Join eredmények Lekérdezés alapú tömörítése

Készült
Christopher M. Mullins
(University of Hawai'i at Manoa, USA)
Lipyeow Lim
(University of Hawai'i at Manoa, USA)
Christian A. Lang
(Acelot Inc., USA)
„Query-Aware Compression of Join Results"
című tanulmányának alapján (EDBT '13 March 18-22-2013 Genoa, Italy).

Készítette:
Horváth Viktor János – leírás
Dananaj Pál – program
Ancsin Attila - prezentáció

2013.

Absztakt

A kliens-szerver adatbázis lekérdezés feldolgozás napjainkban sok adatfeldolgozással foglalkozó alkalmazásoknál fontos szerepet tölt be. Felhő alapú rendszerekben, például egy strukturált adathalmaz felett értelmezett lekérdezések érkeznek a felhő alapú szervernek, és az eredményt visszaküldi a szerver, a kliens eszközökre. A hálózati sávszélesség általában korlátozott a kliens eszközök és a felhő alapú szerverek között, ezért az adatok tömörített továbbítása mindenképpen javíthat a hálózat leterheltségén és az átvitel sebességén. Ezen felül kevesebb adat küldése/fogadása a hálózaton, kevesebb energiával jár, ami mobil eszközök esetén fontos tényező. A jelenlegi lekérdezés sorokat tömörítő eljárások nem használják fel a lekérdezés szerkezetét a redundáns elemek azonosításához.
Az algoritmus hatékonyságát gyakorlati példák méréseinek alapján mutatjuk be. Ezt az algoritmus használva, akár kétszer nagyobb tömörítési hatékonyság érhető el, mint gzip esetében.

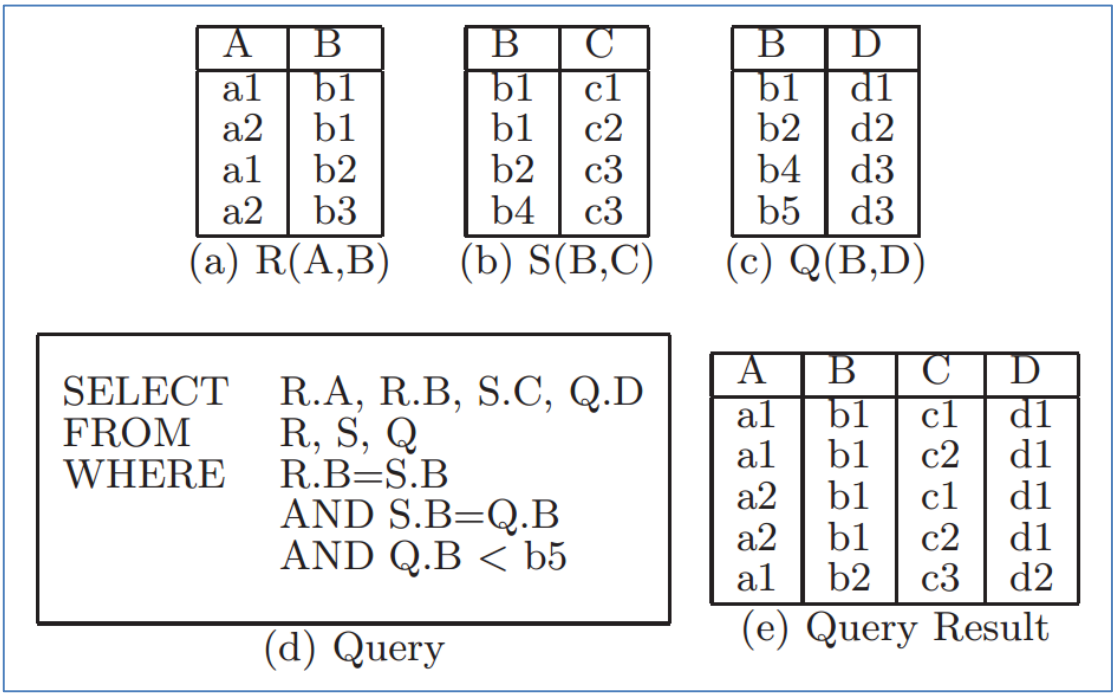
Bevezetés

Kliens-szerver adatbázis lekérdezések nagyon sok adatközpontú alkalmazás fő alkotóelemei. Vegyük példának a felhő alapú szolgáltatásokat és az elosztott adattárházakat. Felhő alapú szolgáltatásoknál, a szerveren végrehajtott lekérdezés eredményeit, a szerver elküldi a mobil kliens eszköznek. Az ilyen mobil eszközök gyakran limitált hálózati sávszélességgel rendelkeznek, valamint rövid akkumulátoridővel. Tehát bármilyen erőfeszítés, amit az átküldendő adatmennyiség csökkentése érdekében tennénk, nemcsak a szükséges sávszélességet csökkentené, de az akkumulátort is kímélné, hiszen mobil eszközöknél a hálózati kommunikáció az egyik legenergiaköltségesebb dolog. Elosztott adattárházak esetében a felhasználó által megadott lekérdezések, további al-lekérdezésekre partícionálódnak és az egyes ilyen al-lekérdezések külön worker csomóponton kerülnek lefuttatásra. Az egyes worker csomópontokon kapott lekérdezés eredmények következő worker csomópontokoz kerülnek további feldolgozásra, vagy egyesülnek a koordinátor csomópontban, hogy a végleges eredmény eljusson a felhasználóhoz. Láthatóan a worker csomópontok közötti adatmozgatás sebességének növelése, jelentős gyorsítással járhat. Megfelelő tömörítéssel kevesebb adatot kell mozgatni a csomópontok között, vagyis nő az adatmozgatás sebessége, tehát ez esetben is jelentős gyorsulás érhető el tömörítéssel.

Az adat tömörítő módszerek és adatbázisrendszerekben történő alkalmazásuk nem új keletű dolog. Az itt bemutatott algoritmus abban különbözik az eddigiektől, hogy a lekérdezést használja redundáns elemek azonosítására, ezáltal nagyobb tömörítési hatásfok érhető el, vagyis a méret még jobban csökkenthető. A következőekben tehát egy újfajta lekérdezés alapú (query aware, QA) tömörítést mutatunk be. Az algoritmus két ötleten alapul:

- a lekérdezés plan-jéből (szerkezetéből) és az adatbázis sémából nyerünk információt a redundáns elemek azonosításához
- az elemek (attribútumok) hatékony és könnyűsúlyú kódolásához szótárakat használunk

Vegyük a *1. ábrán* található példát! A lekérdezés az *1 (d) ábrán* látható, és a hozzá tartozó eredmény sorok az *1 (e) ábrán*. Ezek a rekordok elég sok redundáns értékeket tartalmaznak, amiket jó lenne, ha tömörítéssel kiküszöbölhetnénk: az (A, B) oszlopokban az (a1, b1) értékek és az (a2, b1) értékek kétszer ismétlődnek; a (B, D) oszlopokban a (b1, d1)



1. ábra

értékek négyszer ismétlődnek; a (B, C) oszlopokban a (b1, c1) értékek kétszer ismétlődnek. Hagyományos oszlop alapú vagy sor alapú szótáras tömörítő eljárások nem lennének képesek ezeket a redundanciákat észlelni, mert nem foglalkoznak azzal az információval, hogy az eredmény sorok az R(A, B), S(B, C) és a Q(B, D) relációk join-olásával keletkeztek.

Ezeket a fajta redundanciákat pusztán az eredménysorokból kikövetkeztetni, megtalálni kombinatorikai probléma, és a gyakorlatban megvalósíthatatlannak tűnik. Ezért e helyett, az itt bemutatott algoritmus, ezeket a redundanciákat a lekérdezésből fogja kikövetkeztetni. Ehhez a lekérdezéshez tartozó join fát (join tree) fogja használni, és szótárakat, pontosabban szótár hierarchiát fog készíteni. A szükséges szótár hierarchia mérete a legrosszabb esetben a lekérdezésben részt vevő relációk száma. A kliensnek – aki a szervertől megkapta a tömörített eredményt – a kitömörítéshez újra fel kell építenie a szótár hierarchiát. Ehhez nagyobb memória is szükséges lehet, egyes könnyűsúlyú kliensek esetében ez nem biztos, hogy rendelkezésre áll. A memóriahiány probléma megoldására limitált méretű szótárakat lehet használni az algoritmus során.

Megjegyezzük, hogy az itt bemutatott tömörítő eljárás nem ekvivalens azzal, hogy a lekérdezés alap relációit elküldjük a kliensnek (szükséges szelekciók és projekciók elvégzése után) és a kliensen végezzük el a join műveleteket és az egyéb projekciókat. Ebben az esetben a kliensnek kellene a join feltételeket kiértékelnie, vagyis a teljes alaprelációkkal rendelkeznie kellene (a szükséges projekcióktól eltekintve). Az általunk használt algoritmus nem csinálja ezt, a teljes lekérdezést a szerveren hajtjuk végre és a lekérdezés eredményét tömörítjük szótár hierarchiák segítségével. Ezen felül limitált méretű szótárak használata esetén nem szükséges rendelkezni a teljes alaprelációkkal se a betömörítésnél, se a kitömörítésnél.

Az itt használt QA tömörítés használható bármely olyan alkalmazásokban, ahol egy adatbázis szervernek szükséges a lekérdezés eredményét közvetíteni hálózaton keresztül kliensalkalmazások felé. Összetömörített eredményhalmazok esetén kisebb sávszélesség szükséges az átvitelhez, vagyis csökken a lekérdezés végrehajtásának ideje, a kliens

szemszögéből. Továbbá elképzelhetőnek tartjuk, hogy ezt a módszer hasznosítható lenne ODBC/JDBC rétegekben, és elosztott relációs adatbázis architektúrák (DRDA) esetén is. Elosztott relációs adatbázis környezetben a módszer használható az al-lekérdezések eredményeinek tömörítésére, amik az egyes adatbázis csomópontokon (node) hajtódnak végre. Ez a módszer elég általános, és olyan tetszőleges relációkkal használható, melyek veszteségmentesen join dekompozícionálhatók.

Közlemények az eredeti készítőktől:

- Mi egy újfajta QA tömörítő módszert terveztünk, amivel lekérdezések eredménysorait lehet tömöríteni, a lekérdezés struktúrájából nyert információk alapján. Nincs tudomásunk létező módszerről, amik ehhez hasonlóan működne.
- Memória limitált szótár adatstruktúrát terveztünk, ami szabályozza a memóriahasználat mértékét, így a kitömörítés jobban helytakarékos lehet.
- A módszer hatékonyságát, gyakorlati mérések alapján igazoltuk.

Kapcsolódó munkák.

Több tömörítő eljárást is dolgoztak már ki korábban lekérdezés sorok tömörítésére. Ebben a rövid fejezetben, ezekre adunk példákat.

A legszélesebb körben alkalmazott tömörítő módszerek Huffman kód [1], aritmetikai kódolás [2], vagy a Lempel-Ziv szótár tömörítési stratégia [3, 4] alapján működnek. Egy szótárat használnak, hogy kódolják a bemenetről érkező tetszőlegesen nagy méretű adatokat. Huffman kód esetén például a gyakrabban előforduló részek, rövidebb kóddal fognak a szótárba kerülni. Join-olt lekérdezések eredménysorai bonyolult és többszintű ismétlődési mintával rendelkeznek, ezért egy ilyen szótár túl nagy lenne (növelné a kitömörítés erőforrásigényét).

Mostanában, formális nyelv (tan) alapú tömörítésekről is lehet hallani. A bemeneti adatokat környezetfüggetlen nyelvtan (CFG) segítségével írják le. A legkisebb nyelvtan megtalálása NP nehéz probléma, ezért különböző heurisztikák terjedtek el: Sakamoto [5] lineáris futási idejű algoritmust mutatott be. ami hatékonyabban teljesít sok ismétlődést tartalmazó input esetén, mint egy hagyományos szótár alapú algoritmus.

Adattömörítés alkalmazása relációs táblákra, indexekre jellemző probléma. Cormack [6] készített ilyen algoritmus, ami Huffman kódot használ. Roth és Van Horn [7] több féle módszert is kidolgozott relációs adatbázis környezetben történő tömörítéssel kapcsolatban. Ng és Ravishankar [8] olyan algoritmust készítettek, ami jól használható lokális kitömörítéssel, és a tömörített adaton hatékonyan lehet lekérdezéseket végezni.

Az itt bemutatott algoritmushoz legközelebb álló megközelítés Goh és társainak [9] publikációja, ahol az adatbázis táblákból szerzett információ (asszociációs szabályok) alapján történik a tömörítés.

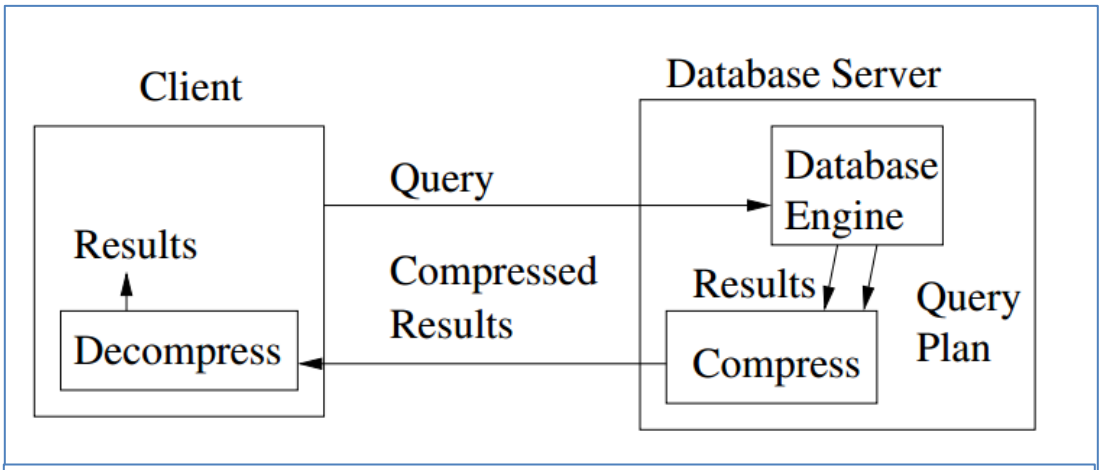
Tudomásunk szerint, az itt ismertetett megközelítést senki nem alkalmazta még eddig.

Tömörítés

A be és kitömörítés működése a *2. ábrán* látható. A kliens elküld egy adatbázis lekérdezést az adatbázis szervernek. Az adatbázis motor feldolgozza a lekérdezést és elküldi az eredménysorokat és a lekérdezés végrehajtási tervet (query execution plan) a tömörítőnek. A tömörítő betömöríti az eredménysorokat, a plan-ből nyert redundancia

információk segítségével, és továbbítja a tömörített adatfolyamot hálózaton a kliens felé. A kliens megkapja ezt a tömörített adatfolyamot és odaadja a kitömörítőnek, aki kitömöríti. Az így kitömörített adatok kerülnek a kliens programok kezébe. A tömörítő algoritmus szimmetrikus, vagyis a kitömörítő algoritmus „leutánozza” a tömörítő algoritmus állapotait, hogy dekódolja a tömörített adatfolyamot. A következőkben több fogalmat definiálunk, hogy az algoritmus működését leírhassuk.

A lekérdezés végrehajtási terv (query execution plan, továbbiakban plan), egy fa. A fa

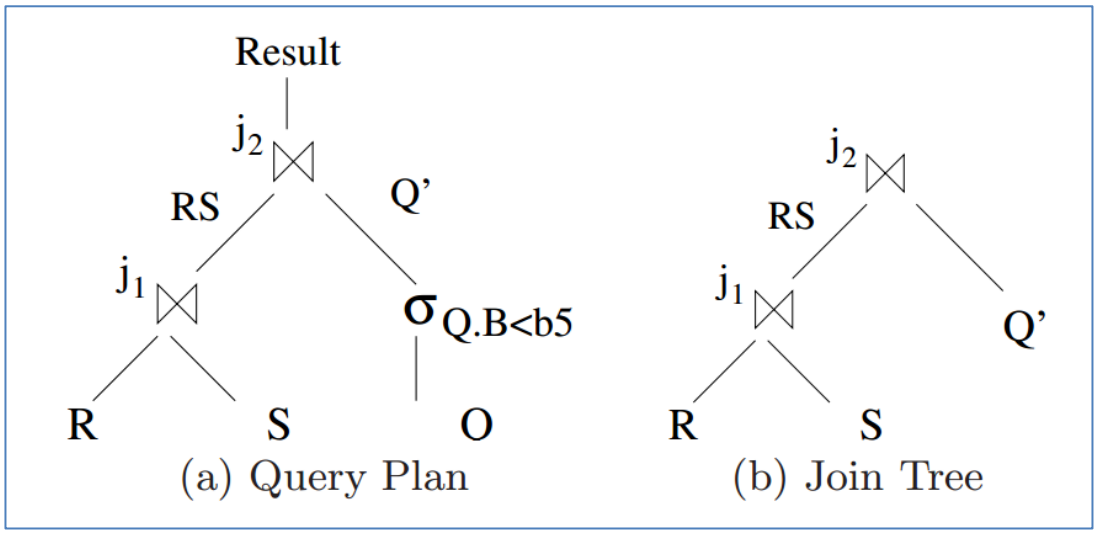


2. ábra

nem levél csomópontjaiban relációs algebrai (továbbiakban RA) operátorok szerepelnek *(σ, π, join),* a levél csomópontokban relációk szerepelnek. A fa szemlélteti, hogy a levél csomópontokban levő relációkon milyen műveleteket végzünk (a nem levél csomópontok műveletei), a fa csúcsa a teljes lekérdezés szerkezetét mutatja. Az algoritmus feltételezi, hogy a plan-ben, minden nem join művelet a lehető legközelebb szerepel az alaprelációkhoz (levelekhez). Ezt megtehetjük, hiszen létezik általános algoritmus, ami bármilyen plan-hez megadja, a plan-nel ekvivalens plan-t amiben a nem join műveletek a relációkhoz a lehető legközelebb szerepelnek.

Join fa (join tree, továbbiakban jtree), olyan fa, ahol a belső csomópontok join operátorok, a levél csomópontok relációk, vagy RA műveletekkel képzett ideiglenes relációk. A jtree elkészíthető a plan-ből, úgy, hogy minden egyes olyan levéllel rendelkező részfát, amelyben nincs join operátor, egy ideiglenes reláció csomóponttá összekapcsolunk. Látható tehát, hogy a jtree, egy olyan plan, amiben csak join operátorok vannak, és a levelek RA operátorokkal képzett ideiglenes (összeolvasztott) relációk, de mondhatjuk akár azt is, hogy minden levél egy nézet (view). Amennyiben egy join csomópont alatt a plan-ben szerepel egy RA operátor, akkor ez a join csomópont nem vesz részt a jtree-ben (az RA operátorokat a lehető legalacsonyabb szintre vittük, ez volt a feltételezés, de attól még ilyen eset előfordulhat). A *3. ábrán* látható egy plan és a hozzá tartozó jtree. Látható, hogy a szelekció művelet a join műveletek alatt található, a lehető legközelebb a relációkhoz.

A tömörítés végrehajtása során, a lekérdezés eredményének minden sorát kódolni fogjuk, egymásba ágyazott szótárak segítségével – a jtree minden belső csomópontjához lesz egy szótár. Továbbá minden az eredmény minden oszlopát is kódolni fogjuk oszlop-alapú szótárakkal. Mivel a kitömörítés szimmetrikus, ezért a hálózaton először a jtree-t és a szótárakat küldi a szerver adatfolyam szerűen a kliens felé, és csak legvégén a kódolt eredménysorokat. Az algoritmus tehát stream-elve (adatfolyamon) végzi el a tömörítést, ahogy ez majd látható lesz. A kitömörítő a másik oldalon (a kliens oldalon) ugyanúgy felépíti saját magának a szótárakat - amikor megkap egy szótár bejegyzést, akkor beszúrja magához. A 3 (b) ábrán látható jtree-hez, üres szótárakat inicializálunk minden belső csomóponthoz: { D(R), D(S), D(Q'), D(j1) }, valamint minden reláció-oszlophoz (a lekérdezés eredménysorainak oszlopai): { D(A), D(B), D(C), D(D) }. Amikor egy új bejegyzés készül valamelyik szótárba, az új bejegyzés és a szótár azonosítót elküldjük a kitömörítőnek, hogy a szótárak szinkronban maradjanak a ki és betömörítő között. Az új bejegyzés mellé egy DENTRY flag is kerül, hogy a kitömörítő tudja, hogy most szótárbejegyzést kapott és nem egy kódolt eredménysort. Az algoritmus végigiterál az eredmény minden során és lekódolja a sort, a jtree mélységi bejárásával. A sor minden mezőjét oszlop-alapú szótárakkal kódolja l. A kód, amit a jtree egy csomópontjából készít, az levelek esetén a kódolt oszlopokból áll, belső csomópontok esetén rekurzívan a csomópont gyerekeinek kódjából. A nem gyökér csomópontok ezen felül még a



3. ábra

csomóponthoz tartozó szótárral is kódolva lesznek. A gyökér csomóponthoz nem tartozik szótár (ez látható a példában is), csak a gyerekeinek kódja lesz továbbküldve a kitömörítő felé. A *3 (b) ábrán* látható jtree-hez tartozó lekérdezést a következő szótárakkal kódoljuk: ‹ D(A), D(B), D(R), D(C), D(S), D(j1), D(D), D(Q') ›.

Betömörítő algoritmus

A lekérdezés eredménysorait betömörítő algoritmus, a *4. ábrán* található (COMPRESS). Bemenetként megkapja a jtree-t (T) és az eredménysorokat (W) az adatbázis motortól. A W, olyan sorok halmaza, aminek a sémája a lekérdezés SELECT részében levő oszlopokkal megegyezik. A tömörítő először sorosítja (serialize) a jtree-t és elküldi a kitömörítőnek. A jtree levél csúcsaiban levő relációk sémáit is elküldi a kitömörítőnek. Mi itt feltesszük, hogy a sémák logikai részei a jtree-nek (benne vannak a sémák), amit az algoritmus bemenetnek kap. Ezután a jtree és a sémák segítségével inicializálja a szótárakat – minden belső nem

Algorithm 1 COMPRESS(T, W)

Input: Join tree T, Result set W
Output: Sends compressed result set to decompressor

```
1: send join tree T and schema of leaf nodes
2: 𝒟 ← ∅
3: for all non-root node v ∈ T do
4:    initialize dictionary D(v)
5:    𝒟 ← 𝒟 ∪ D(v)
6:    if v is a leaf node then
7:       for all columns c of relation v do
8:          initialize dictionary D(c)
9:          𝒟 ← 𝒟 ∪ D(c)
10: for all row r ∈ W do
11:    send CompressRow(root(T), 𝒟, r)
```

4. ábra

Algorithm 2 COMPRESSROW$(t, \mathcal{D}, r)$

Input: Join tree node t, collection of dictionaries $\mathcal{D}$, a row r to be compressed
Output: Sends dictionary entries to decompressor and returns compressed row

```
1: rowcode ← ϵ
2: if t is a leaf then
3:    for all columns c of relation t do
4:       code ← DictLookup( r[c], D(c))
5:       if code = ϵ then
6:          code ← DictAdd(D(c), r[c])
7:          send DENTRY, c, r[c],
8:       rowcode ← rowcode · code
9: else
10:   leftcode ← CompressRow(t.left, 𝒟, r)
11:   rightcode ← CompressRow(t.right, 𝒟, r)
12:   rowcode ← leftcode · rightcode
13: if t is non-root then
14:   code ← DictLookup( rowcode, D(t))
15:   if code = ϵ then
16:      code ← DictAdd(D(t), rowcode)
17:      send DENTRY, t, rowcode,
18:   rowcode ← code
19: return rowcode
```

5. ábra

gyökér csomópontnak 1 darabot. Jelölje v egy belső nem gyökér csomópontját a jtree-nek, és D(v) a v-hez tartozó szótárak halmazát, és jelölje D a teljes jtree-hez tartozó szótárakat.

Minden ilyen szótárat, arra fogunk használni, hogy kódoljuk minden nem gyökér csomóponthoz tartozó mezőket. A levelekhez további szótárakat inicializál az algoritmus, a reláció egyes oszlopainak kódolására. Jelölje D(c), a c oszlophoz tartozó szótárat (feltételezzük, hogy az oszlopok, oszlopnevek egyediek minden lekérdezésben részt vevő táblán, együttesen). Egy csomóponthoz tartozó relációra, gyakran csak a csomóponttal fogunk hivatkozni. Amikor a szótárakat inicializálta, akkor a betömörítő végigiterál minden soron és betömöríti a jtree és a szótárak segítségével. Figyeljük meg, hogy a kitömörítő egyaránt kaphat szótárbejegyzéseket vagy kódolt sorokat a betömörítőtől (stream-elve dolgozza fel a bemenetet).

Egy sornak a betömörítési algoritmusa az *5. ábrán* látható (COMPRESSROW). Egy rekurzív függvény a jtree-n. Bemenete a jtree egyik csomópontja, szótárak halmaza, a tömörítendő sor. Kimenete a tömörített sor reprezentációja a megadott jtree és a szótárak alapján. Jelölje a bemeneti sort (a sor mezőinek értékeit) r, és r[t] az r értékeinek részhalmazát, amelyik a t csomóponthoz tartoznak. Hasonlóan r[c] jelöli az r csomópont (levél) c oszlopba tartozó értékeit. A rekurzió alapesete, amikor az input jtree csomópont levél (1-8. sor). Az algoritmus megpróbálja betömöríteni az input sort oszloponként, az oszlop szótárak segítségével. Az így kapott tömörített sort még tovább tömöríti a t input csomópont D(t) szótárát használva (13-18. sor). Általános szótár alapú kódolás során, ha egy olyan érték kerül sorra, ami még nem szerepel a szótárban, akkor az új érték bekerül a szótárba és a kódszót rendel hozzá az algoritmus. A másik oldalon a kitömörítőnek is frissíteni kell a szótárat ez esetben, ezért a szótár azonosító és az új érték is átküldésre kerül a kitömörítőhöz. Ha a kitömörítőben található szótárba ugyanolyan módszerrel (determinisztikus módon) generálja a kódszót, mint a betömörítőben, akkor betömörítő nem szükséges, hogy átküldje a kódszót.

A sor tömörítés rekurzív esete, amikor egy belső csomópontot kap input-nak. Ilyenkor az algoritmus először a bal oldali részfát dolgozza fel (tömöríti), utána a jobb oldalit. Majd ezután még a saját magához (legyen t a csomópont) tartozó D(t) szótárat használva, a kapott gyerek fáinak kódját tovább tömöríti (13-18. sor).

DE, D(A), a1, DE, D(B), b1, DE, D(R), 0, 0, DE, D(C), c1, DE, D(S), 0, DE, D(j_1), 0, 0, DE, D(D), d1, DE, D(Q′), 0 TF, 0, 0	DE, D(C), c2, DE, D(S), 1, DE, D(j_1), 0, 1, TF, 1, 0

6. ábra

Példa. Nézzük meg az *1(e) ábrán* található eredménysorok tömörítését a *3(b) ábrán* levő jtree segítségével! Az algoritmus működése ezen a példán (az első két sorra) a *6. ábrán* található (jelmagyarázat: DE – dictonary entry/szótárbejegyzés, TF – tuple fragment/kódolt sor). Az egyszerűség kedvéért tegyük fel, hogy itt a szótárakban levő kódok egész (integer) számok 0-tól kezdve. Minden mező minden sorban először az oszlop alapú szótárral lesz kódolva, utána a csomóponthoz tartozó szótárral. Az algoritmus először az A oszlop a1 értékét találja meg, és a megfelelő oszlopszótárba rakja be. Majd ugyanez történik a B oszlop

b1 értékével, hiszen mindkettő oszlop az R csomóponthoz (relációhoz) tartozik. Utána a csomóponthoz tartozó szótárba is kódolásra kerül ez a két érték (a 0, 0 jelenti azt, hogy D(A) 0. eleme, és D(B) 0. eleme, ezek az a1 és b1), tehát az (a1, b1) párosra ezen túl elég lesz a D(R) szótárban levő 0. elemre hivatkozni. Az algoritmus ezután továbbmegy az S csomópontra és ott is végigmegy az oszlopokon (most egy darab oszlop). Ezért a D(C) szótárba bekerül a c1 érték, majd a csomópont szerint is lekódolja és a D(S)-be is bekerül (látható, hogy nem a c1 érték kerül a D(S)-be, hanem az, hogy a c1 milyen kódszóval/kulccsal szerepel a D(S)-ben). A j1 csomópont mindkét gyerekét feldolgozta, ezért most a gyerekeinek összefűzését keresi meg a hozzá tartozó D(j1) szótárban. Mivel nem találja, ezért beszúrja (figyeljük meg, hogy a D(R) és D(S)-beli kódszavakat/kulcsokat szúrja be a szótárba). A j2 bal oldali részfája feldolgozva tehát, most következik a jobb oldali részfa, ami most a Q' csomópont. Megint a Q'-ben levő oszlop-szerinti szótártömörítés következik, az ábrán látszik is, hogy a d1 belekerül a szótárba (hiszen eddig üres volt a D(D) szótár). Oszlopszótár után, megint a csomópont szótár jön, D(Q')-be bekerül a 0, vagyis a D(D) szótár 0. kódszavú/kulcsú elemére hivatkozás. Most a j2 mindkét gyerekének részfája feldolgozásra került, ezért mivel a j2 gyökércsúcs, most visszaadjuk a kódolt sornak a reprezentációját, ami „0, 0", a D(j1) és D(Q') szótár kódszavak/kulcsok/indexek. A következő sor ehhez hasonlóan alakul, annyi különbséggel, hogy több értéket is meg fog találni szótárakban (redundanciát észleli), de pl. a c2 új érték, amit berak a megfelelő szótárba. Az algoritmus még jobban szemléltethető, ha minden lépésnél feljegyezzük a szótárak tartalmát, és akkor könnyebben megérthető a példa (és ez által az algoritmus működése is).

Kitömörítő algoritmus

A kitömörítési eljárás szimmetrikus. A kitömörítő újraépíti a kliens oldalon, a szervertől kapott jtree-t és a szótárakat a folyamatosan kapott bejegyzések alapján. A teljes kitömörítő

Algorithm 3 DECOMPRESS

Input: Receives join tree and compressed result set from compressor
Output: Returns the uncompressed result set

1: $T \leftarrow$ receives join tree and schema of leaf nodes
2: $\mathcal{D} \leftarrow \emptyset$
3: **for all** non-root node $v \in T$ **do**
4: initialize dictionary $D(v)$
5: $\mathcal{D} \leftarrow \mathcal{D} \cup D(v)$
6: **if** v is a leaf node **then**
7: **for all** columns c of relation v **do**
8: initialize dictionary $D(c)$
9: $\mathcal{D} \leftarrow \mathcal{D} \cup D(c)$
10: $W \leftarrow \emptyset$
11: **while** receives message m from compressor **do**
12: **if** $m.flag =$ DENTRY **then**
13: DICTADD$(D(m.nodeID), m.value)$
14: **else**
15: $W \leftarrow W \cup$ DECOMPRESSROW$(T, \mathcal{D}, m.value)$
16: **return** W

7. ábra

algoritmus a *7. ábrán* látható, az egy sort kitömörítő algoritmus a *8. ábrán* található. Az utóbbiban található REVERSEDICTLOOKUP(D, i) függvény visszaadja a D szótárban az i kódszó/kulcs/index alatt levő értéket.

Algorithm 4 DECOMPRESSROW$(t, \mathcal{D}, e)$

Input: Join tree node t, collection of dictionaries $\mathcal{D}$, a tuple of dictionary codes e
Output: Returns the decompressed row

```
1: if t is a leaf node then
2:     row ← ∅
3:     for all columns c in relation t do
4:         row ← row ∪ REVERSEDICTLOOKUP(D(c), e[c])
5:     return row
6: else
7:     l ← REVERSEDICTLOOKUP(D(t.left), e[0])
8:     r ← REVERSEDICTLOOKUP(D(t.right), e[1])
9:     row ← DECOMPRESSROW(t.left, D, l)
10:    row ← row ∪ DECOMPRESSROW(t.right, D, r)
11:    return row
```

8. ábra

Példa. Nézzük a *6.ábrán* levő betömörítési üzeneteket! Tegyük fel, hogy a kitömörítő ezeket az üzeneteket kapta, megnézzük, hogyan dolgozza fel őket, és állítja elő a lekérdezés sor(oka)t. Mielőtt a kitömörítő megkapná a tömörített sort, előtte megkapja a szükséges szótárbejegyzéseket is, ezért lehet megvalósítani a kitömörítést a most bemutatott módon. A kitömörítő először felépíti a szótárakat a kapott bejegyzések alapján, a felépített szótárak a *9. ábrán* láthatóak. A „Row 1” oszlopban levő értékek az 1. sor érkezése előtt kerültek be, a „Row 2” oszlopban levő értékek az 1. sor érkezése után és a 2. sor érkezése előtt kerültek be

	Row 1	**Row 2**
$D(A)$	0 = a1	
$D(B)$	0 = b1	
$D(C)$	0 = c1	1 = c2
$D(D)$	0 = d1	
$D(R)$	0 = (0,0)	
$D(S)$	0 = (0)	1 = (1)
$D(Q')$	0 = (0)	
$D(j_1)$	0 = (0,0)	1 = (0,1)

9. ábra

a szótárakba. Jelölje D(X):n, a D(X) szótár n kódszavú/kulcsú/indexű értékét. Amikor a kitömörítő megkapja a „TF 0, 0” bejegyzést, akkor tudja, hogy két érték konkatenációját kell

keresni: D(j1):0 és D(Q'):0. A bejegyzés dekódolása lekérdezés sorrá a következő lépések szerint történik:

1. TF = (0, 0)
2. (D(j1):0 = (0, 0), D(Q'):0 = (0))
3. (D(R):0 = (0, 0), D(S):0 = (0), D(D):0 = d1)
4. (D(A):0 = a1, D=(B):0 = b1, D(C):0 = c1, d1)
5. (a1, b1, c1, d1)

Látható, hogy a szótárakban levő értékeket addig „dereferáljuk" amíg nem kapunk konkrét értéket. Azt, hogy a szótárakban levő értékek „hova mutatnak" a jtree-ből egyértelműen ki lehet következtetni. Következőnek jönnek a 2. sorhoz tartozó szótárbejegyzések, majd a „TF 1, 0" ami maga a 2. sor kódolva (tömörítve). A 2. sor kitömörítésének menete:

1. TF = (1, 0)
2. (D(j1):1 = (0, 1), D(Q'):0 = (0))
3. (D(R):0 = (0, 0), D(S):1 = (1), D(D):0 = a1)
4. (D(A):0 = a1, D(B):0 = b2, D(C):1 = c2, d1)
5. (a1, b1, c2, d1)

Itt is sikeresen visszakaptuk a teljes sort.

Limitált szótárak

Eddig a generált szótárak, bejegyzéseinek maximális száma nem volt korlátozva, de ez gyakorlatban nem mindig kivitelezhető. Ha pl. nagyon sok eredménysort ad vissza a lekérdezés, és nagyon sok egyedi érték szerepel benne, akkor a szótárak mérete jelentősen megnőne. Itt most nemcsak az alacsony akkumulátor energiával rendelkező mobil klienseket érdemes megemlíteni, hanem pl. nagy adatbázisok esetén, akár még egy átlagos PC memóriájába sem férne bele az összes generált szótárbejegyzés, hiszen táblák kétoldalú join-jával ugrásszerűen megnő a sorok száma. A véges memória problémaköre mellett nem érdemes elmenni szó nélkül.

A véges memória kezelés problémának megoldására limitált méretű szótárakat használhatunk. Az ilyen szótárak fix memóriaméretet foglalnak el, ezt a méretet byte-ban vagy bejegyzés számban lehet megadni, az egyszerűség kedvéért itt az utóbbiról lesz szó. Egy limitált szótár n darab hellyel képes n darab bejegyzés tárolására. A példák terminológiáját folytatva, így egy kódszó/kulcs/index [0..n-1] tartományba esik. Kezdetben a szótár üres. Amikor egy új bejegyzés érkezik a szótárba (új=nincs benne az az érték), akkor egy üres helyre bekerül az érték, és a maradék helyek száma 1-el csökken. Ha nincs üres hely, akkor egy meglévő lefoglalt bejegyzés törlődik, és a helyére kerül az új érték (az index most már az új értékre fog mutatni). Ez nem okozhat semmilyen problémát kitömörítésnél (a helyesség megmarad), ha az előbbiekben szemléltetett stream-eléssel küldjük a bejegyzéseket, valamint ha a kitömörítő szimmetrikusan működik. Értékcsere esetén az, hogy melyik indexet cseréljük ki az új értékre, nem mindegy. Hasonló lehetőségek vannak, mint cache-elésnél (hiszen az is hasonlóan működik): legutoljára használt (Last Recently Used - LRU), legkevésbé használt (Last Frequently Used - LFU), legutoljára hozzáadott (Last Recently Added - LRA) stb. Az LRU a legrégebben használt értéket dobná el, az LFU a legritkábban használt értéket dobná el. Mindkét módszerhez szükséges letárolni a bejegyzésekhez egy időbélyeget vagy egy gyakoriságot, ami alapján a bejegyzések összehasonlíthatóak.

Eredmények

Implementáció. Az algoritmus megvalósítása C++ nyelven történt meg, sqlite3 adatbázis motor használatával. A stream, amit az algoritmus készít néhány kisebb optimalizáción esett át, ami itt nincs részletezve. A szótár indexelése például változó bájt szélességű formátumot használatával történik, az alacsony overhead miatt.

Adatbázis. A sebesség mérésekhez a szerzők TPC-H [10] adatbázist használtak.

Mérési metrika. A tömörítési arány: nyers adatok mérete byte-ban/tömörített adatok mérete byte-ban. A nyers adatok CSV fájlba készültek, hogy a méret meghatározható legyen. A gzip méretek, gzip programmal tömörített fájlok méretei. Az időmérés a szokásos időmérő Linux paranccsal történt.

Algoritmusok. A mérések során a következő algoritmusokat mérték:

- gzip: a nyers eredmény gzip-el tömörítve
- query-aware-gzip: a bemutatott QA tömörítése a nyers eredménynek, majd az így kapott eredmény-t gzip-el tömörítve

A QA algoritmust önmagában nem hasonlítjuk a gzip-hez, mert nem helyettesítheti.

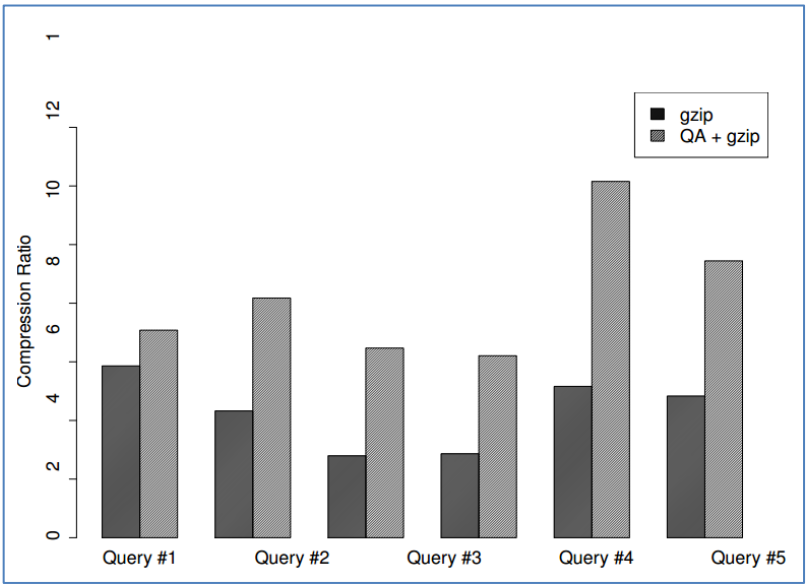
Lekérdezések. Csak join-okat tartalmazó lekérdezések lettek tesztelve, minden szűrőfeltétel, csoportosítás, rendezés nélkül. A szelekciók és projekciók is ki lettek hagyva, mert nem befolyásolják a tömörítési arányt.

A következőkben mérésekkel megvizsgáljuk, hogy a szótárméretek és az eredménysorok száma hogyan befolyásolják a tömörítési arányt. A szótár méret megadja, hogy az egyes szótárakban maximum mennyi bejegyzés van.

Általános tömörítési arány

Először egy általános esetben vizsgáljuk a tömörítési arányt, rögzített eredménysor mennyiséggel és rögzített szótárérettel. A szótárméret 50ezer darab bejegyzés, az arány faktor 0,21 (TPC-H fogalom, „scale factor”, az eredménysorok mennyiségét határozza meg, egyenesen arányos vele). Az eredmények a *10. ábrán* találhatóak.

vÉszrevehető, hogy a legtöbb lekérdezésnél, a query-aware-gzip megoldás majdnem

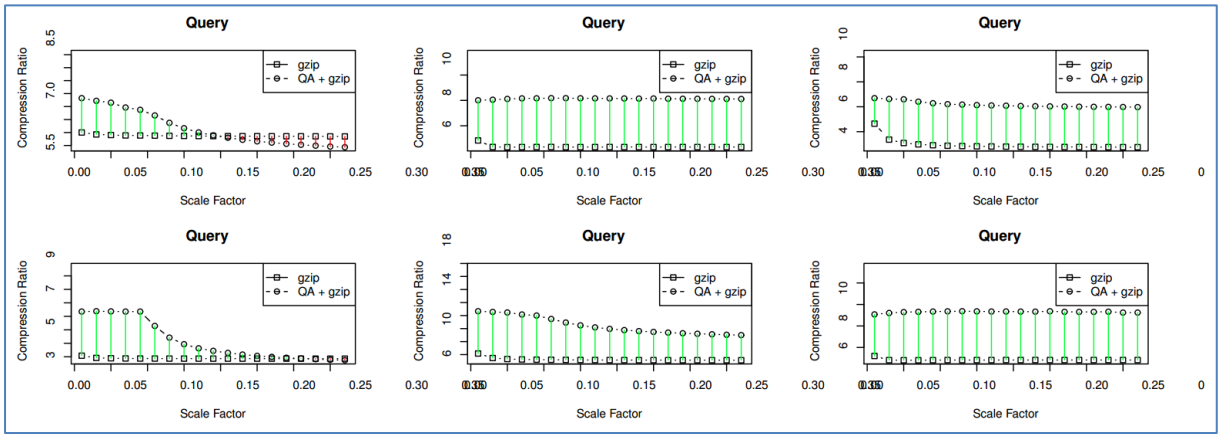


10. ábra

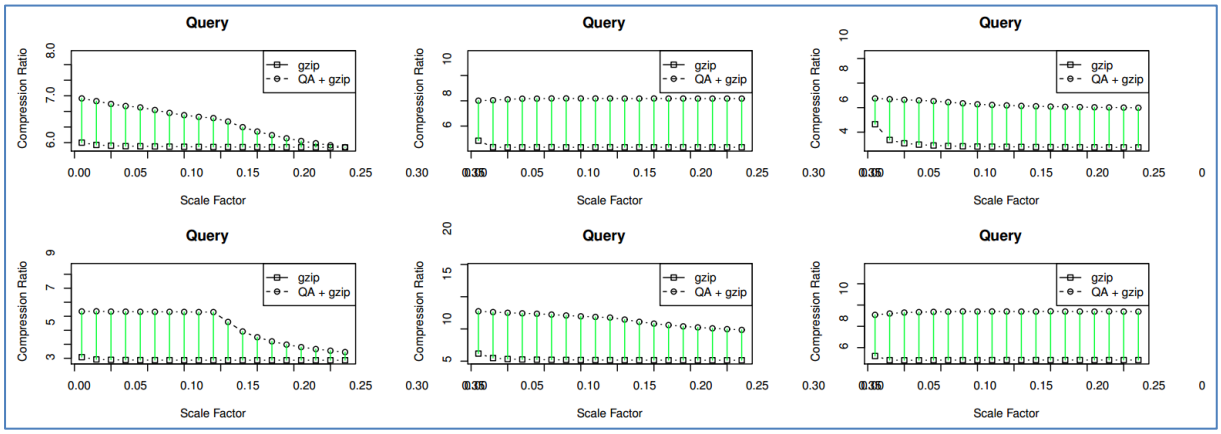
megduplázta a tömörítési arányt a gzip megoldáshoz képest. A 5-ös lekérdezés (Query 5) esetén 840MB méretről 69MB méretre tömörítette be a query-aware-gzip, ami 12-szeres tömörítési arány. Csak gzip megoldást használva az arány 5-szörös és 163MB a mérete.

Adatmennyiség vs tömörítési arány

Ezen kísérlet során az eredménysorok mennyiségét (adat mennyiséget) tekintjük változónak és a szótárméret fix marad. Ezzel megvizsgáljuk, hogy a tömörítési arány hogyan változik, ha növeljük az adatok számát. A *11. ábrán* látható kísérletben 6db lekérdezést vizsgáltunk 10ezres szótárméretttel. Tisztán látszik, hogy minél több az adat, annál kisebb a tömörítési arány, igaz kis mértékben csökken. Ez a limitált szótárméretnek köszönhető. Ha sok értéket kell kicserélni a szótárakban, az csökkenti redundanciát. A *12. ábrán* levő kísérletben ugyanazokat a lekérdezéseket vizsgáltuk, de 20ezres szótárméretttel. Látható, hogy a tömörítési arány kisebb mértékben csökken, hiszen így már nagyobbak a szótárak és kevesebbszer kell értéket cserélni.



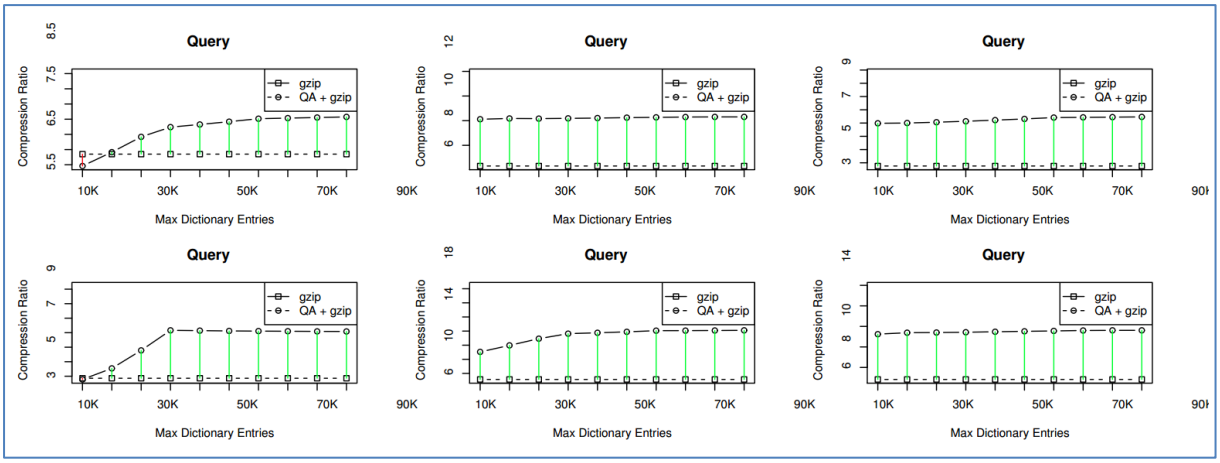
11. ábra



12. ábra

Szótárméret vs tömörítési arány

Most az eredménysorok rögzítettek és a szótárméretet változtatjuk. Ez alapján megbecsülhető, hogy az algoritmusnak mennyi memória szükséges, hogy hatékonyan tudjon működni nagy adatbázisok esetén is. *A 13. ábrán* látható az elért eredmény 0,35 arány faktor mellett. Tisztán látható, hogy a szótárméret növelésével a tömörítési arány növelhető. Ha



13. ábra

nagyon kevés a szótárméret, akkor még rosszabb eredményt is produkálhat az algoritmus, mint a sima gzip.

Összefoglalás

Az előbbiekben bemutattunk egy lekérdezés eredménysorokat tömörítő algoritmust. Az algoritmus join-olt lekérdezésekben, hatékonyabb redundancia felismerésre képes, mint egy általános tömörítő eljárás. Az algoritmus hatékonyságát mérési eredményekkel igazoltuk. A mérésekből kiderült, hogy a legjobb esetben akár kétszeres tömörítési arányt lehet elérni egy általános tömörítéshez (gzip) képest. Ennek ellenére több esetben is tapasztalhattunk nagyon kismértékű tömörítési aránynövekedést vagy éppen csökkenést. Tehát az algoritmus hatékonysága nagyban függ a lekérdezés szerkezetétől és az adatbázis szerkezetétől (kis, nagy stb.).

A negatív esetek ellenére, a módszernek lehet jövője, az értelme nem kérdőjelezhető meg. Mivel ez a cikk az első (a szerzők tudomása szerint) ami ezzel a témával/módszerrel foglalkozik, ezért még biztosan lesz továbbfejlesztése, és el fog jutni egy olyan szintre, ahol már minden esetet le fog fedni és minden lekérdezésre általánosan lehet alkalmazni, és nem csökkenti a tömörítés arányát. Ehhez negatív esetek részletesebb elemzése szükséges, és a negatív esetet előidéző okokat kell megszüntetni.

Irodalomjegyzék

[1] D. Huffman. A method for the construction of minimum redundant codes. In Proc. IRE, volume 40, 1098-1101, 1952

[2] I. H. Witten, R. M. Neal, J. G. Cleary, Aritmethic Coding for data compression. Commun. ACM, 30:520-540, 1987

[3] J. Ziv, A. Lempel. A universal algorithm for sequential data compression. IEEE Transactions on Information Theory, 23(3):337-343, 1977

[4] J. Ziv, A. Lempel. Compression of individual sequences via variable-rate coding, IEEE Transactions on Information Thepry, 24(5):530-536, 1978

[5] H. Sakamoto, T. Kida, S: Shimozono. A space-saving linear-time algorithm for grammar based compression. In SPIRE, 218-229, 2004

[6] G. V. Cormack. Data compression on a databse system. Commun. ACM, 1336-1342, 1985

[7] M. A. Roth, S. J. Van Horn. Database compression. SIGMOD Rec., 22:31-39, 1993

[8] W. Ng, C. Ravishankar. Relational database compression using augmented vector quantization. In ICDE, 540-549, 1995

[9] C.-L. Goh, K. Aisaka, M. Tsukamoto, S. Nishio. Database compression with data mining methods. In K. Tanak, S. Ghaneharizadeh, Y. Kambayashi Information Organization and Databases, volume 579 ofTHe Kluwer International Series in Engineering and Computes Science, 177-190, 2001

[10] T. P. P. Council. Tpc benchmark™, 2011