Università degli Studi di Modena e Reggio Emilia

Dipartimento di Ingegneria "Enzo Ferrari"

Corso di Laurea in Ingegneria Informatica

Scalabilità di Tecniche di Analisi di Dati con la libreria Modin

Relatore:
Chiar.ma Prof.ssa Sonia Bergamaschi

Correlatore:
PhD. Luca Gagliardelli

Candidato:
Francesca Prontera

Anno Accademico 2019/2020

0. Abstract

In questo elaborato verrà per prima analizzata la struttura dati *DataFrame*, molto popolare in ambito di analisi dei dati perchè rende più rapida ed efficace tale analisi.

Si parlerà poi della libreria che rappresenta lo standard de facto in ambito di gestione dei DataFrame, *Pandas*, sviluppata in linguaggio Python. Nonostante tale libreria offra diversi metodi per la manipolazione dei dati all'interno dei DataFrame, a causa della sua limitata scalabilità non risulta la scelta più adatta se si vuole lavorare con grandi quantità di dati. Per sopperire a tale problema, l'università di Berkley ha progettato una nuova API, che prende il nome di *Modin* ed è in grado di sfruttare tecniche di multicore per l'elaborazione parallela dei dati all'interno dei DataFrame.

La nuova libreria ha come obiettivo quello di sostituirsi in modo trasparente a Pandas, utilizzando quindi la sua stessa sintassi in modo da ridurre il tempo di apprendimento degli utenti che già utilizzano Pandas.

L'obiettivo di questo elaborato è quello di analizzare entrambe le librerie e di metterle a confronto, facendone emergere i rispettivi vantaggi e svantaggi tramite dei confronti (*benchmark)* effettuati su computer con caratteristiche computazionali diverse.

Tali *benchmark* dimostreranno come l'utilizzo di Modin possa essere non efficiente se il dataset è centralizzato, ma efficiente se si opera su sistemi distribuiti per l'elaborazione di grandi quantità di dati.

Indice

1. Introduzione

I DataFrame nascono con l'obiettivo di sopperire alla limitazione a cui sono soggette altre strutture dati ovvero, l'obbligo di dover definire i tipi di dati prima di poter effettuare qualsiasi operazione su di essi.
Per cui in analisi complesse, la dichiarazione di query come quelle SQL all'interno di un database relazionale rende scomodo lo sviluppo e il debug di esse.

In alternativa, linguaggi di programmazione come Python supportano la cosiddetta *astrazione del DataFrame*, fornendo così alla struttura dati un'interfaccia funzionale utile per rappresentare, preparare e analizzare i dati.
Per quanto sia stato notevole il successo delle librerie di DataFrame Python, esse devono affrontare problemi prestazionali, soprattutto se si sta lavorando su set di dati moderatamente grande. Questi problemi prestazionali li ha anche la libreria principale per l'elaborazione dei dati in Python ossia, Pandas.

Non volendo far abbandonare agli utenti un'interfaccia ormai familiare come quella di Pandas, si è deciso di implementare una nuova libreria che si rifacesse alla principale ma che riuscisse a parallelizzare le varie operazioni in fase di esecuzione, riuscendo ad essere performante anche su grandi dataset. Questa libreria prende il nome di Modin.
Modin è una libreria ancora in fase di sviluppo e miglioramento, per cui si richiederà un'estensione dello stato dell'arte in molte dimensioni per la gestione dei dati.

In questo documento si darà una visione d'insieme per sistemi di DataFrame scalabili in particolare, si discuterà di:

- DataFrame: le loro origini e la loro architettura, che li rende una scelta valida in ambito dell'esplorazione dei dati. Verranno poi posti in relazione alle varie strutture dati;
- Pandas: l'API scritta in Python maggiormente utilizzata in ambito di manipolazione dei DataFrame. Verranno esposti i suoi punti di forza e la sua evoluzione;
- Modin: la nuova API che integra la sua versione precedente andando ad implementare il parallelismo nell'esecuzione di query all'interno di un dataset, ne verrà descritta l'architettura e il relativo funzionamento;

- Dask: uno dei due backend sul quale viene distribuito Modin, verrà discusso il suo funzionamento in ambito centralizzato mettendolo a confronto con le diverse API e facendo emergere i suoi punti di forza e quelli che sono i suoi punti critici;
- Benchmark: in una prima parte verranno esposti e commentati i report sui tempi di esecuzione di query Pandas e Modin relativi alle diverse macchine e backend, ed infine si farà riferimento ad un caso studio su di un cluster che sfrutta uno scheduler distribuito.

2. DataFrame

Dare una definizione di DataFrame non è immediato, poichè ad oggi tanti sistemi diversi prendono questo nome e il termine è quindi sul punto di non avere alcun significato.
Per cui, per dare un quadro d'insieme di questa struttura, si partirà dalla definizione risalente alla sua nascita.

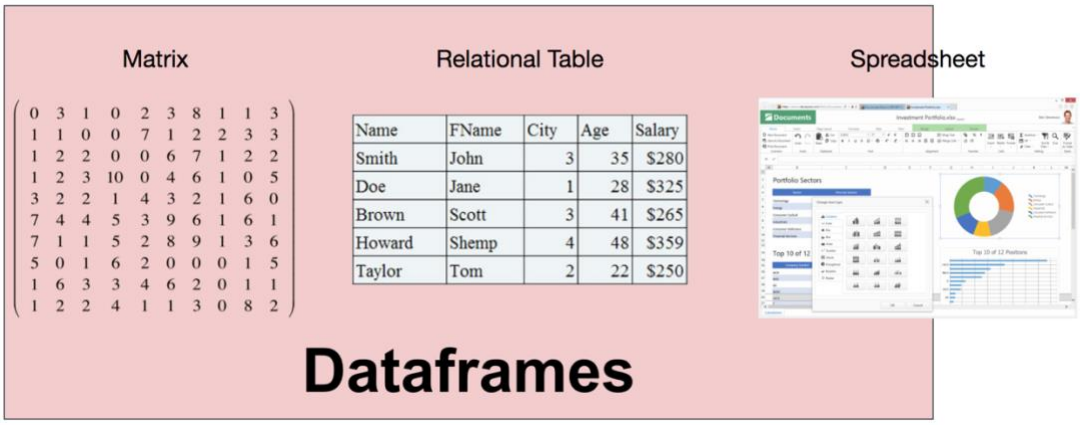
La prima definizione di DataFrame è emersa nel linguaggio di programmazione *S* presso i Bell Labs, nel libro "*Statistical Models in S* ".
Gli autori, Chambers e Hastie, descrivono una struttura simile a una tabella relazionale ma che conserva le proprietà di una matrice.
R, la versione open source di *S*, ha visto la sua prima versione stabile nel 2000 e da quest'ultimo ha ereditato i DataFrame.
Nel 2009, Pandas è stato rilasciato per portare la semantica dei frame di dati R in Python.
Tutte queste implementazioni dei DataFrame provenivano da un'unica fonte, ereditando la stessa semantica e lo stesso modello di dati.

Figura 1

2.1 Il modello di dati del DataFrame

I DataFrame sono nati dalla necessità di trattare i dati sia come una matrice che come una tabella.
Le matrici di tipo singolo sono troppo restrittive, mentre le tabelle relazionali richiedono che i dati siano definiti prima come schema.

In un DataFrame, il tipo di una colonna può essere dedotto in fase di esecuzione e non è necessario che sia noto in anticipo, né tutte le righe in una colonna devono essere dello stesso tipo.
I DataFrame sono effettivamente una combinazione di sistemi relazionali, matrici e, per estensione, fogli di calcolo.



Name	FName	City	Age	Salary
Smith	John	3	35	$280
Doe	Jane	1	28	$325
Brown	Scott	3	41	$265
Howard	Shemp	4	48	$359
Taylor	Tom	2	22	$250

Figura 2: I DataFrame supportano operatori di algebra lineare (matrici), algebra relazionale (tabelle) e alcune formule di fogli di calcolo.
Fonte dell'immagine: (2)

I DataFrame hanno diverse caratteristiche che li rendono una scelta vantaggiosa per l'esplorazione dei dati:

- un modello dati intuitivo che tratta righe e colonne in modo simmetrico;
- un linguaggio di query che supporta diverse modalità di analisi dei dati, inclusi operatori relazionali (ad esempio Filtro, join), algebra lineare (ad esempio Trasposizione) e simili a fogli di calcolo (ad esempio Pivot);
- una sintassi di query componibile in modo incrementale che permette di verificare il loro output in una fase preliminare, che permette inoltre la composizione di query più complesse;
- un sistema che consente all'interno di una colonna di avere diversi tipi di dati, per esempio, una stringa in una colonna di int;
- incorporamento nativo in una lingua ospitante come Python, con una semantica imperativa familiare.

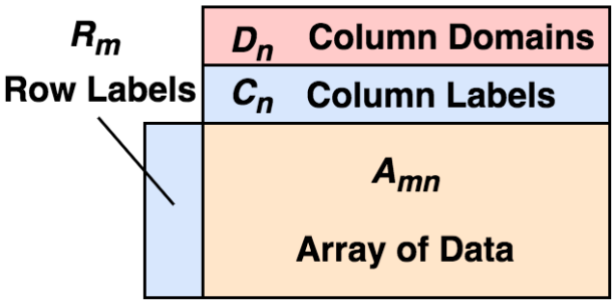


Figura 3: DataFrame Data Model
Fonte dello Schema: (2)

I DataFrame sono costituiti da una matrice bidimensionale con dati di tipo misto, una serie di etichette di riga, una serie di etichette di colonna e tipi (domini) per ciascuna colonna.
I tipi sono facoltativi in base alla colonna, ma il sistema può comunque richiedere un tipo specifico su una colonna in fase di esecuzione se necessario.
Utilizzando una "*lazy evaluation*[1]" si riesce a trattare il DataFrame come una tabella relazionale, senza sacrificare le altre proprietà che rendono un DataFrame simile ad una struttura matriciale.

2.1.1 Confronto con matrici

Tutte le matrici possono essere rappresentate come DataFrame (con etichette nulle). Tuttavia, non tutti i DataFrame possono essere matrici, anche se rimuoviamo le loro etichette.
Le matrici hanno una struttura omogenea all'interno del loro schema, mentre i DataFrame consentono schemi con più tipi.

In casi speciali, un DataFrame senza le sue etichette è una matrice, ma tutti i dati devono essere di tipo omogeneo e appartenere ai tipi *int* o *float* o qualche altro tipo che soddisfi la definizione algebrica di un campo.
In questi casi si parla di *DataFrame a matrice,* utilizzati ad esempio per mettere in relazione DataFrame e le pipeline di machine learning.

[1] Lazy evaluation: tecnica che consiste nel ritardare una computazione finché il risultato non è richiesto effettivamente.

2.1.2 Confronto con Tabelle relazionali

All’interno di un database di tipo relazionale ci sono molte possibili istanze di una relazione, cioè insiemi di tuple che soddisfano lo schema. Un'istanza può essere pensata come una tabella relazionale fissa.
I DataFrame sono qualcosa di simile alle istanze di relazione: rappresentano un insieme fisso di dati.
Tuttavia, il loro schema può essere non specificato e quindi indotto in base al loro contenuto da una funzione di induzione dello schema.
Questa flessibilità è fondamentale per i DataFrame, per via della gestione di grandi quantità di dati e aiuta ad evitare errori di “tipo” a runtime, come fatto da R e Python.

2.2 L’importanza di una corretta definizione di DataFrame

L'approccio che viene adottato da molti sistemi per aumentare la scalabilità dei DataFrame consiste nel rimuovere le proprietà difficili da rendere scalabili o equipararli a delle tabelle relazionali.
Questo comportamento ha l'effetto collaterale di ridefinire cosa sia effettivamente un DataFrame.
Tuttavia, poiché sempre più sistemi continuano a essere etichettati come *"DataFrame systems"*, si rischia di perdere una parte importante del flusso di lavoro del *data scientist.*
Il termine "DataFrame", come detto in precedenza, rischia di perdere il suo significato, vista la scorretta associazione del termine alle tabelle relazionali, rendendo ancora più difficile per gli utenti discernere l’utilità di una risorsa piuttosto che dell’altra.

Come evidenziato all’inizio del capitolo, si dovrebbe evitare di modificare la definizione del DataFrame, ma piuttosto cercare di comprenderla nella sua interezza ed eventualmente ridimensionarne la semantica, senza modificarla, mantenendo così una definizione formale che preserva le proprietà dei DataFrame e la semantica di S, R e Pandas, poiché le loro proprietà sono ciò che rende i DataFrame utili e unici.

3. Pandas

Pandas è una popolare libreria software open source scritta per il linguaggio di programmazione Python, utilizzata per la manipolazione e l'analisi dei dati. In particolare, offre strutture dati quali i DataFrame e fornisce delle operazioni per una loro gestione.
Le scelte progettuali fatte durante la definizione dei DataFrame, gli hanno permesso di guadagnare popolarità nel campo dell'EDA (Exploratory Data Analysis).
Per questa struttura dati, avere un incorporamento nativo in una lingua ospitante come Python, ha fatto si che la popolarità di quest'ultimo fosse merito anche del successo riscosso da Pandas, che supportando l'astrazione dei DataFrame ne permette una più facile esplorazione dei dati. Di fatto Pandas è stato scaricato oltre 300 milioni di volte, a partire del 2020 .

3.1 Panoramica su Pandas

Pandas, ideato da Wes McKinney, è un pacchetto Python che fornisce strutture dati veloci, flessibili e progettate per rendere il lavoro con dati "relazionali" o "etichettati" facile ed intuitivo.
Mira ad essere l'elemento fondamentale di alto livello per eseguire analisi pratiche e reali dei dati in Python.
Inoltre, ha un obiettivo ben più ampio, ovvero quello di diventare lo strumento di analisi e manipolazione dei dati open source più potente e flessibile. Dai report raccolti nel tempo è possibile affermare che sia già sulla buona strada.

Pandas è adatto per diversi tipi di dati, come:

- *Tabular data* con colonne di tipo eterogeneo, come in una tabella SQL o in un foglio di calcolo Excel;
- *Time series data,* ordinati e non ordinati (non necessariamente a frequenza fissa);
- *Arbitrary matrix data* (tipizzati in modo omogeneo o eterogenei) con etichette di riga e di colonna;
- Qualsiasi altra forma di set di dati di natura statistica o derivati da osservazioni.

I dati non hanno bisogno di essere etichettati per essere inseriti in una struttura dati Pandas.

Le due strutture dati primarie di Pandas sono: *Series* (monodimensionali) e *DataFrame* (bidimensionali), gestiscono la maggioranza dei casi d'uso tipici in finanza, statistica, scienze sociali e molte aree dell'ingegneria.
La struttura *DataFrame* di Pandas incorpora tutte le funzioni già distribuite dalla versione precedente contenuta in R (*data.frame*) e molte altre ancora.
Pandas è costruito su *NumPy* ed è pensato per integrarsi bene in un ambiente di elaborazione scientifica, con molte altre librerie fornite da terze parti.

NumPy è uno strumento open source di calcolo numerico che offre funzioni matematiche complete, algebra lineare, trasformate di Fourier e molto altro ancora.
Ad oggi i concetti di vettorizzazione e indicizzazione e trasmissione di *NumPy* sono gli standard de facto dell'elaborazione di array, grazie al codice C ben ottimizzato e alla flessibilità di Python.

3.2 Punti di forza di Pandas

Verranno elencati dei principi che hanno colmato le carenze in cui ci si poteva imbattere nell'utilizzo di altri linguaggi o ambienti di elaborazione di grandi quantità di dati e di ricerca scientifica:

- Facile gestione dei dati mancanti (rappresentati come NaN) nei dati in virgola mobile e non;
- Modifica delle dimensioni: le colonne possono essere inserite ed eliminate dai *DataFrame*;
- Automatico ed esplicito allineamento dei dati: i dati possono essere esplicitamente allineati ad una serie di etichette, oppure l'utente può anche non specificarle e lasciare alle strutture (*Series*, *DataFrame*, ecc..) l'onere di allineare automaticamente i dati;
- Il *group by* ha funzionalità più flessibili e performanti, permettendo di effettuare operazioni di divisione e combinazione tra *dataset* sia in fase di aggregazione o trasformazione dei dati;
- *Merging* e *joining* più intuitivo sui *dataset;*

- Un *reshaping* e *pivoting* più flessibile sui *dataset;*
- *Intelligent label-based slicing, fancy indexing, e subsetting* per *dataset* di dimensioni più elevate;
- Strumenti che supportano IO per il caricamento di dati da *lat files* (CSV), file Excel, databases e salvataggio o caricamento di dati dal formato ultraveloce *HDF5 format*.

Per i data scientist, l'elaborazione dei dati viene tipicamente suddivisa in più fasi: munging [2] e pulizia dei dati, analisi e modellazione di essi, allo scopo di organizzare i risultati dell'analisi in una forma adatta per la rappresentazione grafica o tabellare.
Per cui questo processo rende ideale l'utilizzo di uno strumento come Pandas.

3.3 L'evoluzione del progetto di Pandas su Ray

In una prima fase di valutazione e ricerca, si è deciso di implementare in Pandas su Ray le operazioni usate con una maggiore frequenza da parte degli utenti, per non incorrere nel rischio di avere un API con prestazioni facili ma con effettiva richiesta da parte degli utenti bassa.
Risalendo dai Notebook più votati dagli utenti *Kaggle*[3], si sono evidenziati i metodi per *DataFrame* più utilizzati.

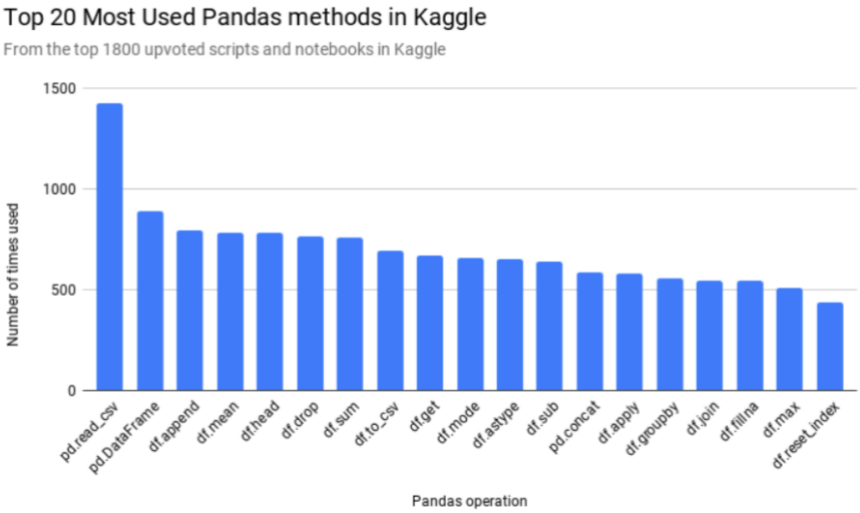


Figura 4: I 20 metodi più utilizzati in Pandas. Fonte del grafico: (3).

[2] Munging: processo di trasformazione e mappatura dei dati da un modulo dati " grezzo " in un altro formato con l'intento di renderlo più appropriato e prezioso per una varietà di scopi a valle come l'analisi.
3 Kaggle: comunità online di data scientist e professionisti dell'apprendimento automatico.

Per questa nuova implementazione di Pandas più leggera si è deciso di fare affidamento ad un backend che fosse totalmente compatibile con l'API, il suo nome è Ray.
Ray è un framework open source che fornisce un'API semplice ed universale per la creazione di applicazioni distribuite ed offre come grande vantaggio quello di disporre di un'elevata velocità computazionale.
Attualmente, Pandas su Ray supporta circa 180 metodi Pandas DataFrame; l'intera implementazione è stata guidata quindi dalla scelta degli utenti e non da pregiudizi da parte degli sviluppatori.

La chave del proggetto era quella di creare un'API, partendo da quella di Pandas, scalabile in base alla disponibilità dell'hardware, di dimensioni più piccole rispetto all'originale.
Il voler mantenere una retrocompatibilità con Pandas era dovuto al fatto che quest'ultimo sia, ad oggi, una delle API maggiormente utilizzate in ambito di *Data Science* e in molti corsi universitari.
Molti Data Scientist hanno più anni di esperienza su Pandas, per cui si vuole garantire che con questo progetto la loro esperienza e conoscenza non vengano sprecate. Infatti, l'apprendimento di una nuova API richiede tempo, così come l'apprendimento di un nuovo framework. Tali costi non devono essere trascurati quando si crea un'API o si sceglie uno strumento migliore per il proprio flusso di lavoro.

3.3.1 Pandas su Ray: Lazy Evaluation o *Eager* Evaluation?

In campo di *Data Science* si parla di iterazione con i dataset, ovvero visualizzare e raccogliere informazioni dai dati. Ciò vuol dire che il processo dietro questa attività richiede una gestione ben accurata dei blocchi all'interno dei notebook.
L'osservazione appena fatta va in contrasto con la modalità con la quale questi sistemi sono progettati, ovvero in modalità *lazy execution*.
Il sistema viene infatti ottimizzato per ridurre al minimo i cicli di CPU necessari per eseguire una determinata query o comando.
Questo tipo di esecuzione lavora molto bene se si sta parlando di elaborazione dei dati in batch, tuttavia la vera disconnessione si ha in flussi di lavoro che richiedono interazione con l'utente.

In un *eager system*, mentre l'utente sta digitando un'istruzione, la precedente viene eseguita, in contrasto a quanto succede invece in una *lazy evaluation*, ovvero non viene eseguito nulla finchè non si necessita di un output.

In un flusso di lavoro interattivo, tale *evaluation* fa si che la CPU non lavori perché non si necessita di risultati da parte di una query, portando così ad uno scarso utilizzo di tale risorsa per l'intera durata del flusso di lavoro.

In aggiunta a tale problema:

- *nei lazy systems* è difficile da eseguire il debug, in quanto l'errore può verificarsi in una qualsiasi delle istruzioni che vengono valutate in modo *"lazily";*
- i *lazy systems* sono di difficile comprensione, soprattutto poiché la maggior parte degli sviluppatori è abituata a *eager-execution systems*.

Alla luce delle differenze sul funzionamento tra un *lazy* e un eager *system*, si decide di adottare una soluzione che renda Pandas su Ray un sistema che possa passare da un'esecuzione di tipo *eager* a una di tipo *lazy*, beneficiando così di entrambi gli aspetti positivi dei sistemi, ovvero, ottenere una maggiore scalabilità e i relativi vantaggi da parte della CPU, evitando l'utilizzo inefficiente delle risorse e la scarsa capacità di debug fornita per flussi di lavoro interattivi di piccole e medie dimensioni.

4. Modin

Il progetto denominato “Pandas su Ray” prende nome di Modin.
Nonostante Pandas garantisca in generale elevate performance, quando si lavora con grandi quantità di dati l’utilizzo di un singolo core diventa insufficiente, per cui gli utenti devono ricorrere a dei sistemi distribuiti per aumentare le proprie prestazioni.
Il compromesso per poter avere prestazioni migliori, tuttavia, comporta una curva di apprendimento ripida.
Ciò che viene richiesto dagli utenti è che i loro programmi funzionino più velocemente e che scalino meglio senza modifiche significative al codice, utilizzando lo stesso script per un set di dati da 10 KB o da 10 TB.

Figura 5: Logo Modin

Modin è un progetto in fase iniziale presso i *RISELab* dell’università di Berkley, completamente open-source e reperibile su *GitHub*.
A differenza di altri sistemi DataFrame paralleli, Modin è un DataFrame estremamente leggero e robusto e proprio grazie a questa sua leggerezza promette di offrire una velocità fino a 4 volte superiore su un laptop con 4 core fisici, richiedendo agli utenti di modificare solo una singola riga di codice nei loro notebook.
Modin, abilita anche un supporto *out of core* (ancora sperimentale), ovvero se si lavora con file molto grandi o se si desidera superare la memoria utilizzabile, è possibile cambiare la posizione principale del DataFrame, utilizzando il disco come overflow per la memoria.

4.1 Come Modin accelera il flusso di lavoro

Prendendo in considerazione un laptop moderno a 4 core e un DataFrame da analizzare, Pandas lo fa utilizzando solo uno dei core della CPU, mentre Modin è in grado di usarli tutti.

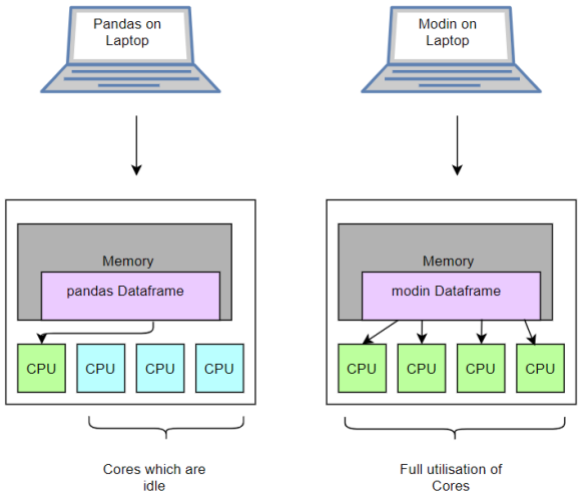


Figura 6: Utilizzo core Pandas vs Modin
Fonte dello Schema: (4)

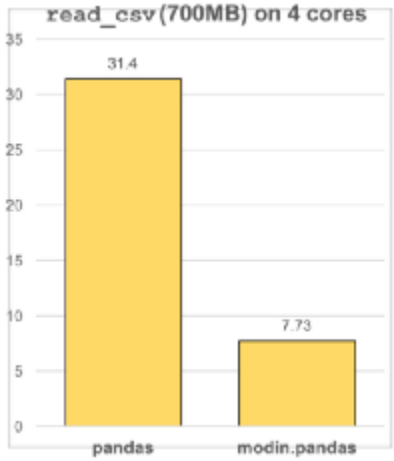


Figura 7: Performance della 'read_csv' su un DataFrame di 700MB, usando 4 cores.
Fonte del Grafico: (4)

Prendendo in considerazione alcuni report presenti nella documentazione di Modin, avendo a disposizione una macchina di dimensioni ben più grandi rispetto al caso riportato precedentemente, per esempio con 144 cores, il divario tra le prestazioni delle API diventa molto più evidente.

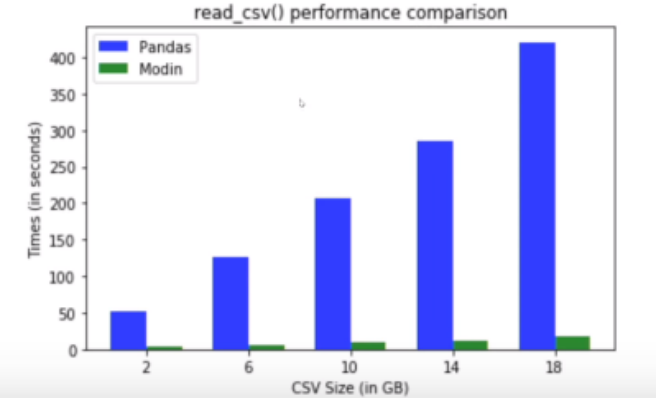


Figura 6: Performance della 'read_csv' di una grande quantità di dati, usando 144 cores
Fonte del Grafico: (4)

4.2 Architettura di Modin

Un DataFrame viene partizionato e, nel caso di Modin, esso viene diviso sia per colonne che per righe, offrendo così flessibilità e scalabilità per entrambe le componenti.

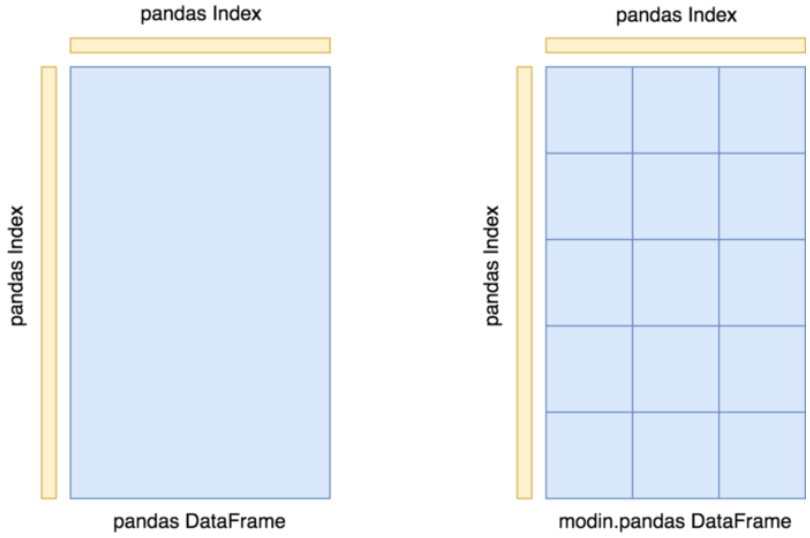


Figura 9: Partizionamento Pandas vs Partizionamento Modin. Fonte (12)

Modin è diviso in più livelli:

1. L'API è esposta al livello più alto
2. Il livello che ospita il compilatore di query che riceve le query dal livello API ed esegue determinate ottimizzazioni.
3. All'ultimo livello si trova il gestore delle partizioni ed è responsabile del layout dei dati e del mescolamento, del partizionamento e della serializzazione delle attività che vengono inviate a ciascuna partizione.
4. Ogni partizione mantiene poi una parte dell'intero dataset.

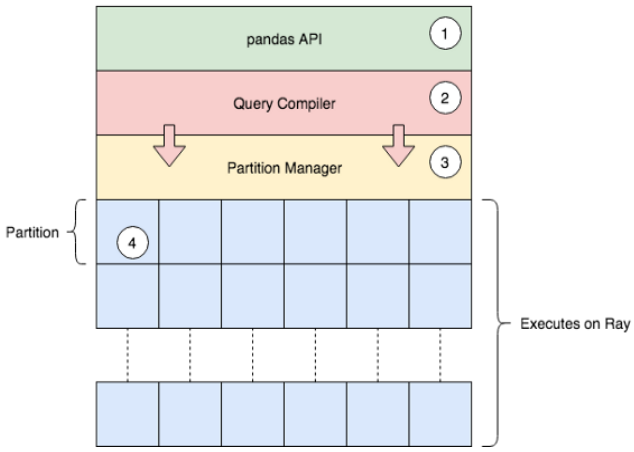


Figura 9:Architettura generica Modin Fonte (12)

Avendo dato precedenza nell'implementazione a funzioni più popolari all'interno di Pandas, alcune funzioni che non sono implementate in Modin, restano comunque eseguibili in Pandas. Infatti, nel caso in cui si tenti di utilizzare una funzione non fornita da Modin, la libreria utilizza l'implementazione predefinita di tale funzione in Pandas, convertendo il frame di dati Modin in frame di dati Pandas, eseguendo le operazioni e infine riconvertendo nel frame di dati Modin.

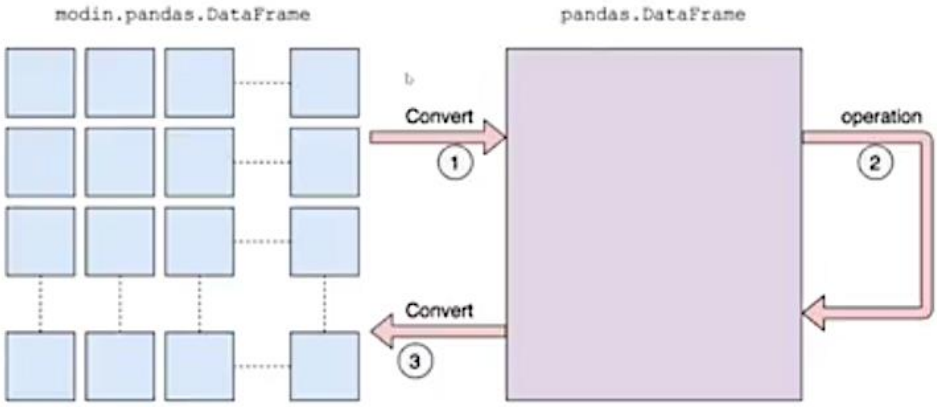


Figura 10: Come viene eseguita un' operazione non predefinita
Fonte: (4)

Rimane comunque vantaggioso utilizzare Modin in sostituzione a Pandas, poiché parallelizza l'implementazione e riduce il tempo di calcolo, anche nel caso in cui si debba ricorrere ad alcune funzionalità definite solo in Pandas.

Modin è stato progettato per funzionare su una varietà di sistemi, in modo che gli utenti possano spostare il notebook sul quale si sta lavorando su ambienti diversi senza avere problemi nell'esecuzione.

L'architettura di Modin è modulare, per cui si è in grado di aggiungere diversi motori di esecuzione o utilizzare le risorse hardware in modo differente. Supporta l'esecuzione sul motore di calcolo fornito da Dask (analizzato nel prossimo capitolo), oltre che su quello di Ray, per parallelizzare il più possibile l'API Pandas.

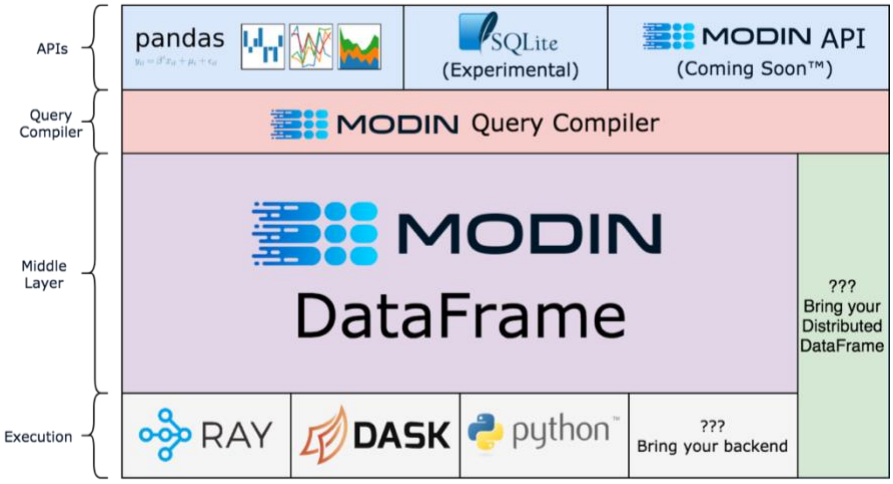


Figura 11: Architettura Modin
Fonte: (12)

A lungo termine, Modin si pone come obiettivo quello di diventare una libreria DataFrame che supporta le API più popolari (SQL, Pandas, ecc..).

5. Dask DataFrame

La libreria Dask ha un'API simile a quella di Pandas e Numpy ed ha il compito di accelerare il flusso di lavoro sfruttando il parallelismo.

Un DataFrame Dask è partizionato per riga e ogni blocco contiene un frame di dati Pandas, raggruppando poi le righe in base all'indice per aumentarne l'efficienza.

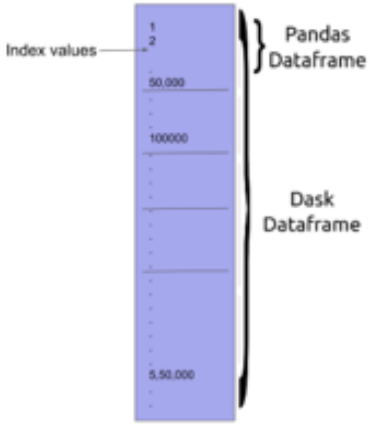


Figura 12:Architettura Dask DataFrame
Fonte: (6)

Facendo questa suddivisione in frame più piccoli, i calcoli vengono eseguiti in parallelo. Dask utilizza la mappatura della memoria, quindi non carica tutti i dati in una volta ma punta solo alla posizione dei dati in memoria.

5.1 Dask DataFrame vs Pandas

All'interno di Dask non vengono fornite molte delle funzioni native di Pandas, perché questo framework, al contrario di Modin, non punta a diventare un suo sostituto. Inoltre, molte delle funzionalità non possono essere implementate in modo efficiente o non possono essere implementate affatto a causa dei contrasti con l'architettura di Dask.

L'API di Dask DataFrame, in aggiunta, è di tipo *lazy* e ciò rende l'utilizzo meno conveniente sotto alcuni punti di vista, ma consente a Dask di eseguire ottimizzazioni sulle query, che ne determinano l'aumento di velocità.

Dask DataFrame viene utilizzato in situazioni in cui sarebbe necessario Pandas, ma quest'ultimo porterebbe ad un fallimento a causa dell'elevata dimensione dei dati o della scarsa velocità di calcolo. Dask ad esempio è utile nei casi in cui si vuole:

- Manipolare set di dati di grandi dimensioni, anche quando non si adattano alla memoria;
- Accelerare calcoli dispendiosi utilizzando il multi-core;
- Fare un'elaborazione distribuita su set di dati di grandi dimensioni, con operazioni come: calcoli di groupby, join..

Dask DataFrame invece potrebbe non essere la scelta migliore nel caso in cui il set di dati in esame si adatti perfettamente alla RAM che si ha a disposizione, facendo quindi rimanere Pandas l'opzione più valida, poiché potrebbero esserci scelte di progettazioni più semplici per migliorare il proprio flusso di lavoro rispetto al parallelismo.

Come si può notare nel grafico sottostante, preso dallo studio svolto nella fonte (6), se si deve lavorare con delle quantità di dati di ragionevole dimensione, il parallelismo di Dask non aiuta ad incrementare la velocità computazionale. Non è così, invece, per dataset di elevate dimensioni. In questo caso, l'unico modo per poter avere dei risultati dalle query è utilizzare Dask piuttosto che Pandas.

Operations	Data with 0.6M Records		Data with 63M Records		Data with 200M Records	
	Pandas	Dask	Pandas	Dask	Pandas	Dask
Read	8s	12.3ms	40min	438ms	ME	3.58s
df.head()	1.01ms	2.56s	20ms	349ms	ME	481ms
df.shape()	0ns	6.36s	49s	13.5s	ME	5m 4s
df.describe()	78.6ms	8.59s	1m 39s	25.4s	ME	3m 56s
df[col].value_counts()	75.3ms	6.45s	34s	20.3s	ME	3m 54s
df.groupby(col).mean()	253ms	6.64s	9m 5s	22.7s	ME	4m 4s
df.groupby(col1, col2).mean()	901ms	12.1s	56.5s	28.7s	ME	4m 24s
df[col].mean()	2.53s	6s	4m 11s	50s	ME	4m
df[text'].apply(count_words).mean()	14s	27.9s	10m 6s	4m 1s	ME	42m 13s
1D Plot	1.08s	7.14s	TLE	TLE	ME	TLE
10^{th} Percentile	8.95ms	6.64s	3.1s	47.5s	ME	4m 9s
Search on a Condition	39.3ms	2.92s	1m 49s	52.1s	ME	5m 30s
Filter on a Condition	51.9ms	2.64s	1m 44s	51.5s	ME	4m 6s
Save to disk as CSV file	5.44s	11.4s	33.8s	52.1s	ME	52s

Figura 13: Benchmark Pandas vs Dask
Fonte dei dati: (6)

5.2 Dask DataFrame vs Modin

Il partizionamento effettuato da Modin è molto più flessibile rispetto a quello fornito da Dask, poiché il sistema può scalare in entrambe le direzioni e avere un partizionamento più fine, al contrario del secondo che fornisce partizionamento unicamente per le righe.

Avendo un controllo più dettagliato sul partizionamento, è possibile supportare una serie di operazioni più impegnative (ad esempio, trasposizione, mediana e quantile), che nel caso di Dask DataFrame sarebbe stato impossibile sviluppare, poiché quest'ultimo tratta le operazioni sui dati come operazioni di MapReduce[4].
Questa flessibilità nel partizionamento permette inoltre a Modin di apportare dei miglioramenti nell'utilizzo dell'intero cluster.

5.3 Dask DataFrame vs Ray

Dask e Ray hanno come obiettivo comune quello di rendere più semplice l'esecuzione di codice Python in parallelo su cluster di macchine.

Le differenze tra questi due framework sono:

- Dask utilizza uno scheduler centralizzato per distribuire il lavoro su più core, mentre Ray utilizza una pianificazione di tipo bottom-up distribuita, per avere un elevato throughput delle attività (arrivando anche a poter eseguire milioni di attività al secondo);
- Ray si concentra molto sulla latenza, ottenendo così una latenza circa 30 volte inferiore rispetto a Dask;
- Dask è disponibile sia per Linux, macOs e Windows. Al contrario, Ray, viene reso disponibile solo per i primi due.

Le differenze tra i due backend per Modin sono:

- Modin su Ray: quest'ultimo mira a raggiungere una piena compatibilità con Pandas, quindi un grande vantaggio è sicuramente quello della sintassi. E' possibile infatti mantenere invariato il contenuto dei notebook Pandas, ad eccezione di una sola riga (`*import pandas as pd*` diventerà ` *import modin.pandas as pd*`).

[4] MapReduce= suddividere un'operazione di calcolo in diverse parti, per poi processarle in modo autonomo.

- Modin su Dask: essendo un progetto ben più grande del precedente, non ingloba tutte le funzionalità distribuite da Pandas, bensì fornisce *Dask.DataFrame*: una struttura di livello superiore rispetto a quella fornita da Pandas che aiuta l'utente a gestire *l'out of core*.

6. Benchmark

Dopo aver dato una visione teorica di entrambe le API e dei backend che utilizza Modin, verranno riportati i benchmark che mettono in correlazione i tempi di esecuzione di alcune query tra quelle più comunemente utilizzate da parte degli utenti

Sono state prese in esame 3 macchine diverse e 3 diversi sistemi operativi per testare i gli aspetti principali di Modin, ovvero la ripartizione del flusso di carico sui core e la gestione da parte dei due backend.

Nome macchina 1	Dell Inspiron 15 7501 2020
Sistema Operativo	Windows 10
Processore	2.6 GHz Intel Core i7-10750H hexa-core
Memoria RAM	16 GB, DDR4, 2.933 MHz
Backend testati	Dask

Nome macchina 2	Asus Vivobook Pro 2018	
Sistema Operativo	Windows 10	Linux 20.04
Processore	2.8 GHz Intel Core i7-7700 quad-core	
Memoria RAM	16 GB, DDR4, 2.400 MHz	
Backend testati	Dask	Ray

Nome macchina 3	MacBook Air 2015
Sistema Operativo	macOs Catalina
Processore	1,6 GHz Intel Core i5 dual-core
Memoria RAM	4 GB, DDR3, 1.600 MHz
Backend testati	Dask

In questo caso studio è stato utilizzato un dataset contenente tutte le corse dei taxi della linea gialla della città di New York nel gennaio 2020. Tale dataset è salvato in un file csv con una dimensione di circa 600 MB per un totale di più di 6 milioni di records al suo interno. Per manipolare il dataset verrà utilizzato JupyterLab, tool utile per lavorare in ambito di data science poichè facilita l'esplorazione e la visualizzazione dei dati. Il tool è incluso all'interno di Anaconda, software open source che distribuisce i principali pacchetti e librerie Python e R.

6.1 Query

Dopo aver fatto gli import necessari:

```
import pandas as pd #Per Modin sostituire con 'modin.pandas'
import matplotlib as mp
%load_ext autotime
```

Le librerie importate sono:
- quella di Pandas, da sostituire con quella di Modin all'occorrenza;
- *Matplotlib* è una libreria per la creazione di grafici, utilizzata anche per quelli presenti in questo documento.

Infine, per riportare e successivamente registrare il tempo di esecuzione di una cella, sarà necessario installare l'estensione *ipython-autotime* e una volta aperto il Notebook basterà semplicemente avviarla.

In seguito, sono state eseguite le seguenti query sui Notebook.

a. Read dataset

```
df=pd.read_csv("CorseNY.csv")
```

Legge un file con valori delimitati da virgole (csv) e crea una struttura DataFrame.

b. Describe

```
df.describe()
```

Genera statistiche descrittive che includono la moda, la dispersione e la forma della distribuzione di un set di dati, escludendo i valori NaN.

c. Value_counts

```
df['VendorID'].value_counts()
```

Restituisce una serie contenente conteggi di righe univoche nel DataFrame, in questo caso le righe che hanno il medesimo '*VendorID'*.

d. Pivot

```
df.pivot_table(index=['VendorID'],columns=['passenger_count'],
values='fare_amount')
```

Crea una tabella Pivot, come viene fatto su un foglio di calcolo, specificando i campi:

- *index*: valore per il quale raggruppare nella tabella di pivot. In questo caso verranno raggruppati in base al '*VendorID'*;
- columns: valore per il quale raggruppare per colonna nella tabella di pivot. In questo caso verranno raggruppati in base al '*passenger_count*;
- values: valori da addizionare. In questo caso i '*fare_amount'*.

e. Group by

```
df.groupby('passenger_count').agg({'extra': 'sum','fare_amount': 'mean','VendorID' : 'last'})
```

Utilizzato per raggruppare grandi quantità di dati ed effettuare delle operazioni di calcolo su questi gruppi.
Nella query si è raggruppato in base al '*passager_count*', inoltre l'opzione '*agg'* permette di aggregare le colonne utilizzando una o più operazioni specificate tra le parentesi.
In questo caso vengono sommati gli '*extra'*, viene preso il valore medio dei '*fare_amount'* e viene riporatato solo l'ultimo '*VendorID'* del gruppo esaminato dalla *group by*.

f. Sort Values

```
df.sort_values(by='extra',ascending=False)
```

Ordina il dataset in base ad uno dei valori specificati, di riga o colonna. In questo caso in base ai valori contenuti nella colonna '*extra',* in ordine decrescente.

g. Plot Scatter

```
df[df['trip_distance'] < 10] .sample(100).plot.scatter(x='trip_distance', y='fare_amount')
```

Plot scatter è un grafico a dispersione semplice nel quale i dati vengono rappresentati attraverso dei punti o cerchi, funzione fornita dalla libreria *Matplotlib*.
Nella query vengono presi in considerazione le *'trip_distance' < 10* e vengono messe in relazione sul grafico, dove sull'asse x si ha la '*trip_distance'* e sulla y il '*fare_amount*'.

h. Fillna

```
df.fillna(0)
```

Sostituisce ai valori NaN il valore specificato tra parentesi. In questo caso ‘0’.

i. Duplicate columns

```
df['Supplementi'] = df['improvement_surcharge']
```

Crea una nuova colonna ‘*Supplementi’* e copia al suo interno i valori contenuti nella colonna ‘*improvement_surcharge’*.

j. Delete columns

```
del df['improvement_surcharge']
```

Elimina la colonna ‘*improvement_surcharge’*.

k. Delete Empty Rows

```
df.dropna(subset=['VendorID'])
```

Elimina le righe che hanno valore NaN all’ interno del *VendorID*.

6.2 Tempi di Esecuzione

Prima macchina sistema operativo Windows, con backend Dask

N°	QUERY	PANDAS	MODIN
1	read_csv	10,524 s	6,161 s
2	describe	2,232 s	12,922 s
3	value_counts	0,078 s	0,703 s
4	pivot	0,406 s	1,051 s
5	groupby	0,188 s	7,624 s
6	sort_values	1,112 s	1,57 s
7	plot.scatter	1,214 s	4,487 s
8	fillna	0,860 s	0,860 s
9	duplicate	0,016 s	0,015 s
10	delete columns	0,157 s	0,047 s
11	delete empty rows	1,912 s	3,789 s

Tabella 1

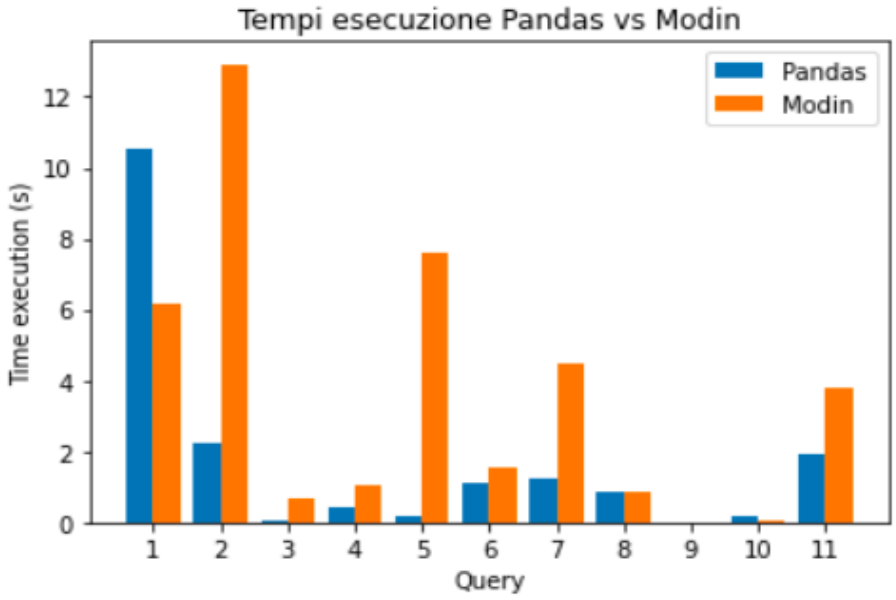


Grafico 1

Seconda macchina: sistema operativo Windows, con backend Dask

N°	QUERY	PANDAS	MODIN
1	read_csv	9.835 s	6,064 s
2	describe	4.663 s	26.273 s
3	value_counts	0.076 s	1.011 s
4	pivot	0.459 s	12.344 s
5	groupby	0.159 s	13.465 s
6	sort_values	1.179 s	64.342 s
7	plot.scatter	1.106 s	15.961 s
8	fillna	1.476 s	0.025 s
9	duplicate	0.015 s	0.016 s
10	delete columns	0.181 s	0,019 s
11	delete empty rows	1.898 s	6.584 s

Tabella 2

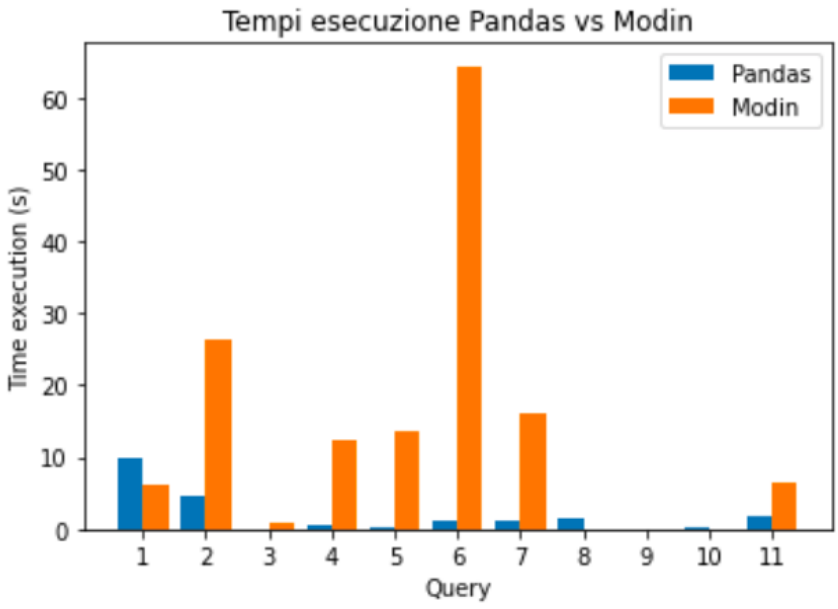


Grafico 2

Seconda macchina: sistema operativo Linux, con backend Ray.

N°	QUERY	PANDAS	MODIN
1	read_csv	9.434 s	7.731 s
2	describe	2.071 s	7.426 s
3	value_counts	0.055 s	1.438 s
4	pivot	0.322 s	1.462 s
5	groupby	0.148 s	4.678 s
6	sort_values	1.047 s	12.39 s
7	plot.scatter	0.900 s	4.807 s
8	fillna	1.137 s	3.281 s
9	duplicate	0.012 s	0.001 s
10	delete columns	0.139 s	0.001 s
11	delete empty rows	1.766 s	6.846 s

Tabella 3

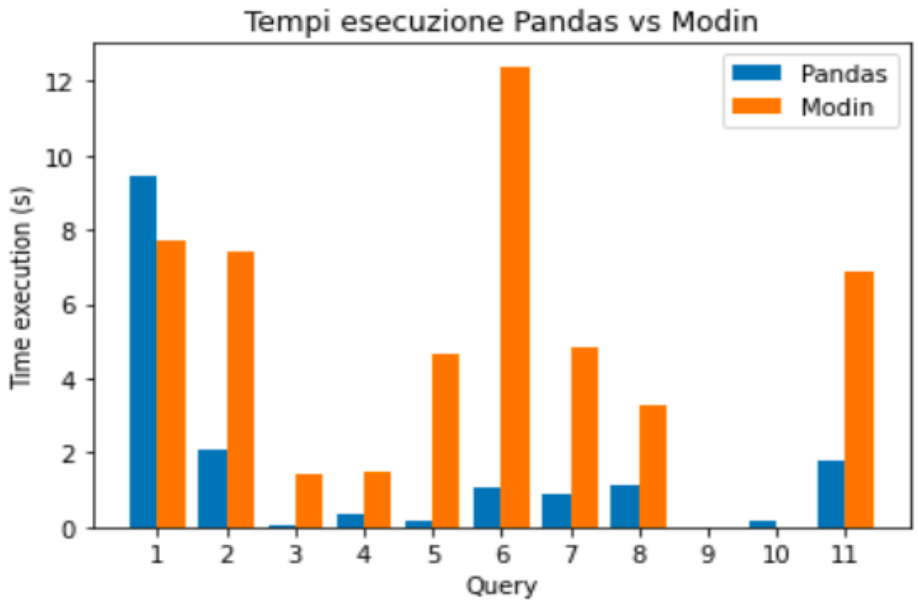


Grafico 3

Terza macchina: sistema operativo macOS, con backend Dask.

N°	QUERY	PANDAS	MODIN
1	read_csv	19.223 s	64.021 s
2	describe	5.959 s	-
3	value_counts	0.134 s	9.823 s
4	pivot	0.792 s	12.971 s
5	groupby	0.247 s	118,223 s
6	sort_values	3.598 s	-
7	plot.scatter	5.904 s	-
8	fillna	2.813 s	-
9	duplicate	0.901 s	-
10	delete columns	4.584 s	-
11	delete empty rows	43.227 s	-

Tabella 4

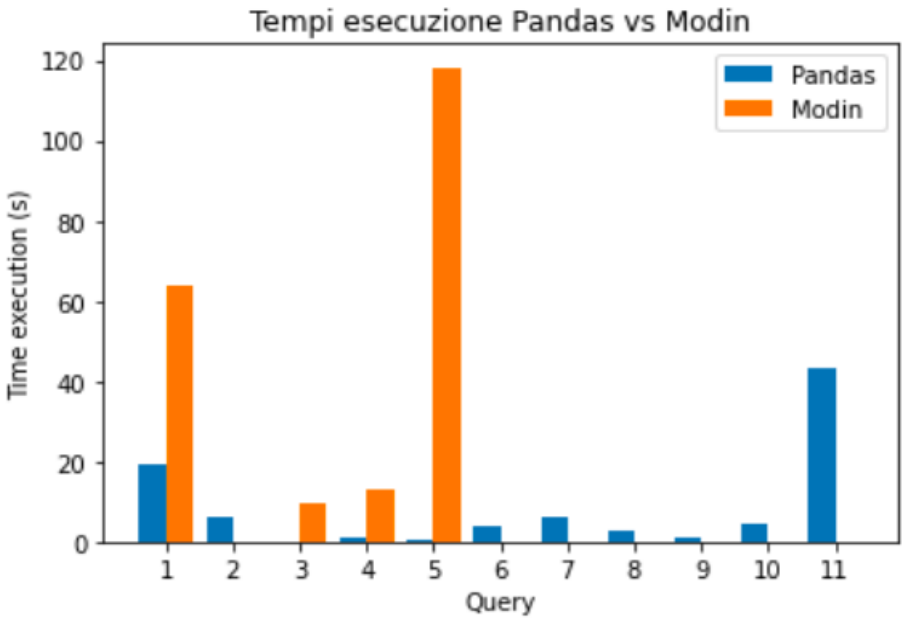


Grafico 4: Le query eseguite in Modin andate in time out non sono state riportate nel grafico

6.3 Considerazioni

Dai dati rilevati, Modin sembra non essere una valida scelta per velocizzare l'esecuzione di query all'interno di un DataFrame. Si nota infatti come siano veramente pochi i casi in cui superi la sua versione precedente e molti quelli in cui venga di gran lunga superato.
Inoltre, se si lavora con un hardware con meno di 4 core fisici, l'esecuzione delle query con Modin non riesce a giungere al termine o comunque non in tempi ragionevoli.
In delle macchine non ottimizzate la comunicazione tra i processi generati dall'esecuzione di query in Modin, non fa altro che creare un overhead di controllo, deleterio per l'esecuzione delle singole query.
Tutto questo è dovuto dal fatto che l'esecuzione di una query in Modin comporta la creazione di due processi per ogni core presente nella macchina, poiché avendo a che fare con dei processori Intel per via dell'*Hyper-Threading*, il sistema operativo riesce a vedere il doppio dei core fisici, e quindi a crea un processo per ogni core "logico".

Nome Processo	Byte inviati	Byte rice...	Pacchetti inviati	Pacchetti ricevuti	PID
python3.8	550,1 MB	89 KB	39.686	669	24021
python3.8	514,6 MB	311,5 MB	37.562	21.995	24023
python3.8	245,4 MB	245,6 MB	19.807	19.886	24001
python3.8	12,2 MB	225,9 MB	998	16.320	24167

Figura 14

La gestione dei processi generati da Modin può essere verificata all'interno di uno dei suoi file di configurazione, *distributed.yaml*, in particolare nella porzione di file qui riportata:

```
# Fractions of worker memory at which we take action to avoid memory blowup
# Set any of the lower three values to False to turn off the behavior entirely
memory:
  target: 0.60  # target fraction to stay below
  spill: 0.70  # fraction at which we spill to disk
  pause: 0.80  # fraction at which we pause worker threads
  terminate: 0.95  # fraction at which we terminate the worker
```

Figura 15.1: Porzione del file distributed.yaml

Il significato di tali parametri è:

- *target: 0.60*, percentuale di memoria utilizzabile da parte del worker;
- *spill: 0.70*, percentuale di memoria utilizzabile oltre alla quale si inizierà a scrivere su disco per liberare la RAM che occupa il worker;
- *pause: 0.80*, dopo questa percentuale di memoria occupata, il worker verrà messo in uno stato di sleep, perché la velocità con la quale si stanno generando i dati è maggiore di quella con cui si riesce a copiare i dati su disco e quindi si mette in attesa il processo fino a che non viene liberata la memoria RAM;

- *terminate: 0.95*, percentuale dopo la quale il worker verrà terminato.
- Un processo una volta messo in pausa può essere rianimato per un massimo di 3 volte. Anche questa specifica viene espressa all'interno del file di configurazione *distributed.yaml.*

```
scheduler:
  allowed-failures: 3     # number of retries before a task is considered bad
```

Figura 15.2: Porzione del file distributed.yaml

Quindi come detto in precedenza, eseguire Modin su un solo nodo può portare a dei problemi nell' utilizzo della memoria.

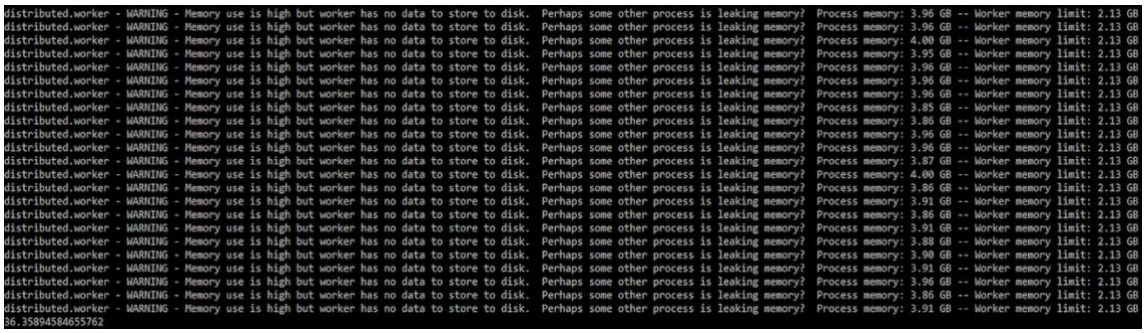


Figura 16: Schermata Prompt memory leak

Come si può osservare in figura, un processo creato da Modin sta rilevando molta più memoria allocata rispetto a quella effettivamente da lui utilizzata, per cui ipotizza che ci sta un altro processo che ha causato un memory leak.
Verificando tutti i processi attivi si nota che uno dei processi generato da Modin all'avvio dell'esecuzione della query è stato ucciso, poiché andato in stato di sleep dopo aver superato l'80% della memoria a lui dedicata e dopo 3 tentativi fallimentari di rianimazione è stato ucciso, non cancellando la memoria da lui allocata ma perdendo il riferimento a tale memoria, dando così vita al memory leak.

La query viene comunque portata a termine poiché un altro dei parametri definiti all'interno del file *distributed.yaml,* permette agli altri processi in esecuzione di "rubare" il lavoro assegnato agli altri assicurando che il lavoro sia portato a termine.

```
work-stealing: True     # workers should steal tasks from each other
```

Figura 15.3: Porzione del file distributed.yaml

I benchmark effettuati, oltre a riportare i tempi con i quali sono state svolte le query dalle due API, mettono in confronto anche i due backend nell'utilizzo di Modin. A parità di hardware andando a confrontare il secondo e il terzo benchmark, possiamo verificare uno dei principi alla base di Ray, ovvero una velocità computazionale maggiore rispetto a Dask.

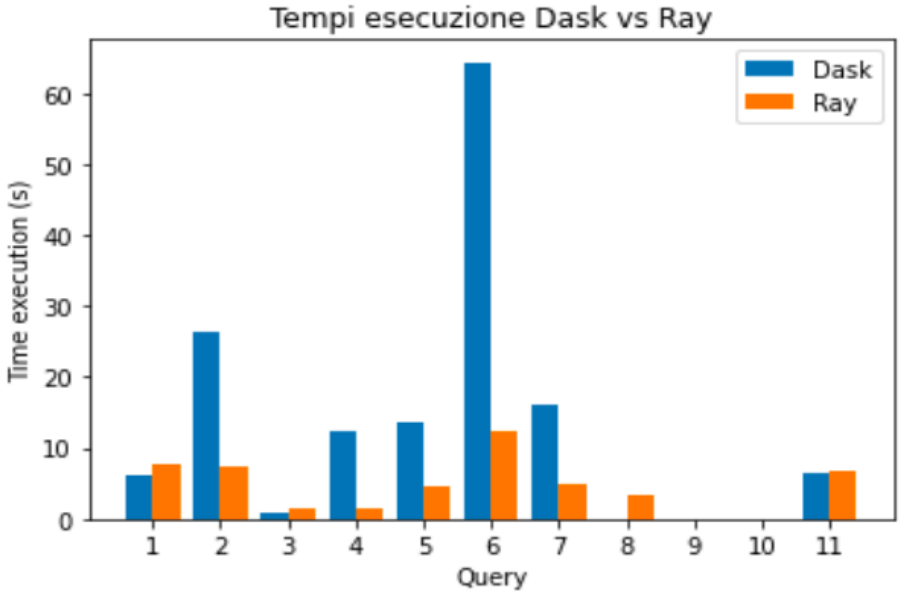


Grafico 5

Questo rappresenta sicuramente un grande vantaggio se si lavora su di un singolo nodo.

Ma se avessimo un dataset di dimensioni più elevate?
Sicuramente non si potrebbe più lavorare sfruttando le risorse hardware di una singola macchina, bensì bisognerebbe disporre di un cluster, nel quale ogni macchina presente può accedere al dataset e, in questo caso bisognerebbe scegliere lo scheduler centralizzato fornito da Dask, si potrebbe distribuire il lavoro su più hardware in parallelo all'interno del cluster.

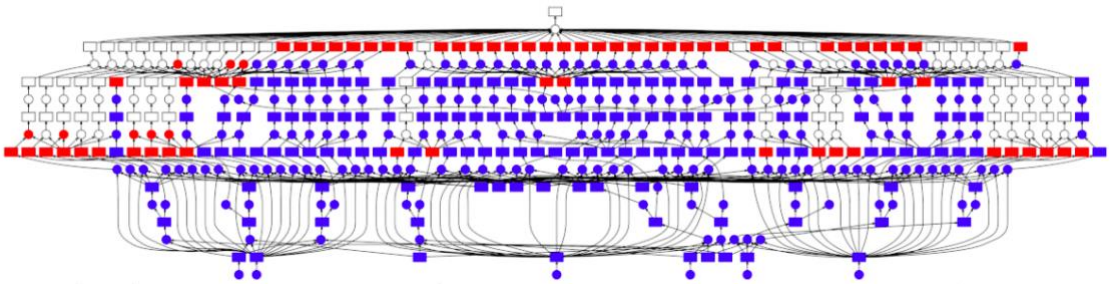
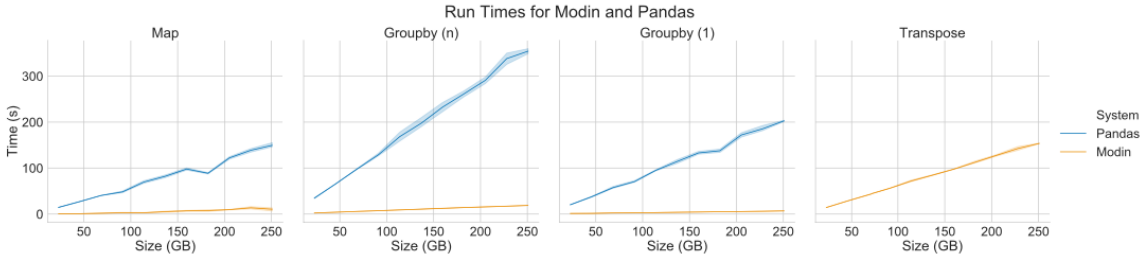


Figura 17: grafico di esempio utilizzando lo scheduler di tipo distribuito di Dask
Fonte dello schema: (6)

Analizzando i risultati ottenuti nello studio (1), in cui è stato utilizzato un dataset di dimensioni pari a 250 GB con più di 1,6 miliardi di righe ed effettuato dei benchmark su un EC2[5] (128 cores e 1952 GB di RAM), sono state eseguite le seguenti operazioni:

- map: controlla che ogni valore all'interno del DataFrame sia nullo e lo sostituisce con TRUE, altrimenti con FALSE se non lo è;
- group-by(*n*): raggruppa per le colonne non nulle dei "*passenger_count*" e conta il numero di righe presenti in ogni gruppo;
- group-by(*1*): conta il numero delle righe non nulle all'interno del DataFrame;
- transpose: inverte le colonne con le righe del DataFrame e applica una funzione di *map* sulle nuove righe.

Descriviamo ora nello specifico i seguenti risultati ottenuti dallo studio (1):



Grafici 6
Fonte dei dati e dei grafici: (6)

In particolare, si vuole far notare la differenza tra una *group by* eseguita su un solo gruppo e una eseguita su *n* gruppi, perché se si lavora con *n* gruppi, mescolare i dati e farli comunicare tra loro riduce particolarmente le performance, invece con la *group by(1)* non vi è alcun overhead dovuto dalla comunicazione tra i vari gruppi.

Con l'operazione di transpose, invece si è voluto dimostrare che Modin, a differenza di Pandas, riesce ad operare con dataset aventi miliardi di righe.
I grafici dimostrano inoltre che, per le operazioni *group-by(n)* e *group-by(1)*, Modin ottiene una velocità rispettivamente fino a 19 e 30 volte superiore rispetto a Pandas.

5 EC2: è una parte centrale della piattaforma di cloud computing Amazon Web Services (AWS) di Amazon che permette agli utenti di affittare computer virtuali sui quali eseguire le loro applicazioni.

Ad esempio, un *group-by(n)* su un DataFrame da 250 GB, in Pandas impiega circa 359 secondi mentre su Modin ne impiega solo 18,5, quindi una velocità superiore di 19 volte. Per l'operazione di *map*, Modin è circa 12 volte più veloce di Pandas.

Come detto in precedenza, questi miglioramenti in termini di prestazioni derivano dalla parallelizzazione delle operazioni all'interno di Modin, in contrapposizione a Pandas che utilizza solo un singolo core.
Durante l'operazione di *transpose*, Pandas non è stato in grado di trasporre nemmeno un DataFrame più piccolo di 20 GB, dopo 2 ore. È stato poi dimostrato attraverso dei test separati al caso studio, che Pandas può trasporre solo dataset fino a 6 GB, prendendo in considerazione lo stesso hardware utilizzato per i test.

7. Conclusioni

Nel presente elaborato è stata svolta un'analisi delle architetture delle due API Pandas e Modin e successivamente un'attività di confronto tra i tempi di esecuzione di alcune query. Pandas rappresenta ad oggi la scelta migliore e la più utilizzata da parte degli utenti in ambito di data analysis. Mentre la seconda nasce con lo scopo unico di sopperire al problema principale di Pandas, ovvero la scarsa scalabilità se si lavora con grandi quantità di dati.

Il progetto di Modin è stato analizzato nella sua interezza, ovvero: la sua architettura, la gestione dei task in modalità multicore, i due backend che lo ospitano, la sintassi ed infine sono stati prodotti dei report riguardanti l'esecuzione di query in contrapposizione a quelli prodotti in Pandas.

Vista la stretta correlazione prestazionale di Modin al numero di processori disponibili e i backend sui quali viene distribuito, si è reso necessario dover eseguire i notebook su diverse macchine al fine di ottenere dei risultati che prevedessero ogni casistica nel quale si possa trovare l'API.

Dai dati emersi dai benchmark effettuati non è stato possibile validare quanto promesso dal progetto di Modin, ovvero avere una velocità computazionale superiore rispetto a Pandas anche su di un singolo nodo. Per cui si è arrivati alla conclusione che scegliere Modin in sostituzione a Pandas in scenari che prevedono una manipolazione di quantità di dati non elevata, non comporta dei vantaggi, bensì provoca un utilizzo della CPU eccessivo e non proporzionale al lavoro richiesto, causando in alcuni casi un overhead tra i processi comunicanti provocando quindi un calo prestazionale rispetto a Pandas e nei casi peggiori una mancata esecuzione della query.

Si è reso necessario illustrare con particolare attenzione uno dei due backend, ovvero quello di Dask, per via della presenza di uno scheduler centralizzato che riesce a distribuire il lavoro su più core all'interno di un cluster, risolvendo il problema della comunicazione non efficiente tra i vari processi assegnati al task.

Secondo quanto emerso dagli studi (1), all'interno di un sistema distribuito nel quale più macchine collaborano insieme per manipolare grandi quantità di dati, scegliere l'implementazione fornita da Modin, grazie all'aiuto dello scheduler di Dask, può

portare ad uno speed-up significativo e in alcuni scenari diviene addirittura necessario l'utilizzo di Modin per portare a termine l'esecuzione dei task.

Questa seconda casistica è ancora in fase sperimentale, ma dai dati di cui si è a disposizione grazie agli studi effettuati dall'università di Berkley (1), si ha ragion di credere che questa nuova API possa effettivamente apportare delle migliorie allo stato dell'arte, divenendo così necessaria se si hanno delle esigenze computazionali superiori a quelle supportate da Pandas.

8. Bibliografia

1. *Towards Scalable DataFrame Systems.* Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi MoJoseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, Aditya Parameswaran.

2. Petersohn, Devin. Preventing the Death of the DataFrame. [Online] .https://towardsdatascience.com/preventing-the-death-of-the-DataFrame-8bca1c0f83c8.

3. Pandas on Ray. *Riselab.* [Online] 7 Luglio 2018. https://rise.cs.berkeley.edu/blog/pandas-on-ray-early-lessons/.

4. Modin: accelera i tuoi taccuini, script e librerie di Panda. *ICHI.PRO.* [Online] https://ichi.pro/it/modin-accelera-i-tuoi-taccuini-script-e-librerie-di-panda-214046102281077.

5. DataFrame. *Dask.* [Online] https://docs.dask.org/en/latest/DataFrame.html#common-uses-and-anti-uses.

6. Come Dask accelera l'ecosistema di Pandas? *ICHI.PRO.* [Online] https://ichi.pro/it/come-dask-accelera-l-ecosistema-di-pandas-243949949183541.

7. Documentazione Dask. [Online] https://dask.org/.

8. Yong, Geng. Ray VS Dask. [Online] 4 Gennaio 2019. https://coolgeng.medium.com/ray-vs-dask-d0154a774f2a.

9. Pandas References. *Pandas.* [Online] https://pandas.pydata.org/.

10. Pandey, Parul. Get faster pandas with Modin, even on your laptops. . *Towards Data Science* . [Online] 27 Marzo 2019. https://towardsdatascience.com/get-faster-pandas-with-modin-even-on-your-laptops-b527a2eeda74.

11. Modin References. *Modin.* [Online] https://modin.readthedocs.io.

12. Modin Project. *GitHub.* [Online] https://github.com/modin-project/modin.