Marc Hostau Guimerà

AVALUACIÓ DELS MECANISMES DE COMPARTICIÓ D'INFORMACIÓ AMB ARQUITECTURES FaaS

TREBALL DE FI DE GRAU

dirigit per Marc Sánchez Artigas

Grau d'Enginyeria Informàtica/Telemàtica

UNIVERSITAT ROVIRA I VIRGILI

Tarragona

2023

Resum.

La computació sense servidor va facilitar als usuaris menys experimentats la interacció amb els recursos del núvol. El paradigma *serverless* fa responsable al proveïdor del núvol de l'aprovisionament dels recursos, l'escalabilitat i la gestió de les execucions. La seva elasticitat, temps de resposta veloços i un model de preus precís, fan que sigui un gran atractiu pels sistemes d'anàlisi de dades a gran escala. Un exemple pot ser PyWren i la seva extensió, Lithops, *frameworks* que permeten l'execució de funcions independents en paral·lel i la comunicació de dades entre les quals són dependents. El model de funcions sense servidor presenta certes limitacions donada la inviabilitat de la comunicació entre funcions mitjançant la xarxa. Lithops com solució al problema, realitza aquesta comunicació a través de sistemes d'emmagatzematge remot d'alta latència, el que implica un intercanvi poc eficient d'informació entre funcions. En aquest treball es presenta un model de comunicació entre funcions que s'executen en una mateixa màquina virtual, mitjançant memòria compartida i còpia zero. Per comprovar el seu rendiment, s'elaboren diferents estudis i anàlisis mitjançant comparacions amb altres sistemes d'emmagatzematge. Finalment, es valora la incorporació de la nova estratègia de compartició en Seer, un sistema d'anàlisi de dades construït sobre el model de funcions.

Resumen.

La computación sin servidor facilitó a los usuarios menos experimentados la interacción con los recursos de la de la nube. El paradigma *serverless* hace responsable al proveedor de la nube del aprovisionamiento de recursos, la escalabilidad y la gestión de las ejecuciones. Su elasticidad, los veloces tiempos de respuesta y un modelo de precios preciso, lo hacen atractivo para los sistemas de análisis de datos a gran escala. Un ejemplo puede ser PyWren y su extensión, Lithops, *frameworks* que permiten la ejecución de funciones independientes en paralelo y la comunicación de datos entre las que son dependientes. El modelo de funciones sin servidor presenta limitaciones dada la inviabilidad de la comunicación entre funciones a través de la red. Lithops, como solución al problema, realiza esta comunicación a través de sistemas de almacenaje remoto de alta latencia, lo que implica un intercambio poco eficiente de información entre funciones. En este trabajo se presenta un modelo de comunicación de datos entre funciones que se ejecutan en una misma máquina virtual, mediante memoria compartida y copia cero. Para comprobar su rendimiento, se elaboran diferentes estudios y análisis mediante comparaciones con otros sistemas de almacenaje. Finalmente, se valora la incorporación de la nueva estrategia de compartición en Seer, un sistema de análisis de datos construido sobre el modelo de funciones.

Abstract.

Serverless computing made it easier for less experienced users to interact with cloud resources. The serverless paradigm makes the cloud provider responsible for resource provisioning, scalability, and execution management. Its elasticity, fast response times, and precise pricing model make it highly attractive for large-scale data analytics systems. An example could be PyWren and its extension, Lithops, frameworks that enable the parallel execution of independent functions and data communication between dependent ones. The serverless function model has limitations due to the impracticality of communication between functions over the network. Lithops, as a solution to this problem, performs this communication through high-latency remote storage systems, leading to inefficient information exchanges between functions. This work presents a model for data communication between functions that run on the same virtual machine, using shared memory and zero-copy techniques. To assess its performance, various studies and analyses are conducted, comparing it with other storage systems. Finally, the incorporation of the new sharing strategy in Seer, a data analysis system built on the function model, is evaluated.

Índex

Índex de taules

Índex de figures

1 Introducció

1.1 Computació al núvol i el paradigma serverless

La computació al núvol [1] és un paradigma que consisteix a proporcionar diversos recursos informàtics, com servidors, emmagatzemament, bases de dades, xarxes, programari i més, a través d'internet. En lloc de mantenir maquinari i programari físics, els usuaris i les organitzacions poden accedir a aquests recursos segons el que necessitin, pagant sol per allò que utilitzin, a través de proveïdors de serveis en el núvol. Aquest enfocament ofereix molts avantatges com escalabilitat, flexibilitat, eficiència en costs i possibilitat de delegar la gestió de la infraestructura a experts.

L'any 2006 Amazon va oferir per primera vegada S3 (un sistema d'emmagatzemament escalable i de baix cost) [2] i EC2 (màquines virtuals remotes configurables) [3]. EC2 seria el primer servei de computació ofert pels proveïdors del núvol, catalogat com IaaS (*Infrastructure as a Service*) [4]. La *IaaS* és un model on els clients poden aconseguir maquinari, com màquines virtuals, emmagatzematge i xarxa, tot això sota demanda sense la necessitat d'invertir i controlar els recursos físics. La computació al núvol compta amb més usuaris cada any i les empreses cada cop aposten més per aquest paradigma davant la metodologia tradicional d'utilitzar servidors físics en propietat. Tot i això, els usuaris han de tenir uns coneixements específics per fer un bon ús de les *IaaS*. Això implica saber aplicar una correcta configuració dels recursos, sabent identificar els tipus d'instància de la màquina virtual a usar, nombre d'instàncies, model de preu..., inclús els usuaris més experimentats poden arribar a no aprofitar adequadament totes les capacitats que ens proporcionen els serveis bàsics del núvol.

A mesura que les plataformes del núvol van anar popularitzant-se, es va identificar la necessitat d'atreure a aquells usuaris menys experimentats, facilitant la interacció amb els recursos disponibles. La necessitat d'una alternativa al model *IaaS* clàssic, que permetés una forma d'explotar els recursos del núvol amb uns coneixements tècnics més limitats, va augmentar clarament al llarg de l'última dècada. En resposta a aquesta necessitat, va aparèixer la computació sense servidor [5]. Aquesta nova proposta fa responsable el proveïdor del núvol de l'aprovisionament de recursos, l'escalabilitat i el maneig de les eines. Idealment, l'usuari sol s'encarrega d'escriure el codi de les seves aplicacions i és la plataforma qui s'encarrega dels aspectes operacionals.

Amazon Web Services [6] va ser el primer a llançar un servei sense servidor, AWS Lambda [7], una *FaaS* (*Function as a Service*) [8]. *FaaS* és un model de computació al núvol que permet als desenvolupadors executar individualment codi de funcions específiques sense estat, d'una forma altament escalable i eficient a nivell de costs. Aquestes funcions són sense estat, ja que han de rebre totes les dades necessàries com a paràmetres d'entrada i han de retornar tots els resultats desitjats com a sortida, no persisteix un estat intermediari que emmagatzemi tota aquesta informació. Aquestes funcions són executades com a resposta a esdeveniments, sense la necessitat de controlar o aprovisionar servidors. Els esdeveniments poden ser crides HTTP, canvis en una base de dades, càrrega de fitxers, temporitzadors... Quan un esdeveniment és activat, el proveïdor del núvol manega automàticament l'aprovisionament de recursos, escalabilitat, execució i manteniment. Un cop l'esdeveniment és activat, el proveïdor del núvol s'encarrega de gestionar i aprovisionar automàticament un entorn d'execució dins un container, aïllat de qualsevol altra funció, per executar el codi de forma agnòstica para l'usuari.

En definitiva, existeixen unes característiques que fan clara la diferència entre un model sense servidor (*o serverless)* i un model "amb servidor". Primer de tot, el maneig dels

recursos no és gestionat pel desenvolupador, deixant així al proveïdor del núvol la responsabilitat de l'aprovisionament, maneig o escalament de servidors. En segon lloc, tenim l'execució de funcions sense estat, és a dir, les dades d'entrada i les respostes a retornar s'han d'emmagatzemar en un sistema d'emmagatzemament extern. Finalment, una característica molt interessant, els costs d'execució. En el cas d'un model amb servidor, els pagaments es faran pel temps d'aprovisionament d'uns recursos predefinits, independentment de si són utilitzats o no. En canvi, en un model sense servidor, es cobra els usuaris en funció de l'ús real dels recursos, com el temps d'execució i la memòria, amb tarifes predefinides.

1.2 FaaS en l'anàlisi de dades

Avui en dia les empreses cada cop utilitzen més *FaaS* per agilitzar les seves operacions, millorar la seva eficiència i oferir serveis innovadors. El fet de tenir una gran elasticitat i un temps de resposta molt ràpid, fan que siguin un gran atractiu per abordar tasques de baixa latència i escala variable. En concret, podem parlar d'Airbnb, empresa que usa *FaaS* per processar i analitzar les dades de les reserves en temps real [9], o de Coca-Cola, que aprofita aquests serveis per l'anàlisi de dades de les vendes, monitoratge d'inventaris o generació d'informes [10]. També hi ha altres exemples on es treballen camps diferents, com pot ser el de Netflix, que fa servir *FaaS* per processar arxius multimèdia, incloent-hi la codificació de vídeos, manipulació d'imatges i preparació de contingut [11].

Els serveis *FaaS* han estat adoptats per tots els proveïdors del núvol importants. Alguns exemples poden ser AWS Lambda, Azure Functions [12], Google Cloud Functions [13]... Tots aquests serveis *FaaS*, normalment segueixen uns patrons comuns, que els fan molt apropiats per l'anàlisi de dades. Primer de tot, els desenvolupadors defineixen la funció que durà a terme una tasca específica, aquesta inclourà el codi a executar escrit en un llenguatge de programació com pot ser Python o Java. En segon lloc, seleccionarà l'esdeveniment que activarà l'execució de la funció, ja sigui una petició HTTP, un canvi en la base de dades, un temporitzador, etc. A continuació, serà una plataforma proveïdora del núvol qui s'encarregarà de la gestió de l'execució.

Els avantatges de *FaaS* són evidents i cada cop s'estudia més com portar-los a diferents àmbits per optimitzar el rendiment. L'anàlisi de dades a gran escala és un dels camps on ha augmentat més l'interès en aquests últims anys, donant lloc a un increment de la inversió i del nombre d'investigadors treballant en l'aprofitament d'aquest nou model. Pel volum de dades a analitzar, i la variabilitat de les càrregues de treball, encaixa perfectament amb la necessitat de comptar amb característiques com l'escalabilitat, elasticitat i un model de costs eficient.

Com a resultat d'aquesta nova tendència d'investigació van aparèixer *frameworks* d'anàlisis de dades com PyWren [14] i la seva extensió, Lithops [15]. Aquest és un *framework* de codi obert que busca simplificar el desenvolupament i execució de funcions *serverless* en diferents proveïdors del núvol. Entre altres, permet comunicar dades entre funcions i executar funcions independents en paral·lel utilitzant entorns de computació sense servidor, com poden ser plataformes *FaaS* com AWS Lambda.

El model de funcions presenta certes limitacions pel que fa a la compartició de dades entre funcions. La comunicació directa entre funcions a través de la xarxa és inviable o difícil. És per aquest motiu que Lithops busca mètodes alternatius com pot ser la comunicació a través de sistemes d'emmagatzematge del núvol com Amazon S3. Aquests tipus d'emmagatzematges basats en objectes tenen grans avantatges com la seva gran escalabilitat i durabilitat, però tenen un gran desavantatge quan volem buscar una comunicació ràpida, són sistemes amb una alta latència. A més, el fet d'usar un sistema com

aquest implica dues còpies de les dades compartides, la primera per escriure la informació a S3 i la segona quan es llegeixen les dades de S3.

L'objectiu d'una anàlisi de dades a gran escala és poder optimitzar al màxim els temps de processament. És per això que una forma d'accelerar aquests temps és intentant evitar fer servir sistemes d'emmagatzemament amb alta latència. Per aquest motiu, una optimització podria ser l'execució de les funcions en una mateixa màquina virtual, permetent una comunicació molt més ràpida entre elles gràcies a la utilització de sistemes d'emmagatzematge més eficients com memòria compartida [16]. A més, utilitzant aquests sistemes podem aconseguir una compartició de les dades sense haver de realitzar cap còpia en la lectura i obtenció de subconjunts de dades (*o slicing*), permetent l'accés directe a la informació.

1.3 Objectius

L'objectiu principal d'aquest treball de final de grau és estudiar quin és el benefici que es pot obtenir d'utilitzar tècniques de compartició de dades a través de memòria compartida i còpia zero [17] dins d'una mateixa màquina virtual. L'estudi ajudarà a decidir si serà útil la implementació d'aquest mecanisme de compartició dins de Seer [18], un sistema d'anàlisi de dades construït sobre el model de funcions. Aquest sistema està escrit amb Python, per això totes les implementacions treballaran amb aquest llenguatge de programació. Més concretament, els objectius del projecte són els següents:

1. Disseny i implementació d'un sistema de memòria compartida de còpia zero sobre Python multiprocessing [19] per minimitzar els temps d'intercanvi de dades entre processos i la utilització de memòria.
2. Comparació, a nivell teòric, de diferents mètodes de compartició de dades a través de memòria compartida com són multiprocessing.shared_memory [20] i pyarrow.plasma [21].
3. Estudi de llibreries de serialització de dades d'alt rendiment com PyArrow [22].
4. Estudi de diferents llibreries per interactuar amb Amazon S3.
5. Comparació dels temps d'execució, mida de les estructures de dades i consum de memòria davant l'escriptura i lectura d'informació mitjançant memòria compartida versus un sistema d'emmagatzematge remot d'alta latència com Amazon S3.
6. Comparació de la compartició de dades mitjançant memòria compartida versus disc i un sistema d'emmagatzematge remot d'alta latència com Amazon S3 mitjançant el model MapReduce.
7. Valoració de la integració del nou mecanisme de memòria compartida dins de Seer.

En l'àmbit formatiu els objectius són la familiarització a nivell teòric i pràctic amb diferents sistemes d'emmagatzematge per l'intercanvi de dades: distribuïts d'alta latència (*object store*) i locals de baixa latència (*shared memory).* Tot això, passant per una investigació i implementació dels mètodes de serialització i deserialització més eficients i la cerca de les interfícies d'interacció amb sistemes remots més ràpides. Finalment, aplicar tots els conceptes estudiats en entorns paral·lels Python i arquitectures distribuïdes *FaaS.*

2 Antecedents

2.1 Backends de compartició de particions

2.1.1 Sistemes d'emmagatzemament local

2.1.1.1 Multiprocessing Shared Memory

El mòdul `multiprocessing.shared_memory (shared_memory)` és un mòdul introduït per Python3.8 que proporciona una interfície d'alt nivell per crear i manipular regions de memòria compartida volàtil. Aquest mòdul permet a múltiples processos escriure i llegir un bloc comú de memòria com si fos dins el seu propi espai de memòria. La memòria compartida pot ser útil per la compartició d'informació entre processos que estan treballant de forma paral·lela o concurrent en una mateixa màquina amb múltiples nuclis (computadora local, instància d'EC2...), evitant així la necessitat d'utilitzar altres mètodes més lents com poden ser la utilització de disc. A més, amb una correcta serialització i utilització d'estructures, es pot arribar a aconseguir compartir informació entre processos evitant copies.

Quan un procés crea una instància de `shared_memory`, pot **a) reservar un nou bloc de memòria compartida** o **b) enllaçar la instància a un bloc prèviament reservat especificant-ne el nom**. Cada nou bloc de memòria està identificat per un nom únic, el qual serà usat per tot procés que es vulgui connectar i manipular aquesta zona d'informació. La vida de la memòria compartida està lligada directament al procés que l'ha creat. Això significa que quan el procés creador acaba, la memòria compartida és alliberada i deixa de ser accessible per la resta de processos.

Cada memòria compartida pot ser utilitzada per múltiples processos i s'ha de garantir que el *garbage collector* [23] faci una neteja adequada dels recursos, quan la memòria ja no es necessiti. Per complir amb aquest objectiu, un cop ja no s'hagi de fer ús d'un bloc de memòria, cada procés ha de cridar al mètode *close()* sobre la instància de `shared_memory` corresponent, alliberant així els recursos associats. Finalment, per alliberar una zona de memòria compartida, es crida una sola vegada entre tots els processos el mètode *unlink()*. Aquest mètode serà cridat per l'últim procés que faci ús de la memòria, ja que un cop sigui cridat, la memòria deixarà de ser accessible per qualsevol procés.

2.1.1.2 Plasma

Plasma és un *object store* [24] a memòria desenvolupat per Apache Arrow [25]. Plasma permet emmagatzemar objectes immutables a memòria compartida que poden ser accedits per múltiples clients des de diferents processos. Davant la tendència que ens porta cap a màquines amb múltiples nuclis cada cop més grans, Plasma permet optimitzar l'execució local de càrregues de treball *Big Data* [26].

Aquest o*bject store* va ser dissenyat amb l'objectiu d'abordar els reptes de la compartició de dades i la comunicació entre processos entre diferents llenguatges de programació de forma eficaç. En concret, ofereix les següents característiques:

- **Memòria compartida:** Plasma utilitza mecanismes de memòria compartida per permetre a múltiples processos accedir les mateixes dades sense realitzar còpies. Això resulta en una reducció de *l'overhead* de transferència de dades i a conseqüència, una millora del rendiment.

- **Object Store:** Plasma actua com un *object store* on els objectes de dades poden ser emmagatzemats, accedits i eliminats. A més, compta amb un maneig de memòria, *garbage collector* i polítiques pròpies d'eliminació d'objectes.
- **Còpia zero:** La memòria compartida de Plasma permet que diferents processos accedeixin a la mateixa regió de memòria directament, sense la necessitat de copiar aquesta informació en l'espai de memòria del procés. Plasma gestiona la còpia zero a través d'estructures de dades PyArrow.
- **Compatibilitat entre llenguatges:** Com aquest *object store* a memòria forma part d'Apache Arrow, Plasma també suporta la compartició d'informació entre processos implementats amb diferents llenguatges.

Plasma es va desenvolupar inicialment com a part de Ray [27] i l'any 2017 va passar a formar part d'Apache Arrow. A pesar de la intenció de treballar amb aquesta eina per poder aconseguir una forma ràpida i eficient de compartir informació, Plasma com a llibreria independent ha quedat obsoleta des de la versió 10.0.0 d'Apache Arrow. Això ha estat degut a la falta de manteniment d'aquest projecte de codi obert, el qual té un programari complex i requereix uns usuaris amb uns amplis coneixements tècnics per realitzar el manteniment. Aquesta manca d'atenció al projecte ha fet que a pesar de poder continuar treballant amb aquest *object store*, doni certs problemes i deixi inoperatives certes operacions de l'API (*Application Programming Interface*). Tot i això, Plasma encara és utilitzada com a unitat integrada en Ray per la compartició interna de dades.

2.1.1.3 Shared Memory vs Plasma

Tant Multiprocessing Shared Memory com Plasma són tecnologies molt eficients que permeten la compartició de memòria entre múltiples processos en Python. Tot i això, hi ha certes diferències que poden portar a fer una anàlisi sobre quin dels dos tipus de memòria compartida pot arribar a ser més eficient per compartir grans quantitats de particions entre múltiples processos. Aquestes són les diferències a destacar:

- **Gestió dels recursos:** Plasma compta amb un servidor, anomenat Plasma Store, per gestionar els recursos de memòria compartida i facilitar l'accés i manipulació de les dades des dels diferents processos. Aquest servidor és el punt central per la gestió dels objectes a memòria compartida, i són els diferents clients, normalment processos, mitjançant IPC (*inter-process comunication*) [28], els que es comuniquen amb ell per allotjar, accedir i alliberar regions de memòria. En canvi, `shared_memory`, no compta amb cap servidor centralitzat que gestioni els recursos a memòria. Aquesta llibreria proporciona una interfície per crear i accedir a recursos de memòria directament. Això significa que davant d'un intercanvi de particions entre múltiples processos, no existirà un *overhead* de comunicació entre els clients i el servidor, com passa a Plasma. A Plasma, a mesura que el nombre de processos creixen, el servidor centralitzat es podria convertir en un coll d'ampolla. Per tant, Plasma podria limitar el rendiment d'execucions amb un elevat grau de paral·lelisme.
- **Interoperabilitat:** Plasma forma part del projecte Apache Arrow, plataforma que, entre altres, destaca per la seva interoperabilitat. Això fa que Plasma pugui ser utilitzat des de múltiples llenguatges de programació i sistemes diferents que siguin compatibles amb el format columnar d'Arrow [29]. En canvi, la `shared_memory` està limitada a treballar amb Python.
- **Immutabilitat:** Plasma emmagatzema objectes immutables a memòria compartida i permet l'accés a aquests amb còpia zero. Això significa que múltiples processos poden accedir directament a les mateixes dades sense la

necessitat de realitzar cap còpia local. `shared_memory` emmagatzema diferents tipus d'informació mutable a memòria, la qual pot ser accedida també amb còpia zero per múltiples processos. Aquest accés serà sense còpies sempre que es faci servir una serialització i deserialització amb un format correcte, com pot ser el format columnar. El fet que Plasma treballi amb immutabilitat dels objectes a memòria, suprimeix la necessitat de sincronitzar l'accés a les dades. Amb multiprocessing, tenim la possibilitat de modificar les dades compartides, però s'ha de tenir en compte la sincronització dels processos que volen accedir i modificar les mateixes dades.

- **Serialització i deserialització:** Plasma compta amb un sistema de serialització dels objectes preceptiu i automàtic. Es serialitzen tots els objectes en format columnar, per permetre una lectura i *slicing* de la informació amb còpia zero. `shared_memory` deixa la responsabilitat de la serialització en mans dels usuaris. Això provoca que l'usuari hagi de tenir uns coneixements més elevats sobre la manipulació de la informació en memòria, però ens dona major llibertat a l'hora de guardar els elements com desitgem. La còpia zero en les lectures serà resultat d'una implementació correcta de la serialització i deserialització dels objectes.
- **Manteniment:** Plasma es va quedar obsoleta a partir de la versió 10.0.0 d'Apache Arrow, cosa que significa que a poc a poc, a pesar que alguns usuaris encara en fan ús, deixarà de funcionar correctament. En contraposició, `shared_memory` està en continu manteniment pel fet de ser part de `multiprocessing`, una llibreria estàndard de Python.
- **API:** Les dues llibreries compten amb una API extensa per una fàcil manipulació de la memòria compartida. `shared_memory` requereix un major coneixement per part de l'usuari sobre la gestió de memòria per aprofitar els recursos al màxim, però es pot arribar a manipular la informació amb una major precisió. Tot i això, disposa d'una gran comunitat i quantitat d'experts per poder solucionar els problemes. Plasma té una API transparent, molt fàcil i intuïtiva per poder interaccionar amb la memòria compartida, però no totes les operacions funcionen correctament. El fet d'haver quedat obsoleta fa un temps es reflecteix en errors de funcionament, donant lloc a resultats erronis o amb un comportament inesperat.

Tots dos models de memòria compartida són eficients i poden arribar a ser molt útils i interessants d'estudiar. El problema és que si s'ha d'implementar un sistema de compartició de dades que hagi de ser funcional a llarg termini, l'opció de plasma no és la més intel·ligent, ja que actualment està obsoleta i no tindrà actualitzacions.

Com a part de la investigació, també es va estudiar l'ús de les dues interfícies de memòria compartida. Es va comprovar que el rendiment d'usar Plasma, respecte a un ús correcte de `shared_memory`, era lleument inferior. Per tant, optem per `shared_memory` per a la nostra implementació.

2.1.1.4 Còpia zero

La còpia zero és una tècnica de gestió de dades on la informació és transferida o compartida entre diferents processos sense la necessitat de copiar dades d'una zona de memòria a una altra. Això implica utilitzar punters i referències a la zona original de memòria compartida, permetent que els processos puguin llegir informació de zones fora del seu espai de memòria local com si realment en fossin propietaris. La còpia zero només serà

possible en la compartició d'informació entre processos que s'executen en una mateixa màquina, amb la possibilitat d'accedir a una memòria conjunta.

Les operacions de còpia zero són molt eficients donat que redueixen la memòria utilitzada i la sobrecàrrega de CPU que requereix la còpia de dades d'un espai de memòria a un altre. Aconseguir treballar amb aquests avantatges requereix un maneig intel·ligent de les dades i una sincronització adequada entre els processos que intervenen en la compartició. A més, s'ha de seleccionar un medi adequat on emmagatzemar la informació i l'ús de tècniques de serialització i deserialització correctes.

2.1.2 Sistemes d'emmagatzemament remot

2.1.2.1 AWS

AWS és una plataforma de serveis en el núvol d'Amazon. Aquesta ofereix una àmplia gamma de serveis d'infraestructura i plataforma que permeten a individus, empreses i organitzacions implementar aplicacions i serveis en línia sense la necessitat d'invertir en maquinari físic i gestió d'infraestructura. És el proveïdor del núvol més utilitzat i amb més prestigi a escala internacional.

Els serveis d'AWS cobreixen una àmplia varietat d'àrees, incloent-hi computació, emmagatzemament, bases de dades, anàlisis, aprenentatge automàtic, seguretat, internet de les coses (IoT) [30], desenvolupament d'aplicacions... Aquests serveis estan dissenyats per ser escalables i flexibles, el que permet als usuaris pagar només pels recursos que es fan servir i ajustar-los segons les seves necessitats.

Algun dels serveis més populars i coneguts d'AWS inclouen:

- **Amazon EC2:** Ofereix instàncies virtuals que permeten als usuaris executar aplicacions en màquines virtuals configurables.
- **Amazon S3:** Proporciona emmagatzemament escalable i d'alta disponibilitat en el núvol per dades i arxius.
- **Amazon Lambda:** Permet executar codi sense necessitat d'aprovisionar o administrar servidors.

Figura 1. Logotip d'AWS

2.1.2.1.1 Amazon S3

Amazon Simple Storage Service és un servei d'emmagatzemament d'objectes que ofereix escalabilitat, disponibilitat de dades, seguretat i un alt rendiment. S3 s'utilitza per emmagatzemar gran varietat de tipus de dades, des d'arxius individuals fins grans quantitats de dades estructurades o no estructurades, com imatges, vídeos, arxius de registre... A continuació, es presenten alguns dels conceptes clau relacionats a Amazon S3 essencials per entendre el seu funcionament:

- **Buckets**: Un *bucket* és un contenidor fonamental en Amazon S3 per emmagatzemar objectes (arxius) [31]. Cada objecte ha d'estar contingut en un *bucket*. Els *buckets* en Amazon S3 són globals, és a dir, cada un d'ells ha de tenir un nom únic a través de tota la xarxa d'AWS.
- **Objectes**: Els objectes són els elements bàsics d'emmagatzemament en S3 [32]. Representen els arxius emmagatzemats i estan formats per les dades en si, un identificador únic anomenat clau i metadades opcionals. Les claus s'estructuren com rutes d'arxius i s'utilitzen per accedir als objectes.

Des de Python, podem interaccionar amb S3 a través de diferents llibreries amb diferents APIs. A continuació presentem les dues opcions principals: boto3 i s3fs.

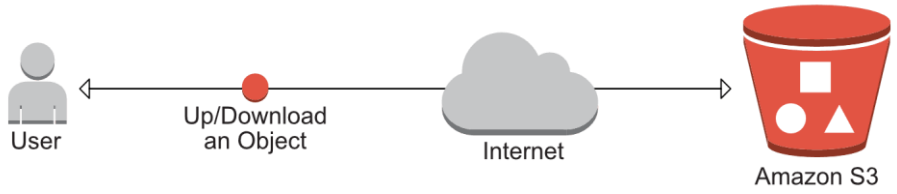

Figura 2. Diagrama que representa la interacció d'un usuari amb un *bucket* d'Amazon S3 per carregar o descarregar un objecte [33].

2.1.2.1.1.1 Boto3

Boto3 [34] és la biblioteca de *software* de Python que proporciona una interfície de programació d'aplicacions (API) a baix nivell per interactuar amb els serveis d'Amazon Web Services. Aquesta biblioteca és el SDK (*Software Development Kit*) oficial per Python quan treballem amb AWS.

En el meu cas utilitzaré clients de boto3, que és una funció proporcionada per la biblioteca que permet crear instàncies de clients capaços d'interactuar amb els serveis AWS. Aquesta instància la pots associar a un servei específic i al teu compte d'AWS a través de l'especificació de les claus d'usuari (*aws_access_key_id*, *aws_secret_access_key*) i la regió en la qual es troba el servei a utilitzar.

En el cas de treballar amb S3 tindrem diferents opcions com pot ser carregar objectes, descarregar objectes o enumerar *buckets*.

2.1.2.1.1.2 S3fs

S3fs [35] et permet simular el muntatge d'un *bucket* de s3 com un sistema de fitxers local, permetent interactuar amb els objectes de s3 com si fossin arxius i directoris d'un sistema de fitxers tradicional. Aquesta abstracció està pensada per treballar de forma transparent amb un únic servei, Amazon S3, seguint regles POSIX. A més, la configuració necessària per interactuar amb S3 és menor, ja que l'API abstrau detalls de baix nivell.

2.2 Mètodes de serialització

2.2.1 Apache Arrow

Apache Arrow és una plataforma de codi obert i de desenvolupament de *software* per aplicacions d'alt rendiment que processen i transporten grans conjunts de dades en memòria. Està dissenyat per millorar tant el rendiment d'algorismes d'anàlisi de dades, com l'intercanvi de dades entre sistemes i llenguatges. Bàsicament, serveix d'intermediari en el

processament de dades, perquè aquestes siguin consumides de forma ràpida i eficient per diferents components o serveis. Una característica important d'Apache Arrow és el format columnar per a les dades serialitzades en memòria. La plataforma, continuant amb el context de l'anàlisi de dades a gran escala, es centra a treballar amb la serialització i deserialització de dades estructurades (informació organitzada en columnes amb un esquema definit). Tot i això, també té la capacitat de treballar amb cadenes de caràcters, dades binàries...

El projecte Arrow és interoperable, compta amb llibreries que et permeten treballar amb el format columnar de dades amb diferents llenguatges de programació com C++, C#, Go, Java, JavaScript, Python, Julia i Rust. Això permet que sigui una opció molt atractiva pels usuaris que necessitin processar dades massives independentment del llenguatge amb el qual estigui treballant. En concret, la llibreria d'Arrow per Python s'anomena PyArrow.

Figura 3. Logotip d'Apache Arrow

2.2.1.1 Format Columnar

Apache Arrow serialitza les dades amb format columnar. A diferència del format en fila tradicional, on cada fila de dades s'emmagatzema de manera contigua, són les dades de cada columna les que es mantenen contigües en memòria. Cada columna que forma l'estructura de dades representa un atribut o un camp específic. El format columnar d'Arrow inclou una forma de representar les estructures de dades en memòria, la serialització de les metadades i un protocol per serialitzar i importar la informació. Aquest format pot proporcionar diferents beneficis:

- **Accessos seqüencials:** Quan les dades estan emmagatzemades en format columnar, tots els valors d'una columna estan guardats de forma contigua en memòria. Aquest fet és molt beneficiós en escenaris on es necessiten accessos seqüencials a un únic atribut de les dades, com un escaneig de les dades per a processaments analítics o filtratges.
- **Accés aleatori eficient:** En un format columnar, accedir a un element específic d'una fila i columna en particular és eficient donada la forma com està emmagatzemada l'estructura de dades. Cada columna s'emmagatzema per separat, fet que provoca que els càlculs dels índexs per accedir a un valor específic siguin directes i no depenen de la mida del conjunt de dades. Això dona un accés aleatori de cost constant (O(1)), cosa altament desitjable per consultes interactives i cerques.
- **SIMD (*Single Instruction/Multiple Data*):** Arrow organitza les dades d'una forma que afavoreix l'execució d'instruccions SIMD. Les operacions SIMD permeten als processadors moderns executar una mateixa operació sobre múltiples elements de dades en paral·lel en un sol cicle de rellotge.

- **Còpia zero*:*** La deserialització PyArrow permet accedir a les dades emmagatzemades en memòria sense còpies. Aquest accés amb còpia zero funciona tant per fer una lectura com una obtenció del subconjunt de dades (*o slicing*), el que significa que múltiples processos poden accedir a les dades sense una duplicació innecessària de la memòria. PyArrow proporciona un conjunt d'eines per poder treballar directament amb la vista de la memòria.
- **Serialització i deserialització**: Quan passem a treballar amb estructures Arrow, la serialització i deserialització de les dades al format columnar és automàtica i portada a terme amb uns temps d'execució molt reduïts, inclús treballant amb grans quantitats de dades.

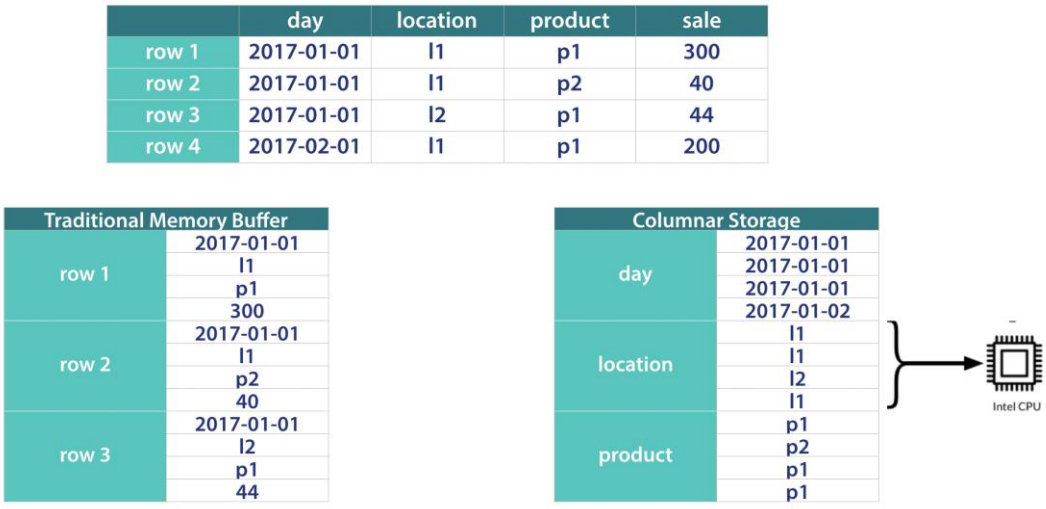

Figura 4. Format en files versus format columnar [36]

2.2.1.2 Estructures de PyArrow

Apache Arrow defineix estructures de dades de matrius columnars en combinar metadades de tipus amb *buffers* de memòria.

Les metadades ens proporcionen la informació necessària per interpretar i processar correctament els valors de les columnes que formen l'estructura de dades columnar. En concret, descriuen els tipus utilitzats i l'esquema que segueixen els diferents elements emmagatzemats en memòria.

Els *buffers* de memòria estan implementats sobre el tipus `arrow:Buffer` de C++, que permet que les classes *array* d'alt nivell interactuïn de forma segura amb la memòria assignada. És la classe base de tots els *buffers* Arrow i proveeix una vista de còpia zero d'una àrea de memòria contigua. Per evitar treballar amb punters de dades, `arrow:Buffer` proporciona una abstracció genèrica encapsulant un punter a memòria i una mida del que ocupen les dades. Aquesta abstracció també permet ser *zero-copy sliced*, és a dir, permet que un *buffer* referencií a un altre sense la necessitat de copiar les dades, obtenint així un nou *buffer* que apunta a un subconjunt de la memòria del *buffer* original, el qual encara seria propietari d'aquesta.

Un cop explicats aquests dos conceptes essencials que componen les estructures de dades de matrius columnars, ja es pot explicar les diferents estructures de dades que estan disponibles en Python. Aquestes estructures de dades estan compostes per una sèrie de classes interrelacionades:

- **pyarrow.DataType:** Classe de la llibreria PyArrow utilitzada per representar i definir els tipus de dades que poden ser utilitzats en l'ecosistema Arrow. Els tipus Arrow estan dissenyats per optimitzar l'emmagatzemament, serialització i processament de les dades. PyArrow ens dona la possibilitat de treballar amb els següents tipus: tipus primitius de mida fixa (números, booleans, dates...), tipus de mida variable (strings i binaris), *nested types* (llistes, map, struct i union) i tipus diccionari.
- **pyarrow.Schema:** Classe usada per definir l'estructura d'un *dataset*, particularment treballant amb un format columnar. Estableix els noms de les columnes i els tipus en un pyarrow.RecordBatch i una pyarrow.Table, proporcionant una idea de com les dades estan organitzades i guardades. A més, també compta amb metadades de les columnes. Per exemple, si es converteix un Pandas DataFrame a un esquema de PyArrow, l'esquema pot retenir metadades sobre els tipus de Pandas per permetre una conversió posterior als tipus originals.
- **pyarrow.Array:** Aquesta classe implementa una estructura de dades utilitzada per guardar *buffers* de memòria que defineixen un únic fragment contigu de dades en format columnar. Cada pyarrow.Array emmagatzema dades d'un tipus específic en una zona de memòria contigua, fent possible un emmagatzemament òptim i una manipulació i anàlisis de les dades eficient. Un pyarrow.Array està format per una o més instàncies de pyarrow.Buffer que apunten a les dades físiques en memòria i tenen metadades i informació sobre l'estructura en si.
- **pyarrow.RecordBatch:** Representa un conjunt d'instàncies de pyarrow.Array de mida igual organitzades amb un pyarrow.Schema específic.
- **pyarrow.Table:** Representa una col·lecció de múltiples instàncies de pyarrow.RecordBatch que comparteixen el mateix esquema. És útil quan es treballa amb conjunts de dades que s'han separat en diferents lots i han de ser combinats i manipulats en conjunt.

Pyarrow ofereix una gran varietat de possibilitats per convertir les estructures natives de Python a estructures de dades columnars de Pyarrow. Per exemple, L'API de Pyarrow compta amb la possibilitat de fer la conversió a estructures Pyarrow des de llistes i diccionaris de Python, NumPy arrays o Pandas Dataframes. Tot això a través d'un conjunt de funcions de conversió i constructors que permeten una transformació ràpida i eficient.

Aquesta llibreria també ofereix la possibilitat de guardar dades tabulars de forma eficient amb una alta compressió mitjançant fitxers Parquet. Aquests són molt utilitzats per emmagatzemar informació en sistemes remots com Amazon s3.

2.2.2 Pickle

Pickle és un mòdul del llenguatge de programació Python, no compatible amb altres llenguatges, que permet serialitzar una àmplia gamma d'objectes Python, incloent-hi tant *built-in types* (enters, strings, floats...) com classes i objectes definits pels usuaris. En aquest cas, es tracta d'un mòdul per tota mena de dades, no sol les dades estructurades.

La serialització en pickle es pot utilitzar per emmagatzemar un objecte Python en un fitxer, a memòria o enviar-lo a través de la xarxa. La serialització d'aquest mòdul implica convertir els objectes en un format binari compacte que pot ser fàcilment guardat o transmès. La deserialització du a terme el procediment invers, converteix el conjunt de bytes en l'objecte Python original amb la seva estructura i valor.

La utilització de pickle implica la còpia de dades des de la zona de memòria origen fins a l'espai de memòria del procés. Aquesta còpia es fa, ja que el codi Python, fins que no es fa la deserialització dels bytes, no pot comprendre la informació codificada. Aquest fet significa la desaparició de qualsevol possibilitat de treballar amb còpia zero en les lectures i *slicing* de les dades, com si es pot arribar a aconseguir amb la serialització Arrow.

2.3 Model MapReduce

El model MapReduce permet dur a terme un processament paral·lel de grans conjunts de dades en dues fases consecutives, la fase de map i la fase de reduce.

La fase de mapatge de les dades consisteix en la crida d'un nombre determinat de funcions en paral·lel i l'assignació d'una porció de les dades d'entrada (o partició) a cada una de les funcions. Cada una de les funcions aplicarà una operació predefinida sobre la partició. En aquest punt, és molt important tenir en compte una correcta divisió de les dades d'entrada entre els diferents *mappers*, d'aquesta forma es podrà tenir un bon balanceig de la càrrega de feina que donarà com a resultat uns temps d'execució similars entre les funcions.

Un cop totes les funcions de la fase map hagin acabat les seves execucions, s'inicia la fase de reduce. En aquesta fase, es torna a cridar a un conjunt de tasques paral·leles que reuneixen resultats parcials de la fase de mapeig i apliquen una segona operació predefinida sobre les dades recopilades.

Entre les dues fases de *mapping* i *reduce* es produeix una operació anomenada *shuffling*. Aquest pas intermediari implica la transferència de les particions necessàries generades pels *mappers* als diferents *reducers*. En el cas de treballar amb una comunicació *all-to-all*, el *shuffling* implica que tots els *mappers* han d'enviar una partició a tots i cada un dels *reducers*, donant lloc a una gran transferència de dades entre múltiples funcions que pot arribar a ser un coll d'ampolla en sistemes amb un paral·lelisme elevat. Per exemple, en el cas de tenir M *mappers* i N *reducers*, podem arribar a tenir N x M arxius intermedis a transmetre entre funcions, on el nombre de particions a transmetre creix de forma quadràtica amb relació al nombre de tasques a executar.

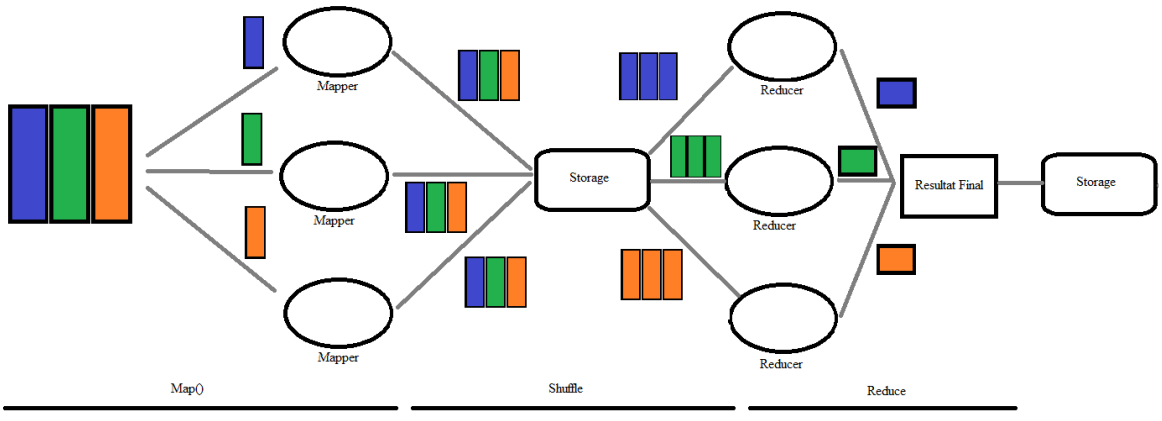

Figura 5. Exemple de MapReduce amb comunicació tots a tots

Com es pot observar en la figura 15, la fase de *shuffle* en un model MapReduce utilitzant *FaaS*, implica la utilització d'un sistema d'emmagatzemament intermediari per poder compartir les particions entre elles.

Les *FaaS* no són adreçables directament a través de la xarxa, és per això que s'ha de transmetre la informació entre elles mitjançant un emmagatzematge remot. En el cas

d'executar les funcions en sistemes com Amazon Lambda, la compartició de particions es pot fer a través d'emmagatzemaments d'alta latència com pot ser Amazon S3.

Si s'executessin les funcions en una màquina local o màquina virtual EC2, la compartició de dades es podria fer a través d'altres medis més ràpids com pot ser memòria compartida i disc.

3 Implementacions

3.1 Implementació de l'escriptura i lectura de dades en diferents sistemes d'emmagatzematge

Les comparticions de dades entre funcions executades en una mateixa màquina virtual es poden realitzar mitjançant múltiples sistemes d'emmagatzemament intermediaris. L'objectiu principal del treball de final de grau és poder avaluar el benefici d'utilitzar tècniques de compartició de dades a través de memòria compartida i còpia zero. És per això que s'han dut a terme una sèrie d'implementacions pensades per fer unes comparatives que ajudin a treure conclusions.

En primer lloc, s'han realitzat dues implementacions sobre multiprocessing.shared_memory. La primera busca una compartició de dades que no impliqui còpies en la lectura i la segona treballa per compartir informació de forma veloç, però compta amb sobrecàrregues de CPU i memòria donades certes còpies. Per aconseguir l'intercanvi d'informació amb un accés a les dades amb còpia zero, s'utilitza la serialització al format columnar donada per la llibreria PyArrow. La segona implementació és obtinguda mitjançant la llibreria Pickle, capaç de codificar i descodificar ràpidament una àmplia gamma d'objectes Python. Aquestes dues implementacions seran utilitzades en l'estudi per demostrar les diferències de rendiment utilitzant un mecanisme de còpia zero i un altre que impliqui còpies, fent servir el mateix sistema d'emmagatzemament intermediari.

En segon lloc, s'han dut a terme dues implementacions que treballen amb Amazon S3 utilitzant les llibreries s3fs i boto3. L'estudi podrà utilitzar aquestes per conèixer quin es el mecanisme de compartició de dades més eficient fent ús d'Amazon S3. A més, seran usades per conèixer les diferències de rendiment entre l'ús d'un sistema intermediari d'alta latència i un de memòria compartida amb còpia zero.

3.1.1 multiprocessing.shared_memory i serialització PyArrow

3.1.1.1 Escriptura de dades a multiprocessing.shared_memory

```
@profile
def put_df_arrow(df, usingPyArrow = False):
    # Conversió del Pandas DataFrame a PyArrow RecordBatch
    if(usingPyArrow):
        table = (pyarrow.Table.from_pandas(df)).combine_chunks()
        record_batch = table.to_batches(max_chunksize=sys.maxsize)[0]
    else:
        record_batch = pa.RecordBatch.from_pandas(df)

    # Determina la mida del buffer a sol·licitar des de la memòria
compartida
    mock_sink = pa.MockOutputStream()
    stream_writer = pa.RecordBatchStreamWriter(mock_sink,
record_batch.schema)
    stream_writer.write_batch(record_batch)
    stream_writer.close()
```

```
    data_size = mock_sink.size()

    # Creació de la memòria compartida amb la mida determinada i obtenció
del buffer
    sm_put = SharedMemory(create=True, size=(data_size))
    buffer = pa.py_buffer(sm_put.buf)

    # Escriptura del PyArrow RecordBatch a la memòria compartida
    stream = pa.FixedSizeBufferWriter(buffer)
    stream_writer = pa.RecordBatchStreamWriter(stream,
record_batch.schema)
    stream_writer.write_batch(record_batch)
    stream_writer.close()

    # Eliminació de totes les referències existents a memòria compartida
    del stream_writer
    del stream
    del buffer
    if(usingPyArrow):
        del table
    del record_batch

    # Alliberament els recursos lligats a la memòria compartida
    sm_put.close()
    return sm_put.name
```

En aquest codi es pot observar l'escriptura d'un Pandas Dataframe a memòria compartida mitjançant una serialització PyArrow. En concret les diferents fases del codi són les següents:

1. **Conversió del Pandas Dataframe a PyArrow RecordBatch:** En aquest cas, depenent de si és treballa amb un DataFrame amb tipus PyArrow o amb tipus Python, s'han de convertir les dades estructurades a `pyarrow.RecordBatch` d'una forma o una altra.
   En el cas que el DataFrame tingui valors del tipus PyArrow, al fer la conversió a una estructura de PyArrow, automàticament s'intenten agafar les columnes del DataFrame com `pyarrow.ChunkedArray`. Aquest tipus d'arrays estan formats per múltiples `pyarrow.Array`, continus o no continus entre ells en memòria. Com s'ha comentat en l'apartat 2.2.1.2 (Estructures de PyArrow), l'estructura columnar d'un `pyarrow.RecordBatch` està formada per múltiples `pyarrow.Array`, on cada columna ha d'estar emmagatzemada en una zona de memòria contigua. No es pot obtenir aquesta estructura a partir cadenes que no estan guardades en una única regió contigua de memòria. Per solucionar aquest problema, primer es fa la conversió a `pyarrow.Table`, estructura que si accepta cadenes disperses en memòria, i després es realitza

l'agregació dels múltiples `pyarrow.Array` de cada columna a partir del mètode *combine_chunks()*.
Gràcies a aquesta combinació ja tindrem les columnes emmagatzemades de forma contínua en memòria i podem aconseguir l'estructura de dades `pyarrow.RecordBatch` utilitzant el mètode *table.to__batches(max_chunksize=sys.maxsize)[]*, en el qual es força que doni com a resultat un únic *batch*, especificant la mida màxima del *chunk* com un valor virtualment inabastable. A partir del `pyarrow.RecordBatch` es farà l'escriptura a memòria compartida.
En el cas que el DataFrame tingui valors del tipus Python, la conversió serà directa a `pyarrow.RecordBatch`. Això és donat que al fer la conversió del DataFrame a una estructura de tipus PyArrow, les columnes són agafades com `pyarrow.Array`.

2. **Determina la mida del buffer a sol·licitar des de la memòria compartida:** Per poder saber la mida exacta de memòria que es necessita per emmagatzemar el `pyarrow.RecordBatch` s'utilitza `pyarrow.MockOutputStream()`. Es simula una escriptura de la informació, en la que no es fa cap còpia ni retenció de les dades, per poder saber exactament la quantitat de memòria compartida a reservar.
3. **Creació de la memòria compartida amb la mida determinada i obtenció del *buffer*:** Es crea una instància de `shared_memory` encarregada d'emmagatzemar el `pyarrow.RecordBatch` obtingut a partir del Pandas Dataframe inicial. Després s'aconsegueix el *buffer*, de tipus `pyarrow.py_buffer`, que es farà servir per guardar les dades. Aquest representa una vista de còpia zero d'una zona de memòria contigua.
4. **Escriptura del PyArrow RecordBatch a la memòria compartida:** S'escriu el pyarrow.RecordBatch dins el *buffer* de memòria compartida. Per escriure lots de dades de forma eficient en un *buffer* de mida fixa ja creat, utilitzem `pyarrow.FixedSizeBufferWriter` i `pyarrow.RecordBatchStreamWrite`r, una interfície per escriure *record batches* seguint un esquema determinat a un *stream*.
5. **Eliminació totes les referències existents a memòria compartida:** Per poder alliberar els recursos de memòria compartida a través del *sm_get.close()*, primer s'han d'eliminar totes les referències existents a aquesta memòria. És per això que s'aplica un *del*, comanda usada per eliminar qualsevol referència a un objecte, sobre totes les variables que estan relacionades a nivell de referència amb la memòria compartida.
6. **Alliberament els recursos lligats a la memòria compartida:** cridem a *close()* de la instància de `shared_memory`.

3.1.1.2 Lectura de dades des de multiprocessing.shared_memory

```
@profile
def get_df_arrow(sm_get_name, usingPyArrow = False):
    # Connexió amb la memòria compartida que emmagatzema el PyArrow
RecordBatch
    sm_get = SharedMemory(name = sm_get_name, create = False)
    buffer = pa.BufferReader(sm_get.buf)
```

```
# Lectura del PyArrow RecordBatch
reader = pa.RecordBatchStreamReader(buffer)
record_batch = reader.read_next_batch()

# Conversió del PyArrow RecordBatch a Pandas Dataframe
if(usingPyArrow):
    data = record_batch.to_pandas(types_mapper = pd.ArrowDtype)
else:
    data = record_batch.to_pandas()

print(data)

# Eliminació de totes les referències existents a memòria compartida
del data
del buffer
del reader
del record_batch

# Alliberament els recursos lligats a la memòria compartida
sm_get.close()
sm_get.unlink()
```

La lectura del DataFrame segueix un procediment semblant a l'escriptura però amb l'ordre invers. En concret, les diferents fases del codi són les següents:

1. **Connexió amb la memòria compartida que emmagatzema el PyArrow RecordBatch:** Per poder obtenir l'estructura de dades, primer es fa una connexió a la memòria compartida que l'emmagatzema a partir del nom, el qual és un identificador únic. Després es crea un lector de còpia zero d'objectes convertibles a *buffer* de PyArrow.
2. **Lectura del PyArrow RecordBatch:** En aquesta part del codi es produeix la lectura del `pyarrow.RecordBatch` de memòria compartida mitjançant un `pyarrow.RecordBatchStreamReader`, una interfície per accedir als *record batches* des d'una font d'entrada en format stream.
3. **Conversió del PyArrow RecordBatch a Pandas Dataframe:** Depenent de si volem retornar el Pandas DataFrame amb tipus PyArrow o amb tipus Python, utilitzarem una conversió o una altra. Si utilitzem el mètode *to_pandas()* de `pyarrow.RecordBatch` sense cap paràmetre, els valors del Pandas DataFrame resultant seran de tipus Python, tipus per defecte d'aquesta estructura de dades. En canvi, si volem obtenir els tipus PyArrow en els valors del Pandas DataFrame, hem d'especificar el paràmetre *types_mapper* amb el valor *pd.ArrowDtype*. D'aquesta manera, l'estructura de dades resultant comptarà amb els tipus de PyArrow.
4. **Eliminació totes les referències existents a memòria compartida:** Aquest és un punt crític per poder alliberar els recursos relacionats amb la memòria

compartida. Com es pot observar, s'eliminen totes les variables que tenen referències a la memòria. En el cas de treballar amb *zero-copy*, és essencial eliminar també la variable data, la qual és una referència a les dades del DataFrame, no és una còpia d'aquestes.

3.1.2 multiprocessing.shared_memory i serialització Pickle

3.1.2.1 Escriptura de dades a multiprocessing.shared_memory

```
@profile
def put_df_pickle(data):
    # Serialització del Pandas DataFrame
    pickled_df = pickle.dumps(data)
    data_size = len(pickled_df)

    # Creació de la memòria compartida amb la mida determinada
    sm_put = SharedMemory(create=True, size=data_size)
    sm_put.buf[:data_size] = pickled_df

    # Alliberament els recursos lligats a la memòria compartida
    sm_put.close()
    return sm_put.name
```

L'escriptura a memòria compartida utilitzant la llibreria Pickle resulta més senzilla d'implementar. Com es pot observar es comença amb una serialització de les dades i una obtenció del nombre de bytes a reservar. Es continua creant la memòria compartida amb la mida corresponent i emmagatzemant les dades serialitzades en aquesta a través del *buffer*. Finalment, s'elimina els recursos enllaçats amb la memòria compartida amb el mètode *close()*.

3.1.2.2 Lectura de dades des de multiprocessing.shared_memory

```
@profile
def get_df_pickle(sm_get_name):
    # Connexió amb la memòria compartida que emmagatzema les dades
serialitzades
    sm_get = SharedMemory(name = sm_get_name, create = False)

    # Lectura i deserialització de les dades en format pickle
    data = sm_get.buf[:]
    df = pickle.loads(data)

    # Eliminació de totes les referències existents a memòria compartida
    del data
```

```
    # Alliberament els recursos lligats a la memòria compartida
    sm_get.close()
    sm_get.unlink()
    return df
```

La lectura utilitzant la deserialització amb el format pickle comença amb una connexió a la memòria compartida que emmagatzema les dades mitjançant el seu nom. A continuació, es fa una lectura de les dades a través del *buffer* de la memòria i una deserialització d'aquestes per retornar l'objecte al format original. Finalment, s'esborra la variable data que té una referència a memòria compartida mitjançant el seu *buffer* i s'acaba tancant i desenllaçant la memòria compartida.

3.1.3 Amazon S3 utilitzant llibreria s3fs

3.1.3.1 Escriptura de dades a Amazon S3

```
@profile
def put_df_s3fs(data, s3_filepath, region_name, endpoint_url,
aws_access_key_id, aws_secret_access_key):
    # Establiment de la connexió a Amazon S3 amb un compte d'usuari
determinat
    fs = s3fs.S3FileSystem(
        anon = False,
        use_ssl = True,
        client_kwargs={
            "region_name": region_name,
            "endpoint_url": endpoint_url,
            "aws_access_key_id": aws_access_key_id,
            "aws_secret_access_key": aws_secret_access_key,
            "verify": True,
        }
    )

    # Escriptura del Pandas Dataframe a Amazon S3. Primer passem el
dataset a PyArrow Table i finalment escrivim la informació al backend amb
un fitxer PyArrow Parquet
    with fs.open(s3_filepath, 'wb') as f:
        pq.write_table(Table.from_pandas(data), f, version='2.4',
use_dictionary=True, compression='snappy')
```

L'escriptura a Amazon S3 utilitzant la llibreria s3fs es comença inicialitzant una instància de *s3fs.s3FileSystem*, eina que permet interactuar amb l'emmagatzemament d'Amazon S3. Això es fa especificant que volem treballar amb un mètode d'accés mitjançant autenticació AWS (*anon = False*), utilitzant encriptació SSL en la comunicació (*use_ssl = True*) i especificant els paràmetres d'autenticació del compte AWS (*client_kwargs*). Després s'escriu el Pandas DataFrame en el *bucket* i *path* determinat del sistema d'emmagatzemament. Això es fa passant el Pandas Dataframe a `pyarrow.Table`,

estructura usada per passar les dades a un arxiu Parquet i enviar-les a Amazon S3 mitjançant el mètode *write_table* de `pyarrow.parquet`. Aquesta escriptura es du a terme comprimint el fitxer Parquet amb *snappy* [37], un algoritme de compressió àmpliament utilitzat per l'emmagatzemament i transmissió de dades.

3.1.3.2 Lectura de dades des d'Amazon S3

```
@profile
def get_df_s3fs(s3_filepath, region_name, endpoint_url,
aws_access_key_id, aws_secret_access_key, usingPyArrow=False):
    # Establiment de la connexió a Amazon S3 amb un compte d'usuari
determinat
    fs = s3fs.S3FileSystem(
        anon = False,
        use_ssl = True,
        client_kwargs={
            "region_name": region_name,
            "endpoint_url": endpoint_url,
            "aws_access_key_id": aws_access_key_id,
            "aws_secret_access_key": aws_secret_access_key,
            "verify": True,
        }
    )

    # Lectura del fitxer Parquet des de S3 i conversió a PyArrow Table
    with fs.open(s3_filepath, 'rb') as f:
        parquet_table = pq.read_table(f)

    # Conversió de PyArrow Table a Pandas DataFrame
    if(usingPyArrow):
        df = parquet_table.to_pandas(types_mapper = pd.ArrowDtype)
    else:
        df = parquet_table.to_pandas()

    return df
```

La lectura de la informació des d'Amazon S3 comença igual que l'escriptura, inicialitzant una instància de *s3fs.s3FileSystem* que ens permetrà tenir una connexió al sistema d'emmagatzemament. A continuació, a través del mètode de `pyarrow.parquet` anomenat *read_table*, es fa la lectura del fitxer Parquet que està guardat en un *bucket* i *path* específics i es transforma a una estructura `pyarrow.Table`, la qual permet fer transformacions a estructures DataFrame. Finalment, es converteix la taula en format columnar en el Pandas DataFrame original tenint en compte el tipus de dades inicial de l'estructura.

3.1.4 Amazon S3 utilitzant llibreria Boto3

3.1.4.1 Escriptura de dades a Amazon S3

```
@profile
def put_df(data, path, region_name, aws_access_key_id,
aws_secret_access_key, bucket):
    # Creció del client boto3 que utilitzarà Amazon S3 amb un compte
d'usuari determinat
    s3_client = boto3.client('s3',
                             region_name = region_name,
                             aws_access_key_id = aws_access_key_id,
                             aws_secret_access_key =
aws_secret_access_key)


    # Creació d'un buffer a memòria utilitzat per guardar temporalment el
fitxer Parquet
    parquet_buffer_w = io.BytesIO()
    data.to_parquet(parquet_buffer_w, index = False)


    # Càrrega del fitxer Parquet en el Bucket i path determinats
    s3_client.put_object(Bucket=bucket, Key=path,
Body=parquet_buffer_w.getvalue())
```

L'escriptura de dades comença creant un client boto3, el qual està enllaçat al servei d'Amazon S3 mitjançant un compte d'usuari que tingui accés al *bucket* amb el que es vol treballar. Després, es crea un *buffer* intermediari a memòria per poder guardar temporalment el fitxer Parquet (*parquet_buffer_w = io.BytesIO()*), el qual finalment serà utilitzat per transmetre els bytes d'informació a Amazon S3 amb el mètode *put_object* del client boto3.

3.1.4.2 Lectura de dades des d'Amazon S3

```
@profile
def get_df(path, region_name, aws_access_key_id, aws_secret_access_key,
bucket, usingArrow=False):
    # Creció del client boto3 que utilitzarà Amazon S3 amb un compte
d'usuari determinat
    s3_client = boto3.client('s3',
                             region_name = region_name,
                             aws_access_key_id = aws_access_key_id,
                             aws_secret_access_key =
aws_secret_access_key)


    # Lectura del fitxer Parquet de Amazon S3
    response = s3_client.get_object(Bucket=bucket, Key=path)
```

```
    # Lectura del contingut del fitxer i emmagatzemament a un buffer
intermediari.
    parquet_buffer_r = io.BytesIO(response['Body'].read())


    # Lectura del fitxer Parquet des del buffer intermediari i conversió
a PyArrow Table
    parquet_table = pq.read_table(parquet_buffer_r)


    # Conversió de PyArrow Table a Pandas DataFrame
    if(usingArrow):
        df = parquet_table.to_pandas(types_mapper = pd.ArrowDtype)
    else:
        df = parquet_table.to_pandas()


    return df
```

La lectura de les dades des d'Amazon S3 comença igual que l'escriptura, creant un client boto3. A continuació s'aconsegueix l'objecte del *bucket* i *path* determinats per després poder ser llegit mitjançant el *io.BytesIO(response['Body'].read()*, un *buffer* a memòria que emmagatzema els bytes de l'objecte descarregat. Posteriorment, es converteixen els bytes de la resposta en una estructura `pyarrow.Table`, la qual serà feta servir per fer la conversió al Pandas DataFrame original amb els seus tipus corresponents.

3.2 Integració amb un model MapReduce

El model MapReduce està pensat per poder dur a terme un processament paral·lel de grans conjunts de dades en dues fases consecutives, la fase de map i la fase de reduce. Totes dues fases compten amb un nombre determinat de funcions que s'executen de forma paral·lela amb l'objectiu de processar informació. Entre les dues fases es produeix un gran intercanvi de dades utilitzant un sistema d'emmagatzemament intermediari. Aquesta fase de comunicació pot arribar a ser un coll d'ampolla quan es treballa amb un paral·lelisme elevat, és per això que el sistema intermediari ha de comptar amb una baixa latència per poder optimitzar el rendiment. Les característiques d'aquest model fan que sigui ideal per valorar l'eficiència d'un mecanisme de compartició amb còpia zero.

Per poder estudiar el rendiment dels diferents mecanismes de compartició amb aquest model, s'ha fet servir com a base una simplificació del codi del *framework* Seer, on s'implementa un MapReduce amb Python. Aquesta implementació utilitza el paral·lelisme de la llibreria multiprocessing perquè múltiples processos processin particions d'un Pandas DataFrame [38]. Un cop els *mappers* han acabat les seves execucions, la informació es comparteix als diferents *reducers* mitjançant disc (s'usa l'API Storage de Lithops amb el mode localhost [39]. Aquesta permet emmagatzemar i recuperar informació serialitzada mitjançant el disc de la màquina). Com l'únic *backend* d'emmagatzematge intermediari era disc, s'ha dut a terme una adaptació del codi per poder treballar amb diferents sistemes d'emmagatzematge. Els sistemes intermediaris incorporats han estat pensats per poder fer una correcta avaluació del rendiment de la compartició amb còpia zero versus mecanismes amb una major latència. En concret, els sistemes d'emmagatzematge que seran intermediaris en la compartició d'informació en aquest codi que segueix el model MapReduce són els següents:

- **Memòria compartida:** L'escriptura i lectura de les dades es farà utilitzant PyArrow.
- **Disc:** Per escriure i llegir la informació de disc s'utilitzarà l'API Storage de Lithops amb el mode localhost.
- **Amazon S3:** Per escriure i llegir de memòria remota s'usaran les llibreries s3fs i boto3. D'aquesta manera també es podrà avaluar el rendiment de les dues llibreries davant una major interacció amb les interfícies.

La incorporació dels nous mecanismes de compartició de dades en el codi ha estat una tasca costosa i ha necessitat uns temps de comprensió i anàlisi del codi base. En concret, aquests són els punts a destacar de l'adaptació:

- S'ha dut a terme una comprensió del model MapReduce implementat que comptava amb 1559 línies de codi.
- S'ha realitzat una adaptació i modificació de l'estructura perquè permetés utilitzar diferents sistemes d'emmagatzematge com intermediaris. Aquestes modificacions han estat donades per 1110 línies de codi.
- Les escriptures i lectures als diferents *backends* es fan a partir del codi de l'apartat 3.1.

4 Estudi sobre la compartició de dades utilitzant diferents backends

4.1 Objectius

Els *frameworks* d'anàlisi de dades a gran escala que utilitzen FaaS per executar funcions de forma paral·lela, necessiten un sistema d'emmagatzemament intermediari per poder comunicar les funcions entre elles. La comunicació directa entre FaaS a través de la xarxa és inviable o molt difícil, és per això que *frameworks* com Lithops utilitzen sistemes d'emmagatzematge remot com intermediaris per la comunicació de dades entre funcions (un exemple on ens podem trobar el comentat és amb una execució del mètode map_reduce de l'API de Lithops [40]. Aquest mètode pot usar Amazon Lambda per executar les funcions i Amazon S3 per compartir informació entre elles). Aquests tipus d'emmagatzematges són lents i provoquen còpies de dades innecessàries. Una forma d'aconseguir uns millors temps de compartició de dades és executant les funcions dins una mateixa màquina virtual, com una instància d'EC2, i que la comunicació entre les funcions sigui a través de memòria compartida amb còpia zero.

L'objectiu principal de l'estudi és valorar l'eficiència d'utilitzar mecanismes de compartició de dades sense còpies davant mecanismes de compartició de dades amb còpies utilitzant diferents sistemes d'emmagatzemament. Per poder aconseguir-ho es portaran a terme les següents anàlisis:

1. **Anàlisi de les escriptures i lectures de Pandas DataFrame en diferents sistemes d'emmagatzemament i utilitzant diferents codificacions.** L'anàlisi donarà importància a diferents paràmetres crítics com la mida de les estructures de dades, control de l'ús de memòria i temps d'execució. En aquest cas, s'utilitza el codi implementat en l'apartat 3.1.
2. **Anàlisi de la compartició de particions amb un model MapReduce utilitzant diferents sistemes d'emmagatzemament intermediaris.** L'anàlisi donarà importància a la velocitat amb la que es possible fer el processament de les dades. En aquest cas, es fa ús de l'adaptació del codi del *framework* Seer que s'ha fet referència en l'apartat 3.2, el qual segueix l'estructura del model MapReduce.

4.2 Anàlisi de les escriptures i lectures de Pandas DataFrame en diferents sistemes d'emmagatzematge

4.2.1 Metodologia

Per aquesta anàlisi es realitzaran escriptures i lectures de DataFrames mitjançant dos sistemes d'emmagatzematge: memòria compartida i Amazon S3. Per interaccionar amb la memòria compartida es farà ús de `multiprocessing.shared_memory`. Per escriure i llegir informació d'Amazon S3 s'usaran les llibreries boto3 i s3fs, buscant així el mètode més eficient i competitiu. El codi utilitzat per poder fer les escriptures i lectures es troba en l'apartat 3.1.

Quant a la serialització i deserialització de les dades, es faran servir diferents mètodes per poder valorar les diferències entre ells. Per escriure la informació a memòria compartida es faran servir les llibreries Pickle i PyArrow. D'aquesta forma es podrà comprovar utilitzant el mateix sistema d'emmagatzematge els avantatges que hi ha en una compartició de dades sense còpies (PyArrow) sobre una compartició de dades amb còpies (Pickle). Per escriure la informació a Amazon S3 es farà ús d'un fitxer PyArrow Parquet.

Aquestes són les diferents opcions que hi han per escriure i llegir els Pandas DataFrame:

- **multiprocessing.shared_memory utilitzant serialització PyArrow.**
- **multiprocessing.shared_memory utilitzant serialització Pickle.**
- **Amazon S3 utilitzant la llibreria s3fs per interaccionar amb el backend.**
- **Amazon S3 utilitzant la llibreria boto3 per interaccionar amb el backend.**

L'estudi treballarà amb dos tipus de CSV que seran llegits a l'inici del codi i transformats en un format DataFrame. Els dos tipus de CSV són aquests:

- **Arxiu Terasort:** Arxiu CSV que està format per dues columnes de strings. Aquests strings seran tractats fent servir diferents tipus, el string de Python, l'object de Python i el string de PyArrow. Es tractaran diferents mides del fitxer per poder fer una anàlisi depenent de la mida de l'entrada. En concret, les mides del CSV seran de 100, 200, 300, 400 i 500 MB.
- **Arxiu Brain:** Arxiu CSV que està format per tres columnes, una d'integers i dos de float. Els valors de les columnes seran tractats usant diferents tipus, els int32 i float64 de Python i els int32 i float64 de PyArrow. Les diferents mides de l'arxiu seran de 32, 63, 95, 127 i 150MB.

;GH:JDM>;>	00000000000000000000000000000000AAAAAAAA222200002222000022220000222200002222000000001111
WrtPDbw}":	00000000000000000000000000000001AAAAAAAA9999444488885555CCCC777755555555BBBB666644446666
:XV?pw?/4B	00000000000000000000000000000002AAAAAAAAFFFF777799995555FFFF11112222999988884444DDDDFFFF
R0!h(40DLC	00000000000000000000000000000003AAAAAAAA1111000000000000066668888BBBB33333333AAAA1111CCCC

Taula 1. Exemple de part del CSV Terasort

58	114.08143449328479	1469313.6
58	132.98055504406673	2779472.8
58	132.98153464009434	5186672.5
58	151.67038870791575	1846254.1

Taula 2. Exemple de part del CSV Brain

En aquest cas, l'anàlisi valorarà diferents paràmetres que tenen una alta incidència en l'eficiència de processar grans quantitats de dades. En concret, són els següents:

- **Mida de les estructures de dades:** A l'hora de guardar un DataFrame a un sistema d'emmagatzemament, l'estructura de dades inicial va modificant-se i adaptant-se a un format òptim per poder ser transmès en memòria. El mateix passa quan volem llegir la informació, hi haurà unes certes conversions fins que recuperarem la forma inicial de l'estructura de dades. L'objectiu d'analitzar aquest paràmetre és observar com varia la mida de les dades

depenent del format en el qual es troba el DataFrame i en quin lloc està emmagatzemat.

- **Control de la utilització de la memòria:** Per tenir un control de les variacions de la memòria utilitzada al llarg de l'execució del codi s'utilitzarà una eina de Python anomenada memory_profiler [41]. El paquet memory_profiler permet identificar, entre altres, les zones on es produeixen còpies de dades, ja que van lligades a pics de memòria. D'aquesta forma es podrà saber quina implementació permet una compartició de dades amb còpia zero.
- **Temps d'execució:** Els sistemes d'emmagatzemament intermediaris entre funcions han de ser tan ràpids com sigui possible per agilitzar la comunicació, és per això que és primordial controlar els temps d'escriptura i lectura de les dades.

4.2.2 Valoració dels resultats

4.2.2.1 Mida de les estructures de dades

La mida de les estructures de dades és un factor clau per poder entendre els diferents mètodes de serialització i tenir un control sobre la utilització de memòria. A més, depenent d'aquest paràmetre, es poden entendre molts dels temps d'execució obtinguts en les escriptures i lectures de les dades. És per això que s'ha fet un seguiment de les mides dels diferents tipus d'estructures de dades en les quals es transforma el fitxer CSV inicial i també un control de la mida reservada per guardar la informació en el sistema d'emmagatzemament. En concret, per poder veure les diferències de forma més clara, s'ha fet l'estudi sobre el CSV Terasort de 500MB i el CSV Brain de 150MB.

Quan s'utilitza memòria compartida per emmagatzemar la informació, en el cas d'utilitzar la serialització PyArrow, s'ha estudiat la mida de l'arxiu CSV, del Pandas DataFrame, del `pyarrow.RecordBatch` aconseguit de la serialització del DataFrame i de la memòria compartida reservada. En el cas d'utilitzar la serialització Pickle, s'ha estudiat la mida del arxiu CSV, del Pandas DataFrame i de la memòria compartida reservada pel Pandas DataFrame serialitzat.

Quan s'utilitza Amazon S3 com a sistema intermediari, s'ha estudiat la mida de l'arxiu CSV, la mida del Pandas DataFrame aconseguit a partir del fitxer CSV i la memòria ocupada un cop el fitxer parquet es troba en el *bucket*.

Com es pot veure a les taules 3, 5 i 7, en el cas de treballar amb cadenes de caràcters com es fa amb l'arxiu Terasort, la mida dels Pandas DataFrame depèn molt del tipus utilitzat. Això és causat per la forma en que són representades les dades depenent del tipus. En canvi, en el cas de treballar amb tipus integer i float, com passa amb l'arxiu Brain (taules 4, 6 i 8), tots els Pandas DataFrame derivats de l'arxiu CSV són de la mateixa mida.

Els tipus object i string de Python compten amb una considerable sobrecàrrega de memòria donada per una capçalera que conté metadades sobre els objectes. A més, també s'ha de tenir en compte que Pandas representa aquests tipus amb cadenes NumPy de punters a objectes tipus string, els quals cada un d'ells afegeixen una sobrecàrrega de memòria de vuit bits. Tot això provoca que inclús tenint un objecte buit d'aquests tipus, acabi ocupant un cert nombre considerable de bits. Per exemple, si tenim una cadena buida, s'ha de tenir un compte la sobrecàrrega de vuit bits donada pel punter i la sobrecàrrega donada per la capçalera amb metadades que ocupa quaranta-nou bits. Si tenim en compte que s'ha de treballar amb Pandas DataFrame amb una gran quantitat d'aquests valors, és pot arribar a tenir una alta ocupació de memòria.

En canvi, quan Pandas DataFrame utilitza el tipus string de PyArrow, la sobrecàrrega ja no és tant notable i s'aconsegueix unes estructures més comprimides. El tipus string de PyArrow sol compta amb una sobrecàrrega d'uns 33 bits en total. Aquest avantatge pot ser molt valuós treballant amb grans estructures de dades, sobretot si les cadenes utilitzades són petites, ja que com més grans siguin, més insignificant serà la sobrecàrrega de les metadades davant la quantitat de bits proporcionats per la cadena en sí. En el cas de l'estudi, les cadenes de caràcters utilitzades són bastant grans i igualment es pot observar una disminució considerable de l'espai de memòria utilitzat.

En el cas de treballar amb els tipus integer i float, no hi ha un avantatge utilitzant els tipus PyArrow. Això ve atès que aquests tipus sempre utilitzen una mida fixa per representar-se. En el cas d'int32 seran trenta-dos bits i en el cas de float64 seran seixanta-quatre bits. És per això que totes les mides dels Pandas DataFrame derivats de l'arxiu CSV Brain, independentment del tipus utilitzat, són iguals.

Si mirem les taules 3, 4, 5 i 6, i ens centrem ara en la mida de la informació serialitzada (Size of the PyArrow RecordBatch amb codificació PyArrow i Size of the Shared Memory amb codificació Pickle), podem veure uns comportaments bastant diversos. En el cas d'utilitzar la serialització Pickle i l'escriptura a memòria compartida dels Pandas DataFrame amb tipus Python (taula 6), podem veure com la memòria necessària per emmagatzemar la informació és menor que quan hem de guardar les dades codificades amb PyArrow (taula 4). Cosa que indica que **Pickle comprimeix millor les estructures que treballen amb aquests tipus Python del que ho fa PyArrow. En canvi, si es treballa amb els tipus PyArrow, la codificació al format columnar és més eficient a nivell de memòria (taula 3) de la que s'obté utilitzant la codificació Pickle (taula 5).** Això ve donat que la codificació al format columnar pot optimitzar els seus mètodes de compressió al màxim sempre que s'utilitzin els tipus natius de la llibreria PyArrow.

Finalment, **en el cas de l'escriptura a Amazon S3 (imatge 7 i 8), podem observar com la mida de la memòria necessària per emmagatzemar les estructures de dades és molt menor que quan s'utilitza memòria compartida.** Això ve donat, a part de per l'eficient compressió de les dades proporcionada per l'ús d'arxius PyArrow Parquet, per la compressió snappy. Aquest és un algoritme de compressió dissenyat per una compressió i descompressió de les dades de forma ràpida i eficient en termes de memòria.

terasort	**0 string 1 string dtype: object**	**0 object 1 object dtype: object**	**0 string[pyarrow] 1 string[pyarrow] dtype: object**
terasort500m	Bytes	Bytes	Bytes
Size of the csv	500000000	500000000	500000000
Size of the dataframe	1010869371	1010869371	527265995
Size of the Pyarrow RecordBatch	527265867	527265867	527265867
Size of the Shared Memory	527266864	527266864	527266864

Taula 3. Resultats de les mides de les estructures de dades utilitzant el fitxer Terasort de 500MB i la serialització PyArrow.

Brain	**0 int32 1 float64 2 float64**	**0 int32[pyarrow] 1 double[pyarrow] 2 double[pyarrow]**
Brain1.csv	Bytes	Bytes
Size of the csv	157551403	157551403
Size of the dataframe	94570048	94570048
Size of the Pyarrow RecordBatch	94569920	94569920
Size of the Shared Memory	94571080	94571104

Taula 4. Resultats de les mides de les estructures de dades utilitzant el fitxer Brain de 150MB i la serialització PyArrow.

terasort	**0 string 1 string dtype: object**	**0 object 1 object dtype: object**	**0 string[pyarrow] 1 string[pyarrow] dtype: object**
terasort500m	Bytes	Bytes	Bytes
Size of the csv	500000000	500000000	500000000
Size of the dataframe	1010869371	1010869371	527265995
Size of the Shared Memory	518346242	518345996	527270396

Taula 5. Resultats de les mides de les estructures de dades utilitzant el fitxer Terasort de 500MB i la serialització Pickle.

Brain	**0 int32 1 float64 2 float64**	**0 int32[pyarrow] 1 double[pyarrow] 2 double[pyarrow]**
Brain1.csv	Bytes	Bytes
Size of the csv	157551403	157551403
Size of the dataframe	94570048	94570048
Size of the Shared Memory	94570701	94575068

Taula 6. Resultats de les mides de les estructures de dades utilitzant el fitxer Brain de 150MB i la serialització Pickle.

terasort	**0 string 1 string dtype: object**	**0 object 1 object dtype: object**	**0 string[pyarrow] 1 string[pyarrow] dtype: object**
terasort500m	Bytes	Bytes	Bytes
Size of the csv	500000000	500000000	500000000
Size of the dataframe	1010869371	1010869371	527265995
Size in S3	197500000	197500000	197500000

Taula 7. Resultats de les mides de les estructures de dades utilitzant el fitxer Terasort de 500MB i l'escriptura a S3 utilitzant s3fs o boto3.

Brain	**0 int32** **1 float64** **2 float64**	**0 int32[pyarrow]** **1 double[pyarrow]** **2 double[pyarrow]**
Brain1.csv	Bytes	Bytes
Size of the csv	157551403	157551403
Size of the dataframe	94570048	94570048
Size in S3	36500000	36500000

Taula 8. Resultats de les mides de les estructures de dades utilitzant el fitxer Brain de 150MB i l'escriptura a S3 utilitzant s3fs o boto3.

Els punts clau de l'anàlisi de les mides de les estructures de dades són els següents:

- Els tipus object i string de Python compten amb un major *overhead* de memòria que els tipus string de PyArrow. Aquest fet és causat per la diferència en el nombre de metadades utilitzades en els tipus. Això provoca que els arxius Terasort interpretats amb tipus string de PyArrow ocupin menys en memòria.
- Els tipus integer i float, com es representen amb una mida fixa, no hi ha diferència en la utilització de la memòria entre els tipus.
- Pickle comprimeix millor que PyArrow les estructures que treballen amb tipus Python. En canvi, PyArrow comprimeix millor que Pickle les estructures que treballen amb tipus PyArrow.
- Amazon S3 és el que necessita menys memòria per emmagatzemar les diferents estructures. Això és resultat de l'ús de fitxers PyArrow Parquet i el mètode de compressió snappy.

4.2.2.2 Consum de memòria

4.2.2.2.1 Escriptura del Pandas DataFrame a memòria

A l'hora d'escriure les dades a memòria ens trobem amb uns resultats diferents depenent de l'arxiu emprat, tipus utilitzat, serialització implementada o sistema d'emmagatzemament destí. En les taules 9-16, en les que es mostra els punts del codi que provoquen un increment considerable de la memòria (en Mebibytes (MiB). 1 MiB és 1048576 bytes), es pot observar com **per fer l'escriptura es produeix una increment del consum de memòria quan treballem amb Amazon S3 com a destí i dos increments quan es fa servir memòria compartida com intermediària.**

En el cas de treballar amb memòria compartida com a destí, els increments de memòria utilitzada apareixen en el moment de la serialització i escriptura de les dades (taules 9, 10, 11, 12). Aquest comportament és igual independentment de si codifiquem amb Pickle (taula 10 i 12) o amb PyArrow (taula 9 i 11). En la primera còpia, el consum de memòria és causat per la transformació de les dades a un altre format. En la segona còpia, l'increment de memòria usada ve definit pel moviment de dades a una zona de memòria compartida. **En totes dues còpies, les dues serialitzacions emprades donen un increment de la memòria similar**, donant lloc a unes dades codificades de mida similar. Finalment, el consum de memòria s'estabilitza tant en la serialització amb PyArrow com amb Pickle. En les taules 9 i 11 podem observar com utilitzant PyArrow, al final decreix l'ús de memòria gràcies a l'eliminació de les referències a aquesta (*del record_batch*) i l'alliberament de recursos lligats a la memòria compartida (*sm_put.close()*). En el cas de la utilització de Pickle (taula 10 i 12) també decreix l'ús de memòria gràcies a l'alliberament de recursos de memòria compartida (*sm_put.close()*).

Terasort (PyArrow Serialization)	Python String 0 string 1 string dtype: object		Python Object 0 object 1 object dtype: object		PyArrow String 0 string 1 string dtype: object	
terasort500m	**Mem Usage (MiB)**	**Increment (MiB)**	**Mem Usage (MiB)**	**Increment (MiB)**	**Mem Usage (MiB)**	**Increment (MiB)**
@profile	1193,8	1193,8	1536	1536	1193,5	1193,5

if(usingArrow):
table = (pyarrow,Table,from_pandas(data)),combine_chunks()
record_batch = table,to_batches()[0]
else:
record_batch = pa,RecordBatch,from_pandas(data)
	1816,5	622,7	2576,1	1040,1	1696,1	502,6

stream_writer,write_batch(record_batch)	2269,6	451,8	2806,8	230,7	2198,8	502,7
del record_batch	1869,5	-400,1	2704,3	-102,5	1696,1	-502,7
sm_put,close()	1366,7	-502,8	2201,6	-502,7	1193,5	-502,6

Taula 9. Punts del codi on es produeix un increment del consum de memòria escrivint a memòria compartida. En aquest cas s'usa la serialització PyArrow i es treballa amb el fitxer Terasort de 500MB.

Terasort (Pickle Serialization)	Python String 0 string 1 string dtype: object		Python Object 0 object 1 object dtype: object		PyArrow String 0 string 1 string dtype: object	
terasort500m	**Mem Usage (MiB)**	**Increment (MiB)**	**Mem Usage (MiB)**	**Increment (MiB)**	**Mem Usage (MiB)**	**Increment (MiB)**
@profile	1193	1193	1461,2	1461,2	1025	1025
pickled_df = pickle.dumps(df)	1687,4	494,4	1976,5	515,3	1527,7	502,7
sm_put.buf[:data_size] = pickled_df	2182,1	494,3	2470,8	494,2	2030,4	502,7
sm_put.close()	1687,8	-494,3	1976,5	-494,2	1527,7	-502,7

Taula 10. Punts del codi on es produeix un increment del consum de memòria escrivint a memòria compartida. En aquest cas s'usa la serialització Pickle i es treballa amb el fitxer Terasort de 500MB.

Brain (PyArrow Serialization)	0 int32 1 float64 2 float64		0 int32[pyarrow] 1 double[pyarrow] 2 double[pyarrow]	
Brain1	**Mem Usage (MiB)**	**Increment (MiB)**	**Mem Usage (MiB)**	**Increment (MiB)**
@profile	178,7	178,7	278,6	278,6

if(usingArrow):
table = (pyarrow,Table,from_pandas(data)),combine_chunks()
record_batch = table,to_batches()[0]
else:
record_batch = pa,RecordBatch,from_pandas(data)
	181,3	2,6	368,9	90,2

stream_writer,write_batch(record_batch)	272,9	90,2	453,9	85
del record_batch	272,9	0	363,8	-90,1
sm_put,close()	182,7	-90,2	278,9	-84,9

Taula 11. Punts del codi on es produeix un increment del consum de memòria escrivint a memòria compartida. En aquest cas s'usa la serialització PyArrow i es treballa amb el fitxer Brain de 150MB.

Brain (Pickle Serialization)	0 int32 1 float64 2 float64		0 int32[pyarrow] 1 double[pyarrow] 2 double[pyarrow]	
Brain1	**Mem Usage (MiB)**	**Increment (MiB)**	**Mem Usage (MiB)**	**Increment (MiB)**
@profile	177,9	177,9	365,6	365,6
pickled_df = pickle.dumps(df)	286,2	108,3	401,2	35,6
sm_put.buf[:data_size] = pickled_df	376,6	90,1	491,3	90,1
sm_put.close()	286,5	-90,1	401,2	-90,1

Taula 12. Punts del codi on es produeix un increment del consum de memòria escrivint a memòria compartida. En aquest cas s'usa la serialització Pickle i es treballa amb el fitxer Brain de 150MB.

En el cas de treballar amb Amazon S3 com a sistema d'emmagatzemament, ja sigui utilitzant la llibreria Boto3 (taules 14 i 16) o s3fs (taules 13 i 15), sol es produeix un increment del consum de memòria en el moment de serialitzar el Pandas DataFrame a un fitxer Parquet. En el moment de l'escriptura de les dades es produeix un decrement del consum de memòria donat per l'alliberament del fitxer que ara passa a estar emmagatzemat al *bucket* d'Amazon S3, en el cas de boto3 (taules 14 i 16) es pot observar clarament el comportament. Quan es fa servir s3fs es pot veure com l'increment i decrement del consum de memòria es produeix en la mateixa línia de codi, donant resultat a una estabilització del creixement.

Terasort (s3fs)	Python String 0 string 1 string dtype: object		Python Object 0 object 1 object dtype: object		PyArrow String 0 string 1 string dtype: object	
terasort500m	Mem Usage (MiB)	Increment (MiB)	Mem Usage (MiB)	Increment (MiB)	Mem Usage (MiB)	Increment (MiB)
@profile	1210,6	1210,6	1210,6	1210,6	1162,1	1162,1
pq.write_table(Table.from_pandas(data), f, version='2.4', use_dictionary=True, compression='snappy')	1226,5	15,8	1226,5	15,8	1163,4	1,2

Taula 13. Punts del codi on es produeix un increment del consum de memòria escrivint a Amazon S3. En aquest cas s'usa la llibreria s3fs i es treballa amb el fitxer Terasort de 500MB.

Terasort (boto3)	Python String 0 string 1 string dtype: object		Python Object 0 object 1 object dtype: object		PyArrow String 0 string 1 string dtype: object	
terasort500m	Mem Usage (MiB)	Increment (MiB)	Mem Usage (MiB)	Increment (MiB)	Mem Usage (MiB)	Increment (MiB)
@profile	1199,9	1199,9	1201,7	1201,7	668	668
df.to_parquet(parquet_buffer_w, index = False)	1813,7	604,1	1813,2	601,8	872,3	204,2
s3_client.put_object(Bucket='proof-bucket1', Key=random_name, Body=parquet_buffer_w.getvalue())	1411,4	-402,4	1413,2	-400	868,5	-3,9

Taula 14. Punts del codi on es produeix un increment del consum de memòria escrivint a Amazon S3. En aquest cas s'usa la llibreria boto3 i es treballa amb el fitxer Terasort de 500MB.

Brain (s3fs)	0 int32 1 float64 2 float64		0 int32[pyarrow] 1 double[pyarrow] 2 double[pyarrow]	
Brain1	Mem Usage (MiB)	Increment (MiB)	Mem Usage (MiB)	Increment (MiB)
@profile	1486,4	1486,4	531,8	531,8
pq.write_table(Table.from_pandas(data), f, version='2.4', use_dictionary=True, compression='snappy')	1453,1	-33,4	534,7	2,9

Taula 15. Punts del codi on es produeix un increment del consum de memòria escrivint a Amazon S3. En aquest cas s'usa la llibreria s3fs i es treballa amb el fitxer Brain de 150 MB.

Brain (boto3)	0 int32 1 float64 2 float64		0 int32[pyarrow] 1 double[pyarrow] 2 double[pyarrow]	
Brain1	Mem Usage (MiB)	Increment (MiB)	Mem Usage (MiB)	Increment (MiB)
@profile	194,2	194,2	217,8	217,8
df.to_parquet(parquet_buffer_w, index = False)	276,1	66,7	278,6	58,1
s3_client.put_object(Bucket='proof-bucket1', Key=random_name, Body=	264,4	-11,7	262,1	-16,5

Taula 16. Punts del codi on es produeix un increment del consum de memòria escrivint a Amazon S3. En aquest cas s'usa la llibreria boto3 i es treballa amb el fitxer Brain de 150 MB.

Els punts clau de l'anàlisi del consum de memòria en el moment de l'escriptura de les dades són els següents:

- Hi ha un increment del consum de memòria per fer l'escriptura de dades a Amazon S3 i dos increments per escriure les dades a memòria compartida (passa el mateix utilitzant Pickle i PyArrow).
- En el cas d'utilitzar la serialització Pickle i PyArrow es produeix una còpia en el moment de serialitzar les dades i una altra en el moment d'escriure-les. Els increments de consum de memòria són similars.
- En el cas d'usar Amazon S3 com a destí es produeix un increment en el moment de la serialització de les dades. En el moment de l'escriptura hi ha una baixada del consum donat l'alliberament del fitxer emmagatzemat.

4.2.2.2.2 Lectura del Pandas DataFrame des de memòria

En el cas de treballar amb memòria compartida com a sistema intermediari, pot haver-hi una còpia de dades o cap, depenent de la serialització emprada i dels tipus utilitzats en el Pandas DataFrame. **Quan s'utilitza la codificació Pickle**, es pot observar en tots els casos com es produeix un **increment de l'ús de la memòria quan es deserialitzen**

les dades (taules 18 i 20). Aquest increment és causat per la còpia de l'estructura codificada des de la zona de memòria compartida fins a la memòria local del procés, per poder acabar fent la descodificació de les dades**. En el cas d'utilitzar la serialització PyArrow (taules 17 i 19), sol treballant amb un Pandas DataFrame amb tipus PyArrow s'obtindrà la còpia zero. Quan es treballa amb els tipus Python, la còpia de dades serà en el moment de passar l'estructura pyarrow.RecordBatch a un Pandas DataFrame**. Aquest fet apareix perquè no hi ha compatibilitat entre els tipus de dades utilitzats per l'estructura RecordBatch de PyArrow i el Pandas DataFrame destí, que en aquest cas vol treballar amb tipus Python. Si hi ha compatibilitat entre els tipus, com les dues instàncies treballen amb lots de dades contigües en memòria, totes dues estructures compartiran una vista del *buffer* de memòria amb les dades, evitant així la sobrecàrrega de CPU i el fet de tenir dades repetides emmagatzemades. La còpia zero també s'aconseguiria fent un *slicing* de l'estructura, és a dir, aconseguint un subconjunt de les dades totals a partir d'índexs.

Terasort (PyArrow Serialization)	Python String 0 string 1 string dtype: object		Python Object 0 object 1 object dtype: object		PyArrow String 0 string 1 string dtype: object	
terasort500m	Mem Usage (MiB)	Increment (MiB)	Mem Usage (MiB)	Increment (MiB)	Mem Usage (MiB)	Increment (MiB)
@profile	1366,6	1366,6	2304,1	2304,1	1193,6	1193,6

if(usingArrow):
  data = record_batch,to_pandas(types_mapper = pd,ArrowDtype)
else:
  data = record_batch,to_pandas()
	2692,1	1333,3	4336,5	2032,4	1193,6	0

del data	1711,9	-980,2	3357,4	-979,1	1193,6	0
sm_get,close()	1209,1	-502,8	2378,3	-501,6	1193,6	0

Taula 17. Punts del codi on es produeix un increment del consum de memòria llegint de memòria compartida. En aquest cas s'usa la deserialització PyArrow i es treballa amb el fitxer Terasort de 500MB.

Terasort (Pickle Serialization)	Python String 0 string 1 string dtype: object		Python Object 0 object 1 object dtype: object		PyArrow String 0 string 1 string dtype: object	
terasort500m	Mem Usage (MiB)	Increment (MiB)	Mem Usage (MiB)	Increment (MiB)	Mem Usage (MiB)	Increment (MiB)
@profile	1193,4	1193,4	1482,2	1482,2	1024,8	1024,8
df = pickle.loads(data)	2672,5	1479,1	2961,9	1479,7	2370,8	1346
sm_get.close()	2178,1	-494,3	2467,5	-494,3	1868,1	-502,6

Taula 18. Punts del codi on es produeix un increment del consum de memòria llegint de memòria compartida. En aquest cas s'usa la deserialització Pickle i es treballa amb el fitxer Terasort de 500MB.

Brain (PyArrow Serialization)	0 int32 1 float64 2 float64		0 int32[pyarrow] 1 double[pyarrow] 2 double[pyarrow]	
Brain1	Mem Usage (MiB)	Increment (MiB)	Mem Usage (MiB)	Increment (MiB)
@profile	182,7	182,7	278,8	278,8

if(usingArrow):
  table = pyarrow,Table,from_batches([record_batch])
  data = table,to_pandas(types_mapper = pd,ArrowDtype)
else:
  data = record_batch,to_pandas()
	363,6	180,7	278,8	0

del data	363,6	0	278,8	0
sm_get.close()	273,7	-89,9	278,8	0

Taula 19. Punts del codi on es produeix un increment del consum de memòria llegint de memòria compartida. En aquest cas s'usa la deserialització PyArrow i es treballa amb el fitxer Brain de 150 MB.

Brain (Pickle Serialization)	0 int32 1 float64 2 float64		0 int32[pyarrow] 1 double[pyarrow] 2 double[pyarrow]	
Brain1	Mem Usage (MiB)	Increment (MiB)	Mem Usage (MiB)	Increment (MiB)
@profile	196,3	196,3	311	311
df = pickle.loads(data)	358,9	162,6	480,8	169,9
sm_get.close()	268,7	-90,2	390,9	-89,9

Taula 20. Punts del codi on es produeix un increment del consum de memòria llegint de memòria compartida. En aquest cas s'usa la deserialització Pickle i es treballa amb el fitxer Brain de 150 MB.

Quan es fa una lectura des d'Amazon S3, sempre es produeixen còpies de dades. **En el cas de treballar amb la interfície proporcionada per la llibreria s3fs (taules 21 i 23), hi ha un increment del consum de memòria en el moment de la lectura**, originat per l'aparició de noves dades emmagatzemades en el sistema (*parquet_table = pq.read_table(f)*). **També pot arribar a haver-hi un altre increment del consum en el cas que l'estructura `pyarrow.Table` no tingui compatibilitat de tipus amb el Pandas DataFrame destí**, com és el cas quan volem que l'estructura final compti amb tipus Python. **Quan s'utilitza la llibreria boto3 (taules 22 i 24) pot haver-hi dos o tres increments del consum de memòria. El primer** és l'increment menys considerable i **és el donat per la lectura de l'objecte** comprimit des del *bucket* d'Amazon S3 (*parquet_buffer_r = io.BytesIO(response['Body'].read()*). **El segon i més important apareix en el moment de descomprimir les dades** i de transformar el fitxer PyArrow Parquet a una estructura pyarrow.Table (*parquet_table = pq.read_table(parquet_buffer_r)*). Finalment, **el tercer apareixerà sempre que no existeixi compatibilitat entre les dades de les estructures de PyArrow i el Pandas DataFrame.**

Terasort (s3fs)	Python String 0 string 1 string dtype: object		Python Object 0 object 1 object dtype: object		PyArrow String 0 string 1 string dtype: object	
terasort500m	Mem Usage (MiB)	Increment (MiB)	Mem Usage (MiB)	Increment (MiB)	Mem Usage (MiB)	Increment (MiB)
@profile	1226,5	1226,5	1226,5	1226,5	1163,4	1163,4
parquet_table = pq.read_table(f)	1847,9	621,4	1847,9	621,4	1712,8	549,4

if(usingArrow):
df = parquet_table.to_pandas()
else:
df = parquet_table.to_pandas(types_mapper = pd.ArrowDtype)
	2754,5	906,6	2754,5	906,6	1712,8	0

Taula 21. Punts del codi on es produeix un increment del consum de memòria llegint d'Amazon S3. En aquest cas s'usa la llibreria s3fs i es treballa amb el fitxer Terasort de 500MB.

Terasort (boto3)	Python String 0 string 1 string dtype: object		Python Object 0 object 1 object dtype: object		PyArrow String 0 string 1 string dtype: object	
terasort500m	Mem Usage (MiB)	Increment (MiB)	Mem Usage (MiB)	Increment (MiB)	Mem Usage (MiB)	Increment (MiB)
@profile	1214,8	1214,8	1216,6	1216,6	671,8	671,8
parquet_buffer_r = io.BytesIO(response['Body'].read())	1412,6	197,5	1414,3	197,5	869,3	197,5
parquet_table = pq.read_table(parquet_buffer_r)	2061,6	649	2152,3	738	1393,8	524,5

if(usingArrow):
df = parquet_table.to_pandas()
else:
df = parquet_table.to_pandas(types_mapper = pd.ArrowDtype)
	2912,4	976,6	4260,5	2108,2	1380,7	0

Taula 22. Punts del codi on es produeix un increment del consum de memòria llegint d'Amazon S3. En aquest cas s'usa la llibreria boto3 i es treballa amb el fitxer Terasort de 500MB.

Brain (s3fs)	0 int32 1 float64 2 float64		0 int32[pyarrow] 1 double[pyarrow] 2 double[pyarrow]	
Brain1	**Mem Usage (MiB)**	**Increment (MiB)**	**Mem Usage (MiB)**	**Increment (MiB)**
@profile	1453,1	1453,1	534,7	534,7
parquet_table = pq.read_table(f)	1582,3	129,2	701,5	166,8

if(usingArrow):
df = parquet_table.to_pandas()
else:
df = parquet_table.to_pandas(types_mapper = pd.ArrowDtype)
	1672,6	90,3	701,5	0

Taula 23. Punts del codi on es produeix un increment del consum de memòria llegint d'Amazon S3. En aquest cas s'usa la llibreria s3fs i es treballa amb el fitxer Brain de 150 MB.

Brain (boto3)	0 int32 1 float64 2 float64		0 int32[pyarrow] 1 double[pyarrow] 2 double[pyarrow]	
Brain1	**Mem Usage (MiB)**	**Increment (MiB)**	**Mem Usage (MiB)**	**Increment (MiB)**
@profile	228,8	228,8	226,4	226,4
parquet_buffer_r = io.BytesIO(response['Body'].read())	265,5	36,4	262,8	36,4
parquet_table = pq.read_table(parquet_buffer_r)	400,4	134,9	388,2	125,5

if(usingArrow):
df = parquet_table.to_pandas()
else:
df = parquet_table.to_pandas(types_mapper = pd.ArrowDtype)
	493	92,6	388,2	0

Taula 24. Punts del codi on es produeix un increment del consum de memòria llegint d'Amazon S3. En aquest cas s'usa la llibreria boto3 i es treballa amb el fitxer Brain de 150 MB.

Davant l'anàlisi de les diferents taules **queda clar que l'opció de treballar amb serialització i tipus PyArrow és la més òptima**. Utilitzant aquest mètode s'eviten els temps de computació derivats de copiar dades des de la zona de memòria compartida fins a l'espai de memòria del procés i s'evita el consum de memòria resultant de tenir repetides les dades amb diferents formats.

Els punts clau de l'anàlisi del consum de memòria en el moment de la lectura de les dades són els següents:

- Quan s'utilitza memòria compartida i la serialització Pickle sempre hi ha una còpia de dades en el moment de la deserialització.
- Quan s'utilitza memòria compartida i la serialització PyArrow pot haver-hi una còpia o cap. El moment crític és quan passem l'estructura PyArrow.RecordBatch a un Pandas DataFrame. Si hi ha compatibilitat entre els tipus de les dues estructures, com és en el cas de treballar amb el tipus string de PyArrow, aconseguirem no tenir cap increment del consum de memòria.
- Si es treballa amb Amazon S3 i s3fs hi haurà un o dos increments del consum de memòria. Un en el moment de la lectura de les dades i un altre en el moment de passar la pyarrow.Table al Pandas DataFrame si no hi ha compatibilitat en els tipus.
- Si es treballa amb Amazon S3 i boto3 hi haurà dos o tres increments del consum de memòria. Els dos primers sempre existiran i són causats per la lectura de l'objecte de memòria i per l'obtenció del pyarrow.Table a partir del pyarrow.Parquet. L'últim increment serà causat sempre que no existeixi compatibilitat entre els tipus de l'estructura pyarrow.Table i el Pandas DataFrame.

4.2.2.3 Temps d'execució

4.2.2.3.1 Escriptura del Pandas DataFrame a memòria

4.2.2.3.1.1 Terasort

No es representen directament els resultats d'entrada/sortida a memòria compartida amb els de S3, ja que els primers són, com cal esperar, significativament més ràpids. Al seu lloc, a les figures 5 a 9 es mostren per separat cada *backend* de *storage* amb les diferents implementacions proposades.

Primer de tot, l'anàlisi es centrarà en l'escriptura de les dades en memòria compartida. En aquest cas, com es pot observar en la figura 5, podem veure la **menor latència amb els DataFrames que treballen amb tipus PyArrow**. El fet de treballar amb els tipus string de PyArrow, que utilitzen una representació de les dades amb menys *overhead* que els tipus Python, fan que els temps de serialització de les dades, sigui amb PyArrow o amb Pickle, donin uns resultats molt més eficients, donat que el DataFrame a codificar és més petit. **En la resta de casos, podem observar uns temps molt semblants** quan treballem amb els tipus string i tipus object, això és degut al fet que les estructures de dades a serialitzar són de la mateixa mida.

Si es comparen els mètodes de serialització, es pot veure clarament com PyArrow és dominant en totes les situacions. En tots dos casos es produeixen dues còpies de mides similars en l'escriptura, una quan es serialitza el DataFrame i l'altra quan s'escriu al sistema d'emmagatzemament. A més, els temps de transferència de la informació a la memòria també són molt similars. Llavors, sol hi ha una via per la qual ha estat més ràpida l'escriptura

amb PyArrow, en el moment de la codificació de la informació. **En la figura 6 es pot apuntar un clar avantatge de la serialització de l'estructura de dades al format columnar sobre la codificació al format Pickle.** En concret, en el cas d'escriure les dades del Pandas DataFrame més gran, derivat del fitxer CSV terasort de 500MB, l'escriptura amb la codificació PyArrow és 2.40 cops més ràpida que l'escriptura amb la codificació Pickle, i la serialització de les dades és 3.71 cops més ràpida. **Com més gran sigui l'estructura millor respon la serialització PyArrow versus Pickle**, advertint una major estabilitat dels temps d'execució davant el creixement de la mida de les dades.

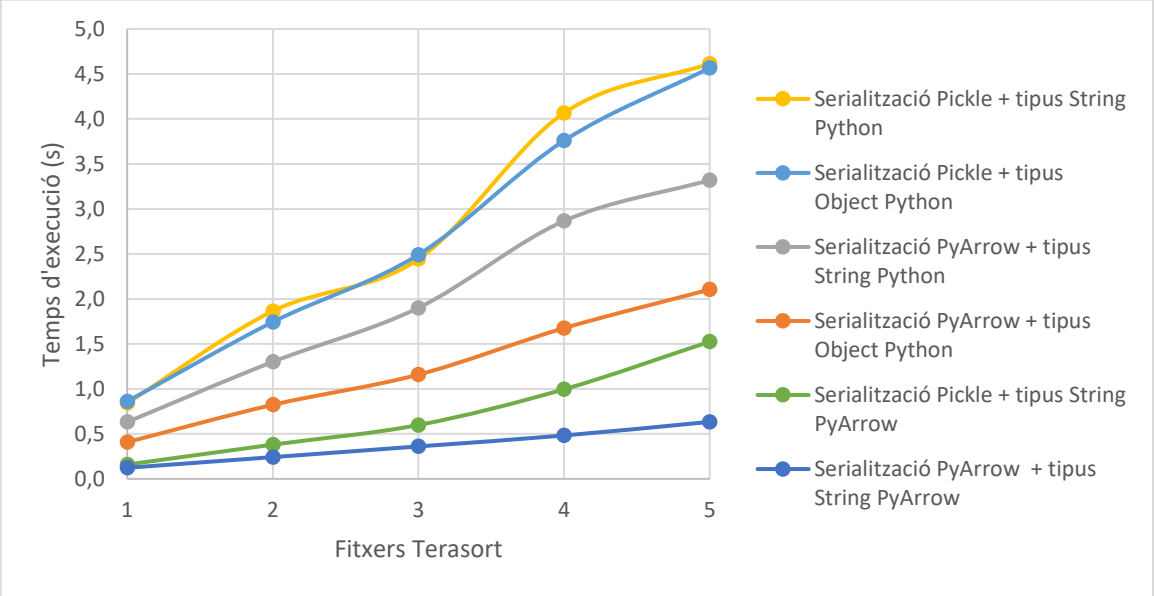

Figura 6. Temps d'execució derivats de l'escriptura a memòria compartida dels fitxers terasort mitjançant la serialització Pickle i PyArrow. En l'eix de les x estan representats els diferents fitxers Terasort amb números: 1 (100MB), 2 (200MB), 3 (300MB), 4 (400MB), 5 (500MB).

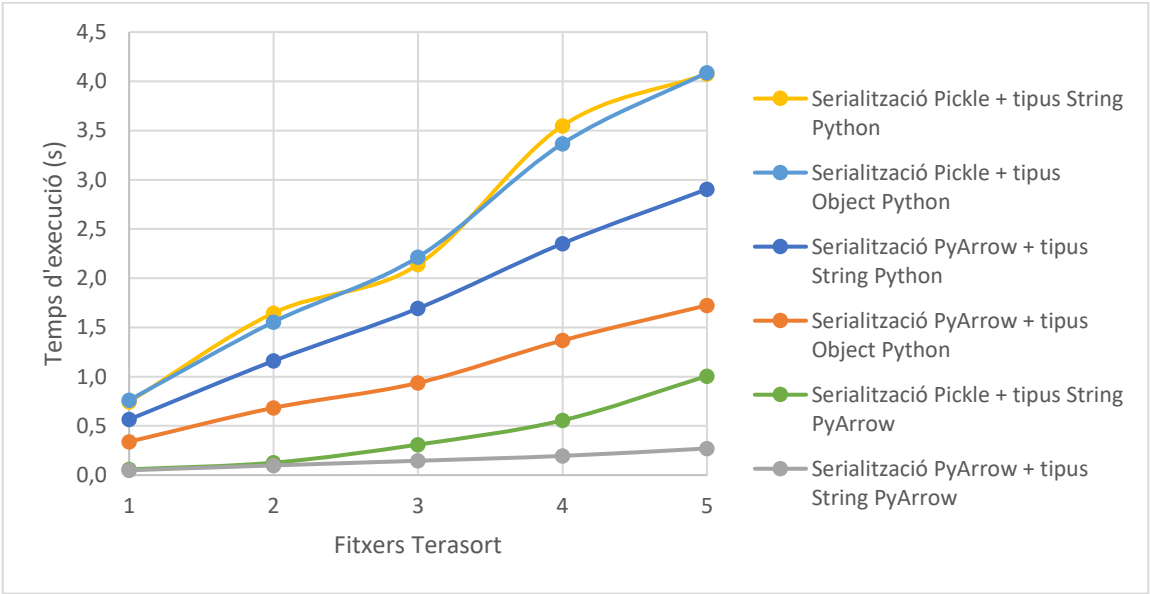

Figura 7. Temps d'execució donats per la codificació dels Pandas DataFrame mitjançant la serialització Pickle i PyArrow. Els Pandas DataFrame provenen dels fitxers CSV terasort. En l'eix de les x estan representats els diferents fitxers Terasort amb números: 1 (100MB), 2 (200MB), 3 (300MB), 4 (400MB), 5 (500MB).

En el cas de l'escriptura a Amazon S3 ens trobem uns resultats molt menys eficients. Si comparem els millors resultats donats per PyArrow serialitzant el fitxer més

gran i els millors aconseguits amb una escriptura a Amazon S3 amb el mateix fitxer, podem veure com la codificació al format columnar i l'escriptura a memòria compartida és 20.32 cops més veloç. Si observem la figura 5 i 7 queda clar que l'escriptura en un sistema intermediari com pot ser memòria compartida ens pot proporcionar un major rendiment. Si ens posem a comparar les dues llibreries (figura 7), tot i tenir uns temps d'execució bastant semblants, podem afirmar que boto3 és en general, una mica més veloç a l'hora d'escriure la informació.

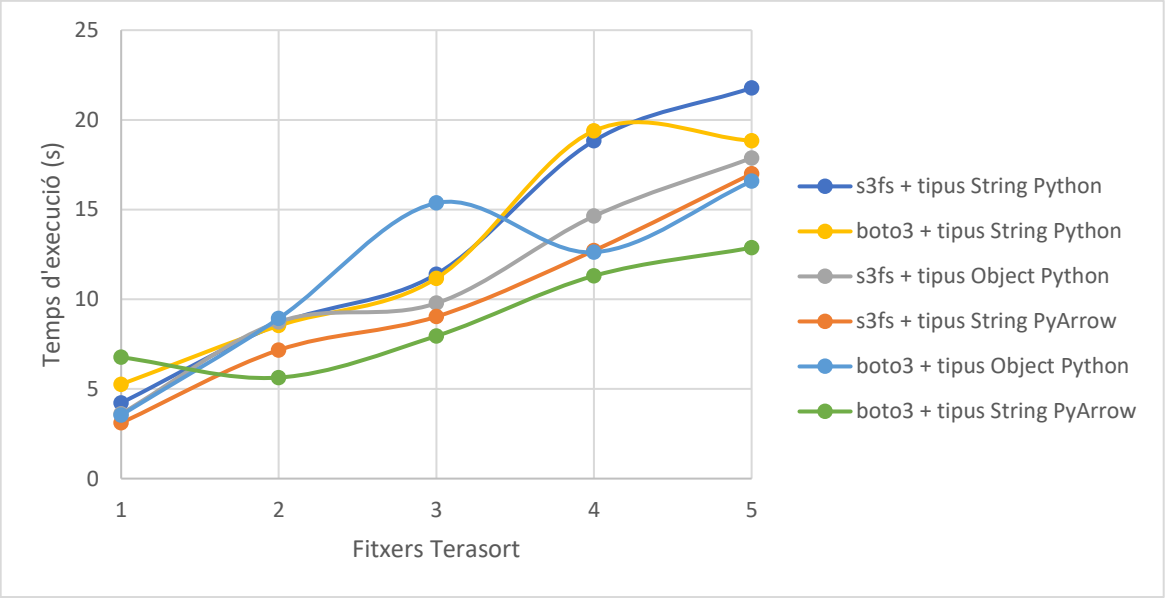

Figura 8. Temps d'execució derivats de l'escriptura a Amazon S3 dels fitxers terasort mitjançant la llibreria s3fs i boto3. En l'eix de les x estan representats els diferents fitxers Terasort amb números: 1 (100MB), 2 (200MB), 3 (300MB), 4 (400MB), 5 (500MB).

4.2.2.3.1.2 Brain

En el cas de treballar amb el DataFrame Brain, que compta amb columnes de tipus integer i float, es perden alguns dels avantatges que et proporciona treballar amb un tipus string i una serialització PyArrow. **Aquests tipus no tenen la mateixa complexitat en termes de variabilitat de la mida de la codificació i PyArrow no pot guanyar avantatge reduint la sobrecàrrega de memòria utilitzada per la representació**. És per això, que a pesar de comptar amb uns resultats d'execució una mica més bons utilitzant la serialització de PyArrow donada una ràpida codificació, sobretot amb estructures de dades més grans, existeix un rendiment similar.

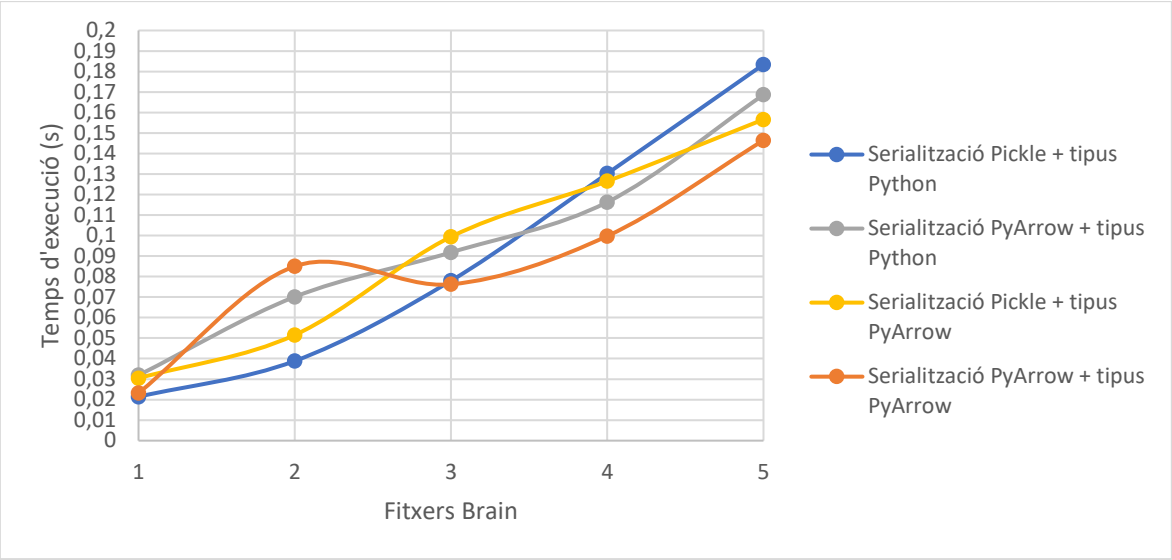

Figura 9. Temps d'execució derivats de l'escriptura a memòria compartida dels fitxers Brain mitjançant la serialització Pickle i PyArrow. En l'eix de les x estan representats els diferents fitxers Brain amb números: 1 (32MB), 2 (63MB), 3 (95MB), 4 (127MB), 5 (150MB).

En el cas de l'escriptura a memòria remota, els temps d'execució són molt similars utilitzant s3fs i boto3. El fet de treballar amb uns tipus menys complexos i més ràpids de serialitzar donada la mida de l'estructura, fa que les diferències siguin encara més mínimes que utilitzant els fitxers terasort.

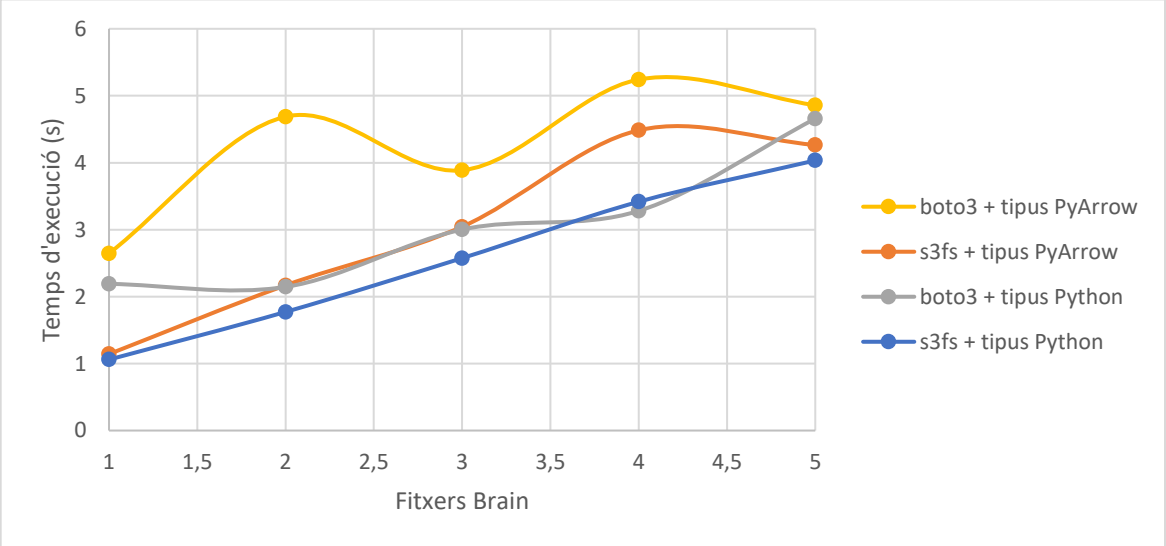

Figura 10. Temps d'execució derivats de l'escriptura a Amazon S3 dels fitxers Brain mitjançant la llibreria s3fs i boto3. En l'eix de les x estan representats els diferents fitxers Brain amb números: 1 (32MB), 2 (63MB), 3 (95MB), 4 (127MB), 5 (150MB).

4.2.2.3.2 Lectura del Pandas DataFrame des de memòria

4.2.2.3.2.1 Terasort

La lectura de les estructures de dades de memòria és una mica més interessant en termes de còpies i com afecten l'obtenció d'un rendiment òptim. Primer de tot, l'anàlisi es centrarà en la lectura dels DataFrames des de memòria compartida (figura 10). En aquest cas, igual que amb l'escriptura, **els temps de deserialització que ofereixen un major rendiment són els donats quan treballem amb un Pandas DataFrame amb tipus**

PyArrow. En el cas de la codificació Pickle, això és causat per la utilització d'una estructura de dades més compacta, amb uns tipus de dades que tenen una sobrecàrrega de memòria menor en la seva representació. **En el cas de la codificació PyArrow, el factor clau per aconseguir temps d'execució gairebé insignificants ha estat aconseguir una lectura de les dades amb còpia zero**, accedint i llegint la informació directament des de la zona de memòria compartida, sense la necessitat de fer una còpia per deserialitzar les dades com passa amb Pickle o amb PyArrow quan utilitzem els tipus de Python. **La lectura amb còpia zero no ha estat possible quan s'utilitzen els tipus Python**, ja que l'estructura Pandas DataFrame, si no treballa amb els mateixos tipus que el pyarrow.RecordBatch, s'ha de fer una còpia i conversió de les dades al format original.

Com es pot observar en la figura 10, a mesura que la mida dels fitxers augmenta, els temps d'execució també ho fan, però **en el cas de treballar amb serialització i tipus PyArrow, el creixement és molt petit a comparació de la resta**. Destacar en aquest punt el rendiment de la deserialització altament escalable, aconseguida per PyArrow, davant el creixement de la mida de les estructures de dades amb tipus PyArrow. Els temps de lectura han estat de 0.013s, 0.025s, 0.035s, 0.046s i 0.067s. En cap dels casos s'ha arribat a un decisegon en les lectures, inclús amb el Pandas DataFrame de 500MB. **En concret, s'ha realitzat una lectura 12.09 cops més veloç que utilitzant Pickle per llegir el Pandas DataFrame de 500MB.** Per posar en context els resultats obtinguts, la lectura del Pandas DataFrame de 500MB amb tipus PyArrow, fent servir la deserialització PyArrow, ha estat sols 0.006s més lenta que la lectura del Pandas DataFrame de 100MB utilitzant la deserialització Pickle.

En gairebé totes les situacions, menys en el cas de treballar amb el DataFrame de tipus Object, es pot determinar que la deserialització de PyArrow ha estat més veloç. En tots els casos l'explicació directa dels fets ha estat una deserialització de les dades més ràpida, com es pot observar en la figura 11, però en el cas d'utilitzar tipus PyArrow també ha tingut a veure el fet de treballar amb còpia zero, evitant els temps invertits a copiar l'estructura d'una zona de memòria a una altra.

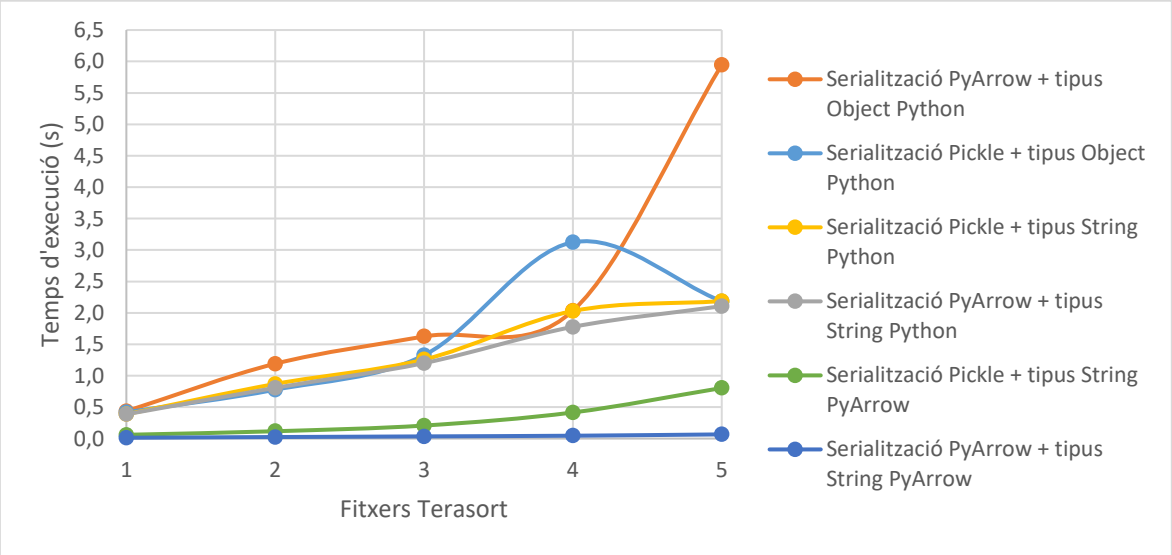

Figura 11. Temps d'execució derivats de la lectura de memòria compartida dels fitxers terasort mitjançant la deserialització Pickle i PyArrow. En l'eix de les x estan representats els diferents fitxers Terasort amb números: 1 (100MB), 2 (200MB), 3 (300MB), 4 (400MB), 5 (500MB).

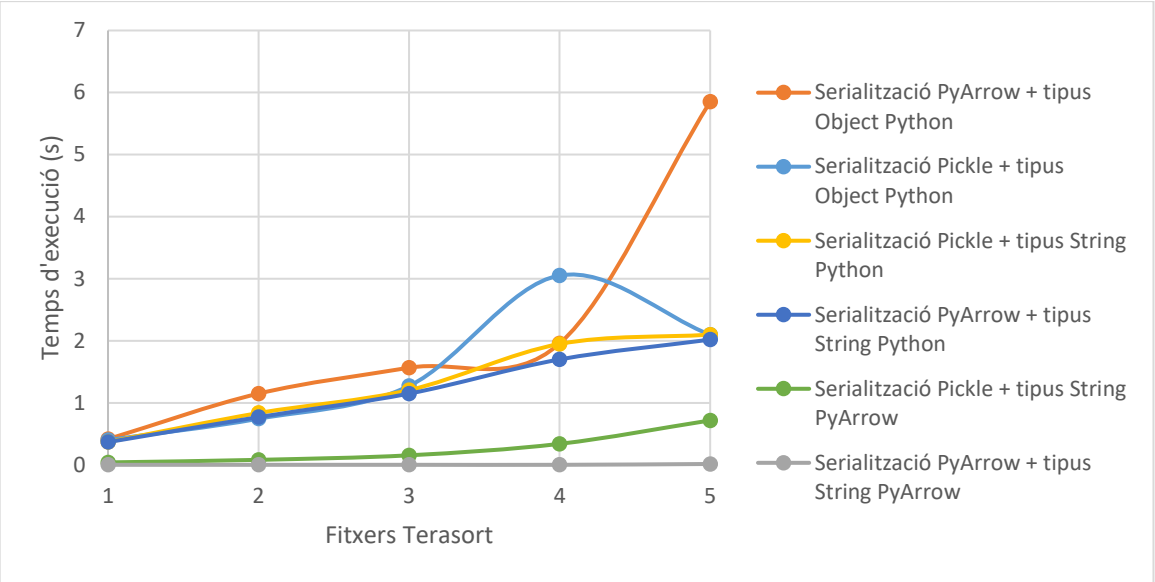

Figura 12. Temps d'execució donats per la descodificació dels Pandas DataFrame mitjançant la deserialització Pickle i PyArrow. Els Pandas DataFrame provenen dels fitxers CSV terasort. En l'eix de les x estan representats els diferents fitxers Terasort amb números: 1 (100MB), 2 (200MB), 3 (300MB), 4 (400MB), 5 (500MB).

La lectura i deserialització de les dades amb Amazon S3, com en el cas de l'escriptura, tornen a tenir un rendiment molt inferior que quan es treballa amb memòria compartida. Per exemple, **treballant amb el fitxer de 500MB amb tipus PyArrow**, si comparem els millors temps de lectura d'Amazon S3 versus els millors fent servir memòria compartida, tenim que **la lectura des de *shared memory* utilitzant la descodificació PyArrow és 130.08 cops més ràpida que la lectura des d'Amazon S3.**

Si comparem les dues llibreries utilitzades per llegir des de S3 es pot observar un rendiment similar i uns temps d'execució amb els tipus PyArrow inferiors a la resta. Això és causat per una major rapidesa en la conversió d'un fitxer pyarrow Parquet a estructures PyArrow com pyarrow.Table. En aquest cas, al contrari que en l'escriptura de les dades, la llibreria s3fs dona un rendiment major a l'hora de llegir la informació.

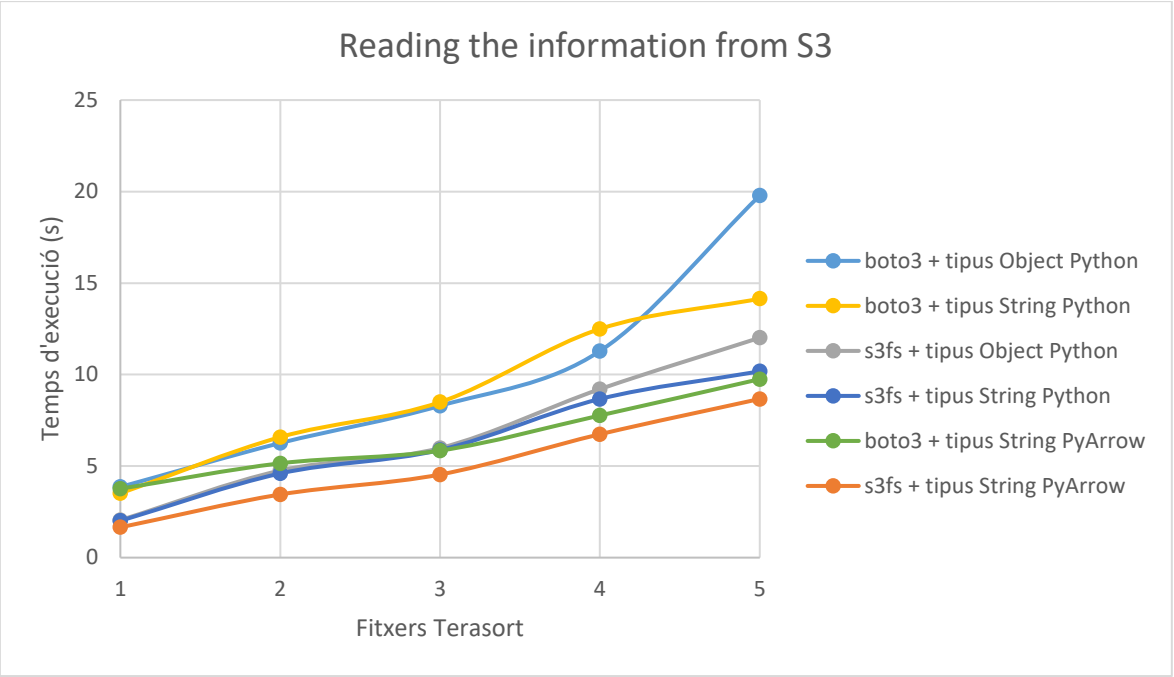

Figura 13. Temps d'execució derivats de la lectura d'Amazon S3 dels fitxers terasort mitjançant la llibreria s3fs i boto3. En l'eix de les x estan representats els diferents fitxers Terasort amb números: 1 (100MB), 2 (200MB), 3 (300MB), 4 (400MB), 5 (500MB).

Els punts clau de l'anàlisi dels temps d'execució donats per la lectura del Pandas DataFrame Terasort són els següents:

- Els temps d'execució més òptims són els donats per la lectura de Pandas DataFrame amb tipus PyArrow.
- La deserialització PyArrow és la que dona un major rendiment. Sobretot en el cas de treballar amb els tipus PyArrow (hi ha compatibilitat de tipus entre estructures), ja que es fa la lectura amb còpia zero i s'han vist resultats molt destacables com una lectura de l'arxiu de 500MB 12.09 cops més ràpida que amb Pickle.
- La lectura i deserialització amb Amazon S3 dona un rendiment molt inferior que treballant amb memòria compartida i deserialització Pickle. En el cas d'aconseguir la còpia zero, PyArrow llegeix 130.08 cops més ràpid el fitxer de 500MB que si es fa la lectura des d'Amazon S3 mitjançant s3fs.

4.2.2.3.2.2 Brain

En el cas de treballar amb el DataFrame Brain, amb unes mides de les estructures de dades més petites i una utilització de tipus amb una codificació de mida fixa, **l'avantatge de la deserialització PyArrow és redueix**. Com es pot veure en la figura 13, **PyArrow segueix sent la forma més bona per descodificar les dades gràcies a la velocitat amb la qual treballa. Aquesta rapidesa encara pot arribar a ser més accentuada en el cas de treballar amb un DataFrame amb tipus PyArrow aconseguint còpia zero**, ja que s'eviten els temps que comporten la còpia d'una estructura de dades d'una zona de memòria a una altra, gràcies a la compatibilitat de tipus entre l'estructura `pyarrow.RecordBatch` i Pandas DataFrame.

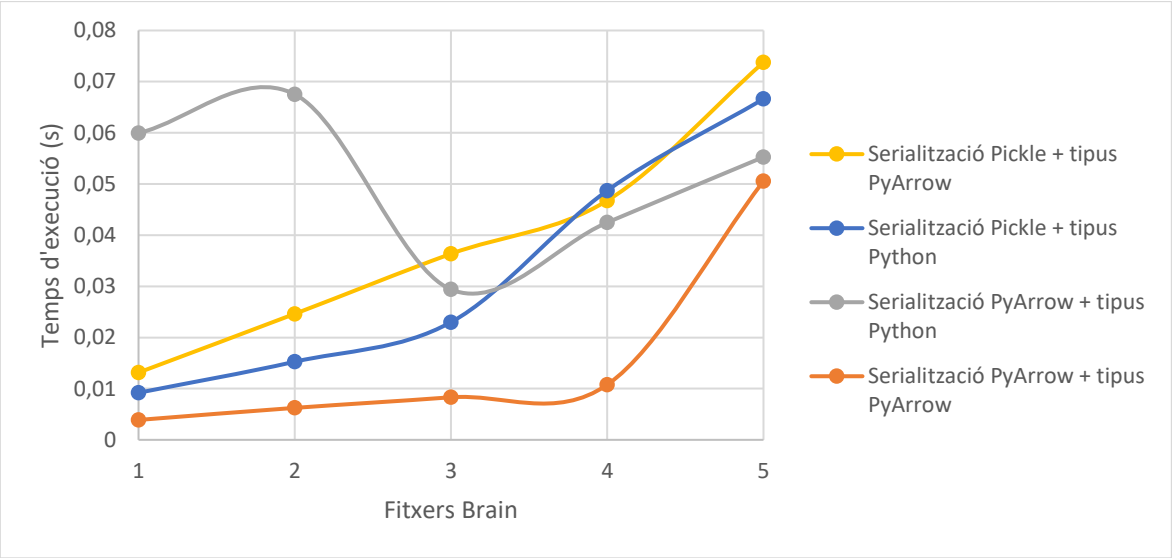

Figura 14. Temps d'execució donats per la descodificació dels Pandas DataFrame mitjançant la deserialització Pickle i PyArrow. En l'eix de les x estan representats els diferents fitxers Brain amb números: 1 (32MB), 2 (63MB), 3 (95MB), 4 (127MB), 5 (150MB).

La lectura i deserialització dels objectes mitjançant Amazon S3 segueixen amb uns temps molt menys eficients que al treballar amb memòria compartida com sistema intermediari. En aquest cas, al igual que passava en la lectura dels fitxers Brain, la llibreria s3fs dona un rendiment més eficient.

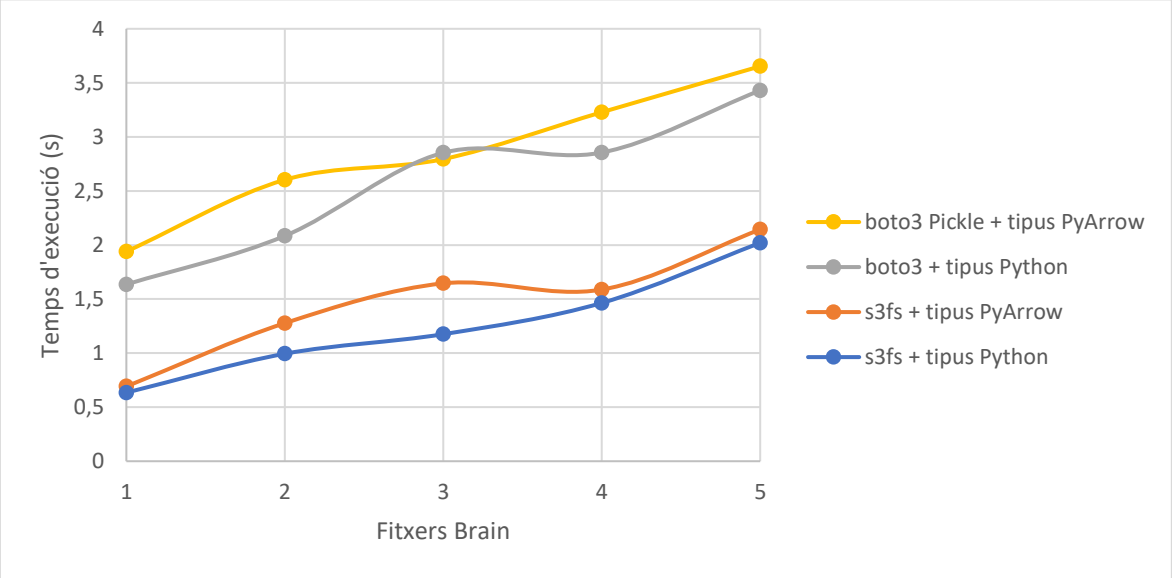

Figura 15. Temps d'execució derivats de la lectura d'Amazon S3 dels fitxers Brain mitjançant la llibreria s3fs i boto3. En l'eix de les x estan representats els diferents fitxers Brain amb números : 1 (32MB), 2 (63MB), 3 (95MB), 4 (127MB), 5 (150MB).

4.3 Anàlisi de la compartició de particions amb un model MapReduce utilitzant diferents sistemes d'emmagatzemament intermediaris

4.3.1 Metodologia

L'estudi executarà un codi que segueix un model MapReduce per processar un conjunt de dades d'entrada de forma paral·lela (apartat 3.2). Aquest codi ha estat adaptat per poder compartir les particions de dades entre els *mappers* i *reducers* a través de diferents sistemes d'emmagatzematge. D'aquesta forma, es podrà valorar l'eficàcia de les comparticions a través de memòria compartida versus altres sistemes d'emmagatzematge més lents i que impliquen un major nombre de còpies. Els sistemes d'emmagatzematge utilitzats són els següents:

- **multiprocessing.shared_memory utilitzant serialització PyArrow.**
- **Disc utilitzant l'API Storage de Lithops amb el mode localhost per interaccionar amb el backend.**
- **Amazon S3 utilitzant la llibreria s3fs per interaccionar amb el backend.**
- **Amazon S3 utilitzant la llibreria boto3 per interaccionar amb el backend.**

Les dades d'entrada que seran dividides en particions i processades pels diferents processos seran fitxers CSV. En concret, es farà servir l'arxiu Terasort. Aquest CSV està format per dos columnes de strings, que aquesta vegada seran interpretats amb els tipus PyArrow amb l'objectiu d'optimitzar el rendiment. Els fitxers CSV tindran una mida d'1GB, 3GB i 5GB.

Les execucions del codi, per poder aprofitar al màxim el paral·lelisme, es faran sobre una màquina virtual d'Amazon (Amazon EC2) amb 32 nuclis. D'aquesta manera, és dura a terme el MapReduce dels tres CSV amb diferents mides mitjançant un paral·lelisme de 4, 8, 16 i 32 processos. Això permetrà tenir fins a 32 processos *mappers i reducer*.

En aquest cas, l'anàlisi valorarà el paràmetre més important a l'hora de processar grans quantitats de dades, els temps d'execució. Tot això tenint en compte altres paràmetres com els estudiats en l'anàlisi de l'apartat 4.2 (mida de les estructures de dades i utilització de la memòria).

4.3.2 Valoració dels resultats

En el cas d'utilitzar memòria compartida com a sistema intermediari entre els *mappers* i *reducers*, podem observar com el temps d'intercanvi disminueixen a mesura que augmenta al nombre de processos. En la figura 16 es pot apreciar com en tots els casos, a mesura que augmentem el nombre de processos encarregats de tractar les particions, els temps d'execució es redueixen. Tot i això, el percentatge de millora dels temps d'execució comencen a reduir-se cada cop que ens aproximem a un major nombre de processos. **Quan passem de treballar de 4 a 8 processos hi ha unes millores dels temps d'un 41,35% (1GB), 42,79% (3GB) i 42,74% (5GB). Si passem a treballar de 8 a 16 processos hi ha unes millores dels temps d'execució d'un 31,48% (1GB), 34,44% (3GB) i 35,49% (5GB). Finalment, quan passem de treballar de 16 a 32 processos hi ha unes millores del 11,20% (1GB), 14,62% (3GB) i 14,23% (5GB).** Això ens indica que hi haurà un moment en que l'increment de processos no significarà una millora dels temps d'execució (sobretot amb els fitxers més petits). Ara bé, amb els processos utilitzats els resultats demostren que la memòria no ha estat un coll d'ampolla a pesar d'haver-hi un increment de la quantitat de dades intercanviades.

L'explicació dels bons resultats adquirits ha estat la reducció del nombre de còpies realitzades (aquestes causen un *overhead* a nivell de CPU i duplicacions en memòria) i de la utilització d'un sistema de serialització i deserialització de les dades eficient. En el cas de les còpies, sol hi ha hagut un increment considerable de la memòria usada en aquests casos: dos cops en l'escriptura, en el moment de la serialització i la transmissió de les dades a memòria, i dos cops en la lectura, en el moment que el *reducer* fa l'ordenació del Pandas DataFrame i de l'escriptura del resultat final a memòria.

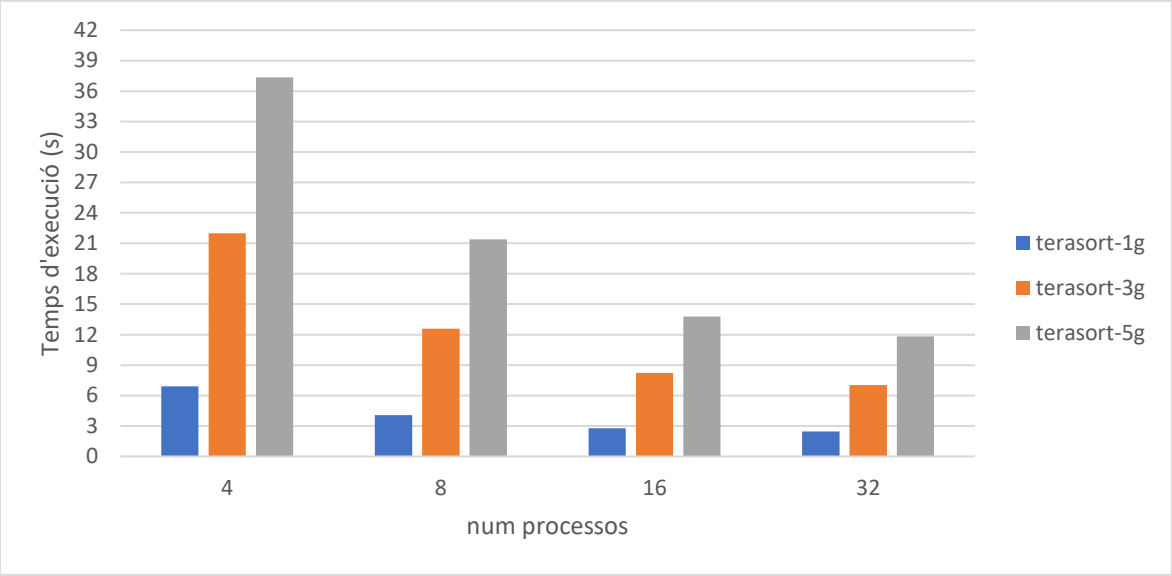

Figura 16. Temps d'execució derivats del processament dels fitxers Terasort d'1GB, 3GB i 5GB amb un model MapReduce. En aquest cas s'utilitza memòria compartida per compartir les particions entre els processos.

Si es compara tots els sistemes d'emmagatzemament intermediaris utilitzats per compartir les particions, queda clar el rendiment diferencial que ofereix l'ús de memòria compartida amb codificació PyArrow sobre la resta (figura 17, 18, 19).

En el cas del disc, podem veure com dona un rendiment lleugerament superior respecte a la utilització d'Amazon S3. Sobretot si tenim en compte els moments on s'usa un alt paral·lelisme. Tot i això, si ens fixem en els resultats adquirits usant els fitxers més grans (figura 18 i 19), podem observar com treballant amb un paral·lelisme de 4 i 8 processos tenim uns resultats bastant pareguts als d'Amazon S3 fent servir la llibreria s3fs o inclús una mica pitjors. Això reflecteix una dificultat en l'escriptura de particions molt grans (com menys paral·lelisme, les particions que s'han d'escriure i llegir seran més grans). Si les mides dels fitxers continuen creixent, podem afirmar que disc provocaria un *overhead* considerable donada la seva lenta entrada i sortida de dades, com també hi hauria un gran *overhead* si es treballa amb fitxers més grans i un paral·lelisme baix. **Si comparem disc amb memòria compartida, les diferències són molt notables.** Per exemple, en el cas de treballar amb el fitxer terasort de 5GB (figura 19), **els temps d'execució utilitzant *shared memory* amb *zero-copy* com sistema intermediari són un 49,33% (4 processos), 47,77% (8 processos), 48,14% (16 processos) i 54,04% (32 processos) més ràpids que els adquirits amb disc.** No sols comencen sent molt millors, sinó que a mesura que augmenta el paral·lelisme la diferència de rendiment també ho fa.

En el cas de l'ús d'Amazon S3, podem veure com la implementació amb la llibreria s3fs dona un molt millor rendiment que la de boto3. Això ve donat que s3fs utilitza diferents fils d'execució per fer les peticions a Amazon S3 per llegir les diferents particions guardades pels *mappers*, en canvi, boto3 no ho fa. L'explicació d'aquest fet és

que els clients de boto3 no són *thread safe* i no poden ser utilitzats alhora en múltiples fils d'execució. **Si comparem els resultats de s3fs amb els donats per *shared memory*, podem veure com les diferències són encara més grans que amb la utilització de disc**. Mentre l'ús d'Amazon S3 a través de la llibreria s3fs comença a empitjorar els resultats un cop tenim un paral·lelisme de 32 processos, l'ús de memòria compartida sempre millora els resultats a mesura que hi ha més processos. A part, els temps d'execució són significativament més ràpids amb memòria compartida. Per exemple**, en el cas de treballar amb el fitxer terasort de 5GB (figura 19), els temps d'execució fent servir *shared memory* com sistema intermediari són un 47,93% (4 processos), 48,48% (8 processos), 56,60% (16 processos) i 65,22% (32 processos) més ràpids que els obtinguts amb Amazon S3.**

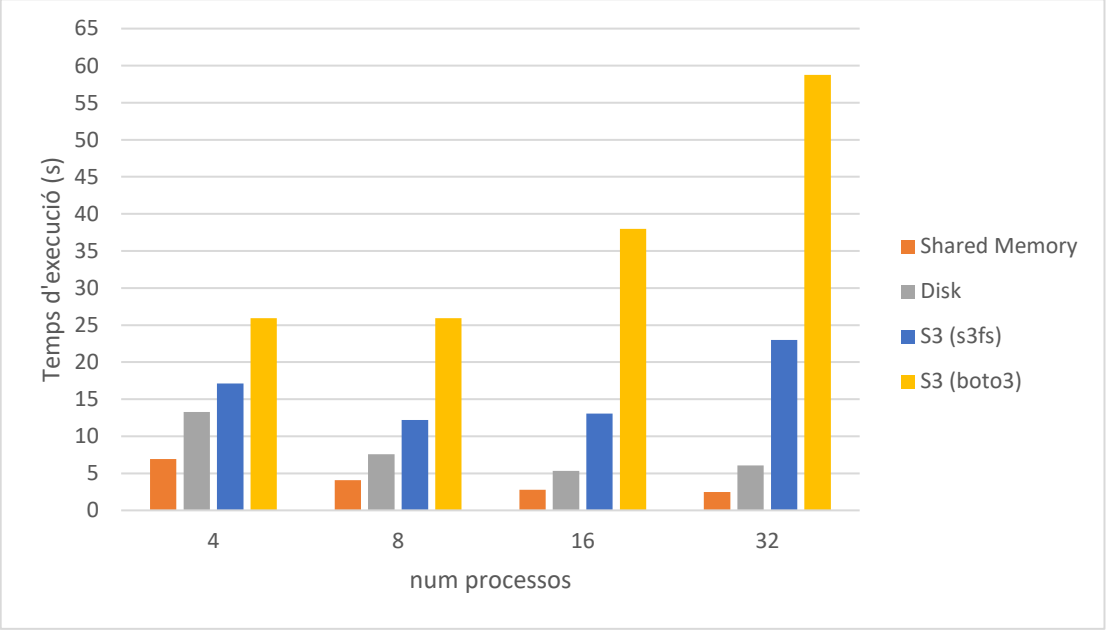

Figura 17. Temps d'execució derivats del processament del fitxer Terasort d'1GB amb un model MapReduce i utilitzant diferents sistemes d'emmagatzematge intermediaris.

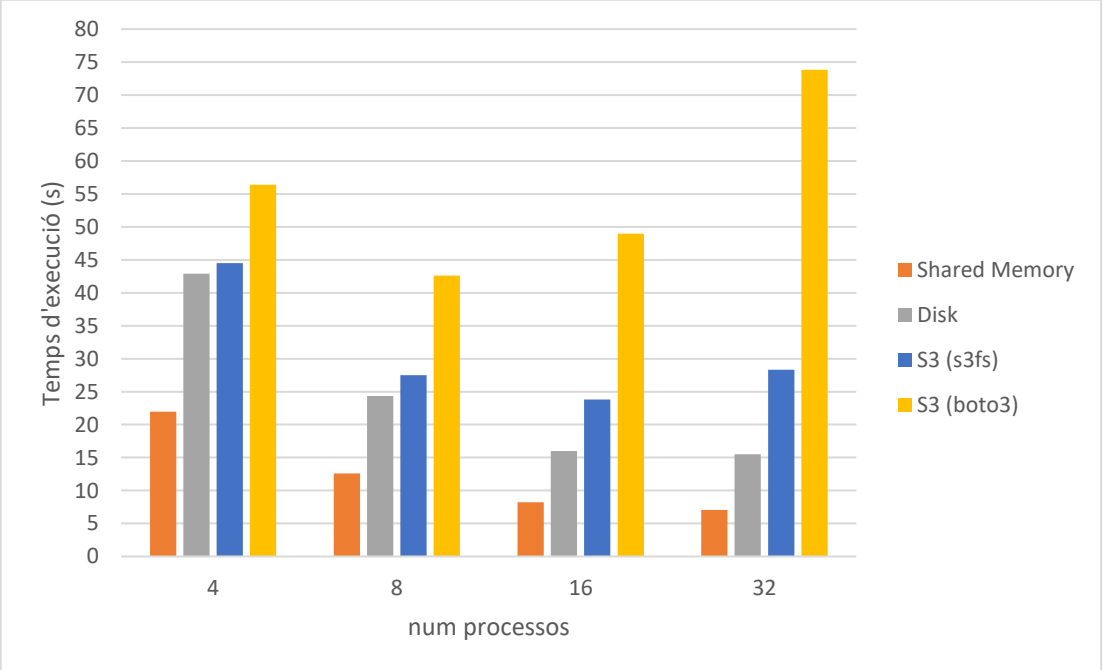

Figura 18. Temps d'execució derivats del processament del fitxer Terasort de 3GB amb un model MapReduce i utilitzant diferents sistemes d'emmagatzematge intermediaris.

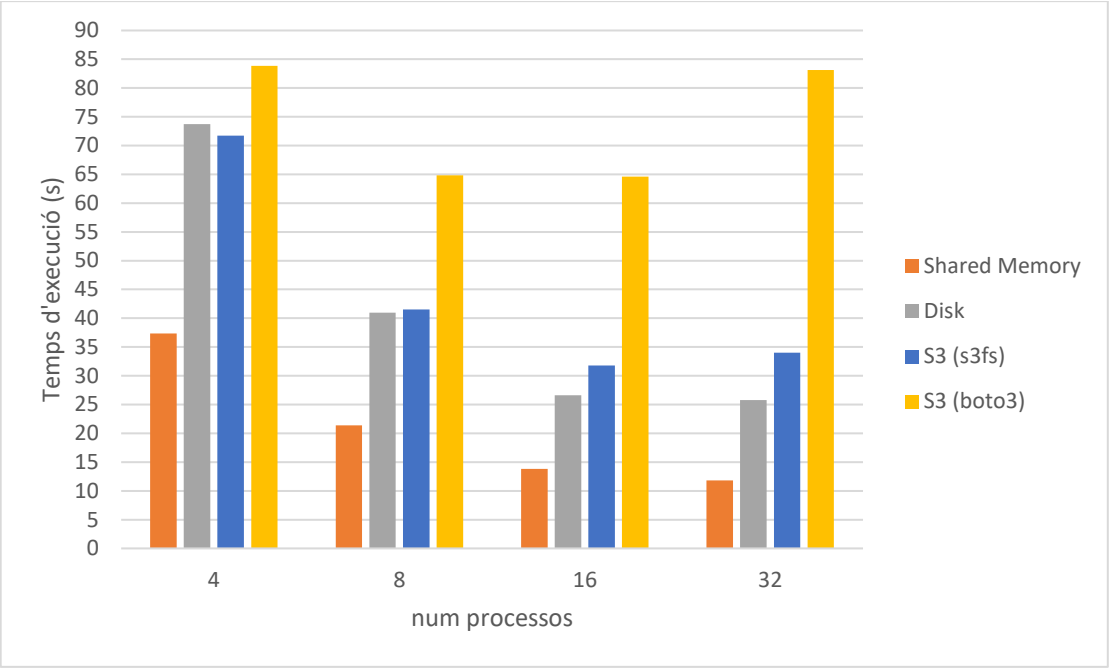

Figura 19. Temps d'execució derivats del processament del fitxer Terasort de 5GB amb un model MapReduce i utilitzant diferents sistemes d'emmagatzematge intermediaris.

Els punts clau de l'anàlisi dels temps d'execució donats per l'intercanvi de dades a gran escala mitjançant el model MapReduce són els següents:

- Utilitzant memòria compartida amb còpia zero els temps d'intercanvi de dades disminueixen a mesura que augmenta el nombre de processos. En el cas d'usar disc i Amazon S3 com intermediaris hi ha un moment que els temps deixen de millorar.
- L'ús de memòria compartida i la serialització Pyrrow donen els resultats més destacables. El fet de no haver de realitzar còpies en la lectura de les dades en el cas de treballar amb memòria compartida i serialització PyArrow ha estat clau per obtenir un rendiment òptim.
- Disc es veu perjudicat quan hi ha un baix paral·lelisme i unes mides de les particions considerables. De fet, a pesar de ser en norma general millor que la utilització d'Amazon S3, quan s'utilitzen els fitxers de 3GB i 5GB amb un paral·lelisme de 4 i 8 processos, s'aconsegueix un rendiment similar.
- En el cas d'usar Amazon S3, podem veure com la implementació amb la llibreria s3fs dona un millor rendiment que la de boto3. Això és causat per l'ús de diferents fils d'execució per fer les peticions a Amazon S3 en el cas de s3fs. En boto3 això no és possible perquè no és *thread safe.*

5 Valoració de la integració del nou mecanisme de compartició dins de Seer

Seer és un sistema d'anàlisi de dades que segueix el model de funcions. Actualment, les tasques de processament s'executen mitjançant FaaS, com Amazon Lambda, i l'intercanvi de dades entre les funcions es fa a través de sistemes d'emmagatzemament remot d'alta latència. Aquest model permet ser molt ràpid en l'aprovisionament dels recursos necessaris per executar les funcions, però els temps de comunicació de particions no són competitius amb sistemes com memòria compartida amb còpia zero. Davant aquestes limitacions i els resultats adquirits durant els estudis, es pot valorar la incorporació d'altres mecanismes que poden permetre un rendiment més competitiu.

Hi ha certes situacions on la utilització d'una màquina local, un clúster o una màquina virtual com EC2 (aprovisionada prèviament per evitar la sobrecàrrega d'iniciar-la) poden ser molt útils per executar les funcions de processament de dades. Si aquestes màquines tenen unes característiques suficients per executar una càrrega de treball determinada, es pot aprofitar la rapidesa en l'intercanvi de particions a través de memòria compartida i còpia zero.

Seer podria mirar la mida de les dades d'entrada a processar i mesurar el nombre de funcions a utilitzar. Si les màquines tenen una capacitat d'emmagatzematge suficient per guardar aquesta quantitat de dades d'entrada i tenen un nombre de CPUs que s'ajusta al paral·lelisme necessitat, poden ser una alternativa a la utilització de sistemes com Amazon Lambda com executores de funcions. El *framework*, usant uns criteris de distribució més refinats, també podria valorar l'execució de part del processament mitjançant Amazon Lambda i part de l'altre a través de màquines capaces de compartir les dades amb memòria compartida i sense còpies.

Una altra situació on podria ser interessant implementar el mecanisme de compartició és en processaments de dades confidencials. Hi ha certes dades personals o empresarials que no poden ser emmagatzemades en sistemes *multitenant* [42], com passa amb la majoria del *cloud*, és per això que s'ha d'usar una memòria intermediària de la qual fóssim propietaris. Aquest és un cas habitual a nivell empresarial i un bon sistema de processament de dades en màquines personals pot ser clau.

Davant els diferents aspectes valorats, Seer podria augmentar la seva API d'aquesta forma:

- 1) Un selector de *backend* (local versus remot) intel·ligent, basat en els paràmetres del treball a executar.
- 2) Un selector de *backend* basat en confidencialitat, definida per l'usuari.

6 Conclusions

En aquest treball de final de grau s'han dut a terme investigacions i desenvolupaments en camps relacionats amb l'anàlisi de dades a gran escala. S'han treballat múltiples conceptes com les arquitectures FaaS i diferents mètodes de compartició de particions entre funcions. Tot això, centrant l'interès en la implementació d'un sistema de compartició de dades amb còpia zero que mostra millores de rendiment davant altres sistemes ja utilitzats en frameworks contrastats com Lithops. Finalment, tenint en compte els diferents estudis realitzats, s'ha valorat la incorporació del mecanisme de compartició implementat en Seer, sistema en el qual treballaré conjuntament amb altres membres del CloudLab (URV) aquests pròxims mesos per poder fer possibles les modificacions.

El treball de final de grau ha estat una experiència molt satisfactòria de la qual estic encantat amb tots els reptes d'aprenentatge que he afrontat i de la utilitat que tindran en un futur pròxim els resultats obtinguts. A més, m'ha servit per donar-me conte dels passos que he de fer a partir d'ara en el meu futur professional.

Referències

[1] Computació al núvol. URL: https://en.wikipedia.org/wiki/Cloud_computing (visitat 03-06-2023)

[2] Amazon S3: https://aws.amazon.com/es/s3/ (visitat 04-06-2023)

[3] Amazon EC2: https://aws.amazon.com/es/ec2/ (visitat 04-06-2023)

[4] Infrastructure as a Service: https://en.wikipedia.org/wiki/Infrastructure_as_a_service (visitat 05-06-2023)

[5] Computació sense servidor: https://en.wikipedia.org/wiki/Serverless_computing (visitat 06-06-2023)

[6] Amazon Web Services: https://aws.amazon.com/es/s3/ (visitat 03-06-2023)

[7] AWS Lambda: https://aws.amazon.com/es/lambda/ (visitat 10-06-2023)

[8] Function as a Service: https://en.wikipedia.org/wiki/Function_as_a_service (visitat 12-06-2023)

[9] StreamAlert: Real-time Data Analysis and Altering: https://medium.com/airbnb-engineering/streamalert-real-time-data-analysis-and-alerting-e8619e3e5043 (visitat 15-07-2023)

[10] Why Coca-Cola Migrated to AWS: https://dheeth.netlify.app/aws-coca-cola-case-study/ (visitat 16-07-2023)

[11] Netflix & AWS Lambda Case Study: https://aws.amazon.com/es/solutions/case-studies/netflix-and-aws-lambda/ (visitat 16-07-2023)

[12] Azure Functions: https://learn.microsoft.com/en-us/azure/azure-functions/ (visitat 20-07-2023)

[13] Google Cloud Functions: https://cloud.google.com/functions (visitat 21-07-2023)

[14] PyWren: http://pywren.io/ (visitat 15-06-2023)

[15] Lithops: https://lithops-cloud.github.io/ (visitat 15-06-2023)

[16] Memòria compartida: https://en.wikipedia.org/wiki/Shared_memory#:~:text=In%20computer%20hardware%2C%20shared%20memory,in%20a%20multiprocessor%20computer%20system. (visitat 20-06-2023)

[17] Còpia zero: https://en.wikipedia.org/wiki/Zero-copy (visitat 22-06-2023)

[18] Marc Sánchez Artigas, Germán Eizaguirre, "A seer knows best: optimized object storage shuffling for serverless analytics – Proceedings of the 23rd ACM/IFIP International Middleware Conference, Nov 2022, pàgines 148-160. (visitat 29-06-2023)

[19] Python multiprocessing: https://docs.python.org/3/library/multiprocessing.html (visitat 30-06-2023)

[20] Multirpocessing Shared Memory: https://docs.python.org/3/library/multiprocessing.shared_memory.html (visitat 01-07-2023)

[21] PyArrow Plasma: https://arrow.apache.org/docs/1.0/python/plasma.html (visitat 30-06-2023)

[22] PyArrow: https://arrow.apache.org/docs/python/index.html (visitat 02-07-2023)

[23] Garbage Collector: https://en.wikipedia.org/wiki/Garbage_collection_(computer_science) (visitat 03-07-2023)

[24] Object Store: https://en.wikipedia.org/wiki/Object_storage (visitat 04-06-2023)

[25] Apache Arrow: https://arrow.apache.org/ (visitat 02-07-2023)

[26] Big Data: https://en.wikipedia.org/wiki/Big_data (visitat 20-08-2023)

[27] Ray: https://www.ray.io (visitat 15-08-2023)

[28] Inter Process Communication: https://en.wikipedia.org/wiki/Inter-process_communication (visitat 30-06-2023)

[29] Arrow Columnar Format: https://arrow.apache.org/docs/format/Columnar.html (visitat 30-06-2023)

[30] Internet de les coses: https://en.wikipedia.org/wiki/Internet_of_things (visitat 20-08-2023)

[31] Bucket d'Amazon S3: https://docs.aws.amazon.com/es_es/AmazonS3/latest/userguide/UsingBucket.html (visitat 04-06-2023)

[32] Objectes d'Amazon S3: https://docs.aws.amazon.com/AmazonS3/latest/userguide/UsingObjects.html (visitat 05-06-2023)

[33] Diagrama interacció amb bucket de S3: https://www.google.com/url?sa=i&url=https%3A%2F%2Fcloudonaut.io%2Fintroducing-the-object-store-s3%2F&psig=AOvVaw2mzRw_E5Swrhez2b3Qa3Dn&ust=1693601464427000&source=images&cd

=vfe&opi=89978449&ved=0CBAQjRxqFwoTCNjD4J_jh4EDFQAAAAAdAAAAABAE (visitat 06-06-2023)

[34] Boto3: https://boto3.amazonaws.com/v1/documentation/api/latest/index.html (visitat 10-07-2023)

[35] S3fs: https://github.com/s3fs-fuse/s3fs-fuse (visitat 12-07-2023)

[36] Format columnar versus format en fila: http://peter-hoffmann.com/2018/euroscipy-2018-apache-parquet-as-a-columnar-storage-for-large-datasets.html (visitat 1-07-2023)

[37] Snappy (compression): https://en.wikipedia.org/wiki/Snappy_(compression) (visitat 20-08-2023)

[38] Pandas DataFrame: https://www.geeksforgeeks.org/python-pandas-dataframe/ (visitat 04-07-2023)

[39] Lithops API Storage: https://lithops-cloud.github.io/docs/source/api_storage.html (visitat 21-05-2023)

[40] Lithops map_reduce: https://github.com/lithops-cloud/lithops/blob/master/examples/map_reduce.py (visitat 22-05-2023)

[41] Memory profiler: https://pypi.org/project/memory-profiler/ (visitat 05-07-2023)

[42] Sistema multitenant: https://platzi.com/blog/multi-tenant-que-es-y-por-que-es-importante/ (visitat 10-08-2023)