Bisoños Usuarios de GNU/Linux de Mallorca y Alrededores | Bergantells Usuaris de GNU/Linux de Mallorca i Afegitons

Optimització de BB.DD. PostgreSQL (6835 lectures)

Per **Joan Miquel**, ***Joanmi*** (http://www.mallorcaweb.net/joanmiquel)

Creado el 25/09/2008 00:20 modificado el 25/09/2008 00:20

En aquest article explic, a partir de la meva experiència, de quina manera podem optimitzar el rendiment d'una base de dades PostgreSQL.

...tant des del punt de vista de configuració, com de la implementació de la pròpia base de dades (dimensionament, decisió dels índexos, composició dels querys...).

Índex

Introducció

Avui en dia, extrany és el sistema informàtic que no es recolzi en una base de dades relacional per a l'emmagatzemament i accés a la informació. PostgreSQL, MySQL, Firebird, Interbase... son noms sobradament coneguts de sistemes gestors de bases de dades relacionals.

Aquests sistemes son prou potents per fer que, amb la potència actual dels ordinadors, no calguin gaires esforços per aconseguir un rendiment acceptable o bò per volums petits o mitjans de dades.

Això fa que l'optimització de la base de dades sigui, habitualment, el gran oblidat. El gran oblidat... fins que et trobes en la necessit de gestionar un elevat volum de dades i que el rendiment no s'et caigui per terra.

Aquest article és un recull, basat en la meva experiència, d'alguns dels aspectes que cal tenir en compte per tal de que no s'ens caigu per terra el rendiment en una base de dades de dimensions considerables implementada amb PostgreSQL

No cal dir que això no és un manual exhaustiu ni pretén substituïr la excel·lent documentació oficial de PostgreSQL[6] sino, en tot c complementar-la posant èmfasi en els petits detalls que de vegades passam per alt i els errors més típics que, per bé puguin semblar òbvis quan s'expliquen, de vegades la inèrcia o la manca d'experiència ens poden induïr a cometre.

Per no fer-me avorrit, he intentat no extendre'm molt a cada punt ni aprofundir en detalls que, tanmateix, ja estan perfèctament explicats a la documentació de PostgreSQL. Tanmateix, ni es tracta de tasques complicades, ni sempre ens seràn necessàries o útils Dependrà en tot cas de les característiques de la nostra base de dades, quines accions poden o no ajudar-nos a millorar el seu rendiment.

Aquest document, per tant, no és més que una llista de verificació per ajudar-nos a trobar el (o els) mal(s) per després subsanar-los.

De tota manera, però, procuraré deixar els enllaços suficients perquè us sigui fàcil trobar la documentació relacionada en cas de ser necessari i, òbviament, si trobau res a faltar, podeu preguntar als comentaris ;-)

Aixímateix, pensant en els més profans en bases de dades, he intentat ser una mica exhaustiu i, per als més experimentats, pot ser q alguns fragments vos resultin més interessants que d'altres. Sentiu-vos lliures per botar-vos el que no vos interessi. Al final trobareu un petit resum dels punts més importants que pot ser vos serveixi també d'ajuda per localitzar els punts que vos puguin venir més d nou. I si teniu algun suggeriment per millorar-lo, serà benvingut ;-)

Configuració

Al capítol 18[7] del manual de Postgres, podem trobar tota la documentació relativa a la configuració del servidor. Però, en quant al que a l'optimització es refereix els punts més importants a tenir en compte son els següents:

Paràmetres de memòria

Al fitxer de configuració de Postgres (postgresql.conf, en Debian el trobam a /etc/postgresql/<versió>/main/postgresql.conf) hi ha alguns paràmetres relacionats amb l'ús de la memòria. Com sempre, podeu trobar la documentació completa[8] al manual de Postgr però els més significatius són els següents:

> **Nota:** Recordau que perquè qualsevol canvi sobre aquest fitxer tengui efecte, haureu de reiniciar el servidor de PostgreSQL.

max_connections =

No és pròpiament un paràmetre de memòria. Determina el número màxim de connexions concurrents que podrà atendre el servidor Però si anam curts de recursos ens pot ser d'utilitat reduïr aquest valor perquè això reduirà la quantitat de memòria compartida que servidor reservarà per les connexions amb els clients.

shared_buffers =

Determina la quantitat de memòria compartida que el servidor reservarà per sí mateix. Disposar de més memòria compartida pot suposar un millor rendiment en consultes complexes, però també pot minvar els recursos que el sistema operatíu té per altres funcio que també fan serveï a la base de dades. Per tant, haurem de jugar amb aquest valor per determinar quin valor ens és més avantatjós per a la nostra configuració hardware i les consultes que necessitam executar més típicament.

sort_mem = / work_mem =

> **Nota:** Aquest paràmetre pareix que ha canviat de nom. Al la versió 7.4 de Postgres es deia *sort_mem* i a la documentació de Postgres 8.3 (que estic fent servir de referència per no quedar-me obsolet tan aviat ;-)) surt com a *work_mem*, però la seva

> funció és la mateixa.

Defineix el tamany màxim de la finestra de memòria que Postgres reserva per a les operacions d'ordenació.

El valor d'aquest paràmetre és **molt important** quan treballam amb taules de gran volum donat que si la taula excedeix aquest tamany, en fer continuades lectures seqüencials (cas més desfavorable) les fallades de paginació seríen contínues donat que les files el cau serien constantment sobreescrites abans de tornar a ser llegides (moment en que caldria tornar a carregar-les).

Això vol dir que si fem moltes lectures, especialment si son seqüencials, sobre una taula determinada i aquesta creix més enllà del cab dins aquesta finestra de memòria, el rendiment d'aquestes consultes caurà en picat debut a les contínues fallades de pàgina. Incrementant el valor d'aquest paràmetre pot ser la sol·lució al nostre problema.

effective_cache_size

La documentació de Postgres 7.4 explica que té a veure amb la quantitat de memòria que el servidor "suposa" dedica el kernel al ca de disc. A la documentació de Postgres 8.3, dirèctament fan la següent recomanació (que m'ha semblat interessant reproduïr aquí):

> *effective_cache_size = 50%-75% of physical RAM (unless you're using 8.1 or older, in which case 75%-100% of physical RAM)*

Ubicació física de les taules

Un dels punts crítics a l'hora d'accel·lerar les consultes, és la càrrega de les dades des del disc a la memòria cau.

Com ja hem vist abans, amb el paràmetre *sort_mem / work_mem* pot ser puguem evitar que les files siguin invalidades entre lecture seqüencials consecutives, però igualment les haurem de carregar un primer cop i, de fet molts més perquè no hi romandrà etername

Per desgràcia els discs durs son molt lents respecte a la velocitat de la RAM i el seu temps de resposta depèn molt de la ubicació fís de les dades.

El sistema operatiu, normalment, es sol encarregar de mantenir els fitxers desfragmentats per tal d'optimitzar les lectures seqüencia Però en el cas d'un join sobre dues taules, la base de dades reclamarà el contingut de no un, sino dos (o més) fitxers (les taules) de manera concurrent.

Això provocarà que, encara que el sistema operatiu procuri trobar la millor manera d'atendre ambdues peticions, serà inevitable qu capçal del disc perdi molt de temps posicionant-se damunt un i altre fitxer per no fer esperar cap dels dos processos un temps excessivament prolongat.

Per tant, si tenim dues taules (o fins i tot més) de tamany considerable que sapiguem que les haurem d'encreuar freqüentment, probablement ens resulti molt avantatjos ubicar-les en discs durs diferents de manera que, en ser accedides concurrentment es podr llegir molt més ràpidament en no competir una i altre pel mateix capçal de disc dur. No cal dir que no podem comprar un disc dur p cada taula que tenguem a la base de dades, però si tenim taules molt grosses que haguem d'encreuar freqüentment i necessitam un rendiment elevat, pot ser ens surti a compte separar les més importants.

La forma de fer això és molt senzilla. Només cal aturar el servidor i moure físicament el fitxer a la seva nova ubicació i, en el seu l deixar un enllaç simbòlic que hi apunti per tal de que el servidor el pugui obrir normalment accedint a la mateixa ruta.

De totes maneres això, com sempre, està explicat més extensament al corresponent capítol[9] del manual de Postgres.

Disseny

Idealment, el disseny d'una base de dades hauria de seguir un model més o menys marcat a partir dels requeriments:

- Estudi de les necessitats.
- Especificació de la sol·lució.
- Model Entitat-Relació
- Model Relacional
- Normalització...

I realment això és ni més ni pus el que hem de fer: realitzar un bon disseny de base de dades, normalitzat i sense redundàncies.

Un cop fet això, botau-vos la resta d'aquest apartat, passau dirèctament al capítol d'Indexs i continuau fins al final. Si després continuau tenint problemes o ja ara estau **MOLT segurs** de que els tendreu, tornau a aquest punt i continuau llegint a partir d'aquí. Però tingueu present que, ben emprada, PostgreSQL pot manetjar bases de dades amb taules de l'ordre de varis milions de registres encreuar-les múltiples vegades en consultes anidades amb temps de resposta gairebé instantanis. I parlo només pel que he pogut experimentar.

Dimensionament de taules

Taules auxiliars

Continuant amb el problema de la memòria cau d'ordenació, el tamany de les taules és un altre dels factors més importants a l'hora d'intentar no excedir el tamany d'aquesta finestra de memòria durant les operacions de recerca.

Per desgràcia però, aquest no és un paràmetre que usualment puguem triar nosaltres sino que, sovint, ens ve imposat per les necessitats del sistema i que, a més, té la emprenyosa mania de crèixer a mesura que passa el temps.

Però sí que és una situació que, de vegades, en el moment de dissenyar la base de dades podem preveure que ens succeirà amb una taula concreta. És la típica taula al voltant de la qual gira tota la resta de dades i que acostuma a tenir un munt de columnes que no podem obviar però que en realitat son necessàries només en casos molt particulars que es donaran relativament poques vegades.

En aquest cas, una sol·lució econòmica i efectiva sol ser dividir aquesta taula en dues (el que en el model Entitat-Relació seria una relació 1:0,1). És a dir: crear dues taules en comptes d'una de manera que en una tendrem la informació *principal* (clau primària, cl forànies i camps de dades més importants o més habitualment accedits) i la segona, la clau primària de la qual pot ser, a l'hora, una clau externa a la taula principal, la que ens servirà per emmagatzemar tota aquella informació accessòria que, a lo millor, no tots el registres necessiten tenir o en moltes consultes no ens interessarà llegir.

Fins i tot en el cas de que sempre necessitem aquestes dades, si sobre aquesta taula s'han de fer múltiples joins en una mateixa consulta (és a dir: encreuar distintes files de la mateixa taula) sempre que els camps *auxiliars* no formin part de la condició de join, ens pot resultar avantatjós tenir-ho separat pel simple fet que això ens reduïrà el consum de memòria a l'hora d'encreuar-les.

Aiximateix, aquesta no és una sol·lució precisament el·legant ni, des del meu punt de vista desitjable. I si ens trobam en la necessita d'haver-hi de recórrer, jo diria que molt probablement és que no hem fet un disseny de la base de dades tan bò com seria desitjable. Però tots sabem que de vegades, i moltes no per voluntat pròpia, ens trobam davant situacions *extranyes* que obliguen a prendre decisions si més no pecul·liars.

Redundància de dades

Una altra sol·lució per minvar l'elevat cost de determinats encreuaments de taules(sobretot quan aquests son múltiples) és, precisament, evitar-los.

En alguns casos, ens adonam que una columna que es troba en una taula determinada, ens vendria molt bé tenir-la *copiada* en una altra taula per evitar-nos haver de fer el join de les dues cada cop que necessitam aquesta dada.

No cal dir que la redundància de dades és el pitjor problema d'integritat que pot tenir el disseny d'una base de dades. Però també és problema que es pot sol·lucionar amb una mica de programació a nivell de base de dades.

El que hem de fer en aquest cas és crear un parell de triggers que s'ocupin de mantenir sincronitzada la columna redundada entre ambdúes taules be sigui fent que s'actualitzin mútuament o que una actualitzi l'altra i aquesta impedeixi la modificació.

El primer problema que ens trobarem amb això és que, si ho seguim al peu de la lletra fabricarem un fantàstic bucle infinit de trigg que es disparen l'un a l'altre contínuament. Però fent servir una mica el caparrot és fàcil caure en que no necessitam modificar un camp quan aquest ja conté el valor correcte. I, si no hi ha escriptura no es dispara cap trigger ;-)

Igualment pagarem també un altre preu que és el petit retard que causarà l'execució del trigger a les operacions d'inserció i actualització, però si ho hem fet bé, amb tota probabilitat aquest temps ens semblarà menyspreable i, fins i tot, es pot veure compen pel propi estalvi en temps de lectura si hi ha altres triggers es puguin veure beneficiats en simplificar-se algunes de les seves

comprovacions.

Com he dit abans, no és objecte d'aquest document aprofundir en excés en sol·lucions que, fet i fet, no sempre seràn el que el lector estigui cercant, així que m'abstendré de posar exemples que, tanmateix, no deixarien de ser casos particulars probablement molt diferents als que vos podrieu trobar vosaltres.

El que sí vos deixaré, òbviament, és l'enllaç a la secció de PL/PGSQL[10] del manual de Postgres i a la de triggers[11].

També cab apuntar que en Postgres no només es poden fer procediments emmagatzemats en PL/PGSQL o C, sino que també dispo de connectors[12] per altres llenguatges com Tcl[13], Perl[14], Python[15] i altres[16]...

...però igualment, com a l'apartat anterior, abans de prendre la decisió de redundar dades, jo vos recomanaria que acabassiu de llegi aquest article. Pot ser aleshores ja no vos faci falta.

Indexació

Els índexs[17] li permeten a la base de dades efectuar recerques sense necessitat d'haver de recórrer tota la taula. Existeixen diferent tipus d'indexs[18] però els més comuns (i el tipus per defecte si no n'especificam un altre) son els *btree* (arbres binaris[19]).

Escollir els índexs adequats

Un índex accel·lera la recerca de fileres en les que el camp (o camps) indexat(s) compleixen una determinada condició. Si be això n és ni màgic ni gratuït: Crear i, sobretot, mantenir un índex té un cost.

El cost de crear un índex esdevé menyspreable des de l'instant en que l'índex ens sigui d'alguna utilitat (només es fà un cop i la tasc d'indexació no carrega excessivament la base de dades).

El cost de mantenir-lo consisteix en que, cada cop que s'inserti, actualitzi o elimini alguna fila de la taula que conté l'índex, aquest haurà de ser actualitzat per tal d'incorporar / modificar / eliminar les entrades dels elements afectats. Aquest cost depèn del tipus d'índex[18] del que es tracti i, si be sol ser relativament baix en comparació amb el benefici que reporten, mai és nul.

És per això que, excepte en els casos en que son estrictament necessaris (claus primàries i úniques), la base de dades mai crearà un índex automàticament, recaiguent així en nosaltres la responsabilitat de crear els que ens siguin necessaris. Si no fos així, es crearie molts d'índexs que, en realitat, mai es farien servir.

Per tant, a l'hora de decidir si cream o no un índex sobre una columna determinada, quasi la única pregunta que ens hem de fer és s farem servir. Rarament ens interessaràn altres consideracions excepte que, per exemple, necessitem una alta velocitat d'escriptura i compensi sacrificar el rendiment per lectura (recordem que estam suposant que l'índex sí es faria servir).

Un cop hem decidit que ens interessa crear un índex, només ens queda assegurar-nos que realment es fa servir i això no és tan òbvi com pugui semblar en principi. Per això us recomano que llegiu el capítol de construcció d'«Anàlisi i Construcció de consultes» me avall.

No tot és automàtic

Com hem comentat abans, quan cream una clau forània, la base de dades ens exigirà que el camp al que apuntam sigui una clau úni (la clau primària, que és l'opció típica, és un cas de clau única) i, per tant, estarà indexat. És fàcil caure en l'error de pensar que el fe de declarar una clau forània en una taula, impliqui que es crei un índex sobre aqueixa automàticament. Però **això no és cert**. Una cl forània ens permet assegurar que existeix un (i només un) element d'una altra taula en el qual, el valor d'una columna determinada concordi en el valor emmagatzemat. Dit d'una altra manera: una clau forània és un punter a una fila d'una altra taula.

...però aixo **no** significa que existeixi cap índex sobre la clau forània que ens permeti agilitzar les recerques sobre aquesta columna. canvi, és bastant típic (però no necessari) que les vulguem fer. Per exemple, en una taula de llibres, podríem tenir una clau forània relacioni cada llibre amb el seu autor. No hi ha cap necessitat de crear un índex sobre aquesta columna perquè la clau forània funci correctament. En canvi, si és previsible que amb certa freqüència vulguem treure un llistat dels llibres d'un determinat autor, indexa aquesta columna ens pot accel·lerar considerablement aquesta consulta.

En resum, sempre que creem una clau forània, és bona idea plantejar-nos si ens interessa o no indexar-la perquè la base de dades no

ho farà automàticament i és altament probable que ens pugui interessar fer consultes filtrant pels valors d'aquest camp.

Els índexs no son màgics

De vegades tendim a pensar en els índexs de base de dades com en l'índex d'un llibre en el que el lector utilitza l'índex per no haver llegir tot el llibre *seqüencialment* per trobar una informació concreta.

Doncs bé. Aquesta associació realment és aproximada, però no exacta: En el cas de l'índex d'un llibre, qui consulta l'índex és una persona que té no només la capacitat, sino el vici, de pensar i que, ni que sigui per curiositat, probablement llegirà tot l'índex fins q trobi el que cerca (per tant no llegeix seqüencialment tot el llibre, però sí bona part de l'índex). El motor d'una base de dades, en ca **no pensa**: senzillament segueix un procés automatitzat que qui el va programar es va preocupar de que fes el que ha de fer de la manera més ràpida possible (i, en això, es pareix més a un *corol·lari* que ens serveix per recordar coses molt òbvies, però sense hav d'aturar-nos a pensar-hi). I si ho ha de fer de la manera més ràpida possible, no es pot entretenir en fer comprovacions, d'entrada innecessàries que, en canvi, els humans no podriem evitar fer fins i tot de manera inconscient quan examinam l'índex d'un llibre.

Per exemple, si recordam que vam llegir quelcom que ens interessa a l'apartat 3 de no sabem quin capítol del llibre, podem fer serv l'índex general del llibre (amb capítols i apartats) i fixar-nos en tots els apartats '3' que hi hagi. En canvi però, si tenim un índex sob les columnes *a* i *b* de la taula *X*, d'entrada **no** podem fer servir aquest índex per fer una recerca per només per la col·lumna *b* perquè que està indexat és la combinació de les dues.

i(a,b) !=> i(a) ni i(b)

Si pensam en com funcionen els índexos de les bases de dades, això és bastant òbvi, perquè, segons el tipus d'índex i la seva implementació, aquest reconeixement que fem de *la part corresponent a l'apartat* podria suposar un processament que ralentiria innecessàriament la recerca en el cas pel que realment ha estat definit l'índex. Però si pensam com en els índexos dels llibres ens po semblar fins i tot lògic que si tenim un índex que engloba dos camps podem fer una recerca sobre un d'ells i "*com que hi ha l'índex, consulta serà eficient*".

De fet, això no és totalment incert (però quasibé): Segons la documentació[20] hi ha casos **molt específics** en que una condició que impliqui un subconjunt de les columnes indexades podria optimitzar-se fent servir l'índex. Però això depèn de quin tipus d'índex es tracti i, així i tot, no seria tan eficient com tenir un índex (més petit) de només les columnes implicades. Per exemple, per una cond que requereixi una igualtat de la primera (o primeres *n*) columnes d'un índex btree, es podrà fer servir l'índex per agilitzar la consul Però si s'ha de complir una desigualtat (com ara *nom_columna* > *7*), a partir d'aquesta, la resta de columnes indexades s'haurà d'escanejar seqüencialment. Com a regla (inexacta) pnemotècnica*[21], podriem dir que:

btree(a,b) => btree(a) però **no sempre** => btree(b)

Però recordau que un índex específic sobre la columna que ens interesi sempre serà més eficient. No obstant, si podem fer servir pa d'un índex existent, en determinats casos podrem estalviar-nos crear-ne un d'específic. Però així i tot és molt recomanable que revis (amb la sentència *explain* la planificació dels querys que voleu fer per comprovar si realment el planificador troba la manera d'obte un benefici d'aquell índex. Pot ser que, senzillament canviant l'ordre en que s'indexen les columnes, obtingueu un major profit de l'índex. O pot ser descobriu que, en general, vos surti més rentable crear dos índexs independents.

Per això és molt millor pensar que, **d'entrada, un índex només ens servirà per fer recerques sobre una combinació completa d les columnes que el conformen** i que després, en funció del tipus d'índex de que es tracti, es podràn fer més o menys filigranes per sempre consultant primer la documentació si no coneixem prèviament què pot fer i que no el tipus d'índex en qüestió o, com a míni revisant la planificació del query per asegurar-nos que realment en treu el profit que nosaltres esperam.

Si no ho tenim en compte, és molt difícil que ens n'adonem a simple vista perquè la base de dades no es queixarà quan li facem la consulta: Senzillament veurà que no té l'índex que necessita i farà una recerca seqüencial cada vegada que li facem la mateixa petic i, mentres la base de dades sigui petita tot funcionarà de meravella, però a mesura que vagi creixent el volum de dades es pot conve en un mal son si no ens adonam que no s'està fent servir l'índex.

Disparadors (triggers)

La regla d'or: Minimalisme.

Lo bò, si breu: dues vegades bò.

Els disparadors[22] (o *triggers*) ens permeten fer que s'executi un procediment emmagatzemat[23] abans o després d'una operació d'inserció, modificació o esborrat sobre una fila (els més comuns) o sobre una taula sencera.

Això ens atorga, a més de la integritat relacional pròpia de qualsevol SGBD, la capacitat d'implementar la *integritat semàntica* de l dades. Per exemple, en una taula de notes d'exàmen, podríem garantir que la nota estigués sempre entre 0 i 10. D'aquesta manera garantim que el que ens diguin les nostres dades, en tot moment tengui sentit i no hi ha possibilitat que cap error en una aplicació client (o en qualsevol operació manual) ens ho desbarati.

Però, com tot, això té un preu. Els procediments cridats per un disparador s'executen un cop per a cada fila que s'inserta, modifica i elimina i, per tant, el seu temps d'execució, es sumarà inevitablement al temps total de l'operació. Per això, és aconsellable:

- Procurar que facin just el que han de fer i res pus, en el menor temps possible.
- Si existeixen condicions que poden fer decidir avortar o finalitzar prematurament la operació, es processaràn en primer llo sempre que sigui possible.
- D'aquestes condicions, donarem preferència sempre que poguem, a les que estadísticament es puguin donar amb més freqüència.

Per exemple, normalment no comprovarem en un disparador, si la fila que insertam o actualitzam incompleix alguna restricció (constraint) de la BD, com ara claus foràanies o clàusules *unique*, etc... perquè això ja ho farà la pròpia BD en el moment de materialitzar la operació.

Però si al disparador d'inserció garantim que la nota està entre 0 i 10, al d'actualització només cal que ho verifiquem si la nota ha canviat. Òbviament aquest és un mal exemple perquè el cost de comparar els valors antic i nou de la nota es pràcticament igual al d verificar si està entre 0 i 10, però ens serveix per veure que, si les verificacions son més complexes, ens les podem estalviar en els casos que les columnes afectades no es modifiquin.

Recursivitat i triggers *before...*

Tampoc cal passar-se amb el minimalisme.

Per exemple, hem dit abans que normalment no farem comprovacions que tanmateix la base de dades ja fa per si mateixa, com ara verificar que el valor d'una clau foràania realment existeix a la taula filla. Però tota bona regla ha de mester una excepció que la confirmi:

Una de les meravelles que pot fer un disparador que s'executa **abans** d'efectuar-se la operació, és modificar la operació en sí matei És a dir: poden modificar les dades a insertar/actualitzar abans de que es realitzi la operació i, per tant, corregir errors en aquestes abans de que es produeixin. Per això, les comprovacions d'integritat que fa la base de dades per ella mateixa, només es poden fer en moment en que, efectivament, es produeix l'execució de la operació.

Per tant, si el procediment que executa aquest disparador és relativament complex i existeix una probabilitat important de que s'executin operacions amb problemes d'integritat, pot ser ens valgui la pena detectar aquesta situació i forçar *manualment* (RAISE EXCEPTION[24]) l'error per tal d'evitar que es tudi més temps en una operació que, tanmateix, no pot arribar a port.

Això darrer és especialment important si implementam disparadors recursius (disparadors que, en determinades ocasions, executen mateixa operació sobre la mateixa taula i, per tant, es disparen a si mateixos de manera --si estan ben pensats-- controlada). En un c així, la operació inicial només s'executarà un cop hagin finalitzat totes les operacions que s'han disparat en cadena i, per tant, si hi h un error d'integritat, només es detectarà un cop finalitzat el procés. A més, en aquest cas, el procés de tornar enrera (rollback) tamb serà més costós pel número d'operacions ja concloses que s'hauràn de fer enrera per desfer la transacció completa.

Manteniment

PostgreSQL no necessita gairebé gens de manteniment. De fet, en una base de dades de dimensions no exageradament grosses, poc viure feliços per la resta dels nostres dies sense preocupar-nos per res de fer-hi cap tipus de manteniment (i, de fet, això és el que succeeïx en la majoria dels casos).

Però si parlam de bases de dades de dimensions considerables o sobre la que s'hagin d'executar consultes molt complexes, pot ser s que hi hagi una o dues operacions de manteniment que ens interessi fer amb certa periodicitat.

Aquestes opreacions no son gens complexes d'efectuar i es poden fer perfèctament mentres la base de dades està treballant i, normalment, els usuaris ni ho notaran. Però així i tot, sempre és preferible realitzar-les en un moment en que no estiguem tenguent una punta de feina.

Per això, normalment aquestes operacions no es realitzen automàticament i és tasca de l'administrador executar-les quan ho cregui oportú o (el més recomanable a la majoria dels casos) programar-les al Cron amb la periodicitat i a la franja horària que estimi òptimes.

Una altra opció també, pot ser realitzar-les a nivell de taula, amb diferentes periodicitats segons les necessitats de la taula i procura que no coincideixin mai entre elles, alleujant així la càrrega que suposa la seva execució i obtenint un resultat igualment òptim.

En qualsevol cas, en versions anteriors a PostgreSQL 8.1 (o posteriors si es desactiva l'auto-vacuum) és recomanable fer un vacuu que sigui un cop a l'any o, en qualsevol cas, abans de cada 4 bilions de transaccions[25] ;-)

Vacuum

Quan s'esborra una fila d'una taula, per questions de rendiment, la fila no s'esborra realment del fitxer que conté la taula, sino que senzillament es desindexa dels índexs que l'afectin i es marca com a esborrada. Les noves insercions tampoc faràn servir aquest esp sino que s'afegiran sempre al final del fitxer.

Això fa que les operacions d'inserció i esborrat siguin molt més ràpides del que ho serien si haguessin d'anar moguent dades d'aquí cap allà per ajustar el tamany de cada fila amb la següent. Però també que si tenim un volum important d'operacions d'esborrat, l'es utilitzat al disc creixi molt més del realment necessari per les dades. A més, la major dispersió de les dades dins la taula, tampoc afavoreix el temps de resposta en les consultes.

La operació vacuum[26] compacta els fitxers de les taules eliminant definitivament les fileres esborrades mantenint, lògicament, la correl·lació dels índexs.

A partir de la versió 8.1 de Postgres, la pròpia base de dades es preocupa[27], per defecte, de fer els vacuums quan ho estima necess Tot i que, si ens interessa, podem deshabilitar aquesta funcionalitat si preferim encarregar-nos nosaltres de programar els vacuums com més ens interessi.

Per exemple, passant-li com a paràmetre el nom d'una o més taules, podem fer vacuum exclusivament d'aquelles taules evitant així perdre temps amb altres taules que sapiguem tenguin un índex de fragmentació molt més baix.

Analyze

A l'hora de planificar un query, Postgres té en compte molts de factors, com ara els índexs de que disposa, el tamany de la taula, i altres dades estadístiques sobre les dades emmagatzemades a la taula.

Obtenir tota aquesta informació a cada query, costaria molt més que l'avantatge que ens reporta, per això no es fa automàticament, sino que es calcula un primer cop quan cream la taula i prou.

A mesura que la taula va cresquent i/o es van modificant les dades, aquesta informació esdevé desactualitzada i pot fer que les decisions que prengui el planificador a l'hora de *pensar* els querys, siguin més dolentes.

Executant la comanda analyze[28], fem que Postgres recalculi tota aquesta informació estadística aconseguint així que l'analitzador molt millor la seva feina. Amb una base de dades petita, pot ser la diferencia no sigui molt grossa, però quan es comencen a tenir taules d'uns pocs milions de registres, la diferència pot estar entre varis segons i unes poques centèssimes d'execució del mateix qu

Anàlisi i construcció de les consultes

Planificació

És bastant típic comprovar si un element ja existeix en una taula abans d'insertar-lo. Exemple (pseudocodi):

```
dni = db->query("select dni from persones where dni = '123456789x'");
if (dni = null) {
```

```
        db->query("insert into persones (dni, nom) values ('123456789x', 'manolo');
} else {
        db->query("update persones set nom = 'manolo' where dni = '123456789x');
};
```

...però aquest exemple és optimitzable: Si la columna *dni* és clau única, la comprovació que fem per comprovar si la fila existeix és redundant, perquè ja la farà la pròpia base de dades quan intentem fer la inserció o actualització.

És més eficient intentar dirèctament una de les dues opcions i, si ens dona error, provar l'altra.

En concret, si estam implementant un procés d'importació inicial, el més lògic és fer primer la inserció i, si falla, l'actualització (suposant que acceptem duplicitats a la font i les darreres aparicions es considerin actualitzacions de les primeres. En altra cas, òbviament, l'actualització no tendria sentit i s'hauria de reportar l'error).

En canvi, en un procés d'actualització, serà més eficient actualitzar primer i, si ens dona error (nou registre), aleshores fer la inserci

De totes maneres, també cal tenir en compte si els processos d'inserció i/o actualització disparen procediments emmagatzemats que puguin efectuar altres tasques més costoses. En aquest cas, ens podria ser més òptim fer la consulta primer. Si bé això igualment es pot resoldre fent la mateixa comprovació al principi del procediment emmagatzemat. No ens incrementarà el rendiment en aquest c però sí, com hem explicat a l'apartat de disparadors (triggers) en el cas d'errors no volguts (es detectaràn abans).

Explain

La sentència *explain* ens mostra, desglosada, la manera en que postgres planificaria una determinada consulta i el cost aproximat qu el planificador preveu que tindrà.

Això, per una banda, en serveix per comparar i avaluar les diferentes possibilitats que tenim a l'hora d'implementar un query per a obtenir una informació determinada i així poder triar la opció que resulti més eficient.

Per altra banda, si tenim un quey molt pesat que volem optimitzar, com veurem més envant, ens donarà informació molt útil per determinar quines son les parts del query que li resulten més pesades i que més ens convé esforçar-nos en millorar.

No tot és el que pareix...

Una de les grans virtuds de Postgres, és la seva capacitat de comparar dades de tipus diferents d'una manera molt senzilla, intuitiva sobretot, que habitualment fa el que l'usuari espera.

...Habitualment, però no sempre. I de vegades ens pot fer males passades. Analitzem si no el següent exemple:

```
test=# create table prova (
test(# id bigserial primary key,
test(# text varchar
test(# );
NOTICE:  CREATE TABLE will create implicit sequence "prova_id_seq" for "serial" column "prova.id"
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "prova_pkey" for table "prova"
CREATE TABLE
test=# explain select * from prova where id = 100;
                       QUERY PLAN
-------------------------------------------------------
 Seq Scan on prova  (cost=0.00..22.50 rows=2 width=40)
   Filter: (id = 100)
(2 rows)
```

Aquí hem creat una taula *prova* amb un camp *id* tipus *biginteger autonumèric (bigserial) com a clau primària i un camp de text adicional.*

Veiem també que, en crear la taula, Postgres ens diu que implícitament ha creat un índex per la columna *id* perquè així es fà sistemàticament amb totes les claus primàries, sense que calgui crear-lo explícitament.

En canvi, quan provam de fer un explain d'un query senzill, com mostrar la fila amb *id* = 100, veim que ens diu que per trobar-lo fa una recerca seqüencial (Seq Scan) en comptes de fer servir l'índex (Index Scan). És cert que el cost no és molt elevat, però això és perquè en aquests moments la taula està buïda. Però en qualsevol cas, si es fes servir l'índex, el cost encara seria molt menor.

Perquè, doncs, no s'està fent servir l'índex?

Com hem dit al principi de l'apartat, Postgres ens permet comparar variables de tipus diferents sense demanar-li explícitament que faci la conversió*. Però això no vol dir que es puguin sumar peres amb melicotons. 100::bigint no és el mateix que 100::integer i, encara que existeixi una funció que estableix la (per nosaltres òbvia) equivalència entre tots dos valors, per una màquina no és tan òbvia una equivalència entre dos objectes de tipus diferents (no hi entèn de semàntica), pel que, per saber quines files coincideixen veu obligat a fer la comparació **per a cada fila** (Seq Scan)

Què li passava al query anterior? Doncs que, quan escrivim un valor numèric en un query, nosaltres pensam en ell com "un número" però per a Postgres és un *tipus numèric* **concret**. I, si nosaltres no li especificam quin, ell agafa el tipus numèric per defecte: **intege** Vegem que passa ara si forçam a Postgres perquè interpreti la constant del query com un valor de tipus biginteger:

```
test=# explain select * from prova where id = 100::bigint;
                              QUERY PLAN
-----------------------------------------------------------------------
 Index Scan using prova_pkey on prova  (cost=0.00..4.82 rows=2 width=40)
   Index Cond: (id = 100::bigint)
(2 rows)
```

El mateix ens pot passar amb tipus més *petits* com ara *smallint*. Amb un *smallint* només podem tenir fins a 2^{1} = 65536 valors diferents, però no tots els índexs son únics, un mateix valor es pot repetir moltes vegades i, el més important, encara que la taula fo petita, podem necessitar construïr joins molt complexos que multipliquin aquest enlentiment fins a magnituts insuportables. Un pet retard repetit moltes vegades és un gran retard.

Vegem un exemple i recordau que, per simplificar, estam provant amb una taula buïda. A mesura que la taula es vagi omplint, aque diferència s'incrementarà linialment i, quan sigui prou grossa, exponencialment si no ho podem resoldre ajustant els tamanys de les finestres de memòria, com s'ha explicat a la primera part d'aquest document:

```
test=# alter table prova add column nota smallint;
ALTER TABLE
test=# create index prova_nota on prova (nota);
CREATE INDEX
test=# explain select * from prova where nota = 10;
                      QUERY PLAN
-------------------------------------------------------
 Seq Scan on prova  (cost=0.00..22.50 rows=6 width=42)
   Filter: (nota = 10)
(2 rows)

test=# explain select * from prova where nota = 10::smallint;
                               QUERY PLAN
------------------------------------------------------------------------
 Index Scan using prova_nota on prova  (cost=0.00..17.07 rows=6 width=42)
   Index Cond: (nota = 10::smallint)
(2 rows)
```

(*) Sempre i quant aquesta conversió sigui òbvia. Exemple:

```
test=# select '34' + 5;
 ?column?
----------
       39
(1 row)

test=# select '34.3' + 5;
ERROR:  invalid input syntax for integer: "34.3"
```

...en el segón cas és necessari fer servir la funció to_number()[29] per indicar-li a Postgres com ha d'interpretar el punt que conté la cadena (podría ser un punt decimal, separador de mil·lers, etc...).

Truc: Estalviar-nos un subquery

Suposem que tenim una taula semblant a la següent i volem obtenir el valor de la columna *text* quan *numero* és màxim:

```
create table llista (
```

```
   id serial primary key,
   text varchar,
   numero integer,
   unique (text, numero)
);
```

Algú intentarà fer alguna cosa com "*select text from llista where numero = max(numero)*", però aviat se n'adonarà que això no és possible perquè no es poden fer servir funcions agregades a la condició d'un query. Per tant, necessitarem un subquery per obtenir el valor màxim de *numero*:

```
select text from llista where numero = (select max (numero) from llista);
```

Una altra estratègia, pot ser evitar el subquery ordenant el resultat i agafant només la primera fila:

```
select text from llista order by numero desc limit 1;
```

Aquesta estratègia, en principi, podria resultar avantajosa des del punt de vista de consum de memòria si, en comptes de la taula *llis* pensam en un resultat parcial procedent d'un subquery més complex (és a dir: que ja el tenguessim en memòria i, per treure el valor màxim en un subquery l'haguessim de repetir). Però per saber de debò quina de les dues versions resultaria més eficient, lo millor é preguntar-ho a qui més en sap:

```
test=# explain select text from llista where numero = (select max (numero) from llista);
                             QUERY PLAN
---------------------------------------------------------------------
 Seq Scan on llista  (cost=22.50..45.00 rows=6 width=32)
   Filter: (numero = $0)
   InitPlan
     ->  Aggregate  (cost=22.50..22.50 rows=1 width=4)
           ->  Seq Scan on llista  (cost=0.00..20.00 rows=1000 width=4)
(5 rows)

test=# explain select text from llista order by numero desc limit 1;
                            QUERY PLAN
------------------------------------------------------------------
 Limit  (cost=69.83..69.83 rows=1 width=36)
   ->  Sort  (cost=69.83..72.33 rows=1000 width=36)
         Sort Key: numero
         ->  Seq Scan on llista  (cost=0.00..20.00 rows=1000 width=36)
(4 rows)
```

Com veim, en el primer cas, recórrer la taula per trobar el valor màxim de numero, té un cost important (gairebé el 100% del cost total), però el procés d'ordenació de tota la taula per després agafar només la primera fila, ha tengut un cost superior.

Vegem ara, però, que hagués passat si la columna *numero* hagués estat indexada:

```
test=# create index llista_numero on llista (numero);
CREATE INDEX
test=# explain select text from llista where numero = (select max (numero) from llista);
                               QUERY PLAN
-------------------------------------------------------------------------
 Index Scan using llista_numero on llista  (cost=22.50..39.57 rows=6 width=32)
   Index Cond: (numero = $0)
   InitPlan
     ->  Aggregate  (cost=22.50..22.50 rows=1 width=4)
           ->  Seq Scan on llista  (cost=0.00..20.00 rows=1000 width=4)
(5 rows)

test=# explain select text from llista order by numero desc limit 1;
                                      QUERY PLAN
---------------------------------------------------------------------------------------
 Limit  (cost=0.00..0.05 rows=1 width=36)
   ->  Index Scan Backward using llista_numero on llista  (cost=0.00..52.00 rows=1000 width=36)
(2 rows)
```

Com veim, per be que es faci servir l'índex que hem creat, en el primer cas, el cost de trobar el valor màxim de *numero* és tècnicam el mateix. En canvi, en el segon, la ordenació és gairebé instantània i el fet de només haver de llegir el contingut de la primera fila f que la diferència sigui fulminant.

Evidentment, aquestes diferències depenen també del volum de dades que contengui la taula. Aquests exemples han estat fets amb taula completament buïda, però ens donen una idea molt realista de quina és la millor estratègia a l'hora de plantejar la consulta (i a

l'hora de triar els indexs).

Per altra banda, aquest era un exemple pensat específicament per mostrar com a l'hora de determinar quina és la millor estratègia p construïr el query, cal també tenir en compte quins camps tenim indexats i quins no. La constraint "*unique (text, numero)* ens crea automàticament un índex btree sobre aquests dos camps (i en el mateix ordre). Si haguessim invertit l'ordre en que els especificave la primera columna de l'índex hagués estat *numero* i, per tant, s'hagues pogut fer servir per optimitzar aquesta consulta. El resultat hauria estat el següent:

```
test=# explain select text from llista order by numero desc limit 1;
                                          QUERY PLAN
-----------------------------------------------------------------------------------------------
 Limit  (cost=0.00..0.05 rows=1 width=36)
   ->  Index Scan Backward using llista_numero_key on llista  (cost=0.00..52.00 rows=1000 width=36)
(2 rows)
```

No comptis si no vols saber quants n'hi ha

És una situació molt típica, voler saber si existeixen files en una taula que compleixin una determinada condició. Per exemple, si du un registre d'activitat dels usuaris d'una aplicació, és bastant normal que vulguem saber si un usuari concret ha tengut mai activitat quins usuaris no s'han fet servir mai.

La sol·lució típica ("*de manual*") en aquests casos, és comptar les files que compleixen la condició i verificar si el resultat és 0:

```
select count(*) = 0 from user_activity where uid = 73;
```

Però això, en realitat és un desgavell com una urbanització sencera. Seria equivalent a que ens demanassin un llistat d'illes desertes món i, per saber si Mallorca està o no deshabitada anassim casa per casa comptant la gent que hi habita i, un cop comptabilitzats el 814.275 habitants[30], pensàssim: "(814.275 != 0) => Mallorca està habitada".

Amb una màquina moderna i una base de dades d'uns pocs centenars de mil·lers de registres podem fer servir aquest mètode i pot s ni tan sols ens n'adonem. Però a mesura que el nostre sistema creixi i, sobretot, si aquesta informació es consulta amb freqüència i/ com a part de consultes iteratives més complexes, pot fàcilment explicar perquè moltes aplicacions que als seus inicis funcionaven meravella, acaben esdevenint inusables quan el volum de dades creix.

Però òbviament a ningú li passaria pel cap comptar tots els habitants de Mallorca per saber si està o no habitada: Al primer individu que véssim passar pel carrer ja diriem: "hi ha algú, per tant Mallorca no està deserta".

Anem a veure ara, com podríem materialitzar aquest mateix mètode en SQL suposant tres possibles escenaris diferents:

1. Necessitam senzillament conèixer aquesta informació com a dada dins el nostre programa.
2. Necessitam aquesta informació com a dada dins un procediment emmagatzemat en plpgsql.
3. Necessitam aquesta informació com a valor booleà per avaluar-lo dins una consulta SQL més complexa.

El primer cas no pot ser més evident: Cercant únicament el primer elment que aparegui i, en comptes de fixar-nos en els valors de l fila retornada, fixar-nos senzillament en si ens ha retornat o no alguna fila. Vegem doncs, la diferència de rendiment respecte del count(*) = 0 inicial:

```
test=# explain select count(*) = 0 from user_activity where uid = 73;
                                       QUERY PLAN
-----------------------------------------------------------------------------------------
 Aggregate  (cost=17.08..17.09 rows=1 width=0)
   ->  Index Scan using user_activity_uid on user_activity  (cost=0.00..17.07 rows=6 width=0)
         Index Cond: (uid = 73)
(3 rows)

test=# explain select * from user_activity where uid = 73 limit 1;
                                       QUERY PLAN
-----------------------------------------------------------------------------------------
 Limit  (cost=0.00..2.84 rows=1 width=52)
   ->  Index Scan using user_activity_uid on user_activity  (cost=0.00..17.07 rows=6 width=52)
         Index Cond: (uid = 73)
(3 rows)
```

En canvi, si necessitam aquesta informació per tractar-la dins un procediment emmagatzemat en plpgsql (en altres llenguatges més genèrics, probablement podrem fer servir la tècnica anterior), no he trobat (que no vol dir que no existeixi) una manera de saber dirèctament si un query torna o no cap fila (sense fer un count(*), òbviament).

Però així i tot, ens les podem enginyar per obtenir aquesta informació sense comptar tots els elements que compleixin la condició. Una primera aproximació seria amb un subselect: comptant les files del resultat del query anterior. No ens evita comptar, però com màxim comptarem un únic element. Vegem la diferència de rendiment:

```
test=# explain select count(*) = 0 from user_activity where uid = 73;
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Aggregate  (cost=17.08..17.09 rows=1 width=0)
   ->  Index Scan using user_activity_uid on user_activity  (cost=0.00..17.07 rows=6 width=0)
         Index Cond: (uid = 73)
(3 rows)

test=# explain select count(*) = 0 from (select * from user_activity where uid = 73 limit 1) as foo;
                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=2.86..2.86 rows=1 width=0)
   ->  Subquery Scan foo  (cost=0.00..2.85 rows=1 width=0)
         ->  Limit  (cost=0.00..2.84 rows=1 width=52)
               ->  Index Scan using user_activity_uid on user_activity  (cost=0.00..17.07 rows=6 width=52
                     Index Cond: (uid = 73)
(5 rows)
```

Com veiem, no és gaire més costós que el pitjor cas de la versió anterior, però continua sent molt més eficient que l'estratègia del count(*) (el primer query és el mateix que a l'exemple anterior, l'he repetit senzillament per facilitar la comparació d'ambdós resultats).

Malgrat tot, però, encara és un cost prou elevat, si consideram que estam pensant implementar-ho en un procediment emmagatzemat que serà disparat per un trigger cada cop que s'efectui un determinat tipus d'operació damunt una taula que, podría ser, fos molt freqüentment modificada.

Però si ens miram la situació des d'una perspectiva més global, veim que el que normalment voldrem fer amb aquest valor booleà s avaluar-lo amb un *if (condició) then...* i no podem optimitzar la condició de l'*if()*, però pot ser sí la operació completa. Exemple:

```
Create or replace function prova ()
Returns trigger AS '
DECLARE
        foo record;
BEGIN
        FOR foo IN
                select * from user_activity where uid = 73
                limit 1
        LOOP
                /************************************
                Codi que executariem en cas d'existir
                activitat per a l'usuari.
                En cas de necessitar-se una clàusula
                "else", es pot implementar inicialitzant
                una variable i modificant el seu valor
                aquí.
                ************************************/
        END LOOP;

END;
' Language plpgsql;
```

Com que limitam el número de resultats de la consulta a 1, el bucle només s'executarà un cop o cap (equivalent a un *if()* sense clàus *else*. Ens estalviam el fatídic count(*) i a sobre queda bastant més elegant (sobretot si no necessitam *else*) que la perrogrullada del count(*) amb el subselect.

Finalment, només ens queda el cas de que necessitem aquest valor com a part d'una consulta SQL més complexa. En aquest cas no podem fer servir el truc del FOR i, de moment, a mi se m'han acabat els *asos* de davall la màniga... així que només no ens quedarà altra opció que fer servir el truc del count(*) amb el subselect i 'limit 1', que hem explicat abans per, al manco, minimitzar la pèrdua

de temps comptant. Però si algú té una idea millor, estic obert a suggeriments...

Vistes

Una vista ens permet veure el resultat d'una consulta complexa com si es tractàs d'una taula independent, en la que les dades es corresponen, en tot moment, amb el resultat de la consulta que implementa la vista.

Les vistes es poden classificar segons si son de només lectura o de lectura/escriptura però, principalment, en si son materialitzades o no:

Vistes no materialitzades

Les vistes més comuns son les vistes no materialitzades[31] que, en principi, son de només lectura i consisteixen, senzillament, en u *taula virtual* que funciona senzillament executant la consulta que defineix la vista cada cop que aquesta és invocada.

Per contra del que pugui parèixer, una vista no materialitzada només és útil en termes de simplificació de les consultes **no aporta c benefici en termes de rendiment** i son totalment equivalents a fer la mateixa consulta com a subquery.

Exemple:

```
test=# create table autors (
test(# aid serial primary key,
test(# nom varchar
test(# );
NOTICE:  CREATE TABLE will create implicit sequence "autors_aid_seq" for "serial" column "autors.aid"
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "autors_pkey" for table "autors"
CREATE TABLE
test=# create table llibres (
test(# lid serial primary key,
test(# aid integer references autors (aid),
test(# titol varchar
test(# );
NOTICE:  CREATE TABLE will create implicit sequence "llibres_lid_seq" for "serial" column "llibres.lid"
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "llibres_pkey" for table "llibres"
CREATE TABLE
test=# create view fitxes as select lid, titol, nom as autor from llibres natural join autors;
CREATE VIEW
test=# explain select lid, titol, nom as autor from autors llibres natural join autors where lid = 23;
ERROR:  column "lid" does not exist
test=# explain select lid, titol, nom as autor from llibres natural join autors where lid = 23;
                                  QUERY PLAN
------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..14.49 rows=3 width=68)
   ->  Index Scan using llibres_pkey on llibres  (cost=0.00..4.82 rows=2 width=40)
         Index Cond: (lid = 23)
   ->  Index Scan using autors_pkey on autors  (cost=0.00..4.82 rows=1 width=36)
         Index Cond: ("outer".aid = autors.aid)
(5 rows)

test=# explain select * from fitxes where lid = 23;
                                  QUERY PLAN
------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..14.49 rows=3 width=68)
   ->  Index Scan using llibres_pkey on llibres  (cost=0.00..4.82 rows=2 width=40)
         Index Cond: (lid = 23)
   ->  Index Scan using autors_pkey on autors  (cost=0.00..4.82 rows=1 width=36)
         Index Cond: ("outer".aid = autors.aid)
(5 rows)
```

Com a curiositat (aquest article tracta sobre optimització), tot i que una vista és, essencialment, de només lectura (com acabam d'explicar no és més que un subquery empaquetat), es poden definir regles que implementin les operacions[32] necessàries per *simul* l'escriptura sobre la vista (transformant aquestes operacions en les operacions necessàries sobre les taules originals).

Vistes materialitzades

Les vistes materialitzades sí que ens poden resultar molt profitoses en termes de rendiment, si be son mes costoses d'implementar i personalment, no m'he vist encara mai en la necessitat de fer-ho.

Una vista materialitzada, com el seu nom indica, no és més que una taula real que conté, idealment en tot moment, la mateixa informació que obtendriem executant el query que representa la vista. S'implementen a base de disparadors (triggers) que s'encarreguen de mantenir sincronitzades vista i taules mare en tot moment.

Igualment com amb les no materialitzades, es pot obtar perquè siguin de només lectura (només cal implementar sincronització en u sentit i prohibir l'escriptura directa) o de lectura escriptura (sincronització bidireccional).

Si vos interessa el tema, jo no l'he llegit completament, però aquest tutorial sobre el tema[33] pareix bastant bò.

Prepare

Quan llençam una consulta SQL, Postgres l'analitza i decideix quina és l'estratègia (o conjunt d'estratègies) més adequada per obter la informació sol·licitada en el menor temps possible. Aquesta anàlisi també consumeix temps en si mateixa i es repeteix per a cada consulta que executam.

Si hem d'executar moltes vegades una mateixa consulta però amb paràmetres diferents, aquest temps ens el podem estalviar gràcies les sentències prepare[34] / execute[35].

Amb prepare, invocam el planificador amb una consulta parametitzada i la planificació resultant quedarà "preparada" amb el nom especificat.

Després, amb la sentència *execute*, podem invocar el query ja preparat amb paràmetres diferents tantes vegades com volguem sense que s'executi de nou el planificador.

Aquesta tècnica ens pot agilitzar una mica consultes molt repetitives. Tot i que no és aconsellable utilitzar una sentència preparada forma indefinida doncs, a mesura que la base de dades va creixent (o canviant), els paràmetres estadístics que ha fet servir el planificador per determinar la millor estratègia, poden haver canviat.

Per tant, si es fa servir aquesta tècnica per processos molt llargs, pot ser sigui convenient regenerar la planificació de tant en tant.

Resum

- Revisar la configuració, especialment els paràmetres relacionats amb l'ús de memòria i ajustar-los a les nostres necessitats. Els més significatius:
  - shared_buffers =
  - sort_mem = / work_mem =
- Estodiar quins índexs ens cal crear i quins no i assegurar-nos de que realment es fan servir en les consultes.
- Verificar la planificació dels querys amb la comanda *explain*, especialment si tenim condicions parcials sobre l'índex.
- Recordar-nos de fer els castings que siguin necessaris en passar constants numèriques als querys si el tipus de la columna és integer.
- Evitar les funcions agregades com max() i min() per exemple, si la columna està indexada, combinant la ordenació descendent o ascendent, respectivament, amb la clàusula límit i, **sobretot** l'ús del fatídic *count(*)* per només saber si existe algún element que compleixi la condició requerida.
- Optimitzar els disparadors, especialment si son recursius i mirant d'evitar els processament pesats en els casos que no sigu necessaris.
- No descuidar el manteniment de la base de dades (Executar *vacuum* i/o *analyze* quan calgui).
- Si tens consultes molts pesants i, tot i les optimitzacions anteriors, et tarden massa, sempre pots provar d'implementar una vista materialitzada (**mai no materialitzada** (si el motiu és el rendiment).
- En consultes molt repetitives, pots porvar d'estalviar-te repetir cada cop la planificació amb prepare/execute.

Lista de enlaces de este artículo:

1. http://bulma.net/body.phtml?nIdNoticia=2468&nIdPage=2

2. http://bulma.net/body.phtml?nIdNoticia=2468&nIdPage=3
3. http://bulma.net/body.phtml?nIdNoticia=2468&nIdPage=4
4. http://bulma.net/body.phtml?nIdNoticia=2468&nIdPage=5
5. http://bulma.net/body.phtml?nIdNoticia=2468&nIdPage=6
6. http://www.postgresql.org/docs/
7. http://www.postgresql.org/docs/8.3/interactive/runtime-config.html
8. http://www.postgresql.org/docs/8.3/interactive/runtime-config-resource.html#RUNT
9. http://www.postgresql.org/docs/8.3/interactive/storage-file-layout.html
10. http://www.postgresql.org/docs/8.3/interactive/plpgsql.html
11. http://www.postgresql.org/docs/8.3/interactive/triggers.html
12. http://www.postgresql.org/docs/8.3/interactive/xplang.html
13. http://www.postgresql.org/docs/8.3/interactive/pltcl.html
14. http://www.postgresql.org/docs/8.3/interactive/plperl.html
15. http://www.postgresql.org/docs/8.3/interactive/plpython.html
16. http://www.postgresql.org/docs/8.3/interactive/external-pl.html
17. http://www.postgresql.org/docs/8.3/interactive/indexes.html
18. http://www.postgresql.org/docs/8.3/interactive/indexes-types.html
19. http://ca.wikipedia.org/wiki/Estructura_arbòria#Definici.C3.B3_d.27arbres_binari
20. http://www.postgresql.org/docs/8.3/interactive/indexes-multicolumn.html
21. http://bulma.net/javascript:return false;
22. http://www.postgresql.org/docs/8.3/interactive/trigger-definition.html
23. http://www.postgresql.org/docs/8.3/static/server-programming.html
24. http://www.postgresql.org/docs/8.3/interactive/plpgsql-errors-and-messages.html
25. http://www.postgresql.org/docs/8.3/interactive/routine-vacuuming.html#VACUUM-FOR
26. http://www.postgresql.org/docs/8.3/interactive/routine-vacuuming.html
27. http://www.postgresql.org/docs/8.3/interactive/routine-vacuuming.html#AUTOVACUUM
28. http://www.postgresql.org/docs/8.3/interactive/sql-analyze.html
29. http://www.postgresql.org/docs/current/static/functions-formatting.html
30. http://ca.wikipedia.org/wiki/Mallorca
31. http://www.postgresql.org/docs/8.3/interactive/sql-createview.html
32. http://www.postgresql.org/docs/8.3/interactive/rules-views.html
33. http://jonathangardner.net/tech/w/PostgreSQL/Materialized_Views
34. http://www.postgresql.org/docs/8.3/interactive/sql-prepare.html
35. http://www.postgresql.org/docs/8.3/interactive/sql-execute.html

---

E-mail del autor: joanmi _ARROBA_ bulma.net
Podrás encontrar este artículo e información adicional en: http://bulma.net/body.phtml?nIdNoticia=2468