UNIVERZITET U NIŠU
ELEKTRONSKI FAKULTET
Katedra za računarstvo

Obrada i optimizacija upita u SQLite

Sistemi za upravljanje bazama podataka

Student: Todor Majstorovic 1088

Sadrzaj

1. Uvod u sqlite
2. SQL upiti
3. SQLite indeksi
4. Optimizator upita
5. Next generation query planner
6. Reference
7. Zakljucak

1. Uvod

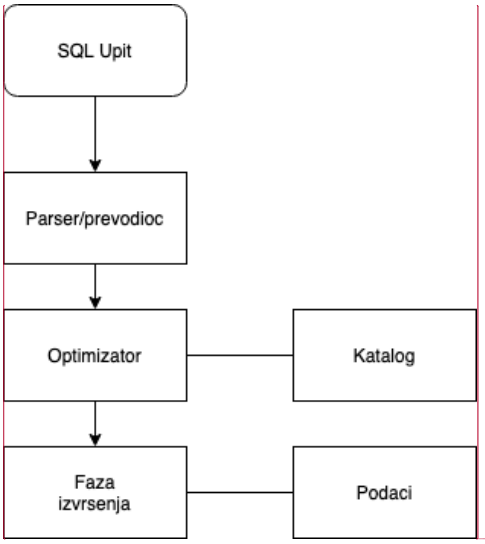
SQLite predstavlja bazu podataka koja se pakuje uz aplikacije. Nije potreban poseban server za njen rad. SQLite je najrasprostranjenija baza podataka na svetu koja se koristi u mnogim aplikacijama.

SQL baza podataka cita sa diska i upisuje na disk. Kompletna baza sa svim podacima se nalazi u jednom fajlu, koji ukljucuju sve tabele, indexe, trigere i poglede. Format baze je cross-platform-ni sto znaci da se ista baza moze iskopirati na svim operativnim sistemima i arhitekturama racunara.

SQLite je kompaktna biblioteka sa svim ukljucenim delovima,ne prelazi velicinu od 600KB,u zavisnosti od ciljane platforme i optimizacije kompajlera.Medjutim, postoji zavisnost izmedju brzine i iskoriscenje memorije, sto je veca memorija rezervisana utoliko je i brzina veca. Ukoliko se iskoristi pravilno SQLite moze biti brzi i od direktnog upisa i citanja u fajl sistem. Sve prethodno navedene odlike SQLite-a je cine prvim izborom za bazu podataka za male i srednje projekte, gde nije potrebna ogromna baza.

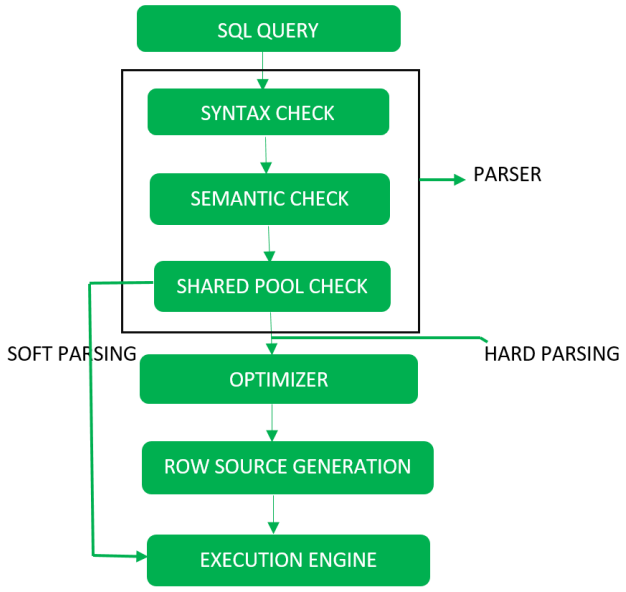
2. SQL upiti

SQL upit je glavno sredstvo za pristup podacima koji se nalaze u bazi. Pravilno definisan i optimizovan upit moze predstavljati razliku u izvrsenju od 1 sekunde do vise-satnog izvrsenja. Kada SQL upit stigne do baze, pre izvrsenja on mora da prodje kroz vise faza.



Slika 1. Blok diagram obrade upita

Commented [tm1]: Follow Up:



Slika 2. Detaljni blok diagram obrade SQL upita

Koraci pri procesiranju SQL upita od strane optimizatora upita su:

- Parser-nakon prevođenja u relacionu algebru, parser vrsi sintaksnu, semantičku i zajedničku proveru(Slika 2).
  Sintaksne provere :
  -Potvrdjuje sintaksnu validnost SQL upita
  *SELECT * FORM employee*
  U ovom primeru je pogrešno spelovan FROM i izbaciće grešku

  Semantičke provere
  -Proverava se da li naredba ima smisla.
  Na primer, ukoliko se nekim upitom pristupa tablici koja ne postoji, javila bi se ova greška

  Provera hash koda

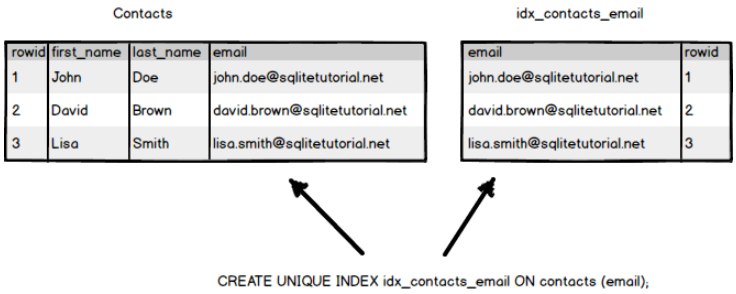
, Svaki upit poseduje hash kod za vreme izvršenja, tako da se ovime proverava da li postoji hash kod, i ukoliko postoji neće biti potrebni dodatni koraci za optimizaciju i izvršenje upita

- Grubo i fino parsiranje Ukoliko postoji upit čiji hash kod ne postoji, taj upit mora da prodje kroz nekoliko faza koje se naziva grubo parsiranje, ukoliko kod postoji, onda upit ne prolazi kroz ove faze, već ide direktno do izvršnog programa(Slika 2). To je fino parsiranje.
- Optimizator-Za vreme optimizacije, baza podataka mora da izvrsi bar jedno grubo parsiranje. Katalog cuva izvršne planove pa optimizator salje najjeftiniji plan za izvršenje.
- Generisanje izvornih redova predstavlja softver koji prima optimalni plan za izvršenje od optimizatora i pravi iterativni plan izvršenja koji može da se koristi od strane cele baze. Ovaj plan predstavlja binarni program koji kad se izvrši od strane SQL-a generiše skup rezultata.
- Izvršna faza: Izvršava se upit i generiše se rezultat

3. SQLite indeksi

U relacionim bazama podataka, tabela predstavlja listu redova. Svaki red ima identični strukturu kolona koja se sastoji od ćelija. Svaki red ima parametar rowid koji predstavlja identifikator reda u tabeli. Tabela se može smatrati kao listom parova (rowId, red).

Suprotno tabeli, indeks ima suprotnu vezu: (red, rowId). Indeks predstavlja dodatnu bazu podataka koja pomaže u izvršenju upita.



Slika 3. SQLite index

SQLite koristi B-stablo za organizaciju indeksa. B predstavlja balansirano, a ne binarno stable.

Balansirano stablo organizuje podatke tako da obe strane imaju isti broj elemenata, kako bi obilazak potreban da se nadje odgovarajući red uvek bio isti. Takodje jedna od funkcionalnosti balansiranog stabla je da ima izuzetno efikasnu pretragu korisćenjem jednakosti (=) ili (>, >=, <,<=) nad indeksima.

CREATE [UNIQUE] INDEX *index_name* ***ON*** *table_name(column_list);*

4. Optimizator Upita

Za svaki SQL upit postoji veliki broj načina na koji on moze da se izvrši, tako da je svrha optimizatora da nadje najoptimalniju opciju kako bi smanjio vreme koje je potrebno da se upit izvrši, kao i redukovati korišćenje memorije i resursa koje taj upit koristi.

4.1. Analiza Where upita

Where upit se razdvaja u uslove gde su svi uslovi razdvojeni AND operatorom.
Da bi se WHERE upit iskoristio od strane indexa, član mora biti u jednom od sledećih formi

column = expression
column IS expression
column > expression
column >= expression
column < expression
column <= expression
expression = column
expression > column
expression >= column
expression < column
expression <= column
column IN (expression-list)
column IN (subquery)
column IS NULL

Ukoliko je indeks kreairan upotrebom naredbe

CREATE INDEX idx_ex1 ON ex1(a,b,c,d,e,...,y,z);

onda se taj indeks samo ukoliko se primarne kolone indeksa(a,b,c...) pojavljuju u formama koje su kompaktne sa WHERE upitom. Primarne kolone se moraju praviti sa *=*, *IN* ili *IS* operatorima. Prva kolona sa desne strane može sadržati nejednakosti. Ova kolona može sadržati najviše 2 nejednakosti koje se moraju nalazile sa obe strane dozvoljene vrednosti. Indeksi se ne moraju prikazivati u WHERE upitu da bi se mogli koristiti, ali ne sme se desiti da postoje

praznine u kolonama indeksa koji se koriste. Tako da za gorepomenuti primer, ukoliko se c ne uklapa u ograničenja WHERE upita, onda se mogu koristiti samo izrazi koji rade sa indeksima a i b. Ukoliko se radi sa izrazima za indekse, svaka "column" naredba se može zameniti sa "indexed expression" i neće se javljati nikakva greška.

4.1.1 Korišćenje indeksa

-Za gorepomenuti indeks i WHERE upit
... WHERE a=5 AND b IN (1,2,3) AND c IS NULL AND d='hello'
prve četiri kolone bi se mogle koristiti zato što one predstavljaju prefiks indeksa povezane su ogranicenjima jednakosti.;
-Za gorepomenuti indeks i WHERE upit
... WHERE a=5 AND b IN (1,2,3) AND c>12 AND d='hello'
moglo bi se pristupiti samo kolonama a, b i c, kolona d bi bila nepristupačna zato što je sa desne strene kolone koja je ograničena nejednakošću(kolona c);
-Za gorepomenuti indeks i WHERE upit
... WHERE a=5 AND b IN (1,2,3) AND d='hello'
Samo bi se moglo pristupati kolonama a i b zato što kolona c nije ograničena ničime a u WHERE upitu ne smeju se pojavljivati praznine;
-Za gorepomenuti indeks i WHERE upit
... WHERE b IN (1,2,3) AND c NOT NULL AND d='hello'
Ne bi se moglo pristupati ni jednoj koloni zato što prva kolona sa leve strane nije ograničena. Pod pretpostavkom da nema drugih indeksa, ovaj upit bi skenirao celu tabelu.
-Za gorepomenuti indeks i WHERE upit
... WHERE a=5 OR b IN (1,2,3) OR c NOT NULL OR d='hello'
U ovom slučaju WHERE upit se ne bi mogao koristiti zato što su izrazi povezani operatorom OR umesto operatora AND. I ovaj upit bi skenirao celu tabelu.

4.2 BETWEEN naredba

Ukoliko je WHERE upit ovog formata

expr1 BETWEEN expr2 AND expr3

onda se kreiraju dva "virtuelna" izraza

expr1 >= expr2 AND expr1 <= expr3

Virtuelni izrazi se koriste samo za analizu i ne generišu nikakav kod. Ukoliko se virtualni izrazi iskoriste kao ograničenja indeksa, onda se prvobitna BETWEEN naredba ignoriše i ne izvršava se test na potrebne kolone. Tako da ukoliko se BETWEEN naredba iskoristi kao ograničenje za indeks, ne izvršava se nikakva naredba na taj izraz. Medjutim ukoliko se BETWEEN naredba ne iskoristi za ograničenje, nego se iskoristi za testiranje ulaznih kolona, expr1 izraz se evaluira samo jednom.

4.3 OR optimizacija

Ograničenja WHERE Upita koja su povezana operatorom OR umesto operatora AND mogu da se razreše u 2 različita načina. Ukoliko se izraz sastoji od vise podizraza koji sadrze prosto ime kolone i odvojeni su operatorom OR:

column = expr1 OR column = expr2 OR column = expr3 OR ...

U tom slučaju se ovaj izraz prevodi u :

column IN (expr1,expr2,expr3,...)

Prevedeni izraz nakon toga moze da bude ograničenje za bilo koji indeks na isti način kao i svaki "IN" operator. Medjutim ograničenje je da kolona mora da bude ista u svakom podnizu koji je povezan OR-ovima, ali ona može da se nađe na bilo kojoj strani operatora *=*.

Ukoliko dođe do greške ili ovakav prevod iz OR u IN ne radi, pokušava se druga OR optimizacija. Recimo da se OR sastoji od vise podizraza;

expr1 OR expr2 OR expr3

Podnizovi mogu biti poređenja samo jednog izraza kao na primer *a=5* ili "x>y", ili mogu biti LIKE ili BETWEEN naredbe, ili taj podizraz može da bude veči broj pod-podizraza koji se nalaze u zagradi i povezani su AND operatorima. Svaki podizraz se analizira i proverava kao da on sam čini WHERE naredbu da bi se utvrdilo da li je sam podizraz indeksabilan. Ako je svaki podizraz OR naredbe indeksabilan, onda se i cela OR naredba može kodirati na način da poseban indeks računa po jedan izraz OR naredbe.

Ovo se može ilustrovati kod primera:

rowid IN (SELECT rowid FROM table WHERE expr1
UNION SELECT rowid FROM table WHERE expr2
UNION SELECT rowid FROM table WHERE expr3)

Preveden izraz je konceptualan; WHERE naredbe koje sadrže OR se ne prevode bas a ovako. Implementacija OR naredbe koristi princip koji je efikasniji i radi čak i za WITHOUT ROWID tabele ili tabele koje ne mogu pristupiti "rowid"-u. Međutim suština implementacije je to da se različiti indeksi koriste da pronađu kandidate rezultujućih vrsta iz svake OR naredbe, a rešenje je unija svih ovih vrsta.

U većini slučajeva SQLite koristi samo jedan indeks za svaku tabelu u FROM naredbi upita. Optimizacija druge OR naredbe je izuzetak. U OR naredbi, različit indeks se može koristiti za svaki podizraz u OR naredbi.

Za svaki upit, optimizacija OR naredbe ne mora da garantuje korišćenje iste. SQLite koristi planer koji se bazira na iskorišćenosti resursa, koji izračunava koliko će se procesor i ulazno-izlazin signali koristiti, i bira najoptimalniji. Ukoliko ima više OR izraza u jednoj WHERE naredbi ili ako neki indeksi podizraza OR naredbe nisu previše probirljivi,SQLite može da se opredeli za drugačiji algoritam, ili čak i za skeniranje cele tabele.

4.4 LIKE optimizacija

WHERE naredba koja koristi LIKE ili GLOB operatore može se koristiti uz indeks da bi se izvršila pretraga opsega, kao da su LIKE i GLOB zamena za BETWEEN operator. Plan izvršenja upita predstavlja skup sledećih stvari:

- Desna strana LIKE ili GLOB operacije mora da bude ili string ili parametar koji je vezan za string i koji ne počinje wildcard-om.
- Ni LIKE ni GLOB operatori ne smeju imati vrednost true tako što će imati brojnu vrednost na levoj strani. Znači ili je :
  A. leva strana LIKE ili GLOB operatora je ime indeksirane kolone sa afinitetom za tekst ili
  B. desna strana LIKE ili GLOB operatora ne počinje sa minusom(-) niti brojem.
  Ovo ograničenje je tu zato što se brojevi ne sortiraju u leksikografskom redosledu. Na primer 9<10 ali '9'>'10'
- Prilikom implementacije LIKE i GLOB operatora ne smeju se overloadovati ugrađene funkcije korišćenjem sqlite3_create_function() API-a
- Za GLOB operator, kolona se mora indeksirati pomoću ugrađene binarno-kolabirajuće sekvence.
- Za LIKE operator ukoliko je uključen case sensitive like mod, onda se kolone takođe moraju indeksirati preko binarno-kolabirajuće sekvence, u suprotnom mora se koristiti NOCASE kolabirajuća sekvenca.
- Ukoliko se iskoristi ESCAPE Opcija, ESCAPE karakter mora biti iz ASCII tabele, ili jednobitni karakter iz UTF-8

LIKE operator ima 2 režima koji se mogu izabrati uz pomoć pragme. Default režim za LIKE da ne bude case sensitive. To jest, sledeci izraz je istinit

'a' LIKE 'A'

Međutim ukoliko je case_sensitive_like pragma uključena:

PRAGMA case_sensitive_like=ON;

U tom slučaju LIKE operator vodi računa o velikim i malim slovima i gornji izraz bi bio netačan. Međutim ovo je samo bitno za ASCII karaktere, ostala međunarodna slova i izrazi su svi case sensitive i vodi se računa o velikim i malim slovima.

LIKE operator je po defaultu case insensitive jer je to standard koji je opisan SQL-om. Ovo se vrlo lako može promeniti korišćenjem SQLITE_CASE_SENSITIVE_LIKE naredbe.

4.5. Skip-Scan optimizacija

Pravilo je da su indeksi jedino korisni ukoliko postoji ograničenja WHERE naredbe na prvoj koloni sa leve strane indeksa. Međutim, u nekim slučajevima SQLite može da koristi indeks čak i ako su prvih par kolona indeksa izostavljene iz WHERE naredbe i samo se koriste kasnije kolone.

Zamislite ovakvu tabelu:

```
CREATE TABLE people(
 name TEXT PRIMARY KEY,
role TEXT NOT NULL,
height INT NOT NULL, -- in cm
CHECK( role IN ('student','teacher') )
);
CREATE INDEX people_idx1 ON people(role, height);
```

Ova tabela ima jedan član za svaku osobu u velikom udruženju. Svaka osoba je ili studenat ili profesor, sto je određeno sa "role" poljem. I čuva se visina u centimetrima svake osobe. Rola i visina su indexirane. Vidimo da kolona koja se nalazi na prvom mestu sa leve strane sadrži samo dve moguće vrednosti, znači nemamo neki preveliki izbor.

Sada recimo da imamo upit koji bi tražio imena svake osobe u ovom udruženju koja je viša od 180cm.

```
SELECT name FROM people WHERE height>=180;
```

Sa obzirom da se prva kolona sa leve strane indeksa ne nalazi u WHERE naredbi upita, pretpostavlja se da indeks nije moguće iskoristiti ovde. Međutim SQLite ga ipak može iskoristiti. SQLite koristi indeks kao da upit izgleda ovako :

```
SELECT name FROM people
WHERE role IN (SELECT DISTINCT role FROM people)
```

AND height>=180;

odnosno

SELECT name FROM people WHERE role='teacher' AND height>=180
UNION ALL
SELECT name FROM people WHERE role='student' AND height>=180;

Ova 2 alternativna upita su samo konceptualna. SQLite zapravo ne transformise upite. U realnom vremenu to izgleda ovako: SQLite pronadje prvu moguću vrednost za "role", tako što vraća indeks the "people_idx1" na početak i čita prvi upis. SQLite upisuje prvu vrednost u internu promenljivu koja se zove "$role", nakon čega on pokreće upit tipa : "SELECT name FROM people WHERE role=$role AND height>=180. Ovaj upit ima jednakost kao ograničenje na prvoj koloni sa leve strane indeksa tako da se indeks može koristiti za rešavanje upita. Kada se upit izvrši, SQLite koristi "people_idx1" index kako bi pronašao sledeću vrednost "role" kolone, koristeći kod sličan "SELECT role FROM people WHERE role>$role LIMIT 1". Ova nova vrednost se upisuje preko vrednosti $role promenljive, i tako sve dok se ne ispišu sve moguće vrednosti za "role".

Ovo se naziva "skip-scan" zato što baza praktično radi potpuno skeniranje indeksa ali ga optimizuje tako što ponekad skoči na narednu vrednost.

SQLite će iskoristi skip scan ako zna da u kolonama postoje duplikati vrednosti. Jedini način da SQLite zna da u nekoj koloni postoje duplikati je ako se iskoristi ANALYZE naredba nad bazom. Bez ANALYZE naredbe, SQLite mora da nagađa izgled baze, a default vrednost je da baza ima oko 10-tak duplikata za svaku vrednost u prvoj koloni sa leve strane indeksa. Međutim skip scan optimizuje tek kada ima 18-tak duplikata tako da se bez ANALYZE naredbe skip scan nikad ne koristi

4.6 Udruživanja (Joini)

ON i USING naredbe unutrasnjeg joina su konvertovane u dodatne izraze WHERE naredbe pre nego što se izvrši analiza WHERE naredbe. Tako da SQLite nema nikakvih prednosti u korišćenju novije sintakse SQL92-e umesto stare SQL89-e tačka-join sintakse.

Za LEFT OUTER JOIN situacija je kompleksnija. Ova dva upita nisu identična:

*SELECT * FROM tab1 LEFT JOIN tab2 ON tab1.x=tab2.y;*
*SELECT * FROM tab1 LEFT JOIN tab2 WHERE tab1.x=tab2.y;*

Za unutrašnju join naredbu, ova dva upita bi bila identična. Međutim drugacije je procesiranje kod ON i USING naredba koji su u OUTER joinu: preciznije, ograničenja ON i USING naredba se ne primenjuju ukoliko je desna tabela joina NULL vrsta, ali se primenjuju ograničenja u WHERE naredbi. Ovo praktično ON i USING naredbe LEFT JOIN-a u WHERE naredbi pretvaraju u običan INNER JOIN- doduše koji se malo sporije izvršava

4.6.1 Uredjenost tabela u Join-u

Trenutna implementacija SQLite-a koristi samo joine koji su petlje, odnosno joini se implementiraju kao ugnježdene petlje.

Default redosled ugnježdenih petlji u joinu je ta da prva tabela sa leve strane u FROM naredbi formira spoljašnju petlju i prva tabela sa desne formira unutrašnju petlju. Sqlite ugnježda petlje u drugačijem redosledu ukoliko će tako imati bolje indekse

Unutrašnji joini mogu da se preraspodeljuju koliko god puta to želimo. Dok levi spoljašnji join nije ni interaktivan ni asocijativan i zbog toga ne smemo da ga preraspodeljujemo. Unutrašnji joini na levo i desno od spoljašnjeg joina mogu da se preraspodeljuju ako optimizator misli da će tako postići optimalnije rešenje.

SQLite tretira CROSS JOIN operator na specijalan način. CROSS JOIN Operator je komunikativan u teoriji, ali SQLite nikad ne preraspodeljuje tabele u CROSS JOINU. Ovim se stvara mehanizam u kojem programer može da primora SQLite da izabere redosled u kome će se petlje ugnježdavati.S

SQLite koristi efikasan polynomsko-vremenski algoritam. Zbog ovoga SQLite moze da isplanira upite sa 50-60 joina u par milisekundi.

Preraspodela joina je automatska i uglavnom radi dovoljno dobro da programeri ne moraju sami da razmišljaju o tome, posebno ako se koristi ANALYZE da bi se dobile neke statistike za dostupne indekse. Međutim nekad je potrebno da programer da poneku pomoć. Recimo, za sledeću šemu;

```
CREATE TABLE node(
  id INTEGER PRIMARY KEY,
  name TEXT
);
CREATE INDEX node_idx ON node(name);
CREATE TABLE edge(
  orig INTEGER REFERENCES node,
  dest INTEGER REFERENCES node,
  PRIMARY KEY(orig, dest)
);
```

```
CREATE INDEX edge_idx ON edge(dest,orig);
```

Ova šema definiše graf koji može da čuva ime na svakom polju.

Sad recimo da imamo ovakvu šemu:

```
SELECT *
  FROM edge AS e,
       node AS n1,
       node AS n2
 WHERE n1.name = 'alice'
   AND n2.name = 'bob'
   AND e.orig = n1.id
   AND e.dest = n2.id;
```

Ovaj upit traži svu informaciju za ivice koje idu od polja "alice" do polja bob". Optimizator ima 2 opcije implementacije ovog upita: Pseudokodom su prikazana oba:

OPCIJA 1

```
foreach n1 where n1.name='alice' do:
  foreach n2 where n2.name='bob' do:
    foreach e where e.orig=n1.id and e.dest=n2.id
      return n1.*, n2.*, e.*
    end
  end
end
```

OPCIJA 2

```
foreach n1 where n1.name='alice' do:
  foreach e where e.orig=n1.id do:
    foreach n2 where n2.id=e.dest and n2.name='bob' do:
      return n1.*, n2.*, e.*
    end
  end
end
```

Isti indeksi su iskorišćeni za ubrzavanje petlje u obe implementacije. Jedina razlika je u tome u kom su redosledu ugnježdene.

Kako znati koji upit je bolji? Zapravo, odgovor na to pitanje zavisi od oblika podataka koji se nalaze u poljima i ivicama tabela.

Recimo da je broj alice polja M a broj bob polja N. Postoje 2 scenarija. U prvom M i N su 2 ali postoje na hiljada ivica u svakom polju. U ovom slučaju bolje je izabrati OPCIJU 1. Sa opcijom 1, unutrašnja petlja proverava da li postoji ivica izmedju polja i ispisuje rezultat. Međutim posto postoje samo 2 polja za alice i boba, unutrašnja petlja mora da se izvrši 4 puta i zato je ovaj upit dosta brz u ovom slučaju. Opcija 2 u ovom slučaju bi trajala mnogo duže. Spoljašnja petlja opcije 2 se samo 2 puta izvrši, ali posto postoji veliki broj ivica, srednja petlja mora da iterira vise hiljada puta. Ova opcija bi bila mnogo puta sporija tako da je bolje koristiti opciju 1.

Sada recimo da imamo slučaj kada su i N i M =3500. Postoji velika količina alice nodova. Ali recimo da su svaka 2 polja povezana samo jednom ili dvema ivicama. U ovom slučaju Opcija 2 je mnogo bolja. Sa njom, Spoljašnja petlja mora da se izvrši 3500 puta, ali srednja petlja će se izvršiti samo jednom ili dva puta za svaku spoljašnju petlju i unutrašnja petlja će se izvršiti maximalno jednom za svaku srednju petlju koja se izvrši.Tako da će totalni broj iteracija unutrašnje petlje biti oko 7000, Opcija 1, u ovom slučaju ce morati da pokrene i spoljašnju i srednju petlju 3500 puta, što je 12 miliona iteracija srednje petlje. Tako da je u ovom slučaju opcija 2 oko 2000 puta brža od opcije 1.

4.6.2 Ručno kontrolisanje plana izvršenja upita putem CROSS_JOIN

Programeri mogu da primoraju SQLite da izvršava odredjeni redosled ugnježdavanja petlji u joinu koristeći CROSS JOIN operator umesto JOIN-a. INNER JOIN-a, NATURAL JOIN-a ili "," join-a. Iako su CROSS JOIN-i komutativna u teoriji, SQLite nikad ne preraspoređuje tabele u CROSS-JOINU. Stoga je leva tabela CROSS JOIN-a uvek spoljašnja petlja u odnosu na desnu tabelu. rogrammers can force SQLite to use a particular loop nesting order for a join by using the CROSS JOIN operator instead of just JOIN, INNER JOIN, NATURAL JOIN, or a "," join. Though CROSS JOINs are commutative in theory, SQLite chooses to never reorder the tables in a CROSS JOIN. Hence, the left table of a CROSS JOIN will always be in an outer loop relative to the right table.
U sledecem primeru optimizator može da prerasporeduje tabele iz FROM naredbe kako god on želi;

```
SELECT *
  FROM node AS n1,
       edge AS e,
       node AS n2
 WHERE n1.name = 'alice'
   AND n2.name = 'bob'
   AND e.orig = n1.id
   AND e.dest = n2.id;
```

Međutim u sledećoj ekvivalentnoj formulaciji istog upita, zamena CROSS JOINA umesto "," obezbeđuje da tabele moraju biti u rasporedu N1,E,N2

```
SELECT *
  FROM node AS n1 CROSS JOIN
       edge AS e CROSS JOIN
       node AS n2
```

```
WHERE n1.name = 'alice'
  AND n2.name = 'bob'
  AND e.orig = n1.id
  AND e.dest = n2.id;
```

Medjutim u drugom upitu, mora se izvršavati pcija 2. Zapamtite da morate koristiti ključnu reč CROSS da bi isključili preraspodelu tabela za optimizaciju, INNER JOIN, NATURAL JOIN, JOIN i ostale combinacije se ponašaju isto kao i normalan join i optimizator raspodeljuje tabele kako on želi.

4.7. Biranje između većeg broja indeksa

Svaka tabela iz FROM naredbe upita može koristiti najviše jedan indeks i SQLite teži da koristi barem jedan indeks u svakoj tabeli. Ponekad, dva ili vise indeksa mogu biti kandidati za izbor u jednoj tabeli. Recimo :

```
CREATE TABLE ex2(x,y,z);

CREATE INDEX ex2i1 ON ex2(x);

CREATE INDEX ex2i2 ON ex2(y);

SELECT z FROM ex2 WHERE x=5 AND y=6;
```

Za ovu SELECT naredbu, optimizator moze da koristi ex2i1 indeks da pretraži vrste ex2 koje sadrže x=5 i da onda testira svaku vrstu u odnosu na y=6 izraz. Takođe može da se uzme i ex2i2 indeks da pretraži vrste ex2 koje sadrže y=6 i onda ih testira u odnosu na x=5 izraz.

Kada dođe u situaciju u kojoj mora da bira između 2 ili više indexa, SQLite pokušava da izračuna koje od tih opcija je najoptimaltija i izabira je.

4.7.1 Opseg upita

Recimo da imamo neki ovakvu situaciju:

CREATE TABLE ex2(x,y,z);

CREATE INDEX ex2i1 ON ex2(x);

CREATE INDEX ex2i2 ON ex2(y);

SELECT z FROM ex2 WHERE x BETWEEN 1 AND 100 AND y BETWEEN 1 AND 100;

Zatim pretpostavimo da kolona x sadrži vrednosti između 0 i 1,000,000 i da kolona y sadrži vrednosti od 0 do 1,000. U ovom slučaju, ograničenje opsega nad kolonom x bi trebalo da smanji veličinu prostora koji treba da se pretražuje za 10,000, dok to isto za y bi trebalo da smani isti taj prostor za samo 10 puta. Tako znamo da je bolji indeks ex2i1.

SQLite bi sam dosao do ovog zaključka, ali samo ukoliko je kompajliran sa SQLITE_ENABLE_STAT3 ili SQLITE_ENABLE_STAT4. SQLITE_ENABLE_STAT3 i SQLITE_ENABLE_STAT4 opcije omogućavaju ANALYZE komandi da prikupi istoriju onoga sto se nalazilo u kolonama u sqlite_stat3 ili sqlite_stat4 tabele i koristi ovaj histogram da bolje odredi koji je najbolji upit da iskoristi u opsegnom ograničavanju. Glavna razlika između STAT3 i STAT4 je da STAT3 čuva histogram samo za prvu kolonu sa leve strane a STAT4 čuva podatke za sve kolone jednog indeksa. Za indekse sa samo jednom kolonom STAT3 i STAT4 imaju istu funkciju

Podaci iz histograma su jedino korisni ukoliko je desna strana ograničenja prosta compile-time konstanta ili parametar ali ne i izraz.

Jos jedna mana histograma je ta da se on samo odnosi na prvu kolonu sa leve strane indeksa. Recimo kad bi imali ovu situaciju:

CREATE TABLE ex3(w,x,y,z);

CREATE INDEX ex3i1 ON ex2(w, x);

CREATE INDEX ex3i2 ON ex2(w, y);

SELECT z FROM ex3 WHERE w=5 AND x BETWEEN 1 AND 100 AND y BETWEEN 1 AND 100;

Ovde su nejednakosti na kolonama x i y koje nisu prve sa leve strane. Zbog toga histogram je beskoristan zato što on uzima jedino podatke za prvu kolonu sa leve strane.

4.8 Covering index(pokrivajući indeks)

Pri indeksiranju petlje, procedura je uglavnom da se uradi binarno traženje po indeksu da bi se nasao ulaz, onda se izvuče rowID iz indeksa i koristi se da se uradi binarno traženje originalne tabele. Zbog toga se indexirana pretraga uglavnom sastoji od 2 binarna traženja. Međutim ukoliko su sve kolone već na raspolaganju u samom indeksu. SQLite će da iskoristi vrednosti koje se nalaze u indeksu i nikad nece tražiti po vrstama prvobitne tabele. Ovako se štedi za po 1 binarno traženje za svaku vrstu i time se upiti duplo ubrzavaju.

Kada se u indeksu nalazi svi potrebni podaci koji su potrebni za upit, i kad se nikad ne treba vraćati na prvobitnu tabelu, taj indeks zovemo pokrivajući index.

5.Next Generation Query Planner - NGQP

Zadatak planera upita je da pronadje najbolji algoritam ili plan upita kako bi izvršio SQL naredbu. OD SQLite verzije 3.8.0, prepisana je komponenta za planer upita kako bi brže i bolje generisala planove. Ova prepisana komponenta je nazvana planer upita nove generacije (NGQP)

NGQP je u većini slučajeva bolji od svog prethodnika. Medjutim, mogu postojati problemi sa backwards-compatibility, odnosno da dodje do regresije performansi prilikom prelaska na noviji planer. Rizici su detaljno opisani zajedno sa načinom kako da se te greške isprave.

Prilikom izvršenja jednostavnih upita nad jednom tabelom sa nekoliko indeksa, trivijalan je odabir najboljeg algoritma. Sa porastom kompleksnosti upita, poput multi-way joina sa mnogo indeksa i podupita, moguće je postojanja velikog broja potencijalnih algoritama za dobijanje rezultata. Zadatak planera je da odabere najbolji mogući algoritam.

Planeri upita predstavljaju glavnu odliku SQL baza. Oslobadjaju programera od izbora odredjenog plana, pri čemu on može da se fokusira na aplikacione probleme koje se tiču korisnika. Kada se poveća kompleksnost upita, dobar planer može napraviti ogromnu razliku u brzini pristupa, što će u mnogome olakšati izradu aplikacije.

SQLite rešava veze pomoću ugnježdenih petlji, jedna petlja za svaku tabelu u join-u. Jedan ili više indeksa se se mogu koristiti u svakoj petlji za ubrzavanje pretrage ili se može vršiti full table scan koji čita sve redove.

Deli se u dva dela:
1. Odabir ugnježdenog redosleda petlji
2. Odabir dobrih indeksa za svaku petlju

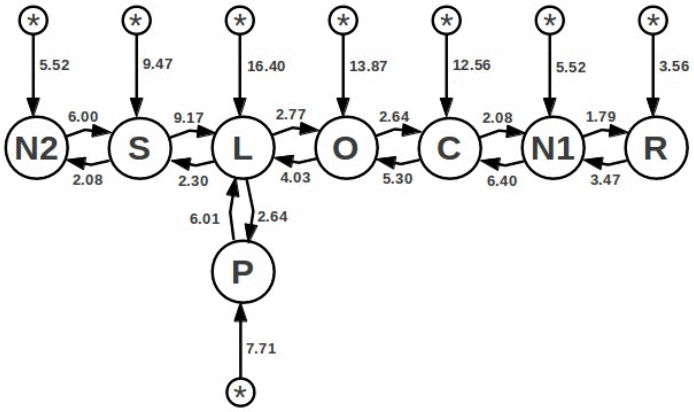
Odabir redosleda je najteži deo optimizovanja. Ukoliko se ovaj zadatak izvrši kvalitetno, odabir indeksa je uglavnom očigledan.

5.1 QPSG

Jedna korisna opcija za razvoj aplikacija je QPSG, odnosno Garant stabilnosti planera upita, koji podrazumeva da će SQLite odabrati isti plan za upit pod uslovom da se
1. Šema baze ne menja , poput dodavanja-brisanja indeksa
2. ANALYZE komanda ne izvrši ponovo
3. Promeni verzija SQLite baze.
Defaultna vrednost je disabled za ovu opciju, i može se uključiti opcijom *SQLITE_ENABLE_QPSG*, ili u run-time-u *sqlite3_db_config(db,SQLITE_DBCONFIG_ENABLE_QPSG,1,0).*

TPC-H Q8
TPC-H Q8 je predstavlja JOIN koji se sastoji od 8 povezanih tabela. Glavni zadatak planera predstavlja pronalazak najjeftinijeg ugnježdavanja petlji koje predstavljaju JOIN-ove. Cena je logaritmička, sa ugnježdenim join-ovima ukupna cena je proizvod pojedinačnih. Rešenje ovog problema je u suštini najjeftiniji obilazak grafa. Da bi našli realnu cenu, moramo proći kroz ceo graf po svim putevima. SQLite pretpostavlja najjeftiniji put na osnovu indeksa i ograničenja unutar WHERE naredbe. Ova nagadjanja ne moraju biti uvek tačna, ali se korišćenjem ANALYZE komande mogu prikupiti tačniji statistički podaci.

Pre uvođenja NGQP, SQLite je koristio NN heuristiku, koja bira najjeftiniji put između suseda. Ova heurstika je izuzetno brza i uglavnom tačna, tako da SQLite može pronaći dobar plan čak i za 64 JOINA. Ostale baze imaju problem kada broj JOINA predje 10.



Slika 4. TPC-H Q8-way join

Put koji pronalazi NN heuristika za TPC-H Q8 nije optimalan. Put koji je pronašla R-N1-N2-S-C-O-L-P ima cenu 36.92. Najoptimalniji put je P-L-O-C-N1-R-S-N2 sa cenom sa 27.38. Iako ova razlika ne deluje preveliko, logaritamske su cene pa je realna razliku u performansama za drugi put 750 puta brži od prvog.

Da bi se obišli svi putevi u grafu, potrebno je proći kroz sve puteve da bi se odredio najefikasniji put. Ta operacija je medjutim veoma skupa i uglavnom se ne obilaze svi putevi, već se procenjuje koji će biti najefikasniji.

5.2 NGQP N Nearest neighbors heuristika

Umesto uzimanja u obzir najjeftinijeg puta do suseda, NGQP uzima N najboljih puteva u svakom koraku.
Primer za N=4. U prvom koraku se uzimaju
Prvi korak

R	(cena:	3.56)
N1	(cena:	5.52)
N2	(cena:	5.52)

P (cena: 7.71)

Drugi korak nalazi 4 najkraća puta počevši od koraka iz prethodnog primera. Ukoliko su dva puta ekvivalentna, uzima se prvi najjeftiniji.

Drugi korak:

R-N1	(cena:	7.03)
R-N2	(cena:	9.08)
N2-N1	(cena:	11.04)
R-P	(cena:	11.27)

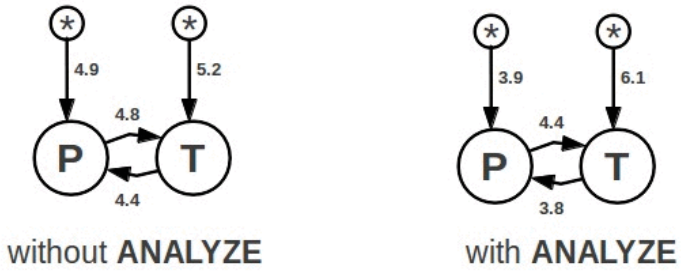
Treći korak:

R-N1-N2	(cena:	12.55)
R-N1-C	(cena:	13.43)
R-N1-P	(cena:	14.74)

R-N2-S (cena: 15.08)

Pošto TPC ima 8 čvora, ovaj proces je potrebno obaviti 8 puta. U generalizovanom slučaju, kada imamo K spoja, prostorni zahtev je O(n), a složenost izvršenja je O(n*k) što je prostije od O(2^k)

Kako se odredjuje N? Ukoliko se uzme N=K, optimalnost algoritma je O(K^2) što je još uvek gotovo optimalno, ali ne daje uvek najbolje rešenje. SQLite je implementiran tako da se koristi N=1 za jednostavne upite, N=5 za jedan JOIN, i N=10 kada ih ima više.



Slika 5. NN heuristika. Uporedjivanje sa i bez prethodnog izvršrnjsaANALYZE.

5.3 NGQP rizici i saveti

- **Kreirajte odgovarajuće indekse.** Uglavnom problemi optimizacije nastaju zbog korišćenja neodgovarajućih indeksa, a ne zbog problema sa planerom. Osigurajte da postoje indeksi za svaki veći upit.
- **Ne pravite indekse niskog kvaliteta.** Ovo podrazumeva indekse koji imaju istu vrednost za prvu kolonu sa leve strane, tj. Izbegavajte korišćenje bool ili enum vrednosti za prvu kolonu sa leve strane u indeksu**.**
- **Ako koristite indekse niskog kvaliteta, pre izvršenja obavezno izvršite ANALYZE komandu.**
- **Koristite EXPLAIN QUERY PLAN,** kako bi mogli što brže pronaći upit ili deo upita koji vam usporava performanse. Prikazuje se način izvršenja upita, odnosno da li se koriste indeksi ili ne prilikom izvršenja i kako se ugnježdavaju tabele prilikom spoja.
- **Koristite likelihood i unlikely() SQL funkcije.** SQLite pretpostavlja da su izrazi unutar WHERE naredbe koji se ne koriste od strane izraza uglavnom tačni. Ukoliko je ova pretpostavka netačna, dolazi se do lošeg plana. Korišćenjem ovih funkcija olakšavamo planeru odabir najboljeg plana tako što nagoveštavamo rezultate izraza.

CREATE TABLE composer(cid INTEGER PRIMARY KEY, cname TEXT);
CREATE TABLE album(aid INTEGER PRIMARY KEY, aname TEXT);
CREATE TABLE track(tid INTEGER PRIMARY KEY, cid INTEGER REFERENCES composer, aid INTEGER REFERENCES album, title TEXT);
CREATE INDEX track_i1 ON track(cid);CREATE INDEX track_i2 ON track(aid);
SELECT DISTINCT aname FROM album, composer, track WHERE cname LIKE '%bach%' AND composer.cid=track.cid AND album.aid=track.aid;
Zato što SQLite bira kako će najoptimalnije povezati tabele, zbog svojstva da pretpostavlja da je izraz *cname LIKE %bach%* tačan, iako to nije tačno, izvršava se loš odabir ugnježdavanja. Bira se track-composer-album

SELECT DISTINCT aname FROM album, composer, track WHERE likelihood(cname LIKE '%bach%', 0.05) AND composer.cid=track.cid AND album.aid=track.aid;
Bira se najoptimalniji spoj, composer-track-album

- **Koristite CROSS-JOIN sintaksu da osigurate redosled ugnježdenih petlji prilikom spoja kod upita koji mogu koristiti indekse niskog kvaliteta bez odgovarajuće statistike.**
- **Koristite unarni operator + kako bi odbacili indeks koji se koristi u WHERE naredbi.** Ukoliko postoji bolji indeks, moguće je diskvalifikovati loš indeks pomoću operatora +. Izbegavati ovu naredbu jer se može izvršiti konvertovanje tipova što dovodi do lošeg rešenja.

- **INDEXED BY naredba.** Ručno biramo indeks koji će biti korišćen

6. Reference

1. Uvod u sqlite https://www.sqlite.org/about.html
2. SQL obrada upita https://www.geeksforgeeks.org/sql-query-processing/
3. Indeksi https://www.sqlitetutorial.net/sqlite-index/
4. Optimizator upita https://www.sqlite.org/optoverview.html
5. NGQP https://www.sqlite.org/queryplanner-ng.html

7.Zaključak

U ovom radu smo se upoznali sa radom SQLite baze i načinom na koji ova baza obradjuje i optimizuje upite. Detaljno su opisane sve optimizacije koje baza interno vrši, kao i način rada planera upita koji predstavlja najbitniju komponentu SQLite. Objasnili smo sve prednosti i nedostatke osnovnog planera, kao i planera nove generacije NGQP. Iako optimizacija može u mnogome ubrzati rad aplikacije, potrebno je voditi računa o tome u kojoj fazi zapravo vršimo optimizaciju, iz razloga što ukoliko se u početku fokusiramo na optimizaciju to može biti kontraproduktivno. Optimizacija se uglavnom treba prepustiti planeru, a zadatak programera je da kreira što efikasnije indekse koji će planeru olakšati odabir pravog algoritma. Identifikovanje loših upita se vrši pomoću komande EXPLAIN QUERY PLAN, koja nam može reći na koji način se upit zapravo izvršava i da li koristi odgovarajući indeks ukoliko postoji. Koristeći funkcije likelyhood() i unlikely() možemo u mnogome olakšati odabir pravog algoritma planera, pošto u nekim slučajevima odabir algoritama može zavisiti od strukture podataka koji je poznat programeru, ali nije planeru.