EXECUTAREA UNEI CERERI (QUERY) în MS SQL SERVER

Ion MORARU

Departamentul Ingineria Software și Automatică, grupa TI-181 F/R, Facultatea Calculatoare, Informatică și Microelectronică, Universitatea Tehnică a Moldovei, Chișinău, Republica Moldova

Autorul corespondent: Ion MORARU, e-mail: ion.moraru@isa.utm.md

Coordonator științific: Dorian SARANCIUC, DISA, FCIM, UTM

Rezumat. *În articolul dat este reprezentat cum funcționează SQL Server și cum să-i valorificam puterea. O cerere de date poate fi trimisa spre execuție in mai multe moduri, dar SQL Server are un regim definit cum sa le execute si sa le returneze clientului. Fluxul de cererii si modalitatea de execuție este descrisa in acest document din perspectiva clientului si modalitate de lucru a serverului. Sistemul de Gestiune a bazelor de date SQBD este un sistem complex de lucru cu datele in procesul de prelucrare si păstrare a datelor.*

Cuvinte cheie: *SGBD, TDS, CLR, RPC.*

1. Query

SQL Server este o platformă client-server. Singura modalitate de a interacționa cu baza de date back-end este prin trimiterea de query, care conțin comenzi pentru baza de date. Protocolul folosit pentru a comunica între aplicația dumneavoastră și baza de date se numește TDS (Tabular Data Sream). Aplicația poate folosi una dintre cele mai multe implementări la nivelul clientului ale protocolului: *SqlClient*, OleDB, ODBC, JDBC, PHP Driver pentru SQL Server gestionat CLR(Common Language Runtime) sau implementarea FreeTDS cu sursă deschisă. Esența este că atunci când aplicația funcționează pe o baza de date pentru funcționare, va trimite o solicitare prin protocolul TDS. Cererea în sine poate lua mai multe forme: [1]

a) Solicitare date pe loturi (Batch Request)

Acest tip de cerere conține doar text T-SQL pentru ca un lot să fie executat. Acesta este tipul de solicitare pe care *SqlClient* o trimite oricare dintre *SqlCommand*: *ExecuteReader(), ExecuteNonQuery(), ExecuteScalar(), ExecuteXmlReader()* (sau echivalentele asincrone respective) pe un obiect *SqlCommand* cu o listă de parametri goală. Monitorizarea cu SQL Profiler, o clasă de evenimente *SQL:BatchStarting*.

b) Solicitare date prin procedură (RPC Request)

Acest tip de cerere conține un identificator de procedură, care trebuie de executat împreună cu de parametri care se despart prin virgula. Un interes deosebit este atunci când id-ul procedurii va fi 12, adică. *sp_execute*. În acest caz, primul parametru este textul T-SQL de executat. Cererea o va trimite spre executare *SqlCommand* cu cel putin un parametru. Pentru monitorizare folosim SQL Profiler RPC:StartingEvent.

c) Solicitare prin bulk (Bulk Load Request)

Bulk Load este o solicitare de tip special utilizată de operațiunile de inserare în bloc, cum ar fi utilitarul *bcp.exe*, interfața *IRowsetFastLoad* din OleDB sau de clasa *SqlBulkcopy*. Încărcarea în bloc este diferită de celelalte solicitări, deoarece este singura solicitare care începe execuția înainte ca cererea să fie finalizată pe protocolul TDS. Acest lucru îi permite să înceapă execuția și apoi să înceapă să consume fluxul de date de inserat.

După ce o solicitare TDS completă ajunge la motorul bazei de date, SQL Server va crea o sarcină pentru a gestiona cererea. Lista cererilor de pe server poate fi interogată din *sys.dm_exec_requests* [2].

2. Sarcini (Task)

Sarcinile menționate mai sus. sunt create pentru a gestiona cererea, va reprezenta cererea de la început până la finalizare. De exemplu, dacă cererea este o solicitare de tip lot (SQL Batch), sarcina va reprezenta întregul lot, nu instrucțiuni individuale. Declarațiile individuale din lotul SQL nu vor crea sarcini noi. Anumite instrucțiuni individuale din cadrul lotului se pot executa cu paralelism (denumite adesea DOP, Degree of Parallelism), în cazul lor, sarcina va genera noi sub-sarcini pentru executarea în paralel. Dacă cererea returnează un rezultat, lotul este complet când rezultatul este consumat complet de client (de exemplu, când eliminați SqlDataReader). Puteți vedea lista sarcinilor de pe server interogând *sys.dm_os_tasks*.

Când o nouă solicitare ajunge pe server și sarcina este creată pentru a gestiona acea cerere, în starea *PENDING*. În această etapă, serverul nu are încă idee care este de fapt cererea. Sarcina trebuie să înceapă să se execute mai întâi, iar pentru aceasta motorul trebuie să îi atribuie un worker.

3. Muncitorii (Workers)

Workers sunt grupul de fire (*threads*) al SQL Server. Un număr de workers este creat inițial la pornirea serverului și mai mulți pot fi creați la cerere până la fire (*threads*) maxime de lucru configurate. Numai *workers*-rii execută cod. *Workers* așteaptă ca sarcinile cu statut *PENDING* să devină disponibile (din cererile care vin pe server) și apoi fiecare *worker* preia exact o sarcină și o execută. *Worker*-ul este ocupat (ocupat) până când sarcina se termină complet. Sarcinile care sunt în așteptare atunci când nu mai sunt *workers* disponibili vor trebui să aștepte până când una dintre sarcinile de execuție (în rulare) se finalizează și *worker*-ul care a executat acea sarcină devine disponibil pentru a executa o altă sarcină în așteptare. Pentru o solicitare de lot (batch) SQL, *worker*-ul care preia sarcina respectivă va executa întregul lot SQL (fiecare instrucțiune). Acest lucru ar trebui să rezolve întrebarea adesea pusă dacă instrucțiunile dintr-un lot SQL (=> cerere => sarcină => worker) se pot executa în paralel: nu, deoarece sunt executate pe un singur fir (=> *worker*), atunci fiecare instrucțiune din lot nu va incepe sa se execute pina nu se finalizeaza precedenta. Pentru instrucțiunile care folosesc intern paralelism (DOP > 1) și creează sub-sarcini, fiecare sub-sarcină trece exact prin același ciclu: este creată ca *PENDING* și un *worker* trebuie să o ridice și să o execute (un worker diferit de de cel ce executa sarcina parinte!). Listele și starea *workers* din SQL Server pot fi văzute interogând *sys.dm_os_workers*.

4. Analiză și compilare

Odată ce o sarcină a început să execute o solicitare, primul lucru pe care trebuie să-l facă este să înțeleagă conținutul cererii. În această etapă, SQL Server se va comporta ca un limbaj VM (virtual machina) pentru interpretarea limbajului: text T-SQL va fi analizată și va fi creat un arbore de sintaxă abstractă pentru a reprezenta cererea. Întreaga cerere (lotul) este analizată și compilată. Dacă apare o eroare în această etapă, cererile se încheie cu o eroare de compilare (cererea este apoi finalizată, sarcina este finalizată și *worker*-ul este liber să preia o altă sarcină în așteptare). SQL și T-SQL este un limbaj declarativ de vârf cu instrucțiuni extrem de complexe. Compilarea loturilor T-SQL nu are ca rezultat un cod executabil similar cu instrucțiunile native ale procesorului și nici măcar codul octet JVM, dar are ca rezultat planuri de acces la date (sau planuri de interogare). Aceste planuri descriu modalitatea de deschidere a tabelelor și indexurilor, de a căuta și de a localiza rândurile de interes și de a efectua orice manipulare a datelor, așa cum este solicitat în lotul SQL. De exemplu, un plan de interogare va descrie o cale de acces precum „deschide indexul idx1 pe tabelul t, localizează rândul cu cheia „k” și returnează coloanele a și b”. Ca o notă secundară: o greșeală comună făcută de dezvoltatori este încercarea de a veni cu o singură interogare T-SQL care să acopere multe alternative, de obicei prin utilizarea expresiilor inteligente în clauza WHERE, având adesea multe alternative SAU (de ex. (COLUMN = @ parametrul SAU @parametrul ESTE NULL). Pentru dezvoltatorii care încearcă să mențină lucrurile DRY și evitarea repetării sunt bune practici, pentru interogările SQL sunt pur și simplu proaste. Compilarea trebuie să vină cu o cale de acces care să funcționeze pentru orice valoare a parametrilor de intrare și rezultatul este cel mai adesea sub-optim [3].

5. Optimizare

Vorbind despre alegerea unei căi optime de acces la date, aceasta este următoarea etapă din durata de viață a cererii: optimizarea. În SQL și în T-SQL, optimizarea înseamnă a alege cea mai bună cale de acces la date dintre toate alternativele posibile. Luați în considerare că dacă avem o interogare simplă cu îmbinare între două tabele și fiecare tabel are un index suplimentar, există deja 4 moduri posibile de a accesa datele și numărul de posibilități crește exponențial pe măsură ce complexitatea interogării crește și sunt disponibile mai multe căi de acces alternative (practic, mai mulți indici). Adăugați la aceasta că JOIN-ul se poate face folosind diverse strategii (neted loop, hash, merge) și vom vedea de ce optimizarea este un concept atât de important în SQL. SQL Server folosește un optimizator bazat pe costuri, ceea ce înseamnă că va lua în considerare toate (sau cel puțin multe) alternative posibile, va încerca să facă o presupunere despre costul fiecărei alternative și apoi o va alege pe cea cu cel mai mic cost. Costul este calculat în primul rând luând în considerare dimensiunea datelor care ar trebui să fie citite de fiecare alternativă. Pentru a veni cu aceste costuri, SQL Server trebuie să cunoască dimensiunea fiecărui tabel și distribuția valorilor coloanelor, care sunt disponibile din statisticile asociate datelor. Alți factori luați în considerare sunt consumul CPU și memoria necesară pentru fiecare alternativă de plan. Folosind formule reglate de-a lungul a mulți ani de experiență, toți acești factori sunt sintetizați într-o valoare unică a costului pentru fiecare alternativă și apoi alternativa cu cel mai mic cost este aleasă ca plan de interogare care trebuie utilizat.

Explorarea tuturor acestor alternative poate fi consumatoare de timp și de aceea, odată ce un plan de interogare este creat, este de asemenea stocat în cache pentru reutilizare ulterioară. Solicitările similare viitoare pot sări peste faza de optimizare dacă pot găsi un plan de interogare deja compilat și optimizat în memoria cache internă a SQL Server.

6. Execuţie

Odată ce un plan de interogare este ales, cererea poate începe să se execute. Planul de interogare este tradus într-un arbore de execuție real. Fiecare nod din acest arbore este un operator. Toți operatorii implementează o interfață abstractă cu 3 metode:

- *open()*,
- *next()*,
- *close()*.

Bucla de execuție constă în apelarea *open()* pe operatorul care se află la rădăcina arborelui, apoi apelarea *next()* în mod repetat până când returnează false și în final apelarea *close()*. Operatorul de la rădăcina arborelui va apela la rândul său aceeași operație pe fiecare dintre operatorii săi copii, iar aceștia la rândul lor apelează aceleași operații pe operatorii lor copii și așa mai departe. La frunză arborilor există de obicei operatori de acces fizic care preiau efectiv date din tabele și indici. La niveluri intermediare există operatori care implementează diverse operațiuni de date precum filtrarea datelor, efectuarea de JOIN-uri sau sortarea rândurilor. Interogările care utilizează paralelismul folosesc un operator special numit operator de schimb *Exchange*. Operatorul *Exchange* lansează mai multe fire de execuție (sarcini => workers) în execuție și cere fiecărui fir de execuție să execute un subarbore al planului de interogare. Apoi agreghează producția de la acești operatori, folosind un model tipic de mai mulți producători - un singur consumator [4].

Acest model de execuție se aplică nu numai la interogări, dar și la modificarea datelor (inserare, ștergere, actualizare). Există operatori care se ocupă de inserarea unui rând, operatori care se ocupă de ștergerea unui rând și operatori care se ocupă de actualizarea unui rând. Unele solicitări creează planuri banale (de exemplu, *INSERT INTO... VALUES ...*), în timp ce altele creează planuri extrem de complexe, dar execuția este identică pentru toate și are loc exact așa cum am descris: arborele de execuție este repetat apelând *next( )* până când este gata.

Unii operatori sunt foarte simpli, luați în considerare, de exemplu, operatorul *TOP*(N): atunci când *next()* este apelat, tot ce trebuie să apeleze *next()* pe arbore și să țină un numărin contorul intern. După apelarea de N ori, pur și simplu returnează false fără a mai apela pe arbore, terminând astfel iterația acelui sub-arbore.

Alți operatori au un comportament mai complex, luați în considerare ce trebuie să facă un operator de Nested Loops: trebuie să țină evidența poziției iterației buclei atât pe copilul exterior, cât și pe copilul interior, să apeleze *next()* pe copilul exterior, să deruleze copilul interior și apelați *next()* pe copilul interior până când predicatul join este satisfăcut.

Anumiți operatori au un comportament *stop-and-go*, ceea ce înseamnă că nu pot produce nicio ieșire până când nu consumă toate intrările de la operatorii proprii. Exemple de astfel de operatori este SORT: primul apel la *next()* nu revine până când toate rândurile create de operatorii copii sunt preluate și sortate.

Un operator precum *HASH JOIN* va fi atât un comportament complex, cât și de tip stop-and-go: pentru a construi tabelul *hash*, trebuie să apeleze *next()* pe elementul secundar de construcție până când acel operator returnează false. Apoi apelează *next()* pe operatorul arbore din partea sondei până când se găsește o potrivire în tabelul *hash,* apoi revine. Apelurile ulterioare continuă să apeleze *next()* pe operatorul copil din partea probei și să revină la potrivirea tabelului *hash*, până când operatorul secundar *next()* returnează ***false***.

7. Rezultate

Rezultatele sunt returnate în programul client pe măsură ce execuția continuă. Pe măsură ce rândurile „se returneaza" în arborele de execuție, operatorul de top are de obicei sarcina de a scrie aceste rânduri în containerele de rețea și le trimite înapoi către client. Rezultatul nu este creat mai întâi într-un spațiu de stocare intermediar (memorie sau disc) și apoi trimis înapoi către client, ci este trimis înapoi așa cum este creat (pe măsură ce interogarea se execută). Trimiterea rezultatului înapoi către client este, desigur, supusă protocolului de control al fluxului de rețea. Dacă clientul nu consumă în mod activ rezultatul (de exemplu, apelând *SqlDataReader.Read()*), atunci în cele din urmă controlul fluxului va trebui să blocheze partea de trimitere (interogarea care este în curs de executare) și aceasta, la rândul său, va suspenda execuția interogare. Interogarea se reia și produce mai multe rezultate (continua să repete planul de execuție) de îndată ce controlul fluxului de rețea eliberează resursele de rețea necesare.

Un caz interesant este parametrii OUTPUT asociați cu cererea. Pentru a returna valoarea de ieșire înapoi către client, valoarea trebuie să fie inserată în fluxul de date din rețea care curge de la executarea interogării înapoi către client. Valoarea poate fi scrisă înapoi clientului doar la sfârșitul execuției, pe măsură ce cererea se termină. Acesta este motivul pentru care valorile parametrilor de ieșire pot fi verificate numai după ce au fost consumate toate rezultatele.

Concluzii

SQL, T-SQL este un limbaj script nativ pentru comunicarea cu bazele de date. Înțelegerea corecta a formarii si executării unui script pe baza de date ne formează o practica optima de formare a interogărilor, in plus putem înțelege fiecare etapa in executarea si interveni in cazul unor abateri.

Bibliografie

1. SQL Server Protocols [online]. [accesat 20.12.2022]– Disponibil: https://docs.microsoft.com/en-us/openspecs/sql_server_protocols/ms-sqlprotlp/f16558b2-4561-45be-89c9-6f9114514c97
2. Abstract syntax tree [online]. [accesat 20.12.2022]– Disponibil: https://en.wikipedia.org/wiki/Abstract_syntax_tree
3. Transact-SQL [online]. [accesat 20.12.2022]– Disponibil: https://docs.microsoft.com/en-us/sql/t-sql/language-reference?view=sql-server-ver15
4. Execution Plan Caching and Reuse [online]. [accesat 20.12.2022]– Disponibil: https://docs.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/ms181055(v=sql.105)?redirectedfrom=MSDN