Títol: Construcció de cubs de dades usant MapReduce sobre clústers
Autor: Víctor Herrero Otal
Departament: ESSI
Volum: 1/1

Data: 22/Gener/2014

Director: Òscar Romero Moral
Departament: ESSI

Co-Director: Alberto Abelló Gamazo
Departament: ESSI

DADES DEL PROJECTE

Títol del Projecte:	Construcció de cubs de dades usant MapReduce sobre clústers
Nom de l'estudiant:	Víctor Herrero Otal
Titulació:	Enginyeria en Informàtica
Crèdits:	37.5
Director:	Òscar Romero Moral
Departament:	Enginyeria de Serveis i Sistemes d'Informació

MEMBRES DEL TRIBUNAL *(nom i signatura)*

President:	Antoni Urpí Tubella
Vocal:	Ramón Nonell i Torrent
Secretari:	Òscar Romero Moral

QUALIFICACIÓ

Qualificació numèrica:
Qualificació descriptiva:
Data:

Als meus avis, perquè tots em van veure
néixer, no tots em van veure créixer, i
cap m'ha vist arribar fins aquí.

Índex

1 Introducció

1.1 Treball previ

Aquest treball neix com la continuació del PFC: *Aplicació per fer consultes de cubs de dades usant MapReduce* realitzat per en Jaume Ferrarons Llagostera, Juny 2011.

En aquest treball anterior, en Jaume va elaborar els algorismes per a la generació de cubs de dades fent servir MapReduce i els va posar a prova sobre un petit clúster local. De forma extra a aquest treball, també va intentar explotar aquests algorismes sobre un clúster amb hardware més adient. Concretament, sobre el clúster del DAC. No obstant, les complicacions a l'hora de configurar les tecnologies emprades sobre un entorn distribuït i la falta de temps no van permetre que aquestes proves sobre el nou clúster es duguessin a terme satisfactòriament.

Aleshores, aquest projecte pretén dur a terme aquesta segona part del projecte d'en Jaume: executar aquests algorismes sobre un clúster real per tal de desenvolupar proves exhaustives de rendiment i d'escalabilitat en la construcció dels cubs de dades i trobar factors d'optimització.

1.2 Context

Actualment vivim en una era on l'Internet s'ha convertit en una eina d'ús diari per a la gran majoria de la societat. A més, com a usuaris d'Internet, esperem que la informació que se'ns retorna sigui acurada a les nostres necessitats. A canvi, de forma potser més aviat inconscient, publiquem a Internet porcions de les nostres pròpies definicions com a persones: estils de música, de roba, ideologies, on treballem, on vivim, etc. Especialment amb la sorgida de les xarxes socials. Tot això ha estat ràpidament identificat com a una nova clau d'èxit per a les companyies.

No obstant, per sort o per desgràcia, aquestes noves fonts de dades es basen en uns requisits que difícilment poden ser satisfets mitjançant les tecnologies clàssiques. Aquests nous requisits són anomenats les tres V's i defineixen el terme Big Data:

- **Volume.** Comencem a parlar de volums de dades de l'ordre de petabytes[1].
- **Variety.** Les dades que cal processar són molt més desestructurades que anteriorment i es fa impensable continuar amb la solució actual d'afegir capes software per tal de traduir les dades a un model normalitzat.
- **Velocity.** S'espera que aquestes noves fonts de dades ajudin a una presa de decisions que ha de ser molt més imminent. Presa de decisions en temps real.

La part positiva de totes aquestes noves necessitats és que suposen un nou repte tecnològic en el món de les bases de dades. Fins aleshores, les bases de dades s'havien basat en un únic model: el model relacional, que complia certes propietats (com les propietats ACID) i que tenia una definició molt concreta. Amb aquest nou entorn de treball es qüestionen aquestes propietats a l'hora de dissenyar una solució concreta; no perque hagin deixat de ser certes, sinó perque aquests nous requisits han demostrat que cada problema ha de tenir la seva pròpia solució, i que una solució requereixi d'unes propietats concretes no vol dir que la resta de solucions també hagin de satisfer les mateixes propietats. Aquest és el significat de la frase *one size does not fit all.*

[1] 1 Petabyte = 2^{50} bytes, 1 Terabyte = 2^{40} bytes

D'aquesta manera, sorgeix el terme NOSQL que intenta englobar totes aquestes solucions que s'allunyen del món relacional. A diferència del que a vegades es pensa, NOSQL no significa "No SQL", sinó que el seu significat real és "Not Only SQL", que vol dir "no només SQL". Aquest terme és en realitat un sinònim del *one size does not fit all* ja que el significat que hi ha darrere no és abandonar l'SQL[2], sinó aprofitar només allò que pugui servir per solucionar el nostre problema i tot allò que no solucionar-ho a partir d'una altra banda, però no convertir cap solució en universal.

Tot això ha derivat en el desenvolupament de noves tecnologies per tal de donar suport a aquest nou concepte. D'entre les diferents solucions, la més popular sens dubte és Hadoop. Aquesta tecnologia és la que s'utilitza en aquest projecte i es podria definir com una plataforma per treballar amb sistemes distribuïts. El terme Hadoop en si és massa general. Hadoop en realitat és un conjunt de tecnologies que donen suport al sistemes distribuïts. Així, triar la tecnologia Hadoop que més útil hauria d'anar en funció del propòsit del sistema.

De la mateixa manera, també han hagut de sorgir noves formes de consultar aquestes bases de dades. El MapReduce és una d'aquestes formes. A diferència de llenguatges declaratius com l'SQL per exemple, el MapReduce és més aviat un framework de programació. Això és degut principalment a la barreja de dos factors clau en el món NOSQL:

1. Les dades no tenen perquè haver patit una normalització prèvia i per tant és el programador qui ha de definir, en temps de processat o de recuperació de les dades, quina forma és la adequada per a cada consulta.

2. Moltes d'aquestes noves bases de dades estan pensades principalment per a treballar en distribuït i per tant calen nous paradigmes de programació basats en el *dividir i vèncer*, de forma que trenquin el problema en subproblemes més reduïts (p.ex: obtenció de dades de forma local a cada node), i facin l'agregació final amb les solucions de cada un d'aquests subproblemes.

Al llarg del contingut d'aquesta memòria s'entrarà més en detall en la definició i funcionament del MapReduce, però per tal d'ajudar al lector a comprendre com MapReduce compleix amb els dos punts anteriors, cal mencionar que MapReduce són essencialment dues funcions que són les que han de ser implementades pel programador: el map i el reduce. El map és bàsicament la conversió d'una entrada trencada en fragments en un conjunt de parelles **key-value**. Un cop generades totes aquestes parelles de key-value, es fa una ordenació de les mateixes per tal de fer una agrupació de les parelles amb la mateixa key. Aleshores, aquest conjunt resultant es converteix en l'entrada del reduce, que és qui fa l'agregació final. Com que és el propi programador qui ha d'implementar aquestes dues funcions, ja es contempla que sigui el programador l'encarregat de donar la forma adequada a les dades en aquest moment.

Així doncs, les bases de dades es troben en un moment de molta heterogeneïtat tecnològica, on tothom vol aportar la seva pròpia solució. A més, tota aquesta heterogeneïtat fa que aparegui molta terminologia nova amb termes amb significats molt semblants entre ells i altres amb significats massa generals.

Aleshores, aquest projecte neix amb l'objectiu d'estudiar aquestes noves bases i els nous llenguatges de consultes mitjançant proves exhaustives que permetin determinar el seu comportament sota diferents paràmetres i, de forma empírica, ens ajudin a trobar patrons d'optimització.

[2]Concretament, SQL és el llenguatge declaratiu per a treballar amb les bases de dades relacionals, però en aquest apartat l'utilitzem com a sinònim directe del món relacional.

1.3 Motivació

Al llarg de la carrera, he tingut la oportunitat de fer assignatures que m'han introduit en diferentes aspectes de la informàtica. Moltes d'elles han aconseguit produïr en mi la motivació per tal d'especialitzar-me en aquell àmbit tot escollint com a optatives, aquelles assignatures que aprofundien en els aspectes que més m'interessaven. Aquests aspectes han estat essencialment les bases de dades i les arquitectures concurrents i distribuïdes, i dintre d'aquestes àmbits, la optimització d'aquests sistemes.

És clar, doncs, que aquest projecte em dona la oportunitat de treballar seguint aquestes línies de més interès personal ja que en aquest projecte hi participen cada un d'aquests aspectes més destacats, i a més, amb l'objectiu d'optimitzar el rendiment.

Per altra banda, aquest projecte també em dona la oportunitat de participar en una línia d'investigació del departament ESSI. Això em permet involucrar-me un pèl més en el món de la recerca i espero que m'aporti experiències que repercuteixin positivament en la meva maduresa com a futur enginyer ja que el sentit crític que hi haurà d'haver darrere de cada decisió presa haurà de ser clau per tal de que els resultats finals siguin innovadors.

En definitiva, per a mi aquest projecte és la possibilitat d'explotar els coneixements adquirits al llarg de la carrera, principalment en els àmbits de més interès mencionats anteriorment, tot col·laborant en un projecte de recerca i, per tant, nodrint-me de noves experiències. D'aquesta manera, espero que aquest projecte es converteixi en la meva petita aportació a la recerca i em permeti adquirir coneixements que siguin clau per al meu futur professional.

1.4 Objectius

El conjunt d'objectius que defineixen aquest projecte i marquen quin és el resultat final que s'espera són els detallats a continuació:

a) Objectius del sistema:

- El sistema ha de funcionar sobre el clúster del DAC i s'ha d'ajustar als requisits d'aquest.
- El sistema ha de permetre la creació de cubs de dades OLAP sobre un clúster Hadoop i emprant MapReduce.
- El sistema ha d'implementar fins a tres algorismes d'accés a les dades.

b) Objectius de les proves:

- El primer objectiu és el de comprendre el funcionament correcte de les tecnologies emprades per tal de dur a terme el segon objectiu.
- Aquest objectiu és el principal, però és conseqüència de l'anterior. Consisteix en trobar un model empíric d'optimització que permeti decidir sota quines circumstàncies l'accés a les dades és més òptim si es fa d'una forma o una altra. Per tal de dur a terme aquest marc d'optimització, caldrà executar un conjunt de proves que tindran els següents subobjectius:
  - Les proves hauran de valorar diferents paràmetres que es classificaran essencialment en tres grups:
    * **Tipologia de la consulta:** S'avaluaran els diferents paràmetres que formen part d'una consulta.

    - ∗ **Tipologia de les dades:** S'avaluaran diferents graus de dispersió i de volum de les dades a consultar.
    - ∗ **Tipologia del sistema**: Es posaran a prova diferents paràmetres de configuració del sistema.
  - Les proves hauran de valorar l'eficiència de cada algorisme implementat.
  - Finalment, les proves han de fer-se a partir de la combinació dels diferents paràmetres i dels algorismes implementats per tal d'analitzar, de forma independent, un conjunt de situacions òptimes que finalment permetin definir el model d'optimització.
- Finalment, el tercer i últim objectiu és aconseguir derivar dels resultats de les proves anteriors les formules dels costos de cada algorisme per tal de reforçar el model d'optimització construït a partir de factors de més baix nivell, com ara costos de lectura de disc, de xarxa...

2 Conceptes bàsics

2.1 OLAP i OLTP

En el món de les bases de dades, es poden trobar dos grans grups de tecnologies que es diferencien principalment per l'objectiu pel qual estan pensades. Es tracta de les bases de dades OLTP i OLAP. De forma general, es pot assumir que els OLTP proveeixen dades mentre que els sistemes OLAP ajuden a analitzar-les. La figura *2.1-1*[3] il·lustra la interacció entre aquestes dues tecnologies.

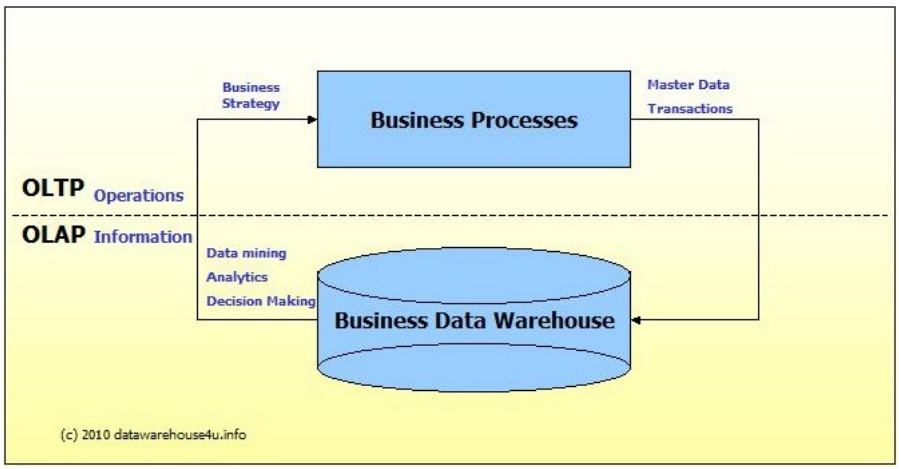


Figura 2.1-1: OLAP vs. OLTP

- **OLTP *(On-Line Transaction Processing).*** Aquesta tecnologia es caracteritiza pel gran volum de petites transaccions amb les que opera. Aquestes petites transaccions estan formades principalment per les operacions d'inserció, modificació i esborrat. Així doncs, es tracta d'un tipus de tecnologia operacional. Les bases de dades OLTP generalment emmagatzemen dades actualitzades, i l'esquema utilitzat és el model de negoci.
- **OLAP *(On-Line Analytical Processing).*** En canvi, OLAP es caracteritza per treballar amb un gran volum de consultes que acostumen a ser complexes i involucren agregacions. Així, el cost d'execució d'aquestes consultes es fa elevat i el temps de resposta es converteix en una mesura d'avaluació efectiva d'aquest tipus de tecnologia. Les bases de dades OLAP contenen dades històriques, representades a partir d'un esquema multidimensional (generalment estrella).

2.2 Data Warehouse (DW)

Un Data Warehouse és una base de dades amb l'objectiu de ser una eina de consulta i d'anàlisis de les dades que emmagatzema més que una eina d'operacions. Aquesta base de dades acostuma a ser poblada a partir de la integració de dades provinents de diferents fonts d'informació. Degut a aquest fet, les dades amb les que poblar aquesta base de dades poden tenir estructures diferents i per aquest motiu cal que primerament passin un procés de transformació ETL (veure apartat *2.5 Extract, Transform, Load (ETL)*).

[3]Font: http://datawarehouse4u.info/OLTP-vs-OLAP.html

Aquests Data Warehouse solen estar relacionats amb dades històriques ja que estan pensades per a la realització d'informes i així, per exemple, facilitar la presa de decisions en el món dels negocis. Degut a aquestes característiques, els Data Warehouse són bases de dades més pensades per a les consultes que no pas per a les transaccions i per tant conformen una base de dades de tipus OLAP.

2.3 Esquemes multidimensionals: Esquema en estrella

En les tecnologies OLAP, és comú l'elaboració d'esquemes de dades més fitats per a la resposta ràpida de consultes. El més típic d'aquests esquemes és l'esquema en estrella.

A l'esquema en estrella, les dades transaccionals són particionades en el que s'anomena **taula de fets**, on s'hi emmagatzema generalment dades numèriques. Un segon tipus de taula son les **dimensions**, que contenen informació que aporta context a les dades contingudes per la taula de fets. Un exemple senzill podria entendre's com una taula de fets que conté dades com el preu o el nombre de productes adquirits, i les dimensions podrien ser la zona geogràfica, la data, etc. És degut a aquesta referència de la taula de fets amb les dimensions, el motiu pel qual aquest tipus d'esquemes s'anomenen esquemes multidimensionals.

D'aquesta manera, una consulta amb l'objectiu d'obtenir el nombre total de productes adquirits a Europa es podria fer a partir de la taula de fets i de la dimensió de la zona geogràfica. La figura *2.3-1*[4] mostra l'estructura d'un esquema d'aquest tipus.

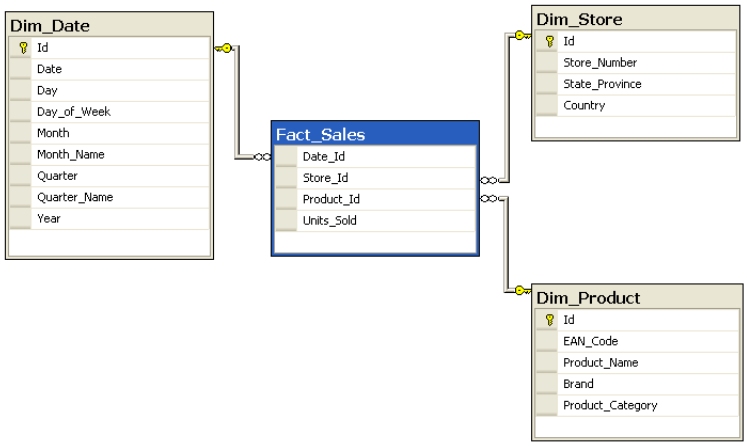


Figura 2.3-1: Exemple d'un esquema en estrella

2.4 Cub de dades

Un cub de dades (o formalment un cub de dades OLAP) és una estructura de dades que permet representar cada una de les dades obtingudes de la taula de fets a partir de les dimensions demanades. És senzillament un vector de dades entès en termes de zero o més dimensions[5].

A partir de lo explicat a l'apartat *2.3 Esquemes multidimensionals: Esquema en estrella*, és fàcil veure com la taula de fets es converteix en cada una de les caselles del

[4] http://en.wikipedia.org/wiki/Star_schema
[5] http://en.wikipedia.org/wiki/OLAP_cube

cub i les dimensions es converteixen en les pròpies dimensions del cub. La figura *2.4-1*[6] mostra el dibuix típic d'un cub de dades.

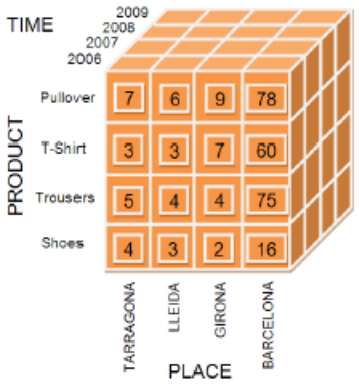


Figura 2.4-1: Exemple cub de dades

En terminologia de cubs de dades, es poden definir el següent dos conceptes a partir de l'esquema en estrella definit a l'apartat *2.3 Esquemes multidimensionals: Esquema en estrella*:

1. **Mesura.** Les mesures són cada un dels atributs que conformen la taula de fets i que són seleccionats per analitzar. Es tracta, doncs, d'atributs tals com la quantitat de vendes, el número d'habitants, etc.

2. **Nivell d'agregació.** Una mateixa dimensió pot tenir diferents atributs, però que es relacionen entre ells segons una determinada jerarquia. L'exemple més clar és una data: una data està composta per un dia, un mes i un any. Així doncs, cada un d'aquests elements és el que s'anomena nivell d'agregació. Els nivells d'agregació més elevats són aquells on els atributs tenen una granularitat més gran (en aquest cas és l'atribut any). Un altre exemple típic pot ser una zona geogràfica, que es pot trencar en els nivells d'agregació (de menys a més nivell d'agregació): ciutat, comarca, país i continent.

A més, aquests cubs de dades poden aplicar diferents operacions o transformacions per tal de presentar les dades d'una forma o una altre. A continuació es descriuen aquestes transformacions[7]:

- **Selection.** Permet obtenir un subconjunt de punts d'interès a partir d'un conjunt multidimensional.

- **Roll up/Drill down.** Agrupen caselles segons la jerarquia dels nivells d'agregació de les dimensions. La operació *roll up* agrupa en nivells d'agregació més grans. La operació *drill down* és el contrari, agrupa en nivells d'agregació més petits.

- **Change base.** Aquesta operació implica una relocalització de les mateixes dades però en un nou espai multidimensional.

- **Drill across.** Permet modificar el contingut de les dades analitzades mantenint el mateix espai multidimensional.

[6] Alberto Abelló, Oscar Romero. *Automating the Multidimensional Design of Data Warehouses.* 2009
[7] Peu de pàgina 6

- **Projection.** Permet seleccionar un subconjunt de mesures.
- **Set Operations.** Aquest subconjunt d'operacions permet combinar dos cubs de dades si tots dos estàn definits sobre el mateix espai multidimensional. Aquest subconjunt d'operacions conté la unió, la intersecció i la diferència, entre d'altres.

Finalment, la figura *2.4-2* il·lustra cada una de les operacions definides anteriorment.

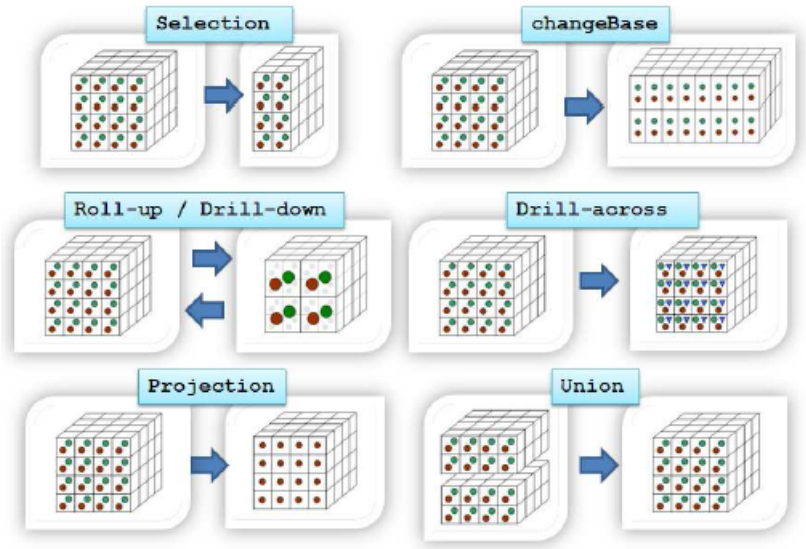


Figura 2.4-2: Transformacions aplicables a un cub de dades

2.5 Extract, Transform, Load (ETL)

ETL són les sigles de Extract, Transform and Load. Tal i com el propi nom suggereix, les eines ETL tenen per objectiu:

- **Extract.** Extracció de les dades de fonts externes.
- **Transform.** Transformació d'aquestes dades per a que encaixin en el model de la base de dades.
- **Load.** Càrrega de les dades en la base de dades.

Cal mencionar que ETL sol ser definit com un procés de tres fases, però en realitat el terme ETL indica un únic procés.

A continuació es detalla un exemple d'aplicació d'ETL. Suposem que el departament de Recursos Humans d'una organització fa servir unes eines basades en DBase[8] i FoxPro[9] per a la recaptació de nous empleats, però les dades d'empleats s'emmagatzemen en una base de dades Microsoft SQL Server[10]. Per tant, la font a partir de la qual volem extreure les dades seria semblant a la mostrada a la figura *2.5-1*:

[8] http://en.wikipedia.org/wiki/DBase
[9] http://en.wikipedia.org/wiki/FoxPro
[10] http://www.microsoft.com/en-us/sqlserver/default.aspx

NAME	STREET	CITY	STATE	ZIP	DEAR WHO	TEL HOME	BIRTH DATE	HIRE DATE	INSIDE
Guiles, Makenzie	145 Meadowview Road	South Hadley	MA	01075	Macy	(413)555-6225	19770201	20060703	yes
Forbes, Andrew	12 Yarmouth Drive	Holyoke	MA	01040	Andy	(413)555-8863	19600330	20060710	yes
Barra, Alexander	4327 Spring Drive	Sandwich	MA	02537	Al	(508)555-8974	19861221	20060717	no
Mellon, Philip	123 Pendexter Street	Boston	MA	01321	Phil	(781)555-4289	19700625	20060724	no
Clark, Pamela	44 Mayberry Circle	Bedford	MA	01730	Pam	(860)555-6447	19500502	20060731	yes

Figura 2.5-1: Exemple d'aplicació d'ETL

- La fase d'extracció obtindrà les dades del fitxer en DBase i el convertirà en un format més adient.
- El procés de transformació adaptarà la data a un format estàndard, separarà el nom en nom i cognoms i assignarà el cap apropiadament segons si l'empleat és assignat al departament de ventes.
- Finalment el procés de càrrega introduirà les dades al Microsoft SQL Server.

2.6 Denormalització

La denormalització és el procés que pretén eliminar les foreign keys d'una base de dades amb una estructura normalitzada per tal d'incrementar-ne el rendiment augmentant-ne la redundància de les dades[11].

Les foreign keys permeten encapsular les dades que formen un objecte que representa un element del món real en estructures de dades independents entre elles sense perdre la capacitat de poder unir aquestes estructures per tal de recuperar aquest objecte. Les figures *2.6-1* i *2.6-2*[12] mostren una mateixa situació amb un esquema normalitzat i un desnormalitzat. D'aquesta manera, també permeten comprovar la funció de les foreign keys que es representen a les figures a partir de fletxes.

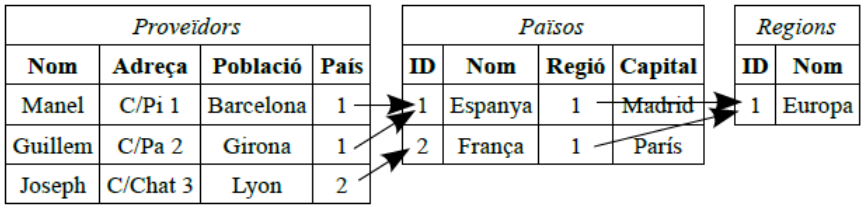


Figura 2.6-1: Exemple d'un esquema normalitzat

Nom	Adreça	Població	Nom país	Capital	Regió
Manel	C/Pi 1	Barcelona	Espanya	Madrid	Europa
Guillem	C/Pa 2	Girona	Espanya	Madrid	Europa
Joseph	C/Chat 3	Lyon	França	París	Europa

Figura 2.6-2: Exemple d'un esquema desnormalitzat

D'aquesta manera, és fàcil veure com per tal de traduir d'un esquema a una altre, i per tant aplicar denormalització, cal aplicar operacions *join*.

[11] Candace C. Fleming, Barbara Von Halle. *Handbook of Relational Database Design.* 1989

[12] Ferrarons Llagostera, Jaume. *Aplicació per fer consultes de cubs de dades usant MapReduce.* Juny 2011

Durant el procés de denormalització es poden produir duplicats i redundància de dades. A les figures *2.6-1* i *2.6-2.* això ocorre amb les dades de regions. Com que en el model normalitzat cada objecte es trenca en estructures de dades independents, no cal repetir valors ja que si per exemple en aquest cas, dos proveïdors son de la mateixa regió, és suficient amb fer que tots dos referenciïn a la mateixa regió. En el cas de l'exemple desnormalitzat, això no és possible ja que, al no haver foreign keys, no és possible trencar un mateix objecte en estructures diferents i per tant, dos objectes diferents amb el mateix valor en un atribut conformaran un cas de redundància de dades.

3 Requisits funcionals i no funcionals

3.1 Quins requisits?

En aquest apartat es detallen els requisits que defineixen la implementació d'aquest projecte. Com és habitual en el món de l'enginyeria del software, es diferencia entre requisits funcionals i no funcionals. Els funcionals són aquells que defineixen exactament què ha de fer el sistema, com ha de funcionar. Els requisits no funcionals són més generals i marquen quines tecnologies cal fer servir, amb quins altres tipus de sistema s'ha de comunicar, etc.

La figura *3.1-1* mostra un petit diagrama d'aquests requisits, classificats en funcionals i no funcionals. L'objectiu d'aquesta figura és mostrar les relacions entre els diferents requisits ja que, malgrat ser independents entre ells, el bon compliment de certs requisits va permetre ampliar l'àmbit del projecte i això es va traduir en l'aparició de nous requisits.

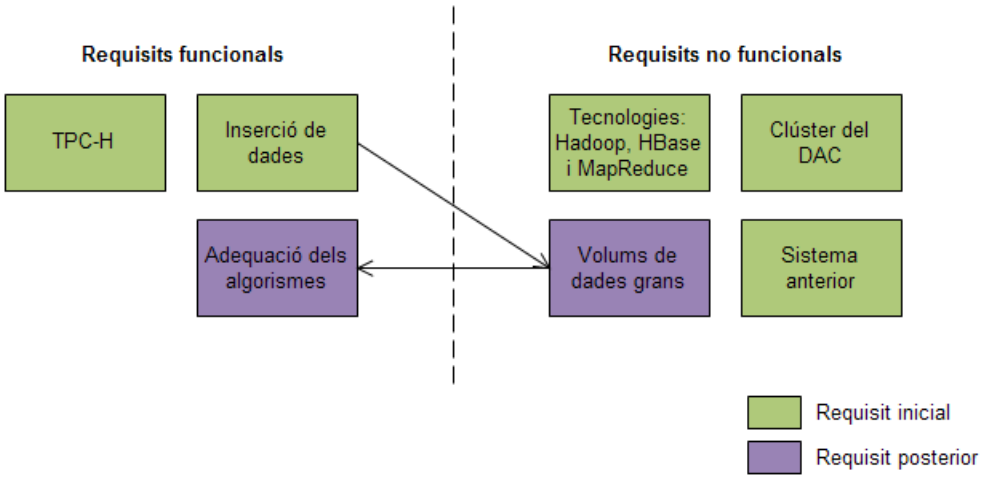


Figura 3.1-1: Conjunt de requisits del sistema

Per tal de permetre lligar millor uns requisits amb els altres, es defineix una petita nomenclatura que permet identificar cada un d'ells. Aquesta nomenclatura és de la forma:

[F | NF]*número_identificació*: *text_descriptiu*

on els caràcters F i NF indiquen si es tracta, respectivament, d'un requisit funcional o no funcional, el número d'identificació és únic per cada requisit dins del grup funcional/no funcional, i finalment el text descriptiu és opcional.

Així, el conjunt de requisits es registre de la següent manera:

- **F1: Inserció de dades**
- **F2: TPC-H**
- **F3: Adequació dels algorismes**
- **NF1: Tecnologies Hadoop, HBase i MapReduce**
- **NF2: Clúster del DAC**
- **NF3: Sistema anterior**
- **NF4: Volums de dades grans**

3.2 Requisits funcionals

El conjunt de requisits funcionals es basa principalment en aquells aspectes del treball anterior a partir del qual apareix aquest projecte que en el seu moment no van acabar de funcionar correctament. Així, aquests requisits tracten de trobar implementacions alternatives que funcionin correctament i que permetin dur a terme els consegüents objectius redactats a l'apartat *1.4 Objectius*.

3.2.1 Requisit F1: Inserció de dades

El sistema ha de garantir la correcta inserció de les dades. Aquest és un requisit heretat del projecte anterior ja que el principal problema que va existir en el seu moment es devia al procés de denormalització (veure apartat *2.6 Denormalització*), necessari per tal d'inserir les dades en el format adequat.

No obstant això, aquest procés de denormalització és només un pas intermig degut a l'enfoc que se li va donar al procés inserció de dades. En aquest sentit, el requisit en aquest projecte es converteix simplement en **inserir les dades d'una forma alternativa** per tal que les proves es puguin dur a terme seguint els objectius marcats pel projecte. Aleshores, es dóna llibertat de decidir quin procés d'inserció és més adequat per tal de garantir els objectius. D'aquesta manera, la denormalització queda al marge de la decisió de si forma part d'aquest enfoc més adequat o no.

3.2.2 Requisit F2: TPC-H

Aquest ja era un requisit en el treball anterior, però en qualsevol cas, degut a que el requisit **F1** indica que el procés d'inserció s'ha de fer diferent, aquest continua sent un requisit d'aquest projecte, ja que s'ha de fer tot seguint les línies del TPC-H.

3.2.3 Requisit F3: Adequació dels algorismes

Amb les modificacions en el procés d'inserció i el consegüent nou requisit de treballar amb volums de dades grans va sorgir aquest últim requisit funcional. Aquest és degut principalment a que originalment els algorismes desenvolupats en el projecte anterior no estaven pensats per treballar amb els volums de dades amb els quals aquest projecte sí pot treballar a partir dels requisits **F1** i **NF4**.

3.3 Requisits no funcionals

3.3.1 Requisit NF1: Tecnologies Hadoop, HBase i MapReduce

Les tecnologies a emprar en aquest projecte són: Hadoop, HBase i MapReduce. Aquest és un requisit imposat pel client.

3.3.2 Requisit NF2: Clúster del DAC

Aquest requisit és possiblement el més important de tots. Es tracte de que el sistema desenvolupat sigui capaç de funcionar correctament sobre les màquines del clúster del DAC. Així, la importància d'aquest requisit ve donada pel fet que el clúster del DAC està format per màquines sobre les que s'hi pot llençar proves realment exhaustives. A més, en cas de no satisfer aquest requisit, no s'hi té accés a cap altre arquitectura física que sigui tan adequada per aquest projecte com és el clúster del DAC, i per tant la resta de requisits deixen de ser vàlids.

3.3.3 Requisit NF3: Sistema anterior

Un altre requisit imposat pel client va ser continuar amb el sistema anterior desenvolupat en el projecte previ a aquest.

3.3.4 Requisit NF4: Volums de dades grans

Gràcies als bons resultats mostrats per la implementació del requisit **F1**, el procés d'inserció es va poder aconseguir en un temps molt reduït la qual cosa va permetre encaminar el projecte cap a un nou requisit conseqüència d'aquest.

Aquest requisit consisteix en permetre treballar amb volums de dades més grans dels inicialment esperats. Inicialment es plantejava la opció de seguir amb els volums de dades del treball anterior, però amb el nou procés d'inserció és fàcilment viable arribar a volums de dades molt més grans. Concretament, s'espera poder doblar com a mínim els volums de dades anteriors.

4 Arquitectura del sistema

4.1 Arquitectura lògica

4.1.1 Descripció de l'arquitectura

En aquest apartat s'il·lustra l'estructura lògica del sistema dissenyat. Bàsicament aquest està compost per les tecnologies emprades que conformen l'estructura necessària per tal d'arrencar la base de dades. Aquestes tecnologies són:

- **HDFS.** Sistem de fitxers distribuït.
- **HBase.** Base de dades.
- **MapReduce.** Accés de consulta a les dades.

En realitat, però, a aquests elements cal sumar-hi les implementacions que han calgut fer per tal poder poblar la base de dades i construir els cubs de dades. Aquestes implementacions les definirem com:

- **Generador.** Aquesta és l'aplicació que realitza un determinat volum d'insercions a la base de dades.
- **Consultes.** Amb aquest nom ens referim a l'aplicació que fa les consultes necessàries per tal de construir els cubs de dades.

Un cop presentats de forma breu tots els elements que formen part de l'arquitectura lògica del sistema, només falta presentar com es connecten entre ells.

Així doncs, la figura *4.1-1* mostra amb més detall la connexió entre tots aquests elements. El color blau indica totes dues aplicacions, mentre que la resta de colors indiquen elements totalment diferents entre ells.

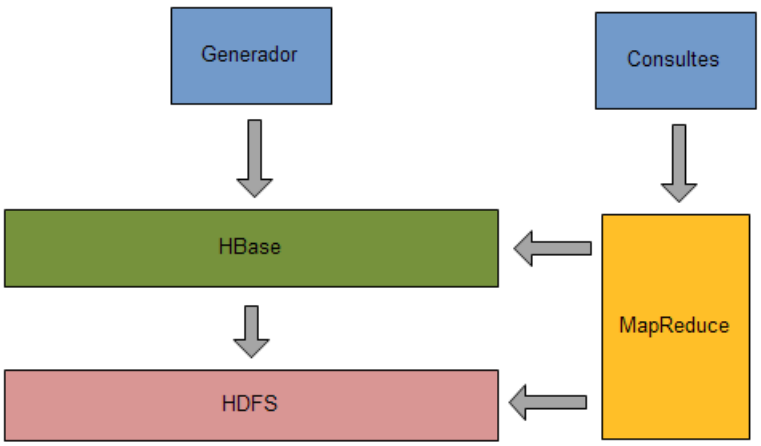


Figura 4.1-1: Arquitectura lògica del sistema

Quant a les implementacions, totes elles accedeixen al sistema a partir de punts diferents:

- El generador accedeix a partir del propi HBase ja que insereix les tuples directament sense necessitat d'accedir al MapReduce. No obstant això, es podria haver dissenyat

per tal d'accedir primerament al MapReduce, però com que aquest segon està pensat per processar dades ja inserides doncs realment no hagués tingut cap efecte positiu en la inserció, sinó que més aviat hagués estat negatiu, ja que les hagués alentit malgrat el paral·lelisme. Això és degut a que en aquest cas el coll d'ampolla es troba en el propi HBase.

- Les consultes sí cal que accedeixin a partir del MapReduce, ja que, ara sí, cal fer un processament de les dades ja inserides.

Les tecnologies emprades s'estructuren de forma tal que la base de dades ha d'estar en un nivell superior al sistema d'arxius, ja que aquest ha de ser transparent per a la base de dades: la base de dades rep una nova operació, la processa a nivell lògic i l'envia al sistema de fitxers per tal de materialitzar-la.

El cas del MapReduce es troba en un punt mig entre l'HBase i l'HDFS. Bàsicament perquè aquest es comunica amb tots dos i en tots dos sentits. MapReduce inicialment llegeix la informació de l'HBase per tal de saber quins son els grups de dades i els fitxers que ha de llegir, però seguidament opera directament sobre el sistema de fitxers. És per aquest motiu pel qual se'l situa en una capa intermitja entre les altres dues tecnologies.

Així doncs, aquesta és la estructura lògica del sistema i com es posicionen els diferents elements que la formen. De totes maneres, en aquest apartat es mencionen certs aspectes del sistema molt superificialment. Per tal d'obtenir més informació sobre el funcionament intern de les tecnologies emprades, així com els detalls de les implementacions fetes, es poden revisar els apartats *4.2 Arquitectura de les tecnologies emprades* i *7 Implementació*, respectivament.

4.1.2 Versions utilitzades

Les versions utilitzades per a les diferents tecnologies usades són:

- **Hadoop (HDFS i MapReduce):** Versió 1.0.4
- **HBase:** Versió 0.94.4
- **Java (JDK[13] i JRE[14]):** Versió 1.6.0 update 36

4.2 Arquitectura de les tecnologies emprades

4.2.1 HDFS

Introducció

HDFS són les sigles de *Hadoop Distributed File System*. Tal i com el seu propi nom indica, es tracta del sistema de fitxers distribuït implementat per la comunitat Apache en relació al sistema de fitxers distribuït utilitzat per Google i anomenat GFS (*Google File System*).

Tal i com moltes arquitectures distribuïdes, l'HDFS fa servir una arquitectura màster-esclau per tal de centralitzar en un sol node aquella informació que no pot ser distribuïda. Generalment perquè es tracta d'informació del sistema. Aleshores, la nomenclatura que segueixen aquests nodes és:

[13] Java Development Kit
[14] Java Runtime Environment

- **NameNode:** Per referir-se al node màster.
- **DataNode:** Per referir-se als nodes esclaus.

Addicionalment es pot definir el que s'anomena un **Secondary NameNode** que realitza tasques de backup del màster.

A la figura *4.2-1* es veu el funcionament intern de l'HDFS. Tal i com es menciona anteriorment, existeixen dos tipus de nodes: el màster i els esclaus.

- El NameNode s'encarrega de centralitzar aquella informació que es pròpia del sistema i que per tant no pot ser distribuïda, a la vegada que monitoritza l'estat de les màquines esclaves.
- Els DataNodes emmagatzemen les dades.

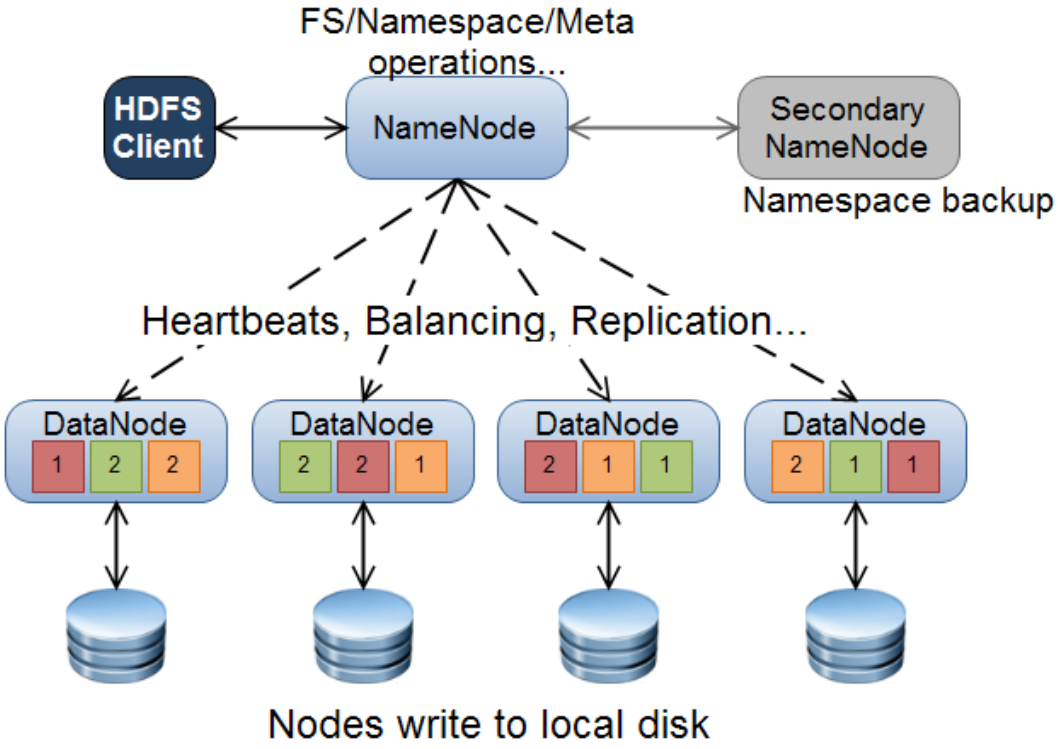


Figura 4.2-1: Estructura de funcionament de l'HDFS

De fet, les dades pròpies de l'usuari només poden ser emmagatzemades en els DataNodes. En aquest sentit, el NameNode fa principalment una tasca de gestió dels nodes esclaus.

Quan un client vol escriure un fitxer, aquest es trenca en blocs de mida fixa. Aquest blocs són materialitzats a disc de forma distribuïda, de forma que cada un d'aquests blocs s'emmagatzema en una màquina diferent. Finalment per motius de disponbilitat i de temps de resposta aquests blocs es repliquen en màquines diferents.

A la figura *4.2-1*, els colors indiquen un mateix fitxer mentre que els números indiquen el número de bloc. Així, es pot comprovar com cada DataNode conté una rèplica d'un bloc de cada fitxer.

Aleshores, quan un client vol fer la lectura d'un fitxer, no té perquè fer-ho tot accedint als mateixos nodes on es va fer l'escriptura d'aquests blocs, sinó que se'n pot aprofitar de les rèpliques. D'aquesta manera, es guanya en rendiment perquè s'aconsegueix distribuir les lectures al llarg de diferents nodes amb l'objectiu de reduir la latència tot escollint aquelles rèpliques més properes, i es guanya en disponibilitat perquè la mateixa dada està en diferents nodes i per tant en cas de fallida d'un node hi haurà algun altre que podrà fer la lectura del bloc.

Així, la característica més important d'aquest sistema de fitxers és aquest últim pas: la gestió dels blocs de forma distribuïda i la replicació d'aquests. Aquest serà l'aspecte en el que s'entrarà en més profunditat sobre aquesta tecnologia.

Blocs i replicació

Tal i com es comenta al subapartat d'introducció, la forma d'emmagatzemar els fitxers a l'HDFS és trencar aquests fitxers en blocs més petits i distribuir-los al llarg del clúster. Realment, aquest concepte ja forma part dels sistemes de fitxers de disc. La diferència fonamental és que els blocs de disc s'emmagatzemen sempre a la mateixa màquina ja que no formen part de cap entorn distribuït. A més, en disc els blocs acostumen a ser molt més petits (mides properes als 4K) en comparació als sistemes de fitxers distribuïts, on per exemple els blocs de l'HDFS ocupen per defecte 64MB. No obstant això, aquests valors són perfectament configurables, però permeten introduir les similituds i les diferències entre els dos entorns.

No obstant això, aquests dos sistemes de fitxers no són comparables, ja que interactuen en diferents nivells. Per exemple, de l'HDFS se'n diu un sistema de fitxers distribuït quan en realitat és una capa intermitja que permet gestionar els diferents nodes que el formen, però en qualsevol cas, les dades sempre acaben sent materialitzades sobre el sistema de fitxers sobre el que corre el sistema operatiu dels nodes.

La qüestió és que des del punt de vista de l'usuari, l'HDFS és un sistema de fitxers distribuït amb una mida de bloc per defecte de 64M, i que garanteix la replicació d'aquests blocs en diferents nodes.

Així doncs, l'HDFS trenca els fitxers en blocs de mida 64M (per defecte) i els escriu al llarg dels diferents DataNodes. El motiu de que aquests blocs siguin tan grans en comparació als blocs de disc és minimitzar el cost de moure el capçal. Fent els blocs lo suficientment grans, el guany és que, donada una mida fixa de dades a llegir, el cost de fer les lectures d'aquesta manera és menor que el cost de buscar l'inici del bloc, ja que s'espera que la lectura de cada bloc sigui seqüencial. Així el temps de transferir un fitxer gran és més proper al rati de transferència de dades.

Un exemple senzill és que si el temps de moure el capçal és proper als 10ms, i el rati de transferència és de 100 MB/s, aleshores per fer que la latència del capçal sigui un 1% del temps de transferència total, cal que el tamany del bloc estigui al voltant dels 100MB.

La figura *4.2-2*[15] mostra com s'estructura tot aquest conjunt de dades al llarg de diferents nodes.

Aquesta figura mostra l'exemple d'un sol fitxer distribuït al llarg de diferents DataNodes. Els números indiquen el número de bloc del fitxer. Així la concatenació dels blocs *1-2-3-4-5* conforma el fitxer sencer. Per altra banda, tal i com es pot veure, cada un d'aquests blocs està replicat 2 o 3 vegades, però en cap cas dues rèpliques estan al mateix node.

[15] https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html#Data+Replication

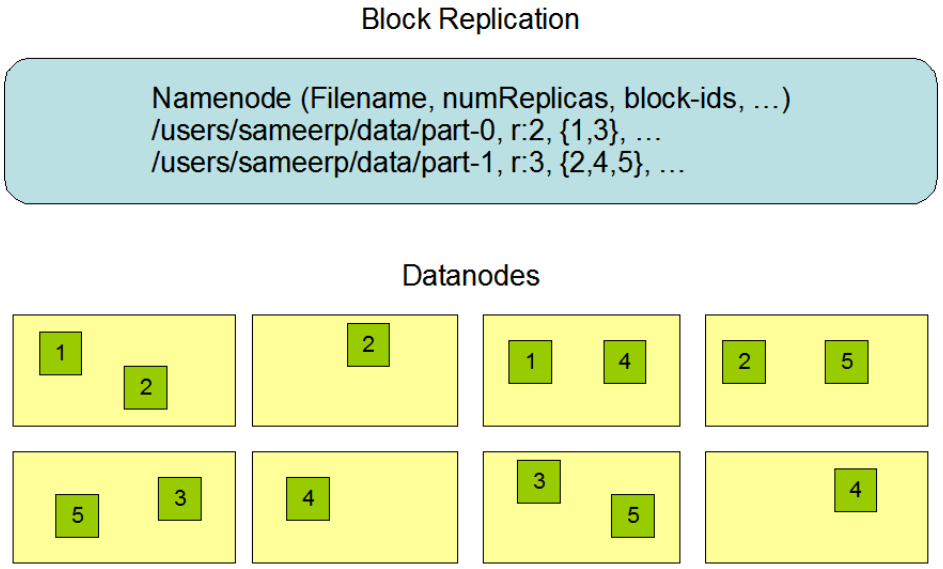


Figura 4.2-2: Distribució i replicació d'un fitxer HDFS

Aquesta és la idea darrere de l'HDFS. D'aquesta manera, si un DataNode qualsevol cau i per tant és perd l'accés als blocs que conté, queda garantit que l'accés a un altre DataNode determinat oferirà els blocs demanats.

Safemode

Durant el procés d'arrencada, el node màster s'inicia en un estat anomenat **safemode**. La replicació dels blocs mai pot ocórrer durant aquest estat. Una de les funcions del NameNode és rebre el heartbeat i la informació de bloc per part dels DataNodes. Aquesta informació conté, entre d'altres coses, un llistat dels blocs de cada DataNode. A més, cada bloc té un nombre determinat de rèpliques. Aleshores, un bloc se'l considera correctament replicat quan el NameNode té constància de l'existència d'aquest nombre mínim de rèpliques.

Així, aquest estat de safemode s'inicia durant el procés d'arrencada, i no s'atura fins que el NameNode té constància de que un cert percentatge de tots els blocs està correctament replicat. En aquest moment l'HDFS abandona aquest estat, però en cas de que encara quedin blocs sense el número mínim de rèpliques, aleshores el NameNode fa la replicació de cada un d'aquest. El tant per cent de blocs que cal tenir correctament replicat és un paràmetre de configuració de l'HDFS.

XCievers

Un altre aspecte de l'HDFS són els **xcievers**, que correspon al nombre màxim de fitxers que un DataNode pot servir. En altres paraules, quan un DataNode ha de servir un fitxer, aquest crea un nou thread encarregat de gestionar tot aquest procés. Mentre el fitxer estigui obert, el thread existeix. Aleshores, existeix aquest paràmetre que regula el nombre de threads màxim que un DataNode pot córrer en un moment donat, i a la vegada, com que cada thread gestiona un únic fitxer, també indica el nombre màxim de fitxers que pot servir en aquest mateix instant.

Aquest no és un paràmetre remarcable de l'HDFS, però s'introdueix en aquest apartat perquè va ser necessari ampliar-lo per aquest projecte.

4.2.2 HBase

Introducció

HBase és la implementació Apache del DBMS[16] BigTable desenvolupat per Google. Es tracta d'una base de dades no relacional, lo qual significa que ja no parteix de la solució fins ara universal del món relacional sinó que fa ús d'una solució pròpia. Concretament, segueix un model key-value on l'únic tipus d'atribut són els strings. Aquesta solució és ideal per Google ja que s'ajusta al problema d'indexació de pàgines web: per cada URL (key) és molt fàcil obtenir el codi HTML corresponent (value).

A més, un valor s'identifica per la combinació d'una **family**, un **qualifier** i una **version**[17] (generalment les versions són un timestamp). La figura *4.2-3* mostra amb més deteniment aquesta estructura interna.

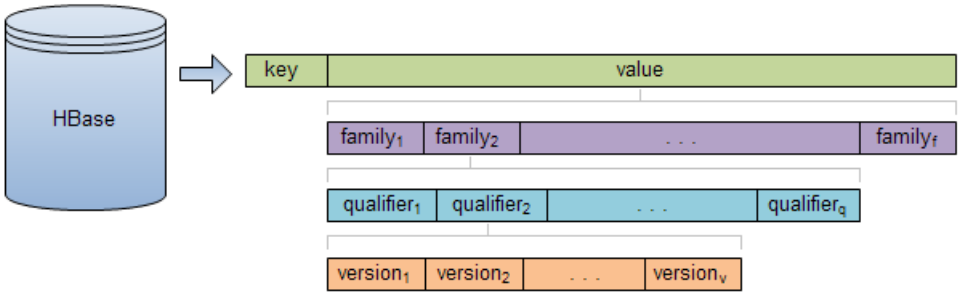


Figura 4.2-3: Estructura interna de l'HBase

Així, si posem per exemple un esquema d'immobiliàries, podríem tenir una taula amb la family *casa* a la qual li podrien pertànyer els qualifiers *preu* o *superfície*. Cada un d'ells formaria una columna que s'identificaria seguint el patró *family:qualifier*, de forma *casa:preu* i/o *casa:superfície*. A més, aquestes columnes podrien tenir diferents versions per tal de mantenir un històric de les modificacions de l'atribut. Per exemple, la primera versió de la columna *casa:preu* podria tenir el valor 250000 en una tupla qualsevol, però una revaloració d'aquesta casa podria fer que en una segona versió més recent, aquest mateix atribut valgués 200000.

Quant a l'estructura dels nodes, en HBase hi ha dos tipus de nodes: el màster i l'esclau (anomenat **RegionServer**). En termes d'especificacions tècniques cal dir que el màster no requereix de tant espai físic com un RegionServer ja que les dades només s'emmagatzemen aquests esclaus (veure apartat *4.2.2 Fragmentació horitzontal: Autosharding*). El màster, en canvi, és l'encarregat de centralitzar les metadades del clúster, com per exemple quines dades hi ha en cada RegionServer.

A més, a diferència de les bases de dades relacionals, les dades no tenen perquè estar sempre físicament emmagatzemades a disc. Primerament, les dades queden allotjades en un buffer a memòria volàtil anomenat **memstore**. Posteriorment, quan aquest buffer està ple, les dades es materialitzen a disc en fitxers anomenats **storefile** i es buiden els memstore.

Per altra banda, a partir d'aquesta estructura interna, l'HBase conté altres propietats que es discuteixen en els apartats posteriors. Tanmateix, només es discuteixen aquelles

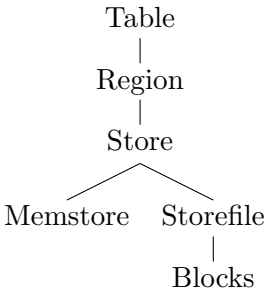
[16] DataBase Management System
[17] Nomenclatura anglesa pròpia de l'HBase

característiques vitals per a aquest projecte i s'obvien totes aquelles que es mantenen al marge.

Jerarquia de l'HBase

De forma semblant a altres sistemes d'emmagatzematge, l'HBase fa servir un conjunt de terminologies i d'arquitectures per disposar les dades físicament a disc d'una determinada forma.

Aquesta jerarquia comprèn la següent definició:



- **Table:** En HBase, s'entén per taula el mateix concepte que al món relacional. Es podria definir com un simple conjunt de dades organitzat a partir de files i columnes.
- **Region:** Aquesta és la unitat més bàsica d'escalabilitat i de distribució de la càrrega. Una region es pot entendre com una porció de tuples emmagatzemades en un únic node.
- **Store:** Un store és la unitat que emmagatzema, per cada region, el conjunt de dades corresponents a una única family.
  - **Memstore:** Els memstores són buffers a memòria on es guarden inicialment les dades de l'HBase.
  - **Storefile:** Quan un d'aquests memstore està ple, es buida salvant les dades a disc en nou fitxer anomenat storefile.
    - **Blocks:** Aquests són els blocs d'un storefile on es guarden les dades al més baix nivell físic.

La figura *4.2-4* posa en comú totes aquestes definicions respecte la jerarquia màster-esclau definida anteriorment.

Per altra banda, també cal mencionar part de la terminologia que envolta aquesta arquitectura:

- **Compactacions:** Les compactacions busquen reduir el nombre de storefiles existents. Aquestes compactacions s'executen amb certa periodicitat i en concret són:
  - **MINOR:** Aquesta compactació intenta agrupar varis storefiles en un de sol. Fa un intent d'execució cada cop que es realitza el buidat d'un memstore i la consegüent construcció d'un nou storefile. Si es donen certes condicions, aleshores té lloc una compactació d'aquest tipus. Aquestes condicions són:

$$B_{storefile} \leq (\sum B_{storefiles\_més\_petits}) * rati$$

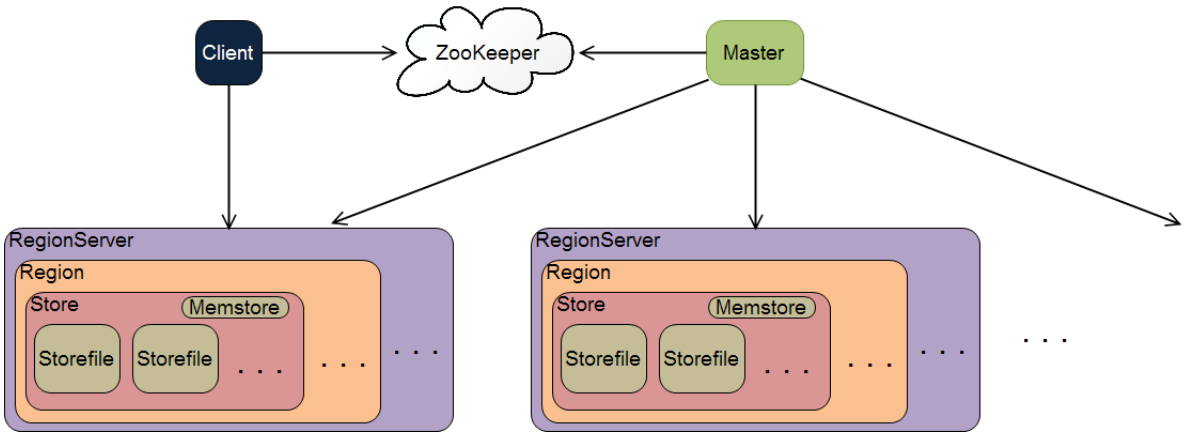


Figura 4.2-4: Arquitectura de funcionament de l'HBase

On B és l'espai ocupat per cada storefile en bytes.

A continuació es mostra un exemple extret de la documentació oficial de HBase[18] que il·lustra el procediment per tal de determinar quins storefiles han de ser compactats. Suposem que tenir la següent configuració:

a) Rati de compactació (hbase.store.compaction.ratio) = 1.0f
b) Mínim de storefiles a compactar (hbase.hstore.compaction.min) = 3
c) Màxim de storefiles a compactar (hbase.hstore.compaction.max) = 5
d) Mínim nombre de bytes a compactar (hbase.hstore.compaction.min.size) = 10
e) Màxim nombre de bytes a compactar (hbase.hstore.compaction.max.size) = 1000

I que existeixen cinc storefiles que ocupen respectivament el següent: 100, 53, 23, 12 i 12 bytes.

En aquesta situació es compactarien els storefiles 23, 12 i 12, ja que la formula es compleix en aquests casos:

* 100 → No, perquè $(50 + 23 + 12 + 12) * 1.0 = 97 \leq 100$
* 50 → No, perquè $(23 + 12 + 12) * 1.0 = 47 \leq 50$
* 23 → Sí, perquè $(12 + 12) * 1.0 = 24 \geq 23$
* 12 → Sí, perquè l'storefile previ ha estat inclòs i encara no s'ha superat el màxim de storefiles a compactar.
* 12 → Sí, perquè l'storefile previ ha estat inclòs i encara no s'ha superat el màxim de storefiles a compactar.

– **MAJOR:** Aquesta compactació agrupa tots els storefiles d'un sol store en un únic storefile. Aquesta compactació es molt costosa i es realitza de forma periòdica a menys que es produeixi degut a una compactació MINOR promocionada a MAJOR. Aquesta promoció es produeix quan una compactació MINOR ha d'agrupar tots els storefiles d'un mateix store.

- **Flush:** Es parla de flush quan els memstores es buiden i les seves dades es passen a un storefile nou.

[18] http://hbase.apache.org/book/regions.arch.html

Clustered index

La localitat de les tuples es fa tot seguint un **clustered index** distribuït. Un clustered index és un tipus d'índex, existent també en el món relacional, que consisteix bàsicament en un arbre B+ on les últimes fulles indexen directament a les tuples, que ja es troben físicament ordenades a disc. En el cas de l'HBase, aquesta ordenació és lexicogràfica i es fa a través de la key[19] que se li assigna a cada tupla. A més, l'arbre B+ pot tenir com a màxim profunditat 3. La figura *4.2-5* il·lustra l'estructura típica d'un clustered index.

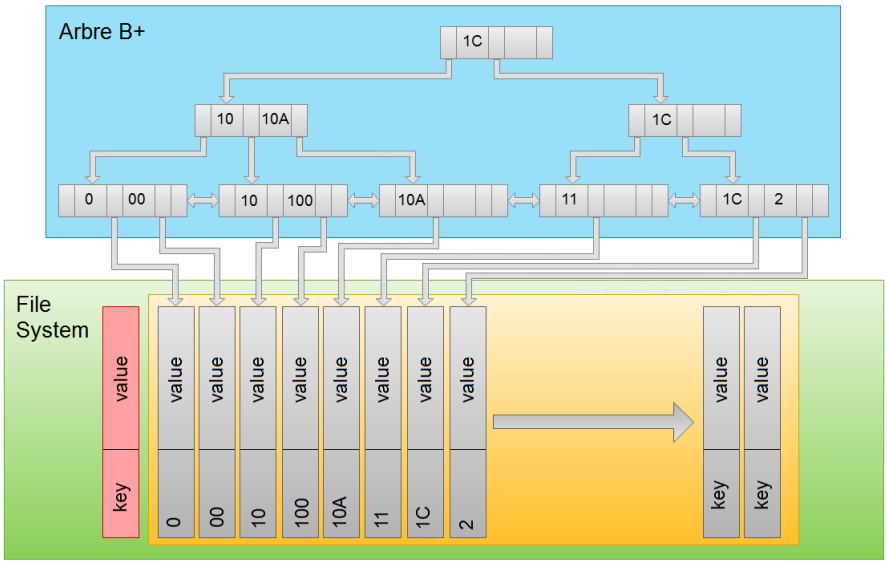


Figura 4.2-5: Estructura d'un *clustered index*

Una propietat dels arbres B+ és tal que donat un ordre d:

- El nombre màxim de claus en un registre és d.
- El nombre mínim de claus per registre és $d/2$.

A l'exemple de la figura *4.2-5*: $d = 2$, per lo que podem comprovar que compleix ambdues condicions.

Fragmentació horitzontal: Autosharding

Tal i com s'ha definit anteriorment, una region és la unitat més bàsica d'escalabilitat i d'equilibri de la càrrega. De forma addicional, aquestes regions no tenen perquè ser estàtiques, en el sentit de que es poden distribuir al llarg del clúster, es poden unir per reduir-ne el nombre total i/o es poden trencar en regions més petites.

La unitat que dóna servei a aquestes regions són els RegionServers. Una region només pot ser hostatjada per un RegionServer, però en canvi un RegionServer pot hostatjar múltiples regions. La figura *4.2-6* il·lustra un exemple de diferents *regions* hostatjades per RegionServers.

Aquesta forma d'assignar diferents tuples a diferents RegionServers recorda a la fragmentació horitzontal del món relacional. De fet, els termes fragmentació horitzontal i

[19] En aquest cas, al ser ordenació lexicogràfica, les keys numèriques s'han de transformar a una longitud fixa. Per exemple la key 25, es converteix en la key 0000000025.

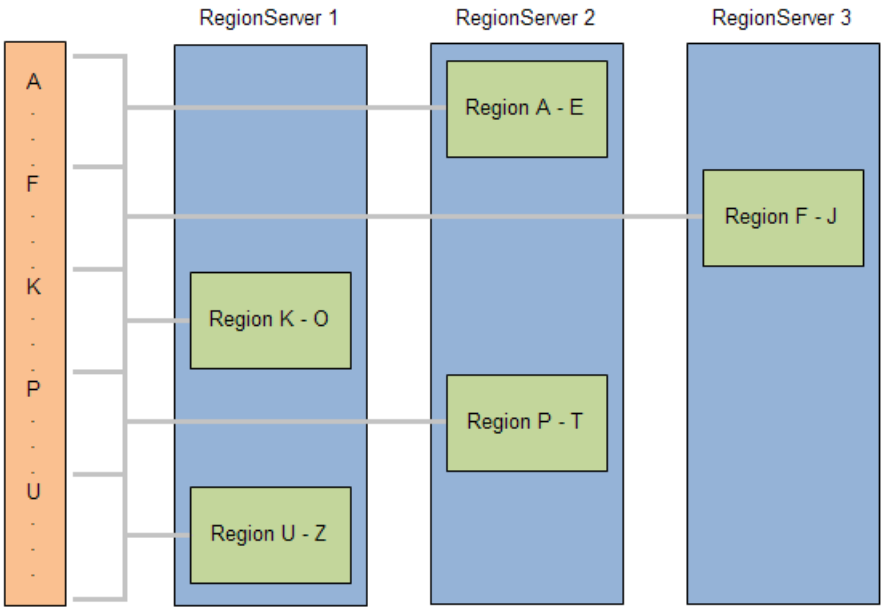


Figura 4.2-6: Fragmentació horitzontal dels RegionServers

sharding denoten el mateix concepte, i donat que aquesta fragmentació es fa de forma automàtica per tal d'equilibrar la càrrega de les diferents regions, aquest procés s'anomena autosharding.

Aquest procés es realitza a través del que s'anomena **autosplit**. Inicialment, l'HBase no sap quin es el volum de dades que emmagatzemarà i per tant la seva forma de procedir és carregar una region fins que es donin certes condicions, i llavors dur a terme una divisió (split o region split d'ara endavant) d'aquesta region en dues, de manera que es mantingui la fragmentació horitzontal actualitzada. Aquestes condicions que indiquen que una region ha de ser fragmentada s'anomenen *Split Policies* i l'HBase proporciona tres:

a) **ConstantSizeRegionSplitPolicy:** Aquesta política produeix un split quan un (qualsevol) storefile ocupa igual o més espai que un paràmetre predeterminat (per defecte són 10 GB).

b) **IncreasingToUpperBoundRegionSplitPolicy:** Aquesta és un política més agressiva que l'anterior. De la mateixa forma, produeix un split de la region quan un (qualsevol) storefile ocupa igual o més espai que el marcat per la formula:

$$split = min(R^2 hbase.hregion.memstore.flush.size, hbase.hregion.max.filesize)$$

On *hbase.hregion.memstore.flush.size* i *hbase.hregion.max.filesize* són els paràmetres que defineixen el tamany dels memstores i el tamany màxim d'un storefile, respectivament, i R és el nombre de regions assignades al mateix RegionServer que la region que ha de fer split. D'aquesta manera, es pot comprovar com el segon paràmetre de la funció és, en efecte, un limitador al creixement exponencial del volum de les regions amb aquesta política. Tal i com s'indica en la política anterior, el valor per defecte de *hbase.hregion.max.filesize* és 10 GB, i per *hbase.hregion.memstore.flush.size* és 128 MB. Tot això indica que, en un mateix RegionServer:

1. Primer split (R = 1): $min(R^2 128 \text{ MB}, 10240 \text{ MB}) = 128 \text{ MB}$
2. Segon split (R = 2): $min(R^2 128 \text{ MB}, 10240 \text{ MB}) = 512 \text{ MB}$
3. Tercer split (R = 3): $min(R^2 128 \text{ MB}, 10240 \text{ MB}) = 1152 \text{ MB}$
4. Quart split (R = 4): $min(R^2 128 \text{ MB}, 10240 \text{ MB}) = 2048 \text{ MB}$
5. Cinquè split (R = 5): $min(R^2 128 \text{ MB}, 10240 \text{ MB}) = 3200 \text{ MB}$
6. Sisè split (R = 6): $min(R^2 128 \text{ MB}, 10240 \text{ MB}) = 4608 \text{ MB}$
7. Setè split (R = 7): $min(R^2 128 \text{ MB}, 10240 \text{ MB}) = 6272 \text{ MB}$
8. Vuitè split (R = 8): $min(R^2 128 \text{ MB}, 10240 \text{ MB}) = 8192 \text{ MB}$
9. Novè split (R = 9): $min(R^2 128 \text{ MB}, 10240 \text{ MB}) = 10240 \text{ MB}$

Efectivament, això indica que calen nou splits amb la configuració per defecte per tal de que aquesta segona política apliqui el valor màxim d'un storefile com a límit d'una region. És clar, doncs, que aquesta política és molt més agressiva que la primera.

c) **KeyPrefixRegionSplitPolicy:** La tercera i última política proporcionada per l'HBase es basa en les keys. Aquesta política pren una cadena de caràcters com a prefix per a un conjunt de keys, i agrupa en cada region aquelles keys que comencin amb el prefix establert.

En addició, cal també mencionar que l'HBase permet implementar una política de fragmentació pròpia.

La figura *4.2-7* intenta fer una comparativa de com creix el nombre de regions amb les dues primeres polítiques. Clarament, la primera política segueix un model lineal mentre que la segona política, tal i com s'explica anteriorment, es comporta com una funció definida en dues parts: la primera segueix un model logarítmic, i la segona part és igual que la política anterior. L'objectiu d'aquesta figura és il·lustrar quant de més agressiva és realment la segona política respecte la primera. En aquest sentit, és fàcil observar com en el mateix instant de temps (76), la segona política ha generat ja un total de nou regions mentre que l'altra política tan sols ha fet un split.

Per altra banda, la figura *4.2-8* mostra amb més deteniment com funciona aquest procés d'autosplit. El procediment aquí representat es basa principalment en la primera i segona política explicades anteriorment ja que són les que es basen en la grandària d'un storefile. A més, la política utilitzada en aquest projecte és la segona ja que és la predeterminada a la versió de l'HBase que s'ha fet servir.

- **Fase 1:** Aquesta part de la figura mostra una única region on hi arriben noves insercions. A mesura que es materialitzen aquestes insercions el nombre de storefiles creix, ja que després de cada flush dels memstores es crea un storefile nou. En aquest moment, l'HBase llença un intent de compactació MINOR i si es compleixen les condicions, aquesta compactació es produeix. És aleshores quan el volum dels storefiles comença a créixer. Finalment, quan un d'aquests storefiles creix lo suficient, es produeix un region split.

- **Fase 2:** Els storefiles són immutables. En altres paraules, els storefiles no poden canviar de RegionServer. Aleshores, quan es produeix un region split, les dades no es mouen físicament d'un RegionServer a un altre, sinó que es crea un storefile punter sobre el punt de split entre les dues regions.

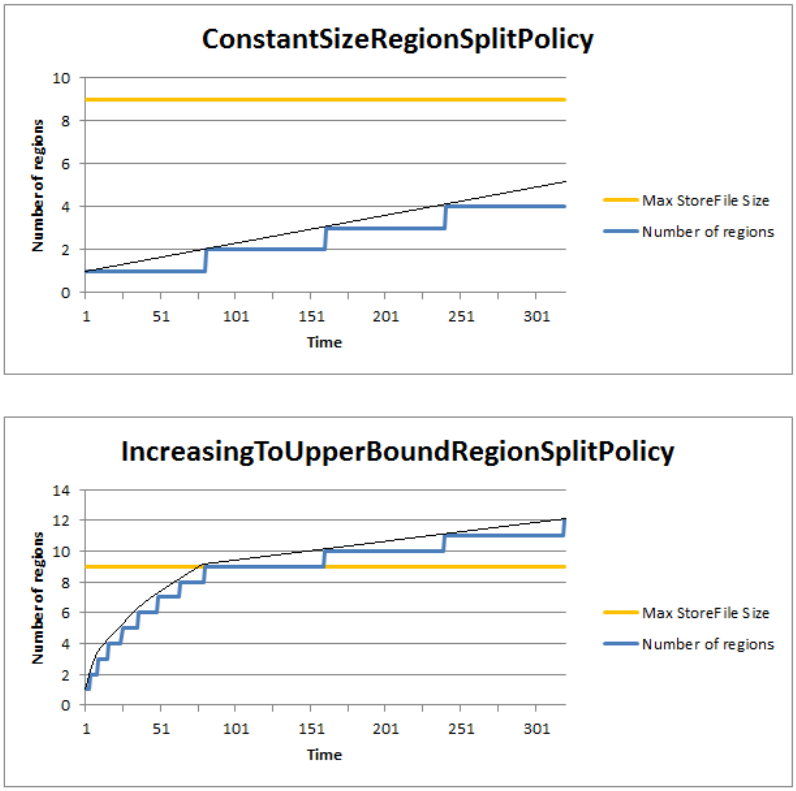


Figura 4.2-7: Creixement del nombre de regions respecte dues polítiques diferents

- **Fase 3:** Amb el region split dut a terme, continuen arribant noves insercions. Moltes d'elles hauran de ser materialitzades a la nova region. És en aquest moment en el que, de manera semblant a la primera fase d'aquest procés, es llencen compactacions MINOR sobre aquesta nova region que agrupen storefiles. Aleshores, és en el moment d'agrupar els storefiles punters, que les dades canvien físicament de RegionServer i aquests storefiles desapareixen.

- **Fase 4:** Finalment, les tasques de neteja del màster són les encarregades de detectar i eliminar aquells storefiles que havien estat referenciats i eliminar-los degut a que ja no hi ha cap referència sobre ells.

Fragmentació vertical: *Families*

La unitat de localitat de dades amb granularitat més baixa és el concepte de column family. Una **column family** permet emmagatzemar juntament a disc totes les columnes que pertanyen a una family determinada. Aquest concepte recorda a la fragmentació vertical del món relacional.

Quan creem una nova taula a l'HBase, és necessari definir exactament el nom de les column family que tindrà. Posteriorment, les columnes que pertanyen a cada column family poden ser definides *on-the-fly*. En aquest sentit, podem definir l'HBase com una base de dades *schemaless*.

Una column family és de fet un terme massa genèric. La realitat és que l'HBase permet definir families que contindran qualifiers, i la combinació de les dues dóna lloc al que en el món relacional es coneix com una columna. El terme column family engloba totes les

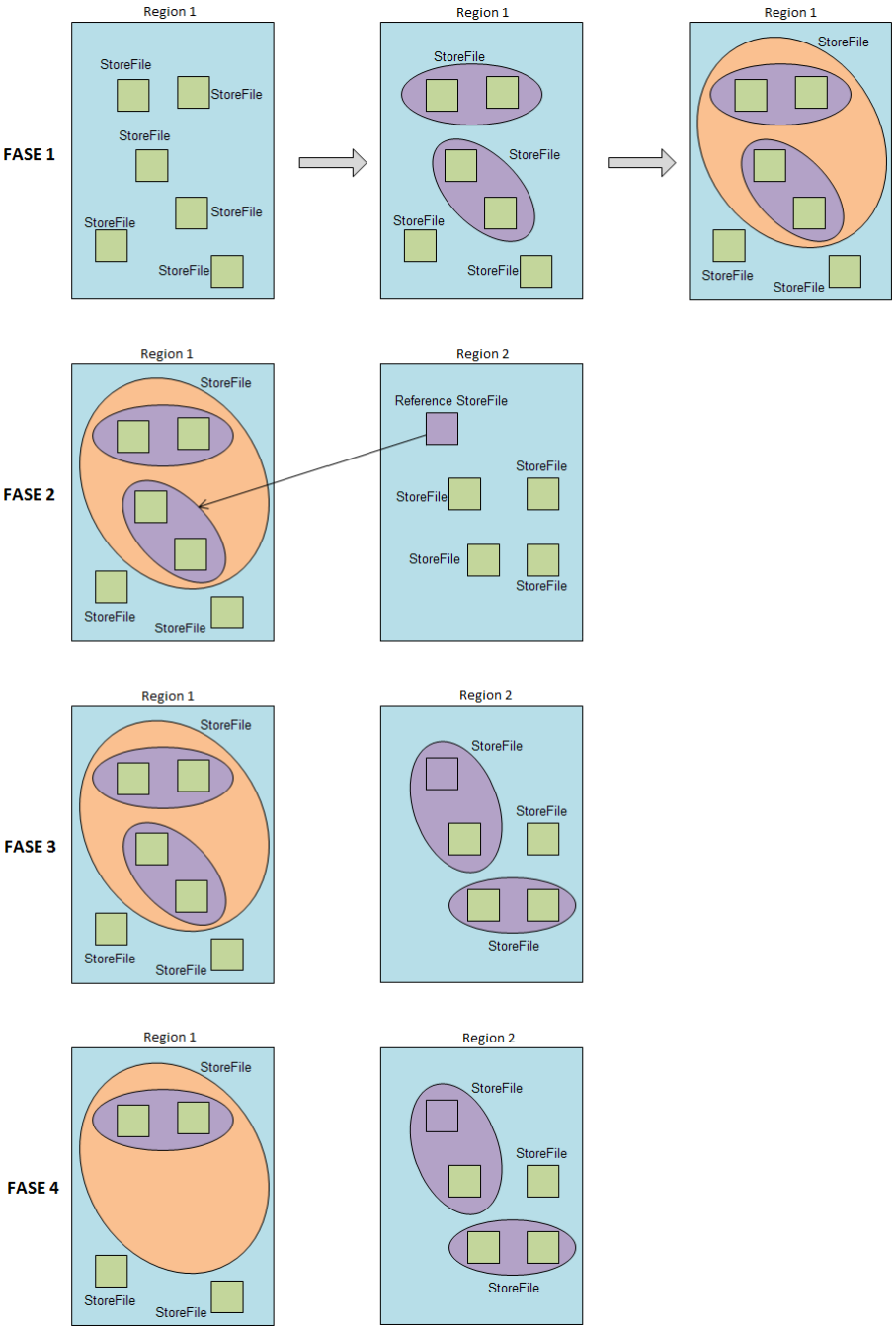


Figura 4.2-8: Procés d'autosplit de l'HBase

parelles *family:qualifier* d'una mateixa family.

De forma addicional, l'HBase permet emmagatzemar fins a un determinat nombre de versions per a un mateix atribut. És a dir, una parella *family:qualifier* pot tenir diferents versions. Això vol dir que, donada una tupla, el valor és la conjunció de cada una d'aquestes subestructures $valor = family + qualifier + version$

Aleshores, l'estructura física que permet assignar aquestes dades de forma seqüencial a disc és l'storefile. Dins de cada storefile s'emmagatzemen tots els qualifiers de la family corresponent. Aquest emmagatzematge es fa tot seguint una ordenació lexicogràfica d'aquests qualifiers respectant, prèviament, l'ordre lexicogràfic de les keys (veure punts anteriors *4.2.2 Clustered index* i *4.2.2 Fragmentació horitzontal: Autosharding*). Cada valor s'emmagatzema dins d'una cel·la que a la vegada conté la key, la family, el qualifier i la versió, per tal de poder ser fàcilment identificat.

La figura *4.2-9* mostra amb més detall aquesta estructura dins d'un storefile.

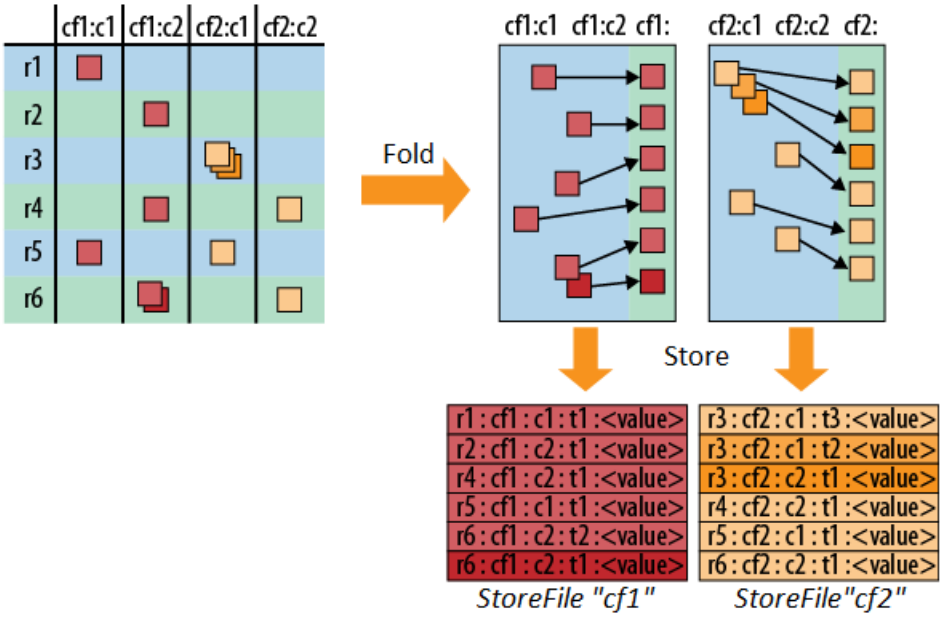


Figura 4.2-9: Estructura d'emmagatzemament de les *column families*

És important destacar com en aquesta mateixa figura *4.2-9*, les versions més noves queden per sobre de les més velles. Això és així per tal d'estalviar temps d'accés, ja que d'aquesta forma els primers valors llegits seran els valors més recents. En aquest mateix diagrama s'identifiquen les versions amb una lletra t inicial ja que les diferents versions d'un mateix atribut acostumen a ser timestamps.

WAL: Write-Ahead Logging

El WAL és senzillament un sistema que escriu al log abans de materialitzar l'escriptura. D'aquesta manera, si la màquina cau, l'escriptura queda registrada en aquest mateix log i és possible recuperar-la quan la màquina torna a donar servei.

Buffer d'escriptura (client-side): Autoflush

Cada connexió entre un client i l'HBase té assignat un buffer local al client on s'emmagatzemen les insercions abans d'enviar-les contra l'HBase. Llavors, existeixen dues

opcions: a) enviar les tuples una a una, o b) esperar a omplir el buffer i llavors enviar-les totes de cop. Això és l'autoflush. Quan l'autoflush està activat (això és així per defecte), les insercions es fan seguint el model a), altrament es segueix el model b). Per tant, l'avantatge de desactivar l'autoflush és que s'aprofita molt més l'ample de banda de la connexió amb l'HBase i per tant, temps d'execució que abans era latència de xarxa, ara desapareix.

Quan l'autoflush està desactivat i s'han d'enviar totes les tuples contingudes en el buffer del client contra l'HBase, primerament es fa una ordenació d'aquestes tuples. D'aquesta manera, li és més fàcil al client saber quin subconjunt de tuples van a cada RegionServer. La figura *4.2-10* representa aquest procés.

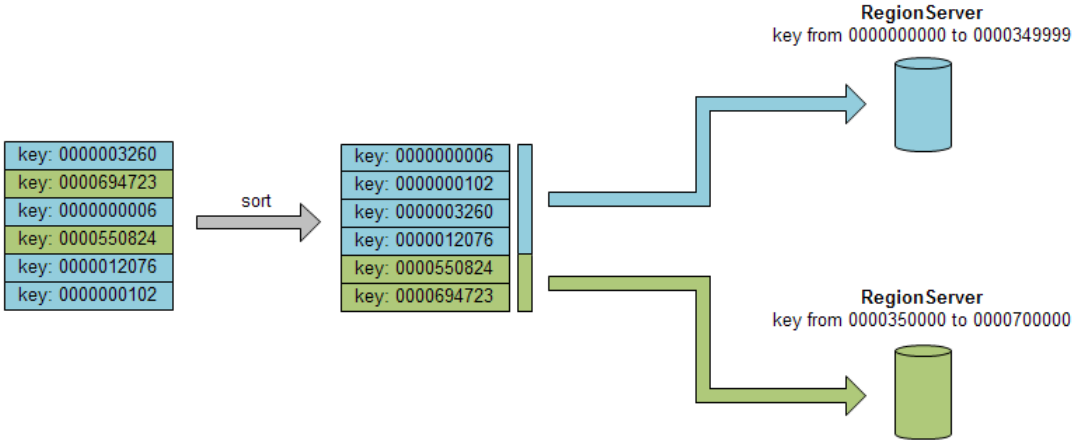


Figura 4.2-10: Procés d'inserció amb l'autoflush desactivat

No obstant això, aquesta ordenació significa que s'afegeix un nou cost de CPU degut a aquesta ordenació, però aquest cost és molt més menor que la latència de xarxa. De totes maneres, com a mera anotació, cal notar que podria arribar un punt en el que inserir amb l'autoflush desactivat afegís més overhead a l'execució. Fixem-nos en els següents temps d'inserir K tuples, essent K el nombre de tuples que hi ha en el buffer d'un mateix client:

$$T_{autoflush} = K(T_{generacio} + T_{xarxa} + T_{HBase})$$
$$T_{\overline{autoflush}} = K(T_{generacio} + T_{HBase}) + T_{xarxa} + T_{ordenacio}$$

on $T_{generacio}$ i T_{HBase} són valors constants que indiquen el temps de generar una tupla i el temps de fer l'escriptura de la mateixa a l'HBase. També podem suposar que el temps d'ordenació és menor que la caiguda de la latència de la xarxa entre els dos models:

$$T_{ordenacio} < KT_{xarxa} - T_{xarxa}$$

i que per tant, el temps d'execució amb l'autoflush desactivat és menor que no pas en el cas contrari. Malauradament, aquesta formula no és totalment certa. La formula del cost amb l'autoflush desactivat fa una petita suposició que en realitat no hauria de fer, i és que no s'està tenint en compte que a l'hora d'enviar les tuples contra l'HBase, és possible que no totes elles vagin contra el mateix RegionServer. Aleshores, el temps total d'inserció té un petit cost addicional. Per exemple, si la meitat de les insercions anessin contra un determinat RegionServer, i l'altra meitat contra un RegionServer diferent, el temps total seria:

$$T_{\overline{autoflush_2}} = K(T_{generacio} + T_{HBase}) + 2T_{xarxa} + T_{ordenacio}$$

o contra tres RegionServers diferents:

$$T_{\overline{autoflush_3}} = K(T_{generacio} + T_{HBase}) + 3T_{xarxa} + T_{ordenacio}$$

Per tant, és fàcil veure que el temps real de produir K tuples amb l'autoflush desactivat és:

$$T_{\overline{autoflush_R}} = K(T_{generacio} + T_{HBase}) + RT_{xarxa} + T_{ordenacio}$$

on R és el nombre total de RegionServers on es fan insercions. Aleshores, si suposem el cas en que a l'hora d'enviar les insercions per la xarxa, tenim que cada una de les tuples del buffer va a un RegionServer diferent de la resta, tenim que $R = K$ i per tant la formula del temps ja simplificada resulta en:

$$T_{\overline{autoflush_K}} = K(T_{generacio} + T_{HBase} + T_{xarxa}) + T_{ordenacio}$$

La qual cosa implica que en alguns casos l'autoflush acaba perjudicant el temps total d'execució:

$$T_{\overline{autoflush_K}} > T_{autoflush}$$

Segurament no calgui arribar a tal extrem per trobar una situació en la que l'autoflush perjudica a l'execució en general, ja que si recordem prèviament fèiem la suposició que el temps d'ordenar el buffer era menor que la caiguda de la latència de xarxa. En aquest cas, el que està succeint és que no hi ha caiguda del temps de xarxa entre el model d'autoflush activat i desactivat degut a que cada tupla va a un RegionServer diferent (és com enviar-les una a una), i per tant la suposició inicial que havíem fet deixa de ser certa. Però segurament aquesta suposició ja es trenqui amb un valor per R menys extremista.

Compressions

La base de dades HBase també permet emmagatzemar les dades tot fent servir diferents tipus de compressió. En concret, les compressions possibles són:

- **Cap**
- **Compressió lleugera:** SNAPPY
- **Compressió pesada:** GZIP

La diferència fonamental entre les dues compressions és que les compressions pesades segueixen un algorisme de compressió molt més fort que no pas les lleugeres i per tant comprimeixen molt més. Per contra, seguir algorismes més complexos també implica un major cost de CPU.

També és possible combinar l'HBase amb la compressió LZO. No obstant això, la llicència d'aquesta compressió (GPL[20]) no és compatible amb la llicència de l'HBase (Apache), i per tant la integració de la LZO s'ha de fer després de la instal·lació i no s'ofereix de forma nativa amb l'HBase.

Per altra banda, les altres dues compressions sí són compatibles amb la llicència Apache. Per a poder fer ús de la compressió SNAPPY cal seguir un petits passos d'instal·lació i configuració per tal de fer-lo córrer de forma nativa. Per a la compressió pesada, en

[20] General Public License

canvi, no cal fer cap instal·lació prèvia i fer-la servir és tan fàcil com fer-la explícita en la sintaxis de creació de la taula.

Entrant en més detall en aquesta sintaxis, cal mencionar primerament que la compressió no es fa genèrica a la base de dades. Ni tan sols a la taula, sinó que es fa a nivell de family. Això significa que donat un conjunt de families d'una mateixa taula, els valors de diferents families podries estar comprimits o no, o estar comprimits de formes diferents. Així, la nomenclatura que segueix és del tipus (des de la shell):

```
$ hbase> create 't1', {NAME => 'cf1', COMPRESSION => 'SNAPPY' }
$ hbase> create 't1', {NAME => 'cf1', COMPRESSION => 'GZ' }
```

És fàcil veure com la primera instrucció defineix la taula *t1* amb una sola family *cf1* que es comprimirà amb la compressió lleguera, mentre que la segona instrucció defineix la mateixa taula i family, però amb compressió pesada. En cas que es vulguin combinar diferents compressions en una sola taula, la sintaxis haurà de ser:

```
$ hbase> create 't1', {NAME => 'cf1', COMPRESSION => 'SNAPPY' }, {NAME =>
'cf2', COMPRESSION => 'GZ' }
```

Procés *region lookup* i ZooKeeper

Una part fonamental de l'HBase és el procés que permet a un client esbrinar en quina region, i per tant en quina màquina, pot trobar les dades amb les que treballar.

Aquest procés s'anomena *region lookup* i es fa mitjançant tot un arbre B+. La figura *4.2-11* detalla els tres passos que duen a terme aquest procés:

Concretament, cada un dels tres *lookups* correspon a cada un dels diferents nivells del B+. La part més important en aquest procés és que cada un dels diferents nivells del B+ és una taula HBase. Això implica dos fets principalment:

- Tots els nivells del B+ es troben fragmentats horitzontalment en regions, i aquestes estan distribuïdes al llarg de tot el clúster HBase.
- A diferència de com es pot pensar inicialment, no existeix una jerarquia de màquines corresponent a la jerarquia del B+, sinó que tots els nivells estan dispersos al llarg dels RegionServers.

Aleshores, per tal de poder accedir al primer node del B+ cal mantenir certa informació centralitzada. Aquest és l'objectiu del ZooKeeper: mantenir centralitzada aquella informació que no pot ser distribuïda en sistemes distribuïts.

Per tant, el procés de *region lookup* és defineix de la següent manera:

1. Inicialment el client contacta amb el ZooKeeper per tal d'obtenir el RegionServer on podrà trobar l'arrel del B+.
2. A continuació, el client contacta amb el RegionServer que conté l'arrel del B+ per tal d'esbrinar quin RegionServer li podrà indicar en quin RegionServer es troba el fragment horitzontal de la taula que vol accedir.
3. El client realitza el segon *lookup* i troba en quina màquina trobarà les dades amb les que vol operar.

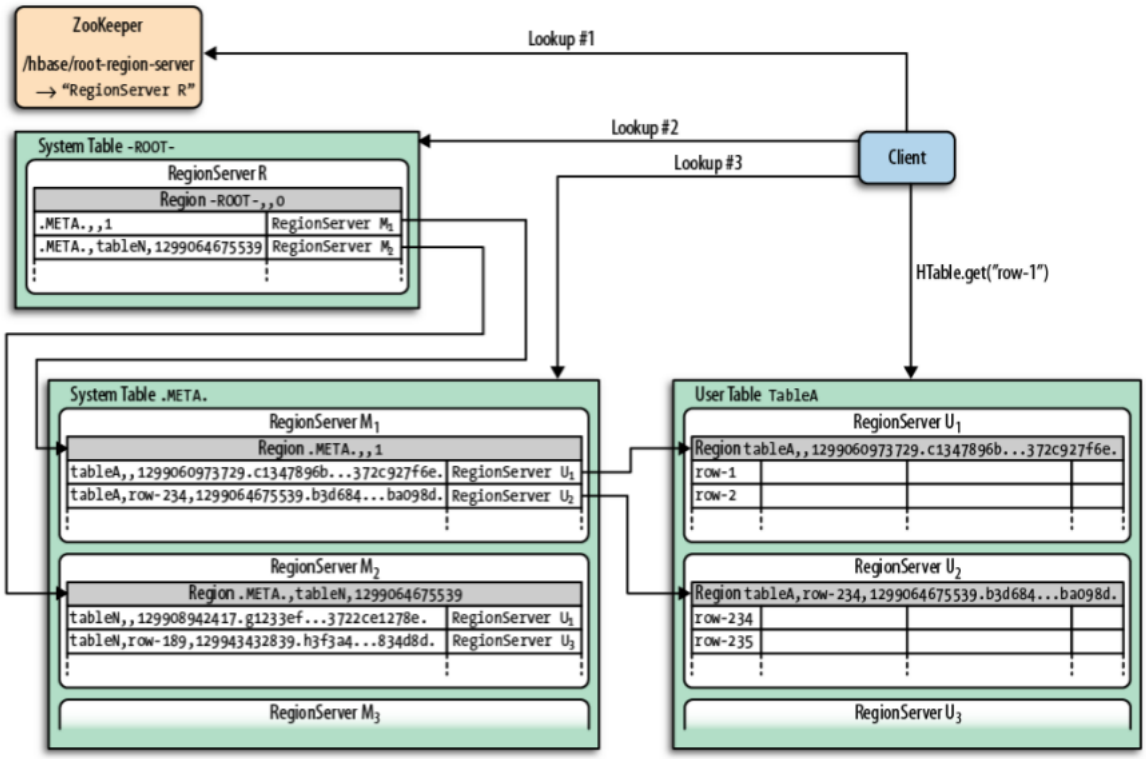


Figura 4.2-11: Procés de *region lookup* de l'HBase

4. Finalment el client es comunica directament amb el RegionServer que serveix la region.

Finalment, la figura *4.2-12* indica més clarament l'estructura d'aquest B+.

4.2.3 MapReduce

Introducció

MapReduce és un model de programació amb l'objectiu de processar grans volums de dades en paral·lel i de forma distribuïda sobre un clúster.

Un programa MapReduce està composat per les següents funcions, que s'executen de forma distribuïda al llarg de màquines diferents:

- Una funció **map()** que filtra, ordena i converteix la entrada del programa en un conjunt de subproblemes en format key-value.
- Una segona funció **reduce()** que du a terme una funció d'agregació per cada un dels conjunts de subproblemes emesos per la funció anterior.

Això implica que la sortida de les funcions map() ha de ser enviada cap als nodes on s'executin les funcions reduce(). Per intentar reduïr aquests costos, aquest model de programació també permet la implementació d'una tercera funció anomenada **combine()**. Aquesta simplement és una funció d'agregació (com el reduce()) intermitja. El seu objectiu és aprofitar les sortides de les funcions de map() executades en una mateixa màquina per fer una preagregació inicial en aquesta mateixa màquina. Així, el que es pretén és que quan

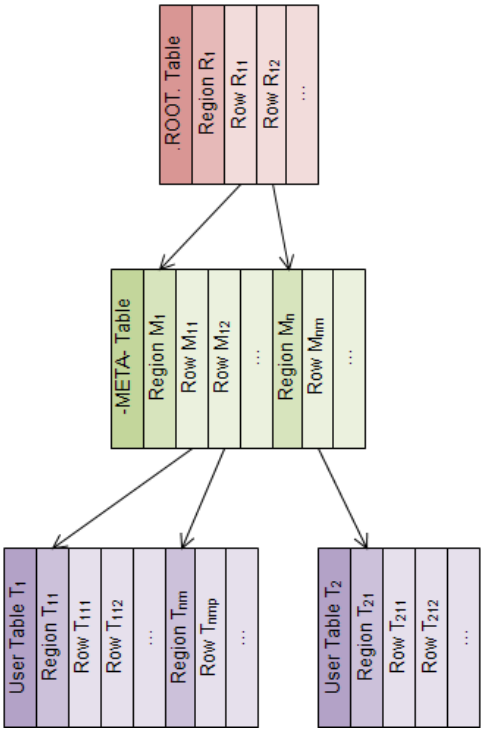


Figura 4.2-12: Arbre B+ del *region lookup*

les funcions de reduce() facin l'agregació final del problema, aquesta ja estigui parcialment feta i per tant s'estalviïn costos hardware.

Després de l'execució de la funció map() i abans de la funció reduce(), MapReduce porta a terme un procés anomenat **merge-sort** en el que s'agrupen i s'ordenen aquells subproblemes on la seva representació key-value té la mateixa key.

Respecte el punt de vist arquitectònic, MapReduce és una altra tecnologia màster-esclau. En aquest cas, la nomenclatura que defineix els dos tipus de nodes és:

- **JobTracker**: És el node màster que gestiona i monitoritza els nodes esclaus.
- **TaskTrackers**: Són els nodes esclaus, i els que realment duen a terme les tasques MapReduce.

A la figura *4.2-13* es pot veure la connexió entres els diferents tipus de nodes MapReduce.

Exemple de funcionament

Un exemple d'execució d'un programa MapReduce mostra millor la idea darrere aquest model de programació. La figura *4.2-14* mostra l'exemple típic d'introducció a aquesta tecnologia: el *wordcount*. Aquest problema consisteix en, donat un conjunt de fitxers, trobar el nombre de vegades que cada paraula apareix.

Tal i com es pot veure en aquest figura, un procés MapReduce consta de les tres etapes definides anteriorment: map(), merge-sort i reduce(). Les funcions combine() s'han obviat per simplificar en aquest cas. També es pot veure el nivell key-value amb el que es treballa, i les conversions que es fan.

Aleshores, el flux d'execució d'aquest exemple segueix de la següent manera:

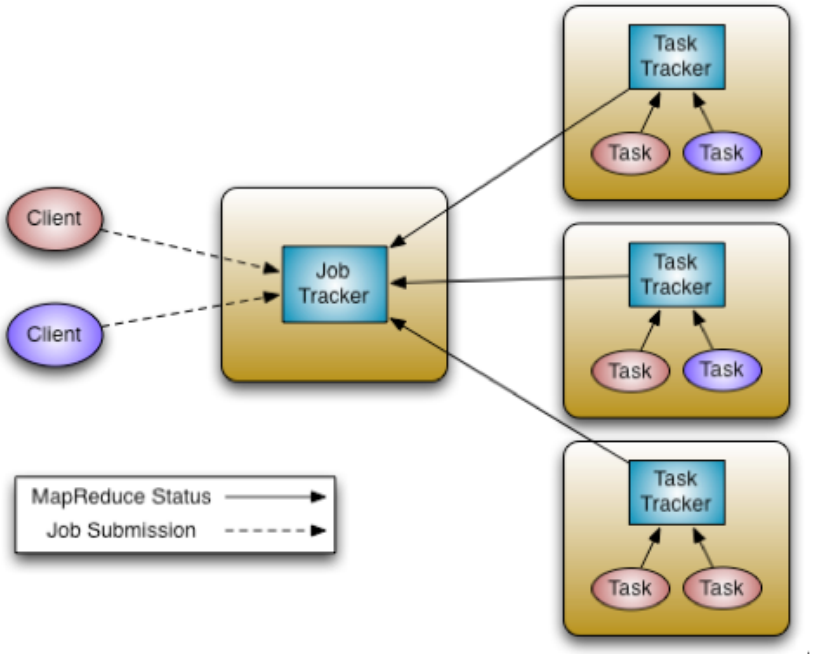


Figura 4.2-13: Arquitectura MapReduce

1. Inicialment, es contenen dos fitxers amb la informació que es vol processar. En aquest cas, el problema consisteix en trobar quantes vegades apareix cada paraula en tots dos fitxers.

2. A continuació s'inicia la tasca MapReduce. Es llegeixen tots dos fitxers, es trenquen per paraules i s'executa una funció map() per cada paraula.

3. Cada funció de map() rep com a entrada una d'aquestes paraules i l'emet cap a la sortida, tot convertint-la en una parella key-value on la key correspon a la paraula i el value és un 1, tot indicant que la solució al subproblema que representa aquest parella és que la paraula representada per la key apareix un sol cop.

4. Quan totes les funcions map() han emès els seus key-values i han finalitzat la seva execució, s'executa el procés de merge-sort que ordena aquestes parelles emeses per les funcions map(). A més de la ordenació, du a terme un procés de unió que consisteix en unir, en un sol value, tots els values d'una mateixa key rebuts des de les diferents funcions de map().

5. Finalment s'executen les funcions reduce(). Aquestes simplement han de fer l'agregació final. Com a entrada, cada reduce() rep una parella key-value resultant del procés anterior. Gràcies al procés d'unió previ, aquesta funció simplement ha de recórrer el value tot aplicant la funció d'agregació corresponent.

Mapper i map, Reducer i reduce

Per tal de comprendre les implementacions MapReduce que s'han fet en aquest projecte, cal diferenciar entre els termes Mapper i map (i el seus homogenis durant la fase de la funció de reduce()). A l'apartat anterior es defineixen les funcions map() i reduce() com si es tractessin de funcions distribuïdes. Però aquesta és la teoria, en el moment d'aplicar

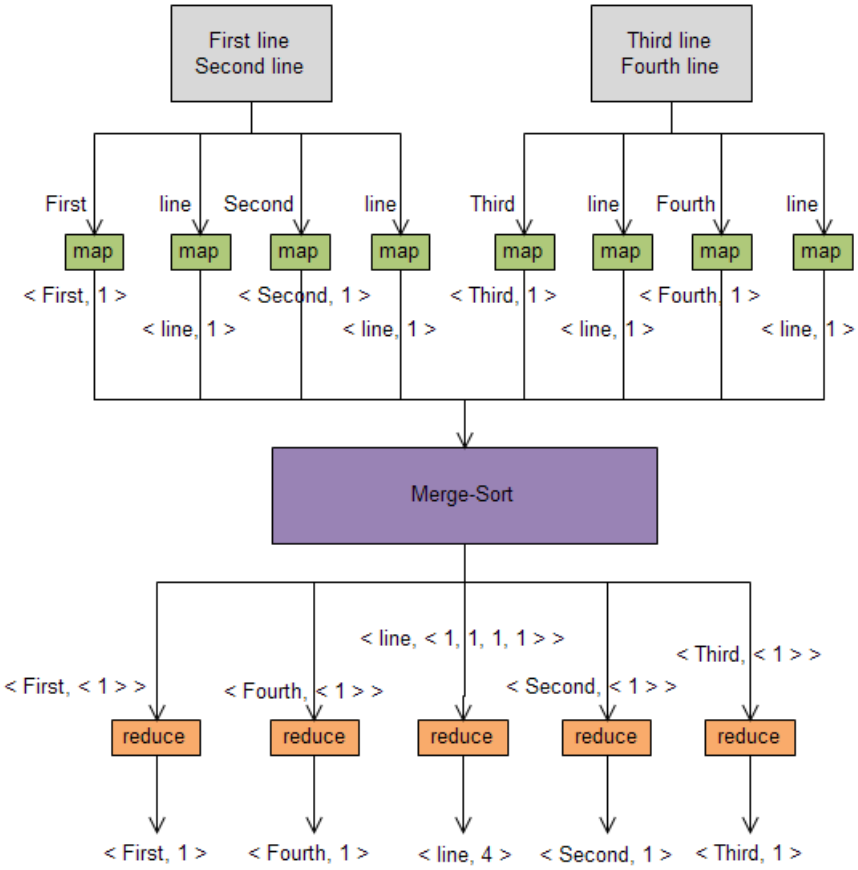


Figura 4.2-14: Solució del *wordcount* amb MapReduce

aquest model de programació sobre els llenguatges convencionals com Java apareixen unes petites variacions.

Concretament, el que succeeix és que aquestes funcions s'encapsulen dins d'un paquet, i aquest paquet és el que es distribueix. Per exemple, en el cas de l'HBase aquests paquets s'assignen a una region diferent. Aleshores, les funcions map() contingudes en aquest procés són les que processen cada una de les files de la region. Així, la terminologia compleix que aquests paquets són els Mappers i les funcions internes són les funcions de map. El mateix ocorre amb les funcions de reduce().

La figura *4.2-15* mostra una execució MapReduce sobre una taula HBase amb la corresponent terminologia *Mapper i map, Reducer i reduce*.

Aquesta terminologia Mapper i map, Reducer i reduce és heretada de les classes Java, però això no eximeix que en altres situacions, o altres projectes la terminologia pugui variar. Per aquest motiu, és tan important definir exactament què és un Mapper i què és un map, ja que són termes que s'utilitzaran molt habitualment en aquest projecte.

Així, en Java, els Mappers són les classes que es distribueixen al llarg de les diferents màquines i que s'executen com si fos un nou fil d'execució, i cada un d'aquests Mappers executa un map per cada un dels elements (línies o tuples) dels paquets de dades (fitxers o regions) que li han estat assignats.

Localitat de dades: HDFS i HBase

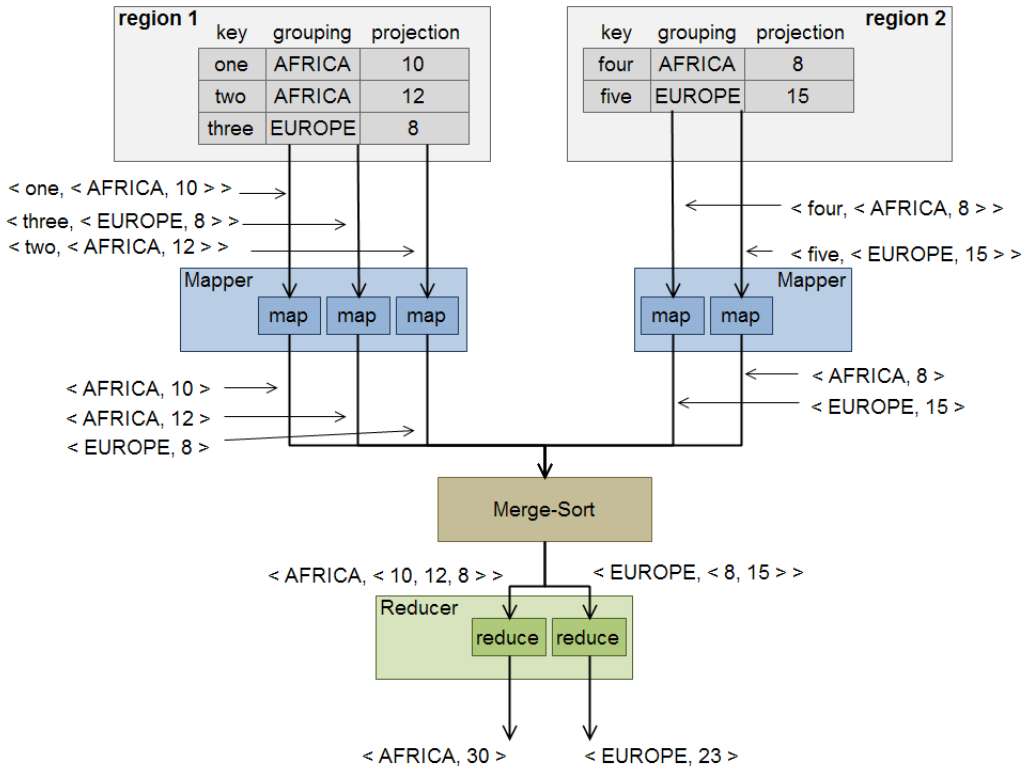


Figura 4.2-15: Aplicació MapReduce sobre l'HBase

Una altra característica remarcable del MapReduce és que permet explotar la localitat de dades tot emplaçant les tasques de Mapper en els mateixos nodes on les dades estan emmagatzemades. Òbviament, això sempre i quan hi hagi una instància TaskTracker corrent sobre aquesta màquina.

D'aquesta manera, la optimització està en evitar costos de latència de moure les dades de les màquines on estan emmagatzemades cap a les màquines on es processaran. No obstant això, aquesta optimització té un cost important de gestió de les màquines i d'assignació de màquines per a la tasca MapReduce. Aquest és un dels principals motius pels quals és ben sabut en el món MapReduce que aquesta tecnologia té un cost d'inicialització massa elevat. En aquest sentit, és encertat afirmar que MapReduce està pensat per treballar amb volums de dades lo suficientment grans com per a que aquest temps d'inicialització sigui menyspreable respecte el temps total de processament de les dades.

4.2.4 Arquitectura global

Finalment, una part important del projecte va ser esbrinar com es du a terme l'acoblament de les tecnologies HDFS i HBase. Principalment, entendre la unió d'aquestes dues tecnologies correctament va permetre tenir un punt de vista a més baix nivell (a nivell de sistema de fitxers) sobre el funcionament intern de la base de dades.

Tal i com es comenta a l'apartat *7.1.7.2 Concentració de dades*, durant les primeres proves d'inserció de les dades es va descobrir que les dades s'emmagatzemen físicament en fitxers al DataNode sobre el que corre el RegionServer que les gestiona. No obstant això, aquesta afirmació és contradictòria amb la definició de l'HDFS, que busca trencar les dades en blocs més petits i distribuir-los al llarg de tot el clúster. A més, des del punt

de vista MapReduce és interessant que les dades d'una única region s'emmagatzemin a la mateixa màquina per tal d'explotar la localitat amb els Mappers.

En aquest sentit, la pregunta que sorgeix a continuació és com garanteix l'HBase que les dades romanen al mateix DataNode que el RegionServer que les gestiona.

La figura *4.2-16* és un diagrama UML que connecta els diferents elements interns de cada tecnologia i que per tant permet donar resposta a la pregunta anterior.

El que sí és segur, és que la unió entre l'HBase i l'HDFS s'ha de fer forçosament al nivell de les màquines esclaves, és a dir, entre els DataNodes i els RegionServers. Així doncs, en aquesta figura es pot comprovar com l'associació entre aquestes dues classes es fa mitjançant dues associacions diferents, on totes dues elles passen per la classe **DFS client**:

1. Quan el RegionServer vol fer una escriptura a disc, aquest ho comunica al seu DFS client.

2. El DFS client obre una canal d'escriptura.

3. El RegionServer comença a escriure les dades en aquest canal a la vegada que el DFS client va agregant i empaquetant aquestes dades.

4. Quan el DFS client detecta que ha agregat un volum de dades suficient (aquest volum és el tamany d'un bloc de l'HDFS), aquest li comunica al NameNode que vol fer l'escriptura d'un bloc.

5. Aleshores, el NameNode busca una màquina apropiada per fer l'escriptura. Aquesta decisió és fa mitjançant la següent política de replicació[21]: *The replica placement strategy is that if the writer is on a datanode, the 1st replica is placed on the local machine, otherwise a random datanode. The 2nd replica is placed on a datanode that is on a different rack. The 3rd replica is placed on a datanode which is on the same rack as the first replica.*

6. Com que l'escriptor d'aquest bloc és el DFS client, el NameNode detecta que aquest corre sobre un DataNode i aleshores fa l'escriptura en aquesta mateixa màquina, que és a la vegada la màquina sobre la que corre el RegionServer.

7. Finalment es produeix l'escriptura i finalitza el procés.

D'aquesta manera, és fàcil observar com les dues associacions que relacionen el DFS client amb el DataNode (juntament amb les restriccions textuals) corresponen a cada una de les possibles rèpliques.

Un altre detall important d'aquest UML és l'extensió de la jerarquia HBase en taules i regions d'usuari i de catàleg. Aquesta petita diferència és per tal de clarificar el procés que es segueix per tal que un client pugui identificar a quina region, i per tant a quin RegionServer, pot trobar les dades que vol accedir. Les explicacions corresponents es troben a l'apartat *4.2.2 Procés region lookup i ZooKeeper*.

[21] http://grepcode.com/file/repository.cloudera.com/content/repositories/releases/com.cloudera.hadoop/hadoop-core/0.20.2-320/org/apache/hadoop/hdfs/server/namenode/ReplicationTargetChooser.java

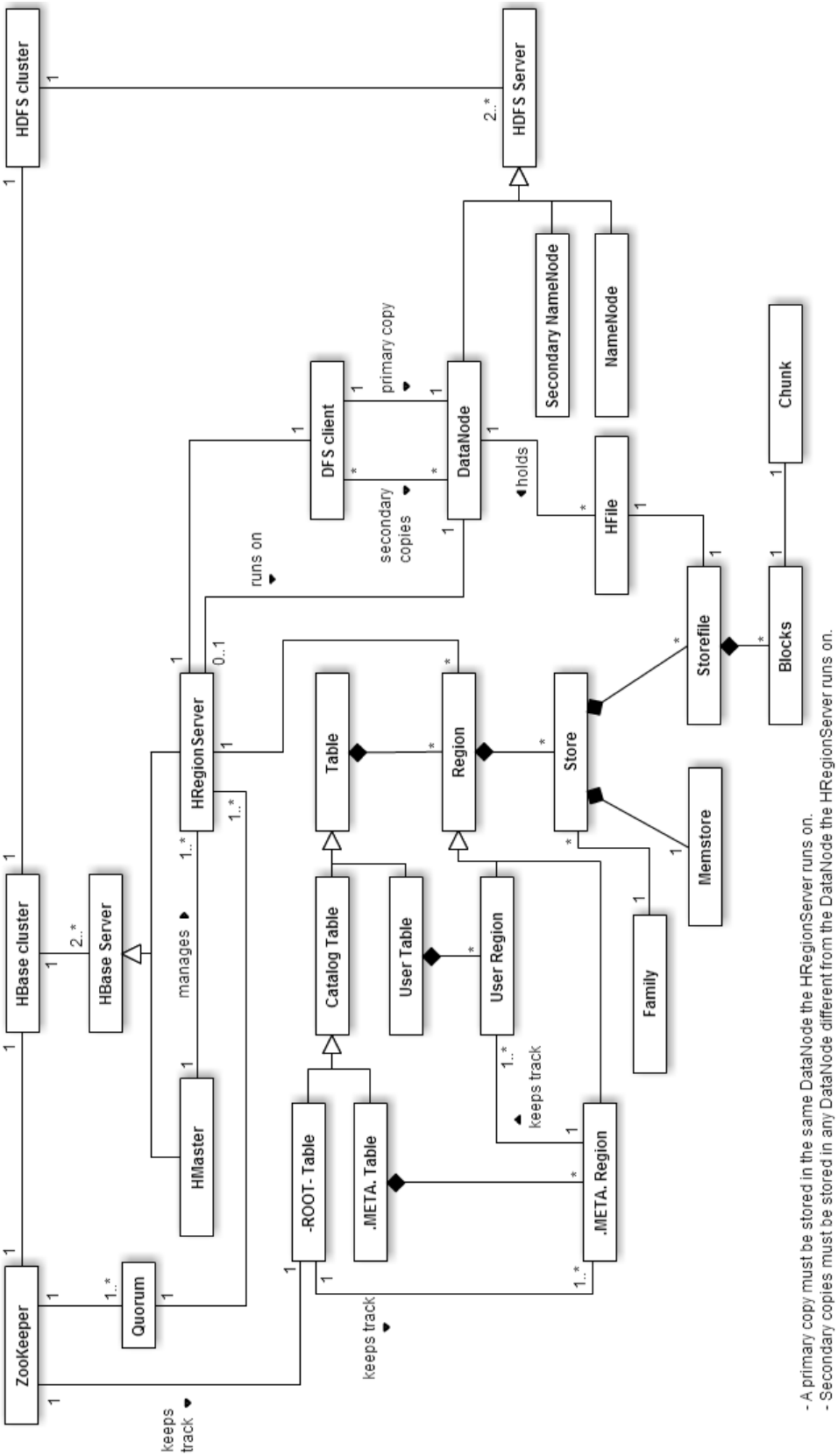


Figura 4.2-16: Arquitectura global. Unió de tots els sistemes

5 Situació de partida

5.1 Descripció inicial

En aquest apartat es fa una descripció de quins algorismes i quines parts del sistema es van heretar del projecte anterior. D'aquesta manera, es busca fer una introducció al punt de partida d'aquest projecte i definir el funcionament intern de cada un dels aspectes que no han calgut implementar perquè ja ho estaven. Cal aclarir que malgrat s'hagin heretat subsistemes del què conforma el sistema general d'aquest projecte, aquests subsistemes no estan exempts de poder ser modificats i per tant, tota aquesta altra part de modificacions i implementacions noves es troba definida a l'apartat *7 Implementació*.

Així doncs, els subsistemes heretats han estat un petit conjunt d'algorismes que compleixen els següents objectius:

- Creació dels índexs.
- Execució de les queries.

Concretament, aquesta secció del projecte ja es situa pròpiament en la construcció dels cubs de dades. La figura *5.1-1* ho mostra amb més detall.

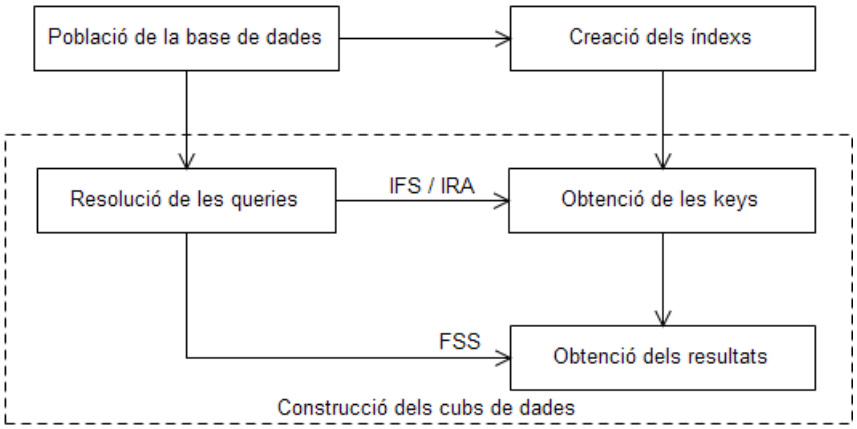


Figura 5.1-1: Flux d'execució del sistema

Es pot veure doncs, com aquesta part heretada és tot el subsistema encarregat de generar els cubs de dades. Això és així, perquè tal i com es defineix a l'apartat *1.1 Treball previ*, aquest projecte neix com a ampliació per definir proves molt més exhaustives del sistema. Això implica que el sistema en si ja es troba desenvolupat.

No obstant això, la exhaustivitat d'aquestes proves serà una de les raons per les quals caldrà solventar els problemes apareguts en el projecte anterior i així poder dur a terme aquestes proves correctament.

Finalment, per tal de continuar amb els apartats següents cal definir un petit vocabulari:

- **Atributs de projecció:** Són els atributs que una query ha de projectar. En altres paraules, són els atributs de sortida.
- **Atributs d'agregació:** Són un subconjunt dels atributs de projecció. Concretament, són atributs amb l'objectiu de ser agregats per alguna funció aritmètica. Per exemple el número de vendes, el número d'accessos, etc.

- **Atributs de selecció:** Els atributs que conformen les diferents parts de la selecció, que a l'hora defineix quin és el subconjunt de dades al que cal accedir.
- **Atributs de condició:** El mateix que els atributs de selecció.
- **Atributs d'agrupació:** Són els atributs que defineixen l'espai multidimensional en el que es situarà el cub de dades.

5.2 Creació dels índexs

5.2.1 Per què índexs?

Un aspecte important del projecte és la creació d'uns índexs que permetin seleccionar aquelles keys a les que realment cal accedir durant el procés de resolució de queries.

En aquest projecte, aquests índexs s'allunyen dels pensats pel món relacional i seguint el model MapReduce, constitueixen parelles key-value on la key és cada valor possible de l'atribut que indexa i el value és tot el conjunt de keys amb aquell valor. La següent taula mostra un exemple de com s'espera que les dades s'organitzin en aquests índexs:

Nivells d'agregació	**Llistat de keys**
Any=2013, Mes=Octubre, Dia=15	1, 4, 6, 7, 8
Any=2013, Mes=Octubre, Dia=31	2, 3, 9, 12
Any=2013, Mes=Novembre, Dia=1	5, 10, 11
Any=2012, Mes=Gener, Dia=4	13, 14

La construcció d'aquests índexs ha de ser prèvia a la resolució de les queries i no es tindrà en compte en el temps total d'execució de les mateixes. Això és degut a que un índex és una estructura auxiliar que es construeix tan sols un cop i que pot ser llegida tants cops com es vulgui. No es tracta de que cada query nova hagi de passar per aquest procés.

Per a la construcció d'aquests índexs es fa servir una tasca MapReduce, ja que un cop més es tracta de llegir parelles key-value per tal de generar unes de noves.

Als subapartats següents s'entra en més detall en cada un dels tipus d'índexs definits i en els algorismes que els implementen, però per tal de posar al lector al corrent, aquests índexs són:

- **Single Level Index.** Parelles key-value a un únic nivell d'agregació.
- **Multiple Level Index.** Parelles key-value a varis nivells d'agregació.

5.2.2 Single Level Index

Aquest tipus d'índex consisteix en indexar les keys a l'últim nivell d'agregació. Així l'exemple vist a l'apartat *5.2.1 Per què índexs?* correspon a un índex d'aquest tipus, ja que la dimensió *data* conté tres nivells d'agregació: *any, mes* i *dia* i una key es diferencia d'una altra pel conjunt sencer de valors al que pertany.

En aquest tipus d'índex, la part positiva és el volum espacial que ocupa un índex. Es tracta d'un volum relativament reduït ja que una key s'escriu només un cop. La contrapartida d'aquesta estructura és quan els accessos a l'índex són per recuperar nivells d'agregació molt alts, ja que això implica masses lectures l'índex. Per exemple, en el mateix model d'índex que el vist anteriorment, per recuperar totes les keys amb *any=2013* caldria realitzar un total de tres lectures.

El pseudocodi que segueix aquest tipus d'índex és el mostrat a l'algorisme *5.2-1 Construcció del Single Level Index*.

Algorisme 5.2-1: Construcció del Single Level Index

```
action launch(var table, array attributes[0 .. A], var name)
begin
  var scan := createScan(table, attributes)
  setMapReduceConfiguration(''attributes'', attributes)
  setMapReduceConfiguration(''dimension_name'', name)
  launchMapReduce(scan)
end

action map(var<key, value> row)
begin
  array attributes[0 .. A] := getMapReduceConfiguration(''attributes'')
  var dimension := getMapReduceConfiguration(''name'')
  for a in attributes
  do
    var v := getValue(value, a)
    concat(dimension, ''%'', v)
  done
  emit(dimension, key)
end

action reduce(var<key, value[0 .. V]> dimension)
begin
  var first := true
  var result := EMPTY_STRING
  for v in value
  do
    if first = true
    then
      result := v
    else
      concat(result, '','', v)
    fi
  done
  emit(key, result)
end
```

El procés de creació d'aquests índexs segueix un procés MapReduce. Inicialment, aquesta tasca MapReduce rep com a entrada la taula de fets per a la qual cal construir l'índex i el conjunt d'atributs que formaran els diferents nivells d'agregació de la dimensió. Aleshores, es configura la tasca convenientment i es llença l'execució.

Les funcions de map han de realitzar una feina molt simple. Bàsicament el seu objectiu consisteix en emetre el mateix key-value de la entrada però de forma inversa, tal que la key d'entrada sigui el value de sortida i el value d'entrada sigui la key de sortida. D'aquesta manera, el procés de combinació del MapReduce ajuntarà totes aquelles tuples amb els mateixos valors de dimensió. Cal tenir en compte, però, que en aquest índex les keys s'han d'indexar a l'últim nivell d'agregació i per tant abans d'emetre el nou key-value s'han d'ajuntar cada un dels valors dels nivells d'agregació en una sola cadena. Aquest és l'objectiu del bucle que realitza la funció map. A més, el caràcter "%" permet diferenciar cada un d'aquests nivells.

A continuació, la funció reduce tan sols ha de concatenar totes les keys de la taula de fets per cada valor dels nivells d'agregació i emetre aquest resultat per a què sigui inserit a la taula de dimensions. Així, la taula de dimensions tindria un aspecte com el següent:

Rows de la taula de dimensions	**Llistat de keys**
Date%2013%Octubre%15	1, 4, 6, 7, 8
Date%2013%Octubre%31	2, 3, 9, 12
Date%2013%Novembre%1	5, 10, 11
Date%2012%Gener%4	13, 14

5.2.3 Multiple Level Index

El Multiple Level Index es basa en indexar les keys en tots els nivells d'agregació possibles. Així l'exemple vist a l'apartat *5.2.1 Per què índexs?* es desglossa de la següent manera.

Nivells d'agregació	**Llistat de keys**
Any=2013	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
Any=2012	13, 14
Any=2013, Mes=Octubre	1, 2, 3, 4, 6, 7, 8, 9, 12
Any=2013, Mes=Novembre	5, 10, 11
Any=2012, Mes=Gener	13, 14
Any=2013, Mes=Octubre, Dia=15	1, 4, 6, 7, 8
Any=2013, Mes=Octubre, Dia=31	2, 3, 9, 12
Any=2013, Mes=Novembre, Dia=1	5, 10, 11
Any=2012, Mes=Gener, Dia=4	13, 14

L'avantatge d'aquest tipus d'índex és que, sigui quin sigui el nivell d'agregació que cal accedir, el nombre de lectures sempre és una. Per contrapartida, l'espai ocupat creix significativament respecte el Single Level Index.

El pseudocodi que segueix aquest tipus d'índex és el mostrat a l'algorisme *5.2-2 Construcció del Multiple Level Index*. Les modificacions respecte al Single Level Index només calen a la funció de map; la resta de funcions es mantenen intactes. Per aquest motiu, en aquest pseudocodi es mostra només aquesta funció. La resta és el mateix que mostra l'algorisme *5.2-1 Construcció del Single Level Index*.

Algorisme 5.2-2: Construcció del Multiple Level Index

```
action map(var<key, value> row)
begin
  array attributes[0 .. A] := getMapReduceConfiguration(''attributes'')
  var dimension := getMapReduceConfiguration(''name'')
  for a in attributes
  do
    var v := getValue(value, a)
    concat(dimension, ''%'', v)
    emit(dimension, key)
  done
end
```

La diferència principal entre la funció map d'aquest algorisme i la del Single Level Index és en quin moment i quants cops cal emetre cada una de les keys de la taula de fets. Si recordem, l'algorisme anterior es basa en indexar aquestes keys a l'últim nivell

d'agregació, mentre que aquest segon algorisme permet indexar cada key a cada un dels diferents nivells. Per tant, les modificacions que cal fer són molt simples, ja que es tracta d'emetre la key de la taula de fets en cada una de les iteracions del bucle que construeix la cadena que representa els valors de la dimensió. D'aquesta manera, en comptes de produir una sola emissió de la key de la taula de fets, aquest algorisme fa una emissió per cada un dels possibles nivells d'agregació de la dimensió, amb els seus corresponents valors.

Finalment, la funció de reduce només ha d'agrupar i fer la concatenació de les keys de la taula de fets que rep com a entrada. Per tant, es pot fer servir el mateix codi que l'utilitzat a l'algorisme *5.2-1 Construcció del Single Level Index*.

5.3 Accés als índexs. Selecció

El procés de selecció de les queries es produeix mitjançant la execució d'un MapReduce. Aquest MapReduce fa la lectura de la taula de dimensions, i que per tant ha d'haver estat prèviament creada i poblada. La forma de dur a terme la selecció consisteix en trencar el conjunt de la selecció en seleccions atòmiques, i aleshores emetre per cada selecció atòmica el conjunt de keys que la satisfan. Finalment, es descarten aquelles keys que no han estat emeses tantes vegades com seleccions atòmiques conté la selecció global.

Tal i com s'ha vist a l'apartat *5.2 Creació dels índexs*, el sistema permet la utilització de dos tipus d'índexs diferents: el Single Level Index i el Multiple Level Index. En el cas de l'accés als índexs, però, segueix el mateix algorisme per a tots dos casos. Això és així perquè l'estructura general de tots dos índexs és la mateixa. La diferència principal està en les diferents entrades que té cada índex. No obstant això, l'API de l'HBase permet definir un accés a la base de dades a partir d'un prefix i aleshores retornar tot el conjunt de tuples on la key comença per aquell prefix. Aquí és on rau el mecanisme que permet fer servir el mateix algorisme per a tots dos casos.

Concretament, aquest mecanisme de prefix funciona correctament amb ambdós algorismes per les següents raons:

- En el cas del Single Level Index, cal recordar que les keys s'indexen a l'últim nivell d'agregació. Per tant, la tècnica de prefix funciona correctament ja que el prefix d'accedir a l'últim nivell d'agregació és aquest mateix nivell, i tots els nivells d'agregació intermitjos comparteixen el mateix prefix. Per exemple, en el cas d'una dimensió com la següent:

Nivells d'agregació	**Keys**
Date%2013%Octubre%15	1, 4, 6, 7, 8
Date%2013%Octubre%31	2, 3, 9, 12

  a) **Prefix per a obtenir al conjunt de keys de l'any 2013:** *Date%2013*

  b) **Prefix per a obtenir al conjunt de keys de l'Octubre del 2013:** *Date%2013%Octubre*

  c) **Prefix per a obtenir al conjunt de keys del 31 d'Octubre del 2013:** *Date%2013%Octubre%31*

  I per tant queda clar que accedir a qualsevol nivell de l'agregació té un prefix concret.

- Aquesta explicació és també vàlida pel cas del Multiple Level Index, però amb una petita excepció. En aquest segon cas, continua sent cert que cada nivell d'agregació

té un prefix determinat, però no és cert que aquest prefix sigui únic per cada nivell d'agregació i per tant és possible que un accés com el del Single Level Index retorni el mateix conjunt de keys, però duplicat. L'exemple anterior desglossat per al Multiple Level Index es veuria de la següent manera:

Nivells d'agregació	**Keys**
Date%2013	1, 2, 3, 4, 6, 7, 8, 9, 12
Date%2013%Octubre	1, 2, 3, 4, 6, 7, 8, 9, 12
Date%2013%Octubre%15	1, 4, 6, 7, 8
Date%2013%Octubre%31	2, 3, 9, 12

És fàcil veure com per exemple un accés amb el prefix *Date%2013%Octubre* seria per accedir a l'entrada *Date%2013%Octubre*, però també s'accediria a: *Date%2013%Octubre%15* i *Date%2013%Octubre%31*, perquè totes tres entrades comparteixen el mateix prefix. Aleshores, la forma més fàcil de garantir que cada entrada té assignat un prefix únic és la d'afegir una paraula que acrediti el final de cada entrada. Aquesta paraula és "All". D'aquesta manera, l'exemple anterior es converteix en el següent:

Nivells d'agregació	**Keys**
Date%2013%All	1, 2, 3, 4, 6, 7, 8, 9, 12
Date%2013%Octubre%All	1, 2, 3, 4, 6, 7, 8, 9, 12
Date%2013%Octubre%15%All	1, 4, 6, 7, 8
Date%2013%Octubre%31%All	2, 3, 9, 12

I finalment ara sí que s'aconsegueix que cada entrada li correspongui un prefix únic. Aquesta és doncs, la única diferència entre accedir a un tipus d'índex o a un altre. No obstant això, cal aclarir que la paraula "All" pot servir en aquest cas perquè cap atribut del TPC-H pot prendre aquest valor, però en qualsevol altre cas caldria validar que la paraula triada no pogués coincidir amb algun valor d'atribut.

Així doncs, a partir d'aquí l'algorisme d'accés és idèntic per tots dos casos. La figura *5.3-1* representa aquest procés de selecció.

Una petita optimització que es va fer en aquesta part va ser la de escriure cada una de les seleccions atòmiques en fitxers diferents per tal de garantir el paral·lelisme entre cada una d'elles. Inicialment, s'escrivien totes sobre el mateix nivell i per tant es processaven una a una dins del mateix Mapper. Degut a que aquesta modificació és molt simple, es menciona en aquest apartat i no en el corresponent apartat d'implementacions.

Un petit pseudocodi que il·lustra tot aquest procés és el mostrat a l'algorisme *5.3-1 Procés de selecció.*

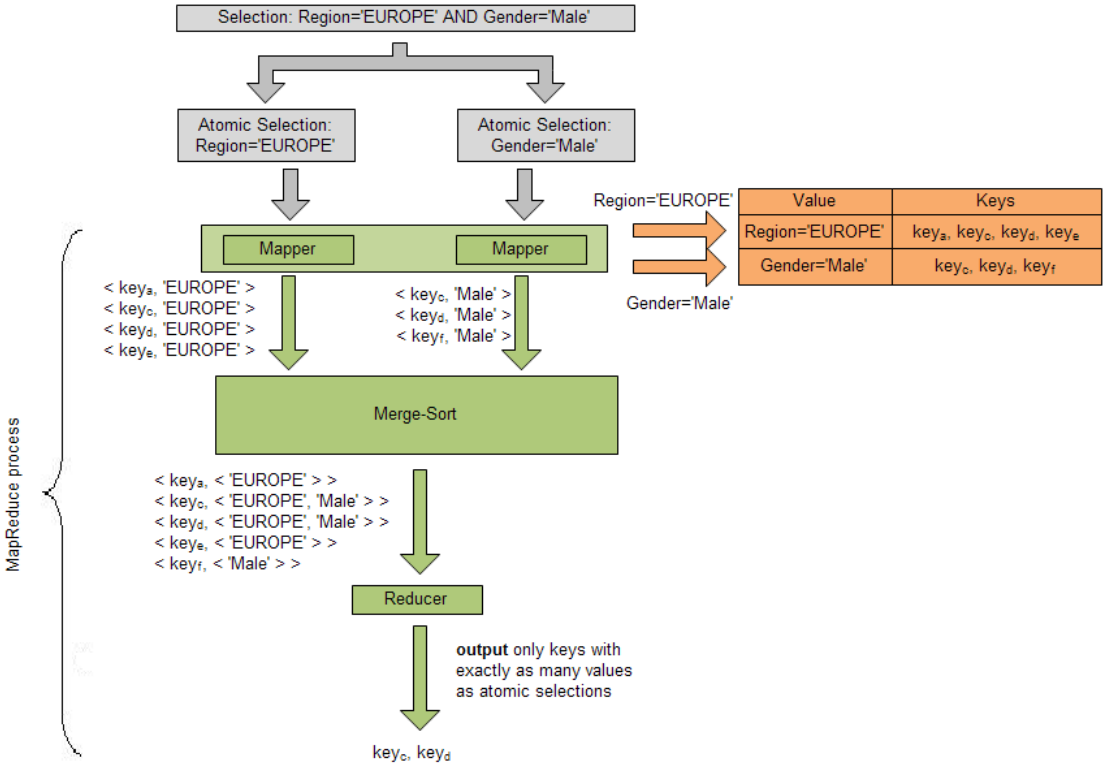


Figura 5.3-1: Execució de la selecció mitjançant l'índex

Algorisme 5.3-1: Procés de selecció

```
action launch(array files[0 .. F])
begin
  setMapReduceConfiguration(''number_of_conditions'', F + 1)
  launchMapReduce(files)
end

action map(var<key, value> row)
begin
  var scan := createScan(DIMENSION_TABLE, ''keys'')
  setPrefix(scan, value)
  array keys[0 .. R] := launchScan(scan)
  for k in keys
  do
      emit(k, value)
  done
end

action reduce(var< key, value[0 .. V] > row)
begin
  var number_of_conditions := getMapReduceConfiguration(''number_of_conditions'')
  if V + 1 = number_of_conditions
  then
    emit(null, k)
  fi
end
```

Tal i com es veu en el pseudocodi, es llença la tasca MapReduce sobre el conjunt de fitxers $files$ que cal accedir. Internament, MapReduce fa la lectura de cada un d'aquest fitxers i es llença una funció map per cada una de les línies dels fitxers. Aquesta funció aleshors fa una lectura de la taula de dimensions per tal de trobar les entrades què tenen com a prefix els valors de la condició (veure apartat *5.2 Creació dels índexs*). Seguidament, aquest accés a la taula de dimensions retorna el conjunt de keys de la taula de fets que compleixen la selecció atòmica donada. Aleshores, la funció map emet una parella key-value on la key correspon a una key de la taula de fets, i el value correspon al value de l'entrada del map, és a dir, al contingut de la selecció atòmica.

A continuació s'executa la funció reduce. Aquesta simplement s'encarrega de validar quines keys de la taula de fets satisfan totes les seleccions atòmiques de la selecció. Per tal de dur aquesta tasca a terme, el que rep com a entrada és un altre conjunt key-value on la key és una key de la taula de fets (sortida del map) i el value és tot el conjunt de seleccions atòmiques que aquella key satisfà (sortida del map i agrupació del MapReduce). Donat aquesta situació, el codi de la funció de reduce és tan fàcil com comprovar que el nombre de seleccions atòmiques que satisfà cada key equivalgui al nombre total de seleccions atòmiques que formen la selecció global.

No obstant això, a les sortides de totes dues funcions map i reduce es troben dos detalls que poden ser d'interès:

1. El primer és que la sortida de la funció map és una parella key-value on el value és un paràmetre molt definit, però la veritat és que aquest paràmetre es podria definir de qualsevol manera ja que lo important aquí és la cardinalitat final de l'agrupació del MapReduce, independentment del seu contingut.

2. El segon és la sortida del reduce. En aquest cas lo important és que la keys que compleixen tota la selecció s'escriguin en la sortida. Per aquest motiu, a la sortida de la funció del reduce aquesta key ha d'aparèixer com a value i és independent de la key de sortida.

5.4 Algorismes d'obtenció dels valors

5.4.1 Full Source Scan. FSS

Aquest algorisme ja es ben conegut. Es tracta d'un **full scan**[22] tal i com es coneix en el món relacional.

L'algorisme *5.4-1 Full Source Scan* mostra amb més detall el funcionament intern d'aquest algorisme.

[22]Lectura de totes les tuples d'una taula.

Algorisme 5.4-1: Full Source Scan

```
action launch(var table, array projection[0 .. P], array groupby[0 .. G], array filters[0 .. S])
begin
  var scan := createScan(table, projection, groupby)
  addFilter(scan, filters)
  setMapReduceConfiguration(''projection'', projection)
  setMapReduceConfiguration(''groupby'', groupby)
  executeMapReduce(scan)
end

action map(var<key, value> row)
begin
  array projection[0 .. P] := getMapReduceConfiguration(''projection'')
  array groupby[0 .. G] := getMapReduceConfiguration(''groupby'')
  var groupby_values := EMPTY_STRING
  for g in groupby
  do
    var v := getValue(value, g)
    groupby_values := concat(groupby_values, ''_'', v)
  done
  for p in projection
  do
    var k := concat(groupby_values, ''_'', p)
    var v := getValue(value, p)
    emit(k, v)
  done
end

action reduce(var<key, value[0 .. V]> result)
begin
  var aggregate := 0
  for v in value
  do
    aggregate := aggregate + v
  done
  emit(key, aggregate)
end
```

Inicialment, la funció de launch és la que s'encarrega de preparar els paràmetres adequadament per tal de llençar el MapReduce correctament. Aquesta funció es tracta principalment de l'assignació de la taula i dels atributs que es llegiran. En el cas d'aquest algorisme, aquesta funció també s'encarrega de construir l'objecte scan que llegirà la taula de fets de manera que filtri aquelles tuples que no satisfan les condicions de la selecció.

Un cop el MapReduce està en marxa, es s'executen les funcions de map. Una per cada una de les tuples de la taula. Tal i com es mostra a l'algorisme, aquesta primera funció de map es basa en emetre parelles key-value amb cada un dels atributs de projecció (agregació) i agrupació correctament definits:

- Els atributs d'agrupació es concatenen tot formant una cadena de valors que representen l'emplaçament de la tupla que actualment s'està processant a l'espai multidimensional del cub.
- Els atributs d'agregació s'emeten un cop cada un, construint parelles key-value jun-

tament amb les cadenes de valors formades a partir dels atributs d'agrupació.

Finalment, aquestes són les parelles key-value que s'emeten des del map. Concretament, en aquests key-values les keys corresponen als atributs d'agrupació i el value als atributs d'agregació.

Durant el procés intermig entre els maps i els reduces es produeix la unió de totes les parelles key-value que tenen la mateixa key. En aquest cas, com que els key-value emesos tenen per key el conjunt de valors marcats pels atributs d'agrupació, es pot assegurar que aquest procés d'unió ajuntarà aquells valors d'agregació que facin referència a objectes emplaçats en el mateix punt multidimensional.

En última instància, la funció de reduce només ha de fer l'agregació final dels valors. En el cas d'aquest algorisme, s'ha simplificat la funció d'agregació a una suma, però podria ser qualsevol altra.

Figura 5.4-1: Diagrama d'execució de l'algorisme FSS

5.4.2 Index Filtered Scan. IFS

L'algorisme IFS és una petita modificació respecte el FSS. De la mateixa manera que l'anterior, realitza un full scan de la taula amb les excepcions:

- En comptes de produir la selecció mitjançant la lectura dels respectius atributs, fa un accés a l'índex tal i com es detalla a l'apartat *5.3 Accés als índexs. Selecció.* Això li permet muntar una estructura de filtre que indica quins maps s'han de llençar. Concretament es tracta d'una estructura bitstring. Aquesta estructura conté tants bits com files hi ha a la taula. Aleshores, si a l'hora de llençar un map per a la fila n el bit n d'aquest bitstring equival a 1, es llença el map en qüestió i si no, s'escapa. D'aquesta manera es permet filtrar aquells maps que no produiran cap resultat perquè no compleixen la selecció. L'algorisme *5.4-2 Construcció del bitstring* mostra aquest procés.

Algorisme 5.4-2: Construcció del bitstring

```
var bitstring := createBitstring(keys, bitmap_entries)

action createBitstring(array keys[0 .. K], var bitmap_entries)
begin
  array bits[0 .. ((bitmap_entries >> 3) + 1)] := INIT_TO_ZERO
  if length(keys) > 1 || (length(keys) = 1 && keys[0] != ''0'')
  then
    for key in keys
    do
      var position := key >> 3
      var bit := 0x01 << (key & 0x07)
      bits[pos] = bits[pos] | bit
    done
  fi
  return bits
end

function getBit(var id)
begin
  var pos = id >> 3
  var b = 0x01 << (id & 0x07)
  return (bitmap[pos] & b) != 0
end
```

És important mantenir al cap que el disseny de les keys es fa en ordre consecutiu. La funció encarregada de construir aquest bitstring és la funció *createBitstring()*. Aquesta funció rep com a entrada dos paràmetres que són, respectivament: el llistat de keys que satisfan la selecció i el nombre total de tuples. Tenint en compte que el bitstring final conté tants bits com tuples hi ha a la taula de fets i que cada bit correspon a una tupla de forma ordenada, aleshores podem redefinir aquests paràmetres com: el llistat de bits que cal activar i el nombre total de bits.

Aleshores, el centre d'aquestes funcions està en treballar pròpiament a nivell de bit a partir de bytes (que és el nivell més baix al qual els llenguatges de programació permeten treballar) i per tant es fa imprescindible que operador lògics com la OR, AND o SHIFT no entrin en joc. Concretament, la funció *createBitstring()* comença assignant espai al bitstring mitjançant un operador SHIFT. Això és així per a fer la conversió entre el nombre de bits i el nombre de bytes. Com que un byte correspon a vuit bits, aleshores és fàcil veure com N bits = $N/8$ bytes.

A continuació, s'activen els bits corresponentment. Un altre cop, la operació aquí ha de tenir en compte que només es pot treballar a nivell de bytes. Això significa que activar un cert bit implica llegir el bytes que conté aquell bit i activar-lo sense modificar la resta del byte. Per aquest motiu cal tornar a fer ús dels operadors lògics.

Inicialment cal trobar en quin byte es troba el bit que es vol activar. Per dur a terme això, només cal aplicar la mateixa justificació que en el moment de calcular el nombre de bytes del bitstring i dividir la key que cal activar entre vuit. El següent pas és esbrinar quin d'aquests vuit bits correspon al bit en qüestió. Això s'obté mitjançant la operació AND amb una màscara $0x07_h = 00000111_b$. D'aquesta manera, el que realment s'està calculant és el residu de la divisió anterior i per tant,

com que a cada byte li corresponen vuit keys diferents, la posició de la key dins del byte correspon al residu (ja que si a dues keys d'un mateix byte li corresponen el mateix residu, aleshores es tracta de la mateixa key). Finalment, només cal activar el bit convenientment. La següent taula mostra aquest procés més clarament:

Key a activar	**Correspon al byte**	**Correspon al bit**
0	0 (0 $\gg$ 3)	0 (0 & 0x07)
1	0 (1 $\gg$ 3)	1 (1 & 0x07)
2	0 (2 $\gg$ 3)	2 (2 & 0x07)
3	0 (3 $\gg$ 3)	3 (3 & 0x07)
4	0 (4 $\gg$ 3)	4 (4 & 0x07)
5	0 (5 $\gg$ 3)	5 (5 & 0x07)
6	0 (6 $\gg$ 3)	6 (6 & 0x07)
7	0 (7 $\gg$ 3)	7 (7 & 0x07)
8	1 (8 $\gg$ 3)	0 (8 & 0x07)
9	1 (9 $\gg$ 3)	1 (9 & 0x07)
10	1 (10 $\gg$ 3)	2 (10 & 0x07)
11	1 (11 $\gg$ 3)	3 (11 & 0x07)
12	1 (12 $\gg$ 3)	4 (12 & 0x07)
13	1 (13 $\gg$ 3)	5 (13 & 0x07)
14	1 (14 $\gg$ 3)	6 (14 & 0x07)
15	1 (15 $\gg$ 3)	7 (15 & 0x07)

La funció de lectura *getBit()* consisteix en simplement obtenir el bit per a una determinada key tot seguint les justificacions anteriors i retornar cert o fals depenent del bit si està activat o no.

En termes més tècnics aquesta implementació es fa durant el procés de *start up* de cada un dels maps de cada Mapper. Si recordem a l'apartat *4.2.3 MapReduce* es defineix un Mapper com cada una de les tasques que es distribueixen al llarg dels nodes i que reben com a entrada una region. Aleshores, els maps són cada una de les funcions que processen cada una de les tuples de la region assignada al Mapper al que pertanyen.

Concretament, el que succeeix amb aquest algorisme és que els Mappers s'executen normalment, un per cada region, però és reimplementa la funció de *start up* per tal de validar amb el bitstring quins maps calen realment executar. D'aquesta manera, s'estalvien principalment costos de CPU i de comunicació.

- Donat que en el moment de fer el full scan ja se sap a quines keys s'ha d'accedir (per l'accés a l'índex), aquest algorisme acota aquest full scan per a que es realitzi només des de la key més petita fins a la key més gran. Això també defineix un altre tipus de filtre, ja que permet estalviar els costos de CPU i de comunicació de processar les dades que no formen part d'aquest rang de keys. No obstant això, l'eficàcia d'aquest filtre és molt menor que la del filtre anterior, ja que permet ajustar el full scan a un rang molt determinat de keys, però en cap moment s'avaluen les dades contingudes en aquest rang. Per exemple, no tindrà el mateix efecte si calen recuperar dues tuples que estan molt concentrades, o bé aquestes dues tuples corresponen a la primera i a la última tupla de la taula. En cas que es correspongui a la primera i última tupla, el rang definit serà tota la taula en si i per tant el guany aportat per aquest filtre serà nul.

La figura *5.4-2* mostra el diagrama d'execució d'aquest algorisme. Tal i com es pot apreciar en aquesta figura, l'execució d'aquest algorisme consisteix en l'execució de dos MapReduces: el primer obté el conjunt de keys que satisfan la selecció, i el segon obté els valors de la projecció i de l'agrupació d'aquestes keys.

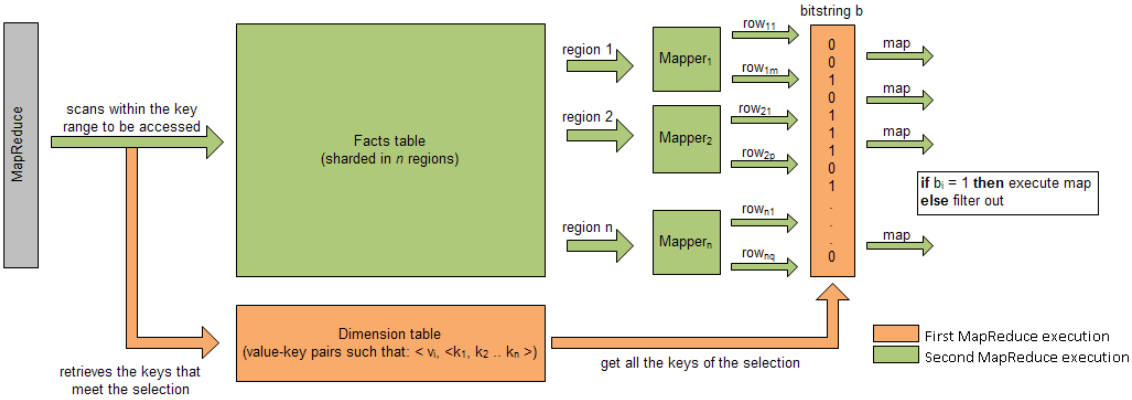


Figura 5.4-2: Execució de l'IFS

Tal i com es menciona anteriorment, en primer lloc es llença el MapReduce que ha de retornar els conjunt de keys a accedir. En segona instància, es llança una nova tasca MapReduce encarregada de trobar els respectius atributs (agregació i agrupació) de cada una de les keys resultants del MapReduce anterior. Aleshores, primerament es construeix el bitstring que ha de servir com a filtre i s'estableix el rang de keys a les que l'scan del MapReduce s'accedirà. En aquest moment, s'executa aquest segon MapReduce on finalment la seva sortida ha de ser el resultat final de la query.

L'algorisme *5.4-3 Index Filtered Scan* representa aquesta segona execució.

Algorisme 5.4-3: Index Filtered Scan

```
var bitstring

action launch(var table, array projection[0 .. P], array groupby[0 .. G])
begin
  var scan := createScan(table, projection, groupby)
  setMapReduceConfiguration(''projection'', projection)
  setMapReduceConfiguration(''groupby'', groupby)
  executeMapReduce(scan)
end

action startup(array<key, value>[0 .. R] rows)
begin
  for row<k, v> in rows
  do
    var b := getBit(bitstring, k)
    if b = 1
    then
      map(row)
    fi
  done
end

action map(var<key, value> row)
begin
  array projection[0 .. P] := getMapReduceConfiguration(''projection'')
  array groupby[0 .. G] := getMapReduceConfiguration(''groupby'')
  var groupby_values := EMPTY_STRING
  for g in groupby
  do
     var v := getValue(value, g)
    groupby_values := concat(groupby_values, ''_'', v)
  done
  for p in projection
  do
    var k := concat(groupby_values, ''_'', p)
    var v := getValue(value, p)
    emit(k, v)
  done
end

action reduce(var<key, value[0 .. V]> result)
begin
  var aggregate := 0
  for v in value
  do
    aggregate := aggregate + v
  done
  emit(key, aggregate)
end
```

5.4.3 Index Random Access. IRA

L'algorisme IRA pretén ser un algorisme eficient a l'hora de fer accessos molt puntuals a la taula de fets. Es diferencia principalment amb l'IFS en la forma de filtrar les keys correctes: si l'IFS llegeix tota la taula de fets i aleshores filtra les keys en qüestió, l'IRA consulta una a una (o rang a rang si hi ha subconjunts de keys consecutives) les tuples de la taula de fets on les keys corresponen a les keys retornades per l'índex.

Es pot definir aquest algorisme com un algorisme de **data shipping**. Això és principalment al fet que el MapReduce que executa aquest algorisme no s'emplaça sobre els nodes que contenen les dades i per tant no aprofita la localitat espacial. Aquest fet implica que les dades llegides han de ser mogudes cap als nodes sobre els que corre el MapReduce.

La figura *5.4-3* mostra la execució d'aquest algorisme amb més detall. Inicialment, durant el primer MapReduce, es fa la selecció mitjançant l'algorisme vist a la secció *5.3 Accés als índexs. Selecció*. En aquest cas, però, l'escriptura de les keys a accedir es fa sobre una taula temporal en comptes d'un fitxer. Posteriorment, i abans de llençar un segon MapReduce, s'intenta simplificar aquest llistat tot identificant subconjunts de keys consecutives. D'aquesta manera s'intenta explotar el paral·lelisme ofert per la pròpia API de l'HBase, que permet definir scans per rang.

L'algorisme és el mostra al pseudocodi *5.4-4 Preprocessament de la taula temporal.*

Algorisme 5.4-4: Preprocessament de la taula temporal

```
action helper(var first, var key, array deletes[0 .. D], array puts[0 .. P])
begin
  var it := first
  while it < first + key
  do
    add(deletes, it)
  done
  var newKey := concat(first, ''−'', first + keys)
  var put := createPut(newKey)
  addValue(put, ''range'', ''true'')
  add(puts, put)
end

action preprocessTemporalTable()
begin
  var scan := createScan(TEMPORAL_TABLE)
  array puts[0 .. P] := EMPTY_ARRAY
  array deletes[0 .. D] := EMPTY_ARRAY
  var first := −1
  var dist := 0
  for key in launchScan(scan)
  do
    if first = −1
    then
      first := key
    else
      if key = first + dist
      then
        dist := dist + 1
      else
        if dist > 1
        then
          helper(first, key, deletes, puts)
        fi
      fi
    fi
  done
  if dist > 1
  then
    helper(first, key, deletes, puts)
  fi
end
delete(TEMPORAL_TABLE, deletes)
put(TEMPORAL_TABLE, puts)
```

En aquest cas, es suposa un disseny de les keys consecutiu i per tant la funció que identifica un rang és l'increment. En qualsevol altre cas aquesta funció hauria de correspondre amb el disseny de les keys fet.

Així doncs, l'estructura d'aquest algorisme és la següent. Inicialment s'inicialitzen les corresponents variables. En aquest punt és important remarcar les variables *scan, puts* i *deletes*. La variable *scan* és l'objecte que es farà servir per llegir la taula temporal, i les variables *puts* i *deletes* són dos buffers que permeten salvar els canvis modificats fins al

final de l'algorisme, que és quan s'aplicaran.

Aleshores, es produeix la lectura de la taula temporal i es processa tot identificant rangs de keys consecutives. Per a fer-ho, és salva la distància entre la primera key del rang i la última processada. Mentre la següent key a processar sigui aquesta distància més una unitat, totes aquestes keys intermitjes s'afegeixen al buffer d'esborrats. Quan es detecta que la següent key no pertany a aquest rang, s'afegeix al buffer d'insercions el nou rang a inserir.

Finalment, a la sortida de l'algorisme es materialitzen els canvis.

Tornant al flux d'execució de l'IRA, aleshores s'executa un segon MapReduce que llegeix d'aquesta taula temporal i fa accessos a la taula de fets mitjançant l'API de l'HBase. Aleshores, emet les dades obtingudes com a resultat de cada un dels maps per tal que l'agrupació i l'agregació la facin els reduces, igual que amb la resta d'algorismes.

La part més negativa d'aquest algorisme és el fet que el segon MapReduce s'executa sobre la taula temporal. Com que aquesta taula temporal no deixa de ser un simple llistat de keys, el volum de la mateixa és més aviat petit i per tant la partició horitzontal és petita. Això implica un paral·lelisme molt pobre, ja que, malgrat es pugui paral·lelitzar l'accés a les dades mitjançant l'API de l'HBase, aquests accessos es fan relativament en sèrie degut al fet que es llencen sobre un MapReduce molt poc paral·lel. En altres paraules, la part negativa està en el procés de data shipping.

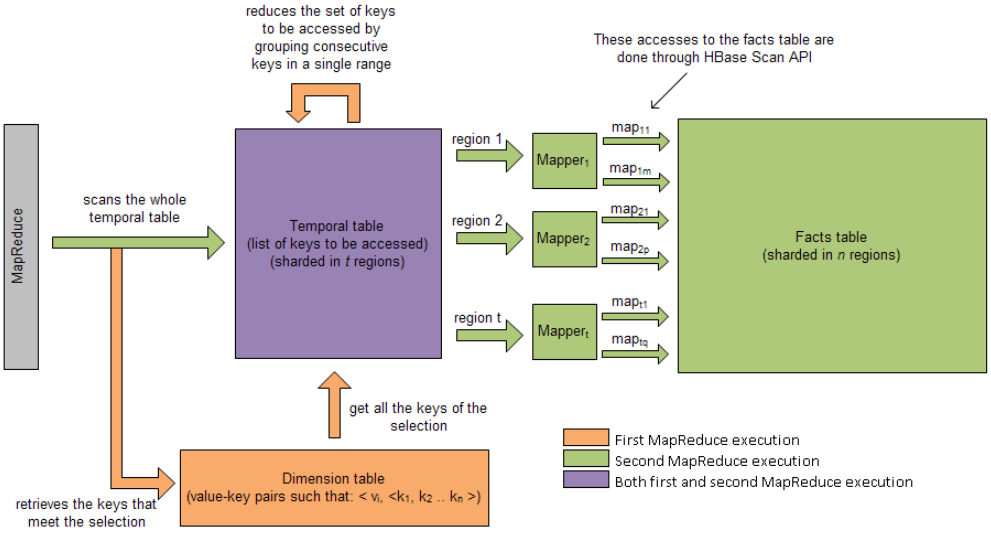


Figura 5.4-3: Execució de l'IRA

Finalment, el pseudocodi que segueix aquest procés és el mostrat als algorismes *5.4-5 Index Random Access - Part I* i *5.4-6 Index Random Access - Part II*.

Algorisme 5.4-5: Index Random Access - Part I

```
var facts_scan

action launch(var temporal)
begin
  var scan := createScan(temporal, ''range'')
  executeMapReduce(scan)
end

action startup(array<key, value>[0 .. R] rows)
begin
  array projection[0 .. P] := getMapReduceConfiguration(''projection'')
  array groupby[0 .. G] := getMapReduceConfiguration(''groupby'')
  facts_scan := createScan(FACTS_TABLE, projection, groupby)
  for row in rows
  do
    map(row)
  done
end
```

Algorisme 5.4-6: Index Random Access - Part II

```
action pre_emit(var<key, value> result)
begin
  array projection[0 .. P] := getMapReduceConfiguration(''projection'')
  array groupby[0 .. G] := getMapReduceConfiguration(''groupby'')
  var groupby_values := EMPTY_STRING
  for g in groupby
  do
    var v := getValue(value, g)
    groupby_values := concat(groupby_values, ''_'', v)
  done
  for p in projection
  do
    var k := concat(groupby_values, ''_'', p)
    var v := getValue(value, p)
    emit(k, v)
  done
end

action map(var<key, value> row)
begin
  var is_range := getValue(value, ''range'')
  if is_range
  then
    array keys[0 .. K] := getRange(key)
    array<k, v> result[0 .. KV] := launchScan(scan, key[0], key[K])
    for r in result
    do
      pre_emit(r)
    done
  else
    var<k, v> result := launchScan(scan, key)
    pre_emit(result)
  fi
end

action reduce(var<key, value[0 .. V]> result)
begin
  var aggregate := 0
  for v in value
  do
    aggregate := aggregate + v
  done
  emit(key, aggregate)
end
```

L'algorisme mostra com el procés inicial necessita un scan addicional per a l'accés a la taula de fets. En realitat, aquest scan es pot definir internament a cada map, però fent-lo global el que es busca és que sigui únic per cada Mapper i per tant pugui ser compartir per cada un dels maps que executa. L'objectiu d'això és evitar malgastar excessiva memòria. A més, també es pot veure com en aquest algorisme també es reimplementa la funció de inicialització *start up*, ja que és en aquest moment on cal configurar l'scan.

Per altra banda, la funció de map és molt simplista. Senzillament s'encarrega de fer

l'accés a les dades i emetre-les cap al Reducer. En primer lloc, aquesta funció de map ha d'identificar si la tupla actual és tracta d'un rang o no. Això és degut al preprocessament anterior que agrupa keys consecutives en un únic rang. Aleshores, per tal de comprovar aquest fet:

- Inicialment es feia tot comprovant la llargada de la key. Com que aquestes keys són de llargada fixa, es feia fàcil comprovar que si la key de la tupla actual era més llarga que la mida fixe, aleshores es tractava d'un rang. Per exemple, una key del tipus *000000023t09* denota una única key, mentre que *000000023t09-000000023t12* denota un rang.
- Més tard es va decidir modificar aquesta verificació tot aprofitant l'atribut "range" que ja determina si es tracta d'un rang o no. Durant el preprocessament de la taula temporal, el sistema per si ja construeix un atribut booleà anomenat "range" que indica si la tupla en qüestió representa un rang de keys o no. Es va fer aquesta modificació per tal de donar més llibertat a que es puguin implementar les keys d'una altra manera. Com que aquesta modificació va ser molt simple, es menciona en aquest apartat en comptes d'explicar-se pròpiament a l'apartat d'implementacions.

Aleshores, identificat si es tracta d'un rang de keys o no, només cal muntar la parella key-value a emetre correctament i enviar-la al Reducer. En cas de tractar-se d'un rang cal fer-ho tants cops com keys contingui el rang, en cas contrari només cal fer-ho un cop. Aquest és l'objectiu de la funció auxiliar *pre_emit*: encarregar-se de muntar i emetre el key-value correctament. La justificació d'aquesta funció és evitar repetir codi.

Finalment, es produeix l'agregació i l'agrupació en el Reducer de forma habitual.

6 Metodologia en el desenvolupament del software

L'objectiu principal d'aquest projecte és executar un conjunt d'experiments que permetin determinar un model d'optimització MapReduce. No obstant això, per tal de poder llençar aquests experiments cal primerament dur a terme un seguit d'implementacions (veure apartat *7 Implementació*). Això implica la decisió de quin tipus de metodologia es seguirà durant el desenvolupament d'aquestes implementacions per tal de garantir la qualitat del software resultant.

En aquest projecte la metodologia triada ha estat una metodologia àgil. Aquest tipus de metodologies estan pensades per desenvolupar software d'una forma més o menys fluida. Es basa principalment en la repetició d'iteracions:

1. La primera iteració consisteix en una fase de planificació i de disseny de la funcionalitat o funcionalitats a desenvolupar en la iteració actual.
2. A continuació, es posen a prova les funcionalitats desenvolupades per tal de deixar-les lliures d'errors.
3. En aquest moment es presenten els resultats de la iteració al client per tal de validar els resultats obtinguts i poder corregir.
4. En cas que el client accepti la iteració, es torna a iniciar el cicle amb la iteració següent.

La figura *6.0-4* mostra amb més claredat tot aquest procés.

Figura 6.0-4: Diagrama d'una metodologia àgil

Una metodologia d'aquest estil és útil en diferents situacions que no són remarcables per a aquest, excepte el fet que són metodologies òptimes per tal de tenir una continuïtat sobre la validesa dels resultats obtinguts. Això és així gràcies a l'etapa de feedback, ja que permet que el client tingui una visió constant del treball realitzat i per tant pugui avaluar si els seus requisits s'estan complint.

En el cas d'aquest projecte, aquesta ràpida valoració dels resultats és fonamental ja que l'objectiu de les implementacions és superar les dificultats que es coneixen del projecte anterior i per tant amb aquest tipus de metodologia queda a l'avast la opció de canviar ràpidament de plantejament en cas que els feedbacks obtinguts no assenyalin cap a una clara millora respecte el projecte anterior.

7 Implementació

7.1 Etapa I: Població de l'HBase. Generador

7.1.1 L'esquema de dades. TPC-H

La primera fase del projecte va consistir en poblar l'HBase amb les dades que posteriorment serien utilitzades per generar els cubs de dades.

Partint de l'experiència del projecte previ, sabíem que aquest generador de dades s'havia de fer d'una forma eficient per tal d'evitar els mateixos problemes de memòria i timeout que van aparèixer en el seu moment en el clúster del DAC. A més, volíem seguir fent servir el *benchmark* TPC-H com a model per a les dades. Aquest benchmark es defineix de la següent manera:

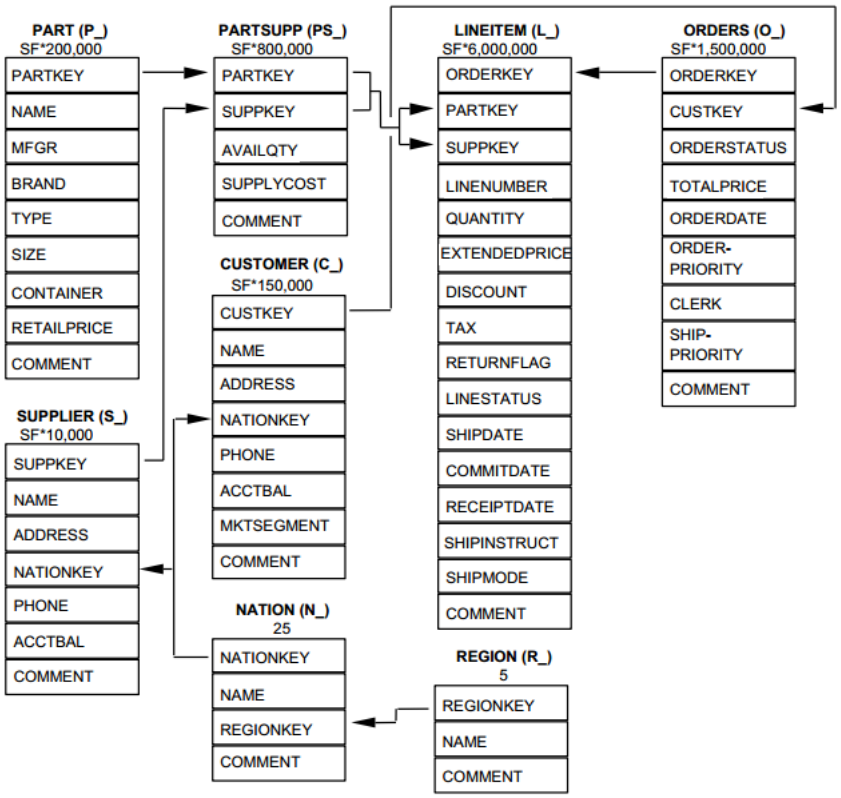


Figura 7.1-1: Esquema del TPC-H

La figura *7.1-1* s'ha de llegir de la forma:

- La primary key de cada taula és el primer atribut que es llista, excepte per les taules *lineitem* i *partsupp*, on les primary keys són la composició dels atributs *orderkey* i *linenumber*, i *partkey* i *suppkey*, respectivament.
- Damunt de cada taula és mostra el nombre de tuples que es generen per aquella taula en funció de l'*scale factor*[23] (abreviat SF d'ara endavant). Així, la taula *partsupp* segueix una funció $\#tuples = 800000SF$, lo qual significa que donat per exemple un $SF = 0.5$, es generaran 400000 tuples per a aquella taula.

[23] L'*scale factor* és un paràmetre d'entrada del TPC-H que defineix el volum de dades a generar. Per exemple, un *scale factor* de 1 significa 1GB de dades, o un *scale factor* de 0.3 significa 300MB de dades.

Les taules *nation* i *region* generen un nombre de tuples prefixat i independent del SF. Aquest nombre és 25 per la taula *nation* i 5 per la taula *region*.

- Les fletxes indiquen les foreign keys.

L'inconvenient de fer servir aquest benchmark sorgeix a l'hora de traduir aquest esquema a l'HBase. El TPC-H implementa un esquema relacional i normalitzat mentre que l'HBase és una base de dades *schemaless*[24]. En altres paraules, per tal d'adequar el TPC-H al model de l'HBase cal, o bé denormalitzar-lo, o bé generar-lo ja desnormalitzat. Aquí rau la primera diferència entre el treball anterior i el meu: prèviament es va apostar per denormalitzar-lo tot fent servir MapReduce, i en el meu cas vam decidir nodrir-nos de l'experiència obtinguda amb el procés de denormalització i dels problemes que van sorgir, i vam apostar per generar-lo de nou a través de l'ús de threads per tal de fer-lo eficient en quant a temps. Tot i així, als apartats següents es veurà com la generació del TPC-H ja desnormalitzat tampoc és trivial, i donat que:

a) La població de l'HBase s'havia de fer en una finestra de temps acceptable, evitant així els timeouts del clúster i,

b) Aquesta generació s'ha de fer degut a que es necessari poblar l'HBase per tal de poder fer proves, però en realitat no s'inclou dins dels objectius d'aquest projecte i per tant havia d'acotar el temps dedicat a aquesta etapa,

el TPC-H generat es va simplificar en la mesura de lo possible quant a atributs. La cardinalitat de les relacions es va mantenir intacte.

El model desnormalitzat del TPC-H consisteix en una única taula que contingui, per cada tupla, tots els atributs que apareixen en totes les taules del model normalitzat. Un exemple de denormalització es pot trobar a les figures *2.6-1* i *2.6-2* de l'apartat *2.6 Denormalització*

No obstant això, abans de prendre la decisió de generar el TPC-H desnormalitzat, vaig provar les següents alternatives per intentar millorar el procés de denormalització:

Tenint en compte que la sortida del TPC-H és un fitxer de text on cada línia representa la tupla d'una taula:

1. La meva idea inicial per denormalitzar era generar cada taula en un fitxer diferent, indexar la posició d'aquella tupla dins del fitxer i simular un *row nested loops* amb fitxers. D'aquesta manera podia evitar els primers n bytes sense haver de llegir-los. No obstant, això no va significar la millora que esperava ja que denormalitzar les primeres tuples era relativament ràpid perque estaven molt a prop de l'inici del fitxer i la latència de moure el capçal del disc era més aviat petita, però amb les últimes tuples el capçal havia de moure's massa i la denormalització d'aquestes feia caure el rendiment global del desnormalitzador.

2. Llavors, vaig voler provar de trencar els fitxers on hi havien les taules en fitxers més petits. Això implicava indexar de cada tupla el fitxer on estava i la posició de la tupla respecte l'inici d'aquest nou fitxer. D'aquesta manera pretenia accelerar el procés ja que al accedir a fitxers més petits, la distància en bytes entre l'inici del fitxer i la tupla era més petita i per tant la latència de moure el capçal menor.

[24]Tipus de base de dades que no requereix d'un esquema prèviament definit.

Malauradament, els temps de denormalització continuaven sent massa grans i aleshores la opció d'implementar un nou generador que generés el TPC-H ja desnormalitzat es va valorar més en detall. Així doncs, aquest nou generador va ser la solució final aplicada i el detalls de la seva implementació es discuteixen en els apartats posteriors.

7.1.2 Requisits previs

La implementació d'aquest generador té com a base tres requisits previs que són imprescindibles. Les raons que fan que aquests requisits tinguin tanta importància són diferents entre elles, i per tant es discuteixen durant l'especificació de cada un d'aquests requisits. Aquests requisits són:

Req. 1	Implementació del TPC-H
Justificació: La implementació del TPC-H és imprescindible per tal de seguir un esquema de dades estàndard. A l'apartat *7.1.1 L'esquema de dades. TPC-H* s'explica amb més detall com és aquest esquema de dades. La única excepció en el cas d'aquest projecte és que, per evitar complicacions, s'ha simplificat la generació d'alguns paràmetres que no influeixen en la generació dels cubs de dades, com per exemple atributs que es calculen a partir de formules complexes. Aquests atributs s'han simplificat per tal de generar-los de forma més senzilla. Per altra banda, la cardinalitat de les relacions entre taules sí que afecta a la generació dels cubs de dades i per tant aquesta sí s'han mantingut originals.	

Req. 2	Localitat de les dades
Justificació: Un altre requisit important és que no hi hagi cap mena de localitat en les dades. En altres paraules, que els atributs estiguin repartits per tota la taula i no molt concentrats en un mateix punt i amb un mateix valor. Si permetem que aquesta localitat de dades tingui lloc, llavors els resultats al construir els cubs de dades no seran realistes ja que obtenir les tuples amb un determinat valor en un atribut serà un accés seqüencial[25] i no un **random accessAccés a un subconjunt de dades distribuït al llarg de tot el conjunt de dades.** com ens interessa.	

Req. 3	Optimització
Justificació: Un dels objectius d'aquest projecte és ser capaç de fer córrer les proves de rendiment sobre l'arquitectura del clúster del DAC. Aquest clúster està pensat per córrer simultàniament les aplicacions de molts usuaris i per tant, segueix un sistema de cues i on cada cua té assignat un temps d'execució i un màxim de recursos. Aleshores, la importància d'aquest requisit és deguda a que aquest generador ha de ser òptim quant a temps per tal d'assegurar que les proves finalitzen correctament dins de la finestra establerta.	

La implementació d'aquests requisits es discuteix en apartats posteriors, però és important que el lector els conegui per tal de comprendre el sentit global de la solució triada per a desenvolupar aquest generador.

7.1.3 Disseny

7.1.3.1 Casos d'ús

El diagrama de casos d'ús per a aquesta etapa del projecte és molt senzill. Només apareix un sol actor: l'usuari, que és qui inicia l'únic cas d'ús possible: la generació del TPC-H. Per tant, el diagrama de casos d'ús correspon al representat per la figura *7.1-2*.

Figura 7.1-2: Diagrama de casos d'ús

7.1.3.2 Diagrama de classes

El diagrama de classes confeccionat per a aquesta part del projecte és molt senzill. Bàsicament perquè el nucli principal del generador és simplement un conjunt de threads amb la seva respectiva associació cap a la classe encarregada d'inserir cada fila generada a l'HBase. A la figura *7.1-3* es mostra el diagrama de classes parcial corresponent als threads.

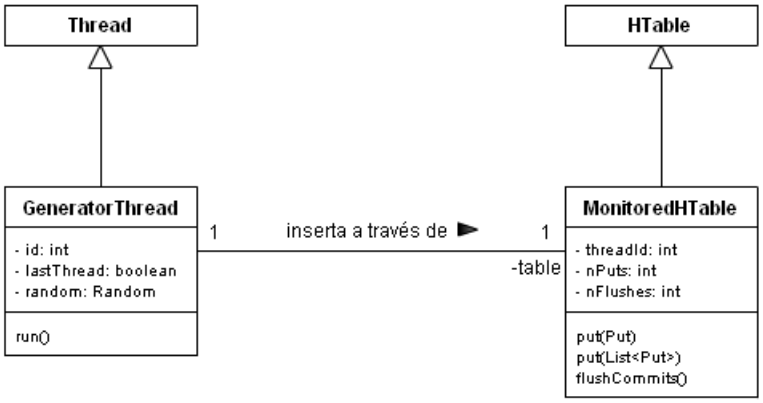


Figura 7.1-3: Diagrama de classes dels threads

La classe GeneratorThread

En Java hi han dues formes de construir una aplicació en *multithreading*: a) reescrivint el mètode *run()* de la interfície Runnable, o bé b) reimplementant la classe Thread. En el cas d'aquest projecte, es va decidir fer-ho a través de la classe Thread ja que permet més fàcilment declarar variables locals a un thread. És per aquest motiu que la classe GeneratorThread hereta de la classe Thread de la API de Java i sobreescriu el mètode *run()*. El significat dels atributs d'aquesta classe és:

Nom atribut	Significat
id	Atribut que identifica cada thread de forma interna al generador. Aquesta identificació es fa mitjançant un nombre enter i correlatiu. Així, $id_{thread1} = 0, id_{thread2} = 1, ..., id_{threadn} = n - 1$
lastThread	Aquest és un atribut de tipus booleà. Indica si aquest thread és l'últim de tots els threads que es crearan en una execució. D'aquesta manera, en cas que la càrrega de tuples a inserir no sigui divisible pel nombre de threads totals, aquest últim thread s'encarregarà també d'inserir la càrrega restant.
random	Aquest és un objecte Java que genera els nombres aleatoris necessaris per generar el TPC-H. La justificació de generar aquests nombres aleatoris mitjançant aquest objecte és que aquesta classe permet assignar la llavor a partir de la qual es generen els nombres aleatoris. Suposem que els threads T_i i T_j on $i \neq j$ generen les *orders* O_i i O_j respectivament, i aquestes *orders* les fa un mateix *customer* C_k (veure figura *7.1-1*). Si cada thread generés C_k de forma diferent, llavors tindríem que les O_i i O_j estarien relacionades pel mateix *customer* C_k, però aquest C_k podria prendre valors diferents per alguns dels seus atributs en tuples diferents i per tant estaríem conduint a l'HBase a un estat d'inconsistència.
table	Aquest atribut és una associació cap a un objecte de tipus MonitoredHTable (veure més avall) mitjançant el qual es fan les insercions a l'HBase.

La classe MonitoredHTable

La classe MonitoredHTable hereta de la classe de l'API de l'HBase que insereix les dades i simplement reescriu alguns dels mètodes d'inserció per tal de monitoritzar el nombre d'insercions que fa cada thread. El significat dels atributs és:

Nom atribut	Significat
threadId	Aquest atribut pretén identificar quina instància de la classe MonitoredHTable correspon a cada instància de la classe GeneratorThread.
nPuts	És un simple valor enter que conté el nombre total d'insercions fetes a través del propi objecte MonitoredHTable. Degut a que una instància de la classe MonitoredHTable correspon a un, i només a un thread, aquest atribut indica el nombre total d'insercions fetes per aquell thread.
nFlushes	Valor enter que indica el nombre de flushes que s'ha fet del buffer local del client (veure apartat d'optimització *7.1.6 Requisit III: Optimització*).

De forma complementària a aquesta lògica de negoci, apareixen les següents classes que contenen paràmetres globals de l'aplicatiu i que permeten als threads poder compartir certes dades[26]. A la figura *7.1-4* es mostren aquestes classes.

[26] Aquestes són dades de lectura que queden ja inicialitzades abans d'arrencar els threads. D'aquesta

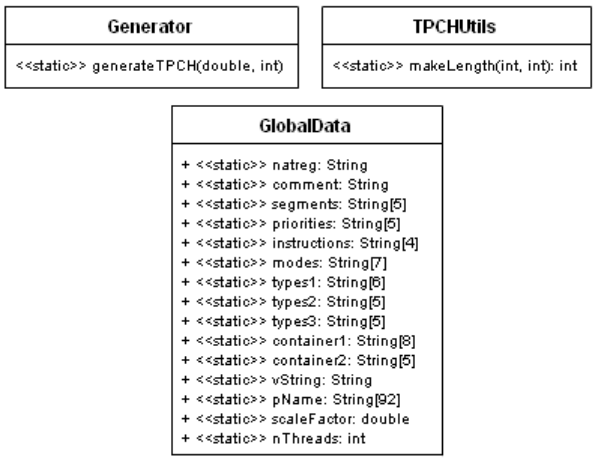


Figura 7.1-4: Classes estàtiques del generador

És important notar que aquestes classes encapsulen atributs/mètodes de tipus estàtic, lo qual significa que es deixa de banda la programació orientada a objectes i que aquests atributs no poden ser instanciats. L'objectiu de declarar aquests atributs com a estàtics és que poden ser compartits per tots els threads sense necessitat d'acoblaments extres. Un plantejament alternatiu podria haver estat la de fer servir un patró *singleton*. Aquesta hauria estat l'alternativa escollida si entre els requisits d'aquest generador haguessin estat la portabilitat o la mantenibilitat, però en aquest cas l'únic requisit és l'eficiència (tant de memòria com de temps) i per tant, ha calgut deixar de banda patrons de disseny que afegeixen overhead a l'execució degut a crides a funcions extres.

La classe Generator

Aquesta classe és el fil principal de l'execució. Conté una única funció estàtica que rep com a paràmetres el nombre de threads i el SF, a continuació genera els paràmetres corresponents per a la generació del TPC-H, crea els threads, els arrenca i espera que finalitzin tots abans de finalitzar la seva pròpia execució.

La classe GlobalData

La justificació d'aquesta classe és contenir tots aquells atributs/variables que seran accessibles des de qualsevol thread. Té la salvedat que aquests són exclusivament de lectura i per tant no cal cap mena de control de concurrència sobre ells mateixos. Així doncs, aquesta classe conté paràmetres globals de l'aplicació com el SF o el nombre de threads però també atributs base per a la generació de cada un dels atributs del TPC-H.

La classe TPCHUtils

L'objectiu inicial d'aquesta classe era contenir totes aquelles funcions que poguessin ser d'utilitat de forma comuna a tots els threads. No obstant això, només va resultar d'utilitat una única funció estàtica encarregada de transformar un nombre enter donat en un string longitud fixa (veure apartat de localitat de dades *7.1.5 Requisit II: Localitat de les dades*).

manera, poden ser compartides sense necessitat de fer cap control de concurrència.

Així doncs, queden justificats els motius els quals m'han portat a dissenyar aquesta solució. Cal dir que per tal de satisfer els requisits previs es van dissenyar prèviament altres solucions que buscaven millorar el rendiment i/o la localitat de les dades. Aquestes alternatives es comenten als apartats corresponents.

7.1.4 Requisit I: Implementació del TPC-H

Tal i com s'ha justificat a l'apartat 7.1.2, el primer requisit d'aquest generador era seguir l'esquema de dades presentat pel TPC-H. Quan es parli d'objectes en aquest apartat, cal tenir present que es refereix a cada una de les taules del model normalitzat. En altres paraules, una tupla en el model desnormalitzat conté diferents objectes corresponents a cada una de les taules del model normalitzat: *lineitem, orders, customer, supplier...*

7.1.4.1 Terminologia

En aquest apartat es mostren els diferents tipus de dades que conformen el TPC-H. Per tal de facilitar la comprensió de cada tipus, aquest apartat es presenta mostrant la definició de cada tipus, extreta literalment de la documentació del TPC-H, i a continuació la justificació de l'algorisme i de les decisions preses.

1. The term **random** means independently selected and uniformly distributed over the specified range of values.

   Aquest tipus de dades està definit al TPC-H, però en realitat cap dada és d'aquest tipus. Els nombres aleatoris es generen a partir dels tipus de dades definits més abaix, on molts d'ells són una petita extensió d'aquest tipus de dades, ja que generen nombres aleatoris amb característiques addicionals. Per exemple, generar un enter aleatori dins d'un conjunt, o generar un string a partir de la concatenació aleatòria d'altres strings, etc.

2. The term **unique within [x]** represents any one value within a set of x values between 1 and x, unique within the scope of rows being populated.

   No hi ha cap algorisme propi per a aquest tipus de dades. Per tal de simplificar la generació de valors únics, la justificació que s'ha seguit és la de fer aquesta generació de forma consecutiva tal que $1..x$ (inclosos). D'aquesta manera, queda garantida la no repetició d'aquests valors amb el mínim cost computacional.

3. The notation **random value [x .. y]** represents a random value between x and y inclusively, with a mean of $(x + y)/2$, and with the same number of digits of precision as shown. For example, [0.01..100.00] has 10,000 unique values, whereas [1..100] has only 100 unique values.

   Aquest tipus de dades s'ha generat fent servir la classe pròpia de Java per tal de generar nombres aleatoris. No obstant això, és possible que una mateixa dada d'aquest tipus s'hagi de generar més d'un cop degut al model desnormalitzat resultant, i per tant, per tal de garantir que aquesta mateixa dada pren el mateix valor sempre, la generació de nombres aleatoris es fa tot seguint la mateixa llavor en tots els cops que s'ha de generar una mateixa dada. La funció que s'encarrega d'això és molt senzilla i és la següent:

Algorisme 7.1-1: Generació de nombres aleatoris

```
function generateInteger(var iId, var min, var max)
begin
  var r := random(iId) % (max − min + 1)
  return r + min
end
```

Les variables *min* i *max* indiquen el rang de valors entre els quals (inclosos) ha de trobar-se el nombre generat. La funció *random(llavor)* retorna un nou nombre natural pseudoaleatori a partir de la llavor que rep com a paràmetre. Al aplicar a aquest nombre aleatori la operació mòdul amb la diferència entre els dos límits del rang de valors i sumar-li 1, el resultat és un nombre aleatori entre zero i aquesta diferència. Finalment, sumant el límit inferior del rang s'aconsegueix que el valor final retornat per la funció estigui dintre del rang desitjat amb tots dos límits inclosos.

La restricció de que els nombres generats han de seguir la mitjana $(x+y)/2$ ha quedat coberta per la uniformitat amb la que Java genera els nombres aleatoris. De totes maneres, aquesta restricció podria haver estat obviada a raó de que no és una restricció clau per a aquest generador i per tant pot ser simplificada, ja que aconseguint que les dades d'aquest tipus prenguin diferents valors al llarg de tota la taula podria haver estat suficient.

Per altra banda, també és possible generar valors reals a partir d'aquesta clàusula. El propòsit inicial era generar-los a partir d'aquest algorisme com a nombres enters i aleshores dividir-los en funció de la potència de la base que correspongués. Per exemple, si l'atribut a generar ha d'estar comprès entre 1 i 10 i amb possibilitat de contenir dos decimals ($10^2 = 100$, només cal aplicar aquest algorisme per tal de generar un nombre natural aleatori entre $1 * 100 = 100$ i $10 * 100 = 1000$ i dividir el nombre obtingut per 100.

No obstant això, les operacions amb nombres reals acostumen a ser costoses. Especialment la multiplicació o la divisió. És degut a això, el motiu pel qual la generació de nombres reals aleatoris es genera finalment a partir d'un algorisme més especialitzat. Degut a que es tracta d'una optimització pròpia d'aquest generador, els detalls es poden consultar a l'apartat 7.1.6.

4. The notation **random string [list_name]** represents a string selected at random within the list of strings *list_name* as defined in Clause 4.2.2.13[27]. Each string must be selected with equal probability.

A partir de l'algorisme 7.1-1 podem generar aquest nou tipus de dades. Donades diferents llistes de strings, aquesta clàusula busca la generació d'un string aleatori a través de la concatenació de strings de diferents llistes. El següent algorisme mostra aquesta generació a partir de tres llistes diferents:

[27] Aquesta clàusula són llistes de strings predefinits per a generar nous strings a partir de la seva combinació.

Algorisme 7.1-2: Generació de strings aleatoris a partir de concatenació

```
array listA[0 .. NA − 1] := {wordA1, wordA2 .. wordAN}
array listB[0 .. NB − 1] := {wordB1, wordB2 .. wordBN}
array listC[0 .. NC − 1] := {wordC1, wordC2 .. wordCN}
function generateConcatString(var id)
begin
  var a := listA[generateInteger(id, 1, NA) − 1]
  var b := listB[generateInteger(id + 1, 1, NB) − 1]
  var c := listC[generateInteger(id + 2, 1, NC) − 1]
  var random := concat(a, b, c)
  return random
end
```

Aleshores, cal suposar que prèviament tenim definits dos tipus de variables diferents:

(a) Un identificador que permeti fer de llavor per generar nombres aleatoris. En aquest cas, es tracta de la variable *id* que arriba com a paràmetre d'entrada.

(b) Un seguit de llistes de strings que contenen els strings a concatenar. En aquest cas són les variables $listA, listB$ i $listC$

Dit això, és fàcil entendre que el que fa l'algorisme no es més que generar un nombre aleatori més petit o igual que la longitud de la llista, i obtenir l'string en aquella posició. Per tal d'evitar que en totes tres llistes la posició accedida sigui la mateixa, la llavor s'incrementa per a cada llistat.

5. The notation **text appended with digit [text, x]** represents a string generated by concatenating the sub-string text, the character "#", and the sub-string representation of the number x.

Aquest és un tipus de dades molt fàcil d'implementar ja que només es tracta d'una simple concatenació de strings, i per tant no hi ha cap algorisme específic per a les dades d'aquest tipus.

6. The notation random **v-string [min, max]** represents a string comprised of randomly generated alphanumeric characters within a character set of at least 64 symbols. The length of the string is a random value between min and max inclusive.

La generació d'aquest tipus de dades es fa mitjançant un string base. Aquest string base és un string fixe al qual se li aplica un natural n generat aleatòriament que indica la longitud de l'string a generar. Finalment, s'aplica una operació de substring per obtenir l'string generat a partir del primer i el n-èsim caràcter d'aquest string base. El següent algorisme mostra aquesta generació amb més detall:

Algorisme 7.1-3: Generació de strings de longitud variable

```
var vString := ... // String base prefixat
function generateVString(var id, var min, var max)
begin
  var length := generateInteger(id, min, max)
  return substring(vString, 0, length)
end
```

La resta de restriccions d'aquest tipus de dades es poden obviar ja que la finalitat d'aquestes dades, en aquest projecte, és afegir espai a les tuples per tal de fer-les més costoses de llegir.

La funció *substring(str, inici, final)* retorna el substring obtingut entre les posicions *inici* i $final$ (inclosos) de l'string str.

7. The term **date** represents a string of numeric characters separated by hyphens and comprised of a 4 digit year, 2 digit month and 2 digit day of the month.
All dates must be computed using the following values:

STARTDATE = 1992-01-01 CURRENTDATE = 1995-06-17 ENDDATE = 1998-12-31

L'elaboració d'aquestes dades es detalla al següent algorisme:

Algorisme 7.1-4: Generació de dates

```
array daysOfMonth := {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
function generateIntervalDate(var id)
begin
  // Any
  var year := ((id / 365) % (1998 − 1992 + 1)) + 1992
  // Mes
  var i := id
  var month := 0
  while i >= daysOfMonths[month % 12]
  do
    i := i − daysOfMonths[month % 12];
    month++;
  done
  var month := (month % 12) + 1
  // Dia
  var day := i + 1
  return {year, month, day}
end
```

Aquest algorisme es basa en simular que el natural que es rep d'entrada és el nombre de dies transcorreguts a partir de la data inicial. Aleshores, la generació d'aquestes dates es fa fent la conversió d'aquest total de dies a un format *yyyy-mm-dd*, tot tenint en compte que no es pot superar la data marcada com a màxima pel TPC-H (ENDDATE):

- El càlcul de l'any és una simple operació algebraica. A través de la divisió, s'obté el nombre d'anys transcorreguts (no es contemplen els anys de traspàs). Per tal de convertir-lo en un any vàlid pel TPC-H, se li aplica la operació mòdul amb el nombre d'anys permesos i finalment es fa una suma

amb l'any vàlid més petit (STARTDATE).

- La generació del mes es fa seguint un bucle que itera repetidament tot restant, en cada iteració, el nombre total de dies de cada mes. Així, la primera iteració resta, al nombre total de dies, el número de dies que té gener, la segona iteració resta els dies de febrer... I així successivament fins que el nombre de dies restants no sigui més gran o igual que els del següent mes. En aquest punt, el mes resultant correspon a la operació *nombre d'iteracions % 12*, on % és la operació mòdul.
- Finalment, el dia equival a la quantitat de dies totals que resten de la última iteració del bucle anterior.

8. The term **phone number** represents a string of numeric characters separated by hyphens and generated as follows:

Let i be an index into the list of strings Nations (i.e., ALGERIA is 0, ARGENTINA is 1, etc., see Clause 4.2.3[28]),
Let *country_code* be the sub-string representation of the number $(i + 10)$,
Let *local_number*1 be random [100..999],
Let *local_number*2 be random [100..999],
Let *local_number*3 be random [1000..9999],
The phone number string is obtained by concatenating the following sub-strings:

country_code, "-", *local_number*1, "-", *local_number*2, "-", *local_number*3

La generació d'aquest tipus de dades és molt senzilla. Simplement es tracta de fer la generació de tres nombres aleatoris: els dos primers dins de l'interval [100..999 i el tercer dins de l'interval [1000..9999]. La obtenció del prefix es fa mitjançant la mateixa funció que calcula la zona geogràfica donat l'identificador de l'objecte. L'algorisme és el descrit a continuació:

[28](0, ALGERIA), (1, ARGENTINA), (2, BRAZIL), (3, CANADA), (4, EGYPT), (5, ETHIOPIA, 0), (6, FRANCE), (7, GERMANY), (8, INDIA), (9, INDONESIA), (10, IRAN), (11, IRAQ), (12, JAPAN), (13, JORDAN), (14, KENYA, 0), (15, MOROCCO, 0), (16, MOZAMBIQUE, 0), (17, PERU), (18, CHINA), (19, ROMANIA), (20, SAUDI ARABIA), (21, VIETNAM), (22, RUSSIA), (23, UNITED KINGDOM), (24, UNITED STATES).

Algorisme 7.1-5: Generació de números de telèfon

```
function generateNation(var id)
begin
  return id % 25
end

function generatePhone(var id)
begin
  var rand1 := random(id)
  var rand2 := random(id + 1)
  var rand3 := random(id + 2)
  var phoneNumber1 := abs((rand1 % (999 + 1 − 100)) + 100)
  var phoneNumber2 := abs((rand2 % (999 + 1 − 100)) + 100)
  var phoneNumber3 := abs((rand3 % (9999 + 1 − 1000)) + 1000)
      var phone := concat(
    generateNation(id) + 10, "−",
    phoneNumber1, "−",
    phoneNumber2, "−",
    phoneNumber3)
  return phone
end
```

D'aquesta manera, assegurem que tots els objectes d'una mateixa zona geogràfica tindran el mateix prefix de telèfon, ja que el càlcul d'aquest prefix i de la zona geogràfica és la mateixa funció i per tant, donat l'identificador d'aquest objecte, aquesta funció sempre retornarà el mateix valor. No obstant això, els prefixos generats no corresponen als indicats en aquesta clàusula (p.ex: ALGERIA és el 0, ARGENTINA l'1, etc.), però aquest és un detall que pot ser perfectament substituït per qualsevol altra relació *prefix-zona geogràfica*. La suma final de 10 unitats a aquest prefix és per tal de simplificar el fet que tots els prefixos tinguin dos dígits (hi ha un màxim de 25 possibilitats en total).

Per altra banda, tal i com es veu a l'algorisme anterior, la part aleatòria del telèfon es genera a partir de la funció de generació de nombres aleatoris de Java. La operació matemàtica que es veu a posteriori converteix l'enter retornat en un nombre positiu (la funció *abs()* correspon al valor absolut) pertanyent a l'interval corresponent, tot seguint la mateixa estratègia amb la operació mòdul que la generació d'altres tipus de dades vistes anteriorment.

9. The term **text string[min, max]** represents a substring of a 300 MB string populated according to the pseudo text grammar defined in Clause 4.2.2.14[29]. The length of the substring is a random number between min and max inclusive. The substring offset is randomly chosen.

La generació d'aquest tipus de dades es fa mitjançant un string fixe de llargada suficient per generar totes les dades d'aquest tipus, però sense arribar als 300 MB que s'expliciten en aquest clàusula. Aquest tipus de dades s'utilitza essencialment per engrandir l'espai ocupat per les tuples. En aquest sentit, la implementació d'aquest tipus de dades es podria haver reduït perfectament a la concatenació d'un mateix caràcter n vegades. No obstant això, la decisió final va ser fer aquesta implementació el més semblant possible al TPC-H per tal de

[29]El text originat en la creació de dades ha de seguir una certa gramàtica.

fer una simulació millor.

Així doncs, l'algorisme que dóna lloc a aquest tipus de dades és el següent:

Algorisme 7.1-6: Generació de comentaris

```
var fixedComment := ...
var generateComment(var cId, var min, var max)
begin
  var commentLength := random(cId) % (max − min + 1) + min
  var commentOffset := random(cId + 1) % fixedComment.LENGTH
  if (commentOffset + commentLength) > fixedComment.length()
  then
    var sub1 := substring(fixedComment, commentOffset, fixedComment.LENGTH − 1)
    var sub2 := substring(fixedComment, 0, commentLength −
      (fixedComment.LENGTH − commentOffset)
    var comment := concat(sub1, sub2)
  else
    var comment := substring(fixedComment, commentOffset,
      commentOffset + commentLength)
  fi
  return comment
end
```

La dificultat més gran d'aquest algorisme és controlar que l'offset inicial més la longitud d'aquest, paràmetres a partir dels quals es construeix l'string resultant, no sigui més gran que la longitud total de l'string prefixat. En aquest cas, cal fer la concatenació dels caràcters que resten des de l'offset obtingut fins al final de l'string amb el substring resultant de l'inici de l'string prefixat i el nombre de caràcters que falten per generar l'string final. En cas contrari, simplement es retorna el substring corresponent.

10. The following list of strings must be used to populate the database:

List name: **Types**
Each string is generated by the concatenation of a variable length syllable selected at random from each of the three following lists and separated by a single space (for a total of 150 combinations).

Syllable 1	Syllable 2	Syllable 3
STANDARD	ANODIZED	TIN
SMALL	BURNISHED	NICKEL
MEDIUM	PLATED	BRASS
LARGE	POLISHED	STEEL
ECONOMY	BRUSHED	COPPER
PROMO		

List name: **Containers**
Each string is generated by the concatenation of a variable length syllable selected at random from each of the two following lists and separated by a single space (for a total of 40 combinations).

Syllable 1	Syllable 2
SM	CASE
LG	BOX
MED	BAG
JUMBO	JAR
WRAP	PKG
	PACK
	CAN
	DRUM

List name: **Segments**

AUTOMOBILE	BUILDING	FURNITURE
HOUSEHOLD	MACHINERY	

List name: **Priorities**

1-URGENT	2-HIGH	3-MEDIUM
4-NOT SPECIFIED	5-LOW	

List name: **Instructions**

DELIVER IN PERSON	COLLECT COD	NONE
TAKE BACK RETURN		

List name: **Modes**

REG AIR	AIR	RAIL
SHIP	TRUCK	MAIL
FOB		

Aquestes són les llistes de strings que el TPC-H fa servir per generar els strings aleatoris vist en els punts anterior. Així doncs, aquest generador fa servir aquestes mateixes llistes en el seus algorismes.

7.1.4.2 Generació dels atributs

En aquest apartat es mostren els algorismes que s'han fet servir per a la generació dels atributs. En la majoria dels casos, aquests atributs no afegeixen cap característica addicional i per tant la seva generació correspon a algun dels algorismes detallats a l'apartat anterior. Quan es produeixi aquest cas, simplement s'indicarà quin algorisme dels anteriors li correspon. En cas contrari, es detallarà un nou algorisme.

- SF * 10,000 rows in the SUPPLIER table with:
  - S_SUPPKEY **unique within [SF * 10,000]**.

    Es genera tal i com s'indica al punt 2 de l'apartat anterior.

  - S_NAME **text appended with minimum 9 digits** with leading zeros ["Supplie#r", S_SUPPKEY].

    Es genera tal i com s'indica al punt 5 de l'apartat anterior.

  - S_ADDRESS **random v-string[10,40]**.

    Es genera tot seguint l'algorisme *7.1-3 Generació de strings de longitud variable* de l'apartat anterior.

  - S_NATIONKEY **random value [0 .. 24]**.

    Es genera tot seguint l'algorisme *7.1-1 Generació de nombres aleatoris* de l'apartat anterior.

  - S_PHONE generated according to Clause 4.2.2.9[30].

[30]La clàsula 4.2.2.9 correspon al punt 8 de l'apartat *7.1.4.1 Terminologia*

> Es genera tot seguint l'algorisme *7.1-5 Generació de números de telèfon* de l'apartat anterior.

– S_ACCTBAL **random value [-999.99 .. 9,999.99]**.

> Es genera tot seguint l'algorisme *7.1-1 Generació de nombres aleatoris* de l'apartat anterior amb optimització per a nombres reals.

– S_COMMENT **text string [25,100]**.

SF * 5 rows are randomly selected to hold at a random position a string matching "Customer%Complaints". Another SF * 5 rows are randomly selected to hold at a random position a string matching "Customer%Recommends", where % is a wildcard that denotes zero or more characters.

> Es genera tot seguint l'algorisme *7.1-6 Generació de comentaris* de l'apartat anterior. La restricció de triar $SF * 5$ tuples per tal de fer que aquest atribut contingui certs substrings s'ha ignorat ja que no és rellevant per a aquest generador. Això és degut a que les consultes que posin a prova la construcció dels cubs de dades no contindran operacions les quals el seu resultat pugui variar segons aquesta restricció.

- SF * 200,000 rows in the PART table with:
  – P_PARTKEY **unique within [SF * 200,000]**.

> Es genera tal i com s'indica al punt 2 de l'apartat anterior.

– P_NAME generated by concatenating five unique randomly selected strings from the following list, separated by a single space:

"almond"	"antique"	"aquamarine"	"azure"	"beige"	"bisque"
"black"	"blanched"	"blue"	"blush"	"brown"	"burlywood"
"burnished"	"chartreuse"	"chiffon"	"chocolate"	"coral"	"cornflower"
"cornsilk"	"cream"	"cyan"	"dark"	"deep"	"dim"
"dodger"	"drab"	"firebrick"	"floral"	"forest"	"frosted"
"gainsboro"	"ghost"	"goldenrod"	"green"	"grey"	"honeydew"
"hot"	"indian"	"ivory"	"khaki"	"lace"	"lavender"
"lawn"	"lemon"	"light"	"lime"	"linen"	"magenta"
"maroon"	"medium"	"metallic"	"midnight"	"mint"	"misty"
"moccasin"	"navajo"	"navy"	"olive"	"orange"	"orchid"
"pale"	"papaya"	"peach"	"peru"	"pink"	"plum"
"powder"	"puff"	"purple"	"red"	"rose"	"rosy"
"royal"	"saddle"	"salmon"	"sandy"	"seashell"	"sienna"
"sky"	"slate"	"smoke"	"snow"	"spring"	"steel"
"tan"	"thistle"	"tomato"	"turquoise"	"violet"	"wheat"
"white"	"yellow"				

> De forma sembla a altres dades, aquest tipus es genera a partir de la generació de cinc nombres aleatoris que representen les cinc posicions del llistat de paraules que s'accediran. Finalment, es realitza una concatenació d'aquests strings obtinguts:

Algorisme 7.1-7: Generació de P_NAME

```
array pName [0 .. N − 1] := {word1, word2 .. wordN}
function generateP_NAME(var id)
begin
  var word1 := generateInteger(id, 0, N − 1)
  var word2 := generateInteger(id + 1, 0, N − 1)
  var word3 := generateInteger(id + 2, 0, N − 1)
  var word4 := generateInteger(id + 3, 0, N − 1)
  var word5 := generateInteger(id + 4, 0, N − 1)
  var name := concat(
    word1, " ", word2, " ",
    word3, " ", word4, " ",
    word5
  )
  return name
end
```

A diferència d'altres algorismes, en aquest és indiferent si la generació d'aquestes dades és uniforme o no, ja que la seva única funció és afegir volum de dades i amb aquest algorisme ja es compleix aquesta funció.

– P_MFGR **text appended with digit [“Manufacturer#”,M]**, where M = random value [1,5].

Es genera tal i com s'indica al punt 5 de l'apartat anterior.

– P_BRAND **text appended with digits [“Brand#”,MN]**, where N = random value [1,5] and M is defined while generating P_MFGR.

No es presenta cap algorisme addicional per a la generació d'aquestes dades ja que la seva construcció és molt semblant a la presentada en el punt 5 de l'apartat anterior. Bàsicament consisteix en generar un nou nombre aleatori complint les condicions de l'enunciat i concatenar-lo amb la *M* generada en el càlcul de P_MFGR. Cal mencionar que la nomenclatura *[“Brand#”,MN]* de l'enunciat es refereix a la concatenació de *MN* i no al seu producte, com podria semblar inicialment.

– P_TYPE **random string [Types]**.

Es genera tot seguint l'algorisme *7.1-2 Generació de strings aleatoris a partir de concatenació* de l'apartat anterior.

– P_SIZE **random value [1 .. 50]**.

Es genera tot seguint l'algorisme *7.1-1 Generació de nombres aleatoris* de l'apartat anterior.

– P_CONTAINER **random string [Containers]**.

Es genera tot seguint l'algorisme *7.1-2 Generació de strings aleatoris a partir de concatenació* de l'apartat anterior.

- P_RETAILPRICE = (90000 + ((P_PARTKEY/10) modulo 20001) + 100 * (P_PARTKEY modulo 1000))/100

  El TPC-H ja dóna la formula per tal de calcular aquest atribut. Per tant, només cal aplicar-la.

- P_COMMENT **text string [5,22]**.

  Es genera tot seguint l'algorisme *7.1-6 Generació de comentaris* de l'apartat anterior.

- For each row in the PART table, four rows in PartSupp table with:
  - PS_PARTKEY = P_PARTKEY.

    Es tracta d'una simple assignació d'un atribut ja generat.

  - PS_SUPPKEY = (ps_partkey + (i * ((S/4) + (int)(ps_partkey-1)/S)))) modulo S + 1 where i is the ith supplier within [0 .. 3] and S = SF * 10,000.

    El TPC-H ja dóna la formula per tal de calcular aquest atribut. Per tant, només cal aplicar-la.

  - PS_AVAILQTY **random value [1 .. 9,999]**.

    Es genera tot seguint l'algorisme *7.1-1 Generació de nombres aleatoris* de l'apartat anterior.

  - PS_SUPPLYCOST **random value [1.00 .. 1,000.00]**.

    Es genera tot seguint l'algorisme *7.1-1 Generació de nombres aleatoris* de l'apartat anterior amb optimització per a nombres reals.

  - PS_COMMENT **text string [49,198]**.

    Es genera tot seguint l'algorisme *7.1-6 Generació de comentaris* de l'apartat anterior.

- SF * 150,000 rows in CUSTOMER table with:
  - C_CUSTKEY **unique within [SF * 150,000]**.

    Es genera tal i com s'indica al punt 2 de l'apartat anterior.

  - C_NAME **text appended with minimum 9 digits** with leading zeros["Customer#", C_CUSTKEY].

    Es genera tal i com s'indica al punt 5 de l'apartat anterior.

  - C_ADDRESS **random v-string [10,40]**.

    Es genera tot seguint l'algorisme *7.1-3 Generació de strings de longitud variable* de l'apartat anterior.

– C_NATIONKEY **random value [0 .. 24]**.

> Es genera tot seguint l'algorisme *7.1-1 Generació de nombres aleatoris* de l'apartat anterior.

– C_PHONE generated according to Clause 4.2.2.9.

> Es genera tot seguint l'algorisme *7.1-5 Generació de números de telèfon* de l'apartat anterior.

– C_ACCTBAL **random value [-999.99 .. 9,999.99]**.

> Es genera tot seguint l'algorisme *7.1-1 Generació de nombres aleatoris* de l'apartat anterior amb optimització per a nombres reals.

– C_MKTSEGMENT **random string [Segments]**.

> Es genera tal i com s'indica al punt 5 de l'apartat anterior.

– C_COMMENT **text string [29,116]**.

> Es genera tot seguint l'algorisme *7.1-6 Generació de comentaris* de l'apartat anterior.

- For each row in the CUSTOMER table, ten rows in the ORDERS table with:

– O_ORDERKEY **unique within [SF * 1,500,000 * 4]**.

Comment: The ORDERS and LINEITEM tables are sparsely populated by generating a key value that causes the first 8 keys of each 32 to be populated, yielding a 25% use of the key range. Test sponsors must not take advantage of this aspect of the benchmark. For example, horizontally partitioning the test database onto different devices in order to place unused areas onto separate peripherals is prohibited.

> Es genera tal i com s'indica al punt 2 de l'apartat anterior. L'aclaració que apareix sota l'enunciat s'ha obviat en aquest generador ja que, segons l'enunciat, només s'han de poblar les primeres 8 de cada 32 orders generades. Degut a això, el rang de valors que pot prendre *O_ORDERKEY* és l'interval $[1..SF * 1,500,000 * 4]$. En el cas d'aquest generador, això es pot simplificar ja que només es tracta de la primary key de cada order. Així doncs, en aquest generador les orders es generen amb *O_ORDERKEY* a l'interval $[1..SF * 1,500,000]$.

– O_CUSTKEY = **random value [1 .. (SF * 150,000)]**.

The generation of this random value must be such that O_CUSTKEY modulo 3 is not zero.
Comment: Orders are not present for all customers. Every third customer (in C_CUSTKEY order) is not assigned any order.

> Es genera tot seguint l'algorisme *7.1-1 Generació de nombres aleatoris* de l'apartat anterior.

– O_ORDERSTATUS set to the following value:

"F" if all lineitems of this order have L_LINESTATUS set to "F".
"O" if all lineitems of this order have L_LINESTATUS set to "O".
"P" otherwise.

La generació d'aquest tipus d'atribut és molt senzilla. Bàsicament es tracta d'un algorisme que, donat un vector amb tots els valors de *L_LINESTATUS*, el recorre en busca de diferències. Si n'hi ha, retorna "P" tal i com indica l'enunciat, si no retorna el valor de qualsevol posició del vector. L'algorisme és el següent:

Algorisme 7.1-8: Generació de O_ORDERSTATUS

```
var generateOrderstatus(array linestatus[0 .. N − 1])
begin
  for i from 1 to i < N
  do
    if linestatus[i] != linestatus[i − 1]
    then
      return ''P''
    fi
  done
  return linestatus[0]
end
```

– O_TOTALPRICE computed as:

sum (L_EXTENDEDPRICE * (1+L_TAX) * (1-L_DISCOUNT)) for all LINEITEM of this order.

El TPC-H ja dóna la formula per tal de calcular aquest atribut. Per tant, només cal aplicar-la.

– O_ORDERDATE uniformly distributed between STARTDATE and (ENDDATE - 151 days).

Per tal de facilitar la generació d'aquest atribut aquest es troba entre *STARTDATE* i *ENDDATE*, en comptes de *ENDDATE - 151 days*. Això és així perquè d'aquesta manera es poden aprofitar els algorismes de l'apartat anterior i s'eviten complicacions al fer càlculs amb les dates. Això és possible degut a que aquest atribut no es contempla en les consultes que posaran a prova la construcció dels cubs de dades. Així doncs, aquest atribut es genera tot seguint l'algorisme *7.1-4 Generació de dates* de l'apartat anterior.

– O_ORDERPRIORITY **random string [Priorities]**.

Es genera tot seguint l'algorisme *7.1-2 Generació de strings aleatoris a partir de concatenació* de l'apartat anterior.

– O_CLERK **text appended with minimum 9 digits** with leading zeros["Clerk#", C] where C = random value [000000001 .. (SF * 1000)].

Es genera tal i com s'indica al punt 5 de l'apartat anterior.

– O_SHIPPRIORITY set to 0.

S'incialitza a 0

– O_COMMENT **text string [19,78]**.

> Es genera tot seguint l'algorisme *7.1-6 Generació de comentaris* de l'apartat anterior.

- For each row in the ORDERS table, a random number of rows within [1..7] in the LINEITEM table with:
  - L_ORDERKEY = O_ORDERKEY.

    > Es tracta d'una simple assignació d'un atribut ja generat.

  - L_PARTKEY **random value [1 .. (SF * 200,000)]**.

    > Es genera tot seguint l'algorisme *7.1-1 Generació de nombres aleatoris* de l'apartat anterior.

  - L_SUPPKEY = (L_PARTKEY + (i * ((S/4) + (int)(L_partkey-1)/S)))) modulo S + 1 where i is the corresponding supplier within [0 .. 3] and S = SF * 10,000.

    > El TPC-H ja dóna la formula per tal de calcular aquest atribut. Per tant, només cal aplicar-la.

  - L_LINENUMBER **unique within [7]**.

    > Es genera tal i com s'indica al punt 2 de l'apartat anterior.

  - L_QUANTITY **random value [1 .. 50]**.

    > Es genera tot seguint l'algorisme *7.1-1 Generació de nombres aleatoris* de l'apartat anterior.

  - L_EXTENDEDPRICE = L_QUANTITY * P_RETAILPRICE Where P_RETAILPRICE is from the part with P_PARTKEY = L_PARTKEY.

    > El TPC-H ja dóna la formula per tal de calcular aquest atribut. Per tant, només cal aplicar-la.

  - L_DISCOUNT **random value [0.00 .. 0.10]**.

    > Es genera tot seguint l'algorisme *7.1-1 Generació de nombres aleatoris*, produïnt un valor entre 0 i 10 (inclosos) i dividint-lo entre 100. Finalment, com que el número a retornar ha de contenir obligatoriament dos dígits decimals i ha de ser un string, per evitar la pèrdua d'aquest dos dígits al fer la conversió a string en cas que el nombre generat sigui 0.00 (que es convertiria en 0) o 0.10 (que es convertiria en 0.1, s'aplica el següent algorisme:

Algorisme 7.1-9: Generació de L_DISCOUNT

```
function generateDiscount(var id)
begin
  discount := generateInteger(id, 0, 10) / 100
  if discount = 0
  then
    return ''0.00"
  elif discount = 0.10
    return ''0.10"
  fi
  return discount
end
```

– L_TAX **random value [0.00 .. 0.08]**.

Es genera tot seguint l'algorisme *7.1-1 Generació de nombres aleatoris*, produïnt un valor entre 0 i 8 (inclosos) i dividint-lo entre 100. Finalment, com que el número a retornar ha de contenir obligatoriament dos dígits decimals i ha de ser un string, per evitar la pèrdua d'aquest dos dígits al fer la conversió a string en cas que el nombre generat sigui 0.00 (que es convertiria en 0), també s'aplica el primer if de l'algorisme anterior *7.1-9 Generació de L_DISCOUNT*.

– L_RETURNFLAG set to a value selected as follows:
If L_RECEIPTDATE ≤ CURRENTDATE
then either "R" or "A" is selected at random
else "N" is selected.

Aquest atribut es una simple comparació de dates. Per tal de comparar les correctament, l'algorisme primerament compara, en aquest ordre, els anys, els mesos, i finalment els dies. Aquest algorisme és el següent:

Algorisme 7.1-10: Generació de L_RETURNFLAG

```
// date[0] -> year
// date[1] -> month
// date[2] -> day
function generateReturnFlag(array date[0 .. 2])
begin
  var year := date[0]
  var month := date[1]
  var day := date[2]
  var rand := (year + month + day) % 2
  boolean isLE := false
  if year < 1995
  then
    isLE := true
  elif year = 1995
  then
    if month < 6
      isLE = true
    elif month = 6
    then
      if day <= 17
      then
        isLE = true;
      fi
    fi
  fi
  if isLE
  then
    if rand = 0
      return ''A";
    else
      return ''R";
  fi
  return ''N";
end
```

La aleatorietat al retornar "R" o "A" en aquest cas no s'ha fet a partir de la funció presentada anteriorment a l'algorisme *7.1-1 Generació de nombres aleatoris*, sinó que ha estat tot calculant el mòdul d'una funció qualsevol dependent de la data proporcionada a la entrada. En aquest cas, ha estat la funció suma de l'any, mes i dia. Això és així perquè tal i com Java genera els nombres aleatoris, va sorgir que amb un interval tan petit sempre es genera 1 i per tant mai es retornava el valor "A".

Aleshores, aquesta generació que fa Java és uniforme si es genera tot un llistat de nombres aleatoris a partir d'una mateixa llavor. Però en el cas d'aquest generador, la llavor es modifica per a cada objecte generat (motius explicats en apartats anteriors) i aleshores la seqüència de nombres generats sempre comença per 1. Per tal de generar una seqüència que comenci per 0 cal fer-ho a partir de llavors molt més grans.

– L_LINESTATUS set the following value:

  "O" if L_SHIPDATE ¿ CURRENTDATE

"F" otherwise.

De la mateixa que l'atribut anterior, aquest també es genera tot fent una simple comparació de dates. L'algorisme és el següent:

Algorisme 7.1-11: Generació de L_LINESTATUS

```
// date[0] -> year
// date[1] -> month
// date[2] -> day
function generateLineStatus(array date[0 .. 2])
begin
  var year := date[2]
  var month := date[1]
  var day := date[0]
  if year > 1995
  then
    return ''O''
  elif year = 1995
  then
    if month > 6
    then
      return ''O''
    elif year = 6
    then
      if day > 17
      then
        return ''O'';
      fi
    fi
  fi
  return ''F''
end
```

– L_SHIPDATE = O_ORDERDATE + **random value [1 .. 121]**.

Aquest atribut també es genera a partir d'un nou algorisme que, donada una data en format de vector de strings, li suma una quantitat de dies. L'algorisme és el següent:

Algorisme 7.1-12: Generació de suma de dates

```
// date[0] -> year
// date[1] -> month
// date[2] -> day
array daysOfMonth[0 .. 11] := {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
function generateLineStatus(array date[0 .. 2], var days)
begin
  var year := date[2]
  var month := date[1]
  var day := date[0]
  var days_tmp := day + days
  var i := month - 1
  var j := 0
  while days_tmp >= daysOfMonth[i % 12]
  do
    days_tmp := days_tmp - daysOfMonth[i % 12]
    i++
    if i % 12 = 0
    then
      j++
    fi
  done
  var finalYear := year + j;
  var finalMonth
  if (i % 12) + 1 >= 10
  then
    finalMonth = (i % 12) + 1
  else
    finalMonth = concat(''0'', (i % 12) + 1)
  fi
  if days_tmp + 1 >= 10
  then
    finalDay := days_tmp + 1
  else
    finalDay := concat(''0'', days_tmp + 1)
  fi
  return {finalYear, finalMonth, finalDay}
end
```

Aquest algorisme és molt semblant a l'algorisme *7.1-4 Generació de dates*. Bàsicament afegeix el nombre de dies a sumar al dia de la data que hi arriba com a paràmetre d'entrada. A partir d'aquí, recorre tots els mesos de l'any restant, al nombre de dies resultant de la suma anterior, el número de dies de cada mes. Aquest bucle finalitza quan el total de dies sigui inferior al nombre de dies del següent mes. D'aquesta manera, s'obté que a la data final:

* El dia final és el nombre restant de dies resultant del bucle.
* El mes correspon a la suma del mes inicial i el número total d'iteracions del bucle, en mòdul 12.
* Finalment, l'any correspon a la suma de l'any inicial i el número total d'iteracions del bucle on el mes de la iteració correspon al canvi de

desembre a gener.

Així doncs, aquest són els pasos per tal d'afegir un número de dies a un data qualsevol. La resta de l'algorisme correspon a donar el format adequat a la data resultant i les explicacions de l'algorisme *7.1-4 Generació de dates* són completament extrapolables a aquest cas.

- L_COMMITDATE = O_ORDERDATE + **random value [30 .. 90]**.

  Es genera tot seguint l'algorisme *7.1-12 Generació de suma de dates* presentat anteriorment.

- L_RECEIPTDATE = L_SHIPDATE + **random value [1 .. 30]**.

  Es genera tot seguint l'algorisme *7.1-12 Generació de suma de dates* presentat anteriorment.

- L_SHIPINSTRUCT **random string [Instructions]**.

  Es genera tot seguint l'algorisme *7.1-2 Generació de strings aleatoris a partir de concatenació* de l'apartat anterior.

- L_SHIPMODE **random string [Modes]**.

  Es genera tot seguint l'algorisme *7.1-2 Generació de strings aleatoris a partir de concatenació* de l'apartat anterior.

- L_COMMENT **text string [10,43]**.

  Es genera tot seguint l'algorisme *7.1-6 Generació de comentaris* de l'apartat anterior.

- 25 rows in the NATION table with:
  - N_NATIONKEY **unique value** between 0 and 24.
  - N_NAME **string** from the following series of (N_NATIONKEY, N_NAME, N_REGIONKEY).

    (0, ALGERIA, 0); (1, ARGENTINA, 1); (2, BRAZIL, 1);
    (3, CANADA, 1); (4, EGYPT, 4); (5, ETHIOPIA, 0);
    (6, FRANCE, 3); (7, GERMANY, 3); (8, INDIA, 2);
    (9, INDONESIA, 2); (10, IRAN, 4); (11, IRAQ, 4);
    (12, JAPAN, 2); (13, JORDAN, 4); (14, KENYA, 0);
    (15, MOROCCO, 0); (16, MOZAMBIQUE, 0); (17, PERU, 1);
    (18, CHINA, 2); (19, ROMANIA, 3); (20, SAUDI ARABIA, 4);
    (21, VIETNAM, 2); (22, RUSSIA, 3); (23, UNITED KINGDOM, 3);
    (24, UNITED STATES, 1)

  - N_REGIONKEY is taken from the series above.
  - N_COMMENT **text string [31,114]**.
- 5 rows in the REGION table with:
  - R_REGIONKEY **unique valu** between 0 and 4.
  - R_NAME **string** from the following series of (R_REGIONKEY, R_NAME).

    (0, AFRICA); (1, AMERICA); (2, ASIA);
    (3, EUROPE); (4, MIDDLE EAST)

  - R_COMMENT **text string [31,115]**.

Degut a la cardinalitat fixa de les taules nation i region (25 i 5 tuples respectivament). La generació dels seus atributs s'ha simplificat tot extraient-los d'una execució del TPC-H real. En altres paraules, a partir d'una execució del TPC-H oficial, s'han agafat les taules generades nation i region i s'han introduït en un vector constant de 25 posicions on cada posició conté les dades de la respectiva nation i de la region a la que correspon. D'aquest manera, per obtenir aquestes dades tan sols cal fer un accés en aquest vector. Tot això implica que totes les execucions d'aquest generador produiran els mateixos valors per a les taules nation i region, però aquest fet és indiferent. De fet, és possiblement inclús beneficiós ja que ajuda a que les dades generades en diverses execucions siguin més semblants i per tant les proves finalment es llencin sobre un mateix joc de proves, i aleshores els resultats obtinguts poden ser més comparables.

Amb la resta de les taules no es pot aplicar aquest simplificació degut a que tenen un volum massa gran per emmagatzemar-los a memòria volàtil.

7.1.4.3 Generació de les taules

A l'apartat anterior es presenten els tipus que conformen els diferents atributs del TPC-H. Ara bé, com es menciona a l'apartat *7.1.1 L'esquema de dades. TPC-H*, la generació del TPC-H en aquest projecte ha de realitzar-se en una versió desnormalitzada. Això significa que per tal de dur a terme correctament la cardinalitat de les taules presentada als apartats anteriors, l'algorisme final no pot generar les taules de forma independent ja que una tupla d'una taula qualsevol ha d'apareixer a tantes tuples desnormalitzades com cardinalitat tenen les seves relacions. En altres paraules, si $A_1 \rightarrow_* B$ indica una relació *one-to-many*, per inserir una relació d'aquest tipus en la versió desnormalitzada caldria inserir A tants cops cop cardinalitat tingués la relació.

En aquest apartat es presenta l'algorisme que dona la estructura final al TPC-H desnormalitzat tot respectant les cardinalitats adequades. Per tal de fer aquesta presentació, les dues següents definicions corresponen a dos tipus de variables que facilitaran la comprensió de l'algorisme:

a) Variables amb el nom tipus *workloadX* que representen el nombre de tuples de la taula X que ha de generar cada thread. Així, per exemple, *workloadOrders* és nombre de tuples de la taula orders que cal que generi cada thread.

b) Variables tipus *capX* que representen el nombre total de tuples de la taula X que ha de generar cada thread. Per exemple: *capOrders*. El valor d'aquestes variables és la cardinalitat de la taula que presenta el corresponent enunciat.

Per simplificar s'assumeix que tots els threads generen el mateix nombre de tuples.

Així doncs, l'algorisme comença inicialment generant la taula orders. El motiu de començar per a aquesta taula és que és la única a partir de la qual, a qualsevol extrem de les associacions, hi ha una cardinalitat coneguda. Això és molt important perquè si la relació fos de tipus *many-to-many* les dificultats per generar el nombre de tuples de l'enunciat augmentarien, i possiblement s'hagués d'afegir comunicació entre els threads per tal de saber el nombre de tuples total. D'aquesta manera, per saber el nombre total de tuples generades només cal consultar les que ha generat un thread i multiplicar-les pel nombre de threads total. Així doncs, a partir de les orders es pot navegar cap a customer, on la cardinalitat és 1 (una order només la fa un customer), o bé també es pot navegar cap a lineitem, on la cardinalitat és 1..7.

Algorisme 7.1-13: Generació del TPC-H

```
action generateTPC−H(var thread_id) // Thread identifier
begin
  // Computation of workload and cap variables
  var ordersGenerated := 0
  var customersGenerated := 0
  while ordersGenerated < workloadOrders
  do
    if customersGenerated < workloadCustomer
    then
      var C_CUSTKEY := ((workloadCustomer ∗ id + customersGenerated) % capCustomer) + 1
    else
      var C_CUSTKEY := generateInteger(workloadCustomer ∗ id + customersGenerated,
            1, capCustomer)
    fi
    // Generate customer
    if C_CUSTKEY % 3 = 0
    then
      var O_ORDERKEY := workloadOrders ∗ thread_id + ordersGenerated + 1
      // Generate order
      if ordersGenerated % 2 = 0
      then
        var nLineitems := random() % 7 + 1
      else
        var nLineitems := nLineitems = 8 − nLineitems;
      fi
      var l := 0
      while l < nLineitems
      do
        if ordersGenerated + o < workloadParts
        then
          P_PARTKEY := ((workloadParts ∗ id + ordersGenerated + o) % capParts) + 1
        else
          P_PARTKEY := generateInteger(workloadParts ∗ id + ordersGenerated + o,
              1, capParts)
        fi
        // Generate part
        PS_PARTKEY := P_PARTKEY
        PS_SUPPKEY := // Veure apartat anterior
        // Generate partsupp
        S_SUPPKEY := PS_SUPPKEY
        // Generate supplier
        // Generate lineitem
        // Insert order + customer + lineitem + part + partsupp + supplier
        l := l + 1
      done
      ordersGenerated := ordersGenerated + 1
    elif customersGenerated < workloadCustomer
    then
      // Insert customer
    fi
    customersGenerated := customersGenerated + 1
  done
end
```

No obstant això, encara que a l'algorisme es vegi com els primers atributs que es generen són els corresponents al customer, cal notar que el bucle principal itera sobre les orders fins que s'hagin generat tantes com siguin necessàries. Fent-ho d'aquesta manera, es permet tenir en compte la clàusula (apartat anterior) que restringeix que un de cada tres customers no fa cap order. D'aquesta manera, si el customer generat compleix mòdul 3 es realitza en una inserció a part, en una tupla on només s'hi conté aquesta informació i no apareix cap atributs de la resta de les taules. En cas contrari, es produeix la generació de la resta d'atributs.

De la mateixa manera, aquests customers que no realitzen cap order només poden apareixer un sol cop en tota la taula desnormalitzada, degut a que no existeix cap associació cap a la resta de taules (excepte les taules nation i region, però aquestes taules en aquest generador es tracten com si fossin atributs de les taules customer i supplier), i per tant només han de ser inserits quan es compleixi la condició $customersGenerated \leq workloadCustomer$.

No obstant això, també cal tenir en compte que ha d'assegurar-se que es generen els $workloadCustomers$ corresponents. En altres paraules, que s'ha d'evitar que la generació aleatòria de customers pugui fer que un mateix customer es repeteixi en masses tuples i per tant un customer diferent que també calia generar no s'acabi generant. Per bregar amb això la solució duta a terme correspon a generar els primers $workloadCustomers$ consecutivament, i a partir d'aquí, generar-los aleatòriament. D'aquesta manera queda assegurat que cada thread genera tots els customers que li corresponen.

L'associació entre orders i lineitems és de cardinalitat [1..7], amb mitjana 4. Per aquest motiu, la generació d'aquesta cardinalitat és fa tot seguint les següents condicions:

- Si el nombre de orders generades fins al moment és parell, aleshores s'escolleix un nombre aleatori entre [1..7]
- Si pel contrari, el nombre de orders generades és senar, aleshores és computa la cardinalitat tal que la mitjana amb la order anterior sigui 4:

$$\#lineitems_{actual} = 8 - \#lineitems_{anterior}$$

D'aquesta manera, queda assegurat que la mitjana total és 4, ja que el que s'està produint aquí realment és que el problema de la mitjana s'està trencant en porcions més petites tot fent que la mitjana entre un parell de orders sigui 4, i per tant la mitjana global sigui també 4.

Finalment, les cardinalitats de lineitems cap a partsupp es generen de forma semblant als customers. Cal evitar que la aleatorietat amb la que es generen aquestes taules no impedeixi que quedi algun objecte sense generar. Per assegurar-se que això no succeeix es segueix el mateix mètode que amb la taula customer, generant primer els objectes de forma consecutiva i a continuació de forma aleatòria.

7.1.5 Requisit II: Localitat de les dades

Un dels requisits d'aquest generador és que la forma d'inserir les tuples a l'HBase trenqui la localitat de les dades. D'aquesta manera, assegurem que a l'hora de construir els cubs de dades realment estem fent *random access* i no accessos seqüencials. Tot això ve donat degut a que si no trenquem la localitat de les dades, llavors les proves de rendiment dels cubs de dades podrien oferir resultats no realistes ja que els accessos a disc es farien de forma seqüencial i per tant ens estaríem estalviant costs de disc (p.ex: la latència de moure el capçal) que no podem ometre.

Per entendre com ha anat evolucionant aquesta part del generador, cal recordar l'estructura *clustered index* comentada a l'apartat *4.2.2 HBase*.

Partint de l'entorn multithreading en el que treballa aquest generador, es pot partir de la suposició que les tuples que genera un mateix thread són tuples amb valors similars i per tant són aquestes les que cal intercal·lar amb les tuples dels altres threads.

Primera solució

La primera solució que vaig fer buscava trencar aquesta localitat a partir d'intercal·lar l'ordre en que els threads inserien les tuples. A més, si cada thread havia d'inserir n tuples, el $thread_1$ inseria les tuples amb key $1..n$, el $thread_2$ les tuples amb key $n+1...2n$, i així successivament. Òbviament, en aquest moment no es tenia consciència del clustered index de l'HBase i per tant la suposició inicial era que les tuples s'emmagatzemaven físicament a disc segons l'ordre d'arribada. Per tant, si es volia trencar la localitat, només calia intercal·lar els threads. Això em conduir a ampliar el diagrama de classes de forma que s'incorporés una nova classe compartida per tots els threads i que s'encarregués de gestionar l'ordre de les insercions.

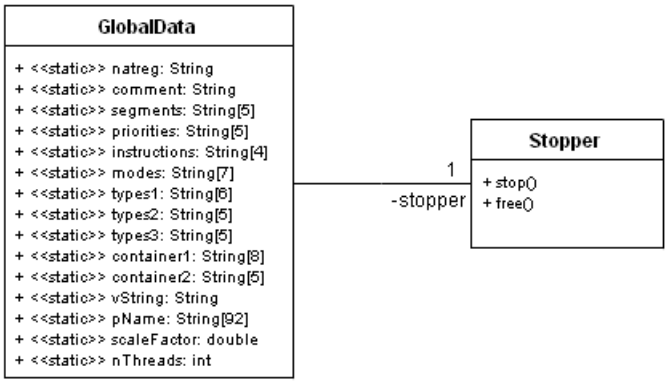


Figura 7.1-5: La classe *Stopper*

La forma d'intercal·lar els threads que vaig implementada en aquesta versió era aturant-los després de cada inserció i despertar-los un cop tots haguessin fet una inserció. La idea inicial era aturar fins a un màxim de threads (el 90% per exemple) per tal de tenir sempre uns quants threads corrent i que aquest màxim pogués ser configurable per l'usuari, però aturar menys threads implica més crides a funcions d'E/S[31] i per tant el rendiment queia lleugerament. Llavors, es va decidir implementar la solució que bloquejava cada thread després de cada inserció fins que tots n'han fet una. El pseudocodi que implementa aquestes funcions és:

[31] Entrada/Sortida

Algorisme 7.1-14: Funcions per al control de concurrència

```
void stop()
begin
  if currentWaits = nThreads − 1
  then
    currentWaits := 0
    notifyAll()
  else
    currentWaits++
    wait()
  fi
end

void free()
begin
  nThreads := nThreads − 1
  if currentWaits = nThreads
  then
    currentWaits := 0
    notifyAll()
  fi
end
```

Les variables currentWaits i nThreads indiquen el nombre de threads aturats i el nombre total de threads de l'execució, respectivament.

Les funcions wait() i notifyAll() corresponen a les funcions pròpies de Java que s'encarreguen d'aturar i de despertar, respectivament, tots aquells threads que estiguin aturats a la instància sobre la que es criden aquestes funcions.

La funció stop() és la encarregada d'aturar els threads. La seva implementació és fàcil de traduir a llenguatge natural: *si sóc l'últim thread que falta per aturar, no em bloquejo i desperto a la resta, en cas contrari em bloquejo.*

La funció free() soluciona la següent situació. Imaginem que tots els threads menys un estan aturats esperant a que aquest faci la seva inserció. En una situació normal, aquest últim thread cridaria a la funció stop() després de fer la seva inserció i despertaria a la resta de threads. Però podria donar-se la situació en que l'últim thread que resta per inserir ja ha fet totes les seves insercions i finalitza la seva execució. En aquest cas, tindríem que aquest últim thread no cridaria mai a la funció stop() i per tant no despertaria a la resta de threads, deixant a l'execució en un estat de deadlock. Així doncs, la funció free() intenta solventar aquesta situació tot essent cridada al final de l'execució de cada thread i pot entendre's de la següent manera: *resto 1 al nombre total de threads en execució i, si la resta de threads estan aturats, els desperto.*

Queda clar que aquesta solució va haver de ser descartada perquè realment no estava satisfent el requisit de trencar la localitat de les dades ja que, tot i fer les insercions de forma desordenada, el *clustered index* de l'HBase acabava ordenant les tuples físicament a disc.

Segona solució

En aquesta segona solució ja es va aconseguir que el trencament de la localitat es fes de forma correcta. En aquest cas, la solució proposada era la construcció d'una nova classe *TicketMaster*. L'objectiu d'aquesta classe era que cada thread, abans de fer una inserció,

li demanés la key amb la que inserir la tupla. Cal mencionar que aquesta segona solució no exclou la primera i per tant, s'ha de tenir en compte que es continua tenint la classe *Stopper*. Així doncs, el diagrama de classes es converteix en el que il·lustra la figura *7.1-6*.

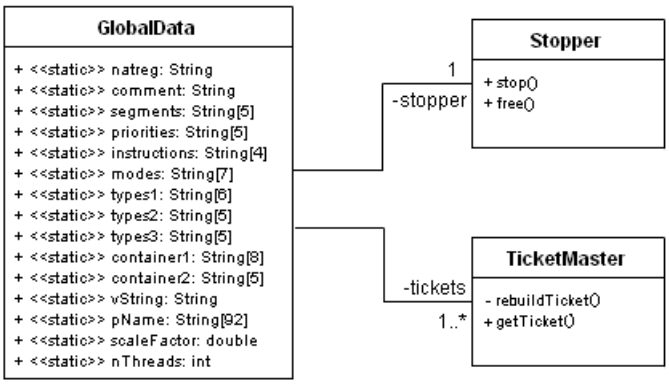


Figura 7.1-6: La classe *TicketMaster*

El problema d'aquesta versió és que si només s'instància un sol objecte *TicketMaster*, es crea un coll d'ampolla en aquesta mateixa classe ja que l'assignació de les keys s'ha de fer de forma exclusiva per evitar problemes de concurrència, i per tant es converteix aquesta instància en un sistema d'exclusió mútua que alenteix la velocitat d'inserció dels threads. La figura *7.1-7* ho mostra amb més detall.

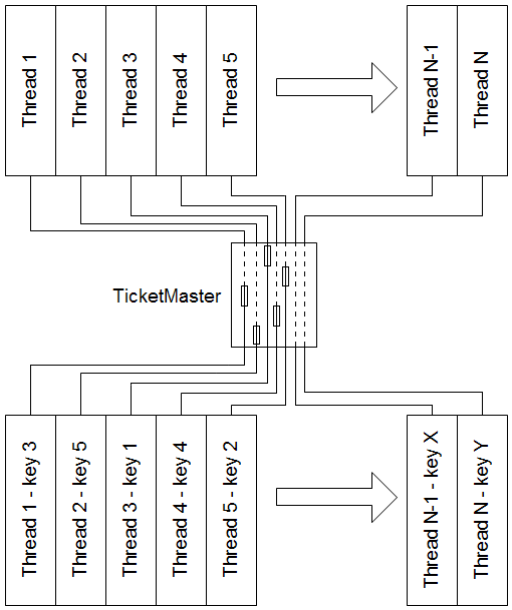


Figura 7.1-7: Coll d'ampolla del *TicketMaster*

Llavors, per evitar aquesta situació, la solució triada va ser la de crear múltiples instàncies de la classe *TicketMaster*. Cada una d'aquestes instàncies era l'encarregada de retornar les keys del conjunt:

$$[\frac{iN_{Rows}}{N_{TicketMaster}} + 1, \frac{(i+1)N_{Rows}}{N_{TicketMaster}} + 1)$$

on $i = 0, 1, 2..N_{TicketMaster} - 1$ depenent de la instància, i on cada N indica el nombre de files i d'instàncies de *TicketMaster*. El motiu de sumar 1 és simplement per construir les keys tal que $key = 1..n$, en comptes de $key = 0..n - 1$. D'aquesta manera, quant cada thread demana una nova key per assignar-li a la tupla a inserir, li demana a una instància aleatòria de *TicketMaster*. A més, pot ocorre que una instància esgoti totes les seves keys i per tant, cal controlar que en cas que un thread torni a demanar una nova key a aquesta instància, aquesta key no pertanyi al conjunt de keys d'una altra instància de *TicketMaster*. Per controlar això, cada cop que un thread demana una nova key a una instància de *TicketMaster*, aquesta instància primer controla que no els hagi ja assignat tots:

- Si no estan tots assignats, retorna la següent key amb normalitat.
- Si estan tots assignats, llavors aquella instància fa un *rebuild* de sí mateixa, que consisteix en generar un nou conjunt de keys de forma tal que:

$$[\frac{(i + N_{TicketMaster}j)N_{Rows}}{N_{TicketMaster}} + 1, \frac{(i + N_{TicketMaster}j + 1)N_{Rows}}{N_{TicketMaster}} + 1)$$

on $j = 1, 2, 3..$ indica el número de *rebuild* d'aquella instància. D'aquesta manera, una instància sempre assignarà keys que seran exclusives de la resta d'instàncies. Tot i així, aquesta solució no controla que totes les instàncies de *TicketMaster* acabin retornant totes les seves keys i per tant pot passar que les keys que finalment s'insereixen a l'HBase, a diferència de la primera solució, no siguin totalment consecutives, però aquest és un detall que no importa per a aquest projecte ja que en cap moment les keys beneficien o perjudiquen els resultats finals de construir els cubs de dades.

De totes maneres, és important notar que la distribució de les keys en vàries instàncies de *TicketMaster* no es desfà del coll d'ampolla que sorgeix amb aquesta solució, sinó que simplement l'alleugereix. Això és degut a que és poc probable que tots els threads demanin una key a la mateixa instància de *TicketMaster* i en el mateix moment, però és possible que només uns pocs sí que ho facin. Tot depèn del nombre $N_{TicketMaster}$. En el cas d'aquest projecte, durant les proves fetes per a aquesta versió, $N_{TicketMaster} = 1000$, i amb prou feines es notava aquest coll d'ampolla en el rendiment.

Per altra banda, quan es va plantejar aquesta solució, la idea inicial era guanyar en rendiment tot desfent-se de la classe *Stopper*, ja que amb aquesta solució les keys s'assignen de forma aleatòria i uniformement distribuïda[32] i no cal intercal·lar els threads. No obstant, al treure la classe *Stopper* d'aquesta solució, van començar a sorgir alguns problemes de timeout.

A l'apartat *4.2.2 Buffer d'escriptura (client-side): Autoflush* es parla amb més detall de l'autoflush, però bàsicament aquest és un mecanisme que proporciona l'HBase per controlar més l'ample de banda de la xarxa que es fa servir al fer les insercions. Amb l'autoflush activat, les tuples s'envien una a una contra l'HBase. En canvi, si està desactivat, el que es guanya és salvar les insercions en un buffer local al client i, un cop aquest està ple, s'envia per la xarxa. D'aquesta manera s'aprofita molt més l'ample de banda i el rendiment global del generador tendeix a augmentar. No obstant això, per tal de

[32]La generació dels nombres aleatoris que fa Java segueix una distribució uniforme.

facilitar-li al client esbrinar quines tuples van a cada RegionServer, aquest buffer local fa una ordenació prèvia (veure apartat *4.2.2 Buffer d'escriptura (client-side): Autoflush*).

Aleshores, la caiguda del rendiment i el conseqüent timeout venien produits per una barreja entre l'entorn multithreading en el que es treballa i aquesta ordenació. Si es té en compte que el buffer del client ocupa 2MB i suposem que llancem el generador amb 100 threads, tenim que en el pitjor dels casos s'està fent una ordenació de 200MB de tuples alhora, lo qual té un cost de CPU important. Per tant, al clúster virtual que vaig muntar a casa podia succeir que l'execució del generador finalitzés inesperadament amb l'excepció: *UnknownScannerException*, que es descriu a la documentació oficial com: *Thrown if a region server is passed an unknown scanner id. Usually means* ***the client has take too long between checkins*** *and so the scanner lease on the serverside has expired OR the serverside is closing down and has cancelled all leases.*

A més, en aquest mateix clúster el client corria sobre la mateixa màquina que el màster de l'HBase, però tota la CPU se l'enduia el generador degut a la ordenació explicada anteriorment, i llavors ocorria que el heartbeat entre el màster i els RegionServers moria per inanició, i a la llarga el clúster acabava fallant. Aquesta inanició també afectava a l'HDFS, però amb la salvedat que el temps des de l'últim heartbeat rebut a l'HDFS ha de ser molt més gran que no pas a l'HBase per considerar que un DataNode ha caigut, i per tant l'HDFS acostumava a mantenir-se en servei.

Aleshores, per tal d'afrontar aquests problemes de timeout, vaig haver de deixar la classe *Stopper*, amb la diferència que el seu objectiu ara no era el d'intercal·lar els threads, sinó més aviat afluixar-los per reduïr el nombre de buffers ordenant-se alhora i evitar així ofegar la CPU del client.

De totes maneres, aquest era un problema degut principalment a la limitació hardware del petit clúster de proves i segurament al clúster del DAC no s'haguessin reproduït aquests errors.

Una petita millora que es va implementar va ser la de fer que el conjunt de keys que retornaven els *TicketMasters* fossin només d'una única key, i cada cop que un thread demanés una nova, es fes el *rebuild*. Si recordem, el *rebuild* d'una instància de *TicketMaster* consisteix en reservar un nou conjunt únic de keys per retornar als threads. En el cas d'aquesta millora, on els *TicketMasters* només reserven una sola key, el *rebuild* segueix l'expressió:

$$key_{nova} = i + N_{TicketMaster} j + 1$$

on $i = 0, 1, 2..N_{TicketMaster} - 1$ i $j = 1, 2, 3..$ és el número de vegades que aquella instància ha fet *rebuild*. Així doncs, és fàcil veure que les keys es van assignant de forma relativament creixent. La figura *7.1-8* il·lustra aquest matís amb més detall. Per entendre aquesta figura, hem de suposar que només tenim tres úniques instàncies de la classe *TicketMaster*.

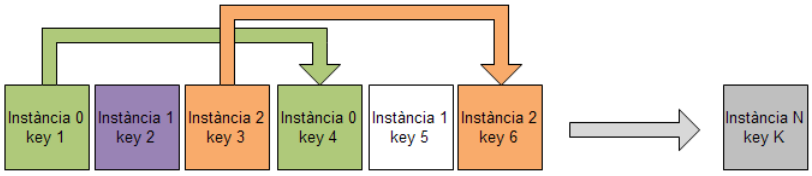


Figura 7.1-8: Evolució del *rebuild* de les instàncies de *TicketMaster*

Com es pot veure, si tenim en compte l'entorn multi-threading en el que treballa aquest generador, també es pot afirmar que en certa manera, quan un thread demana una

nova key, totes les instàncies de *TicketMaster* hauran estat consultades a l'inserir la tupla anterior per un o altre thread i per tant totes elles hauràn fet un *rebuild* de si mateixes. La conseqüència d'això és que la nova key retornada serà més gran que l'anterior. Òbviament, la pràctica no és exactament així. Per aquest motiu, a la figura *7.1-8* es representa la key 5 de color blanc. La key 5 és una key que pot ser assignada per només la instància de *TicketMaster* 1, però com que aquesta instància no ha estat consultada anteriorment, no ha fet *rebuild* de si mateixa. Aquest és el cas on la nova key retornada a un thread és menor que l'anterior.

Tot això implica que el buffer local del client s'omple de forma ja semiordenada i per tant es redueix el cost d'ordenació a l'hora de fer les insercions. La veritat és que aquesta idea va funcionar. La classe *Stopper* podia desaparèixer ja que el cost de la ordenació va disminuir i s'evitava l'excepció *UnknownScannerException* abans comentada. Tot i així, aquesta implementació no acabava de trencar la localitat com s'esperava. Què passava si:

a) un únic thread demanava un conjunt de keys a instàncies consecutives de *TicketMaster*?

a) se li assignava més temps de CPU a un thread que a un altre?

La resposta a la primera pregunta és que, en aquest cas, tuples semblants (les generades per un mateix thread) s'emmagatzemarien juntes a disc i per tant no s'estaria trencant la localitat correctament. La resposta a la segona pregunta és que, malgrat que sí estaríem trencant la localitat, seria un trencament molt lleuger ja que les tuples dels threads amb més prioritat de CPU tindrien keys menors i per tant estarien concentrades a l'inici de la taula i no tan disperses com podrien estar-ho amb altres solucions. Per aquests motius, es va descartar aquesta solució.

En definitiva:

- La classe *Stopper*.
- Múltiples instàncies de la classe *TicketMaster*.

pretenia ser la solució definitiva ja que trencava la localitat de la forma esperada i oferia un rendiment acceptable. No obstant, més tard va néixer la tercera versió que soluciona el requisit de la localitat de les dades sense fer ús de cap de les dues classes que aquesta solució sí necessita i que, per tant, ofereix un rendiment molt millor. Aquesta solució s'explica a l'apartat següent.

Tercera solució

Aquesta versió és la que va quedar com a definitiva. Trenca la localitat de les dades sense necessitat de cap classe addicional la qual cosa implica, respecte a la segona versió, que:

1. No cal fer cap control d'exclusió mútua per evitar problemes de concurrència.
2. No s'han de frenar els threads.

La veritat és que, a més, aquesta solució és la més senzilla de totes. Es basa simplement en la forma de construir les keys per a cada inserció. En les solucions anteriors, les keys es generaven a partir d'un seguit de nombres consecutius, els quals es transformaven en nombres de longitud fixa degut a la ordenació lexicogràfica. Ara, en aquesta solució, tots els threads generen de forma consecutiva les keys:

$$1..\frac{N_{Rows}}{N_{Threads}}$$

però li afegeixen el seu identificador al final de la key. Per exemple, la tupla 1 generada pel thread 1 tindria la key 000000001t1, la tupla 2 generada pel thread 1 tindria la key 0000000002t1, la tupla 15 generada pel thread 3 tindria la key 0000000015t3...

D'aquesta manera el que s'aconsegueix és que:

1. Les tuples que genera un mateix thread ja es generen de forma ordenada i per tant no cal fer cap ordenació prèvia del buffer local i ens podem estalviar la classe *Stopper*.

2. No cal la classe *TicketMaster* ja que les tuples d'un mateix thread ja queden intercal·lades amb les tuples de la resta de threads: la localitat queda de forma que primer queden totes les primeres tuples generades pels threads, després les segones tuples, etc.

La figura *7.1-9* il·lustra amb més deteniment aquests dos últims punts. En aquesta figura, suposem que cada thread insereix fins a un màxim de mil tuples (deixant de banda l'ordre lexicogràfic per simplificar) i que tenim N threads en execució.

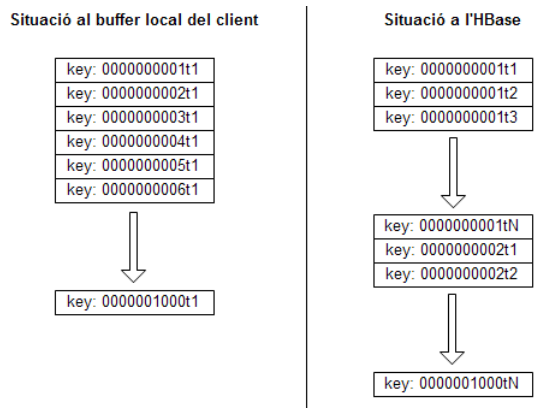


Figura 7.1-9: Situacions del buffer local i de l'HBase amb la tercera versió

Queden doncs, justificats els avantatges d'aquesta tercera versió respecte la resta i així el motiu pel qual es va decidir fer d'aquesta versió, la versió definitiva.

7.1.6 Requisit III: Optimització

Un altre requisit d'aquesta etapa del projecte era construir el generador de tal forma que fos òptim, és a dir, que el temps d'execució capigués dintre de la finestra de temps oferida pel clúster del DAC.

Inicialment, es pensava que el coll d'ampolla més gran seria, a part de les pròpies insercions, la generació de nombres aleatoris per tal de construir els atributs que són necessaris per poder generar el TPC-H degudament. Tampoc s'anava massa mal encaminats, però la realitat és que al final els nombres aleatoris alentien l'execució però no tant com inicialment suposàvem.

Els dos motius més importants de lentitud identificats van ser, per ordre d'importància:

1. Les insercions a l'HBase.

2. Els càlculs amb nombres reals (generalment necessaris ja que constitueixen les mesures al TPC-H).

En aquest apartat es presenten les solucions generades per tal de millorar el rendiment d'aquest generador per cada un dels colls d'ampolla identificats i altres optimitzacions de menys importància.

Optimització de les insercions

Les insercions suposen el coll d'ampolla més gran d'aquest generador. Per poder intentar optimitzar aquestes insercions, es van aprofitar les experiència del projecte previ i es va desactivar directament el WAL i l'autoflush (apartats *4.2.2 WAL: Write-Ahead Logging* i *4.2.2 Buffer d'escriptura (client-side): Autoflush*, respectivament).

En el cas d'aquest projecte, aquest WAL era totalment indiferent ja que no es buscava que les insercions puguessin recuperar-se si la màquina queia: sempre es podia tornar a llençar una nova execució del generador. Era molt més prioritari la generació del TPC-H en una finestra de temps acceptable pel clúster. Llavors, desactivant el WAL, el guany és una escriptura menys a disc per cada inserció.

L'altra millora quant a les insercions va ser la desactivació de l'autoflush. No obstant això, tal i com s'ha explicat amb més detall a l'apartat *4.2.2 Buffer d'escriptura (client-side): Autoflush*, desa.ctivar l'autoflush implica que a l'hora d'enviar les tuples a l'HBase es fa una reordenació prèvia per tal que el client sàpiga quines tuples van a cada RegionServer. Tal i com s'explica en el mateix apartat on es descriu l'autoflush, podria donar-se una situació on desactivar l'autoflush podria afegir més overhead a l'execució.

En aquest projecte, l'autoflush s'aprofita d'una forma relativament important. Això és degut al requisit de localitat de dades explicat a l'apartat *7.1.5 Requisit II: Localitat de les dades*. Si recordem la versió definitiva del sistema que fèiem servir per trencar la localitat, a les tuples se les hi assignaven les keys de forma consecutiva. Si a això, li afegim que:

a) l'HBase balanceja les RegionServers de forma automàtica al llarg del temps, i

b) els RegionServers contenen les tuples per intervals (és a dir, que si un RegionServer conté les tuples amb key tal que $0 \leq key < 1000$, una tupla amb key 2000, per exemple, no podrà anar aquell RegionServer. A més, aquests intervals són completament disjunts),

tenim que les tuples que s'insereixen en un instant determinat van generalment contra un únic RegionServer. La figura *7.1-10* ho mostra amb més detall. Per evitar complicacions amb l'ordre lexicogràfic, suposem que en aquest exemple hi han 10 threads en execució els quals, el seu id no és $id = 1..10$ sinó $id = 0..9$.

No obstant, la realitat s'allunya d'aquest exemple ja que la concurrència dels threads fa que mentre un thread pugui estar inserint la seva n-èsima tupla, un altre thread pugui estar encara inserint la primera, i per tant no totes les tuples emmagatzemades al mateix buffer vagin contra el mateix RegionServer. En altres paraules, pot ocorre que quan els threads que hagin tingut menys temps de CPU facin les seves insercions, l'split que haurà fet l'HBase dels RegionServers ja estarà distribuit al llarg de varis RegionServers i per tant les tuples que resten per inserir també hauran de ser distribuides al llarg dels RegionServers corresponents a la seva key.

En qualsevol cas, durant la major part de l'execució, aquest generador s'aprofita de la desactivació de l'autoflush i per tant el temps total d'execució és menor.

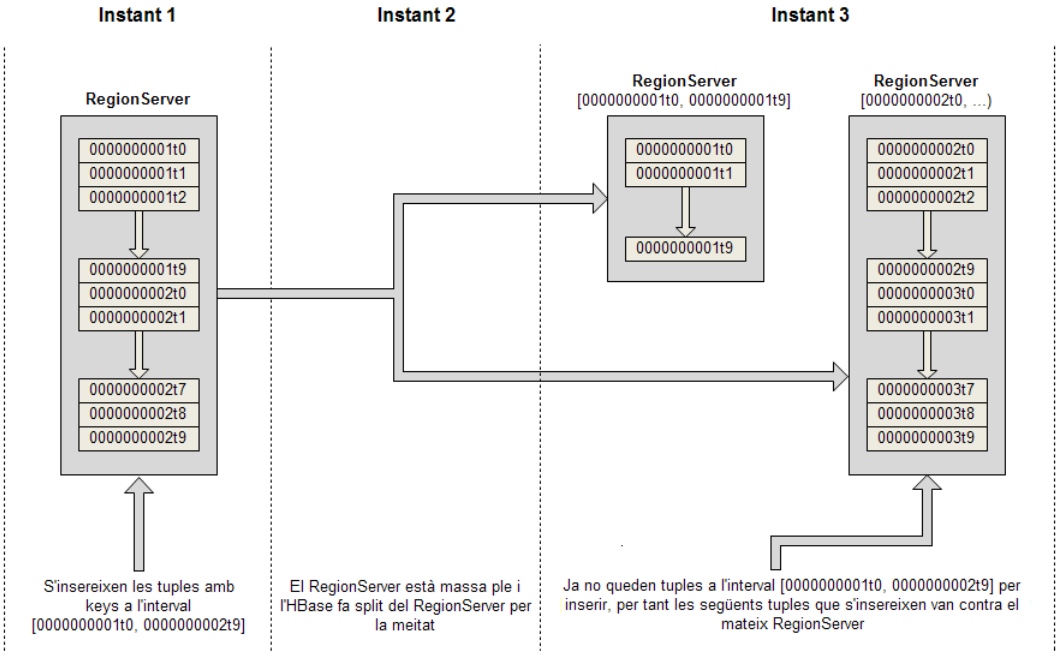


Figura 7.1-10: Evolució dels RegionServers durant les insercions

Malauradament, en el moment d'executar aquest generador sobre l'entorn del clúster del DAC, certes dificultats van fer que s'hagués de renunciar a aquesta optimització amb l'autoflush desactivat. Les explicacios corresponents a aquests canvis es troben a l'apartat *7.1.7.2 Concentració de dades.*

Optimització de les operacions amb nombres reals

Un altre coll d'ampolla important en aquest generador són les operacions amb nombres reals. Al llarg del TPC-H podem trobar varis exemples d'atributs que es generen a partir de l'aritmètica de nombres reals. Per a un processador, operar amb nombres enters és molt fàcil perquè la seva representació en binari permet fer els càlculs de la mateixa forma que amb la seva representació en decimal. En canvi, els nombres reals tenen una representació més complexa quan es tradueixen a binari i per tant no apliquen els càlculs de la mateixa forma que en decimal. Tot això significa que les operacions amb nombres reals es realitzen de forma diferent que amb nombres enters (de fet, cada processador ho implementa de la seva manera), però d'una forma o altra sempre afegeixen overhead a l'execució.

La veritat és que inicialment no se sabia d'aquest coll d'ampolla. Va aparèixer degut a que es van deixar alguns atributs de tipus real sense calcular mentre es feien proves de localitat i optimització (apartats *7.1.5 Requisit II: Localitat de les dades* i *7.1.6 Requisit III: Optimització*, respectivament). Llavors, quan es va afegir la generació d'aquests atributs, es va veure com el temps d'execució augmentava substancialment i es van començar a fer proves per tal de determinar què estava al·lentint l'execució. Finalment, es va descobrir que era degut a les operacions amb nombres reals.

Per poder desenvolupar una solució a aquest ralentiment, em vaig recolzar en la simplificació del TPC-H. Si recordem, a l'apartat *7.1.1 L'esquema de dades. TPC-H* es justifiquen els motius pels quals es permet que aquest generador pugui fer simplificacions en alguns atributs. Realment, aquest projecte consisteix en realitzar proves de rendiment exhaustives en la construcció dels cubs de dades, i el TPC-H, tot i ser un benchmark per a

les aplicacions de suport a la presa de decisions, en aquest cas és simplement una referència per tenir un esquema de dades sobre el qual treballar. Per aquest motiu, es permet la simplificació del TPC-H quant a la generació d'atributs però no quant a la cardinalitat de les relacions.

Així doncs, la solució implementada va ser la d'emular les operacions amb nombres reals a partir de nombres enters. Un nombre real està composat per una part entera i una part decimal. En aquesta solució, la part entera és la que es calcula mitjançant nombres i aritmètica entera, i la part decimal es calcula a partir de la part entera resultant. Per tant, la única funció que es va haver d'implementar va ser una tal que, donat un nombre enter (que seria la part entera resultant), retornés un nombre real. El següent pseudocodi descriu aquesta funció:

Algorisme 7.1-15: Generació de nombres reals

```
function generateDouble(var number)
begin
  var decimals := 0
  if number & 0x00000001 = 0
  then
    decimals := ((number & 0x0000003F) + 1) / 100
  else
    decimals := (number & 0x0000003F) / 100
  fi
  return number + decimals
end
```

La justificació d'aquest codi és molt senzilla. Consisteix bàsicament en aplicar la màscara 0x3F per tal d'obtenir un altre nombre enter de com a màxim dos dígits. A l'aplicar la màscara 0x3F, tenim que $0x3F_h \equiv 63_d \equiv 111111_b$, la qual cosa implica que el nombre generat es trobarà a l'interval $[0, 63]$, i per tant la part decimal generada per aquesta funció mai serà més gran que 63. Això és degut a una petita optimització. És ben sabut que les operacions que fa un processador amb menys overhead són les operacions bitwise[33]. D'aquesta manera, tenim que $2^6 - 1 = 63$, i si afegim un bit més a aquesta màscara, tenim que $2^7 - 1 = 127$, i per tant amb 7 bits ja estaríem generant fins a tres dígits decimals. Tot i així, malgrat que la part decimal no pugui ser més gran que 63, no afecta en aquest projecte ja que l'objectiu principal és fer proves de rendiment en la construcció de cubs de dades.

Cal notar que en realitat encara s'està computant una petita operació amb nombres reals: la divisió per 100. Aquesta divisió és fa per tal de poder fer que el resultat de la suma entre la part entera i la part decimal sigui el nombre real resultant. De totes maneres, dividir per potències de 10 és relativament barat en qualsevol CPU en comparació amb altres operacions amb nombres reals.

El detall del *if-else* és per assegurar que el nombre sobre el que s'aplica la màscara no és divisible per 10. Potser no és la forma més neta, pero sí que és una forma òptima ja que es realitza mitjançant una operació bitwise: si el nombre és parell llavors és possible que sigui divisible per 10 ja que tots els nombres divisibles per 10 són parells, i llavors cal sumar-li 1 per tal que no ho sigui. Tot això és degut a que Java, de forma automàtica, elimina els zeros innecessaris de la part decimal. Per exemple: $20/100 = 0.20 = 0.2$, i per tant hem d'evitar aquests casos ja que el TPC-H indica que els atributs de tipus real han de ser generats amb dos decimals. Aquest restricció sí que l'hem de respectar ja que un

[33] Operacions bit a bit.

nombre real en el món relacional ocupa el mateix nombre de bytes independentment del nombre de decimals (fins al màxim permès), però a l'HBase tot el que es guarda és en format string, i per tant perdre un decimal significa perdre un caràcter, que equival a perdre un byte.

7.1.7 Proves de rendiment

7.1.7.1 Clúster virtual

Inicialment, durant la implementació d'aquest generador, les proves de funcionament i de rendiment es van fer sobre un petit clúster local fet a partir de màquines virtuals. L'objectiu d'aquestes proves era assegurar-se del bon funcionament del generador abans no es pugues executar al clúster del DAC. D'aquesta manera, s'evitava treballar en un entorn amb concurrència d'usuaris i amb els problemes que això implica. Aquests problemes són principalment problemes de temps d'espera a causa del sistema de cues que gestiona l'accés de cada usuari als nodes. A més, de no interferir amb d'altres usuaris que segurament sí estarien fent execucions importants.

Així doncs, les primeres proves es van dur a terme en un petit clúster local. Aquest clúster estava format inicialment per tres màquines: un màster i dos esclaus.

- **Master:**
  - Màquina física - Intel Pentium D 3.4GHz (2 CPUs)
  - 2GB RAM
  - Sistema operatiu Ubuntu Desktop 12.04
- **Esclaus:**
  1. Màquina anomenada CentOS:
     - Màquina virtual - 2.2GHz (1 CPU)
     - 1.5GB RAM
     - Sistema operatiu CentOS 6.4
  2. Màquina anomenada Ubuntu:
     - Màquina virtual - 1GHz (1 CPU)
     - 256MB RAM
     - Sistema operatiu Ubuntu Server 10.04

Com es pot veure, la diferència en les especificacions entre una màquina i una altre era bastant diferent. Això va ser degut a que inicialment també es va voler provar el bon funcionament de les tecnologies Hadoop i HBase en un clúster heterogeni. No obstant això, el fet que a l'hora de dur a terme les proves de rendiment en les insercions, el temps pugues ser massa elevat degut a la màquina *Ubuntu*, es va decidir treure-la del clúster. Una opció podria haver estat haver-li donat més memòria RAM, però com que totes dues màquines corrien sobre una mateixa màquina física, no es podien assignar tants recursos a les màquines virtuals. Així, les proves de rendiment es van fer sobre un clúster format únicament pel màster i la màquina *CentOS*.

La següent taula mostra els resultats obtinguts:

Scale Factor	Número de tuples	Volum de dades	Temps d'inserció
SF = 0.01	60.500	210MB	393,13 s
SF = 0.05	302.500	1.2GB	3262,72 s
SF = 0.1	605.000	2.3GB	8100,49 s
SF = 0.5	3.025.000	12.4GB	39608,21 s

Malgrat els temps d'execució continuaven sent massa elevats, es van considerar els resultats com a molt positius degut principalment a dos motius:

- Les insercions eren lentes, però les màquines sobre les que es realitzaven tampoc eren les adequades i per tant aquesta lentitud podia ser molt probablement (i ho va ser, apartat *7.1.7.2 Clúster del DAC*) deguda a una qüestió hardware de les màquines.
- S'havia aconseguit que execucions amb volums de dades que fins ara no s'havien pogut plantejar finalitzessin correctament la seva execució i per tant això demostrava que la implementació del generador havia estat òptima.

Així doncs, ara només era qüestió de traslladar aquest generador al clúster del DAC.

La memòria RAM de les màquines i el heap dels RegionServers

Durant l'execució d'aquestes proves, només es va haver d'afrontar un petit problema.

Per tal de minimitzar el consum que les màquines virtuals en feien de la màquina física sobre la que corrien, es va intentar configurar la memòria RAM consumida al mínim possible. Inicialment es van considerar que amb 1GB n'era suficient, ja que el heap dels RegionServers és exactament 1GB.

Malauradament, aquesta no va ser una suposició acertada, ja que tal i com es mostra a la figura *7.1-11*, un cop el heap del RegionServer arriba al seu punt màxim, l'execució s'atura i cau aquest RegionServer.

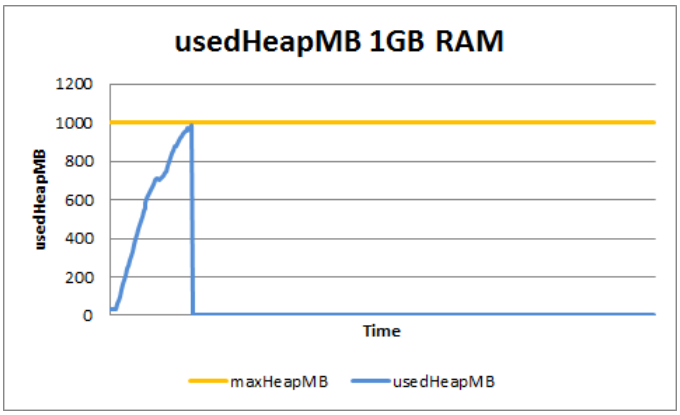


Figura 7.1-11: Evolució del heap del cluster amb 1GB de RAM

El que estava succeint era que el heap creixia automàticament degut als memstores, i quan aquests ocupaven tota la memòria i per tant calia fer flush, no es podia buidar-los i a canvi queia el RegionServer.

Ràpidament es va decidir ampliar un pèl més la memòria RAM de les màquines per tal que el heap dels RegionServers no s'endugués tots els recursos. Tal i com es mostra en la figura *7.1-12*, aquest canvi va donar els fruits esperats.

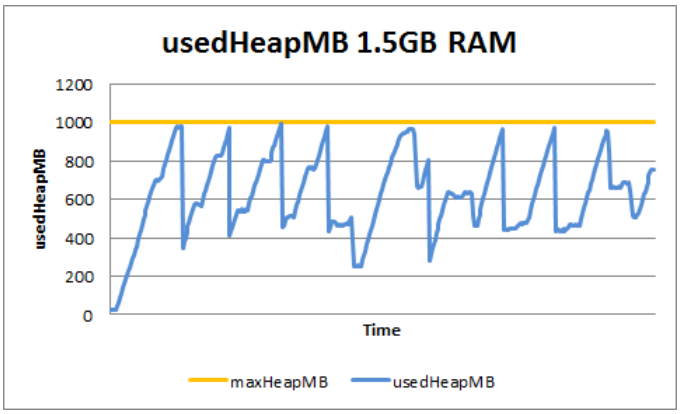


Figura 7.1-12: Evolució del heap del cluster amb 1.5GB de RAM

En aquesta figura, es pot veure com l'evolució del heap del RegionServer és normal. Te una evolució d'un creixement més suau que els punts de decreixement. Això està indicant, tal i com s'ha mencionat anteriorment, l'evolució dels memstores. Aquests creixen contínuament fins que el heap està al seu màxim i aleshores es produeix una davallada en sec degut al flush d'aquest memstores cap a disc.

Així doncs, d'aquesta experiència podem concloure que en el moment de configuració de l'HBase, cal deixar un espai de memòria RAM de marge amb la màquina per tal d'assegurar un funcionament correcte dels RegionServers.

7.1.7.2 Clúster del DAC

Un cop implementat aquest generador i provat el seu funcionament sobre el petit clúster local, es va procedir a fer-lo córrer sobre el clúster del DAC per tal d'obtenir els primers resultats de rendiment i començar a precisar fins a quins volums de dades seria possible arribar. Malauradament, el canvi d'entorn no va afectar positivament a l'execució d'aquest generador i de seguida van aparèixer els problemes:

1. **Caiguda dels RegionServers.** Durant l'execució, els RegionServers s'apagaven fins al punt de quedar un únic RegionServer en marxa al clúster.

2. **Concentració de dades.** Les dades estaven massa concentrades en un sol DataNode en comptes de ser distribuïdes al llarg de tot el clúster.

En realitat, però, aquests dos problemes estaven molt relacionats entre ells, ja que es devien principalment a la forma com s'estava gestionant la fragmentació horitzontal.

Caiguda dels RegionServers

Aquest problema es produïa essencialment degut a la alta velocitat amb la que s'inserien les dades. Era un problema de combinació entre les compactacions i els region split.

A l'apartat *4.2.2 Fragmentació horitzontal: Autosharding* s'explica amb més detall el procés que du a terme l'HBase per tal de realitzar un region split. Bàsicament aquest procés és no moure les dades inicialment, sinó crear un fitxer punter a les dades que han de ser mogudes i, quan es produeixi una compactació en la region nova, moure les dades referenciades a aquesta region.

Així doncs, la causa de que els RegionServers s'anessin aturant un a un era el fet que les insercions s'estaven produint a un ritme tan elevat que les regions noves demanaven un split abans no s'haguessin dut a terme les compactacions, i per tant s'estava demanant un split sobre una region que contenia fitxers punters. A més, sí es pot realitzar un region split d'una region amb fitxers punters si la nova region és assignada al mateix RegionServer, ja que en aquest cas no cal que les dades siguin mogudes. Per aquest motiu sempre quedava un sol RegionServer actiu ja que totes les regions noves només podien anar a parar a ell mateix.

Per tal de posar punt final a aquesta fallida dels RegionServers, va caldre analitzar el propi codi de l'HBase per tal de trobar el punt en el que es decideix si una region pot fer split o no. Concretament, aquest punt es troba al fitxer *Store.java*. Aquest fitxer conté la funció *getSplitPoint()* que signa de la següent manera[34]:

getSplitPoint
protected byte[] getSplitPoint()

Returns: the key at which the region should be split, or null if it cannot be split. This will only be called if shouldSplit previously returned true.

Però certament, analitzant el codi d'aquesta funció es va poder comprovar, com en cas de ser una region amb fitxers punters, aquesta funció mai retorna *null*, sinó que prèviament hi ha una asserció[35] forçada a fals que al no satisfer-se finalitza l'execució, i per aquest motiu el RegionServer s'aturava.

D'aquesta manera, la solució que es va aplicar va ser la d'eliminar aquesta asserció i compilar una versió pròpia de l'HBase. La figura *7.1-13* mostra aquest canvi en el codi:

Com es pot veure en aquesta figura, el text que informa de que la region no ha pogut fer split ara forma part dels logs per tal de poder fer un seguiment més exacte d'aquests errors.

No obstant a totes aquestes explicacions, aquesta solució causa una forta inestabilitat en els temps d'inserció entre diferents execucions. Això és degut a que, eliminant del codi l'asserció anteriorment comentada, una region que demani split però no pugui degut a que la funció *getSplitPoint()* retorna *null* i s'estarà sobrecarregant. Si es donen les circumstàncies de que aquesta region fa les compactacions a temps, aleshores l'execució accelerarà ja que es guanyarà en paral·lelisme a raó de que hi hauran dues màquines més treballant alhora, però si no es produeixen aquestes compactacions, aleshores no hi haurà paral·lelisme ja que només treballarà una màquina i per tant el temps final d'execució empitjorarà.

Tanmateix, els factors que fan que les compactacions es duguin a temps o no són factors externs al nostre àmbit de treball, així com per exemple ho són la càrrega de la màquines, les seves especificacions, etc.

Per tal de minimitzar el nombre de vegades que una region no pot fer split degut a una compactació pendent, un petit pegat aplicat ha estat el d'executar aquest generador en una màquina més lenta que els RegionServers per tal de donar-los més marge de treball, i així evitar que aquests mateixos RegionServers es converteixin en una mena de coll d'ampolla per sobrecàrrega i alenteixin les compactacions.

De totes maneres, es van provar altres solucions abans d'aquesta però cap d'elles va donar el fruit esperat. Aquestes solucions es basaven en reduir el nombre de regions tot

[34] Documentació oficial de l'HBase.

[35] En Java, una asserció és una sentència que atura l'execució i mostra un determinat missatge en cas de no complir-se una condició que rep com a paràmetre.

```java
/**
 * Determines if Store should be split
 * @return byte[] if store should be split, null otherwise.
 */
public byte[] getSplitPoint() {
  this.lock.readLock().lock();
  try {
    // sanity checks
    if (this.storefiles.isEmpty()) {
      return null;
    }
    // Should already be enforced by the split policy!
    assert !this.region.getRegionInfo().isMetaRegion();

    // Not splitable if we find a reference store file present in the store.
    long maxSize = 0L;
    StoreFile largestSf = null;
    for (StoreFile sf : storefiles) {
      if (sf.isReference()) {
        // Should already be enforced since we return false in this case
        assert false : "getSplitPoint() called on a region that can't split!";
        return null;
      }
```

```java
/**
 * Determines if Store should be split
 * @return byte[] if store should be split, null otherwise.
 */
public byte[] getSplitPoint() {
  this.lock.readLock().lock();
  try {
    // sanity checks
    if (this.storefiles.isEmpty()) {
      return null;
    }
    // Should already be enforced by the split policy!
    assert !this.region.getRegionInfo().isMetaRegion();

    // Not splitable if we find a reference store file present in the store.
    long maxSize = 0L;
    StoreFile largestSf = null;
    for (StoreFile sf : storefiles) {
      if (sf.isReference()) {
        // Should already be enforced since we return false in this case
        //assert false : "getSplitPoint() called on a region that can't split!";
        LOG.warn("getSplitPoint() called on a region that can't split!");
        return null;
      }
```

Figura 7.1-13: Modificació del codi de l'HBase

jugant amb el tamany dels memstores i la política de autosharding per defecte (veure apartat *4.2.2 Fragmentació horitzontal: Autosharding*), i per tant donar més marge de temps a les compactacions abans no es demanés un region split. Però no hi va haver èxit.

A més, si es vol mantenir el comportament de la política d'autosharding per defecte alhora que s'amplia el tamany dels memstores s'han d'ajustar correctament els paràmetres de bloqueig:

a) *hbase.regionserver.global.memstore.upperLimit*: Tamany màxim dels memstores abans no es bloquegin noves escriptures i es produeixi flush d'aquests memstores.

b) *hbase.regionserver.global.memstore.lowerLimit*: Tamany màxim dels memstores abans no siguin forçats a fer flush.

Donat que aquests paràmetres són funció del heap de memòria dels RegionServers, ocorre que ampliar el tamany dels memstores implica ampliar també el tamany d'aquest

heap en la mateixa proporció per tal de no alterar el comportament per defecte. Així doncs, si per defecte el tamany del memstore és de 128MB i del heap és 1GB, és fàcil calcular com amb 512MB de memstore calen 4GB de heap, o com amb 1GB de memstore calen 8GB de heap i per tant aquesta solució, a part de no funcionar correctament, es fa del tot inviable degut al gran espai de memòria que requereix el heap dels RegionServers.

Concentració de dades

Aquest problema guardava certa relació amb l'anterior, ja que part de la causa era també el mecanisme de region split que empra l'HBase. Bàsicament es tractava de que, al ser les compactacions tan lentes, els region split no es produïen amb suficient freqüència. Tot això, combinat amb el fet que les insercions es feien ordenadament (veure apartat *7.1.6 Optimització de les insercions*), feia que les noves tuples fossin assignades a la mateixa region, i per tant en el mateix RegionServer. Aquest fet es convertia en certa manera en "el peix que es mossega la cua", ja que es sobrecarregaven massa certs RegionServers i per tant les compactacions eren encara més lentes, a la vegada també ho eren els region splits i les dades seguien molt locals a aquests RegionServers.

Identificat el problema, la solució var ser immediata. Aquesta es tractava de modificar les insercions tal que cada thread comencés en una tupla diferent. Per exemple, el primer thread insereix a partir de la primera tupla, el segon thread a partir de la segona tupla, i així successivament. Quan els threads arriben a la última tupla, continuen per la primera fins inserir-les totes. D'aquesta manera, s'aprofiten els pocs region split que es puguin haver produït, i al fer que diferents insercions vagin a diferents RegionServers s'aconsegueix fer un procés d'inserció més distribuït i per tant no sobrecarregar tan en excès les màquines.

En certa manera aquesta solució és contraposada al que s'explica a l'apartat *7.1.6 Optimització de les insercions* degut a que l'autoflush ara ja no s'aprofita tant com s'exposa en aquest apartat. A més, tot i ser negligible, cada cop que els threads arriben a la última tupla i continuen per la primera, han de fer una ordenació del buffer local tal i com es detalla a l'apartat *4.2.2 Buffer d'escriptura (client-side): Autoflush*. A canvi, el balanç final és que es guanya en paral·lelisme ja que es distribueix la càrrega al llarg de tots els RegionServers i per tant els temps d'inserció milloren.

Malgrat aquesta solució, hi havia una segona causa que interferia en la correcta distribució de les dades. Concretament, es tractava del fet que per tal de moure les dades de RegionServer un cop fet un region split, cal una compactació. Si aquesta compactació no arriba, la dades de la nova region no es mouen al nou RegionServer on són assignades.

El que succeïa aleshores era que amb SF petits (concretament $SF \leq 1$) de seguida es demanaven region splits degut a la política per defecte (apartat *4.2.2 Fragmentació horitzontal: Autosharding*), i a la vegada, amb SF petits, les insercions estaven de seguida fetes i finalitzava l'execució del generador tot deixant regions noves a l'espera d'una compactació per moure les dades. Així doncs, aquesta segona causa era un problema de falta de dades.

Malauradament, no es va trobar cap solució concreta i el que es va fer va ser començar a treballar amb volums de dades més grans. D'aquesta manera, la freqüència amb la que es produeixen region splits es redueix amb el temps i es dona marge a que les últimes insercions provoquin les compactacions necessàries per tal de moure les dades correctament.

Deixant de banda els problemes mencionats anteriorment i havent aplicat les solucions proposades, l'execució d'aquest generador en el clúster del DAC va permetre esbrinar certs aspectes del funcionament intern de l'HBase:

1. **Identificació dels region splits.** La figura *7.1-14* il·lustra l'evolució conjunta dels

memstores i del nombre de storefiles totals, i es poden identificar les tres situacions següents:

- Les caigudes en la gràfica dels memstores corresponen a l'instant en que aquests fan flush.
- Les caigudes en la gràfica dels storefiles indiquen que s'ha produït una compactació.
- Malgrat hi pugui haver una petita diferència en els instants de temps entre els memstores i el nombre dels storefiles degut al temps de CPU, es pot veure com també, en la majoria dels casos, un flush en un memstore implica la creació de nous storefiles.

No obstant això, el que clarament salta a la vista en aquesta figura són els instants 18 i 79 en els que es produeix un flush complert dels memstores alhora que el nombre de storefiles augmenta bruscament. El que està succeint en aquests dos instants és representa més bé en la figura *7.1-15*.

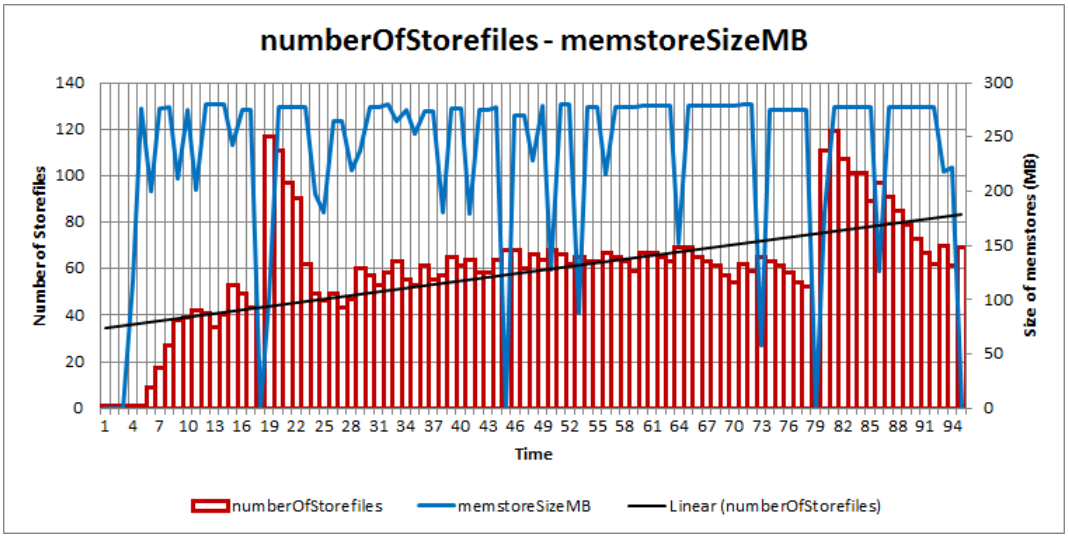


Figura 7.1-14: Evolució dels paràmetres *numberOfStorefiles* i *memstoreSizeMB*

Aquesta figura il·lustra l'evolució conjunta del nombre de regions i de stores. Degut a que existeix una store per cada column family i region, és lògic que ambdues gràfiques segueixin la mateixa evolució. Com es pot veure en aquesta figura:

i) Inicialment tant el nombre de regions com de stores és 2, ja que amb l'HBase buit només hi apareixen les metataules[36] *ROOT* i *META*. Cada una d'aquestes metataules està fragmentada en una sola region i cada region conté un sol store inicialment.

ii) A l'instant 2 es pot veure com s'ha creat la taula per inserir les dades ja que el nombre de regions augmenta a 3 i el nombre de stores a 10. Ara, el nombre de regions és 3 perquè a més de les dues regions anteriors, també tenim una region per a la taula on s'inseriran les dades. I el nombre de stores és 10, perquè la nova taula està formada per 8 column families, que són: *lineitem, orders, customer, partsupp, part, supplier, nation* i *region*.

[36] Taules internes de l'HBase.

iii) A l'instant 18 el que realment és pot observar que està succeïnt és un region split:

a) Inicialment la region a fer split s'oculta i per això el nombre de regions a la gràfica cau de 3 a 2.

b) Posteriorment, s'arrenca tant la region antiga com la nova i aleshores:

- El nombre de regions ha augmentat en 1 respecte abans del region split.
- Conseqüentment el nombre de storefiles ha augmentat en 8, ja que la nova region de la taula conté un store per cada una de les vuit column families mencionades anteriorment.

iv) De la mateixa manera, a l'instant 79 es torna a produir un altre region split.

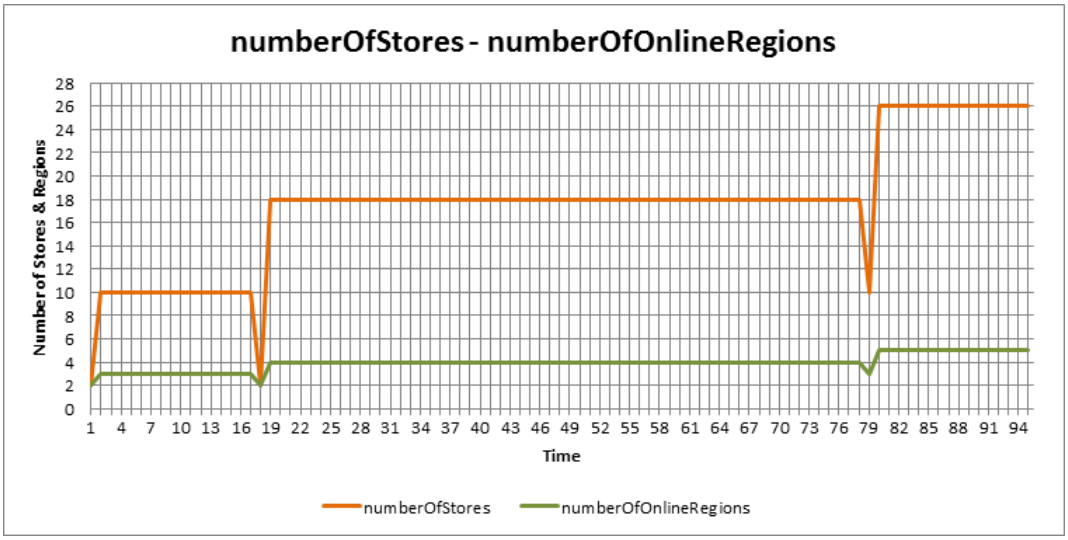


Figura 7.1-15: Evolució dels paràmetres *numberOfStores* i *numberOfOnlineRegions*

Així doncs, podem concluir que la figura *7.1-14* també està representant dos region splits als instants 18 i 79. Efectivament, a continuació del flush del memstore, el nombre de storefiles creix tan bruscament degut als storefiles de la nova region. Com que aquests storefiles són principalment punters a les dades, el que es veu després dels region split és com aquest mateix nombre de storefiles es redueix notablement, ja que es tracta de les compactacions que mouen les dades d'un RegionServer a un altre i de l'eliminació dels storefiles de l'antiga region (veure apartat *4.2.2 Fragmentació horitzontal: Autosharding*).

2. **Distribució de les dades.** Un dels objectius d'aquest projecte era descobrir certs comportaments interns de l'HBase. Un d'aquests comportaments era el model de distribució de les dades al llarg de tot el clúster. Esbrinar aquest factor va ser un punt clau en aquest projecte, ja que certament la teoria indica que l'HDFS trosseja els seus fitxers en fitxers més petits que després distribueix al llarg de tot el clúster i per tant, tenint en compte que l'HBase corre per damunt de l'HDFS, és lògic pensar que els fitxers que contenen les dades també són trossejats i repartits al llarg dels diferents nodes. No obstant això, aquest funcionament seria contradictori amb l'estructura de fitxers presentada al llarg de tot l'apartat *4.2.2 HBase*.

   Així doncs, un cop més la realitat s'oposa a la teoria. El funcionament intern intenta explotar la localitat espacial entre RegionServer i DataNode, tot materialitzant

les dades corresponents de cada region al mateix DataNode sobre el que corre el RegionServer on són assignades. De fet, aquest funcionament alhora també explica perquè és obligatori que tot RegionServer corri sobre un DataNode.

De totes maneres, no s'ha d'oblidar que l'HDFS és un sistema distribuït i per tant, independentment d'on s'escriguin finalment les dades, el sistema funciona correctament. Un exemple és la figura *7.1-16* on s'indica la distribució de les dades amb un SF de 1. En aquest cas, les dades no estàn ben distribuïdes pels motius presentats al punt anterior *7.1.7.2 Concentració de dades* però serveix per demostrar que l'HBase funciona independentment de la distribució de les dades.

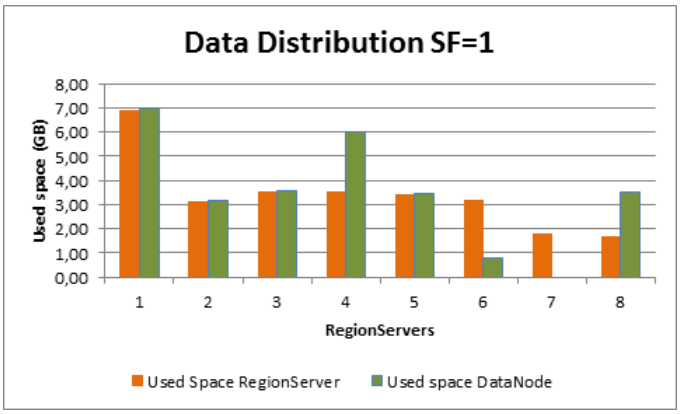
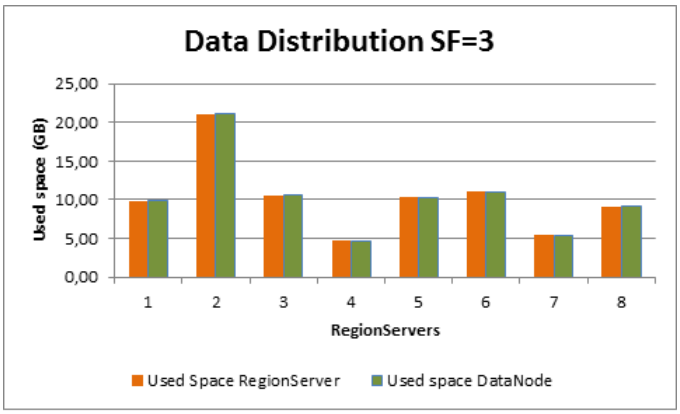


Figura 7.1-16: Distribució de les dades amb un $SF = 1$

A la figura *7.1-17* sí es fa una distribució correcte de les dades i s'hi pot observar la localitat RegionServer-DataNode comentada anteriorment. Com és lògic, el valor ocupat pels DataNodes és lleugerament superior en tots els casos ja que el sistema de fitxers conté dades addicionals (per exemple sobre el seu funcionament) a més de les pròpies de l'HBase.

També és pot observar com el tamany ocupat per cada RegionServer és semblant amb alguns però alhora molt diferent amb altres. Això és principalment degut a la política de autosharding emprada. Amb aquesta política, el volum de les regions de cada RegionServer creix exponencialment fins a un cert límit a partir del qual creix de forma línial (veure apartat *4.2.2 Fragmentació horitzontal: Autosharding*). D'aquesta manera, el balanç que fa l'HBase és respecte el nombre de regions que hi ha a cada RegionServer, però una nova region implica un nou volum de dades més gran que qualsevol region més antiga. Per exemple, la figura *7.1-18* mostra la diferència entre l'espai ocupat entre un RegionServer i un altre és exponencial amb tan sols una region assignada més de diferència.

Finalment, a continuació es mostra una comparativa dels temps d'inserció i el volum de dades inserit amb la següent configuració, que està formada per diferents paràmetres que s'avaluen en les proves posteriors i per tant fan que aquesta configuració no sigui sempre fixe:



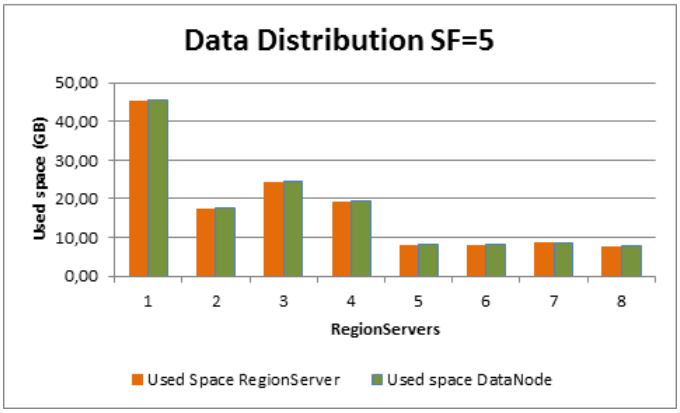


Figura 7.1-17: Distribució de les dades amb un $SF = [3, 5]$

Paràmetre de configuració	**Valor**
Estructura de les column families	Una column family per a cada taula del model relacional del TPC-H. Es construeixen en total 8 column families.
Replicació HDFS	1 (un valor original i cap rèplica).
Nombre de RegionServers	8.

I una descripció dels recursos hardware de les màquines on es van realitzar aquestes proves:

- **1 master:**
  - 2 processadors Intel Xeon Quad-Core L5630 2.13GHz
  - El sistema de gestió de recursos permetia treballar fins amb 6 CPUs per màquina
  - 24 GB memòria RAM en 6 mòduls de 4GB
  - 2 Targetes de xarxa Intel Gigabit Ethernet
- **8 esclaus:**
  - Processador Intel Xeon E5-2630L
  - El sistema de gestió de recursos permetia treballar fins amb 10 CPUs per màquina

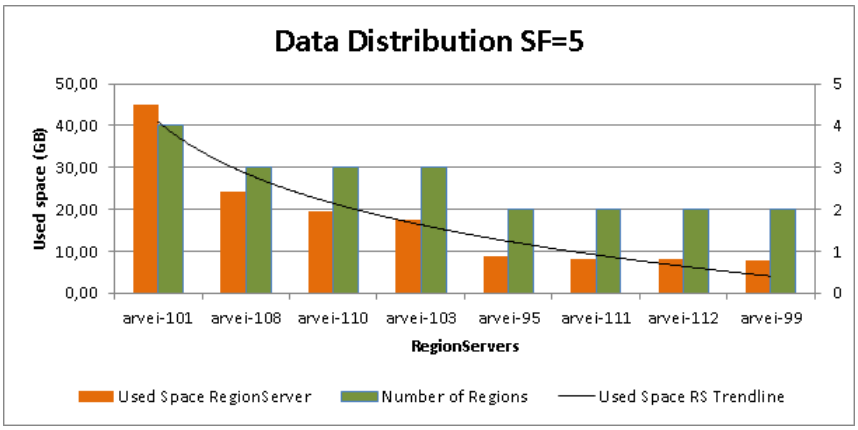


Figura 7.1-18: Comparativa de l'espai ocupat i de les regions per RegionServer

- 64 GB memòria RAM
- 4 Targetes de xarxa Intel Gigabit Ethernet

Així doncs, la comparativa del volum de dades és:

Scale Factor	Número de tuples	Volum dades (GB)
SF = 1	6.050.000	27,30
SF = 3	18.150.000	81,74
SF = 5	30.250.000	139,06

I dels temps d'inserció:

Scale Factor	Inserció (s)	Compactació (s)	TOTAL (h)
SF = 1	1558,988	360	0,53
SF = 3	2860,12	1260	1,14
SF = 5	10056,42	11880	6,09

El temps d'inserció mostrat a la taula anterior correspon purament al temps requerit per inserir un determinat volum de dades. El temps de compactació és posterior al d'inserció, i identifica el temps necessari per tal de que es produeixin totes les compactacions demanades i es faci un balanceig correcte de les dades (veure apartat *4.2.2 Fragmentació horitzontal: Autosharding*).

7.2 Etapa II: Construcció dels cubs de dades. MapReduce

7.2.1 Adequació dels algorismes al nou entorn

7.2.1.1 Canvis en els índexs

El conjunt de canvis més important es va fer en el procés de creació dels índexs. Això va ser degut especialment al fet de treballar amb un nou volum de dades molt més gran que pels quals estaven pensats.

Més concretament, les modificacions que va caldre fer van ser essencialment dos. Totes elles tenen en comú que el seu objectiu és fragmentar d'una forma o altra aquests índexs per evitar sobrepassar certs límits. Aquestes modificacions són:

1. **MaxValues:** La longitud d'una tupla en qualsevol taula està fixada a un màxim. Llavors, amb els volums de dades amb els que es treballa ocorria que el nombre de keys de la taula de fets per un valor i un atribut donats era massa extens i les entrades dels índexs superaven aquest llindar màxim.

2. **Buckets:** Aquest particionament es va fer per evitar un consum excessiu de memòria donades certes condicions. Aquestes són: volum de dades molt gran i un nombre de valors diferents per un atribut (*ndist* d'ara endavant) molt petit. Aleshores, si recordem el procés de creació dels índexs a l'apartat *5.2 Creació dels índexs*, aquests es construeixen mitjançant una tasca MapReduce. És en aquest moment quan es pot produir un desbordament de memòria degut a que amb *ndist* més petits, més gran és el nombre de keys per atribut i per tant arriba un moment en què la memòria no és suficient per emmagatzemar aquests llistats tan grans.

Cal mencionar que malgrat aquestes modificacions es presentin en aquest ordre, realment la implementació de **buckets** inclou la implementació de **maxvalues**. Tot i així, s'expliquen en aquest ordre contrari ja que es tracte de l'ordre en què es van implementar. Primerament, va sorgir el problema solucionat amb *maxvalues* i per tant es va fer aquesta primera implementació, i a continuació la segona implementació que va implicar modificar la primera per tal que l'inclogués.

Així doncs, en els apartats següents s'explica amb més deteniment aquestes dues modificacions. Primerament, però, cal mencionar que els noms escollits per a totes dues solucions venen donats pel canvi de sintaxis a la creació dels índexs, que ara és del tipus:

CREATE DIMENSION *name* **ATTRIBUTES** *attrs* **BUCKETS** *B* **MAXVALUES** *MV*

On *name* és el nom de la dimensió, *attrs* és el llistat d'atributs separat per espais, i B i MV són dos nombres naturals.

Extensió de MaxValues

Aquesta extensió està enfocada en particionar en subconjunts el llistat sencer de keys de cada entrada de l'índex de tal manera que en cap cas una d'aquestes entrades ocupades pugui ocupar més espai del permès.

El valor d'aquest espai ve marcat pel paràmetre de configuració: *hbase.client.keyvalue.maxsize*, que per defecte pren el valor 10485760 bytes = 10 MB. Llavors, la primera solució va ser la de modificar aquest valor. Concretament és va establir a 0, la qual cosa indica que no hi ha límit d'espai per a una tupla de l'HBase. El cert és que aquesta primera modificació va funcionar correctament, però donava peu a què poguessin existir certs problemes de comportament de l'HBase. El més típic d'aquests problemes es deu a la fragmentació horitzontal, ja que si recordem l'apartat *4.2.2 Fragmentació horitzontal: Autosharding*, la política de region split per defecte es basa en el tamany dels storefiles. Si no es limita el valor de les tuples, aleshores es podria donar la situació en què un storefile que conté només una tupla pugui ocupar lo suficient com per aplicar-hi un region split. Aleshores, es tractaria d'una situació de controvèrsia ja que malgrat que l'storefile hagi de fer split, no és possible fer-ho perquè no existeix cap punt de split degut a què es tracta d'una sola tupla.

Segurament aquesta situació es traduiria en una caiguda de rendiment degut a la falta de paral·lelisme que s'arrossegaria i per tant està clar que millor evitar aquest tipus de solucions.

Aleshores va sorgir la opció de **maxvalues**. Aquesta implementació consisteix en modificar la sintaxis de la construcció d'índexs de tal forma que es permeti definir un màxim de keys M que contindrà cada entrada. D'aquesta manera, s'obliga a desglossar el conjunt total de keys en conjunts més petits de fins a M keys com a màxim.

El pseudocodi *7.2-1 Implementació dels maxvalues* mostra les corresponents modificacions respecte el codi de creació d'índexs original *5.2-1 Construcció del Single Level Index*. Aquelles funcions que no ha calgut modificar (per exemple la funció map), han estat obviades ja que no s'ha modificat el codi.

Algorisme 7.2-1: Implementació dels *maxvalues*

```
action launch(var table, array attributes[0 .. A], var name, var maxvalues)
begin
  var scan := createScan(table, attributes)
  setMapReduceConfiguration(''attributes'', attributes)
  setMapReduceConfiguration(''dimension_name'', name)
  setMapReduceConfiguration(''maxvalues'', maxvalues)
  launchMapReduce(scan)
end

function makeKey(key, subset)
begin
  var tmp_key := concat(key, ''%'' subset)
  return tmp_key
end

action reduce(var<key, value[0 .. V]> dimension)
begin
  var first := true
  var result := EMPTY_STRING
  var \textit{maxvalues} := getMapReduceConfiguration(''maxvalues'')
  var length := 0
  var subset := 1
  for v in value
  do
    if first = true
    then
      result := v
    else
      result := concat(result, '','', v)
      if length >= maxvalues
      then
        var tmp_key := makeKey(key, subset)
        emit(tmp_key, result)
        subset := subset + 1
        result := EMPTY_STRING
        length := 0
    fi
    length := length + 1
  done
  var tmp_key := makeKey(key, subset)
  emit(tmp_key, result)
end
```

És fàcil veure com senzillament la funció de reduce s'ha modificat de forma que durant

el bucle de concatenació de les keys, i per tant de la construcció de la entrada de l'índex, es manté un indicador del nombre de keys ja indexades. Si aquest indicador supera el valor permès, aleshores es concatena el número de partició a la key de la dimensió i es fa una emissió del key-value format. Aleshores, es reinicien els indicadors i s'inicia una segona volta.

És important notar com per tal de concatenar cada partició amb la key de la dimensió, la solució implementada és la d'afegir el número de partició al final d'aquesta key. Així, el procés de selecció vist a l'apartat *5.3 Accés als índexs. Selecció* continua sent vàlid, ja que d'aquesta manera són els propis valors dels atributs els que continuen marcant el prefix de cada entrada de l'índex. En certa manera, aquesta solució consisteix en crear un últim nivell d'agregació simulat però transparent a l'usuari, i que per tant permet dur a terme aquest particionament sense més complicació.

Malauradament, aquesta implementació no s'ajusta als requisits del Multiple Level Index. Si recordem l'apartat *5.2.3 Multiple Level Index*, en aquest tipus d'índex s'afegia una paraula al final de cada entrada de la dimensió per tal de poder diferenciar clarament els nivells d'agregació intermitjos respecte els nivells inferiors. Aleshores, afegir el número de partició *maxvalues* com a últim nivell d'agregació implica que aquesta paraula ja no es situa exactament al final dels nivells d'agregació pròpiament dits, sinó que es situa inclús després del número de partició.

A més a més, tal i com es comenta anteriorment, afegir el número de partició al final és per tal de buscar transparència de cara a l'usuari, i per tant una query del tipus:

SELECT SUM(vendes) **WHERE** Date = 2013%All

Hauria de retornar la suma de les vendes on la data és 2013, però malauradament no retornaria res sobre un Multiple Level Index. En realitat, el que ocorre és que si aquesta entrada de l'índex és segmentada segons la implementació maxvalue, les keys fragmentades resultants serien: *Date%2013%MV0%All, Date%2013%MV1%All*, etc. I per tant és fàcil veure com la tècnica de prefix per fer la selecció deixa de complir-se. Aleshores el què cal fer és moure el número de partició per a què sigui posterior a la paraula "All".

En aquest moment, es va aprofitar aquest canvi per elaborar d'una forma més entenedora l'estructura d'entrades a l'índex. Concretament, es va moure la paraula "All" al primer nivell d'agregació, ja que es on realment li correspon estar pel seu significat (en termes de nivells d'agregació) i la paraula que identifica l'últim nivell d'agregació es va convertir en "End".

Així per exemple, una entrada del Multiple Level Index amb aquestes dues implementacions i $maxvalues = 3$ s'hauria de convertir en:

Rows de la taula de dimensions	**Llistat de keys**
Date%All%2013%Octubre%15%End%MV1	1, 4, 6
Date%All%2013%Octubre%15%End%MV2	7, 8, 16
Date%All%2013%Octubre%31%End%MV1	2, 3, 9
Date%All%2013%Octubre%31%End%MV2	12, 15, 17
Date%All%2013%Octubre%31%End%MV3	19
Date%All%2013%Novembre%1%End%MV1	5, 10, 11
Date%All%2012%Gener%4%End%MV1	13, 14, 20
Date%All%2012%Gener%4%End%MV2	26, 28

Aquestes modificacions s'apliquen també als índexs Single Level Index per tal d'aconseguir tenir una estructura d'índexs més genèrica.

D'aquesta manera, es diferencia molt clarament els nivells d'agregació propis de la dimensió respecte els afegits per aquesta implementació i es permet treballar conjuntament amb la implementació del Multiple Level Index, la implementació dels *maxvalues*, i la tècnica de prefixació per realitzar la selecció.

Juntament amb aquest canvi de nomenclatura, es van fer les corresponents modificacions per tal de donar llibertat a l'usuari a l'hora d'executar les queries. Així, les següents queries són la mateixa tant pel Single com pel Multiple Level Index:

```
SELECT SUM(vendes) WHERE Date = 2013
SELECT SUM(vendes) WHERE Date = All%2013
```

I en cas de només el Multiple Level Index, també són la mateixa que les queries:

```
SELECT SUM(vendes) WHERE Date = 2013%End
SELECT SUM(vendes) WHERE Date = All%2013%End
```

En qualsevol cas, l'objectiu de la paraula "All" ara és simplement permetre accedir a tot el conjunt de keys d'una entrada de l'índex independentment dels seus valors als nivells d'agregació. Així per exemple, obtenir totes les keys de la dimensió *Date*:

```
SELECT SUM(vendes) WHERE Date = All%
```

Els codis resultants no es mostren ja que les modificacions necessàries són més aviat petites. Simplement es tracta d'afegir la clàusula "All" a l'inici i la clàusula "End" al final de la construcció de l'entrada de l'índex. A la vegada, es van haver de fer petites modificacions en punts molt puntuals del codi per tal d'assegurar la tècnica de prefix en la selecció al marge de si l'usuari ha executa la query explícitament amb la clàusula "All" o no.

Extensió de Buckets

Tal i com s'ha mencionat anteriorment, aquesta extensió pretén solucionar un problema de memòria degut principalment als atributs amb un *ndist* molt baix. Aleshores, aquesta solució es basa en simular que els atributs tenen un *ndist* més gran que el real. D'aquesta manera, s'aconsegueix que amb aquest nou *ndist* el conjunt total de keys es distribueixi al llarg de més grups i per tant els pics de memòria de processar aquests grups als Reducers siguin més petits.

Per dur a terme tot això, el que es fa és aplicar una simple funció de hash. Concretament, s'aplica la funció mòdul a les keys de la taula de fets per decidir a quina entrada de l'índex han d'anar. Així, si per exemple el nombre de **buckets** definit és 2, es diferencia entre keys senars i keys parelles (per cada valor de l'atribut) i per tant el nombre total de conjunts de keys que s'envia als Reducers és multiplica per 2, però alhora la cardinalitat de cada un d'aquests conjunts es divideix entre 2 i els pics de memòria decreixen.

Per tal de mostrar les modificacions respecte el codi original, el pseudocodi *7.2-2 Implementació dels buckets* mostra les modificacions necessàries per a dur a terme aquesta extensió, i es construeix a partir de l'algorisme original *5.2-1 Construcció del Single Level Index*.

Algorisme 7.2-2: Implementació dels *buckets*

```
action launch(var table, array attributes[0 .. A], var name, var buckets)
begin
  var scan := createScan(table, attributes)
  setMapReduceConfiguration(''attributes'', attributes)
  setMapReduceConfiguration(''dimension_name'', name)
  setMapReduceConfiguration(''buckets'', buckets)
  launchMapReduce(scan)
end

action map(var<key, value> row)
begin
  array attributes[0 .. A] := getMapReduceConfiguration(''attributes'')
  var dimension := getMapReduceConfiguration(''name'')
  var \textit{buckets} := getMapReduceConfiguration(''buckets'')
  for a in attributes
  do
    var v := getValue(value, a)
    dimension := concat(dimension, ''%'', v)
  done
  var b := key % buckets
  dimension := concat(dimension, ''%'', b)
  emit(dimension, key)
end
```

El codi del reduce no es modifica i per tant no forma part d'aquest pseudocodi. Aleshores, tal i com es pot veure, les funcions modificades són les funcions de llançar la tasca MapReduce i la funció map. La funció de llançar simplement es modifica per a que la tasca MapReduce contingui un nou paràmetre de configuració que indiqui el nombre de *buckets* en els que cal trencar cada valor. Les modificacions importants són a la funció map.

Seguint les explicacions anteriors, la primera modificació de la funció map és la que s'encarrega de calcular a quin bucket caldrà finalment assignar cada key de la taula de fets. Cal recordar que les keys en aquest projecte no tenen completament un format numèric i per tant, per poder aplicar la operació mòdul cal fer alguna conversió a un altre tipus de format. Aquesta discussió es pot trobar a l'apartat *7.2.1.2 Tractament de keys consecutives*. Aleshores, un cop calculat aquest bucket només cal concatenar-lo amb la key de la dimensió i finalment fer l'emissió. Cal mencionar que el número de bucket s'afegeix també al final de l'entrada de l'índex i per tant la discussió de l'apartat *7.2.1.1 Extensió de MaxValues* s'aplica perfectament a aquesta qüestió.

A la introducció d'aquest apartat es mencionava que la implementació d'aquesta solució inclou la implementació anterior. Això vol dir que per tal de poder aplicar un índex particionat segons el paràmetre *maxvalues* cal primerament fragmentar-lo en *buckets*. Aquesta és una conseqüència d'una limitació Java. Degut a què totes dues implementacions han de rebre com a entrada un nombre enter que marcarà el nombre final de particions, totes dues extensions defineixen el mateix constructor Java:

Dimension(**string** name, **string**[] attributes, **int** buckets)
Dimension(**string** name, **string**[] attributes, **int** maxvalues)

que clarament no pot existir per duplicat: Dimension(**string**, **string**, **int**). Per tant, es va haver d'afegir un nou constructor amb tots dos paràmetres:

Dimension(**string** name, **string**[] attributes, **int** buckets) Dimension(**string** name, **string**[] attributes, **int** buckets, **int** maxvalues)

I per tant, per tal de fer servir la opció *maxvalues* cal primerament fer servir la opció *buckets*. La part positiva de tot això és que a continuació va resultar més fàcil modificar el parseig de la dimensió ja que les tres sintaxis possibles:

CREATE DIMENSION *name* **ATTRIBUTES** *attrs* **CREATE DIMENSION** *name* **ATTRIBUTES** *attrs* **BUCKETS** *B* **CREATE DIMENSION** *name* **ATTRIBUTES** *attrs* **BUCKETS** *B* **MAXVALUES** *MV*

són una l'extensió de l'anterior, i per tant no s'han de tenir en compte possible disjuncions de sintaxis. Tot al contrari de si els paràmetres *maxvalues* i *buckets* haguessin estat totalment independents:

CREATE DIMENSION *name* **ATTRIBUTES** *attrs* **BUCKETS** *B* **CREATE DIMENSION** *name* **ATTRIBUTES** *attrs* **MAXVALUES** *MV*

Finalment, la opció d'incloure *maxvalues* dins de la implementació de *buckets* o viceversa és totalment indiferent. En aquest cas es va decidir fer-ho simplement seguint l'ordre natural MapReduce: els *buckets* es calculen a les funcions de map, i els *maxvalues* a la funció de reduce.

Deixant de banda les explicacions anteriors de cada una de les implementacions fetes, la següent taula mostra un exemple de com es veuria un cert Single Level Index amb:

- **Nombre de buckets:** 2
- **Nombre de maxvalues:** 3

Rows de la taula de dimensions	**Llistat de keys**
Date%All%2013%Octubre%15%End%B0MV1	4, 6, 8
Date%All%2013%Octubre%15%End%B0MV2	16
Date%All%2013%Octubre%15%End%B1MV1	1, 7
Date%All%2013%Octubre%31%End%B0MV1	2, 12
Date%All%2013%Octubre%31%End%B1MV1	3, 9, 19
Date%All%2013%Octubre%31%End%B1MV2	15, 17
Date%All%2013%Novembre%1%End%B0MV1	10
Date%All%2013%Novembre%1%End%B1MV1	5, 11
Date%All%2012%Gener%4%End%B0MV1	14, 20, 28
Date%All%2012%Gener%4%End%B0MV2	14, 22, 26
Date%All%2012%Gener%4%End%B1MV1	13

En aquest exemple, les nomenclatures *B0* i *B1* indiquen cada un dels *buckets* on s'insereixen les keys amb el mateix valor dimensional. Les nomenclatures *MV0* i *MV1* indiquen els particionaments degut a què el nombre de keys sobrepassa el permès. Òbviament, la construcció dels índexs mitjançant qualsevol d'aquestes implementacions implica el cost de llegir una entrada més de l'índex i per tant una caiguda en el rendiment. En aquest sentit, la recomanació és reduir al mínim el nombre de *buckets* i maximitzar els valors per entrada en els *maxvalues*.

7.2.1.2 Tractament de keys consecutives

Un segon canvi que es va haver de fer va ser el d'implementar funcions que convertissin el format de les keys d'aquest projecte al format del projecte anterior, i viceversa. Això va ser degut al fet que els algorismes de consulta s'aprofiten molt del disseny de la key del projecte anterior, i tal com s'explica a l'apartat *7.1.5 Requisit II: Localitat de les dades* aquest projecte fa servir un disseny diferent per tal de garantir que es trenca la localitat de les dades.

No obstant això, el canvi no és excessivament significatiu i per tant les funcions de conversió són relativament senzilles. Si recordem, el format de key que es segueix en aquest projecte és: *KEY_CONSECUTIVA*t*THREAD*. La part *KEY_CONSECUTIVA* indica el número de key generat pel thread, i *THREAD* indica el número de thread en qüestió. Per exemple, la key *0000000123t12* és la tupla 123 generada pel thread 12. Cal recordar que el procés d'inserció de l'HBase es fa tot seguint un aplicatiu multithreading. Així, quan en aquest apartat es parli de **key consecutiva** es referirà a la part *KEY_CONSECUTIVA*, mentre que quan es parli simplement de **key** es referirà a la key d'una tupla de la taula de fets en el seu format total.

D'aquesta manera, el guany és aprofitar l'ordre lexicogràfic per intercal·lar les tuples generades per cada thread. Per més informació es pot consultar la figura *7.1-9*.

Aleshores, les funcions de conversió simplement han de ser capaces de convertir aquest format de tal manera que es generi un ordre consecutiu. L'algorisme que segueix la conversió al format del treball previ és el definit al pseudocodi *7.2-3 Funció de conversió de keys. Del format nou al format previ.*

Algorisme 7.2-3: Funció de conversió de keys. Del format nou al format previ

```
function newToOldKey(var old)
begin
  array parts[0 .. 1] := split(old, ''t'')
  var key := (parts[0] * N_THREADS) + parts[1]
  return key
end
```

És fàcil veure com es tracta d'una funció molt simple. Bàsicament consisteix en obtenir les diferents parts de la key: key consecutiva i thread, i aplicar una operació de multiplicació i suma per generar la key en format antic. La constant *N_THREADS* indica el nombre total de threads que han participat en la inserció de les dades. Per tant, les tuples amb la mateixa part de key consecutiva són inserides en blocs de *N_THREADS*. D'aquesta manera, l'efecte de la multiplicació és trobar a quin d'aquests blocs pertany la key a convertir. Finalment la suma ajusta a quina tupla dins d'aquest bloc li correspon.

La conversió contrària simplement consisteix en aplicar unes altres operacions aritmètiques. L'algorisme *7.2-4 Funció de conversió de keys. Del format previ al format nou* ho mostra amb més detall.

Algorisme 7.2-4: Funció de conversió de keys. Del format previ al format nou

```
function oldToNewKey(var key)
begin
  var old := key / N_THREADS
  var thread := key % N_THREADS
  var result := concat(old, ''t'', thread)
  return result
end
```

Aleshores, simplement es tracta de dur a terme el procés contrari. Primerament es realitza la divisió amb el nombre de threads per tal d'obtenir de quina key consecutiva es tracta i a continuació la operació mòdul indica el thread en qüestió. Finalment, només resta fer la concatenació d'ambdós resultats per produir la key amb el disseny actual.

Cal mencionar que en aquest algorisme s'ha obviat la funció que hauria d'assignar la key una longitud fixa.

7.2.1.3 Tractament de decimals

Una petita modificació que es va fer respecte el codi original va ser la de construir una forma de tractar amb els decimals que assegurés que els resultats per una mateixa entrada sempre fossin els mateixos. Així doncs, podia ocorre que dues execucions diferents amb la mateixa entrada produïssin resultats substancialment diferents. Això era degut a la manera de Java de gestionar els decimals.

Per tal de determinar quina era la causa d'aquesta divergència, es van fer proves externes amb operacions decimals en Java. Ràpidament es va poder observar com en certes ocasions el resultat generat no era exactament l'esperat. Per exemple, com podia ser que operacions amb nombres amb tres decimals produïssin un resultat on l'n-èssim decimal era 1? Malgrat l'error fos tan petit, amb el volum de dades amb el que es treballa, aquest error causava que el resultat final de les agregacions pogués variar en el segon i tercer decimal.

Aleshores, tot i ser un error de poca importància, es va voler trobar una solució. Aquesta va consistir en fixar el nombre de decimals totals amb el que es vol treballar. Així, la solució és tan simple com multiplicar i dividir per potències de la base decimal. Per exemple, per defecte el nombre de decimals es va fixar a 3, i per tant en aquest cas cal multiplicar i dividir per $10^3 = 1000$. A més, aquestes operacions s'han de fer en moments diferents de l'execució. La figura *7.2-1* mostra millor com encaixa aquest canvi en una execució estàndard de MapReduce.

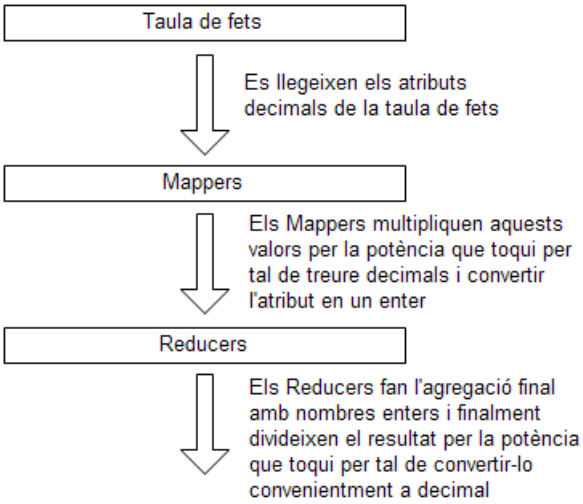


Figura 7.2-1: Fixació de la part decimal

Tal i com es veu a la figura, els Mappers són els encarregats de convertir els decimals en nombre enters tot multiplicant-los per la potència corresponent. D'aquesta manera, aquests nombres enters es converteixen en l'entrada del Reducer i així aquest fa les agregacions amb nombres enters i s'eviten els errors dels nombres decimals. Finalment, només

cal tornar a fer la conversió a nombres decimals tot dividint el resultat per la mateixa potència.

Una petita modificació extra que es va haver de fer, va ser la de modificar el codi de tots els MapReduce per tal que durant el procés d'engegada capturessin les variables globals del sistema. Això ve donat pel fet de que el nombre de decimals amb els que calia dur a terme les operacions es guardava com a variable global, i aleshores aquestes variables globals es reinicialitzaven a l'inici de cada MapReduce. Segurament això és degut a que cada Mapper i cada Reducer és llença en una màquina diferent i per tant tots els valors que no es passin explícitament es perden. Per tant, va caldre modificar els codis per tal que els processos MapReduce inicialitzessin bé aquestes variables.

Degut a què era possible que aquest problema es repetís amb alguna altra variable global de configuració, la solució implementada va ser tan simple com fer arribar al MapReduce la comanda amb la qual s'havia iniciat l'execució, i així fer que el parseig es dugués a terme aquí també. D'aquesta manera, amb el parseig s'aconseguia que tant els Mappers com els Reducers coneguessin la correcta configuració del sistema.

7.2.2 Optimitzacions de memòria

7.2.2.1 Justificació d'aquestes optimitzacions

En els següents apartats es detallen les modificacions que van caldre fer per millorar el consum de memòria produït per les implementacions. Això va ser necessari degut a que en aquest projecte es comença a treballar amb volums de dades relativament grans, i per tant el consum de memòria necessari per processar tot aquest conjunt de dades és molt més elevat que amb SF petits.

Així, per exemple, queries amb un factor de selecció proper a 1 podien ocasionar errors de desbordament de memòria degut a que la màquina virtual de Java era massa petita. En aquests casos, una solució podria haver estat simplement ampliar aquesta màquina virtual, però la realitat és que el consum produït per aquestes queries es feia massa elevat com per simplement ampliar la màquina virtual i aleshores es va fer imprescindible millorar-lo a nivell d'implementació.

Per tal de reduir aquest consum, es van identificar essencialment dos punts en el que el consum es feia excessiu:

1. A l'IFS, en el moment d'enviar els bitstrings construïts cap als Mappers.
2. A l'IRA durant el preprocessament de la taula temporal en certs casos.

Finalment, una última optimització aplicada va ser la d'executar el garbage collector durant determinats moments del sistema. Però aquesta petita optimització ja es detalla en el seu apartat corresponent.

7.2.2.2 Optimització I. Representació del bitstring de l'IFS

La primera d'aquestes optimitzacions va consistir en la forma de transmetre el bitstring utilitzat per l'IFS (apartat *5.4.2 Index Filtered Scan. IFS*) cap als Mappers.

En el codi original, aquesta transmissió no es feia a nivell de bitstring pròpiament dit, sinó que cada Mapper rebia el conjunt sencer de keys a les que calia accedir i el nombre de tuples de la taula de fets. D'aquesta manera, els Mappers s'encarregaven de generar aquest bitstring. És obvi que això requereix un consum de memòria excessiu ja que cada key ocupa per defecte 10 bytes, i per tant amb SF i factors de selecció grans el llistat de keys transmès cap als Mappers podria ser massa elevat.

La solució proposada en aquest apartat consisteix essencialment en crear el bitstring fora dels Mappers i comprimir el resultat en algun format que permeti estalviar memòria. La construcció d'aquest bitstring fora dels Mappers no requereix menció ja que es tracta del mateix algorisme que *5.4-2 Construcció del bitstring*, però desplaçat a un altre punt del codi. El que sí realment és necessari mencionar és la implementació que dóna forma a la manera de comprimir aquest bitstring.

Per tal de dur a terme aquesta optimització el més senzill possible, es va construir una nova classe anomenada **Bitmap**. El nom pot ser una mica ambigu respecte el terme bitstring, i per tant cal aclarir que en realitat són el mateix concepte. La diferència és que en el moment de crear aquesta classe es va respectar la terminologia del projecte anterior, però més endavant es va decidir fer el canvi de nomenclatura ja que un bitmap és en realitat una estructura de dades substancialment diferent.

La firma d'aquesta classe segueix de la següent manera:

Atributs
bitmap : Byte[*]
Mètodes
+ Bitmap(long) : *Construeix un bitstring amb tants bits com indiqui el paràmetre d'entrada i els inicialitza a zero.*
+ Bitmap(String[*], long) : *Construeix un bitstring amb tants bits com indiqui el segon paràmetre d'entrada i activa els bits indicats al primer paràmetre d'entrada. La resta s'inicialitzen a zero.*
+ Bitmap(String) : *Tracta el paràmetre d'entrada com un valor separat per comes i construeix el bitstring a partir d'aquí.*
+ Bitmap(String[*]) : *Construeix un bitstring a partir del paràmetre que rep d'entrada.*
+ toString() : String
+ isActivated(long) : boolean
+ activate(long)

Com es pot veure, aquesta classe conté un únic atribut que correspon al bitstring en si, representat en un array de bytes. Seguidament es mostren els mètodes, on apareixen quatre constructors diferents i tres funcions. Aquestes funcions corresponen a les operacions típiques d'un bitstring:

- **Funció toString().** Funció que converteix el bitstring en un string. Degut a que les dades emeses cap als Mappers han de ser forçosament en format string, és en aquesta funció on es produirà la compressió del bitstring.
- **Funció isActivated().** Funció que comprova si un determinat bit està activat.
- **Funció activate().** Funció que activa un determinat bit. La operació contraria de desactivar un bit és igualment pròpia d'aquest tipus d'estructures, però en aquest cas no s'ha implementat perquè en cap moment de l'execució cal anul·lar un bit.

Els dos últims constructors són els encarregats de convertir un string que representa un bitstring comprimit en un objecte d'aquesta classe.

Presentada l'estructura lògica d'aquesta classe, l'algorisme *7.2-5 Implementació de la classe Bitmap* mostra el pseudocodi que la implementa. La variable bitmap representa l'atribut bitmap d'aquesta classe.

Algorisme 7.2-5: Implementació de la classe Bitmap

```
array bitmap[0 .. A]

function toString()
begin
  if length(bitmap) > 0
  then
    var result := EMPTY_STRING
    for b in bitmap
    do
      var value := asInteger(b)
      result := concat(result, ‘’,’’, value)
      return result
    done
  fi
  return null
end

function isActivated(var id)
begin
  var pos = id >> 3
  var b = 0x01 << (id & 0x07)
  return (bitmap[pos] & b) != 0
end

action activate(var id)
begin
  var pos = id >> 3
  byte b = 0x01 << (id & 0x07)
  bitmap[pos] = bitmap[pos] | b
end
```

Les funcions *activate()* i *isActivated()* segueixen la línia de tractament de bits de les funcions de bitstring original. Les corresponents explicacions es poden trobar a l'apartat *5.4.2 Index Filtered Scan. IFS*. La funció realment nova en aquesta implementació és la funció de compressió *toString()*.

Aquesta funció consisteix en comprimir el bitstring de tal manera que cada byte es tracta com si fos un enter. D'aquesta manera, com que un byte pot representar un enter entre 0 i 255 (o -128 i 127), aleshores el màxim ocupat per cada vuit keys és 3 bytes. Això és degut a que el número 255 conté tres dígits, que al convertir-se en string es transformen en 3 bytes. Així, en 3 bytes com a molt es poden representar vuit keys. Una petita anàlisi d'aquesta millora:

Suposem un conjunt de 10 keys que calen comprimir en el bitstring:

- Sense aquesta petita compressió, el llistat enviat als Mappers hagués estat de deu keys. Si cada key per defecte té llargada 10 bytes, aleshores el cost de comunicació hagués estat el d'enviar 10 keys $*$ 10 bytes = 100 bytes.
- Amb aquest mètode, en el pitjor dels casos cada vuit keys són representades per 3 bytes. Per exemple, les keys *00000007* i *00000000* formen el bitstring

> $10000001_b = 129_d$, que conté tres dígits. Aleshores en aquest cas, calen només sis bytes per representar deu keys en el pitjor dels casos (es necessita un segon byte per representar les últimes dos keys), i per tant el guany és notable.

Una petita millorar respecte aquesta hagués estat representar cada vuit bits com si fossin caràcters ASCII. En aquest cas, el guany hagués estat encara millor, ja que cada vuit keys s'haguessin convertit en un sol byte. Malauradament, el fet de treballar amb Java a vegades complica l'accés a tant baix nivell i diferents problemes de conversions i de tractament de bytes van fer que la solució més viable fos la de manipular-los com si de nombres enters es tractessin.

7.2.2.3 Optimització II. Preprocessament de la taula temporal de l'IRA

Aquesta optimització va venir donada per un fort consum de memòria durant l'execució de l'IRA. Concretament, durant l'etapa de preprocessament de la taula temporal (apartat *5.4.3 Index Random Access. IRA*). Aquesta etapa té per objectiu substituir grups de keys consecutives per rangs. Aleshores, per tal de dur això a terme, l'algorisme de preprocessament manté un buffer amb el conjunt de keys que detecta que formen part d'un rang i que per tant haurien de ser esborrades de la taula a canvi del nou rang. El que ocorria llavors és que amb queries amb factor de selecció massa grans, els rangs identificats tenen cardinalitat molt elevada i en conseqüència els conjunts de keys a esborrar de cada rang eren també massa gran i per tant no hi cabien en un sol buffer a memòria.

Aleshores, la solució aplicada és immediata. Simplement cal limitar el tamany d'aquest buffer fins a un cert nombre de keys i buidar-lo sota les condicions:

a) El buffer s'omple.

b) S'ha arribat al final del conjunt de keys.

L'algorisme resultant s'il·lustra al pseudocodi *7.2-6 Optimització en el preprocessament de la taula temporal*

Algorisme 7.2-6: Optimització en el preprocessament de la taula temporal

```
action helper(var first, var key, array deletes[0 .. D], array puts[0 .. P])
begin
  var it := first
  while it < first + key
  do
    add(deletes, it)
    if (D >= BUFFER_LIMIT)
    then
      delete(TEMPORAL_TABLE, deletes)
      deletes := EMPTY_ARRAY
    fi
  done
  var newKey := concat(first, ''−'', first + keys)
  var put := createPut(newKey)
  addValue(put, ''range'', ''true'')
  add(puts, put)
end

action preprocessTemporalTable()
begin
  var scan := createScan(TEMPORAL_TABLE)
  array puts[0 .. P] := EMPTY_ARRAY
  array deletes[0 .. D] := EMPTY_ARRAY
  var first := −1
  var dist := 0
  for key in launchScan(scan)
  do
    if first = −1
    then
      first := key
    else
      if key = first + dist
      then
        dist := dist + 1
      else
        if dist > 1
        then
          helper(first, key, deletes, puts)
        fi
      fi
    fi
  done
  if dist > 1
  then
    helper(first, key, deletes, puts)
  fi
end
put(TEMPORAL_TABLE, puts)
```

És fàcil veure, doncs, com la solució implementada és de caire més aviat senzill, ja que respecte el seu algorisme original *5.4-4 Preprocessament de la taula temporal* la única diferència és la condició a la funció *helper()* que determina si el buffer està ple o no

i per tant si cal buidar-lo. Per a la resta d'explicacions d'aquest algorisme consultar l'apartat *5.4.3 Index Random Access. IRA*.

7.2.2.4 Optimització III. Memòria entre execucions

Aquesta optimització és la més simple de totes. Ni tan sols és en si una optimització. Simplement, es va forçar per a què el **garbage collector** s'executés abans de començar una nova operació. Així, en aquest cas una operació es pot entendre com una query o la creació d'un índex. D'aquesta manera, el que es busca és esborrar aquelles instàncies d'objectes que han estat generades en operacions anteriors i que per tant ara ja no fan cap falta.

7.2.3 Simplificacions en el codi

Un altre conjunt de modificacions que es van fer respecte el codi original va ser un seguit d'implementacions que intentaven reduir el número de línies de codi. Aquestes modificacions no eren especialment importants respecte els objectius del projecte, però pretenien facilitar les implementacions que sí que eren vitals. El seu objectiu era principalment evitar codi repetit.

Bàsicament el que es va fer va ser tornar a implementar les tasques de MapReduce per tal que es pogués aprofitar el màxim possible els mecanismes que ofereix la programació orientada a objectes i així construir una petita jerarquia de supertipus i subtipus. Concretament, es buscava construir aquesta jerarquia en les classes de Mapper, ja que els Reducers en realitat ja eren compartits per tots els algorismes de consulta.

L'objectiu de tot això era que aleshores, en el moment de fer una modificació important respecte el codi original, s'hagués de fer en una classe o en una altra, però no en totes. D'aquesta manera, aquestes modificacions es podrien fer més ràpid i s'evitarien els riscos de repetició de codi.

Malauradament, el resultat és que no es van poder aplicar aquests canvis a totes les implementacions MapReduce tal i com inicialment s'esperava. Es van poder aplicar només en el cas del procés d'obtenció de valors, ja que en aquest punt es pot generalitzar el procés de MapReduce a simplement l'emissió valors (des del punt de vista de les funcions de map). La forma de realitzar els accessos segons l'algorisme (IRA, IFS o FSS) ho implementa cada subclasse.

La figura *7.2-2* mostra en un petit diagrama el resultat d'aquesta jerarquia. Concretament, es tracta només de generalitzar el procés de Mapper, ja que el codi del Reducer és el mateix per tots.

Així, la classe genèrica BaseMapper implementa les operacions: *keysToEmit()* que és heretada tal qual per les subclasses, i *map()* que aquesta sí que serà reimplementada per tal d'accedir als valors segons cada algorisme. Els codis resultants d'aquesta dues funcions es mostren a l'algorisme *7.2-7 Funcions de la classe general BaseMapper*

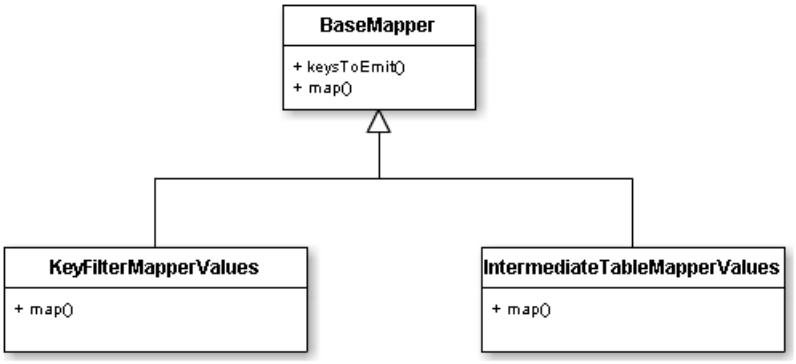


Figura 7.2-2: Jerarquia de classes aplicada per simplificar codi

Algorisme 7.2-7: Funcions de la classe general BaseMapper

```
action keysToEmit(array projection[0 .. P], array groupby[0 .. G], array values[0 .. V])
begin
  var groupby_values := EMPTY_STRING
  for g in groupby
  do
    var v := getValue(value, g)
    groupby_values := concat(groupby_values, ''_'', v)
  done
  for p in projection
  do
    var v := getValue(value, p)
    emit(groupby_values, v)
  done
end

action map(var<key, value> row)
begin
  array projection[0 .. P] := getMapReduceConfiguration(projection'')
  array groupby[0 .. G] := getMapReduceConfiguration(''groupby'')
  keysToEmit(selectionAttributes, groupByAttributes, value);
end
```

Realment, aquest algorisme ja s'ha vist anteriorment. Si recuperem l'algorisme *5.4-1 Full Source Scan* es pot observar com tots dos algorisme són el mateix. La principal diferència és que aquest està fragmentat en dues funcions. Això és així pel comentat anteriorment d'evitar la repetició de codi.

Aleshores, és clar que aquest algorisme correspon íntegrament al codi original de l'FSS.

Per altra banda, el que s'aconsegueix és que les classes que hereten d'aquesta (que són les classes Mappers que implementen els algorismes IFS i IRA) no tinguin repetida aquesta funció *keysToEmit()* en el seu propi codi. En els algorismes originals *5.4-3 Index Filtered Scan* i *5.4-6 Index Random Access - Part II* es veu clarament com en aquests dos casos no existeix una funció *keysToEmit()* específica, sinó que es troba barrejada amb el codi de les funcions de map. Així doncs, el guany de fer aquestes modificacions ha estat pol·lir el codi tal que pogués ser molt més entenedor i per tant més fàcil de dur a terme les implementacions futures.

7.2.4 Proves de validació dels canvis

Per tal de poder anar fent una avaluació de les implementacions addicionals respecte el codi original, es van dur a terme un seguit de petites proves amb l'objectiu de validar aquests canvis. Concretament, la taula de proves es va definir de la mateixa manera que en el projecte anterior.

El conjunt de queries de test es va fer des de zero, ja que no es tenien les queries de test del projecte anterior i per tant no es van poder aprofitar. Aquest conjunt de proves es va dissenyar tal que, donada una dimensió amb diferents nivells d'agregació, s'avaluessin totes les combinacions possibles amb valors existents i no existents a la taula de fets. En altres paraules, tenint en compte la taula de test descrita:

Year	**Month**	**Day**	**Resultat**	**Query**
Si	Si	Si	Si	**SELECT** salary, level **WHERE** Date = 1948%12%24
Si	Si	No	No	**SELECT** salary, level **WHERE** Date = 1948%12%20
Si	No	Si	No	**SELECT** salary, level **WHERE** Date = 1948%8%24
Si	No	No	No	**SELECT** salary, level **WHERE** Date = 1948%8%20
No	Si	Si	No	**SELECT** salary, level **WHERE** Date = 3000%12%24
No	Si	No	No	**SELECT** salary, level **WHERE** Date = 3000%12%20
No	No	Si	No	**SELECT** salary, level **WHERE** Date = 3000%12%24
No	No	No	No	**SELECT** salary, level **WHERE** Date = 3000%12%24

És fàcil, doncs, observar a la taula de queries de test com l'objectiu és avaluar cada un dels nivells d'agregació tot posant valors existents i no existents. Així, s'avalua sobretot el correcte funcionament en cas que existeixin valors en els nivells d'agregació més baixos però no en els nivells més alts. En aquests casos la query hauria de retornar el conjunt buit.

No obstant això, en aquest conjunt de proves s'ha fet servir la dimensió *Date* com a principal avaluador, i això implica que aquesta dimensió té tres nivells d'agregació: *year*, *month* i *any*. A la taula anterior només s'avaluen les queries que accedeixen als tres nivells d'agregació, i per tant cal ampliar aquest conjunt de queries per tal que s'avaluïn queries que accedeixen només a un o dos nivells d'agregació. Aquesta ampliació és la següent:

- Amb dos nivells d'agregació:

Year	**Month**	**Resultat**	**Query**
Si	Si	Si	**SELECT** salary, level **WHERE** Date = 1948%12
Si	No	No	**SELECT** salary, level **WHERE** Date = 1948%8
No	Si	No	**SELECT** salary, level **WHERE** Date = 3000%12
No	No	No	**SELECT** salary, level **WHERE** Date = 3000%8

- Amb un sol nivell d'agregació:

Year	**Resultat**	**Query**
Si	Si	**SELECT** salary, level **WHERE** Date = 1948%
No	No	**SELECT** salary, level **WHERE** Date = 3000%

id	Name	Surname	Gender	Salary	Section	Level	Job	Year	Month	Day	Expenses
1	Luke	Skywalker	male	4000	Direction	5	Director	1948	3	11	300
2	Obi-Wan	Kenobi	male	3500	Direction	4	Administrator	1948	12	24	400
3	Padmé	Amidala	female	3000	Development	5	Programmer	1987	4	5	10
4	Yoda		male	6000	Human-Resources	3	Master	1895	12	12	1900
5	Palpatine	Sidious	male	1500	Human-Resources	3	Senator	1895	11	11	400
6	R2	D2	unknown	15	Development	1	Programmer	2000	1	1	0
7	C	3PO	unknown	15	Development	1	Programmer	2000	1	1	0
8	Qui-Gon	Jinn	male	2000	Development	2	Designer	1960	1	4	200
9	Mace	Windu	male	1000	Development	2	Designer	1960	1	4	300
10	Jar-Jar	Binks	male	0	Direction	1	Public-Relations	1970	4	2	10000

És important notar com en aquest conjunt de queries s'avaluen seleccions que existeixen o no existeixen, però no s'avaluen seleccions contradictòries. Per seleccions contradictòries s'entén seleccions on les dimensions tenen diferents nivells d'agregació i on s'accedeix sempre a valors existents, però contradictoris en el sentit de que no existeix cap tupla amb tots aquests valors en el seus nivells d'agregació.

Per exemple, a la taula de proves es pot veure com existeixen tuples amb $year = 1948$ i tuples amb $month = 4$. Una query que busqués les tuples tal que $year = 1948$ i $month = 4$ estaria fent una selecció contradictòria, ja que malgrat tots dos valors existeixen a la taula de proves, no hi ha cap tupla que contingui tots dos alhora. Per tant, és important crear un segon conjunt de proves que avaluï aquestes situacions.

En altres paraules, dels dos conjunts de proves:

- El primer conjunt de proves es basa en combinar, en cada nivell d'agregació, valors que existeixen amb valors que no existeixen a la taula de proves.
- El segon conjunt garanteix que els valors de la dimensió sempre existeixen, però en aquest cas es tracta de combinar-los juntament amb els nivells d'agregació de forma que no existeixi cap tupla satisfent tots els valors a la vegada.

Per tant, un cop aclarit la diferència entre tots dos conjunts de proves, només falta presentar el segon conjunt. Amb tres nivells d'agregació el conjunt és el següent:

Year	**Month**	**Day**	**Resultat**	**Query**
Si	Si	No	No	**SELECT** salary, level **WHERE** Date = 1948%12%12
Si	No	Si	No	**SELECT** salary, level **WHERE** Date = 1948%11%24
Si	No	No	No	**SELECT** salary, level **WHERE** Date = 1948%11%12
No	Si	Si	No	**SELECT** salary, level **WHERE** Date = 1987%12%24
No	Si	No	No	**SELECT** salary, level **WHERE** Date = 1987%12%12
No	No	Si	No	**SELECT** salary, level **WHERE** Date = 1987%11%24
No	No	No	No	**SELECT** salary, level **WHERE** Date = 1987%11%24

i amb dos nivells d'agregació:

Year	**Month**	**Resultat**	**Query**
Si	No	No	**SELECT** salary, level **WHERE** Date = 1948%11
No	Si	No	**SELECT** salary, level **WHERE** Date = 1897%12
No	No	No	**SELECT** salary, level **WHERE** Date = 1897%11

Aquest segon conjunt no aplica la prova amb un sol nivell d'agregació, ja que no és possible fer combinacions amb un sol element, i per tant qualsevol prova possible amb només el primer nivell d'agregació és avaluar si el seu valor existeix o no existeix a la taula de proves, i això equival a la avaluada al primer conjunt.

7.3 Etapa III: Configuració i automatització de les proves

7.3.1 Descripció de l'entorn de treball. Què cal fer?

Un cop havent dut a terme totes les implementacions i modificacions explicades als apartats *7.1 Etapa I: Població de l'HBase. Generador* i *7.2 Etapa II: Construcció dels cubs de dades. MapReduce*, va caldre desenvolupar un seguit de mecanismes i de configuracions

que permetessin l'execució del sistema en el seu sentit més global (insercions, creació dels índexs i llançament de les consultes) d'una forma aliena a l'usuari.

Això és així perquè l'execució de tot aquest sistema pot sobrepassar un dia d'execució i per tant cal que s'executi de la forma més automàtica possible. A més, l'accés al clúster del DAC s'ha de fer via consola i per tant no hi ha cap mena d'opció de treballar visualment amb eines d'interfície gràfica, com per exemple les webs de monitorització de les tecnologies emprades.

Com que aquest treball és la continuació d'un de previ, s'ha intentat heretar el màxim possible de mecanismes que permeten dur a terme aquestes execucions amb el mínim possible d'interacció humana. Aquests mecanismes són essencialment dos:

1. Un conjunt de scripts que configuren les tecnologies emprades en cada un dels nodes i duen a terme l'execució del sistema.

2. Un conjunt d'arxius de configuració de les tecnologies emprades.

Tal i com s'ha introduït a l'inici d'aquest apartat, l'existència d'aquesta etapa ve principalment justificada per l'automatització de les proves. A més, no hi ha accés a interfícies gràfiques i per tant cal trobar formes alternatives de generar les estadístiques i la informació que en principi són només accessibles a partir d'aquestes interfícies.

Així doncs, en línies generals, el que s'espera d'aquesta etapa del projecte és implementar un conjunt de mecanismes que:

- Duguin a terme les següents execucions sense interacció de l'usuari:
  - Inserció de les dades.
  - Construcció dels índexs.
  - Execució de les consultes.
- Malgrat aquestes execucions s'hagin de produir sense interacció de l'usuari, aquest ha de tenir accés ràpid per saber en quin estat es troben.
- En totes aquestes execucions, cal que els mecanismes permetin obtenir el màxim d'informació per tal de recolzar l'avaluació final del resultat de les proves que finalment es duguin a terme.
- Degut a la llarga durada d'aquestes execucions, també s'espera minimitzar el risc a l'aparició d'errors que puguin obligar a rellançar les execucions des del seu inici, i que en cas d'aparició d'errors, aquest mecanismes puguin actuar mínimament per intentar solventar-los automàticament.
- Permetin estalviar execucions innecessàries. Per exemple, si una mateixa inserció pot servir per avaluar diferents paràmetres o diferents proves, aquests mecanismes han d'assegurar que les dades inserides poden ser utilitzades per altres execucions.

A més, al clúster del DAC no s'hi troba una versió de Hadoop i molt menys de HBase ja prèviament instal·lada. Per tant, aquest conjunt de scripts tenen també l'objectiu de ser capaços de gestionar aquest conjunt de tecnologies. Això implica la configuració d'aquestes tecnologies, la seva instal·lació en els nodes i l'arrancada i aturada de les mateixes.

Cal recalcar que, com que no hi ha cap instal·lació prèvia d'aquestes tecnologies, la forma de fer-ho consisteix en moure cada una d'aquestes a cada una de les màquines a

l'inici de l'execució. Això implica treballar a molt baix nivell, amb totes les dificultats addicionals que això implica respecte treballar a més alt nivell.

Afortunadament, el clúster del DAC ofereix un espai de disc compartit per tots els nodes del clúster. Això permet simplificar en certa manera tota la càrrega de feina que s'espera d'aquests mecanismes d'automatització. Per una banda, perquè permet tenir les tecnologies en un únic espai de disc a partir del qual, poden ser mogudes fàcilment cap als nodes en qüestió, i per l'altra perquè permet definir plantilles de configuració comunes que simplifiquin el procés de configuració d'aquestes tecnologies. A més, pels objectius de log, aquest espai compartit també permet mantenir aquests mateixos logs d'una forma més centralitzada.

Així doncs, en els apartats següents s'entra en més detall en cada un d'aquests mecanismes i de les ampliacions que han calgut fer. Per tal de facilitar al lector aquestes explicacions, a continuació es defineix un petit vocabulari:

- **Treball.** Un treball és cada una de les execucions que es llancen contra el clúster del DAC. Un treball està acotat a una cua d'usuaris que a la vegada està assignada a un temps màxim d'execució i a un nombre de processadors com a màxim predeterminat.
- **Sistema i clúster (Hadoop/HBase).** Per sistema s'entén la unió de totes les tecnologies emprades funcionant al llarg d'un conjunt determinat de màquines, tot formant un clúster HBase, que alhora necessita un clúster Hadoop.

7.3.2 Automatització de les proves. Scripts

7.3.2.1 Implementació general dels scripts. Diagrama

Per a la realització d'aquesta fase del projecte, es va heretar un conjunt de scripts que permetien una automatització mínima de les execucions. Malgrat aquesta automatització mínima, la automatització final que es necessitava per a aquest projecte havia de ser més complexa degut a la exhaustivitat de les execucions. Principalment a causa dels volums de dades amb els què es treballa, que fan que aquestes execucions siguin costoses. Això va implicar canvis en tota aquesta estructura de scripts.

La figura *7.3-1* mostra en un diagrama el flux final d'execució de tots aquests scripts.

Aquest conjunt de scripts es pot classificar en dos grups més generals:

1. Scripts per a la inicialització i aturada del clúster.
2. Scripts que duen a terme l'execució.

A continuació es detalla cada un dels scripts que donen joc a la inicialització i aturada del clúster.

- **envConfig.sh** és el punt d'entrada. S'encarrega simplement de llençar el treball segons els paràmentres:
  1. El tipus d'execució: *full, insert, dimension, scan dimension o query* (veure més abaix).
  2. L'SF de la execució.
  3. Com s'estructuraran les families.
- **startUp.sh** gestiona cada un dels passos que es duran a terme durant el treball: configuració de les tecnologies, configuració dels nodes, llançament de l'execució i aturada del sistema.

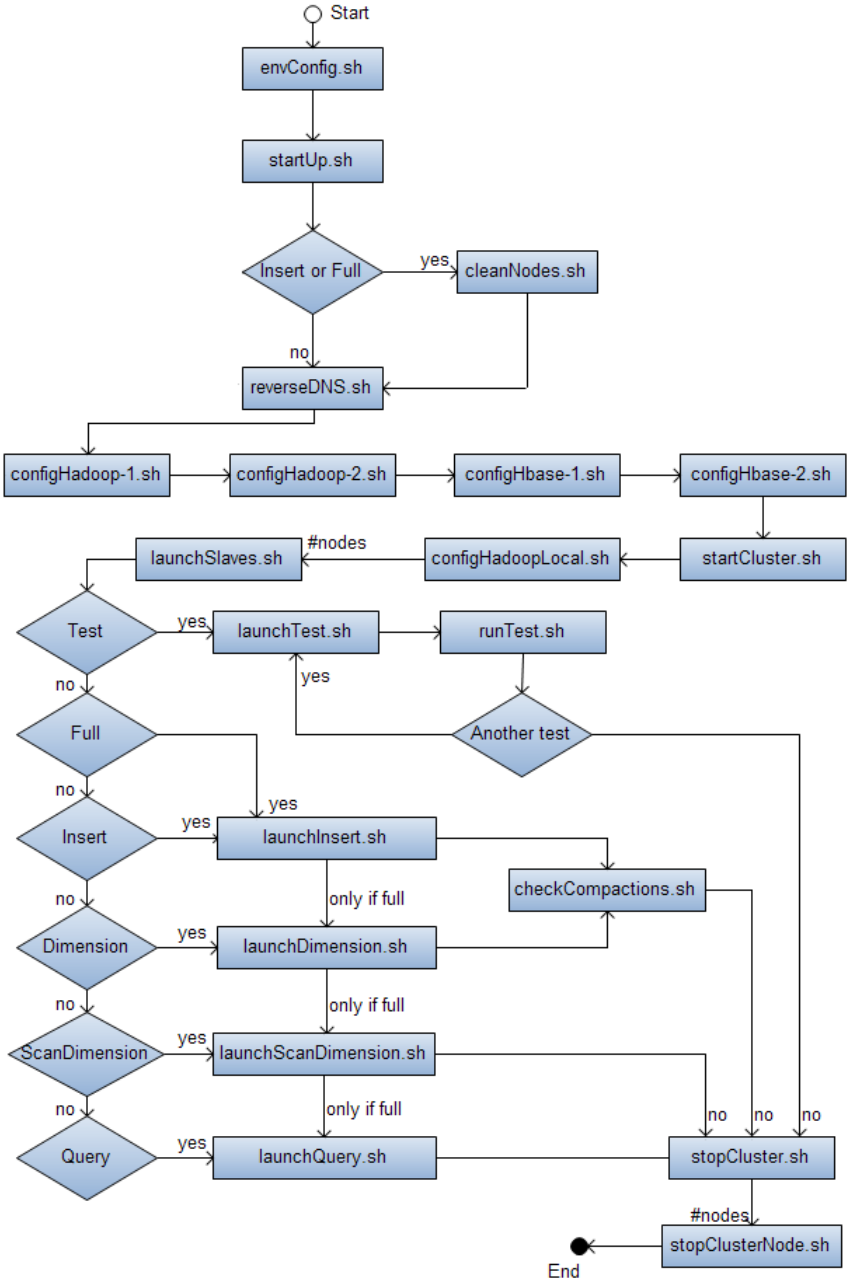


Figura 7.3-1: Diagrama de flux dels scripts durant una execució

- **cleanNodes.sh** esborra qualsevol possible dada dels nodes que hagi resultat d'algun treball anterior. Per aquest motiu és fa només en cas de que es tracti d'una inserció.

- **reverseDNS.sh** construeix un fitxer de text on apareixen cada un dels nodes assignats al treball juntament amb la seva direcció IP. Aquest punt es troba més desenvolupat a l'apartat *7.3.2.2 Reverse DNS lookup*.

- **configHadoop-1.sh** i **configHadoop-2.sh** són els scripts que determinen la configuració de Hadoop i el mouen cap als nodes assignats al treball. Aquest procés està trencat en dos scripts degut a que cada un d'ells rep com a paràmetres diferents variables: el primer rep les variables de configuració de l'execució (per exemple el volum de dades) i el segon rep el llistat de nodes.

  Per a la configuració d'aquesta tecnologia, s'han definit uns arxius de configuració com a plantilles i a partir de les quals, mitjançant expressions regulars, es fan els canvis convenientment i s'assignen els fitxers resultants d'aquests canvis com a fitxers de configuració reals.

- **configHbase-1.sh** i **configHbase-2.sh** apliquen les mateixes explicacions que els scripts de configuració de Hadoop, però per a l'HBase.

- **startCluster.sh** és l'script que s'encarrega d'inicialitzar el clúster en si. Arranca l'HDFS, el MapReduce i finalment l'HBase. Per a fer-ho, es recolza en els dos scripts següents:

  - **configHadoopLocal.sh** que crea els directoris necessaris per a iniciar el clúster.
  - **launchSlaves.sh** que executa en els nodes esclaus cada un dels processos que han de donar lloc al clúster en funcionament.

  Finalment, s'arrenquen els processos en els nodes màster.

- **stopCluster.sh** s'encarrega d'aturar el sistema, i ho fa mitjançant l'script:

  - **stopClusterNode.sh** que atura el sistema per a un node concret.

  A l'apartat *7.3.2.2 Aturada del sistema* es detalla amb més exactitud el perquè cal aturar el sistema de forma tan explícita i no és suficientment amb simplement matar els processos bruscament.

Per altra banda, existeixen els scripts que implementen cada una de les possibles execucions que es poden dur a terme durant un treball. Aquestes operacions són:

- **Test:** Aquesta opció senzillament llença el conjunt de proves de validació descrit a l'apartat *7.2.4 Proves de validació dels canvis*.

- **Insert:** Aquesta operació, tal i com el seu nom indica, fa les corresponents insercions per un SF i una estructura de families donats.

- **Dimension:** Construeix els índexs.

- **ScanDimension:** Llegeix els índexs construïts per tal de poder treure certes estadístiques com el factors de selecció, rangs de keys...

- **Query:** Llança el conjunt de queries i obté els temps d'execució.

- **Full:** Aquesta operació intenta aprofitar un sol treball per dur a terme tota la bateria de proves per a un determinat paràmetre: fa les insercions, construeix els índexs, els llegeix i llança el conjunt de queries.

En el diagrama d'execució mostrat a la figura *7.3-1* es fàcil apreciar com l'execució d'una operació d'inserció o de construcció dels índexs és seguida per un script anomenat **checkCompactions.sh**, mentre que la resta d'operacions simplement finalitzen la seva execució. La justificació de l'existència d'aquest script es troba a l'apartat *7.3.2.2 Compactacions*, ja que es tracta d'un script clau en garantir la correcta distribució de les dades al llarg de tots els nodes.

A més, un altre requisit important era que aquests scripts publiquessin un sistema de logs per tal de poder detectar possibles fallades. Aquest conjunt de logs va ser molt fàcil d'implementar, ja que va consistir en simplement redirigir la sortida de cada procés i de cada script cap a un determinat fitxer. Així, es poden identificar dos conjunts de logs:

- **Logs de *tracking*:** Aquest conjunt de logs permet esbrinar en tot moment en quina fase es troba el treball, així com permet identificar fàcilment quins nodes han estat assignats com a esclaus i quins com a màsters.
- **Logs de sistema:** Aquests logs permeten monitoritzar cada una de les operacions dutes a termes en el sistema per tal d'identificar possible fallades. Es tracta bàsicament dels logs propis de les tecnologies emprades.

7.3.2.2 Implementacions importants

En aquest apartat es ressalten aquells aspectes de la implementació d'aquesta fase del projecte que mereixen ser desenvolupats amb explicacions més detallades ja que constitueixen un punt clau en el correcte funcionament de tota l'estructura de scripts vista a l'apartat anterior.

Reverse DNS lookup

Per a la bona execució de les proves, va ser important trobar una manera de dur a terme el procés *reverse DNS lookup*[37], en principi necessari per a les execucions MapReduce. Així és com es recomana a la documentació oficial i de fet, en el clúster local, va ser necessària la incorporació d'un servidor que dugués a terme aquest procés. Curiosament, en el clúster del DAC apareixien els mateixos errors relacionats amb una mala resolució DNS tot i que les execucions MapReduce funcionaven correctament. No obstant això, per tal de curar-se en salut i garantir que no es produirien errors futurs degut a aquest procés, es va buscar una solució concreta. Aquesta va consistir en compilar una nova versió de HBase que permetés dur a terme aquest procés mitjançant un fitxer de text. Així doncs, amb aquesta nova versió, l'HBase permet definir als arxius de configuració una nova variable:

<property>
<name>hbase.mydns.file</name>
<value>Ruta al fitxer DNS</value>
</property>

I aquest fitxer ha de tenir un aspecte semblant al següent:

[37] Procés de descobrir el nom d'un servidor donada una IP.

172.18.1.58	arvei-58.ac.upc.edu.
172.18.1.57	arvei-57.ac.upc.edu.
172.18.1.25	arvei-25.ac.upc.edu.

És important recalcar el punt final després del nom de cada servidor, ja que en termes de DNS indica que no hi ha cap subdomini a continuació.

Per altra banda, les modificacions realitzades al codi font de l'HBase segueixen de la següent manera. Afortunadament per a aquests canvis, HBase fa el procés de *reverse DNS lookup* mitjançant una cache. Aleshores, els canvis que es van fer van consistir essencialment en modificar la inicialització d'aquesta cache de forma que també s'afegissin les màquines descrites en el fitxer de text especificat.

Aquests canvis es van haver de realitzar al fitxer *TableInputFormatBase.java* del paquet *org.apache.hadoop.hbase.mapreduce*, i l'algorisme és el següent:

Algorisme 7.3-1: Algorisme de *reverse DNS lookup* des de fitxer de text

```
var reverseDNSCacheMap // Cache with all reverse DNS lookup nodes
action setMyDNS(var file)
begin
  while !end_of_file(file)
  do
    var line := read(file) // Reads line by line
    array aux[0 .. 1] := split(line, TABULATOR)
    var ip := aux[0]
    var hostName := aux[1]
    if !contains(reverseDNSCacheMap, ip)
    then
      add(reverseDNSCacheMap, ip, hostName)
    fi
  done
end
```

L'últim canvi va ser afegir aquesta funció al procés d'inicialització de la cache.

Compactacions

Un punt clau en tota aquesta fase del projecte va ser aconseguir la correcte distribució de les dades al llarg de totes les màquines. Aquest apartat segueix en certa manera la línia ja descrita a l'apartat *7.1.7.2 Concentració de dades*, on es discuteixen les raons d'una mala distribució de les dades i la solució aplicada a nivell de procés d'inserció.

No obstant això, un cop el procés d'inserció finalitza, cal donar un marge de temps per tal de permetre que les últimes compactacions es duguin a temps i per tant es moguin les dades als seus respectius nodes i s'esborrin storefiles punters (veure apartat *4.2.2 Fragmentació horitzontal: Autosharding*). Aquest és l'objectiu d'aquest apartat. Concretament, de l'script **checkCompactions.sh**.

La qüestió que emergeix aleshores és: quant de temps cal donar de marge a les compactacions? La veritat és que es fa difícil trobar una resposta única, ja que les variables d'entorn són infinites: volum de dades, nombre i càrrega d'aquests nodes, paràmetres de configuració, etc. Afortunadament, la interfície web de l'HBase permet saber si en un moment donat s'està produint una compactació en una taula determinada.

Aleshores, fent servir la línia de comandes (cal recordar que en el clúster del DAC no hi ha accés a interfícies gràfiques) es va poder accedir a aquesta interfície web i obtenir l'estat de les compactacions en un simple string. A partir d'aquí, l'algorisme que segueix

és molt senzill: *mentre hi hagi alguna compactació no finalitzar*. Aquest algorisme és el descrit a continuació:

Algorisme 7.3-2: Algorisme de resolució de compactacions

```
action checkCompactions()
begin
  var compacting := getCurrentCompaction()
  while compacting != ''NONE''
  do
    sleep 60
    compacting := getCurrentCompaction()
  done
end
```

On *getCurrentCompaction()* correspon a la funció que obté si hi ha alguna compactació pendent. Per a més detall, aquesta funció correspon concretament a les comandes:

- Obtenció de la interfície web en un arxiu *tmp*:

```
wget http://$HOSTNAME:60010/table.jsp?name=$TABLE -O tmp
```

- Obtenció de la compactació a partir del parseig de l'arxiu *tmp*:

```
cat tmp | grep Compaction -A 1 | grep -v Compaction | sed 's/\///' | awk -
F"<td>" '{print $2}'
```

Aturada del sistema

El procés d'aturada del sistema va ser un punt molt important ja que una bona aturada garantia que les dades quedessin ben materialitzades a disc i per tant puguin ser reutilitzades entre execucions diferents.

La dificultat va romandre en el fet que el clúster del DAC limita els recursos per màquina que un usuari pot fer servir si està fora d'un treball. Si l'usuari està en un treball, els recursos són il·limitats, però cal treballar amb unes comandes especials. Per exemple, la comanda *ssh* que permet llançar instruccions a nodes remots és converteix en *qrsh* quan s'executa dins d'un treball.

Llavors, els scripts d'arrancada de Hadoop no estan pensats per aquestes situacions, i els processos de comunicació entre nodes són del tipus del que el clúster del DAC limita els recursos. Per aquest motiu, per posar en marxa el clúster Hadoop/HBase s'ha de fer d'una forma especial, i per tant, apagar-lo també. Realment engegar el clúster és molt fàcil, ja que és suficient amb fer una petita simulació d'aquest procés tot modificant cada una de les comandes de "recursos limitats" i convertir-les en comandes de "recursos il·limitats".

En el procés contrari, la diferència és bàsicament el fet que apagar un sistema implica saber quins són els processos a matar. Originalment, els scripts que es van rebre del treball anterior ja posaven en marxa correctament el sistema en si, però l'aturada era més aviat inexistent: un cop les execucions finalitzaven, ho feia també el treball i per tant el propi clúster del DAC alliberava els recursos utilitzats i el sistema s'aturava bruscament. Per tant, en el cas d'aquest projecte, es van haver de fer les modificacions necessàries per a simular l'aturada. Tal i com s'ha dit anteriorment, solucionar aquesta qüestió implica simplement tenir constància de quins processos cal matar. Després és suficient amb tornar a agafar les comandes de Hadoop que aturen el sistema i convertir-les en comandes de "recursos il·limitats".

Per a dur això a terme, la implementació feta comença en el procés d'arrancada, ja que és en aquest moment quan hi ha constància dels processos que s'estan creant. Durant aquesta etapa es crea una nova carpeta *pid* en tots els nodes del treball, i que conté el número de procés que cal matar per cada una de les tecnologies emprades. Finalment, el pas final no consisteix en res més que aturar aquests processos.

D'aquesta manera, es va aconseguir simular el procés d'aturada del sistema correctament i per tant garantir que les dades perduressin entre treballs diferents.

7.3.3 Configuració de paràmetres

Tal i com es comenta a l'apartat *7.3.1 Descripció de l'entorn de treball. Què cal fer?*, un dels objectius de la tercera fase de la implementació va consistir en configurar el sistema per tal d'acabar de polir algunes errors que van aparèixer.

Cal comentar que molts d'aquests errors van ser deguts als constants canvis de màquines en les que s'executaven les proves. Totes aquestes màquines eren màquines del clúster del DAC, però per tal de minimitzar el temps perdut durant aquesta fase i per tant poder començar l'execució dels experiments el més aviat possible, es van anar canviant de nodes repetidament tot escollint aquells nodes disponibles.

Així doncs, al llarg d'aquest apartat es presenten cada una d'aquestes dificultats i la solució aplicada.

Cal mencionar que els següents paràmetres son els modificats en aquest projecte, però que en el projecte anterior se'n va configurar d'altres que s'han aprofitar també per a aquest mateix projecte. Aquest paràmetres ja configurats del projecte anterior són (de forma resumida):

- En el fitxer **core-site.xml**:
  - Definició del màster de l'HDFS *(fs.default.name)*.
  - Directoris físics on s'emmagatzemaran les dades *(hadoop.tmp.dir)*.
- En el fitxer **mapred-site.xml**:
  - Definició del màster del MapReduce *(mapred.job.tracker)*.
  - Directoris locals *(mapred.local.dir)*.
  - Directoris de sistema *(mapred.system.dir)*.
  - Directoris on s'emmagatzemaran les dades temporals *(mapred.temp.dir)*.
  - Temps màxim per a una tasca MapReduce *(mapred.task.timeout)*.
- En el fitxer **hbase-site.xml**:
  - Directori arrel de l'HBase *(hbase.rootdir)*.
  - Configuració del clúster HBase com a distribuit *(hbase.cluster.distributed)*.
  - Definició de les màquines del ZooKeeper *(hbase.zookeeper.quorum)*.
  - Directori del ZooKeeper *(hbase.zookeeper.property.dataDir)*.

Llindar del safemode

Un dels propòsits era aconseguir que les dades inserides durant una execució fossin també accessibles per a consultes que poguessin formar part d'un altre treball. Recordem que el treball en el clúster del DAC implica un seguit de requisits de temps d'execució i de concurrència d'usuaris i per tant era imprescindible aprofitar les dades d'una inserció per fer vàries proves.

Llançar un nou treball en el clúster implica tornar a arrancar el sistema en la seva totalitat. Malauradament, l'HDFS té un petit sistema de seguretat que no li permet arrencar en cas de que es compleixin certes condicions. Quan això succeeix, se'n diu que l'HDFS està en un estat de **safemode**. Es pot trobar la definició de safemode a l'apartat *4.2.1 HDFS*.

Les possibles solucions eren bàsicament dues:

1. Esperar a que l'HDFS estigués totalment engegat (tant el master com tots els DataNodes) i aleshores executar la següent comanda desde la consola de Hadoop:

   hadoop dfsadmin -safemode leave

   És fàcil veure com aquesta comanda senzillament força a l'HDFS a deixar l'estat safemode.

2. L'altra proposta era rebaixar el llindar de replicació que activa el safemode i així fer que es produeixi només durant el temps d'arrancada, i un cop arrancat tot el sistema es desactivi automàticament. Per defecte aquest llindar pren el valor de que el 99.9% dels blocs han d'estar replicats correctament.

La solució aplicada va ser la segona. Especialment per la facilitat d'implementació: només calia modificar un paràmetre de l'HDFS. L'altra solució, per contra, també era molt fàcil d'executar la comanda, però tenia la pega de que s'havia de fer un cop el sistema estigués completament engegat i això ja era més difícil d'implementar.

Es va configurar el paràmetre *dfs.safemode.threshold.pct* amb el corresponent valor de 0%.

Aquesta modificació s'ha de fer al fitxer de configuració **hdfs-site.xml**.

Màxim de fitxers servits a la vegada

Un altre concepte de l'HDFS, també definit a l'apartat *4.2.1 HDFS*, són els **xcievers**. Els xcievers indiquen el nombre màxim de fitxers que un DataNode pot servir en un moment donat. Bàsicament aquest problema era degut a que el volum de dades amb el que es treballa necessitava ampliar aquest paràmetre. Aleshores la solució aplicada va ser tan senzilla com configurar el paràmetre *dfs.datanode.max.xcievers* al valor 4096.

Aquesta modificació s'ha de fer al fitxer de configuració **hdfs-site.xml**.

Algunes de les excepcions que poden indicar un problema degut a aquesta causa són, a nivell d'HDFS i a nivell d'HBase respectivament:

- *INFO hdfs.DFSClient: Could not obtain block* blk_X_Y from any node: java.io.IOException*
- *UnknownScannerException*

Com es pot veure, la excepció de l'HDFS dona més detall que no pas la de l'HBase. Això és degut a que la primera excepció forma part dels logs que registren l'activitat de l'HDFS (perquè en realitat es tracta d'un problema a nivell de HDFS), mentre que la segona és un simple error que apareix per la sortida estàndard d'una execució qualsevol de l'HBase.

Port dels servidors de l'HDFS

Una petita configuració que val caldre fer va ser la de canviar el port de comunicació sobre el que treballa l'HDFS. Això va ser degut a què amb la concurrència d'usuaris del clúster del DAC, era fàcil que un altre usuari fent servir Hadoop estigués bloquejant aquest mateix port i per tant el sistema no arranqués correctament.

Aleshores, el canvi va ser ràpid de fer i va consistir en modificar l'atribut *dfs.datanode.address* amb el valor 0.0.0.0:51069 per tal que la comunicació es fes a través del prot 51000.

Aquesta modificació s'ha de fer al fitxer de configuració **hdfs-site.xml**.

Espai de la màquina virtual de Java

Aquesta modificació es va fer degut al volum de dades gran amb el que es treballa. En certes ocasions, durant l'execució dels algorismes, cal tenir una bona part d'aquest volum de dades a memòria. Normalment ja es càrrega comprimit, però al ser un volum gran es necessita també ampliar la capacitat de memòria de la màquina virtual (que per defecte són només 128MB).

Així doncs, es va procedir a modificar el paràmetre *mapred.child.java.opts* per a què prengués el valor Xmx4096m, que permet que la màquina virtual creixi fins a un tamany màxim de 4GB.

Aquesta modificació s'ha de fer al fitxer de configuració **mapred-site.xml**.

Espai de la configuració del MapReduce

Un altre paràmetre a tenir en compte durant les execucions és el tamany de l'objecte que emmagatzema la configuració d'una execució en MapReduce. Per defecte, aquest paràmetre pren el valor de 10MB de màxim.

En el cas d'aquest projecte, aquest objecte es situa per sota dels 100MB d'espai (entre 80 i 90MB, depenent de l'execució) i per tant va caldre ampliar aquest paràmetre. El paràmetre modificat és *mapred.user.jobconf.limit* i es va ampliar a 104857600 bytes (100 MB).

Aquesta modificació s'ha de fer al fitxer de configuració **mapred-site.xml**.

Un exemple de missatge degut a aquest fet és:

> *org.apache.hadoop.ipc.RemoteException: java.io.IOException: java.io.IOException: Exceeded max jobconf size: 5445900 limit: 5242880*

Temps de timeout de lectura de les dades

Finalment, a la configuració de l'HBase es va haver d'ampliar el temps que té un objecte Scanner per a fer la lectura de les dades que li han estat assignades. De totes maneres, els errors produïts per aquest timeout ocorrien molt puntualment. Eren deguts principalment a la concurrència d'usuaris que provocava un excés en l'ús dels recursos de les màquines i per tant alentien les lectures.

Un exemple d'excepció que pot indicar un error d'aquest tipus és:

> *org.apache.hadoop.hbase.client.ScannerTimeoutException: 88557ms passed since the last invocation, timeout is currently set to 60000.*

El paràmetre modificat és simplement *hbase.rpc.timeout* i es va modificar amb el valor 600000 (segons).

Aquesta modificació s'ha de fer al fitxer de configuració **hbase-site.xml**.

8 Resultats

8.1 Decisió i planificació dels paràmetres

El conjunt de proves a realitzar correspon al disseny de la base de dades juntament amb la tipologia de les queries. El problema que planteja aquest conjunt de proves és la gran quantitat de paràmetres a tenir en compte, que fa inviable provar-los tots. Concretament, el llistat de paràmetres inicial és:

- Tipologia de les queries:
  - Cardinalitat de la projecció.
  - Cardinalitat de l'agrupació.
  - Cardinalitat de la selecció.
  - Factor de selecció.
- Disseny de la base de dades:
  - Replicació HDFS.
  - Estratègia de fragmentació vertical (families).
  - Nombre de RegionServers.
  - Nombre de mappers.
  - Nombre de reducers.
  - Compressió.

i on cada un d'aquests paràmetres estan pensats per valorar-se entre amb tres valors diferents. A més, cada un d'aquests paràmetres ha de valorar-se sota altres paràmetres com ara el volum de la taula o els algorismes. Així, el nombre total d'execucions que cal fer és el producte dels factors:

- Nombre de dissenys de la base de dades.
- Nombre de SF.
- Nombre de queries.
- Nombre d'algorismes.
- Nombre de repeticions.

I clarament cinc multiplicacions produeixen un nombre d'execucions massa elevat. Un petit exemple de càlcul podria ser:

Factor	Valor	Justificació
Nombre de dissenys de la base de dades	729	6 paràmetres amb 3 valors diferents cada un. Això és un total de 3^6 combinacions possibles.
Nombre de SF	5	Les proves d'inserció (apartat *7.1.7.2 Clúster del DAC*) determinen que es poden inserir SF entre 1 i 5.
Nombre de queries	31	Les mateixes que es van dissenyar al projecte anterior.
Nombre d'algorismes	5	FSS, IFS *Single/Multiple Level Index*, IRA *Single/Multiple Level Index* (apartat *5.2 Creació dels índexs*).
Nombre de repeticions	5	Un nombre acceptable per evitar pics d'execució i outliers.
Temps d'execució mig d'una query	10 min	Així ho mostraven les primeres proves que es van fer.
TOTAL	53,7 anys	

On clarament es veu que formular els experiments d'aquesta manera es fa completament inviable degut al temps necessari. Aleshores, com que no és possible treure cap element de l'equació, la decisió presa consisteix en simplificar els possibles valors de cada factor. Degut a la discussió que envolta cada una d'aquestes simplificacions, les seves corresponents justificacions es troben organitzades al llarg dels següents subapartats.

Dissenys de la base de dades

Per a reduir el conjunt total de paràmetres a provar, es va intentar identificar quins paràmetres es podien justificar com a independents i quins no. D'aquesta manera, l'objectiu era fer agrupacions de paràmetres que reduïssin el nombre de combinacions entre els paràmetres de cada grup, i aleshores fer proves entre grups. Els grups resultants van ser:

1. Replicació:
    - Replicació HDFS.
2. Nombre de regions:
    - Nombre de RegionServers.
    - Estructura de les families.
    - Compressió.

Per tal d'entendre la justificació que hi ha darrere d'aquesta agrupació, cal recordar les polítiques de region split esmentades a l'apartat *4.2.2 Fragmentació horitzontal: Autosharding*. Malgrat l'HBase ofereixi més polítiques de region split i doni opció a l'administrador d'implementar la seva pròpia, en aquest projecte només s'ha fet servir la política per defecte.

Així doncs, en aquesta política una region qualsevol fa split quan el volum ocupat per un dels seus storefiles (qualsevol) creix per sobre d'un cert llindar. Concretament, el càlcul d'aquest llindar ve donat per la formula:

$$storefile.max.size = \\ min(R^2 hbase.hregion.memstore.flush.size, hbase.hregion.max.filesize)$$

On:

- $hbase.hregion.memstore.flush.size$ indica el tamany dels memstores.
- $hbase.hregion.max.filesize$ indica un volum màxim constant.

La variable R indica el nombre de regions que hi ha al RegionServer de la region que vol fer split. Concretament, el comportament d'aquesta política és fer créixer el llindar que indica un region split fins superar el límit constant indicat al segon paràmetre de la funció mínim.

D'aquesta manera, les següents explicacions justifiquen els motius pels quals l'agrupació triada, replicació per una banda i nombre de regions per l'altra, fa referència a la distinció dels paràmetres que poden afectar al nombre final de regions o no.

- **Nombre de RegionServers.** El nombre de RegionServer influeix en el valor R de l'equació. Amb un nombre petit de RegionServers, les regions estan molt concentrades en tan sols uns pocs RegionServers i per tant el valor de la R per a un RegionServer és gran. Aleshores, el tamany d'un storefile per a causar un region split ha de ser més gran i en conseqüència el nombre de regions final és molt més petit. En cas de tenir molts RegionServers, les regions estan molt més distribuïdes al llarg de tots aquests i per tant el valor de la R per a un RegionServer és molt més petit i el nombre final de regions és molt més gran. En altres paraules, suposem una taula HBase amb cinc regions ja creades:
  - Si RegionServers = 5 aleshores es pot assumir que les cinc regions estaran ben distribuïdes i hi haurà una region a cada RegionServer, per tant $R = 1$ i el proper region split succeirà quan:
    $$storefile.max.size = 1^2 hbase.hregion.memstore.flush.size$$
  - Si RegionServers = 1 aleshores les cinc regions estaran concentrades en un sol RegionServer, per tant $R = 5$ i el proper region split succeirà quan:
    $$storefile.max.size = 5^2 hbase.hregion.memstore.flush.size$$
- **Estructura de les families.** Com que els storefiles contenen dades per a una mateixa family, és fàcil justificar que amb una fragmentació vertical molt forta, els storefiles creixen molt lentament i per tant es produeixen pocs region splits. En canvi, amb una fragmentació vertical molt fluixa, els atributs estan concentrats en un conjunt més reduït de families i aleshores els storefiles creixen més ràpidament de volum i forcen a que hi hagi una quantitat major de region splits. De fet, la documentació oficial de l'HBase no recomana fer un disseny amb més de tres families degut a la pèrdua de paral·lelisme. La figura *8.1-1* mostra aquest cas.
- **Compressió.** La compressió simplement afecta al volum dels storefiles. Amb una compressió forta els storefiles ocupen menys espai i per tant es produeixen menys region splits. Amb una compressió més fluixa l'efecte és el contrari. La figura *8.1-2* ho mostra amb més detall.

Com que la replicació no afecta a l'espai ocupat pels storefiles ni a cap element de l'equació, es va decidir agrupar-la a part. El nombre de mappers i el nombre de reducers es va decidir no incloure'ls finalment ja que:

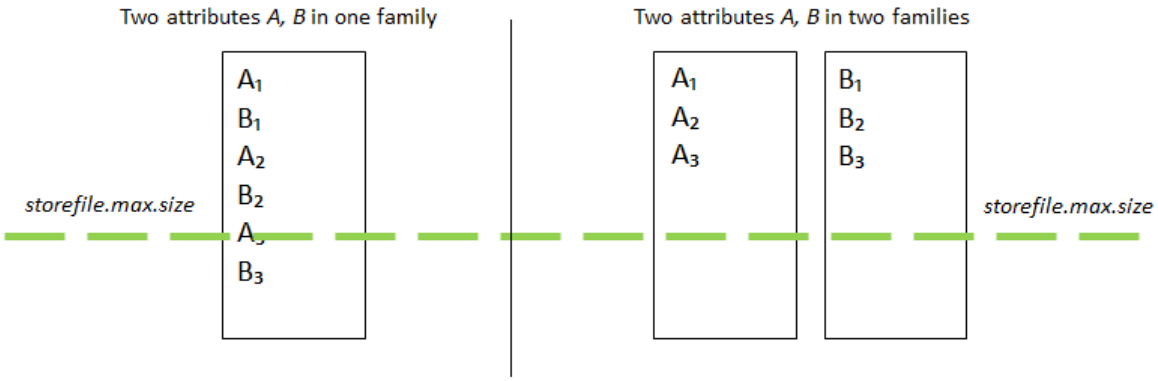


Figura 8.1-1: Implicació de les families en els region splits

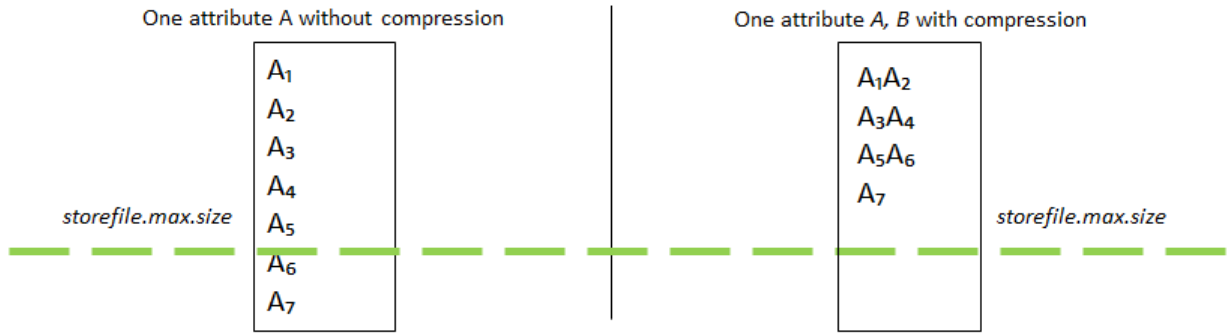


Figura 8.1-2: Implicació de la compressió en els region splits

1. No podien formar part de cap dels grups vists anteriorment ja que en realitat es tracten de paràmetres més propis del temps d'execució d'una query que no pas del moment de disseny d'una base de dades.

2. Ja hi havien masses paràmetres per valorar.

Tot això degut a que les etapes de disseny i d'optimització d'una base de dades són diferents. En aquest sentit, aquest projecte es diferencia de la literatura publicada fins ara en el fet que el focus principal és l'etapa de disseny i es deixa de banda la optimització a nivell d'execució. El motiu és que fins al moment la recerca s'ha focalitzat molt en la fase d'optimització, ja que aquests nous tipus de base de dades es venen sobretot com grans sistemes paral·lels. No obstant això, en aquest projecte s'ha volgut posar a prova si realment tot plegat és una qüestió de paral·lelisme o un bon disseny físic continua sent essencial.

Finalment, els valors pensats per cada un d'aquests paràmetres són:

Replicació HDFS	Sense replicació, replicació 3 (per defecte) i replicació 5.
RegionServers	2, 5 i 8. El punt de partida són 8 RegionServers ja que aparentment un clúster Hadoop comença a oferir millors resultats a partir de 8 RegionServers. També s'han triat 2 RegionServers ja que és el mínim sense perdre el sistema distribuït en sí. Finalment 5 RegionServers perquè és la mitjana.
Fragmentació vertical	Sense, un atribut per fragment i matriu d'afinitat (veure més abaix). La nomenclatura utilitzada al llarg de les proves per tal de distingir entre estratègies de fragmentació vertical és: *SingleColumn, ColumnFamily,* i *AffinityMatrix*, respectivament.
Compressió	Sense i pesada. La compressió lleugera no ve integrada amb HBase i per poder-la utilitzar s'han d'instal·lar paquets i *plug-ins* externs.

Per tal d'aclarir de què es tracta la matriu d'afinitat, cal mencionar que aquest és un algorisme per trobar una certa fragmentació vertical. Donada una càrrega de treball formada per un seguit de queries i un percentatge assignat a cada query, l'algorisme de matriu d'afinitat busca identificar mitjançant la suma de percentatges de les queries quins atributs són els que s'accedeixen més regularment. Així, dos atribut a, b que apareguin tots dos en dues queries diferents amb percentatges x, y respectivament, tindran una afinitat $x + y$. Aleshores, l'objectiu final consisteix en identificar parelles d'atributs amb afinitats més o menys semblants i aleshores agrupar-los en un fragment.

D'aquesta manera, el resultat d'aplicar la matriu d'afinitat al conjunt de queries d'aquest projecte ha resultat en formar un únic grup de sis atributs. La resta no tenen una afinitat suficient i per tant cal que cada un d'ells tingui el seu fragment.

En aquest sentit la matriu d'afinitat permet definir un punt mig entre les estratègies *ColumnFamily* i *SingleColumn*.

Scale Factors

Les proves d'inserció dutes a terme a l'apartat *7.1.7.2 Clúster del DAC* garanteixen que és viable treballar amb SF compresos entre 1 i 5 (inclosos). No obstant això, amb SF 1 la distribució de les dades no es produeix correctament i per tant el mínim recomanable és 2. D'aquesta manera, es va decidir simplificar el nombre de SF a tres: 2 (60 GB) perquè és el mínim, 4 (120 GB) i 6 (180 GB). Per tal de poder treballar amb SF 6, primerament va caldre realitzar una petita prova per tal de valorar si era viable arribar a aquest volum de dades.

Queries

La idea inicial era aprofitar el conjunt de queries heretades del projecte anterior. Entrant en més detall, aquest conjunt de queries intenta ser una petita simplificació del conjunt total de queries del TPC-H degut a que algunes d'aquestes queries segueixen una sintaxi massa complexa en vers a la simplicitat de la sintaxi del sistema desenvolupat aquí. Concretament, es tracta d'agafar el conjunt total de queries del TPC-H i valorar

cada paràmetre de query de mínim a màxim, fixant la resta de paràmetres a la mediana del conjunt total de queries del TPC-H.

Paràmetre de query	**Mínim**	**Màxim**	**Mediana**
Cardinalitat de la projecció	1	9	3
Cardinalitat de l'agrupació	0	7	1
Cardinalitat de la selecció	1	6	2
Factor de selecció	10^{-5}	1	10^{-2}

Malauradament, aquest és un conjunt de trenta queries: nou queries per valorar la cardinalitat de la projecció, vuit queries per a la cardinalitat de l'agrupació, vuit queries per a la selecció i sis queries pel factor de selecció. Per simplificar aquest conjunt de queries, es va decidir mantenir les sis queries del factor de selecció, però reduir les queries de la resta de paràmetres a tres queries per paràmetre: mínim, màxim i mitjana (en els casos en què la mitjana numèrica té part decimal es simplifica a la part entera inferior). Així, el conjunt final es simplifica a quinze queries en total. Si posteriorment cal valorar algun d'aquests paràmetres en més detall ja es farà de forma molt més específica.

És important notar que trobar factors de selecció exactes és una tasca realment difícil, i més tenint en compte que el número de seleccions ha d'estar fixat a la mediana. Aleshores, en aquest cas es relaxen aquestes restriccions i cal tenir en compte que:

- Els factors de selecció reals són aproximacions als que realment haurien de ser.
- Les queries amb factors de selecció més petits tenen permès fer servir més clàusules de selecció.

Així, la següent taula és una comparativa del disseny final de les queries que avaluaran el factor de selecció (FS a la taula) respecte el projecte anterior, aquest projecte i la cardinalitat de la selecció:

FS desitjat	**FS anterior**	**FS actual**	**Selecció**
10^{-5}	$1,66 \cdot 10^{-5}$	$1,77 \cdot 10^{-5}$	4
10^{-4}	$0,99 \cdot 10^{-4}$	$0,86 \cdot 10^{-4}$	3
10^{-3}	$1,08 \cdot 10^{-3}$	$0,96 \cdot 10^{-3}$	2
10^{-2}	$1,00 \cdot 10^{-2}$	$0,99 \cdot 10^{-2}$	2
10^{-1}	$0,92 \cdot 10^{-1}$	$0,90 \cdot 10^{-1}$	2
1	1	0,99	2

Les diferències principals en el factor de selecció resultant entre aquest projecte i l'anterior es deuen al fet que el generador utilitzat en el procés d'inserció és diferent i que en el projecte anterior no es tenia en compte un petit conjunt de tuples que en aquest projecte sí es tenen en compte (i per això els factors de selecció actuals són relativament més petits). En qualsevol cas, la diferència amb els factors de selecció desitjats és menyspreable.

Algorismes

Tal i com es detalla al llarg de tot l'apartat *5 Situació de partida*, en total n'hi han cinc algorismes per executar:

1. El FSS
2. L'IFS amb *Single Level Index*

3. L'IFS amb *Multiple Level Index*
4. L'IRA amb *Single Level Index*
5. L'IRA amb *Multiple Level Index*

Si recordem, a l'apartat *5.2 Creació dels índexs* s'explica com el tipus d'índex *Multiple Level Index* diferencia entre nivells d'agregació i per tant és més òptim quan s'accedeix a dimensions amb varis nivells d'agregació. No obstant això, aquest no és el cas d'aquest projecte, on el conjunt final de queries finals només treballa sobre dimensions amb un únic nivell d'agregació. Tot això implica que, al no haver nivells d'agregacions intermitjos en cada una de les dimensions, tots dos tipus d'índexs funcionen igual. Per tant, es simplifica el conjunt d'algorismes a:

1. El FSS
2. L'IFS amb *Single Level Index*
3. L'IRA amb *Single Level Index*

Repeticions

Es va decidir que el nombre de repeticions d'un mateix experiment per evitar que els resultats poguessin ser distorsionats per l'aparició d'outliers o per un consum puntual i excessiu dels recursos de les màquines fos tres, tot i que posteriorment es redueix a dos per tal d'agilitzar les proves al constatar que els resultats són força constants entre execucions. En qualsevol cas, si posteriorment en calen més, sempre es poden refer.

8.2 Primeres proves

Aquesta part del projecte correspon als resultats obtinguts durant el llançament de les primeres proves. Concretament, aquests són els resultats obtinguts a partir d'executar el conjunt total de queries sota una configuració de disseny de la base de dades determinada.

En aquest sentit, aquesta secció és fonamental ja que porta cap a les dades que posteriorment permeten concloure l'impacte de cada un dels paràmetres sobre el comportament final de l'HBase i per tant són les dades de les que finalment es podrà extreure la informació necessària per tal de complir el propòsit principal d'aquest projecte i construir el marc d'optimització de queries.

Així doncs, al llarg d'aquest apartat es mostren les gràfiques d'execució de cada un dels paràmetres de la tipologia de les queries: nombre d'atributs de projecció, agrupació, selecció i factor de selecció contraposant, per cada estratègia de fragmentació vertical, els costos finals dels tres algorismes d'accés.

Cal mencionar que les gràfiques que es mostren a continuació tenen els següents valors constants:

Scale Factor	6
Número de RegionServers	8
Compressió	Sense

L'objectiu de mostrar els resultats amb aquests valors constants és poder mostrar les gràfiques amb aquells paràmetres que realment influeixen en la tipologia de la query. Els factors que es deixen com a constants afecten al cost d'execució final de les queries

en termes de "maquinari", ja que o bé permeten distribuir la càrrega en més màquines (nombre de RegionServers), o bé ajuden a reduir la càrrega per màquina (SF i compressió). Però en qualsevol cas tenen una relació clara amb els paràmetres de la tipologia de la query. En canvi, el tipus de fragmentació vertical és el paràmetre que realment pot ajudar a començar a entendre el funcionament intern de l'HBase, ja que en funció del tipus triat cada algorisme es pot comportar d'una forma diferent.

De totes maneres, a l'annex *A Gràfiques per configuració de paràmetres* s'hi poden trobar les gràfiques de la tipologia de la query corresponents a cada combinació de paràmetres de disseny de la base de dades per tal de donar llibertat al lector d'aprofundir en més detall en els resultats dels experiments.

Atributs de projecció

Els nombre d'atributs de projecció correspon al nombre de mesures seleccionades. Això vol dir que en aquestes gràfiques s'hauria de poder contraposar el cost de CPU de dur a terme les agregacions amb el cost de llegir de disc més o menys dades. Per tal de dur a terme aquest anàlisi, cal centrar-se primordialment en la gràfica *SingleColumn*, ja que en les altres dues es veu clarament com en la figura *8.2-1* quant més atributs de projecció més alt és el cost. En canvi, quan es tracta de *SingleColumn* el cost final sembla independent del nombre d'atributs de projecció.

El que està ocorrent realment aquí és que el cost de CPU és negligible en vers al cost de llegir de disc. Però, a la vegada, el cost de llegir de disc no és funció del volum de dades a llegir, ja que si fos així, la gràfica *SingleColumn* no seria pas constant. El cost de llegir de disc és en realitat funció del nombre de families a les que cal accedir.

Aleshores, això explicaria el comportament dels algorismes segons les tres estratègies de fragmentació vertical:

- Amb les estratègies *ColumnFamily* i *AffinityMatrix* llegir més atributs de projecció implica un cost final d'execució més gran perquè es tracta de llegir més families en tots dos casos.
- Amb l'estratègia *SingleColumn* sempre s'ha de llegir una sola family i per tant el volum de dades a llegir sempre és el mateix, independentment de la proporció de dades útils i brossa que es llegeixi de més.

Un segon detall important de la gràfica, i que segueix la línia de l'explicació anterior, és el fet que amb *ColumnFamily* i *AffinityMatrix* el temps d'execució estigui molt per sota que amb *SingleColumn*. El motiu és el volum de dades que s'ha de llegir. Malgrat no hi hagi costos addicionals de llegir més families, sí que s'ha de llegir una sola family on el volum de dades està molt per sobre de les altres dues tècniques de fragmentació vertical.

Així doncs, d'aquestes primeres gràfiques es pot extreure dos paràmetres que tenen un efecte notable en el rendiment final dels algorismes:

1. El número de families que cal llegir.
2. El volum de dades total que cal llegir (dades brossa) per tal d'obtenir el volum de dades desitjat (dades útils).

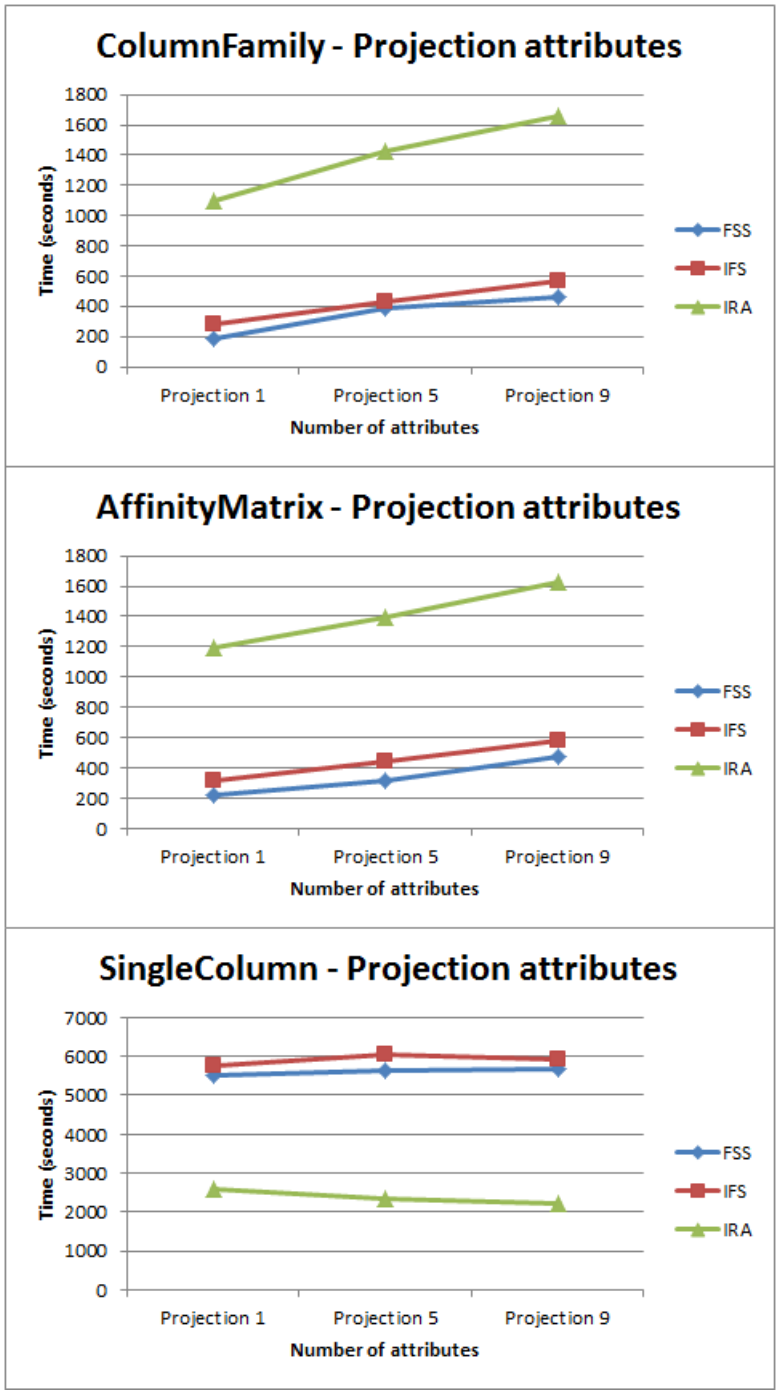


Figura 8.2-1: Gràfiques del temps d'execució segons el nombre d'atributs de projecció

Atributs d'agrupació

El costos de llegir més atributs d'agrupació no afegeixen cap paràmetre rellevant. En realitat, això és degut al fet que els costos de llegir més atributs d'agrupació són els mateixos que els costos de llegir més atributs de projecció. Per tant la discussió és la mateixa que l'anterior. La figura *8.2-2* mostra aquests resultats.

La diferència principal rau en l'ús que se li donen als atributs d'agrupació respecte als de projecció. Mentre que els atributs de projecció són les mesures seleccionades, els atributs d'agrupació són les dimensions del cub de dades. Per tant, la diferència és que els atributs d'agrupació haurien de tenir un cost de CPU més reduït que no pas els atributs de projecció. En qualsevol cas, ja ha quedat discutit a la secció anterior que els costs de CPU són menyspreables respecte els costos de disc i per tant es fa difícil fer-los notar en aquestes gràfiques.

Atributs de selecció

El cas dels atributs de selecció és diferent al de la resta ja que en aquest cas es tracta de comparar la selecció directament sobre la taula de fets (algorisme FSS) contra els accessos a l'índex (IFS i IRA).

Per tal de no perdre el fil de les explicacions fetes fins ara, la primera reflexió ha d'anar encaminada de cara als costos de llegir més families en les tres fragmentacions verticals. Efectivament, la tendència continua sent la mateixa en el cas del FSS segons la figura *8.2-3*: llegir més families amb les estratègies *ColumnFamily* i *AffinityMatrix* té un cost afegit que és inexistent en cas de la gràfica *SingleColumn*.

En canvi, en aquest cas aquests costos no es poden extrapolar als algorismes IFS i IRA, ja que en aquesta gràfica aquests algorismes no fan cap lectura de la taula de fets, sinó que produeixen la selecció mitjançant accessos a l'índex. Per tant és fàcil comprovar que quan més gran sigui la cardinalitat de la selecció, l'empitjorament dels accessos a l'índex és molt més fort que no pas llegir més atributs de la taula de fets.

Per tant, la reflexió final és que per tal d'elaborar el marc d'optimització de queries, un cost addicional que caldrà tenir en compte és el cost d'accedir a l'índex en els algorismes IFS i IRA. Com que aquest índex no és més que una taula HBase auxiliar, el cost d'accedir a l'índex hauria de poder-se formular a partir de les mateixes conclusions que donaran peu a calcular el cost de llegir de la taula de fets.

Factor de selecció

El factor de selecció és possiblement el paràmetre de query més diferenciat de la resta. Principalment pel fet que les explicacions anteriors es redueixen senzillament al fet de llegir més o menys atributs. En aquest cas, es tracta de llegir més o menys tuples. La diferència entre aquest paràmetre i els anteriors es podria definir com la diferència entre llegir "files" o "columnes" de la taula.

Per altra banda, aquestes gràfiques ja permeten discernir entre un algorisme o un altre. En altres paraules, els paràmetres de query anteriors permeten trobar factors de comportament de l'HBase, però en tots els casos les gràfiques de cada algorisme són paral·leles i per tant no permeten intuir les condicions que haurien de permetre decidir l'algorisme a emprar.

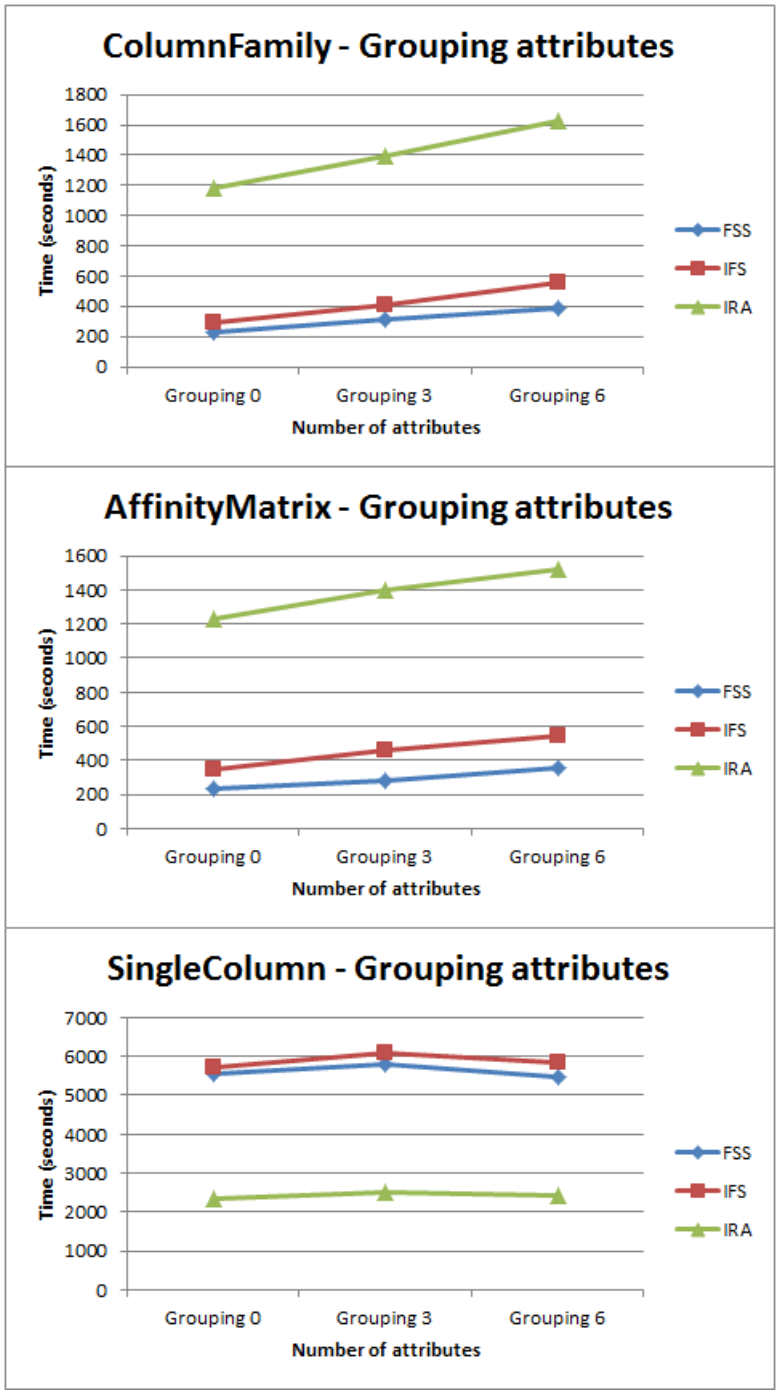


Figura 8.2-2: Gràfiques del temps d'execució segons el nombre d'atributs d'agrupació

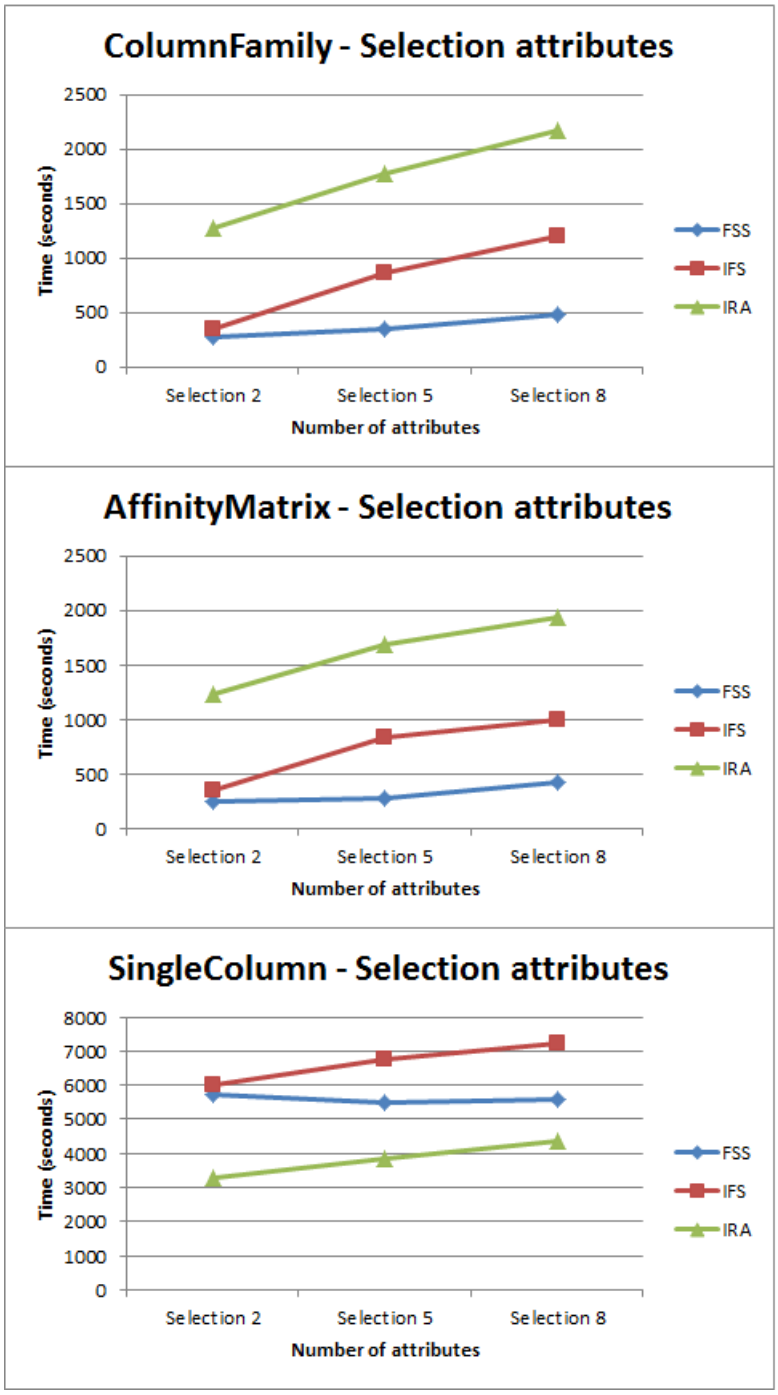


Figura 8.2-3: Gràfiques del temps d'execució segons el nombre d'atributs de selecció

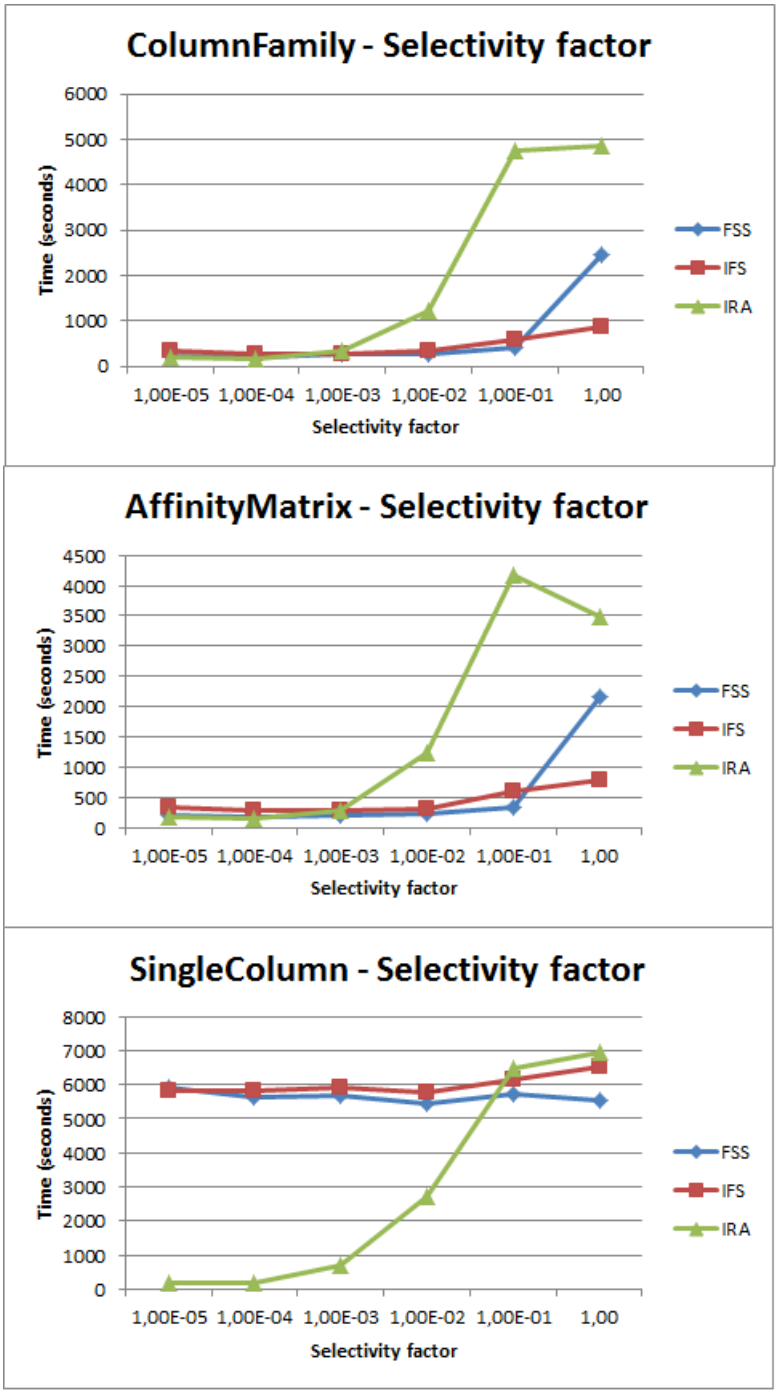


Figura 8.2-4: Gràfiques del temps d'execució segons el factor de selecció

En canvi, tal i com mostra la figura *8.2-4*, en el cas del factor de selecció les gràfiques dels algorismes ja no són paral·leles i per tant es pot identificar fàcilment que el factor de selecció serà un paràmetre més a tenir en compte durant la decisió de l'algorisme d'accés. Més concretament, les línies generals segueixen la següent taula:

Factor de selecció	**Millor algorisme**
10^{-5}	IRA
10^{-4}	IRA
10^{-3}	IRA
10^{-2}	FSS
10^{-1}	FSS
1	IFS

No obstant això, les reflexions fetes anteriorment continuen sent certes i per exemple el cost de llegir més brossa continua tenint un impacte negatiu en els algorismes. El cas més clar és la gràfica de *SingleColumn*, on es pot identificar com la lectura d'aquesta brossa afecta principalment als algorismes FSS i IFS.

En canvi, aparentment l'estratègia de fragmentació vertical no té cap efecte sobre l'algorisme IRA. Això és degut a que els accessos que fa l'IRA sobre la taula de fets són accessos aleatoris cap a una sola tupla. Aleshores, també caldrà diferenciar entre accessos seqüencials i accessos aleatoris a la taula de fets.

Tant és així, que aquests accessos seqüencials poden ser una de les raons de la caiguda de la gràfica de l'IRA entre els factors de selecció 10^{-1} i 1. Més concretament, aquesta caiguda pot seguir dues justificacions diferents:

1. Les dades estan molt agrupades amb factor de selecció 1 (de fet, és tota la taula) i per tant durant el preprocessament de la taula temporal de l'IRA (veure apartat *5.4.3 Index Random Access. IRA*) els accessos aleatoris és converteixen en accessos seqüencials. Això deriva en que el cost d'enviar les tuples de l'HBase a la tasca MapReduce es redueixen, ja que quan es tracta d'accessos seqüencials l'objecte que fa la lectura de l'HBase fa servir un petit buffer com a cache per tal de reduïr costos de comunicació enviant les tuples en paquets. Quan es tracta d'accessos aleatoris, aquests objectes lectors són diferents i per tant les tuples s'han d'enviar una a una i els costos de comunicació són elevats. No obstant això, amb factors de selecció petits aquests costos de comunicació són més petits que els costs de lectura de disc i per tant els accessos aleatoris surten a compte, i per aquest motiu l'IRA és millor en aquests casos.

2. Una segona justificació podria ser que amb el factor de selecció 1, la taula temporal de l'IRA creix lo suficient com per a dur a terme un o varis region splits (veure apartat *4.2.2 Fragmentació horitzontal: Autosharding*) i per tant els accessos a la taula de fets es poden fer amb més paral·lelisme.

De totes maneres, tant el costos de comunicació en funció del número de tuples que es poden enviar a la vegada, com el número de regions de la taula temporal de l'IRA, són factors que caldrà tenir en compte al computar el cost de cada algorisme i per tant, ja sigui degut a una justificació o a una altra, aquests casos ja es veuran reflectits a l'algorisme triat.

Per altra banda, l'altra reflexió important que cal fer en aquest gràfica és el fet que l'IFS ofereixi millors resultats que el FSS amb les fragmentacions verticals *ColumnFamily* i

AffinityMatrix i factors de selecció grans. Això xoca amb el sentit comú, ja que en principi quan més gran sigui el factor de selecció, millors haurien de ser els algorismes de *full scan*.

Per tal de donar una millor resposta a aquesta controvèrsia, la figura *8.2-5* contraposa el funcionament d'ambdós algorismes.

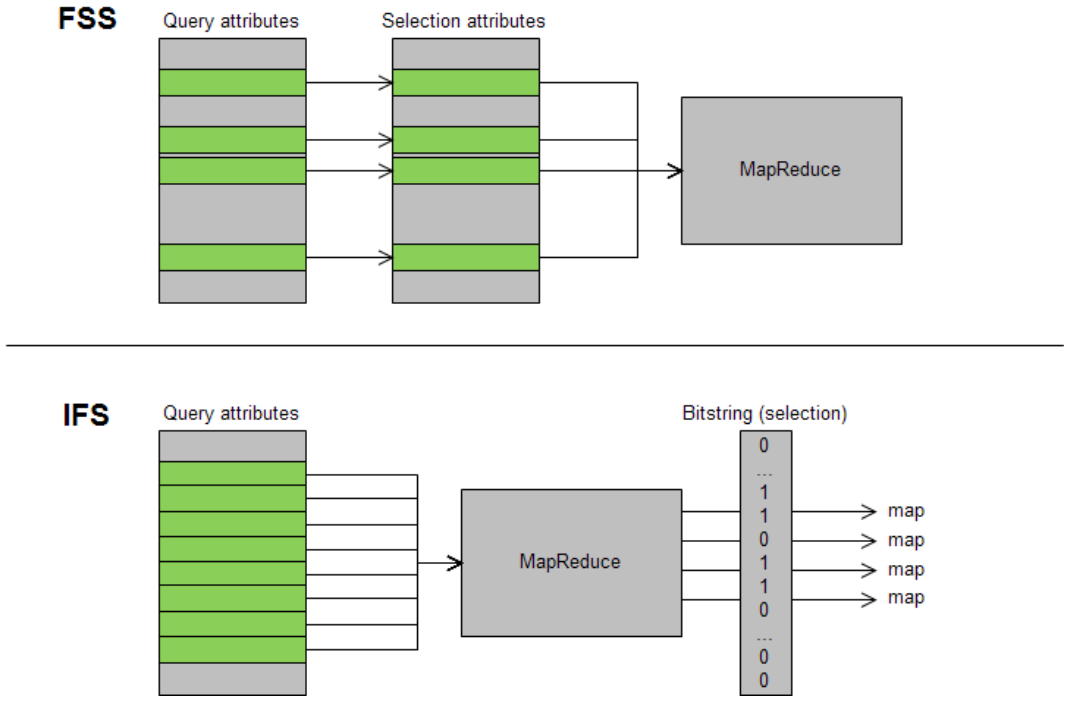


Figura 8.2-5: Comparativa del funcionament intern del FSS i l'IFS

El que exposa aquesta figura és com l'IFS es beneficia del primer accés de l'índex per tal d'evitar els consegüents costos de lectura de la taula de fets i d'enviament dels atributs de selecció cap a l'execució MapReduce.

Així, per a factors de selecció petits és pràcticament impossible que l'IFS ofereixi un millor rendiment que el FSS, ja que l'IFS produeix la selecció en el moment d'executar el map i per tant els costos d'enviar dades innecessàries es fa molt patent (ha d'enviar totes les tuples). En canvi, quan es tracta de factors de selecció més grans, el guany està precisament en estalviar la lectura i l'enviament dels atributs de selecció degut a que en aquestes situacions la selecció és nul·la i per tant enviar els atributs de selecció és innecessari.

En el cas de l'estratègia de fragmentació vertical *SingleColumn* i factor de selecció 1, l'algorisme que ofereix millor rendiment és el FSS ja que en aquesta situació l'IFS no pot evitar haver de fer la lectura dels atributs de selecció i per tant els costos de disc hi són presents.

8.3 Discussió

8.3.1 Discussió general

La discussió dels resultats finals obtinguts ve donada per l'explotació de les dades resultants dels experiments, per tal de trobar els patrons adequats que permetin identificar empíricament on són els costos d'execució de cada query i de cada algorisme. D'aquesta

manera, l'objectiu és dur a terme la valoració d'aquests costos per tal de construir el model d'optimització. Tot això implica que en aquest apartat s'entra en detall en el funcionament intern de les tecnologies emprades i per tant és molt recomanable conèixer el seu funcionament intern (apartat *4.2 Arquitectura de les tecnologies emprades*).

Fragmentació vertical

Per tal de continuar amb la línia principal de les conclusions heretades de les primeres proves (apartat *8.2 Primeres proves*), el punt d'entrada és la comparació de llegir la taula de fets segons una estratègia de fragmentació vertical o una altra. Així la figura *8.3-1* mostra l'efecte que tenen cada una de les tres estratègies de fragmentació vertical respecte l'algorisme FSS.

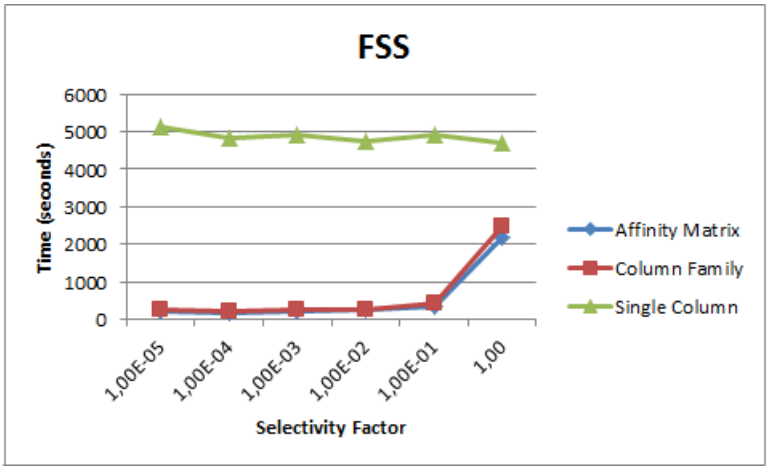


Figura 8.3-1: Estratègies de fragmentació vertical respecte el FSS

En aquesta figura es veu clarament com les estratègies *ColumnFamily* i *AffinityMatrix* produeixen un efecte molt similar en el rendiment del FSS. En canvi, l'estratègia *SingleColumn* produeix un efecte constant.

Això demostra les impressions de l'apartat *8.2 Primeres proves* on es discuteix com afecta la fragmentació vertical al cost final de fer lectures seqüencials. El motiu és que quan es tracta de les fragmentacions *ColumnFamily* i/o *AffinityMatrix*, només cal llegir aquells fragments vertical que contenen algun dels atributs de la query. En canvi, en una situació on la fragmentació vertical és inexistent com és el cas del *SingleColumn*, no importa quins atributs hagin de ser llegits per la query, ja que en qualsevol cas la lectura ha de ser per tots els atributs de la taula.

Aquesta explicació justificaria el perquè els temps d'execució sense fragmentació vertical són significativament més elevats que aquells amb fragmentació vertical *ColumnFamily* o *AffinityMatrix*.

A més, aquesta gràfica permet identificar un segon cost. Aquest seria el cost implicat en els increments que pateix el FSS entre el factor de selecció 10^{-1} i 1, i amb les estratègies de fragmentació vertical *ColumnFamily* i *AffinityMatrix*. Fent servir el coneixement adquirit fins ara, la justificació d'aquest segon cost podria venir per una de les següents branques:

1. Aquest segon cost correspon al cost d'enviar les tuples de l'HBase cap al procés MapReduce. Concretament, parlant a nivell més tecnològic no es pot suposar que el MapReduce obté les tuples directament dels espais de memòria sobre els que treballa l'HBase, ja que tots tres són processos diferents: l'HDFS, l'HBase i el MapReduce.

Per tant, no existeix cap tipus de memòria compartida. A més, les màquines que operen com a servidors de l'HBase no tenen perquè ser màquines del MapReduce, i per tant es pot identificar aquest cost com a cost de xarxa degut a que cal moure les dades d'una màquina a una altra. De fet, quan finalitza la lectura de les tuples a l'HBase, es desconeix si aquesta mateixa màquina està corrent també un procés MapReduce, i més tard, quan ja es coneix, no hi ha espai de memòria compartit per fer-ho d'una forma ràpida. Per tant, la única assumpció possible aquí és que l'HBase sempre ha d'enviar les tuples via xarxa i per tant aquest cos es fa notable. No obstant això, és ben sabut que els costos de xarxa d'enviar dades d'una màquina a ella mateixa són menors que els d'enviar dades a una altra màquina i per tant córrer un procés HBase amb un procés MapReduce continua tenint els beneficis de localitat.

Aleshores, si fos aquest el cas, aquest cost també hauria d'aparèixer amb la fragmentació vertical *SingleColumn* ja que les dades que es processen al MapReduce són les mateixes. No obstant això, el fet que no aparegui no significa que no hi sigui. Podria donar-se el cas que aquest cost d'enviament és negligible en comparació amb el cost de lectura de disc amb l'estratègia *SingleColumn*.

2. Un segon argument és que aquest cost és degut a la tasca d'ajuntar cada un dels fragments verticals per tal de formar la tupla. Com que en el cas del *SingleColumn* tots els atributs estan al mateix fragment vertical, aquest cost és nul.

Aquí la discussió augmenta de to i per tal de donar una resposta clara cal consultar els resultats dels experiments en funció del SF i de la fragmentació vertical *SingleColumn*:

- Si amb SF més petits es produeix aquest petit increment, aleshores el primer dels arguments anteriors es farà patent, ja que el costos de llegir de disc haurien de reduir-se deixant pas als costos d'enviament de tuples.
- En cas contrari, la justificació serà que amb *SingleColumn* mai es produeix aquest increment de temps i per tant l'explicació vàlida serà la segona.

Així doncs, la figura *8.3-2* mostra aquesta informació.

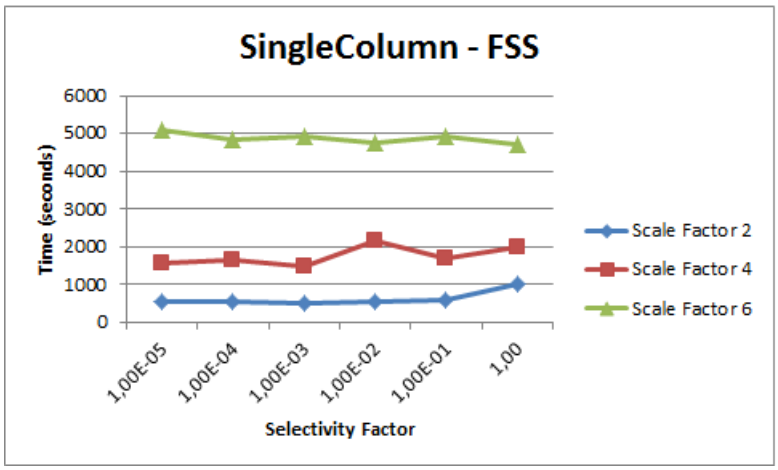


Figura 8.3-2: Volums de dades respecte el FSS amb *SingleColumn*

L'objectiu de mostrar aquesta figura és comprovar com, efectivament, amb la mateixa situació que la gràfica *8.3-1* però amb volums de dades més petits sí apareix un petit increment de temps entre els factors de selecció 10^{-1} i 1. A més, amb els temps a la mà:

Volum de dades	**Factor de selecció** 10^{-1}	**Factor de selecció** 1	**Diferència**
Scale Factor = 2	598,96	1010,31	411,35
Scale Factor = 4	1673,19	1987,35	314,16
Scale Factor = 6	4921,86 (4750,61)	4716,72	-205,14 (-33,88)

es pot apreciar com a mesura que augmenta el volum de dades es redueix la diferència entre els factors de selecció 10^{-1} i 1. Això prova la primera de les justificacions anteriors, on s'estimava que el cost de la segona discussió podria ser el cost d'enviar les tuples de l'HBase cap al MapReduce, però que en cas de volums de dades grans aquest segon cost quedava superposat pel cost de llegir de disc.

De totes maneres, sembla que en el cas de SF = 6 hi ha un petit pic en el temps d'execució del factor de selecció 10^{-1}, i per tant entre parèntesis es mostren els temps amb el factor de selecció 10^{-2}.

Seguint aquesta línia d'argumentació, es podria intentar preveure com hauria de ser la mateixa gràfica *8.3-1* però per l'algorisme IFS. En aquest cas, cal recordar que aquest algorisme du a terme la selecció durant la tasca MapReduce (apartat *5.4.2 Index Filtered Scan. IFS*). Aleshores és inevitable que, independentment del factor de selecció, l'IFS hagi de d'enviar totes les tuples cap al MapReduce, ja que en aquest moment desconeix les tuples que satisfan la selecció. La part positiva és que com que primerament es fa un accés a l'índex ja es coneix quines tuples satisfaran la selecció, i per tant l'IFS es pot optimitzar per fer la lectura només d'aquelles tuples que estiguin en el rang entre la primera i la última tupla que satisfan la selecció. Així, si només existeixen dos tuples que satisfan la selecció i són tuples relativament consecutives, aquest rang de tuples serà més petit que el cas en el que aquestes tuples siguin la primera i la última tupla de la taula.

No obstant això, degut a la distribució aleatòria que segueix el model de dades inserit, es fa pràcticament impossible determinar quin serà aquest rang per cada factor de selecció. Aleshores, l'assumpció ha de ser que amb l'IFS sempre cal llegir la taula sencera. De totes maneres, durant l'execució de les proves es van programar els logs per tal de salvar aquesta informació i el rang de tuples per qualsevol query era molt proper a llegir tota la taula de fets en tots els casos.

Tornant al fil principal de justificació, els arguments anteriors indiquen que les estratègies de fragmentació vertical haurien d'afectar a l'IFS fent que el cost fos sempre constant. Això hauria de ser degut al fet que, partint dels dos paràmetres de cost trobats fins ara: a) el cost de llegir de disc, i b) el cost de moure les dades cap al MapReduce, són independents en qualsevol factor de selecció i estratègia de fragmentació vertical, ja que en qualsevol cas sempre cal moure totes les tuples per tal de dur a terme la selecció a nivell MapReduce.

Tot i així, sí que és cert que en el cas de la fragmentació vertical *SingleColumn*, els costos haurien de ser significativament majors que amb qualsevol altra estratègia, pels mateixos motius que ocorre amb el FSS.

La figura *8.3-3* permet comprovar si aquestes hipòtesis són correctes.

I per tant es pot concloure que els costos identificats a l'algorisme FSS també s'apliquen a l'IFS.

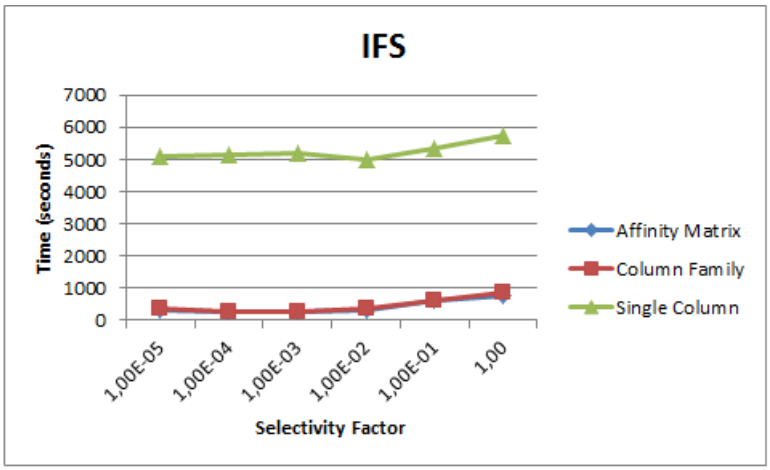


Figura 8.3-3: Estratègies de fragmentació vertical respecte l'IFS

En aquesta gràfica apareix un petit increment en el temps d'execució segons creix el factor de selecció. Aquest increment s'atribueix a costos extres derivats de l'accés a l'índex i d'executar tasques MapReduce amb més tuples.

Per altra banda, el sentit comú diu que quan es tracta de factors de selecció grans, l'algorisme IFS hauria d'oferir millors rendiments que no pas el FSS, ja que en el cas de l'IFS tant el cost de llegir de disc, com el d'enviament de tuples són menors perquè s'estalvien els atributs de selecció. La figura *8.3-4* mostra una taula comparativa entre tots dos algorismes.

FSS	IFS
AffinityMatrix	
200,2	327,87
186,62	282,42
218,18	275,37
238,86	324,93
335,31	597,49
2160,28	776,56
ColumnFamily	
247,08	351,9
200,98	293,16
264,49	291,47
272,5	346,49
431,29	605,76
2456,21	864,36
SingleColumn	
5108,98	5112,54
4827,8	5132,68
4900,8	5200,52
4750,61	4992,27
4921,86	5346,09
4716,72	5718,47

Figura 8.3-4: Comparativa dels temps d'execució entre el FSS i l'IFS

Efectivament, aquest és el cas quan existeix un mínim de fragmentació vertical. Tal i com es mostra en aquesta taula, quan la fragmentació vertical és de tipus *SingleColumn*,

l'algorisme FSS torna a ser el més òptim. L'explicació que s'aplica aquí és que quan no hi ha fragmentació vertical, els costos de llegir de disc són els mateixos i per tant l'IFS no té alternativa per evitar llegir els atributs de selecció.

Finalment, la última discussió en aquesta línia ve quan es contraposa l'estratègia de fragmentació vertical amb l'algorisme IRA. La figura *8.3-5* mostra aquesta comparativa.

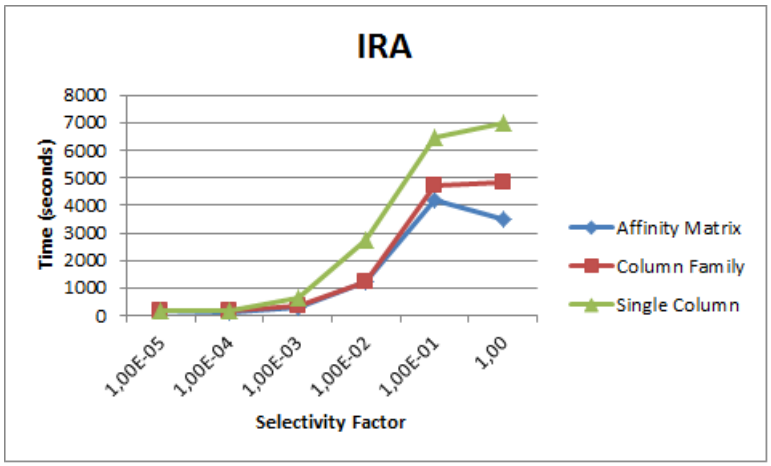


Figura 8.3-5: Estratègies de fragmentació vertical respecte l'IRA

Aquest algorisme és sens dubte el més diferent de tots. Els motius salten a la vista. Per tal de comprendre els motius, cal diferenciar entre dos tipus de lectures de disc:

- La lectura seqüencial que com el seu nom indica implica llegir seqüencialment de disc. Aquest tipus de lectura és el dels algorismes FSS i IFS.
- La lectura aleatòria que implica accedir a posicions de disc no seqüencials mitjançant mètodes d'indexació.

Havent definit això, és lògic que funcioni de forma diferent. En els algorismes anteriors, el paràmetre de discussió més rellevant ha estat el cost de lectura seqüencial, però en aquest cas aquest cost no existeix fins a factors de selecció molt alts, ja que el tipus d'accés és principalment aleatori.

Així, la lectura que s'ha de fer de la consegüent gràfica és que fer un accés aleatori a la taula té un cost fix k, i per tant el cost de fer n accessos aleatoris és nk. Malgrat el creixement de la gràfica anterior sembli exponencial, cal tenir en compte que l'eix d'abscisses creix exponencialment i per tant els costos al augmentar el factor de selecció possiblement siguin linials.

Una característica important d'aquest algorisme (veure apartat *5.4.3 Index Random Access. IRA*) és que després de l'accés a l'índex es realitza un preprocessament de les keys obtingudes per tal de trobar petits rangs de tuples i així poder convertir accessos aleatoris en accessos seqüencials. Partint de l'experiència d'altres àrees de la informàtica, es pot suposar que el cost d'accedir seqüèncialment a disc és més barat que accedir aleatòriament quan el volum de dades és suficientment gran, ja que les latències de moure el capçal del disc desapareixen i es maximitza l'ús dels buffers de lectura. Aleshores, quan el factor de selecció és 1, tot l'accés a la base de dades es fa seqüencial i per tant es talla el creixement deixa de tenir la tendència linial.

A més, sembla que aquest algorisme també es beneficia quan existeix fragmentació vertical. Així, en el cas del *SingleColumn* sembla que el factor linial k té un valor major. Aleshores, això el que pot estar indicant és com funciona l'accés aleatori a l'HBase. El que està clar és que per fer aquest accés aleatori es fa servir algun tipus d'indexació interna, i això és el que justifica que els temps d'execució amb factors de selecció siguin menors amb l'IRA. La resta són pures conjectures, però no és gaire desencertat deduir que aquest accés aleatori dirigeix cap a la primera entrada de la tupla en qüestió dins de l'store, i aleshores la cerca de l'entrada demanada (parella *family:qualifier*, veure apartat *4.2.2 Fragmentació vertical: Families*) és un accés seqüencial des del punt de vista de blocs de disc i no blocs HDFS. Això justificaria que amb una fragmentació vertical més simple com la *SingleColumn* els atributs estan repartits en menys stores, i per tant el cost de trobar la *family:qualifier* demanada és més elevat.

Número RegionServers

Un altre dels paràmetres a avaluar és el nombre de RegionServers. Com que la fragmentació vertical no és un paràmetre el qual es pugui fixar a la mitjana, els resultats en aquesta discussió també es presenten en funció de la fragmentació vertical.

Així, la primera d'aquestes discussions es mostra a la figura *8.3-6*.

El punt de partida d'aquestes tres figures hauria de ser la línia que representa els costos amb RegionServers = 8. Així, el punt de partida són les mateixes gràfiques que les de la discussió de la fragmentació vertical. No obstant això, cal anar en compte que en aquestes noves gràfiques l'escala de l'eix d'ordenades és més petit i per tant pot semblar que el creixement en certs punts és molt més remarcable que en les gràfiques anteriors.

Així doncs, la tendència al afegir RegionServers és molt clara: a mesura que s'afegeixen més màquines, el temps d'execució global millora. No obstant això, millora de forma paral·lela amb els algorismes FSS i IFS i per tant no sembla haver-hi evidències de que el nombre de màquines pugui afectar a la tria entre aquests dos algorismes. De fet, aquestes situacions són molt fàcil de raonar ja que el nombre de màquines permet distribuir la càrrega de dades al llarg de més RegionServers i per tant les lectures seqüencials es paral·lelitzen més. A més, també ho fan els costos de transportar les tuples de la base de dades al MapReduce, ja que s'eviten colls d'ampolla d'enviar les dades contra un conjunt de destinataris més petit.

El cas contrari és l'IRA. Aquest algorisme sembla no beneficiar-se del paral·lelisme afegit per més màquines fins que no arriba a factors de selecció grans. Aquest fet en realitat suporta les conjectures de l'apartat anterior: l'IRA realitza accessos aleatoris fins que el factor de selecció és molt gran. En aquest moment els accessos aleatoris es converteixen en seqüencials i per tant s'apliquen els arguments del FSS i l'IFS, on el nombre de RegionServers sí afecta.

Així, en aquests accessos seqüencials es pot determinar que una execució oferirà millors resultats quan més gran sigui el nombre de màquines. Tant és així que quan el nombre de màquines és petit el paral·lelisme no és suficient per fer aquestes lectures i per tant el cost sempre tendeix a créixer. En canvi, quan s'augmenta el paral·lelisme s'aconsegueix disminuir aquest cost i es produeix la millora entre els factors de selecció 10^{-1} i 1 discutida anteriorment.

En els accessos aleatoris es pot justificar que el nombre de màquines no afecta ja que l'accés a l'índex és transparent al número de RegionServers perquè en qualsevol cas l'índex dirigeix a una sola màquina per a una tupla determinada.

Havent discutit com afecta el número de RegionServers a cada una dels algorismes,

el següent pas és comprovar si les justificacions també es compleixen per les altres dues estratègies de fragmentació vertical.

La figura *8.3-7* mostra el cas de l'*AffinityMatrix*. Amb aquesta estratègia els resultats haurien de seguir la mateixa línia de justificació que amb *ColumnFamily*: el número de RegionServers només afecta a les lectures seqüencials, i quant més màquines, més es redueixen els costos.

Per tant es pot comprovar com aquestes justificacions continuen sent certes. A més, l'increment de RegionServers continua sent independent per tal de discernir entre FSS i IFS.

La figura *8.3-8* mostra la mateixa informació però contra la fragmentació vertical *SingleColumn*. En aquest cas, les justificacions haurien de ser molt més fortes, ja que al no existir cap tipus de fragmentació vertical, fer lectures seqüencials és molt més costos perquè s'han de llegir més dades. Aleshores, l'addició de màquines en aquesta situació hauria d'oferir millores molt més agressives.

Tal i com es pot comprovar, la millora agressiva hi és però entre 2 i 5 RegionServers. Quan s'hi afegeixen 8 màquines els resultats amb prou feines milloren. Aleshores, per tal d'entrar en més detall en aquesta qüestió la figura *8.3-9* mostra el *speed up*[38] obtingut al afegir més màquines a una situació inicial com:

[38]Factor de millora després d'optimitzar un procés, i es calcula com el percentatge del temps d'execució de la optimització respecte el temps d'execució original.

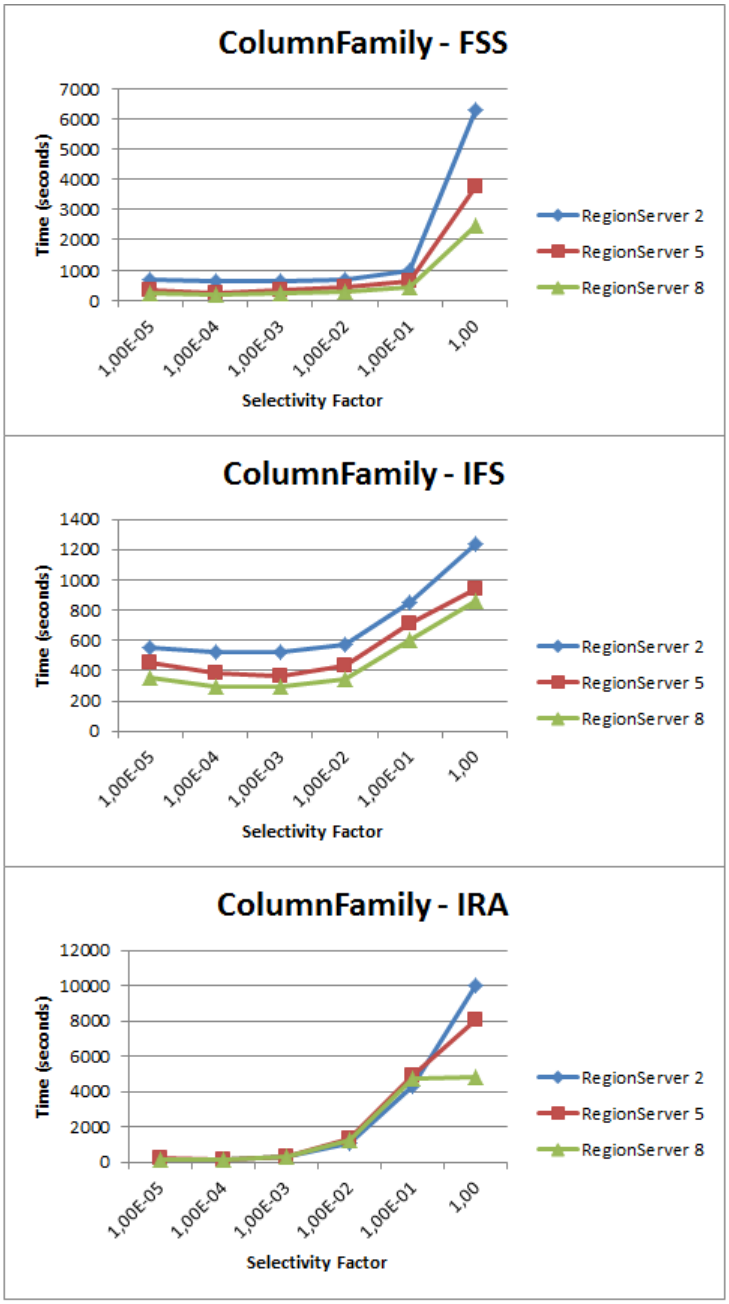


Figura 8.3-6: Comparativa del temps respecte el número de RegionServers i *ColumnFamily*

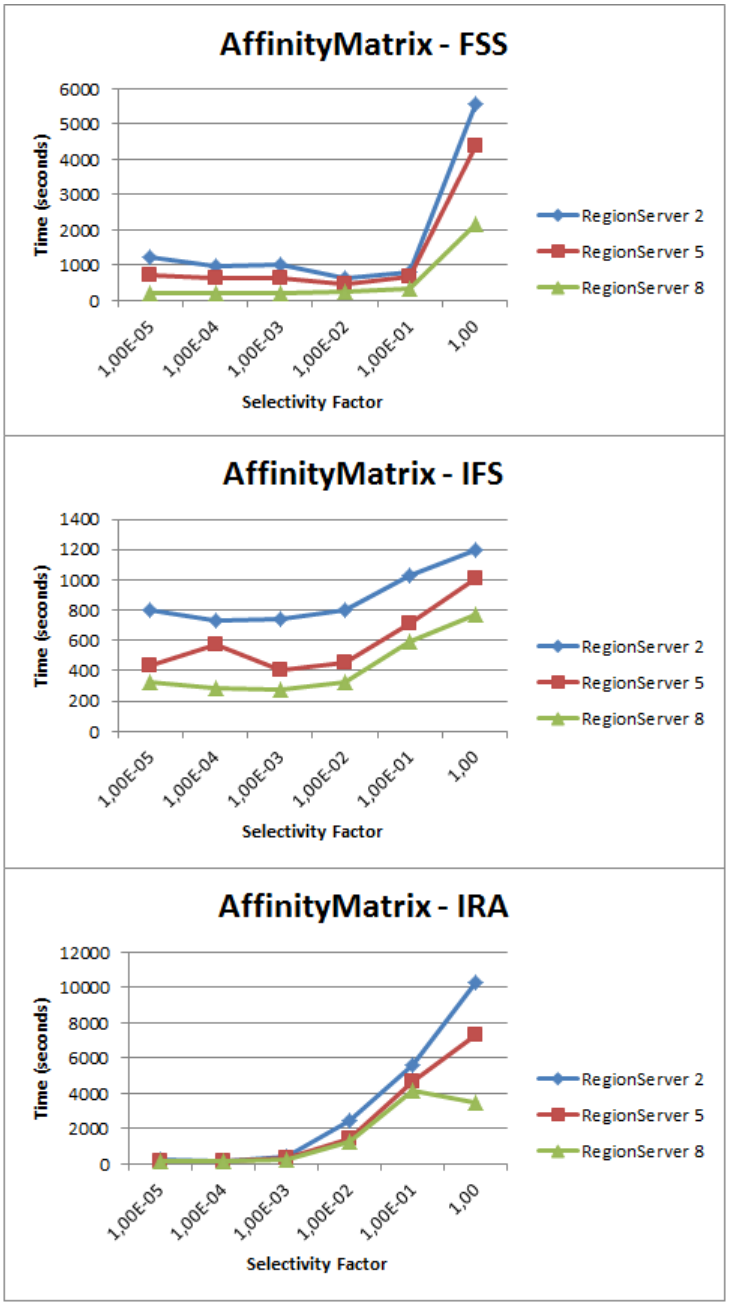


Figura 8.3-7: Comparativa del temps respecte el número de RegionServers i *AffinityMatrix*

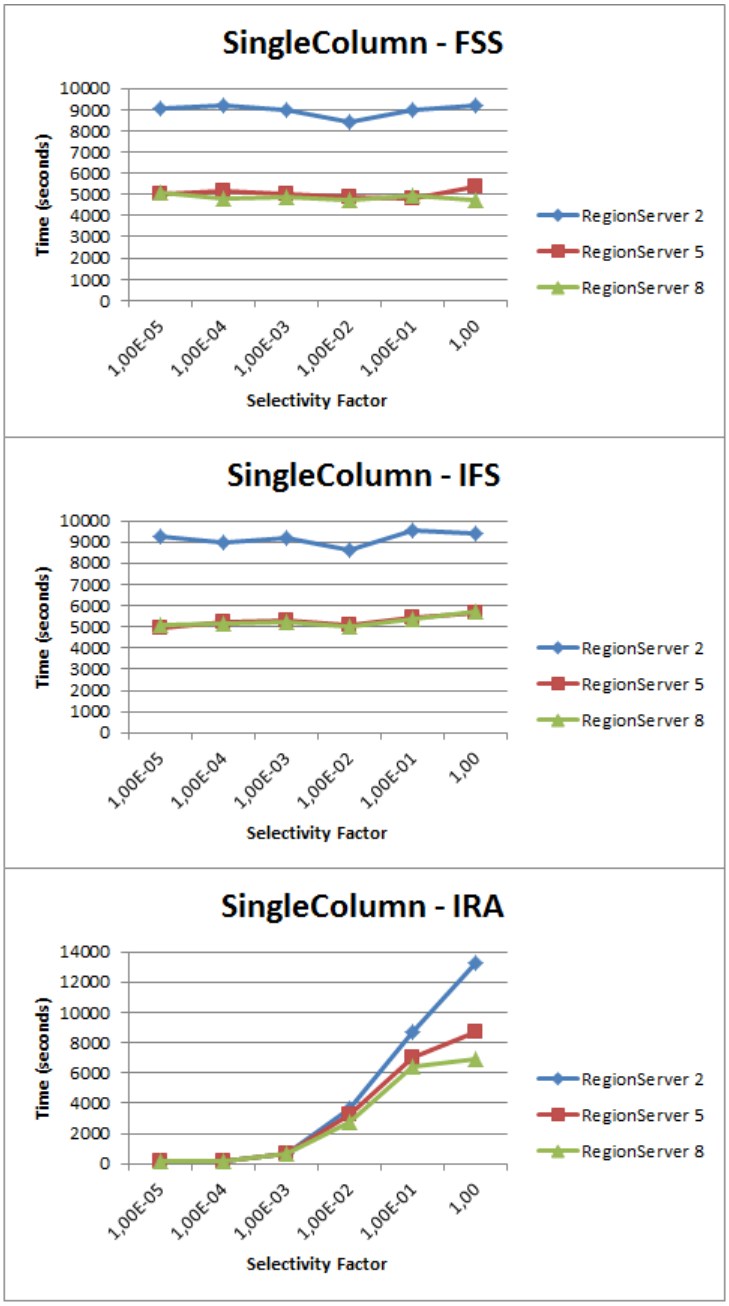


Figura 8.3-8: Comparativa del temps respecte el número de RegionServers i *SingleColumn*

Figura 8.3-9: Speed up obtingut a l'afegir RegionServers amb *SingleColumn*

Scale Factor	6
Fragmentació vertical	*SingleColumn*
RegionServers	2

En aquesta figura es veu clarament que amb poca fragmentació vertical, el guany dels RegionServers està al afegir el nombre just de màquines. Sempre s'hi poden afegir més, però aleshores el guany obtingut no és tan fort. Almenys aquest és el comportament pels algorismes FSS i IFS.

Tal i com comença a ser habitual, el *speed up* obtingut per l'IRA és un pèl diferent. Aparentment sembla que el nombre de RegionServers comença a tenir un efecte directe sobre l'IRA a mesura que s'augmenta el factor de selecció. Aquest cas reforça la justificació feta anteriorment en la que s'expliquen els motius pels quals un accés aleatori queda exempte del nombre de màquines, i a mesura que el factor de selecció augmenta, també ho fan les lectures seqüencials i per tant l'*speed up* provinent del nombre de RegionServers es fa més clar.

Tot això plegat porta cap a la discussió de si un disseny de la base de dades hauria de ser mitjançant, o bé la força bruta en termes de paral·lelisme, o bé mitjançant diferents

estratègies de fragmentació vertical per reduir costos de lectura. Per poder discutir aquesta altra qüestió, caldria tenir a la mà una gràfica resum contraposant ambdós *speed ups*.

Aquesta gràfica s'ha fet tot comparant els dos valors que ofereixen més *speed up*:

1. De les tres tècniques de fragmentació vertical, la que ofereix una millora més important és l'*AffinityMatrix*.

2. El millor rendiment respecte el nombre de RegionServers és 8 màquines.

Les dades que permeten afirmar aquests dos enunciats no es discuteixen en aquest apartat però es poden trobar les seves respectives gràfiques a l'annex *B Gràfiques per paràmetre*.

La idea és agafar la configuració més costosa de totes, que correspon a la mateixa configuració que la taula anterior, i modificar-la a partir dels dos paràmetres anteriors. Així es pot obtenir quina és la millora per cada cas, i visualitzar-la per tal de dur a terme les conseqüents discussions. Aleshores, la primera de les figures *8.3-10* mostra la comparativa final entre aquests dos paràmetres respecte el FSS.

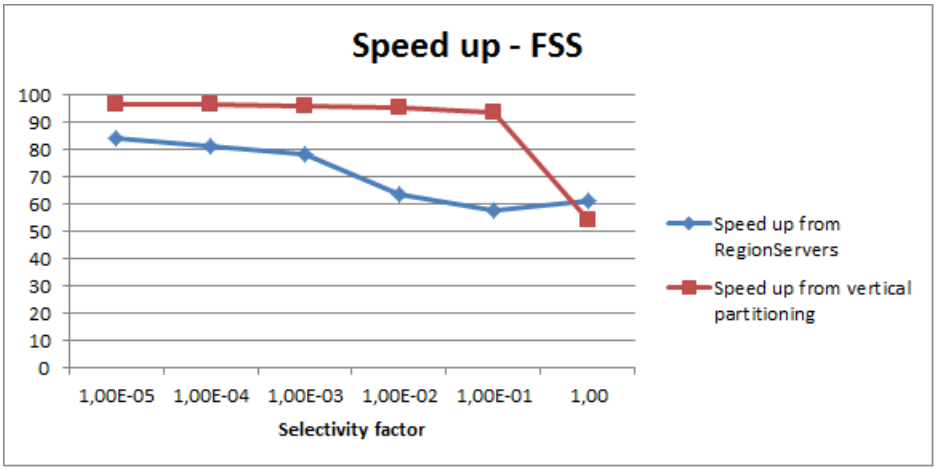


Figura 8.3-10: *Speed ups* del FSS respecte la fragmentació vertical o RegionServers

Segons aquesta gràfica, optimitzant la configuració anterior a partir d'una bona fragmentació vertical s'aconsegueixen millors *speed ups* que tan sols afegint més màquines. No obstant això, aquesta afirmació sembla que perd pes a mesura que augmenta el factor de selecció. La justificació ve donada pel cost d'enviament de les tuples al MapReduce, ja que tal i com es menciona anteriorment aquest cost es beneficia del nombre de màquines. En canvi, en els factors de selecció més petits el que s'aconsegueix és reduir les lectures que cada RegionServer ha de fer i per tant, mentre el cost d'enviar les tuples sigui acceptable amb pocs RegionServers, el *speed up* obtingut a través de la fragmentació vertical serà millor.

Aquest mateix argument s'aplica a l'algorisme IFS mostrat a la figura *8.3-11*.

En aquest cas, però, cal recordar que gràcies a l'accés a l'índex que fa l'IFS, les tuples que satisfan la selecció ja es coneixen abans d'accedir a la taula de fets i per tant no cal tractar amb els atributs de selecció. Tot això provoca que amb una fragmentació vertical on els atributs de selecció puguin estar en families diferents com l'*AffinityMatrix* causarà que:

a) el número de lectures finals sigui més petit, i

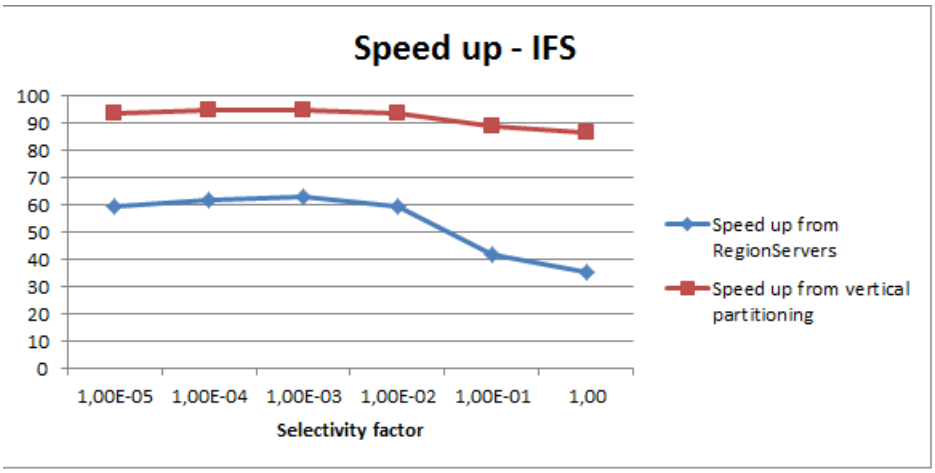


Figura 8.3-11: *Speed ups* de l'IFS respecte la fragmentació vertical o RegionServers

b) els costos d'enviament de les tuples al MapReduce també siguin més petits,

i per tant té sentit que en qualsevol factor de selecció la fragmentació vertical pugui oferir un *speed up* millor que augmentar el nombre de màquines. La discussió és que el nombre de màquines dona *speed ups* millors quan els costos de tractar amb les dades (lectura i enviament) són grossos, però en el cas d'aquest algorisme el primer accés a l'índex li permet obtenir la informació suficient com per reduir minimitzar les dades innecessàries.

Finalment, la figura *8.3-12* mostra l'*speed up* obtingut en cas de l'algorisme IRA.

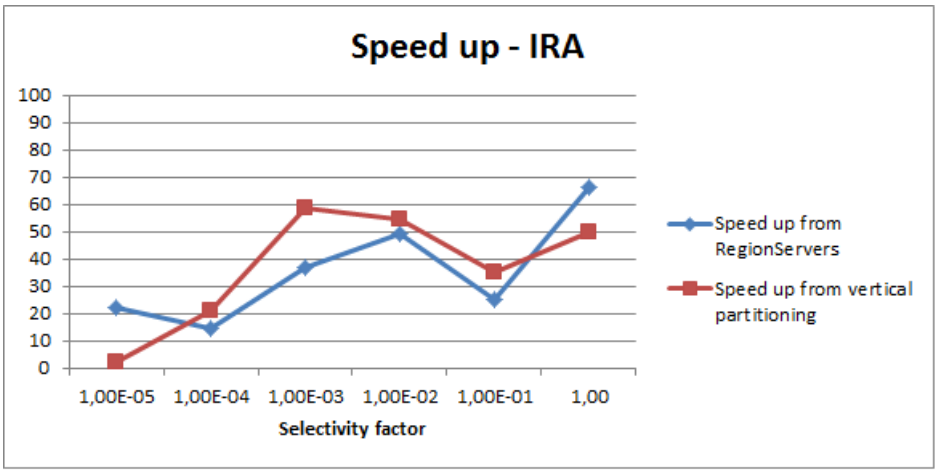


Figura 8.3-12: *Speed ups* de l'IRA respecte la fragmentació vertical o RegionServers

El detall més sorprenent d'aquesta figura és veure com l'*speed up* degut a tots dos paràmetres és similar al llarg de tota la gràfica. Això torna a justificar les hipòtesis mencionades quan s'ha presentat l'IRA respecte la fragmentació vertical i respecte el nombre de RegionServers, on s'explicitaven els motius pels quals la fragmentació vertical té un impacte molt pobre i el nombre de RegionServers un impacte nul en el rendiment de l'IRA.

El punt clau de la gràfica és quan el factor de selecció s'acosta a 1, ja que en aquest moment accessos aleatoris de l'IRA es converteixen en seqüencials i per tant el resultat final és una millora més important per part del número de màquines.

No obstant això, aquesta millora hauria d'anar més enfocada segons la justificació de l'IFS, ja que l'IRA també es beneficia de l'índex per evitar el tractament dels atributs de selecció. En canvi, el número de RegionServers és el paràmetre que ofereix un millor rendiment quan el factor de selecció s'apropa a 1, tal i com el FSS.

Per tal de donar sentit a aquesta situació, va caldre revisar els codis implementats dels tres algorismes, ja que cap de les justificacions anteriors podia donar-hi sentit. Durant aquest pas, es va descobrir que la configuració de l'IFS alterava els costos per defecte d'enviar les tuples de l'HBase cap al MapReduce. Aquesta modificació augmentava el buffer d'enviament de tuples en factor 10. Així, mentre el FSS i l'IRA envien les tuples de cent en cent, l'IFS ho fa de mil en mil. Aquest canvi no és resultat de les modificacions d'aquest projecte sinó que és heretat del codi del projecte anterior.

Malauradament, el projecte ja estava en un estat massa avançat com per repetir tot el conjunt de proves i per tant es va decidir justificar-ho a partir de l'error.

No obstant això, aquest error és important en el fet que es posa en dubte les situacions on l'IFS ofereix els millors temps d'execució, però no afecta a l'hora de construir el marc d'optimització perquè la discussió dels costos continua sent la mateixa, el que canvia és com es combinen aquests costos. En qualsevol cas, les formules de costos presentades a l'apartat *8.3.2 Formules de costos d'execució* s'han dissenyat tenint en compte aquest nou paràmetre de configuració de l'HBase i per tant és encertat concloure que ara són formules més refinades. Potser si no s'hagués produït aquest error les formules de costos finals haguessin estat menys intel·ligents.

Scale Factors

El tercer i últim paràmetre a discutir és el volum de dades. Aquest paràmetre és el més fàcil de discutir perquè tal i com es mostra a la figura *8.3-13*, el seu comportament és igual al mostrat pel nombre de RegionServers.

La discussió és molt simple, ja que el que realment ocorre amb aquests dos paràmetres és que l'un és el contrari de l'altre. Així

- donat un volum de dades fixat i variar el nombre de RegionServers, o bé
- fixar el nombre de RegionServers i variar el volum de dades.

implica produir els mateixos resultats. La conclusió és que la unitat de càlcul realment important de combinar aquests dos paràmetres és el volum de dades per RegionServer.

D'aquesta manera, reduir el nombre de RegionServers i el volum de dades a aquesta unitat implica que les discussions detallades durant la discussió del nombre de RegionServers són perfectament aplicables i per aquest motiu no s'entra en més detall en valorar aquest paràmetre. En qualsevol cas, a l'apartat *B Gràfiques per paràmetre* es poden trobar la resta de figures que donen peu a aquesta discussió.

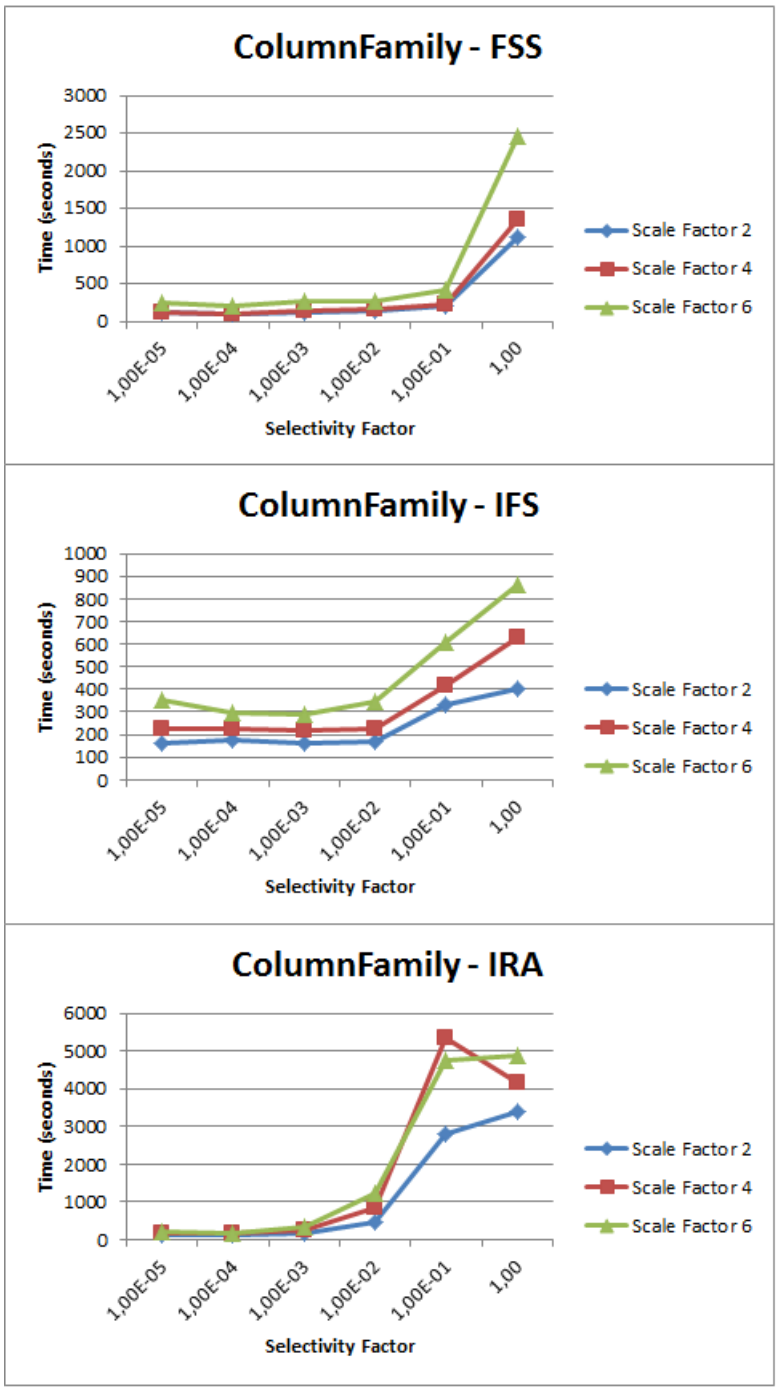


Figura 8.3-13: Comparativa del temps respecte el volum de dades i *ColumnFamily*

8.3.2 Formules de costos d'execució

Partint de les discussions observades a l'apartat anterior, les premisses a partir de les quals es construeixen les formules de costos dels algorismes són:

a) Els elements dels quals es tenen evidències clares de que afecten al rendiment dels algorismes són:

- Les lectures seqüencials de la base de dades.
- Les lectures aleatòries de la base de dades.
- El cost d'enviar les tuples de l'HBase al MapReduce (cost de **fetch** a partir d'ara).

b) Es coneix que el nombre de RegionServers i el volum de dades es combinen entre ells formant la unitat definida a l'apartat anterior: volum de dades per RegionServer.

c) Els costos d'aquests elements es calculen a partir de les variables:

- Variables relacionades amb els costos d'accés:
  - Espai que ocupa un bloc HDFS (64MB).
  - Cost de llegir un bloc HDFS.
  - Cost de fer un accés aleatori. Aquesta variable agrupa els costos de cercar una tupla indexada i de fer-hi la lectura del bloc. Aquest cost hauria de ser més petit que el cost de llegir seqüencialment de disc, ja que s'hauria de permetre indexar directament al bloc real de disc, que ocupa molt menys que un bloc HDFS de 64MB.

  Aquestes variables serveixen per computar els costos de fer lectures seqüencials de disc. Aquests costos es calculen en funció dels blocs HDFS per simular com es fa en el món relacional, ja que un bloc és la unitat mínima d'accés a disc.
- Variables relacionades amb les propietats de la taula i de la query:
  - Espai ocupat per cada un dels stores de cada RegionServer (fragmentacions verticals de la taula).
  - Espai ocupat per les fragmentacions verticals que contenen els atributs d'una query determinada.
  - Número de tuples de la taula.
  - Factor de selecció.

  L'objectiu d'aquestes és ajudar a calcular els costos finals en proporció als costos de manipular les dades necessàries per resoldre la query respecte els costos de manipular la taula sencera.
- Paral·lelisme de les taules:
  - Taula de fets per tal d'identificar quines màquines poden estar treballant en paral·lel. Es defineix com: $min(regions, RegionServers)$, on regions i RegionServers és el nombre de regions i el nombre de màquines, respectivament. La idea és calcular el paral·lelisme màxim com el número de màquines que es poden activar a l'hora: si $regions > RegionServers$, aleshores el paral·lelisme màxim és el nombre de màquines ja que cada màquina processa les regions una a una, i en cas contrari, si $regions < RegionServers$, aleshores el paral·lelisme màxim és el nombre de regions ja que això indicarà que hi han màquines sense dades que per tant no processaran res.

  - Taula de l'índex. Es defineix com $min(regions, RegionServers, |slicers|)$, on $|slicers|$ indica el número de clàusules de selecció. La idea és la mateixa que l'anterior, però tenint en compte també que el paral·lelisme màxim també depèn del nombre de seleccions (veure apartat *5.3 Accés als índexs. Selecció*).
  - Taula temporal de l'IRA. Es defineix com:

$$P_{temp} = \frac{\text{espai que ocupa la taula temporal}}{\text{espai que ocupa un memstore}}$$

    Aquesta formula és purament una estimació, ja que assumeix que el valor del denominador és constant, quan en realitat aquest depèn també del número de regions ja creades (veure apartat *4.2.2 Fragmentació horitzontal: Autosharding*). Però en qualsevol cas, com que aquesta taula temporal és molt petita, es pot assumir que el nombre de regions creades també és molt petit.

- Variables relacionades amb el cost d'accedir a l'índex:
  - Número de blocs que ocupa la sortida de l'índex.
  - Paral·lelisme de l'índex i de la taula temporal vists al punt anterior.
- Variables relacionades amb el cost de fetch:
  - Cost de transportar una sola tupla.
  - Cost de transportar un paquet de tuples.
  - Número de tuples per paquet.

  Es diferencia entre transportar una sola tupla i transportar un paquet de tuples perquè les tuples recuperades a través d'un accés seqüencial s'envien en paquets cap al MapReduce, mentre que els tuples recuperades via accés aleatori són enviades una a una.
- Un marge d'error tal i com existeix en el món relacional. Principalment degut al fet que és obvi que existeixen altres factors que afecten al cost final de cada algorisme però que no es tenen en compte en aquestes formules perquè són molt difícils de valorar. Alguns exemples d'aquests costos podrien ser costos de comunicació, d'heterogeneïtat de les màquines del clúster, etc.

La importància de tenir en ment cada un d'aquestes variables és per tal de garantir els punts a) i b) de la enumeració anterior. Així, les variables relacionades amb els blocs HDFS permeten identificar el volum de dades de la taula, i juntament amb les variables de paral·lelisme, es refinen per indicar el volum de dades per RegionServer. Aleshores, els costos de lectura es calculen directament sobre el resultat anterior en funció de les variables de query i de taula.

A més, els algorismes IFS i IRA efectuen primerament un accés a l'índex que consisteix en una tasca MapReduce i per tant cal mantenir variables que permetin determinar aquest cost. Com que l'índex és igualment una taula HBase, les discussions que aplica són les mateixes que la taula anterior.

La taula següent mostra un resum de cada una d'aquestes variables:

S_B	Espai ocupat per un bloc
D_B	Cost de llegir un bloc
D_R	Cost de fer un accés aleatori
S_{ij}	Espai ocupat per l'store j de la region i
Q_{ij}	Espai ocupat pels atributs de l'store j de la region i amb els que ha de tractar la query Q
FS	Factor de selecció
$\|T\|$	Cardinalitat de la taula de fets. Número de tuples
P_{table}	Paral·lelisme de la taula de fets. Es defineix $min(regions, RegionServers)$
P_{index}	Paral·lelisme de l'índex. Es defineix $min(regions, RegionServers, \|slicers\|)$
$\|slicers\|$	Nombre de clàusules de la selecció
$\overline{B_{slicers}}$	Mitjana del nombre de blocs que ocupa cada entrada de l'índex
P_{temp}	Paral·lelisme de la taula temporal
B_{temp}	Número de blocs de la taula/fitxer temporal
f_{range}	Cost de fetch per a un paquet de tuples
f_{row}	Cost de fetch per a una tupla
C	Número de tuples per paquet que s'envien durant el cost de fetch
ϵ	Marge d'error

Costos del FSS

El FSS és l'algorisme encarregat de dur a terme una lectura complerta de la taula, aleshores la formula del seu cost és:

$$FSS = \frac{\frac{\sum_{i}^{region}\sum_{j}^{store} S_{ij}}{S_B} D_B + F}{P_{table}} \pm \varepsilon$$

On el cost de fetch F:

$$F = \frac{\sum_{i}^{region}\sum_{j}^{store} Q_{ij} FS}{C} f_{range}$$

El cost d'aquest algorisme és potencialment una lectura seqüencial. El primer cost correspon a llegir el número de blocs HDFS que toqui. Així, amb una estratègia de fragmentació vertical *SingleColumn* aquest valor serà més alt perquè els stores ocuparan més degut a que la lectura dels atributs de la query implicarà també la lectura de tota la resta d'atributs. En canvi, amb *ColumnFamily* aquest valor correspondrà exactament al número de blocs que ocupa el conjunt d'atributs a llegir.

De manera semblant, el cost de fetch es calcula en funció del volum de dades útils, que es podria definir com el producte entre l'espai ocupat pels atributs que calen enviar al MapReduce i el factor de selecció (per tal d'obtenir la corresponent proporció). Finalment només cal dividir i multiplicar pels costos d'enviar aquestes tuples per paquets.

Costos de l'IFS

El cost de l'IFS ve donat per la suma del cost d'accedir a l'índex i el cost d'accedir a la taula de fets:

$$IFS = IFS_{index} + IFS_{table} \pm \varepsilon$$

Per una banda, el cost d'accedir a l'índex és defineix de la següent manera:

$$IFS_{index} = \frac{|slicers|(D_R + \overline{B_{slicers}}D_B)}{P_{index}} + B_{temp}D_{hdfs}$$

Aquest formula és en realitat una petita simplificació, ja que s'assumeix que cada clàusula de selecció pot ésser amb només un valor. Així ho són les clàusules de les queries amb les que s'han llençat els experiments, però cal esmentar que per tal de fer-ho més exacte no s'hauria de fer aquesta assumpció. Amb la simplificació, el cost total de llegir de l'índex es calcula del producte entre el número de clàusules i la mitjana de l'espai ocupat per cada clàusula. Per fer la formula en la seva versió completa caldria substituir aquesta part de l'equació per un sumatori del nombre total de blocs ocupats per cada parella clàusula i valor.

Aleshores, aquest primer cost de la formula correspon a la lectura de l'índex. Per aquest motiu cal tenir en compte el número de clàusules, ja que aquest valor indica el número d'entrades que cal llegir de l'índex. Així, a continuació només cal multiplicar pel cost de llegir cada entrada de l'índex (un cost aleatori per trobar la primera dada més accessos seqüencials) i dividir pel nombre de màquines actives en paral·lel.

El segon cost correspon al cost d'escriure la sortida de l'índex a l'HDFS.

Per altra banda, el cost d'accedir a la taula de fets amb l'IFS:

$$IFS_{table} = \frac{D_R + \frac{\sum_{i}^{region}\sum_{j}^{store} S_{ij}}{S_B}\frac{|range|}{|T|}D_B + F}{P_{table}}$$

On el cost de fetch F:

$$F = \frac{\sum_{i}^{region}\sum_{j}^{store} Q_{ij}\frac{|range|}{|T|}}{C} f_{range}$$

De manera semblant als costos del FSS, el cost de llegir de la taula de fets amb l'IFS és potencialment una lectura seqüencial. Les diferències més fonamentals amb el FSS són:

- La lectura seqüencial no es fa de tota la taula, sinó que s'acota al rang de tuples que comprenen la primera i última tupla de la selecció. Així per exemple, una query on la primera tupla que satisfà la selecció és la tercera i la última que la satisfà és la dècima llegirà el rang de tuples entre la segona i la dècima, independentment de si la resta de tuples d'aquest rang compleixen la selecció o no. La cardinalitat d'aquest rang és el que es denota com a $|range|$ (que a l'exemple anterior valdria 8). Aleshores, l'aproximació ve donada pel producte del nombre total de blocs de la taula de fets per la proporció de tuples dins d'aquest rang respecte el nombre total de tuples.

- Durant el càlcul del fetch, el cost $\sum_{i}^{region}\sum_{j}^{store} Q_{ij}$ és lleugerament més petit a l'IFS, perquè en aquest cas no s'han d'incloure els atributs de selecció

Malauradament, dur a terme el comput de costos de l'IFS amb aquestes formules implica que cal mantenir un histograma per tal de conèixer el valor de la variable $|range|$. Mantenir un histograma d'aquest tipus en una base de dades és un cost important i per tant, per tal d'automatitzar la tasca de decisió de l'algorisme, no es pot suposar que aquest histograma hi serà. Aleshores, la suposició ha de ser que en qualsevol cas cal llegir tota la taula. D'aquesta manera, el cost d'accedir a la taula de fets es converteix en la següent formula:

$$IFS_{table} = \frac{\frac{\sum_{i}^{region}\sum_{j}^{store} S_{ij}}{S_B} D_B + F}{P_{table}}$$

I el cost de fetch passa a ser:

$$F = \frac{\sum_{i}^{region}\sum_{j}^{store} Q_{ij}}{C} f_{range}$$

On es veu com la formula de costos de l'IFS respecte el FSS empitjora clarament:

1. El cost IFS_{table} es converteix en el mateix que el FSS.
2. El cost de fetch ara és més gran que al FSS quan $FS < 1$:

$$\frac{\sum_{i}^{region}\sum_{j}^{store} Q_{ij}}{C} f_{range} > \frac{\sum_{i}^{region}\sum_{j}^{store} Q_{ij} FS}{C} f_{range}$$

3. A més cal recordar que existeix el cost d'accedir a l'índex IFS_{index}

No obstant això, la variable C en els experiments d'aquest projecte és més gran a l'IFS per un error en el codi que no es va tenir en compte. A l'IFS les tuples s'envien en paquets de mil en mil, mentre al FSS s'envien de cent a cent. Aquesta és la situació comentada a l'apartat *8.3.1 Discussió general* durant la comparativa dels *speed ups*.

De totes maneres, aquesta discussió forçosament no implica que l'IFS mai pugui oferir millors resultats que el FSS. Tenint un histograma a mà l'IFS podria ser l'algorisme escollit en més d'una ocasió. Principalment quan les tuples a seleccionar tinguin una localitat especialment notable, ja que d'aquesta manera es minimitzarien els costos de fer les lectures seqüencials. Això es podria aconseguir mitjançant un disseny de la key correcte, ja que recordem que a l'HBase les tuples s'ordenen lexicogràficament.

Costos de l'IRA

De la mateixa manera que l'IFS, els costos d'execució de l'IRA venen donats per l'accés a l'índex i l'accés a la taula:

$$IRA = IRA_{index} + IRA_{table} \pm \varepsilon$$

I el cost d'accedir a l'índex té la mateixa forma que en el cas de l'IFS amb la diferència que ara cal escriure a l'HBase:

$$IRA_{index} = \frac{|slicers|(D_R + \overline{B_{slicers}} D_B)}{P_{index}} + 2B_{temp} D_{hbase}$$

En aquest cas, el cost d'escriure el resultat de l'accés a l'índex es multiplica per dos perquè cal una escriptura addicional d'aquest resultat per tal de materialitzar els rangs identificats durant l'etapa de preprocessament.

Aleshores, el cost final d'accedir a la taula de fets per l'IRA és:

$$IRA_{table} = \frac{(FS|T| - (\overline{|ranges|} - 1)ranges)D_R + \frac{\sum_{i}^{region} \sum_{j}^{store} S_{ij}}{S_B} \frac{\overline{|ranges|}ranges}{|T|} D_B + F}{P_{temp}}$$

On el cost de fetch F depèn del factor de selecció:

- Si $FS < 1$ aleshores:

$$F = \sum_{i}^{region} \sum_{j}^{store} Q_{ij} FS f_{row}$$

- Si $FS = 1$ aleshores:

$$F = \frac{\sum_{i}^{region} \sum_{j}^{store} Q_{ij} FS}{C} f_{range}$$

Les variables $\overline{|ranges|}$ i $ranges$ indiquen, respectivament, la mitjana del nombre de tuples que hi ha cada rang resultant del preprocessament i el nombre de rangs trobats. Per exemple, en un base de dades amb 10 tuples, accedir a la segona i tercera tupla, a la sisena i a la setena, i a la novena, totes dues variables prendrien valor 2, ja que en total tenim dos rangs d'accés (segona-tercera, sisena-setena), i cada rang conté dues tuples. Dit això, és fàcil veure com la formula de cost de l'IRA simplement és la suma dels accessos aleatoris que cal fer quan les dades no estiguin agrupades i el cost de les lectures seqüencials quan les dades sí ho estiguin.

La variable F es trenca en una funció definida a parts per tal de diferenciar quan els accessos aleatoris envien les tuples una a una cap al MapReduce, i quan les lectures seqüencials ho fan en paquets.

En qualsevol cas, les variables $\overline{|ranges|}$ i $|ranges|$ haurien de ser obtingudes a partir d'un histograma, i tal i com es justifica en les formules de l'IFS, caldria reajustar les formules per evitar aquestes variables.

Així doncs, el cost d'accedir a la taula de fets es podria refer de la següent manera:

- Si $FS < 1$ aleshores:

$$IRA_{table} = \frac{FS|T|D_R + F + J}{P_{temp}}$$

- Si $FS = 1$ aleshores:

$$IRA_{table} = \frac{\frac{\sum_{i}^{region} \sum_{j}^{store} S_{ij}}{S_B} D_B + F + J}{P_{temp}}$$

El cost de fetch es mantindria igual i l'objectiu de particionar el cost de l'IRA en una funció definida a parts seria aconseguir aproximar les formules antigues el màxim possible tot tenint en compte el factor de selecció, ja que durant l'execució dels experiments es va descobrir que no és fins que el factor de selecció és molt proper a 1 que el valor de la variable $|\overline{ranges}|$ no és prou gran com per parlar d'accessos seqüencials.

9 Conclusions

9.1 Conclusions tècniques

En aquest apartat es presenten les conclusions tècniques derivades del treball dut a terme amb les tecnologies emprades durant tot un any. Són principalment la conseqüència de les limitacions presentades per aquestes tecnologies a l'hora d'implementar accessos més sofisticats, ja que en la majora de situacions una part important del cost d'execució correspon al fet de no poder gestionar certes característiques d'aquests nous tipus d'accés de forma nativa.

Inicialment, cal intentar rebatre alguns consells de disseny esmentats a la documentació oficial de l'HBase. En aquesta documentació, es recomana no fer un disseny de cap taula amb més de tres families. A partir de l'experiència d'aquest projecte, es dedueix que aquesta recomanació està pensada per evitar que masses families provoquin que no es produeixin els splits necessaris (veure apartat *8.1 Decisió i planificació dels paràmetres*) per tal d'explotar el paral·lelisme ofert pel sistema. No obstant això, en aquest projecte s'ha vist com no tot és una qüestió de paral·lelisme, sinó que es poden desenvolupar mètodes d'accés més intel·ligents que es beneficien més del disseny físic de la base de dades, que no pas del paral·lelisme afegit via força bruta. Aquest és el cas de l'IFS i l'IRA, on una bona estratègia de fragmentació vertical hi juga un paper molt important ja que podria eliminar completament el cost degut a la lectura de les dades que no aporten cap valor a aquests algorismes.

Amb una gestió nativa dels recursos emprats per cada un d'aquests algorismes s'aconseguiria:

1. Eliminar les latències d'haver de dur a terme operacions intermitges de forma externa a la tecnologia, és a dir a nivell de programador.

2. Maximitzar el profit resultant de poder encadenar cada una dels processos d'aquests algorisme de manera interna a la tecnologia i per tant aprofitar-se'n de tots els seus recursos.

La primera d'aquestes millora ve justificada per la necessitat de gestionar els índexs de forma diferent a les taules d'usuari. En aquest sistema, els índexs han hagut d'implementar-se mitjançant taules d'usuari, però és innegable que tractar aquests índexs en una espai de noms diferents, amb d'altres propietats segurament hagués conduït a un èxit rotund dels algorismes implementats. Principalment, pel fet que un índex ocupa un volum d'espai molt més petit que una taula d'usuari, i per tant caldria aplicar-hi polítiques de region split diferents que no es basessin tant en el volum de dades ocupat. La qüestió seria afegir un toc de paral·lelisme als índexs.

Amb aquesta funcionalitat implementada es reduiria el cost d'accedir a l'índex, però l'objectiu seria sobretot minimitzar els costos de les operacions entre l'índex i la taula de fets. Aquests guanys serien principalment per l'IRA, ja que s'aconseguiria reduir el temps invertit en gestionar la taula temporal a la vegada que podria incrementar el seu paral·lelisme per tal d'explotar millor els recursos d'un sistema concurrent i distribuït.

De cara a millorar l'IFS, i seguint la mateixa línia que la conclusió anterior, la millora vendria precisament pel fet d'implementar el bitmap de forma nativa. D'aquesta manera, es podria executar l'IFS en un sol MapReduce amb una doble entrada: la parella key-value que ja rep normalment, i el bit resultant d'aplicar les operacions de bitmap. Aquí ja es parla d'operacions de bitmap, ja que implementar el bitmap de forma nativa també

permetria combinar els atributs de selecció tot aplicant les operacions pròpies de bitmap per tal de trobar finalment quines tuples satisfan totes les clàusules de selecció.

Així doncs, els costos de l'IFS caurien en picat ja que el cost d'executar una primera tasca MapReduce desapareixeria completament. De la mateixa manera, es podria anar un pèl més enllà i aprofitar aquest mateix bitmap natiu per tal de fer la selecció a nivell de lectura de la taula de fets tal i com fa el FSS, però mitjançant un bitmap.

Per tant, la conclusió tècnica final és molt clara: el paral·lelisme via força bruta no és la solució única malgrat el paral·lelisme sigui necessari per contrarestrar l'excés del volum de dades per màquina. La solució alternativa aleshores passa per un bon disseny físic de la base de dades i el desenvolupament de mètodes d'accés més intel·ligents que es beneficien d'aquest disseny. Aquests mètodes poden ser desenvolupats per l'administrador de la base de dades o poden ser implementats de forma nativa, però en qualsevol cas, cal que internament a la base de dades s'ofereixi el suport necessari per tal de dur-los a terme el més eficaçment possible. En cas contrari, la situació serà la mateixa que en aquest projecte, on els elements intermitjos que han de permetre dur a terme aquests nous mètodes d'accés no podran ser desenvolupats sota les millors condicions tecnològiques, i per tant estaran sempre en desavantatge. Tot i així, en aquest projecte es veu clarament com malgrat i aquests desavantatges, l'IFS i l'IRA són capaços d'oferir millor rendiment que el FSS sota certes condicions. Això indica que si estiguessin desenvolupats òptimament de forma nativa, tal i com ho està el FSS, els resultats finals podrien ser immillorables i aleshores aquesta conclusió final podria tenir unes bases de justificació molt més fortes.

9.2 Conclusions personals i treball futur

La realització d'aquest projecte m'ha aportat una experiència molt positiva com a futur enginyer. Tornant la vista enrere me'n adono de tot el procés de maduració que he viscut durant aquest últim any de dedicació. Penso en el que he après i en el que no, en les decisions encertades i en les equivocades, en els objectius complerts i en els que no, i malgrat el resultat final sigui més que satisfactori, m'agradaria tornar a enrere per tornar a començar aquest projecte amb un enfoc molt diferent en alguns punts.

Això és així perquè l'elaboració d'aquest projecte mai ha tingut un següent pas massa ben definit. El motiu és que quan vaig iniciar aquest projecte partia de les experiències d'un projecte anterior on alguns objectius no s'havien pogut completar degut a errors que van aparèixer al manegar amb les tecnologies emprades en aquest mateix projecte i amb entorns distribuïts. Aleshores, el primer objectiu d'aquest projecte va ser aconseguir superar aquestes dificultats. Afortunadament, les experiències d'aquest projecte anterior van deixar un fil de possibles causes que podien estar generant els errors. Aleshores, això va permetre donar-li una volta més a les parts del sistema que no van funcionar en el seu moment (sobretot durant els procés d'inserció de grans volums de dades), i fer noves implementacions amb un enfoc totalment diferent.

Aquests nous enfocs van ser un encert indiscutible, ja que les noves implementacions van permetre inserir volums de dades per sobre dels esperats. En aquest moment, es van poder definir molt més clarament els objectius del projecte que apareixen a l'apartat *1.4 Objectius* d'aquesta memòria. No obstant això, ara calia fer front als problemes presentats per aquelles parts de les tecnologies d'on desconeixíem el comportament. Aquí, vaig trobar a faltar que la documentació oficial fos més extensa i inclogués detalls sobre el funcionament intern de les tecnologies, ja que quan es presentava un nou problema es feia difícil identificar les possibles causes. Per sort, en molts casos aquests problemes eren generalitzats amb d'altres usuaris i la informació extreta de fonts d'informació no oficials

es va fer vital. En molts altres casos, els problemes apareguts eren massa concrets a aquest projecte i per tant en aquestes situacions va caldre buscar alternatives o construir solucions pròpies, que moltes vegades van significar haver de capficar-se amb el codi font de les tecnologies. No obstant tot això, crec que tot aquest conjunt de complicacions són les que realment han fet que l'aprenentatge adquirit al llarg de tot aquest projecte hagi estat tan alt, ja que si per contra tot hagués funcionat sense cap més complicació, l'aprenentatge hagués estat purament rutinari. D'aquesta manera, he adquirit molts més coneixements, tan sobre les tecnologies emprades com sobre conceptes més generals de bases de dades.

L'objectiu de discutir tots els problemes presentats al llarg del projecte és poder arribar a la reflexió que on potser no vaig encertar va ser en la metodologia triada.

La primera fase del projecte va consistir en la implementació del generador (apartat *7.1 Etapa I: Població de l'HBase. Generador*) i degut a la urgència de voler obtenir els primers resultats d'inserció el més ràpid possible vaig decidir seguir una metodologia àgil, on jo mateix em donava el feedback per tal de valorar cada una de les funcionalitats desenvolupades. A fi de comptes, jo mateix era l'usuari final de l'aplicatiu. Malauradament, crec que aquesta decisió va ser errònia pels següents motius:

- És cert que els primers resultats es van obtenir ràpid, però no va ser fins que és va finalitzar aquest generador que es va poder fer una valoració complerta de les insercions.
- No vaig aconseguir satisfer l'objectiu de fer servir aquesta metodologia i el temps dedicat finalment a aquesta part del projecte va ser massa elevat (aproximadament cinc mesos).

A més, durant la fase posterior d'execució de les proves es va fer imprescindible modificar algunes parts d'aquestes generador i en aquests moments vaig trobar a faltar que aquesta implementació no estigués coberta per una fase d'anàlisi de requisits i de disseny molt més forta. Combinant aquestes reflexions amb els coneixements adquirits al llarg de la carrera, crec que hauria d'haver triat la metodologia fent l'exercici que ja vaig fer a l'assignatura de PESBD (Projecte d'Enginyeria del Software i Bases de Dades). Aquest consisteix en triar una metodologia basada en iteracions (per exemple RUP) i aleshores decidir quines iteracions són les que realment interessen pel projecte i quines no. En aquest sentit, les iteracions triades haurien de tenir per objectiu maximitzar les etapes de disseny del software però sense comprometre excessivament el temps de dedicació estimat.

En canvi, la metodologia àgil triada va ser molt encertada per a la segona i tercera etapa d'implementació: modificacions dels algorismes i automatització de les proves. Això es justifica per mitjà del fet que a diferència del generador, aquestes dues etapes no consistien en desenvolupar un nou aplicatiu des de zero sinó en adequar el codi ja implementat. No obstant això, cal mencionar que aquestes dues etapes en el projecte anterior es van desenvolupar a partir d'una primera iteració de disseny molt forta, i possiblement la conseqüència va ser justament que pogués capficar-me amb el codi directament i per tant la metodologia àgil fos més encertada. Cal remarcar que la documentació obtinguda del projecte anterior, sobretot en la part de desenvolupament dels algorismes d'accés, ha estat excel·lent i per tant clau en que les dues últimes etapes d'implementacions es poguessin dur a terme tan eficaçment.

Per altra banda, la posada en marxa de les proves tampoc va ser una tasca trivial. Tant la valoració de amb quins paràmetres calia experimentar, així com de quina forma calia fer-ho per tal d'ajustar-se al temps disponible no es van poder completar satisfactòriament fins que no es va acabar d'entendre el funcionament intern de les tecnologies. De totes

maneres, durant aquesta part del projecte crec que l'error més important va ser l'ambició. El fet que aquest projecte es convertís en una línia d'investigació i per tant els resultats obtinguts tinguessin l'objectiu d'anar cap a una publicació va fer que es volgués maximitzar l'exhaustivitat dels experiments. Així, fer les proves tan exhaustives com fos possible va conduir a que aquests experiments requerissin de més temps que unes proves més simples. Aleshores, segurament l'alternativa hagués estat, en comptes de fer proves més simples, definir de forma molt més acotada els objectius d'aquest projecte per tal de discernir més estrictament quina part d'aquesta línia de recerca inclou aquest projecte i quina no.

Tota aquesta discussió ens porta a la valoració dels objectius satisfets. Personalment estic més que satisfet, ja que penso que en línies generals aquests s'han complert. Potser per acabar d'arrodonir els objectius ha faltat valorar paràmetres que es van plantejar com a objectius, però que degut als arguments del paràgraf anterior no ha estat possible. Aquests paràmetres han estat:

a La compressió.

b La replicació.

No obstant això, l'objectiu principal de desenvolupar una eina de decisió que permeti decidir sota quines condicions cal fer servir un algorisme o un altre s'ha complert. Potser les formules de costos presentades a l'apartat *8.3.2 Formules de costos d'execució* són una simplificació de la realitat, ja que en una execució MapReduce hi ha molts més factors que determinen el cost final d'execució, però són sens dubte una primera aproximació. El refinament final d'aquests formules, així com els experiments dels objectius no complerts en aquest projecte encara hi són presents i, malauradament ja no puguin formar part d'aquest projecte degut a la falta de temps, es continuen tenint en compte en els resultats finals de la consegüent publicació. Per tant, en certa manera, aquest projecte com a tal s'ha convertit en una petita simplificació del treball total d'aquesta línia d'investigació però els coneixements i les experiències adquirides hi són, malgrat no apareguin en aquest projecte.

Finalment, no voldria tancar aquest apartat de conclusions personals sense mencionar les experiències viscudes els últims tres mesos d'aquest projecte, en els quals el fet que aquest projecte es convertís en una línia d'investigació m'ha permès formar part d'un equip de recerca dins de la universitat. Les vivències obtingudes durant aquesta etapa m'han fet replantejar-me el meu futur professional per tal de decidir el meu futur més immediat: si entrar a formar part del món laboral, o seguir estudiant per tal de poder dedicar-me a la recerca.

En resum, l'elaboració d'aquest projecte m'ha fet madurar molt com a futur enginyer:

- els coneixements adquirits després de treballar tot un any en aquest projecte,
- l'enriquiment d'eines per trobar solucions i alternatives quan han aparegut problemes, i
- el sentit crític necessari per tal de convertir les dades dels resultats dels experiments en informació útil

han estat indiscutiblement els tres pilars de l'aprenentatge obtingut. A més, gràcies a aquest projecte he tingut la oportunitat de viure la universitat des d'un punt de vista diferent al que he viscut com a estudiant. Així doncs, malgrat les dificultats, l'esforç i la dedicació invertida no podien haver deixat sensacions més positives. Ara només falta

esperar que aquests coneixements adquirits em serveixin de cara al meu futur professional i em permetin dedicar-me a allò que realment m'agrada de la informàtica: les bases de dades.

Treball futur

La realització d'aquest projecte ha estat elaborar un marc d'optimització per a l'execució de queries. Partint des d'aquesta línia es fa difícil pensar en formes de continuar amb aquest treball. No obstant això, això no vol dir que no hi hagi cap mena de treball futur, però sí és cert que se li hauria de donar un enfoc un pèl diferent.

Actualment, tota aquesta onada de bases de dades anomenada NOSQL està obrint les portes a un món totalment desconegut i per tant ara mateix la feina de recerca en aquest àmbit es fa infinita. En aquest projecte s'ha comprovat com, malgrat es vulgui vendre que tot és una qüestió de força bruta en termes de paral·lelisme, un bon disseny i sobretot una volta de cargol més per trobar tècniques d'accés més intel·ligents són bàsiques per tal d'exprimir el màxim rendiment possible a aquestes noves de bases de dades.

Dit això, queda clar que de treball futur en queda i molt, però ha d'encaminar-se cap a altres línies d'investigació.

Possiblement el treball futur més immediat després d'aquest projecte és ampliar l'optimitzador per tal que la seva entrada no sigui tan sols el disseny físic de la base de dades, sinó també altres paràmetres que poden ser definits *on-the-fly* durant l'execució i així els configuri per tal d'exprimir el màxim rendiment. Un exemple d'aquest tipus de paràmetres és el nombre de maps o de reduces d'una tasca MapReduce.

Una altra línia de treball futur podria ser com aplicar les operacions bàsiques de cubs de dades (veure apartat *2.4 Cub de dades*) en un algorisme MapReduce. De totes maneres, potser també és encertat dur a terme un projecte per tal de validar sí en aquest nou món encara té sentit parlar d'esquemes en estrella, dimensions i cubs de dades, ja que un concepte molt relacionat amb el món no relacional és la denormalització de les dades, i aquest és un terme en certa manera contradictori amb un esquema en estrella. Així, potser en una base de dades com l'HBase els cubs de dades haurien de construir-se a partir d'un tipus d'esquema diferent.

En definitiva, el treball futur ve donat pel fet que aquest és un món totalment desconegut i per tant cal fer una tasca de recerca molt forta en tots els sentits per tal de poder donar una resposta encertada als requisits que actualment demanen les bases de dades.

10 Planificació i costos

10.1 Planificació

10.1.1 El projecte va a publicació

Aquest projecte es va iniciar el 21/12/2012. La idea inicial era dur a terme un projecte amb una duració aproximada d'uns deu mesos: iniciar i inscriure el projecte al mes de gener i començar a treballar en les primeres implementacions durant el quadrimestre de primavera, matricular-lo al mes de juliol i executar les proves durant els mesos d'estiu, i així poder presentar a mitjans de tardor. Per aquest motiu, les planificacions que es mostren als apartats següents contemplen només els mesos d'estiu i tardor, ja que es quan es va matricular el projecte.

No obstant això, la posada en marxa de les implementacions dutes a terme durant els mesos de primavera van permetre ampliar l'àmbit de treball d'aquest projecte i per tant es va decidir estendre'l per tal de cobrir objectius més ambiciosos. Tant és així, que durant els mesos d'estiu se'm va proposar ampliar el projecte per tal convertir-lo en una línia de recerca. D'aquesta manera, durant els últims tres mesos de feina se'm va fer un contracte temporal per tal de dur a terme aquest projecte amb una dedicació molt més orientada al món de la investigació, a la vegada que el conjunt d'experiments es va fer de manera molt més exhaustiva. L'objectiu final és treure una publicació de tota aquesta recerca, que inclou tant els resultats presentats en aquest projecte com d'altres que no s'inclouen.

Així doncs, aquest canvi tan radical en el projecte és la principal justificació de que la planificació inicial hagi perdut tot tipus d'importància, ja que:

- Per una banda el projecte, que estava pensat per durar deu mesos, es va allargar a tretze mesos en total.
- Per l'altra perquè segons es guanyava coneixement sobre el funcionament intern de les tecnologies, més capacitat es guanyava per tal d'enfocar els experiments d'una forma molt més exhaustiva.

De totes maneres, les diferències entre la planificació inicial i la realitzada també han vingut donades per altres complicacions que inicialment no es van tenir tan en compte (per exemple el temps d'espera en les cues de treball del clúster del DAC). Però el motiu més important ha estat el d'encarar la feina realitzada en aquest projecte cap a una publicació de recerca.

Finalment, a la figura *10.1-1* s'il·lustra la planificació durant els primers mesos de feina, abans de matricular el projecte. Es mostra d'aquesta manera perquè les planificacions són massa llargues com per mostrar-les senceres i per tant es parteixen. Aquesta primera part de la planificació és la mateixa tant en la planificació inicial com en la planificació final.

10.1.2 Planificació inicial

La planificació inicial tenia per objectiu finalitzar el projecte a finals del mes d'octubre. No obstant això, pels motius presentats a l'apartat anterior, no es va poder complir amb aquesta planificació. De totes maneres, la figura *10.1-2* mostra el diagrama de Gantt.

Tal i com es pot veure, es tracta en realitat d'una planificació molt metòdica. Es tractava d'executar les proves corresponents a cada paràmetre mentre s'avaluaven els resultats de les proves anteriors. La idea de fer-ho així era aprofitar al màxim els intervals de temps d'espera a la obtenció de resultats.

La figura *10.1-3* mostra les dates exactes de dedicació planificada per a cada una de les tasques. Les tasques en gris corresponen a la feina feta durant la fase d'inscripció del projecte abans de matricular-lo. Es mostren en gris perquè no pertanyen a la planificació en si, però són costos a tenir en compte al pressupost.

10.1.3 Planificació realitzada

La planificació finalment realitzada és la que es presenta al diagrama de Gantt de la figura *10.1-4*. Tal i com es pot veure, la diferència principal és l'augment de tres mesos de durada. No obstant això, la idea de la planificació inicial d'aprofitar els intervals de temps d'espera a la obtenció de resultats per treballar en altres tasques continua estant present en aquesta segona planificació. Especialment quan es tracta de redactar la memòria.

A més, durant el mes d'agost es veu una clara diferència entre ambdues planificacions. Mentre la planificació inicial contemplava poder executar els experiments ja durant el mes d'agost, la realitat va ser força contrària ja que les modificacions dels algorismes de consulta van ser més de les esperades degut al nou entorn distribuït. En aquest sentit, aquesta diferència mostra que en qualsevol cas la planificació inicial tampoc era del tot encertada.

Un tercer detall és que ara ja no es diferencia entre executar les proves amb un paràmetre o un altre. Això és així perquè tal i com s'indica a l'apartat *8.1 Decisió i planificació dels paràmetres* els experiments ara s'han convertit en combinacions de paràmetres per tal de fer-los més exhaustius. Si a la planificació inicial els experiments s'estructuraven en molts però petits, en aquesta planificació es defineixen com a un sol grup però molt gran. De fet, el temps de durada d'aquestes proves és d'aproximadament un mes.

La figura *10.1-5* mostra les dates exactes de dedicació planificada per a cada una de les tasques, així com les hores dedicades.

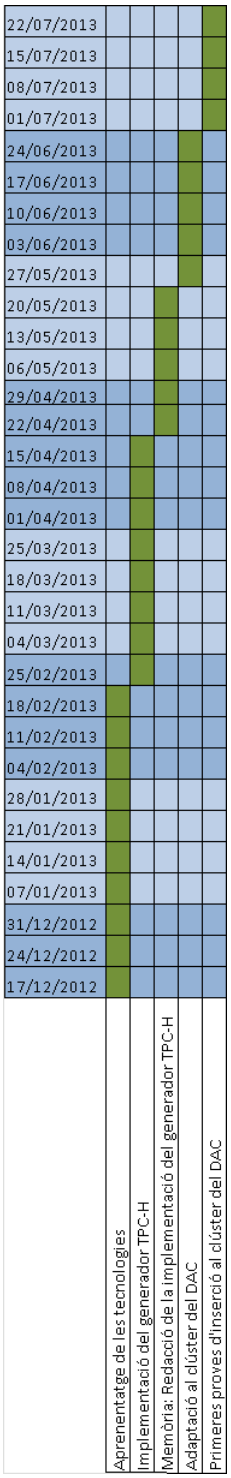


Figura 10.1-1: Diagrama de Gantt de la primera part de la planificació

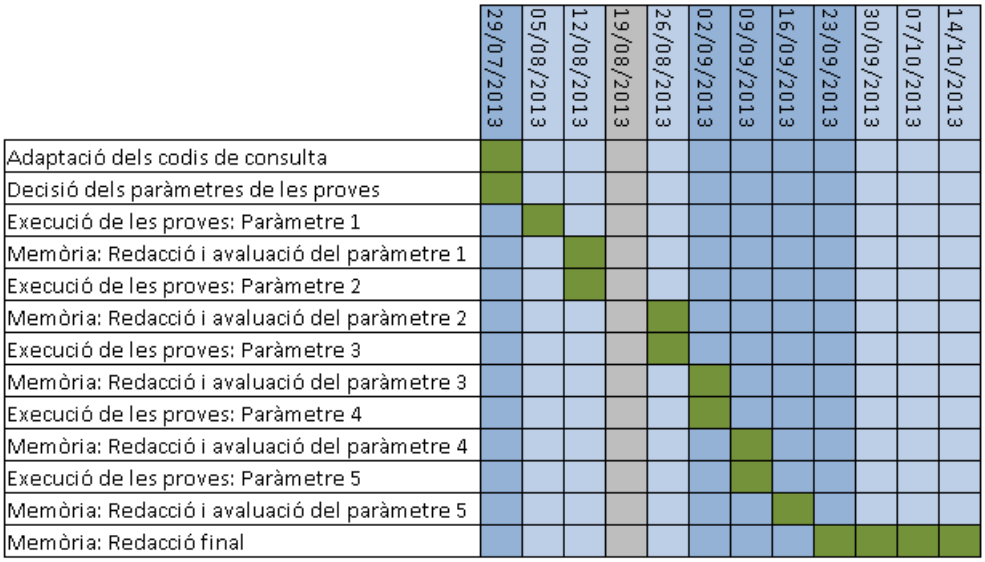


Figura 10.1-2: Diagrama de Gantt de la planificació inicial

Tasca	Duració	Inici	Fi	Hores
Aprenentatge de les tecnologies	46 dies	21/12/2012	22/02/2013	184
Implementació del generador TPC-H	40 dies	25/02/2013	19/04/2013	180
Memòria: Redacció de la implementació del generador TPC-H	25 dies	22/04/2013	24/05/2013	75
Adaptació al clúster del DAC	25 dies	27/05/2013	28/06/2013	150
Primeres proves d'inserció al clúster del DAC	20 dies	01/07/2013	26/07/2013	45
Adaptació dels codis de consulta	4 dies	29/07/2013	01/08/2013	16
Decisió dels paràmetres de les proves	1 dia	02/08/2013	02/08/2013	4
Execució de les proves: Paràmetre 1	1 setmana	05/08/2013	09/08/2013	20
Memòria: Redacció i avaluació del paràmetre 1	3 dies	12/08/2013	14/08/2013	12
Execució de les proves: Paràmetre 2	1 setmana	12/08/2013	16/08/2013	20
Memòria: Redacció i avaluació del paràmetre 2	3 dies	26/08/2013	28/08/2013	12
Execució de les proves: Paràmetre 3	1 setmana	26/08/2013	30/08/2013	20
Memòria: Redacció i avaluació del paràmetre 3	3 dies	02/09/2013	04/09/2013	12
Execució de les proves: Paràmetre 4	1 setmana	02/09/2013	06/09/2013	20
Memòria: Redacció i avaluació del paràmetre 4	3 dies	09/09/2013	11/09/2013	12
Execució de les proves: Paràmetre 5	1 setmana	09/09/2013	13/09/2013	20
Memòria: Redacció i avaluació del paràmetre 5	3 dies	16/09/2013	18/09/2013	12
Memòria: Redacció final	1 mes	19/09/2013	16/10/2013	100
			Total	914

Figura 10.1-3: Taula de la planificació inicial

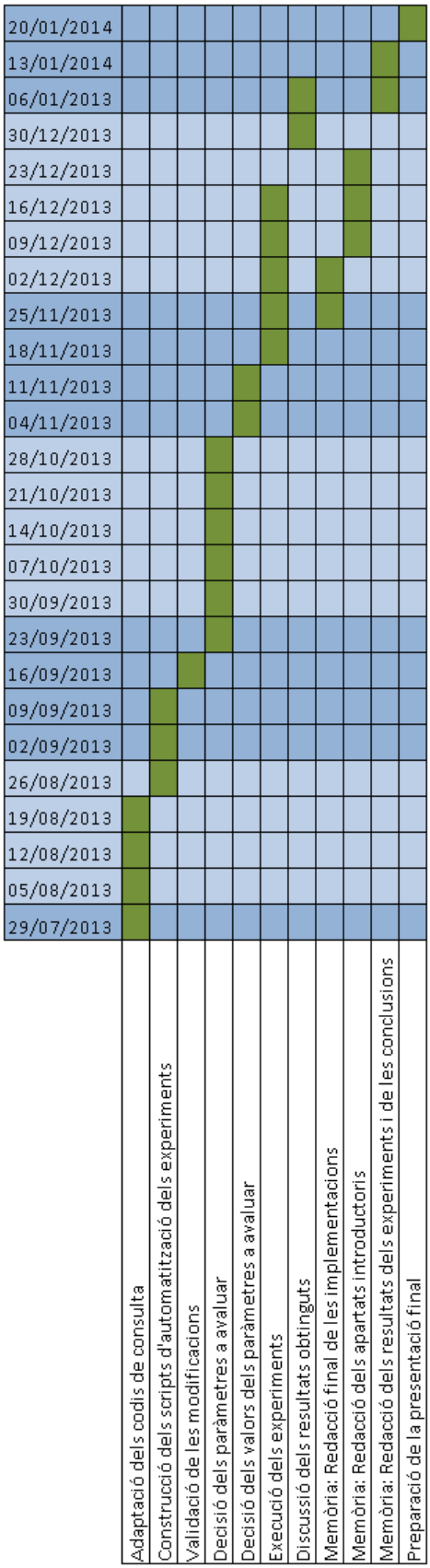


Figura 10.1-4: Diagrama de Gantt de la planificació final

Tasca	Duració	Inici	Fi	Hores
Aprenentatge de les tecnologies	46 dies	21/12/2012	22/02/2013	184
Implementació del generador TPC-H	40 dies	25/02/2013	19/04/2013	180
Memòria: Redacció de la implementació del generador TPC-H	25 dies	22/04/2013	24/05/2013	75
Adaptació al clúster del DAC	25 dies	27/05/2013	28/06/2013	150
Primeres proves d'inserció al clúster del DAC	20 dies	01/07/2013	26/07/2013	45
Adaptació dels codis de consulta	1 mes	29/07/2013	23/08/2013	200
Construcció dels scripts d'automatització dels experiments	3 setmanes	26/08/2013	13/09/2013	120
Validació de les modificacions	1 setmana	16/09/2013	20/09/2013	24
Decisió dels paràmetres a avaluar	30 dies	23/09/2013	01/11/2013	225
Decisió dels valors dels paràmetres a avaluar	2 setmanes	04/11/2013	15/11/2013	120
Execució dels experiments	25 dies	18/11/2013	20/12/2013	75
Memòria: Redacció final de les implementacions	2 setmanes	25/11/2013	06/12/2013	90
Discussió dels resultats obtinguts	5 dies	02/01/2014	08/01/2014	60
Memòria: Redacció dels apartats introductoris	12 dies	09/12/2013	24/12/2013	144
Memòria: Redacció dels resultats dels experiments i de les conclusions	0,8 setmanes	10/01/2014	15/01/2014	48
Preparació de la presentació final	4 dies	16/01/2014	21/01/2014	36
			Total	1776

Figura 10.1-5: Taula de la planificació inicial amb les hores dedicades

10.2 Costos

El pressupost final de dur a terme aquest projecte és de 125500€. La figura *10.2-1* mostra els detalls.

Recurs	Unitats		Preu unitari	Total (€)
Personal desenvolupador (1 informàtic)				
Aprenentatge i configuració del sistema	379	hores	25	9475
Implementacions	500	hores	50	25000
Jocs de proves	24	hores	50	1200
Disseny i execució dels experiments	420	hores	50	21000
Avaluació dels resultats	60	hores	50	3000
Documentació i presentació	393	hores	25	9825
Infraestructura				
Servidors	20	servidors	2500	50000
Manteniment i atenció al client	6	mesos	1000	6000
			Total	125500

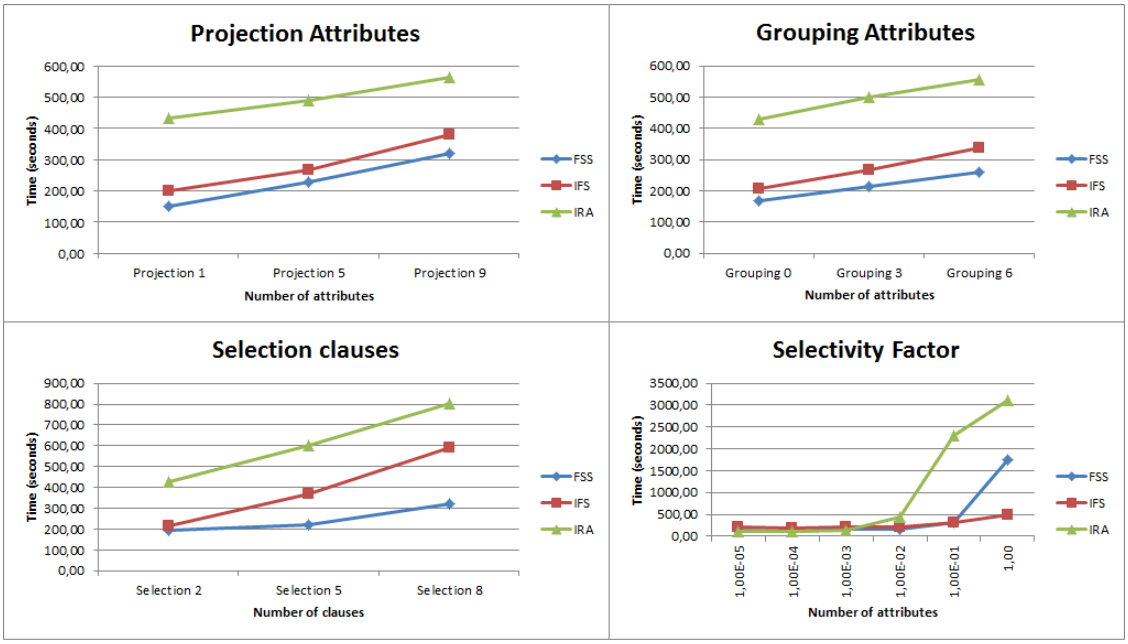
Figura 10.2-1: Pressupost final del projecte

Alguns comentaris:

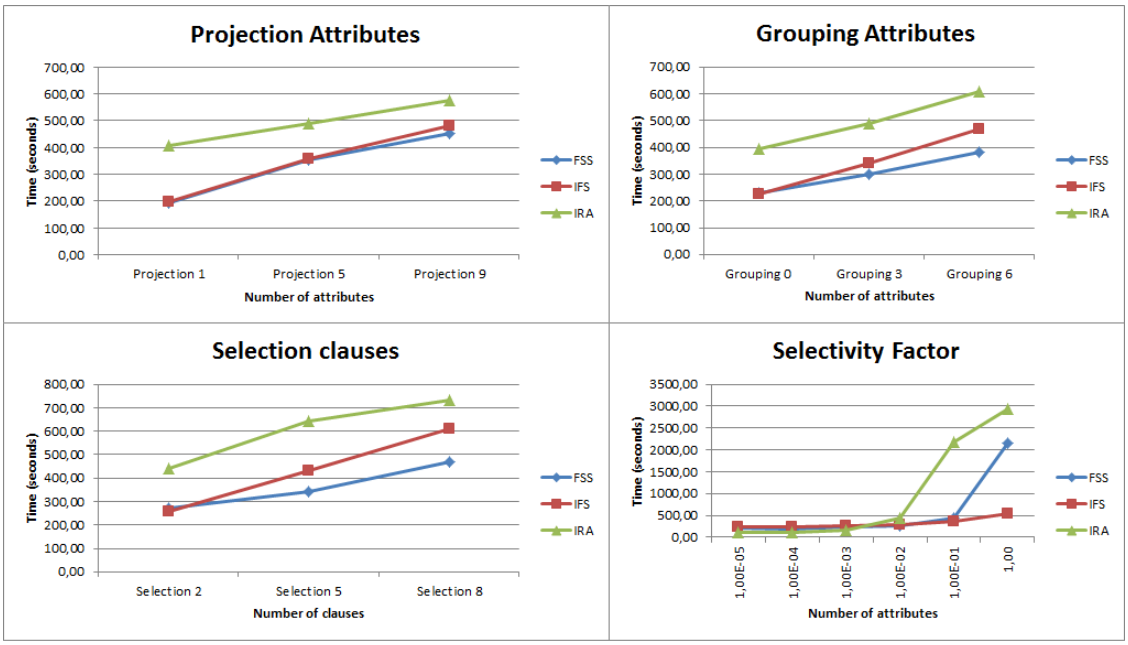
- El costos estàn organitzats segons diferents etapes del projecte. El cost d'un desenvolupar és de 50€/hora, excepte en les etapes d'aprenentatge i de documentació on és la meitat.
  - A l'etapa d'aprenentatge és la meitat perquè durant aquesta etapa estava matriculat d'assignatures de la carrera i per tant el cost no és el mateix que el de un treballador fix.
  - A l'etapa de documentació és la meitat perquè aquesta etapa no seria pròpia d'un projecte real.
- Els costos dels servidors venen donats perquè les màquines emprades són de l'any 2006 i per tant s'ha calculat que el seu cost actual està un pèl per sota de la meitat del cost original (que s'ha estimat a 6000€).
- Tots els preus són en brut.

A Gràfiques per configuració de paràmetres

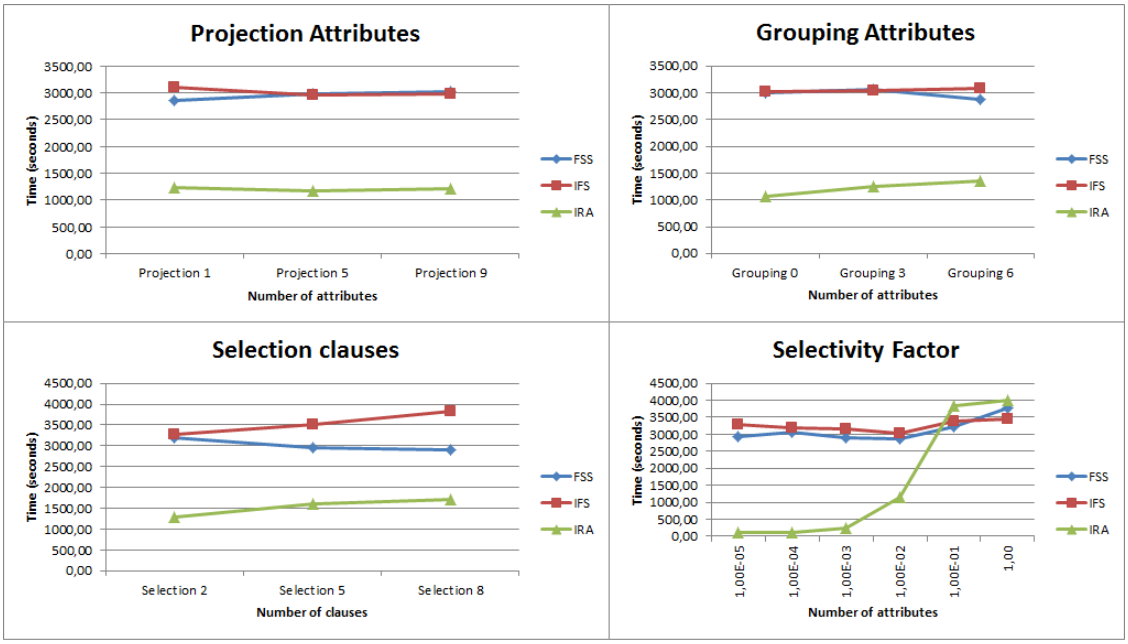
SF = 2, RegionServers = 2, Fragmentació vertical = *AffinityMatrix*



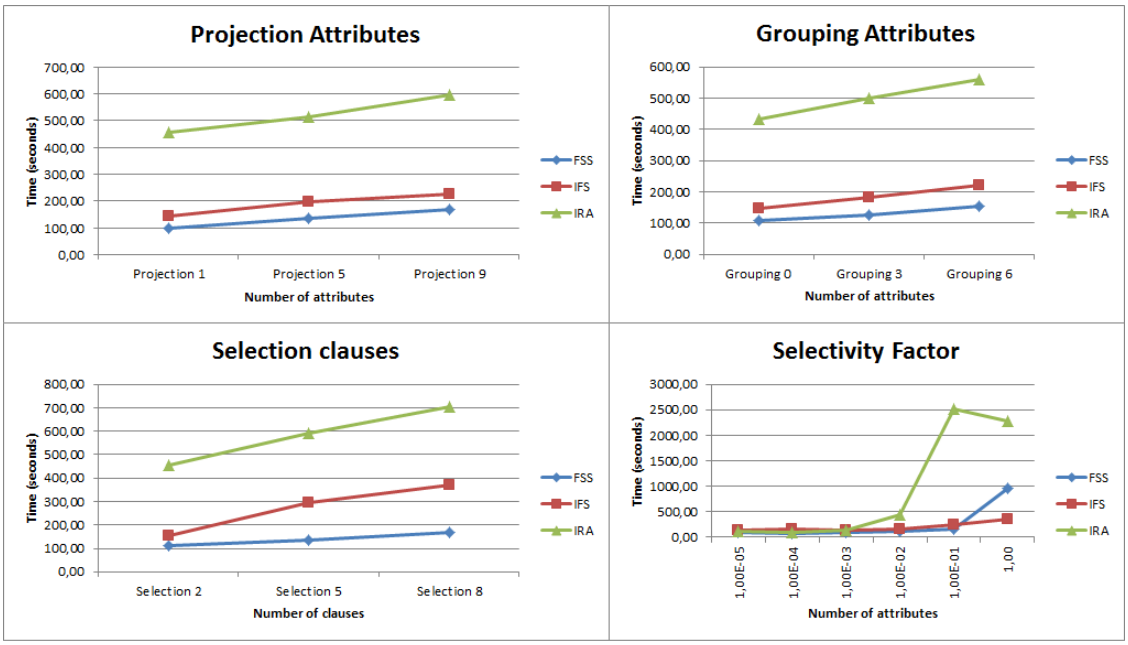
SF = 2, RegionServers = 2, Fragmentació vertical = *ColumnFamily*



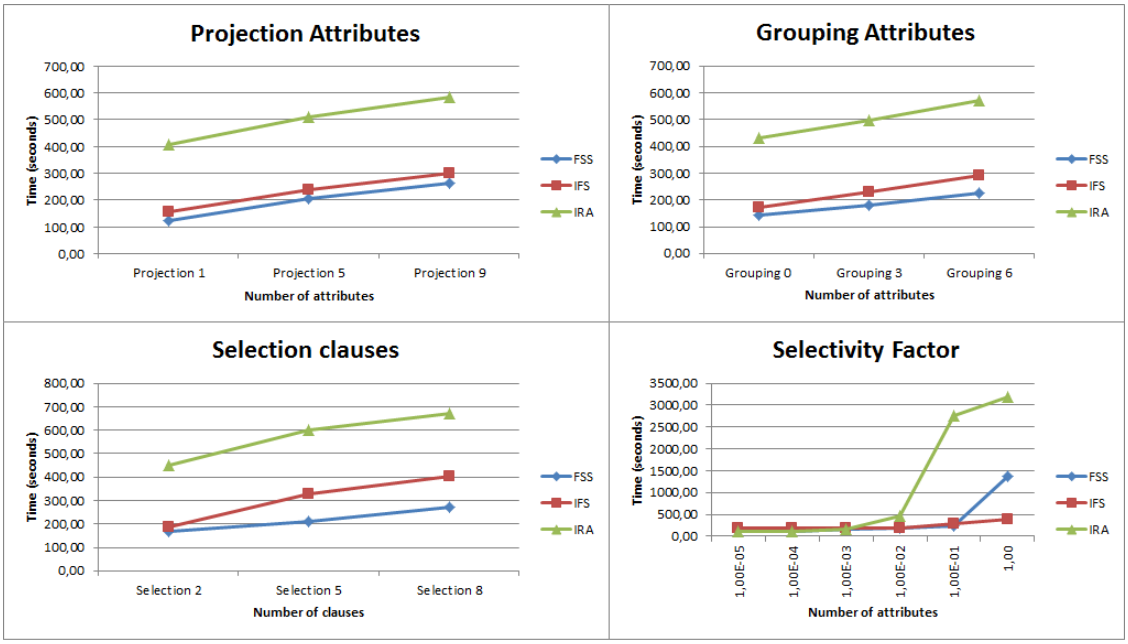
SF = 2, RegionServers = 2, Fragmentació vertical = *SingleColumn*



SF = 2, RegionServers = 5, Fragmentació vertical = *AffinityMatrix*

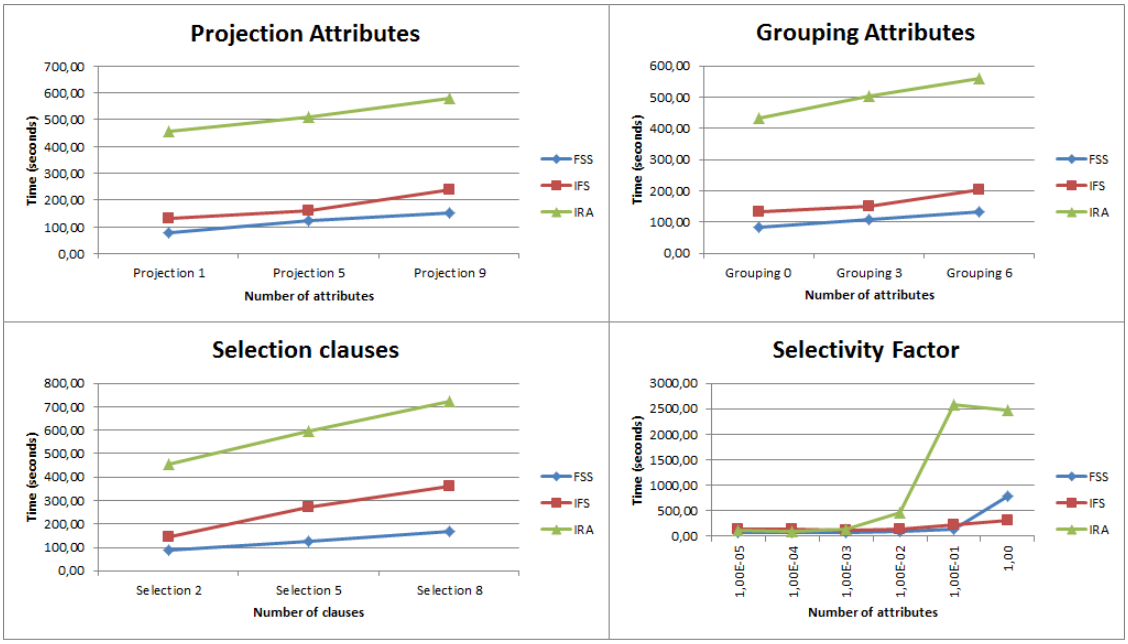


SF = 2, RegionServers = 5, Fragmentació vertical = *ColumnFamily*

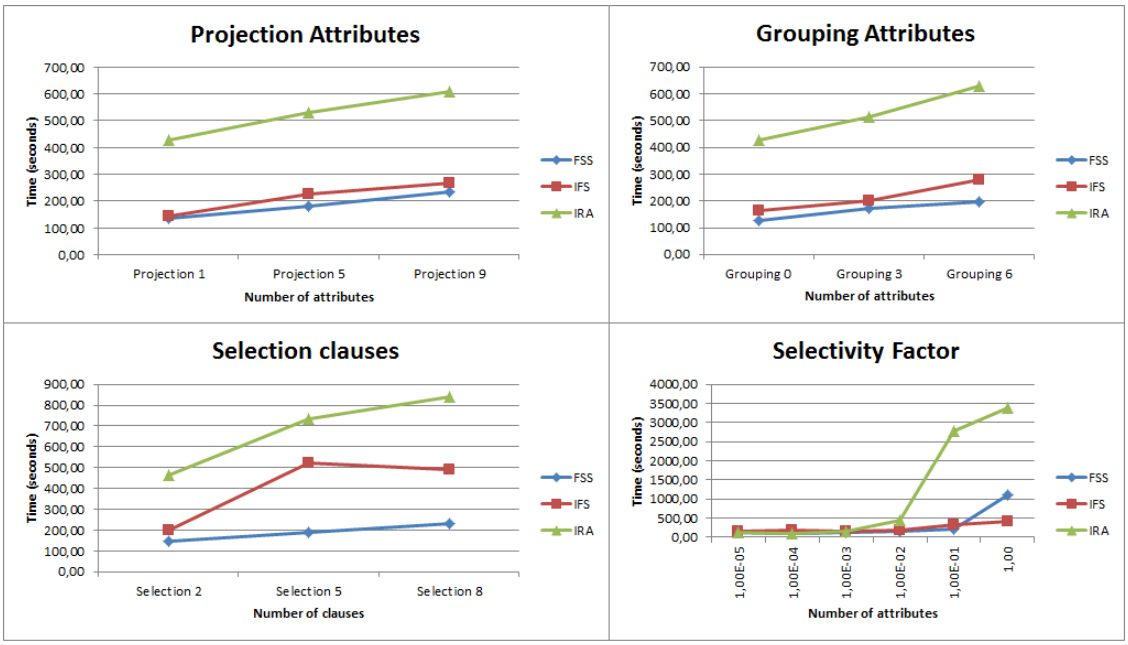


SF = 2, RegionServers = 5, Fragmentació vertical = *SingleColumn*

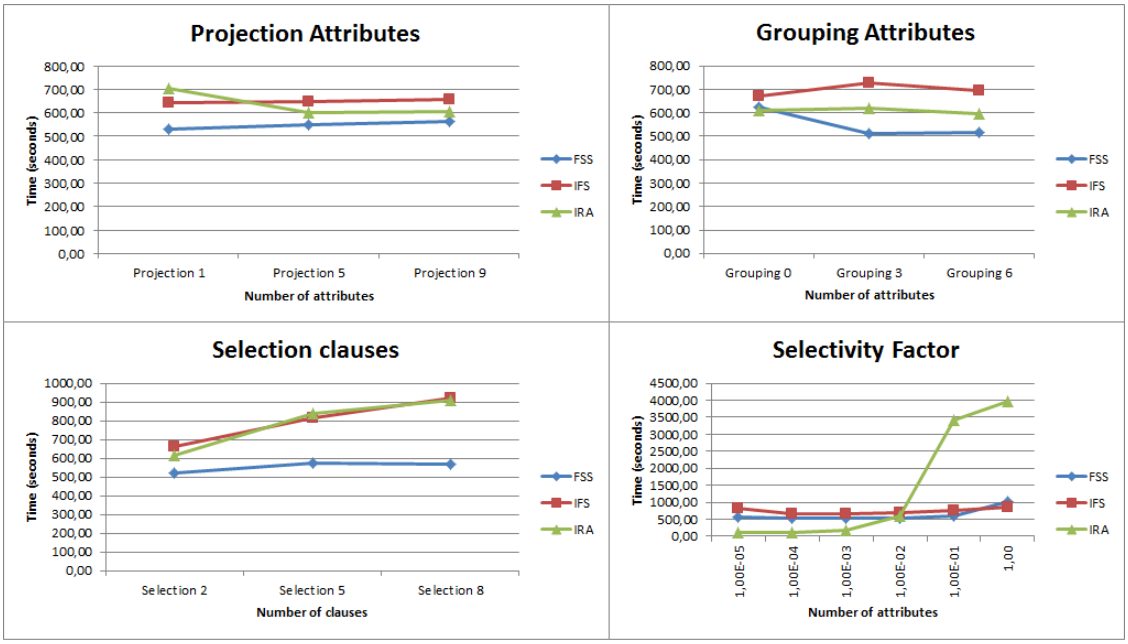
SF = 2, RegionServers = 8, Fragmentació vertical = *AffinityMatrix*



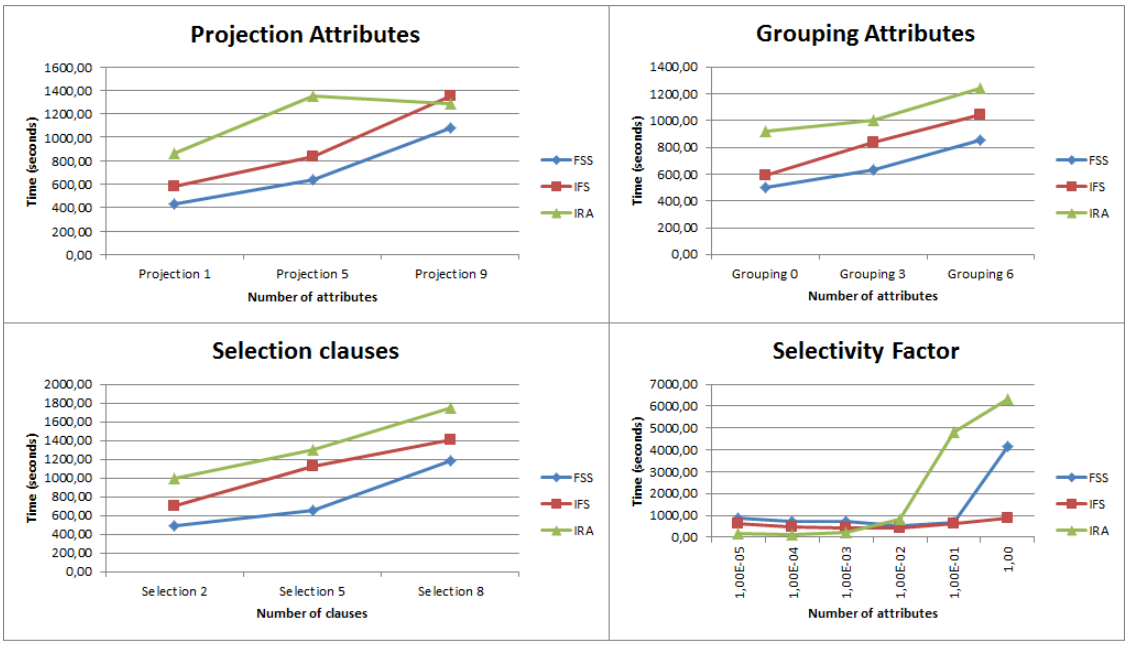
SF = 2, RegionServers = 8, Fragmentació vertical = *ColumnFamily*



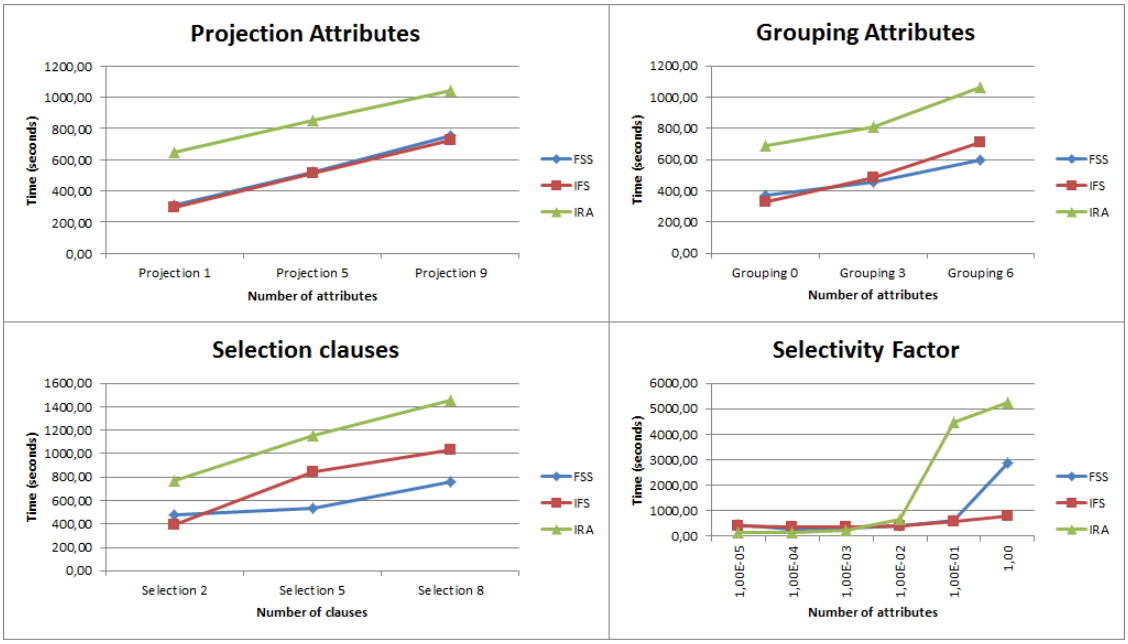
SF = 2, RegionServers = 8, Fragmentació vertical = *SingleColumn*



SF = 4, RegionServers = 2, Fragmentació vertical = *AffinityMatrix*

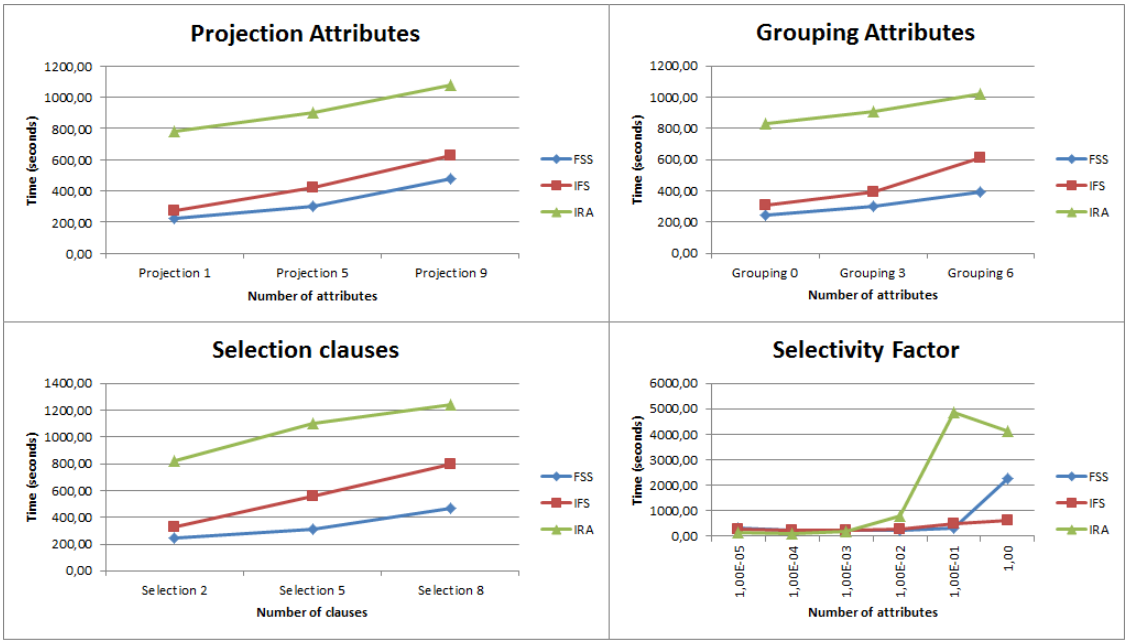


SF = 4, RegionServers = 2, Fragmentació vertical = *ColumnFamily*

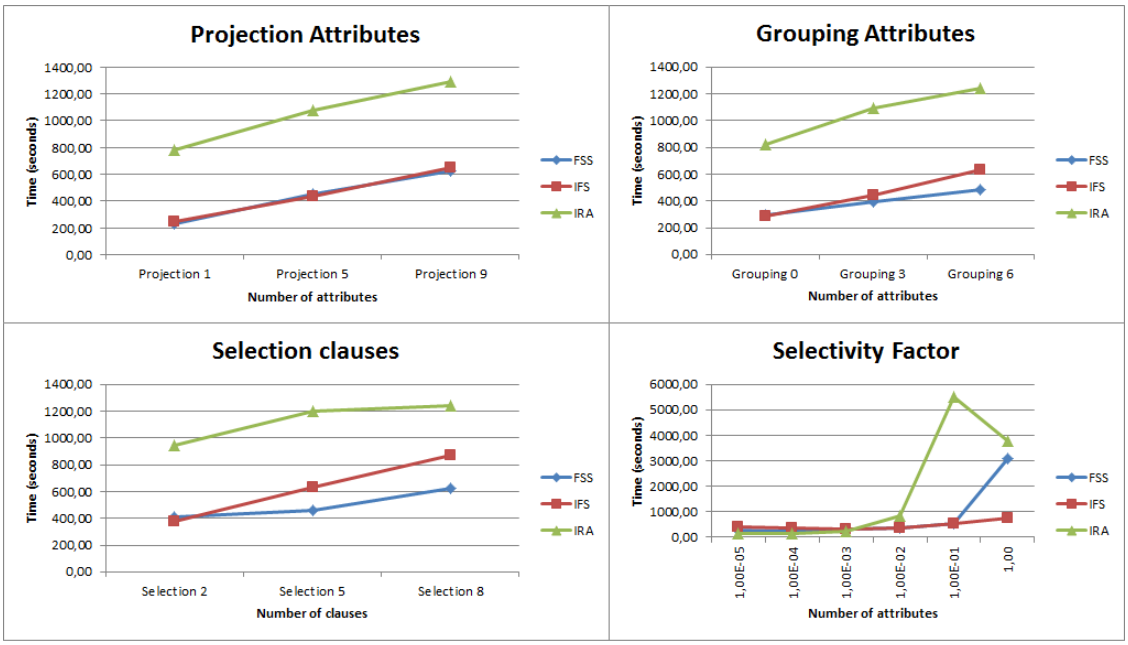


SF = 4, RegionServers = 2, Fragmentació vertical = *SingleColumn*

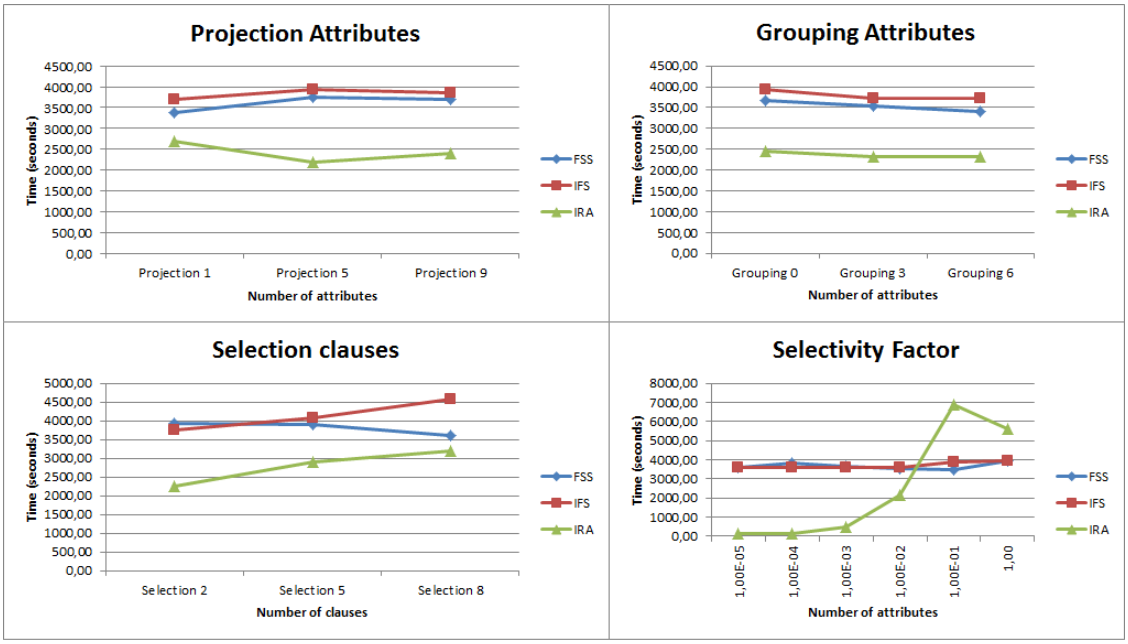
SF = 4, RegionServers = 5, Fragmentació vertical = *AffinityMatrix*



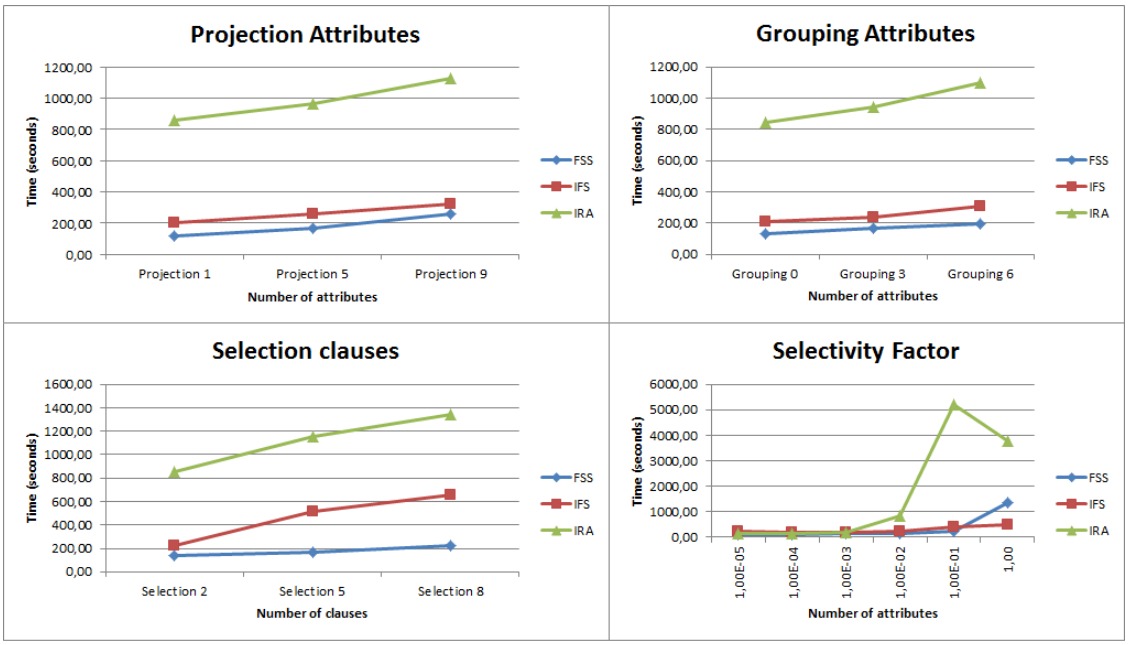
SF = 4, RegionServers = 5, Fragmentació vertical = *ColumnFamily*



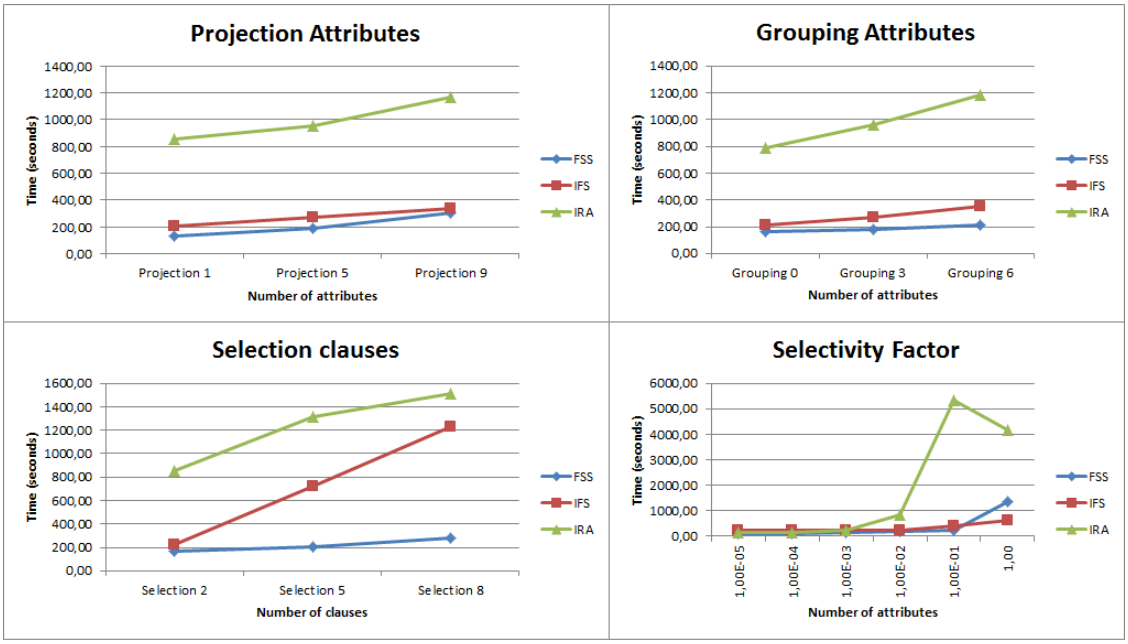
SF = 4, RegionServers = 5, Fragmentació vertical = *SingleColumn*



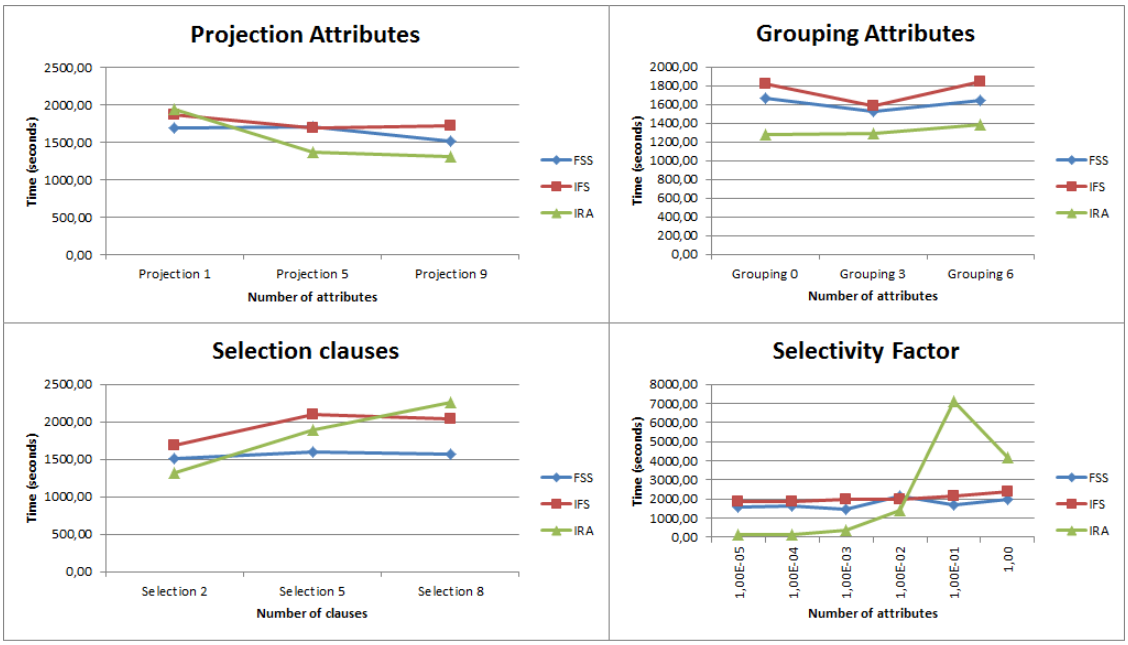
SF = 4, RegionServers = 8, Fragmentació vertical = *AffinityMatrix*



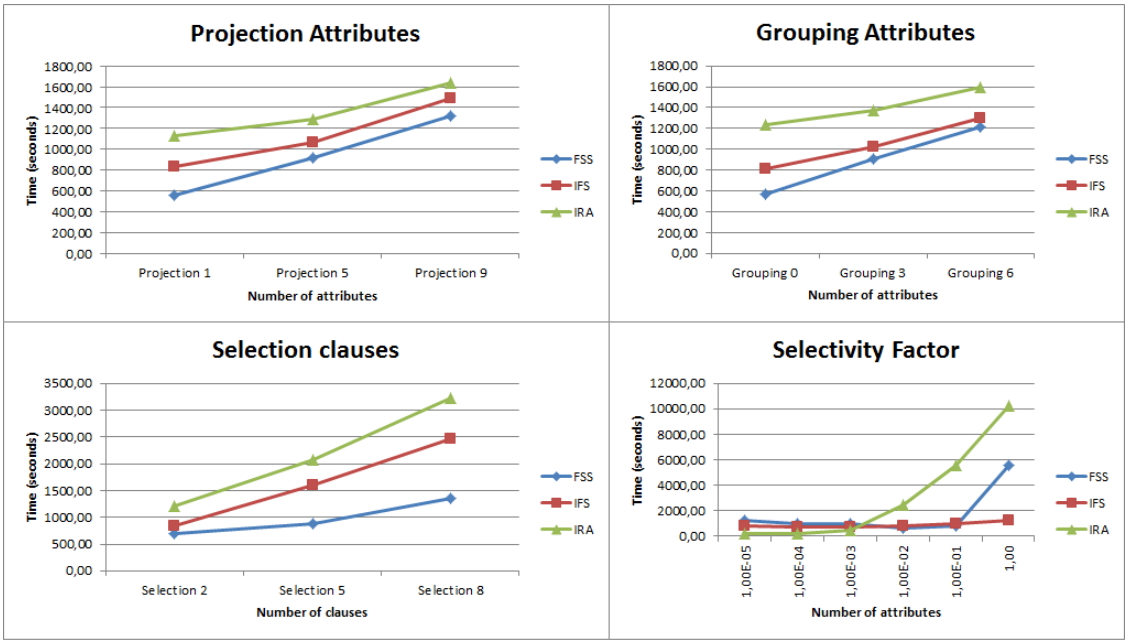
SF = 4, RegionServers = 8, Fragmentació vertical = *ColumnFamily*



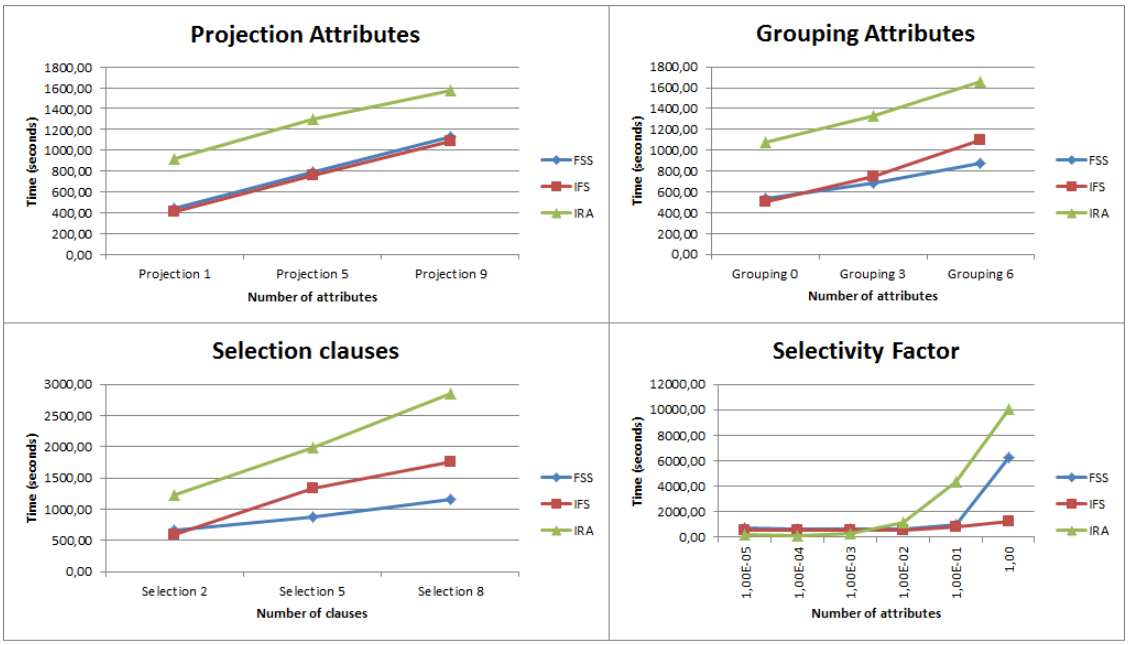
SF = 4, RegionServers = 8, Fragmentació vertical = *SingleColumn*



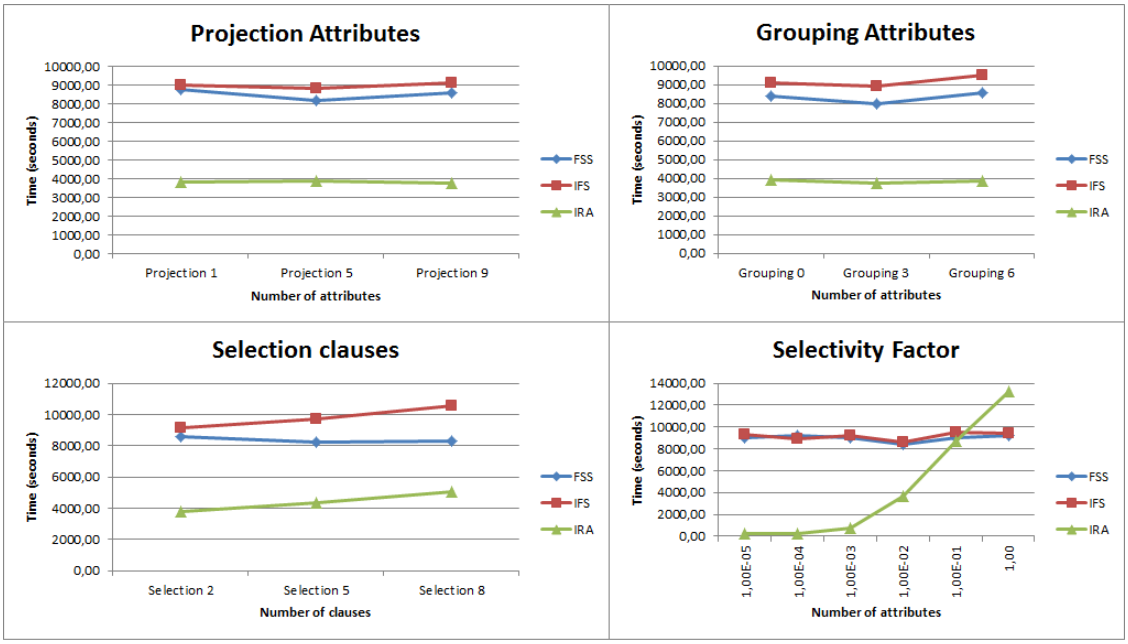
SF = 6, RegionServers = 2, Fragmentació vertical = *AffinityMatrix*



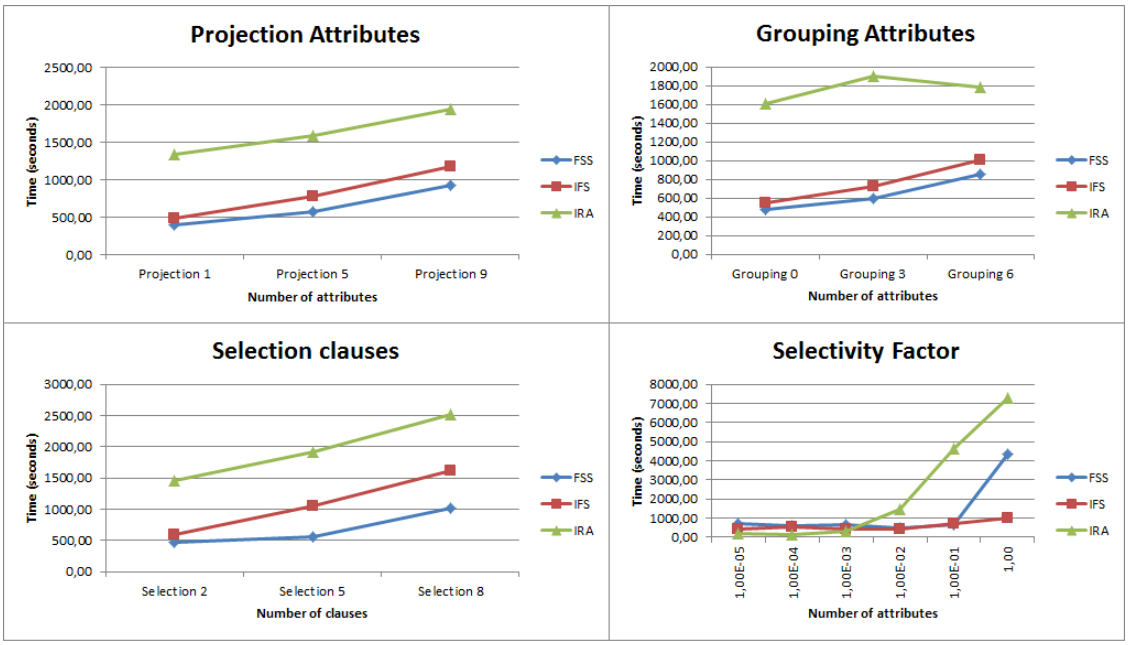
SF = 6, RegionServers = 2, Fragmentació vertical = *ColumnFamily*



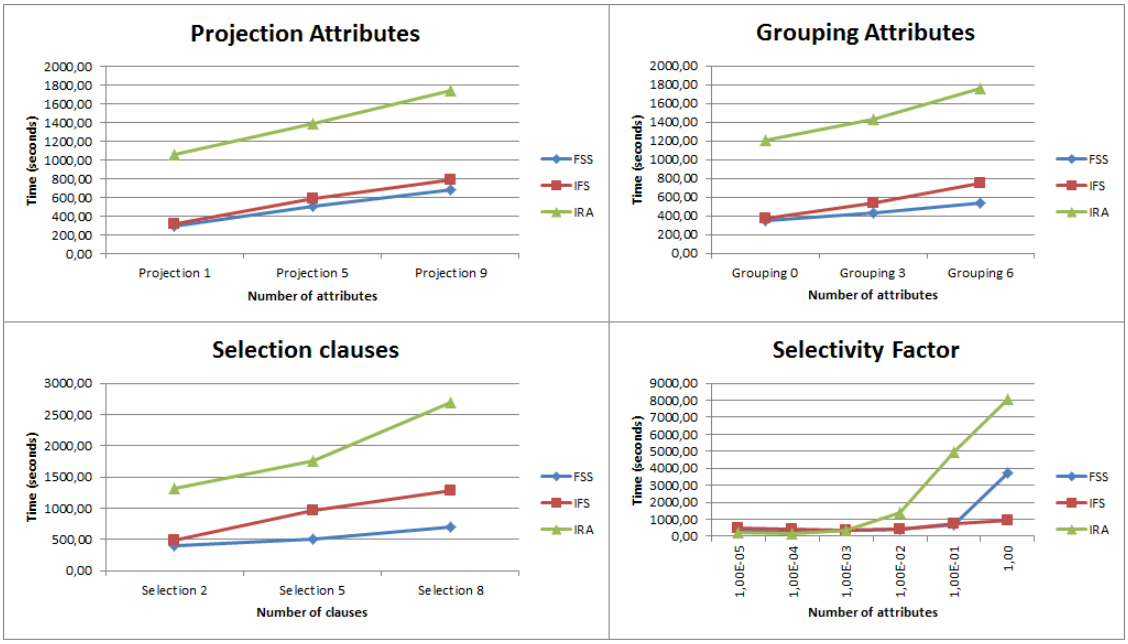
SF = 6, RegionServers = 2, Fragmentació vertical = *SingleColumn*



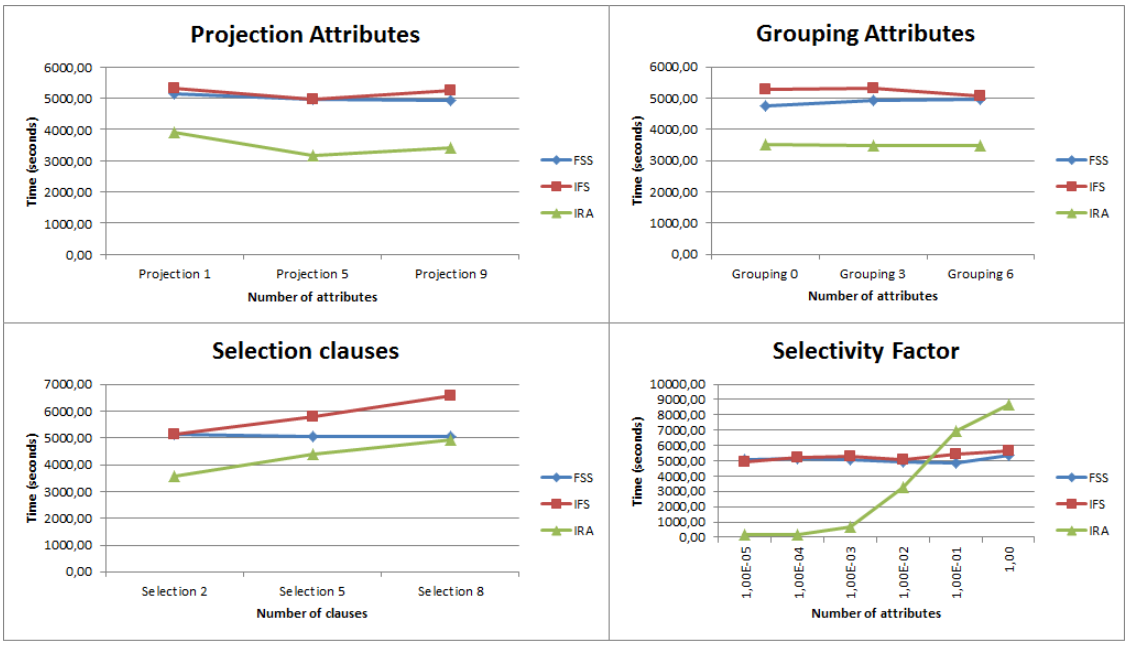
SF = 6, RegionServers = 5, Fragmentació vertical = *AffinityMatrix*



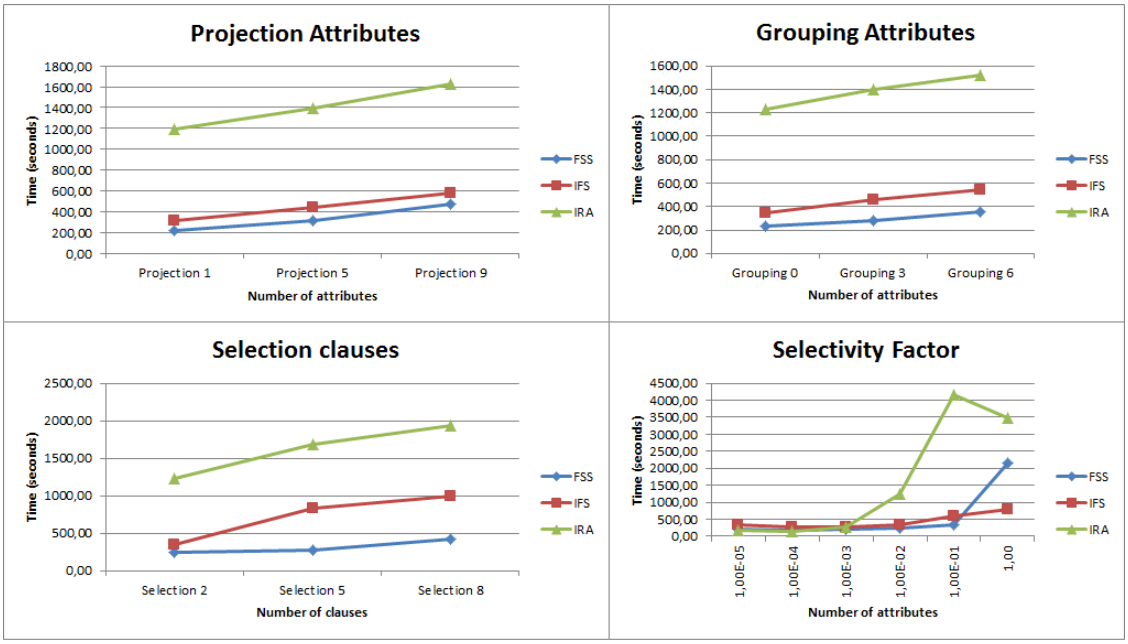
SF = 6, RegionServers = 5, Fragmentació vertical = *ColumnFamily*



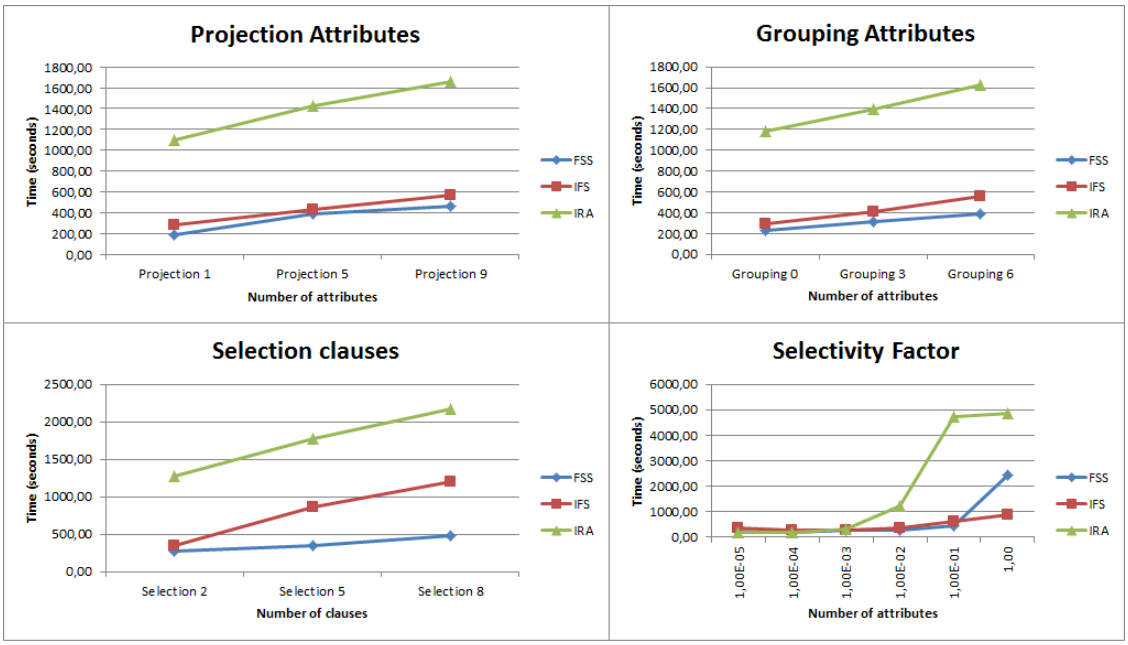
SF = 6, RegionServers = 5, Fragmentació vertical = *SingleColumn*



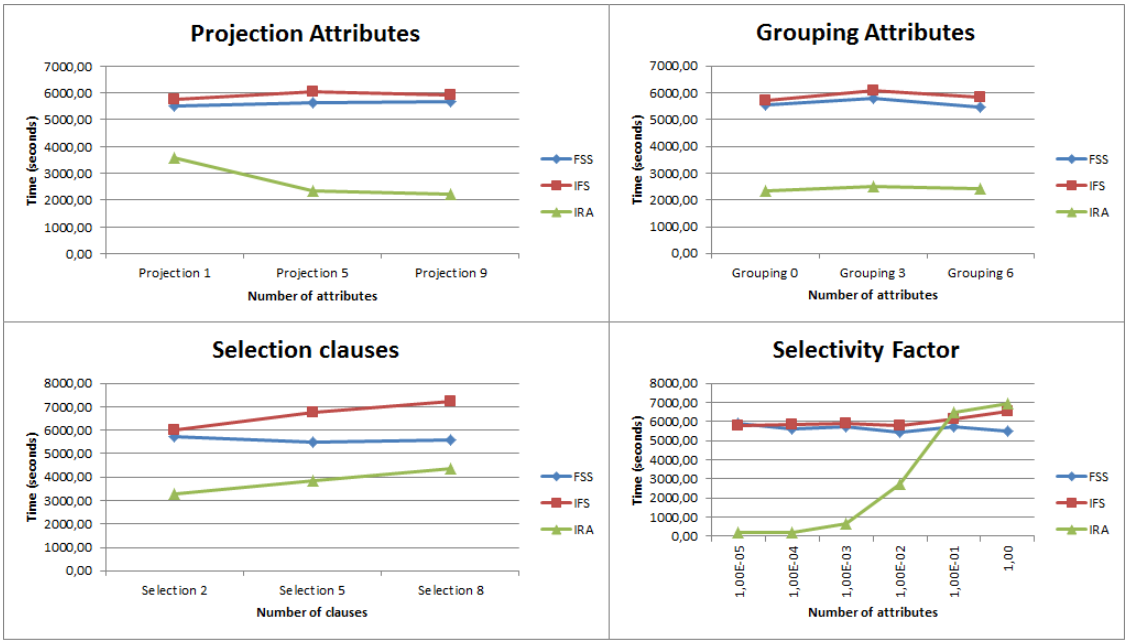
SF = 6, RegionServers = 8, Fragmentació vertical = *AffinityMatrix*



SF = 6, RegionServers = 8, Fragmentació vertical = *ColumnFamily*

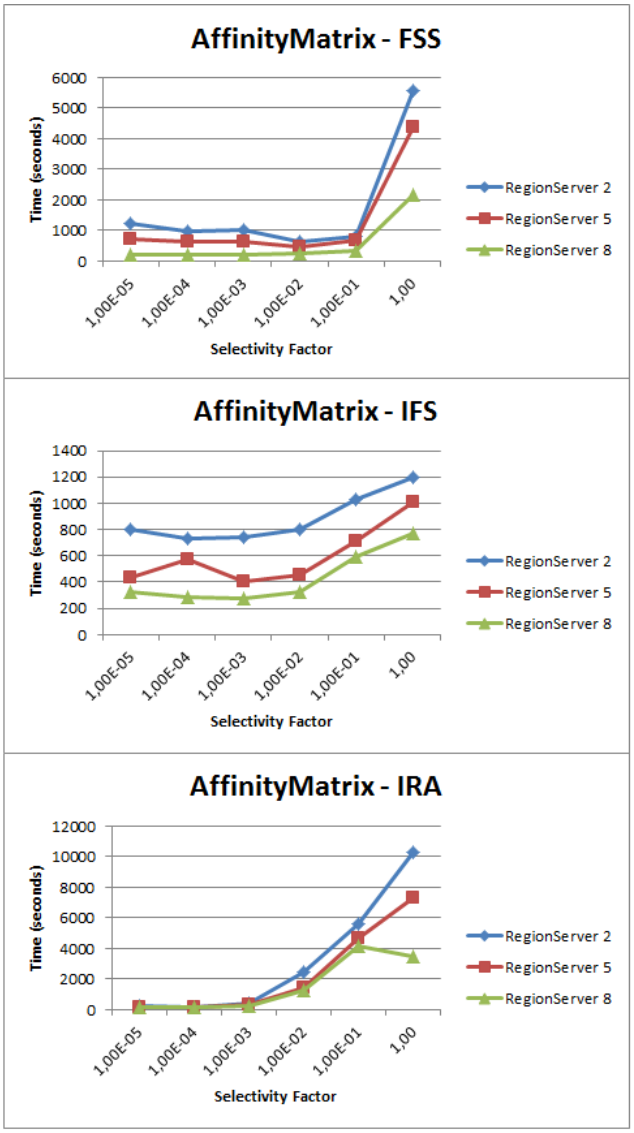


SF = 6, RegionServers = 8, Fragmentació vertical = *SingleColumn*



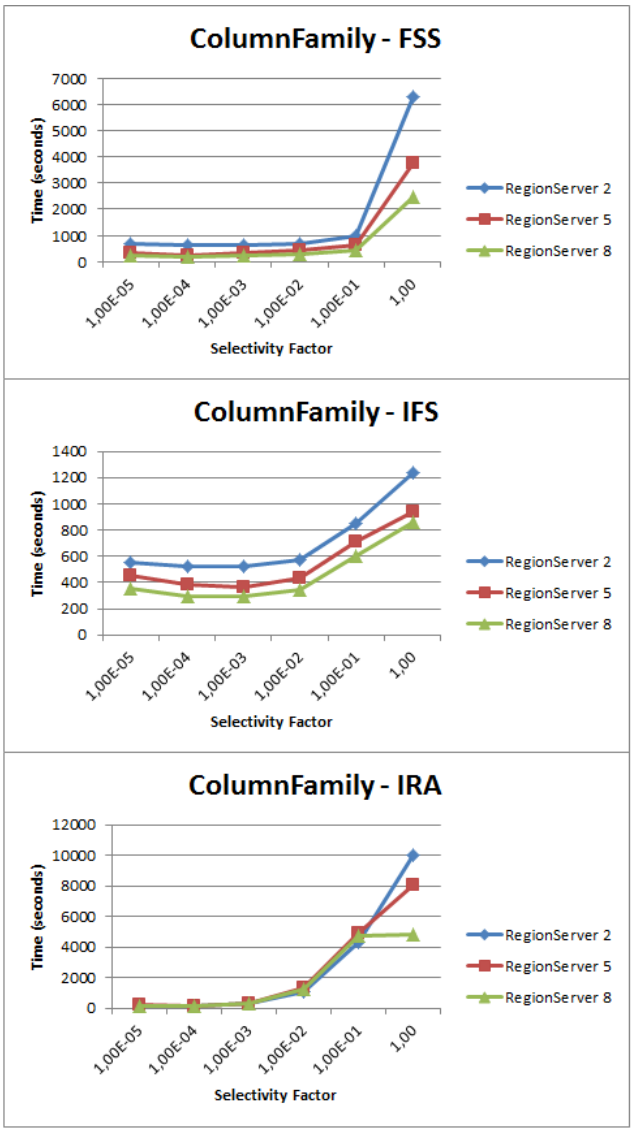
B Gràfiques per paràmetre

Implicació del nombre de RegionServers sobre una fragmentació vertical *AffinityMatrix* i speedup obtingut



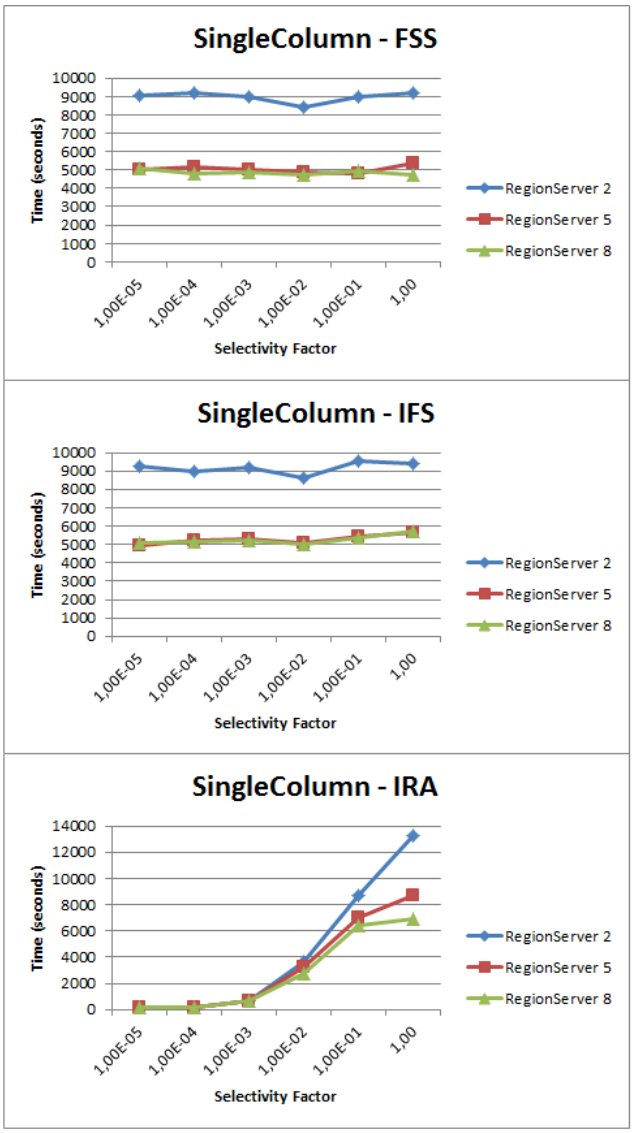

Speed up from RegionServers (AffinityMatrix - Sel. Factor 1e-05)
Speed up from RegionServers (AffinityMatrix - Sel. Factor 1e-02)
Speed up from RegionServers (AffinityMatrix - Sel. Factor 1e-04)
Speed up from RegionServers (AffinityMatrix - Sel. Factor 1e-01)
Speed up from RegionServers (AffinityMatrix - Sel. Factor 1e-03)
Speed up from RegionServers (AffinityMatrix - Sel. Factor 1)
100
80
60
40
20
0
2
5
8
RegionServers
FSS
IFS
IRA

Implicació del nombre de RegionServers sobre una fragmentació vertical *ColumnFamily* i speedup obtingut



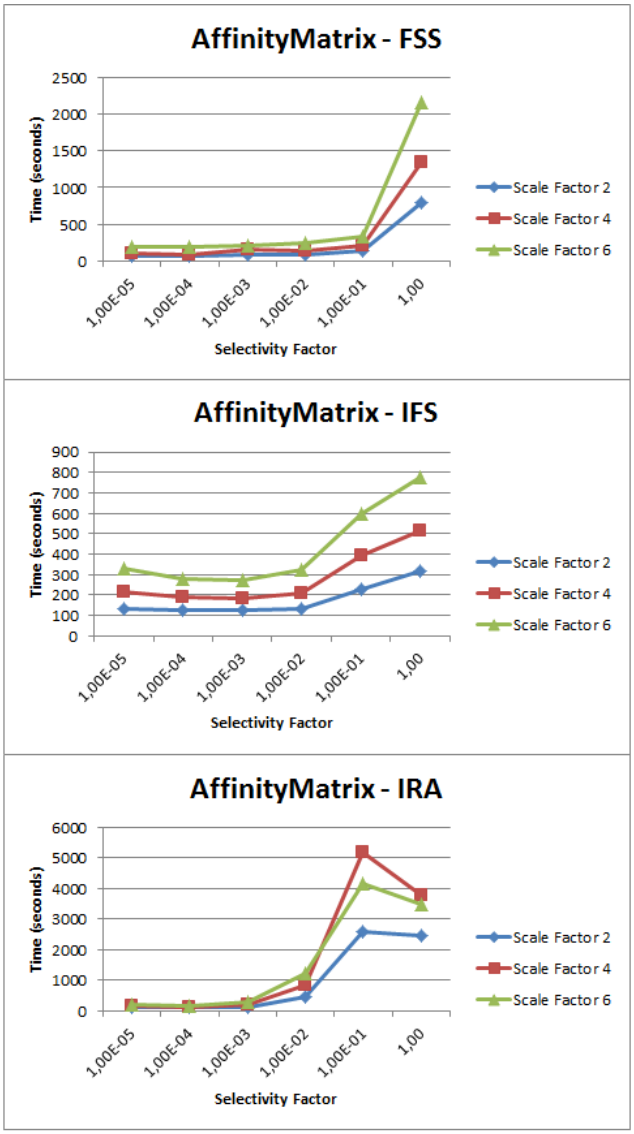

Speed up from RegionServers
(ColumnFamily - Sel. Factor 1e-05)
90
70
50
30
10
-10
-30
2
5
8
RegionServers
FSS
IFS
IRA
Speed up from RegionServers
(ColumnFamily - Sel. Factor 1e-02)
90
70
50
30
10
-10
-30
2
5
8
RegionServers
FSS
IFS
IRA
Speed up from RegionServers
(ColumnFamily - Sel. Factor 1e-04)
90
70
50
30
10
-10
-30
2
5
8
RegionServers
FSS
IFS
IRA
Speed up from RegionServers
(ColumnFamily - Sel. Factor 1e-01)
90
70
50
30
10
-10
-30
2
5
8
RegionServers
FSS
IFS
IRA
Speed up from RegionServers
(ColumnFamily - Sel. Factor 1e-03)
90
70
50
30
10
-10
-30
2
5
8
RegionServers
FSS
IFS
IRA
Speed up from RegionServers
(ColumnFamily - Sel. Factor 1)
100
80
60
40
20
0
2
5
8
RegionServers
FSS
IFS
IRA

Implicació del nombre de RegionServers sobre una fragmentació vertical *SingleColumn* i speedup obtingut

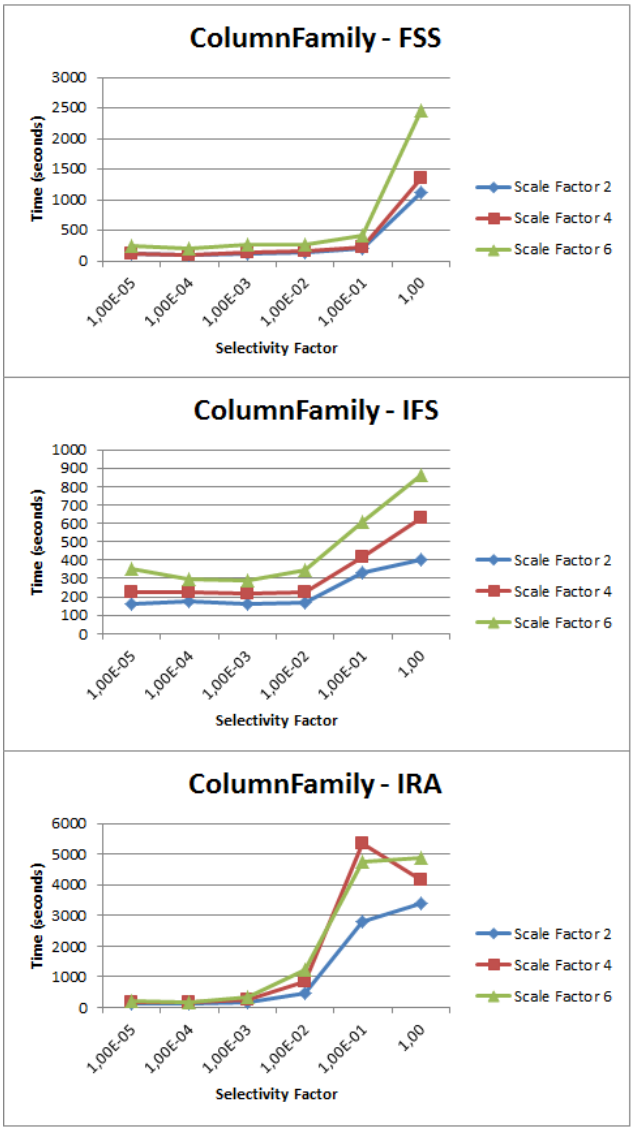



Speed up from RegionServers (SingleColumn - Sel. Factor 1e-05)
Speed up from RegionServers (SingleColumn - Sel. Factor 1e-02)
Speed up from RegionServers (SingleColumn - Sel. Factor 1e-04)
Speed up from RegionServers (SingleColumn - Sel. Factor 1e-01)
Speed up from RegionServers (SingleColumn - Sel. Factor 1e-03)
Speed up from RegionServers (SingleColumn - Sel. Factor 1)
100
80
60
40
20
0
2
5
8
RegionServers
FSS
IFS
IRA

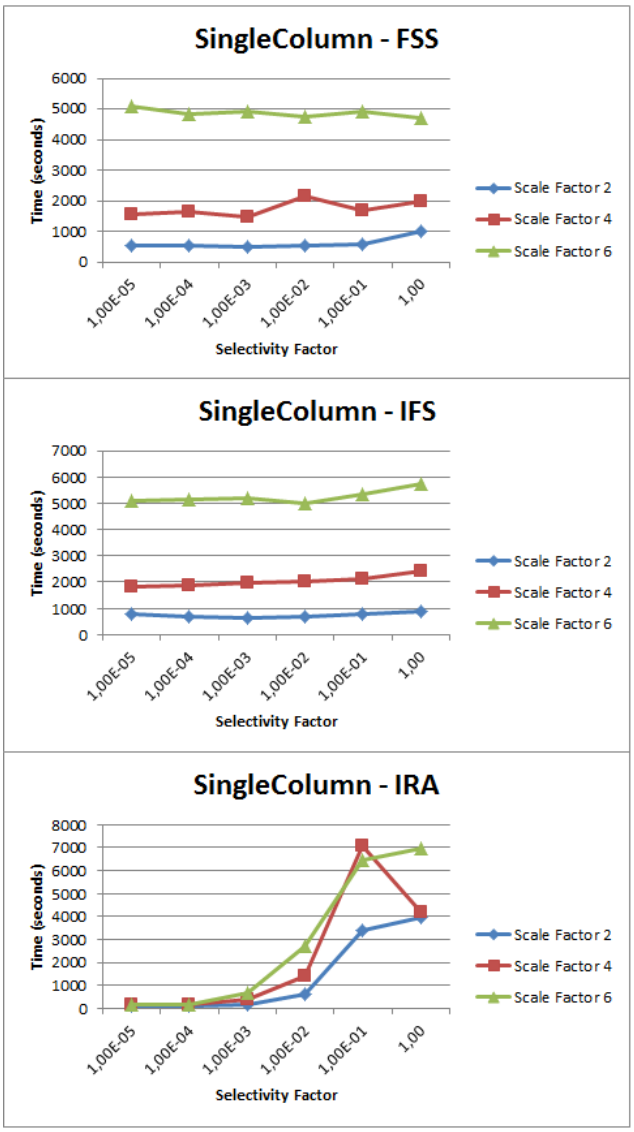
Implicació del volum de dades sobre una fragmentació vertical *AffinityMatrix*



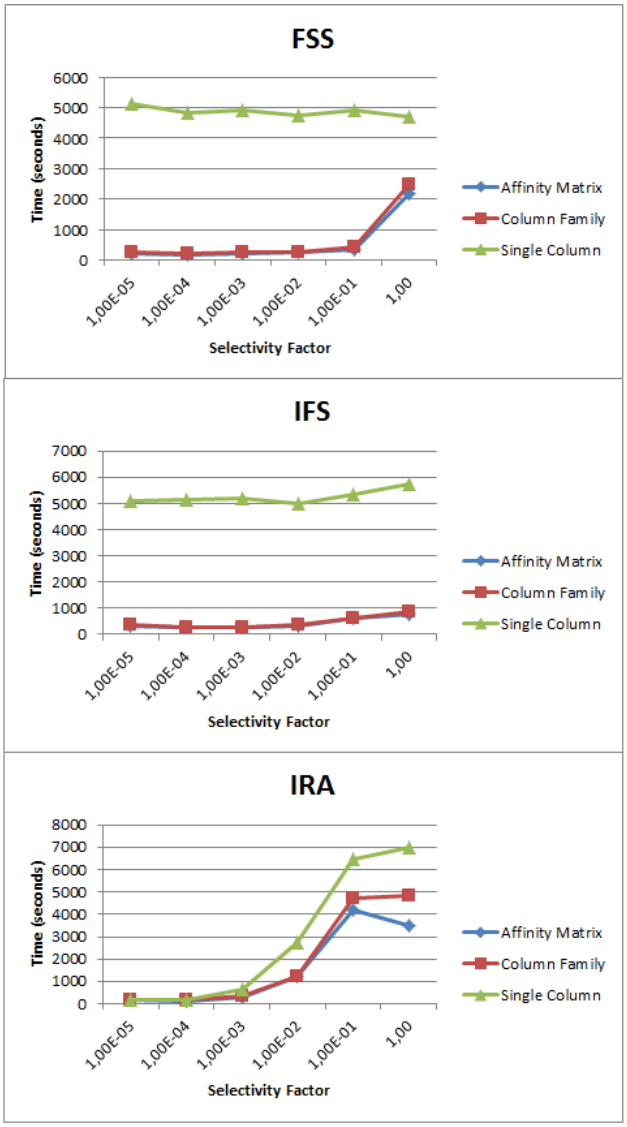
Implicació del volum de dades sobre una fragmentació vertical *ColumnFamily*

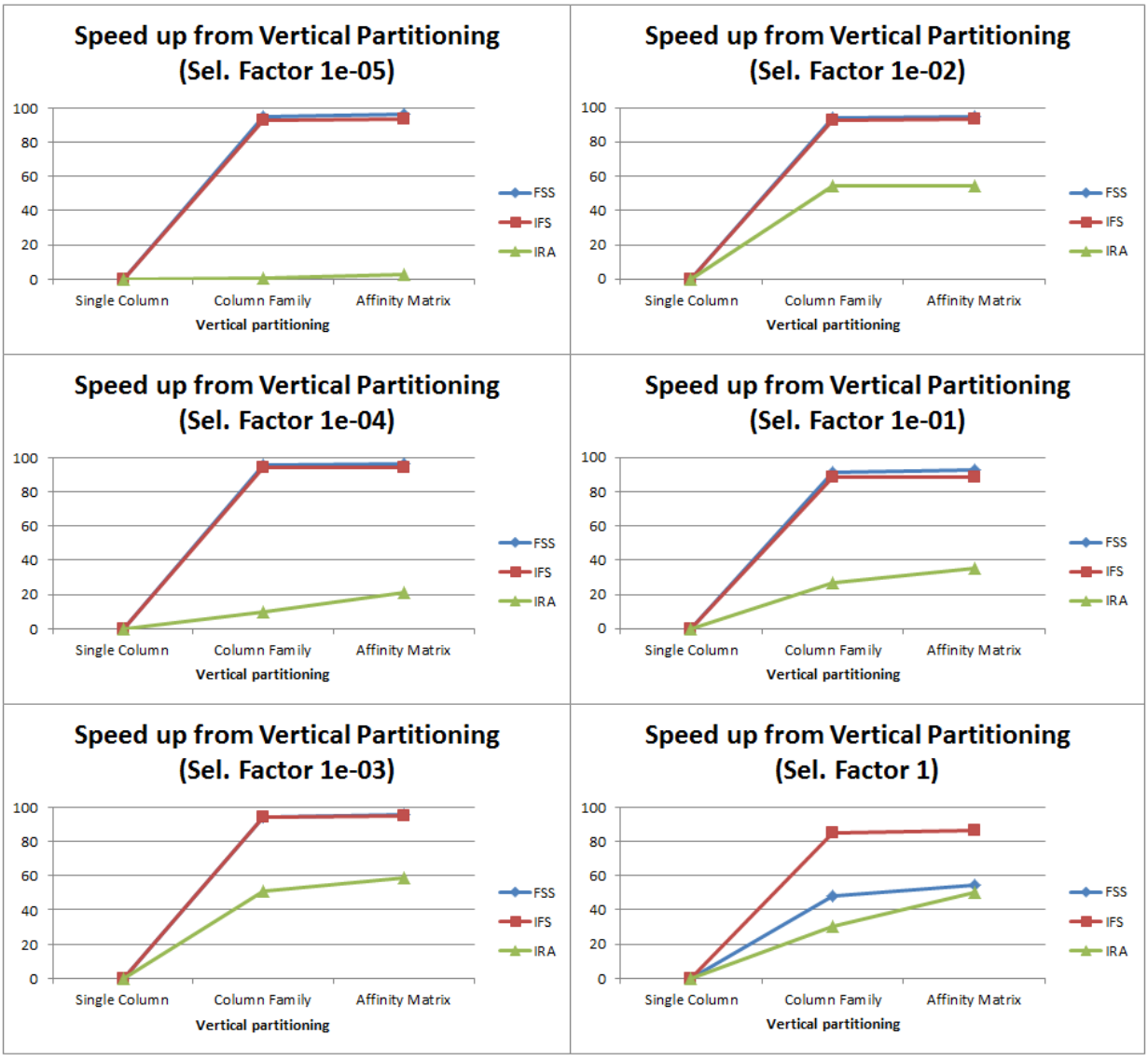


Implicació del volum de dades sobre una fragmentació vertical *SingleColumn*

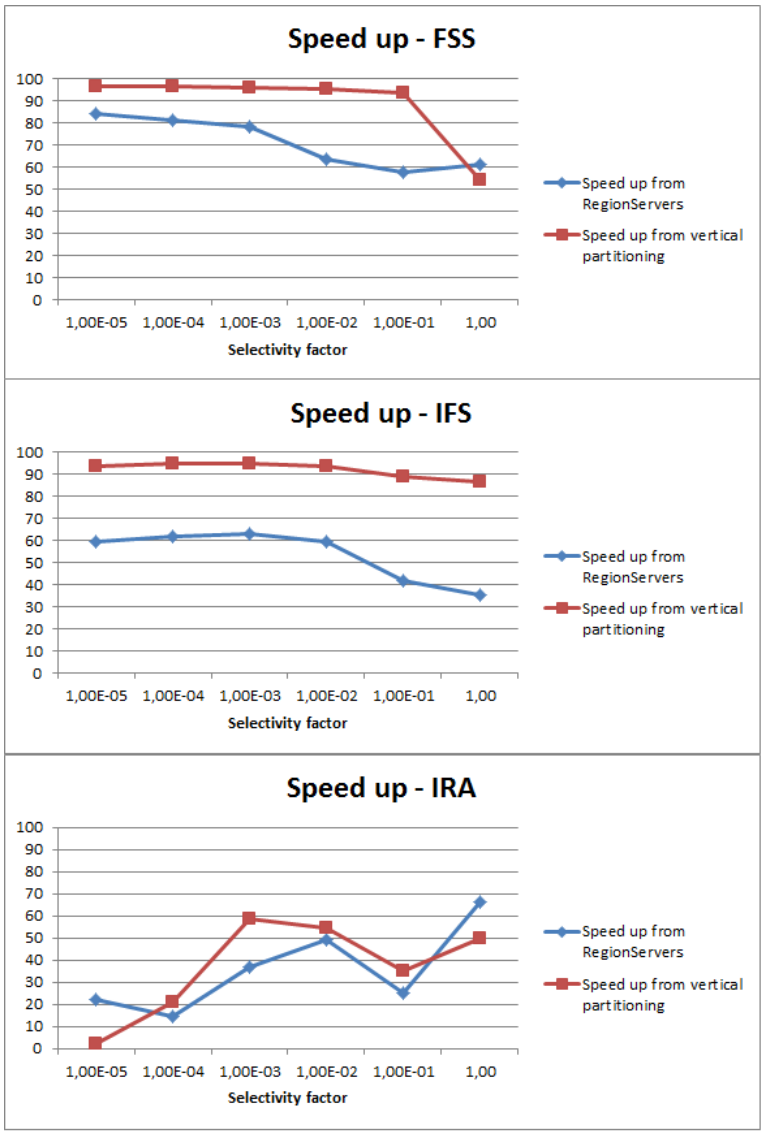


Implicació de l'estratègia de fragmentació vertical i speedup obtingut




Speed up from Vertical Partitioning
(Sel. Factor 1e-05)
100
80
60
40
20
0
Single Column
Column Family
Affinity Matrix
Vertical partitioning
FSS
IFS
IRA
Speed up from Vertical Partitioning
(Sel. Factor 1e-02)
100
80
60
40
20
0
Single Column
Column Family
Affinity Matrix
Vertical partitioning
FSS
IFS
IRA
Speed up from Vertical Partitioning
(Sel. Factor 1e-04)
100
80
60
40
20
0
Single Column
Column Family
Affinity Matrix
Vertical partitioning
FSS
IFS
IRA
Speed up from Vertical Partitioning
(Sel. Factor 1e-01)
100
80
60
40
20
0
Single Column
Column Family
Affinity Matrix
Vertical partitioning
FSS
IFS
IRA
Speed up from Vertical Partitioning
(Sel. Factor 1e-03)
100
80
60
40
20
0
Single Column
Column Family
Affinity Matrix
Vertical partitioning
FSS
IFS
IRA
Speed up from Vertical Partitioning
(Sel. Factor 1)
100
80
60
40
20
0
Single Column
Column Family
Affinity Matrix
Vertical partitioning
FSS
IFS
IRA

Comparativa de l'speed up obtingut de fragmentar verticalment o afegir més RegionServers



Glossari

ACID Sigles de Atomicity, Consistency, Isolation i Durability.. 6

Apache Companyia americana sense ànim de lucre de desenvolupament de software.. 20, 24

API Una Application Programming Interface específica com els components software haurien d'interactuat entre ells.. 47, 57, 59, 67, 68

B+ Indexació del món relacional.. 27, 35–37

BigTable BigTable és la base de dades no relacional implementada per Google.. 24

Buffer Espai de memòria usat temporalment per tal d'emmagatzemar dades que han de ser transferides.. 32, 33, 58, 59, 68, 124, 125, 155, 161

Cache Espai de memòria que emmagatzema un conjunt de dades més petit però que té capacitat per respondre més ràpidament.. 136

Clustered Index Tipus d'índex del món relacional que consisteix en tenir les tuples ordenades físicament a disc per tal d'aprofitar els accessos seqüencials.. 27, 32, 92, 93

Cub De Dades Estructura de dades que permet representar les dades d'un array en un espai de dos o més dimensions.. 8, 11–13, 19, 43, 44, 66, 78, 82, 91, 95, 100, 101, 151, 183

DAC Departament d'Arquitectura de Computadors.. 4, 8, 16, 17, 64, 66, 96, 98, 100, 102–104, 107, 131, 132, 135–140, 184

Data Shipping En sistemes de dades distribuïts, data shipping consisteix moure les dades d'una màquina a una altra per tal de resoldre una query.. 57, 59

Data Warehouse Sistemes de bases de dades orientats a la resolució de queries.. 10, 11

DataNode Node esclau de l'HDFS. 21–23, 40, 41, 96, 104, 109, 110, 139

DBMS Database Management System.. 24

Deadlock Situació en la que dos processos estan ocupant un mateix recurs i no poden continuar fins que aquest recursos sigui lliberat.. 93

Denormalitzar Procés d'intentar optimitzar una base de dades afegint redundància de dades i/o dades agrupades.. 65

Disponbilitat Tolerància d'un sistema a les fallades.. 21

E/S Processos d'entrada i sortida d'una tasca software.. 92

ESSI Enginyeria de Serveis i Sistemes d'Informació.. 8

ETL Extract, Transform and Load. 13, 14

Factor De Selecció Proporció del nombre de tuples que cal recuperar donada en una query respecte la cardinalitat de la taula.. 121, 124, 134

Foreign Key Clau que relaciona un atribut d'una taula amb un atribut d'una altra taula.. 14, 15, 65

Fragmentació Horitzontal Particionament d'una taula en files de manera que cada un d'aquest fragments s'emplaci en una màquina diferent.. 24, 27, 28, 32, 104, 106, 107, 109, 110, 112, 113, 136, 143, 155, 173, 219

Fragmentació Vertical Particionament d'una taula en atributs de manera que cada un d'aquest fragments s'emplaci en una màquina diferent.. 30, 142, 144, 146, 148, 149, 151, 155–163, 167–169, 172, 174, 179, 191–205, 207, 209, 211–214

Full Scan Lectura de totes les tuples d'una taula. 50, 52, 54

Garbage Collector Procés Java d'eliminar aquells objectes que ja no són referenciats.. 121, 126

GFS Sistema de fitxers distribuït de Google.. 20

Google Multinacional americana especialitzada en serveis i productes basats en Internet.. 20, 24

Hadoop Hadoop és un conjunt de diferents tecnologies pensades per treballar en distribuït.. 4, 7, 16, 17, 20, 102, 131, 132, 134, 137, 139, 140, 146

HBase HBase és la base de dades no relacional de Hadoop.. 4, 16, 17, 19, 20, 24–36, 39–41, 47, 57, 59, 64, 65, 67, 68, 93, 95, 96, 98, 99, 102, 104–110, 113, 119, 131, 132, 134–138, 140, 143, 144, 148, 149, 151, 155, 157–159, 162, 170, 172, 173, 176, 179, 183, 219

HDFS Sistem de fitxers distribuïts de Hadoop. 19–23, 40, 41, 96, 109–111, 134, 138–140, 142, 146, 157, 162, 172–175, 217, 219

Heap Espai de memòria ocupat per la màquina virtual de Java.. 103, 104, 106

Heartbeat Missatges enviats de forma periòdica entre dues màquines per tal de validar que l'altra continua activa.. 23, 96

Java Llenguatge de programació orientat a objectes i específicament dissenyat per tenir el mínim de dependències possibles. És un llenguatge semicompilat, lo qual significa que del codi font es tradueix a un altre tipus de codi que interpreta una màquina virtual.. 39, 67, 68, 70, 71, 75, 85, 93, 95, 101, 117, 120, 121, 124, 140

JobTracker Node màster del MapReduce. 37

Key-value Estructura de dades on donada clau, es pot obtenir el respectiu valor.. 7, 36–38, 44, 45, 50–52, 62, 115, 179

Lògica De Negoci Descripció de lògica de negoci. 68

Mantenibilitat Facilitat del software de ser mantingut.. 69

Many-to-many Una relació many-to-many és una relació on moltes instàncies poden estar assignades a moltes altres.. 89

Mapper Objecte distribuït de la primera fase d'una tasca MapReduce.. 38–41, 48, 54, 61, 120–123, 126, 127, 220

MapReduce MapReduce és un paradigma de programació utilitzat per Google per donar suport a la computació distribuïda sobre grans col·leccions de dades.. 4, 16, 17, 19, 20, 36–41, 44, 45, 47, 50, 51, 55, 57, 59, 63, 65, 113, 117, 118, 121, 126, 134, 135, 138, 140, 155–160, 162, 168–170, 172–174, 177, 179, 180, 182, 183, 218, 220

Multithreading Suport a l'execució d'una tasca a través de diferents fils d'execució que treballen concurrentment i on cada un d'ells realitza una petita part del problema general.. 67, 92, 96, 119

NameNode Node màster de l'HDFS. 21, 23, 41

NOSQL Una base de dades NOSQL ofereix mecanismes d'emmagatzematge i de recuperació de les dades alternatius als del món relacional.. 7, 183

OLAP Sistemes de bases de dades orientats a l'anàlisi de les dades que contenen.. 8, 10, 11

OLTP Sistemes de bases de dades orientats a transaccions.. 10

One-to-many Una relació one-to-many és una relació on una instància d'un tipus pot estar assignades a moltes altres.. 89

Overhead Pic de consum de recursos.. 33, 69, 99–101

Portabilitat Capacitat de portar una peça de software cap a diferents plataformes.. 69

Primary Key Clau que inequívocament identifica una tupla.. 64, 81

Reducer Objecte distribuït de la segona fase d'una tasca MapReduce.. 38, 39, 62, 116, 120, 121, 126, 220

RegionServer Node esclau de l'HBase. 24, 27–30, 33–36, 40, 41, 96, 99, 100, 103–107, 109–112, 142–144, 146, 148, 149, 162–170, 172, 173, 191–205, 207, 209, 216

Row Nested Loops Algorísme de join que llegeix files de la primera taula de la join al llarg d'un bucle i les passa a un segon bucle que processa la segona taula de la join.. 65

Scale Factor Paràmetre d'entrada del TPC-H proporcional volum de dades a generar.. 64

Schemaless Tipus de bases de dades on no hi ha un esquema predefinit.. 30, 65

Sharding Té el mateix significat que el terme fragmentació horitzontal ja definit en aquest mateix glossari.. 28

SQL SQL és el llenguatge estàndard d'accés i administració de les bases de dades. 7

TaskTracker Node esclau del MapReduce que pot executar qualsevol de les dues tasques Mapper i Reducer. 37, 40

Timeout Període específic de temps que es permet que transcorri en un sistema abans de que comenci un procés determinat.. 64, 65, 95, 96, 140

TPC-H El TPC-H és un benchmark i un model estàndar de comparació de la presa de decisions.. 4, 16, 17, 48, 64–68, 70, 73, 75, 77, 80, 82, 83, 89, 98–101, 111, 146, 147

WAL Write-Ahead Logging és un sistema de materialitzar els canvis en els logs abans de materialitzar-los a les dades.. 32, 99

ZooKeeper ZooKeeper és un sistema de centralització en sistemes distribuïts d'aquelles dades que no poden ser distribuïdes.. 35, 138

Bibliografía

[1] Largs George, *HBase: The Definitive Guide*. Primera edició, Setembre 2011

[2] Jaume Ferrarons, *Aplicació per fer consultes de cubs de dades usant MapReduce*. PFC, Juny 2011

[3] Tom White, *Hadoop: The Definitive Guide*. Primera edició, Juny 2009

[4] Ahmad Ghazal, Tilmann Rabl, Minqing Hu, Francois Raab, Meikel Poess, Alain Crolotte, Hans-Arno Jacobsen. *BigBench: towards an industry standard benchmark for big data analytics*. SIGMOD Conference 2013: 1197-1208

[5] Alberto Abelló, Jaume Ferrarons, Oscar Romero, *Building Cubes using MapReduce*. DOLAP 2011

[6] Dawei Jiang, Beng Chin Ooi, Lei Shi, Sai Wu, *The Performance of MapReduce: An In-depth Study*. PVLDB 3: 472-483 (2010)

[7] *http://autofei.wordpress.com/2012/04/02/java-example-code-using-hbase-data-model-operations/*
23/12/2012

[8] *http://hbase.apache.org/configuration.html*
23/12/2012

[9] *http://wiki.apache.org/hadoop/Hbase/Shell*
23/12/2012

[10] *http://hbase.apache.org/book/standalone_dist.html#distributed*
24/12/2012

[11] *http://www.thecloudavenue.com/2012/02/getting-started-with-hbase-part-1.html*
05/01//2013

[12] *http://wiki.apache.org/hadoop/FAQ*
16/01//2013

[13] *http://sujee.net/tech/articles/hadoop/hadoop-dns/*
18/01//2013

[14] *http://sekhartechblog.blogspot.com.es/2013/01/installation-of-hbase-in-ubuntu.html*
18/01//2013

[15] *https://github.com/sujee/hbase-mapreduce*
19/01//2013

[16] *http://sujee.net/tech/articles/hadoop/hbase-map-reduce-freq-counter/*
19/01//2013

[17] *http://archive.apache.org/dist/hbase/hbase-0.90.0/docs/apidocs/org/apache/hadoop/hbase/client/Scan byte[])*
20/01//2013

[18] *http://hbase.apache.org/book/perf.writing.html*
31/01//2013

[19] *http://marc.info/?l=hadoop-user&m=119419175925176*
17/03/2013

[20] *http://architects.dzone.com/articles/hbase-error-region-not-online*
21/03/2013

[21] *http://www.mail-archive.com/user@hbase.apache.org/msg22868.html*
27/03/2013

[22] *http://grokbase.com/t/hbase/user/1035e667rk/memory-issue-when-inserting-large-data-into-existed-large-table*
03/04/2013

[23] *http://archive.cloudera.com/cdh4/cdh/4/hbase-0.92.0-cdh4b1/book.html*
03/04/2013

[24] *http://es.slideshare.net/danglbl/schemaless-databases*
04/04/2013

[25] *http://answers.mapr.com/questions/2514/hbase-code-is-not-running-javanetconnectexception*
13/04/2013

[26] *http://zookeeper.apache.org/doc/trunk/zookeeperOver.html*
09/05/2013

[27] *http://hbase.apache.org/book/hbase_metrics.html*
31/05/2013

[28] *http://hadoop.apache.org/docs/stable/hdfs_design.html*
03/06/2013

[29] *http://stackoverflow.com/questions/11840255/hbase-hadoop-dfs-size-not-decreased-even-after-deleting-a-column-family*
08/06/2013

[30] *http://www.ngdata.com/visualizing-hbase-flushes-and-compactions/*
10/06/2013

[31] *http://learnhbase.wordpress.com/2013/03/02/hbase-shell-commands/*
10/06/2013

[32] *http://stackoverflow.com/questions/13741946/role-of-datanode-regionserver-in-hbase-hadoop-integration*
13/06/2013

[33] *http://hortonworks.com/blog/apache-hbase-region-splitting-and-merging/*
25/06/2013

[34] *http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/io/Reference.html*
26/06/2013

[35] *http://hbase.apache.org/book/regions.arch.html*
26/06/2013

[36] *http://hortonworks.com/blog/apache-hbase-region-splitting-and-merging/*
26/06/2013

[37] *http://blog.sematext.com/2012/07/16/hbase-memstore-what-you-should-know/*
27/06/2013

[38] *http://www.ngdata.com/visualizing-hbase-flushes-and-compactions/*
27/06/2013

[39] *http://wiki.apache.org/hadoop/Hbase/MavenPrimer*
30/06/2013

[40] *http://datawarehouse4u.info/OLTP-vs-OLAP.html*
02/10/2013

[41] *http://en.wikipedia.org/wiki/OLAP_cube*
02/10/2013

[42] *http://en.wikipedia.org/wiki/Star_schema*
03/10/2013

[43] *http://hbase.apache.org/book/trouble.namenode.html*
03/10/2013

[44] *http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/filter/SingleColumnValueFilter.html*
12/10/2013

[45] *http://www.eurecom.fr/ michiard/teaching/slides/clouds/tutorial-hbase.pdf*
01/11/2013

[46] *http://detexify.kirelabs.org/classify.html*
04/11/2013

[47] *https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html#NameNode+and+DataNodes*
05/12/2013

[48] *http://grepcode.com/file/repository.cloudera.com/content/repositories/releases/com.cloudera.hadoop/hadoop-core/0.20.2-320/org/apache/hadoop/hdfs/server/namenode/ReplicationTargetChooser.java*
23/12/2013

[49] *http://en.wikipedia.org/wiki/Quorum_(distributed_computing)*
01/11/2013