Optimalizácia dotazov

Ján Mazák

FMFI UK Bratislava

Životný cyklus dotazu

1. Query parsing and translation
   - syntax check — Je to korektné SQL?
   - user privilege check — Má používateľ oprávnenie pristupovať k db objektom?
   - semantic check — Existujú relácie a atribúty? Sedia dátové typy (napr. pri EXCEPT)?
   - cache check — Ak dotaz neprišiel prvýkrát, možno máme v cache už optimalizovaný preklad do relačnej algebry. Ak nie, preložiť.

Životný cyklus dotazu

2. Query optimization
   - Zozbieranie štatistických dát o reláciách (napr. odhad počtu záznamov).
   - Preusporiadanie operátorov relačnej algebry, napr. identifikácia viacnásobného výpočtu toho istého poddotazu či použitie rôznych heuristík.
   - Preklad operátorov relačnej algebry do fyzických operátorov.
   - Možnosť zvážiť niekoľko rôznych zápisov výpočtu, odhad času, výber najlepšej možnosti (nie nutne najrýchlejšej, možno tiež optimalizovať pamäť).
   - Vytvorenie plánu pre výpočet dotazu (query plan, používateľ má možnosť zobraziť pomocou EXPLAIN).

Životný cyklus dotazu

3. Query execution
   - FROM, JOIN
   - WHERE
   - GROUP BY
   - HAVING
   - SELECT (projection, window functions, CASE, ...)
   - DISTINCT
   - ORDER BY
   - LIMIT / OFFSET

Čas výpočtu dotazu možno získať cez EXPLAIN ANALYZE.

Životný cyklus dotazu

4. Transmitting results
   - výsledok výpočtu možno uložiť na disk, alebo priebežne posielať používateľovi
   - typicky zahŕňa manažment sieťového pripojenia
   - rozsiahlejšie výsledky nemožno poslať naraz
   - z pohľadu programovacieho jazyka práca s objektmi ako *cursor*

Heuristiky pre join

Pre optimalizáciu výpočtu joinov existuje rozsiahla literatúra — ide o kľúčovú operáciu. Pár ukážok heuristík (plán výpočtu musíme zvoliť „okamžite“, nemožno vyriešiť NP-ťažké problémy):

- odsunúť projekcie a selekcie čo najnižšie (zmenšuje medzivýsledky; vďaka projekcii sa do diskového bloku vojde viac záznamov)
- uvažovať len o left-deep plans:

$$(((A_1 \bowtie A_2) \bowtie A_3) \bowtie A_4) \ldots$$

(znižuje počet možných plánov na $n!$; pri takýchto plánoch možno vždy uplatniť pipelining — nemusíme ukladať medzivýsledky na disk)

Heuristiky pre join

- počítať karteziánsky súčin, len keď sa tomu nedá vyhnúť (join bez podmienok býva veľký)
- „uši v hypergrafe“ rátať až na konci (keď nejaká relácia má atribúty z joinovacích podmienok spoločné len s jednou inou, odsunúť ju vo výpočte joinu na neskôr)

Ak potrebujete rátať join „manuálne“ (pre rozsiahle dáta mimo relačného DBMS), azda sa oplatí naštudovať si niečo o optimalizácii joinov.

Fyzické operátory

Zápis dotazu v relačnej algebre určuje postup výpočtu, ale len rámcovo. Treba zohľadniť aj fyzické uloženie dát. Každý operátor relačnej algebry má minimálne jeden zodpovedajúci fyzický operátor, často však niekoľko — vyjadrujú rôzne spôsoby výpočtu.

Idey týkajúce sa fyzických operátorov sú užitočné všeobecne aj mimo DBMS — väčšina problémov súvisiacich s veľkým množstvom dát sa dá riešiť skombinovaním hashovania, usporadúvania, filtrovania a vhodného výpočtu joinov.

Fyzické operátory

Algoritmy fyzických operátorov sa snažia minimalizovať počet diskových operácií, nie kroky výpočtu v operačnej pamäti.

Mnohé operátory sú implementované ako iterátory: priebežne spracúvajú záznamy zo vstupu a vytvárajú výstup (*pipelining*). Výhoda: netreba ukladať medzivýsledky na disk.

Fyzické operátory — table scan

- sequential (linear) scan — prejdeme priamo záznamy zaradom, ako sú uložené (efektívne pre malé relácie)
- index scan — využije sa existujúci index, neraz netreba vidieť všetky záznamy, lebo máme podmienku za WHERE
- index-only scan — k záznamom netreba vôbec pristupovať, ak nás zaujímajú len atribúty obsiahnuté v indexe
- bitmap index scan — skenovaním indexu vytvoríme bitmapu udávajúcu, ktoré bloky z heap file treba prečítať, a potom ich načítame celé naraz, nie jednotlivo pre jednotlivé záznamy

Fyzické operátory — join / antijoin

- nested loop join — dva vnorené cykly párujú každý záznam s každým
- merge join — ak joinovacia podmienka s rovnosťou alebo nerovnosťou obsahuje atribúty, podľa ktorých obe joinované relácie vieme usporiadať, join sa ráta ľahko podobným spôsobom, ako zlučovacia fáza merge-sortu (výhodné kombinovať s index scanom, ktorý produkuje záznamy už správne usporiadané)
- hash join — pri podmienke s rovnosťou: nutnou podmienkou na join dvoch záznamov z jednotlivých relácií je, že patria do rovnakých bucketov (nie je to však postačujúce), vieme teda join rátať bucket po buckete; hashovacie buckety môžeme vytvoriť nanovo špeciálne pre účely joinu (pomôže, ak existuje hash index)

Query plans — EXPLAIN

- Plánovanie výpočtu dotazu budeme sledovať pre databázu PostgreSQL.
- Príkaz EXPLAIN SELECT ... miesto výpočtu dotazu vráti plán jeho výpočtu zapísaný pomocou fyzických operátorov, aj s odhadmi času potrebného na jednotlivé kroky.
- Pri odhade ceny sa využívajú konfigurovateľné konštanty (napr. vyhodnotenie bežnej podmienky za WHERE je 400x rýchlejšie ako načítanie bloku z disku).
- Reálny čas výpočtu vieme získať pomocou EXPLAIN ANALYZE SELECT ... .
- EXPLAIN funguje aj pre SQLite, ale zápis plánu je oveľa menej zrozumiteľný.

Query plans — EXPLAIN

Počítame dotaz

```
EXPLAIN
SELECT d.deptno, COUNT(e.empno)
FROM department d
JOIN employee e ON e.deptno = d.deptno
WHERE NOT EXISTS (
    SELECT 1
    FROM project p
    WHERE p.empno = e.empno
)
GROUP BY d.deptno;
```

Pre uvedené relácie nie sú vytvorené žiadne indexy.

Query plans — EXPLAIN

Fyzický plán (query plan):

```
HashAggregate  (cost=58.16..59.51 rows=135 width=12)
  Group Key: d.deptno
  ->  Hash Join  (cost=38.67..57.48 rows=135 width=8)
        Hash Cond: (e.deptno = d.deptno)
        ->  Hash Anti Join  (cost=21.93..40.38 rows=135 width=8)
              Hash Cond: (e.empno = p.empno)
              ->  Seq Scan on employee e  (cost=0.00..12.70 rows=270 width=8)
              ->  Hash  (cost=15.30..15.30 rows=530 width=4)
                    ->  Seq Scan on project p  (cost=0.00..15.30 rows=530 width
        ->  Hash  (cost=13.00..13.00 rows=300 width=4)
              ->  Seq Scan on department d  (cost=0.00..13.00 rows=300 width=4)
```

Query plans — EXPLAIN

EXPLAIN uvádza odhadovanú veľkosť relácií. Je značne nepresná (miesto stoviek je tam záznamov do 20). Presná evidencia by viedla k spomaleniu INSERT/UPDATE/DELETE, preto ide skôr o odhad založený na počte diskových blokov, ktoré zaberá relácia.

Ľavé číslo pri odhade ceny (cost) znamená čas do vypočítania prvého riadka výstupu (napr. pre operátor Sort sa celá práca musí vykonať pred vypísaním prvého riadka). Pravé číslo je čas ukončenia činnosti operátora. Udávané čísla sú kumulatívne (napr. naspodku Hash začína od 15.30, čo je čas výpočtu podriadeného Seq Scan).

Query plans — EXPLAIN

Pokus: vytvoríme index, ktorý by mohol urýchliť prehľadávanie relácie employee; očakávame, že miesto Seq Scan uvidíme Index Scan.

```
CREATE INDEX ON employee USING HASH (empno)
```

Stane sa však niečo úplne iné: zmení sa spôsob výpočtu GROUP BY. (Dôvod je nejasný; súvisí to však s tým, že v tabuľkách máme veľmi málo záznamov, preto sa neoplatí čítať index. Voľba plánu môže súvisieť so stavom systému: pri niektorých operáciách, napr. usporiadavaní, vieme výpočet urýchliť za cenu zvýšenia pamäťových nárokov.)

Query plans — EXPLAIN

Po vytvorení indexu:

```
GroupAggregate  (cost=37.70..37.82 rows=7 width=12)
Group Key: d.deptno
->  Sort  (cost=37.70..37.72 rows=7 width=8)
      Sort Key: d.deptno
      ->  Hash Join  (cost=23.41..37.60 rows=7 width=8)
            Hash Cond: (d.deptno = e.deptno)
            ->  Seq Scan on department d  (cost=0.00..13.00 rows=300 width=4)
            ->  Hash  (cost=23.32..23.32 rows=7 width=8)
                  ->  Hash Anti Join  (cost=21.93..23.32 rows=7 width=8)
                        Hash Cond: (e.empno = p.empno)
                        ->  Seq Scan on employee e  (cost=0.00..1.14 rows=14
                        ->  Hash  (cost=15.30..15.30 rows=530 width=4)
                              ->  Seq Scan on project p  (cost=0.00..15.30 ro
```

Query plans — EXPLAIN

Po vložení 100 000 záznamov do každej tabuľky:

```
HashAggregate  (cost=12844.49..13024.58 rows=18009 width=12)
  Group Key: d.deptno
  ->  Hash Join  (cost=11783.76..12754.45 rows=18009 width=8)
        Hash Cond: (e.deptno = d.deptno)
        ->  Merge Anti Join  (cost=9715.13..10638.55 rows=18009 width=8)
              Merge Cond: (e.empno = p.empno)
              ->  Sort  (cost=4420.10..4510.14 rows=36018 width=8)
                    Sort Key: e.empno
                    ->  Seq Scan on employee e  (cost=0.00..1694.18 rows=36018
              ->  Sort  (cost=5295.03..5418.92 rows=49555 width=4)
                    Sort Key: p.empno
                    ->  Seq Scan on project p  (cost=0.00..1430.55 rows=49555 w
        ->  Hash  (cost=1605.50..1605.50 rows=37050 width=4)
              ->  Seq Scan on department d  (cost=0.00..1605.50 rows=37050 widt
```

Query plans — EXPLAIN

Po vložení 1 000 000 záznamov do každej tabuľky:

```
Finalize GroupAggregate  (cost=135831.70..157009.04 rows=180009 width=12)
  Group Key: d.deptno
  ->  Gather Merge  (cost=135831.70..154458.91 rows=150008 width=12)
        Workers Planned: 2
        ->  Partial GroupAggregate  (cost=134831.68..136144.25 rows=75004 width=12)
              Group Key: d.deptno
              ->  Sort  (cost=134831.68..135019.19 rows=75004 width=8)
                    Sort Key: d.deptno
                    ->  Parallel Hash Join  (cost=121056.81..128758.35 rows=75004 width=8)
                          Hash Cond: (e.deptno = d.deptno)
                          ->  Merge Anti Join  (cost=97707.81..102998.46 rows=75004 width=8)
                                Merge Cond: (e.empno = p.empno)
                                ->  Sort  (cost=29781.77..30156.79 rows=150008 width=8)
                                      Sort Key: e.empno
                                      ->  Parallel Seq Scan on employee e  (cost=0.00..14834.08 rows=1500
                                ->  Sort  (cost=67926.03..69164.38 rows=495338 width=4)
                                      Sort Key: p.empno
                                      ->  Seq Scan on project p  (cost=0.00..14299.38 rows=495338 width=4
                          ->  Parallel Hash  (cost=16512.67..16512.67 rows=416667 width=4)
                                ->  Parallel Seq Scan on department d  (cost=0.00..16512.67 rows=416667 w
```

Query plans — EXPLAIN

- Plánovanie trvalo prvýkrát 22 ms, pri opakovanom výpočte toho istého dotazu už len 0.5 ms.
- Algoritmus pre plánovanie by mal byť deterministický (by default), ale navonok sa tak nejaví; pri opakovanom použití EXPLAIN sa niekedy plány líšia. (Pri veľkom počte joinovaných tabuliek PostgreSQL môže použiť genetický algoritmus miesto deterministického.)
- Väčšina práce pri výpočte dotazu spočíva v usporadúvaní, spájaní a hashovaní (sort / merge / hash), znova a znova.

Fyzické operátory — triedenie

Množstvo triedených dát rádovo prevyšuje dostupnú operačnú pamäť. Riešenie: externý merge sort.
Majme B blokov v RAM a N blokov dát na disku. Dve fázy:

1. Prečítame B blokov dát, usporiadame v RAM, zapíšeme na disk. Opakujeme, kým máme dáta. Vznikne $\lceil N/B \rceil$ usporiadaných „behov“ (runs).
2. Pre výstup rezervujeme 1 blok RAM (až sa naplní, zapíšeme ho na disk). Zlúčime $B-1$ behov tak, že pre každý rezervujeme 1 blok (keď sa načítané dáta minú, prečítame ďalší blok tohto behu z disku). Pri určovaní záznamu, ktorý patrí na výstup, sa stačí pozrieť na prvý záznam každého behu. Opakujeme, kým neostane jediný beh.

Fyzické operátory — triedenie

Zložitosť externého merge sortu: vo všeobecnosti

$$2N \cdot (1 + \lceil \log_{B-1}(\lceil N/B \rceil) \rceil),$$

závisí však od dostupnej pamäti.

Úloha. Koľko pamäti B v závislosti od N treba, aby 2. fázu stačilo vykonať raz a nebolo potrebné ju opakovať? Koľko I/O operácií na blokoch sa vtedy vykoná?

Fyzické operátory — triedenie

Úloha. Relácia s 10^7 záznamami je uložená na disku v blokoch veľkosti 8 kB. V každom bloku je 100 záznamov relácie. K dispozícii nie je žiaden index. Prenos diskového bloku (do RAM aj z RAM) trvá 1 ms. Bloky v RAM a na disku sú rovnako veľké. Na usporiadanie záznamov použijeme merge sort.
(a) Ako dlho bude trvať usporiadanie záznamov tejto relácie, ak máme alokovaných v RAM 200 blokov?
(b) Ako dlho bude trvať usporiadanie záznamov tejto relácie, ak máme alokovaných v RAM 2000 blokov (čiže 10x viac)?

Fyzické operátory — nested loop join

V pamäti máme alokovaných M blokov. Počítame join relácií R a S spôsobom nested loop join.
Predpokladáme, že R zaberá menej blokov ako S.

Rezervujeme 1 blok pre výstup. Načítame M-2 blokov R a 1 blok rezervujeme na čítanie S. Postupne čítame všetky bloky z S, vyhodnocujeme joinovaciu podmienku a výsledky zapisujeme na výstup.
Ak má R ďalšie bloky, načítame B-2 blokov R a zopakujeme čítanie celej S po 1 bloku. Počet opakovaných čítaní S je $\lceil b(R)/(M-2) \rceil$.

Celková cena je $b(R) + \lceil b(R)/(M-2) \rceil \cdot b(S)$.
(Nezahŕňa zapisovanie výstupu, ale jeho veľkosť nezávisí od spôsobu výpočtu joinu.)

Fyzické operátory — nested loop join

Úloha. Počítame dotaz

```
SELECT r.a FROM r, s WHERE r.a = s.a
```

Relácia r je na disku uložená v 100 blokoch, relácia s v 50 blokoch. K dispozícii v RAM je 20 blokov (bloky na disku a v RAM sú rovnako veľké).
(a) Popíšte výpočet metódou nested loop join a zistite počet potrebných I/O operácií.
(b) Dokážte, že iné rozdelenie blokov v RAM medzi r a s by viedlo k suboptimálnemu výsledku.

Literatúra

- https://cs186berkeley.net/notes/note8/
- https://cs186berkeley.net/notes/note9/
- https://cs186berkeley.net/notes/note10/
- nested loop join
- https://www.db-book.com/slides-dir/PDF-dir/ch14.pdf
- https://www.db-book.com/slides-dir/PDF-dir/ch15.pdf
- https://www.db-book.com/slides-dir/PDF-dir/ch16.pdf
- https://pganalyze.com/docs/explain/scan-nodes
- https://www.postgresql.org/docs/current/performance-tips.html