemoveCustomerProducts**: Rimuove l'associazione tra un cliente e un prodotto specifico della tabella CustomerProduct.

- **spConfirmPurchase**: Conferma un acquisto per un cliente specifico. La procedura verifica che il cliente abbia almeno un indirizzo di spedizione associato e, in caso affermativo, aggiorna lo stato dei prodotti nel carrello del cliente a 'Acquistato' (Purchased = 1). Utile per gestire il processo di checkout in un sistema di e-commerce.

- **spUpdateCartQuantity**: Aggiorna la quantità di un prodotto nel carrello. Il campo Quantity nella tabella CustomerProduct può essere modificato solo se il prodotto è attualmente nel carrello quindi In_Cart = 1.

- **spUpdateCartStatus**: Aggiorna lo stato di un prodotto nel carrello della spesa di un cliente. Modifica In_Cart nella tabella CustomerProduct, indicando se il prodotto è nel carrello (1) o meno (0).

- **spGetProductById**: Recupera tutti i dettagli di un prodotto specifico della tabella Product, usando l'ID del prodotto come filtro.

Le stored procedure spGetFilteredProducts e spSearchProducts sono importanti per la gestione di un sistema di filtri per la ricerca di prodotti dalla UI.

SpSearchProducts

La ricerca può avvenire in modo specifico, quindi il cliente sa esattamente cosa sta cercando, dunque inserisce il nome del prodotto nella casella di ricerca. Per questo è necessaria la gestione di una ricerca specifica tramite spSearchProduct. La stored procedure è strutturata in tre fasi:

1. Creazione di una tabella temporanea:

```
CREATE TABLE #SearchResults
(
ProductID int,
Name nvarchar(50),
ProductNumber nvarchar(25),
...
)
```

2. Inserimento dei risultati della ricerca:

```
INSERT INTO #SearchResults
...
WHERE p.Name LIKE '%' + @SearchTerm + '%'
```

3. Restituzione dei risultati impaginati

SpGetFilteredProducts

Questa stored procedure viene utilizzata da un cliente che sa cosa sta cercando ma vuole esplorare l'ambiente con l'obiettivo di raffinare la propria ricerca (la trappola per aragoste). Dunque, in questo scenario l'utente ha una serie di prodotti con caratteristiche miste, e qualora volesse applicare delle specificità alla sua ricerca, tramite categoria, sottocategoria, colore, dimensione e fascia di prezzo, la stored procedure in questione ne è la diretta responsabile per fare in modo che questo tipo di ricerca sia possibile. Il metodo di ricerca è molto simile a spSearchProducts, con la differenza che invece del nome, ricerca altri campi interessati delle rispettive tabelle.