Alma Mater Studiorum - Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica per il Management

Processamento di dati tabulari mediante SQL e Pandas: un confronto sperimentale

Relatore:
Chiar.mo Prof.
MARCO DI FELICE

Presentata da:
ENRICO LANCONELLI

Anno Accademico 2023/2024
Sessione IV

If we all reacted the same way, we'd be predictable, and there's always more than one way to view a situation. What's true for the group is also true for the individual. It's simple: Overspecialize, and you breed in weakness. It's slow death.

Major Motoko Kusanagi

Abstract

Nel mondo della data science, elemento cardine nel business moderno, ottimizzare le operazioni sui dataset è fondamentale per garantire efficienza, fluidità e scalabilità. A tal proposito, la tesi confronta due approcci per la gestione ed il processamento di dati tabulari. Pandas è una libreria di Python ampiamente utilizzata per l'analisi e la manipolazione di dati strutturati, grazie alla sua capacità di operare con facilità ed efficienza su dataset caricabili in memoria.
D'altra parte, SQL rappresenta lo standard per la gestione e l'interrogazione di database relazionali, risultando una scelta consolidata quando si lavora con grandi volumi di dati. La tesi propone una comparazione qualitativa e quantitativa del supporto di Pandas e SQL per dati tabulari. Attraverso una serie di test sperimentali su dataset di dimensioni variabili, si analizza quale dei due strumenti risulti più performante per diverse tipologie di operazioni previste sui dati.

Indice

Elenco delle figure

1. Introduzione

Nel panorama attuale dell'elaborazione dati, la scelta degli strumenti per l'interrogazione e la manipolazione dei database rappresenta un elemento cruciale per garantire efficienza, affidabilità e rapidità nei processi decisionali. Tradizionalmente, SQL è stato lo standard indiscusso per la gestione dei database relazionali, grazie alla sua robustezza e alla capacità di operare su grandi volumi di dati. Tuttavia, SQL presenta dei limiti in termini di flessibilità e programmabilità, caratteristiche sempre più richieste in un contesto in cui l'elaborazione in tempo reale e la personalizzazione delle operazioni diventano determinanti.

Parallelamente, la libreria Pandas di Python ha acquisito notevole popolarità per la sua capacità di gestire dataset interamente in memoria, consentendo operazioni di analisi e manipolazione dati rapide e intuitive. Questa caratteristica la rende particolarmente interessante in scenari dove il volume dei dati, pur rilevante, è tale da poter essere caricato completamente in RAM, permettendo così un'interrogazione più dinamica e flessibile rispetto all'approccio tradizionale basato su SQL.

Nel corso della tesi sono stati progettati ed eseguiti otto test, volti a confrontare le prestazioni di Pandas e SQL in alcune operazioni comuni di processamento dei dati. Sono stati costruiti tre dataset tabellari di 1 milione di righe e 27 colonne, popolando rispettivamente, le prime 25 con stringhe casuali, la 26ª con numeri interi di 6 cifre e la 27ª con un formato categoriale (frutti scelti da un pool di 10 per il primo dataset, 30 per il secondo e 50 per il terzo). Sono stati prodotti otto script Python per implementare le diverse tipologie di query dei test, rispettivamente ricerca di una stringa, raggruppamenti e conteggi senza e con filtraggio condizionale, operazioni aritmetiche sulla 26ª colonna (calcolo di massimo, minimo e somma) e join tra due dataset con e senza indicizzazione preventiva. Ogni test è stato eseguito su diverse configurazioni di dati crescenti, ovvero prendendo in esame un determinato numero di righe dal dataset interrogato da poche decine, fino a giungere alla totalità del volume di dati archiviato. Al fine di aumentare la consistenza delle registrazioni, ciascuna prova è stata eseguita un determinato numero di volte per configurazione, partendo da 30 ripetizioni per le prime cinque configurazioni, e aumentando di cinque ripetizioni per configurazione successiva (fino ad un massimo di 60 esecuzioni sul milione di righe). I tempi d'esecuzione sono stati raccolti e rielaborati mediante l'applicazione di indici statistici (media, varianza, deviazione standard e intervallo di confidenza al 99%). Si sono quindi realizzati dei grafici, con cui poter analizzare i risultati, i quali hanno evidenziato le differenze nelle prestazioni tra le due metodologie, sottolineando come Pandas risulti incredibilmente efficiente in ambito di operazioni aritmetiche e nell'esecuzione di query relativamente poco complesse; mentre

in un contesto in cui la complessità dell'operazione cresce o debbano essere eseguite più richieste nella stessa esecuzione, come nel caso del raggruppamento con filtraggio condizionale, Pandas ha dimostrato prestazioni di carattere inferiore rispetto a SQL. Questa attività sperimentale ha permesso di delineare i punti di forza e le criticità dei due approcci nelle situazioni considerate, fornendo basi solide su cui intavolare le conclusioni della tesi e lasciando spazio a possibili ricerche future.

1.1 Struttura della tesi

La tesi si compone di una parte dedicata alla rassegna dello **Stato dell'Arte del modello relazionale SQL**, con l'obiettivo di fornire un quadro generale su cosa sia il modello relazionale, quali siano le applicazioni in cui il linguaggio SQL è considerato lo standard e come sia nata l'esigenza di sviluppare strumenti alternativi per la gestione e l'interrogazione dei database. A tale rassegna seguirà una panoramica dello **Stato dell'Arte della libreria Pandas**, focalizzata sul suo funzionamento e sulle sue prestazioni, al fine di garantire una chiara contestualizzazione delle prove.

Successivamente verrà discussa la **Progettazione** della parte sperimentale, in cui vengono motivati e precisati gli obiettivi della tesi, illustrata la metodologia seguita per la costruzione dei database utilizzati nelle prove e descritto nel dettaglio come sono stati progettati i test.

Si procederà poi con l'analisi dell'**Implementazione** dei test, spiegando la logica sottostante e il codice, in modo da mettere a disposizione del lettore tutti gli elementi necessari per comprendere il quadro generale della situazione e l'intento delle prove.

Il capitolo dedicato alla **Validazione Sperimentale** sarà incentrato sulla descrizione dei risultati ottenuti, presentati graficamente, e sulle metodologie applicate alle varie prove. In questa parte verrà approfondita la metrica scelta per la rappresentazione grafica, al fine di valutare con chiarezza i risultati e trarne le relative conclusioni, tenendo sempre in considerazione le previsioni attese.

Infine, il capitolo **Conclusioni e Sviluppi Futuri** riprenderà brevemente quanto esposto, consentendo di valutare nel loro insieme le prove svolte e i risultati ottenuti, al fine di trarne le opportune conclusioni. Verranno inoltre ipotizzati possibili sviluppi in campo di sperimentazione, valutando variabili e scenari, che qui non sono stati presi in considerazione, con cui operare confronti di rilievo in futuro, delineando così un naturale seguito al lavoro qui svolto.

Il linguaggio SQL

2.1 L'importanza della gestione dei dati

Negli ultimi decenni, la crescita esponenziale dei dati generati da sistemi informativi, dispositivi mobili e applicazioni web ha reso imprescindibile l'adozione di metodologie e strumenti capaci di archiviare, gestire e manipolare grandi quantità di informazioni.

In questo contesto è ormai ampiamente impiegato il termine *Big Data*, il quale si riferisce a dataset caratterizzati da un volume, una velocità e una varietà tali da superare le capacità dei tradizionali sistemi di gestione dei dati. I Big Data rappresentano insiemi informativi così vasti e complessi da richiedere tecnologie e metodologie avanzate per il loro trattamento, memorizzazione e analisi. Come evidenziato da Laney (10), la definizione dei Big Data si fonda su tre "V": Volume, Velocità e Varietà. Tali caratteristiche implicano che la capacità di estrarre valore dai dati diventi un fattore determinante non solo per il successo aziendale, ma anche per il progresso scientifico e tecnologico. Inoltre, report di mercato come quello di Statista (13) confermano l'importanza strategica delle tecnologie di gestione dati in contesti economici e industriali. Pertanto, le organizzazioni che operano in contesti competitivi, si affidano a soluzioni di gestione dati che garantiscano rapidità, affidabilità e scalabilità, in modo da supportare processi decisionali complessi. La necessità di trattare e analizzare dataset di dimensioni sempre maggiori ha spinto lo sviluppo di numerosi paradigmi e strumenti, tra cui spicca il modello relazionale, la cui adozione ha rappresentato una svolta nel modo in cui i dati vengono organizzati e interrogati.

2.2 Il modello relazionale

Il modello relazionale, introdotto da Edgar F. Codd nel 1970 (2), si fonda sull'organizzazione dei dati in tabelle composte da righe e colonne, in cui le relazioni tra le informazioni sono esplicitamente definite attraverso chiavi primarie e chiavi esterne. Tale approccio ha rivoluzionato la gestione dei dati, sostituendo modelli più rigidi e meno flessibili, e ha offerto numerosi vantaggi: la separazione logica dei dati, l'indipendenza dalla struttura fisica, la capacità di garantire integrità e consistenza delle informazioni e, soprattutto, la possibilità di eseguire interrogazioni complesse in maniera intuitiva. Il modello relazionale ha dimostrato di essere estremamente versatile, riuscendo a mantenere la propria rilevanza nel tempo nonostante l'evoluzione delle tecnologie informatiche. Con l'aumentare delle esigenze in termini di prestazioni e flessibilità, il modello ha subito diverse evoluzioni, rimanendo tuttora uno degli standard principali per

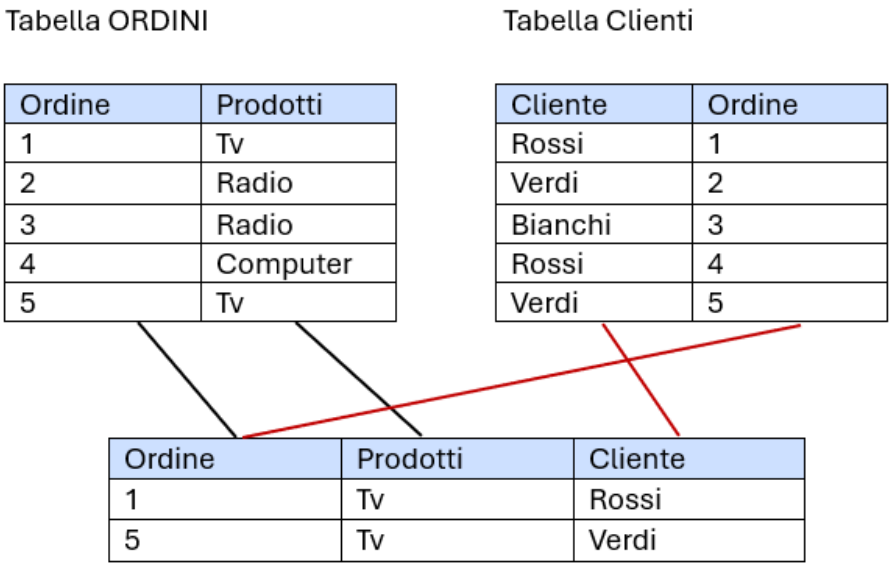


Figura 2.1. Esempio di database relazionale

la gestione dei database in ambito aziendale e accademico. Il suo utilizzo, supportato da numerose implementazioni commerciali e open source, testimonia la solidità e l'efficacia del paradigma relazionale.

Un disegno schematico del modello relazionale evidenzia come i dati siano organizzati in tabelle interconnesse: ad esempio, una tabella "Clienti" può essere collegata a una tabella "Ordini" tramite un vincolo di chiave esterno, garantendo così che ogni ordine sia associato a un cliente specifico. Questo approccio consente di evitare ridondanze e di mantenere una struttura dei dati pulita e normalizzata, facilitando al contempo l'aggiornamento e la manutenzione delle informazioni.

Il modello relazionale si basa su alcuni concetti fondamentali, quali:

- **Chiave primaria**: un attributo (o insieme di attributi) che identifica in modo univoco ogni record in una tabella. La chiave primaria garantisce l'integrità dei dati e ne facilita l'indicizzazione.
- **Chiave esterna:** un attributo che crea un collegamento tra due tabelle, facendo riferimento alla chiave primaria di un'altra tabella. Questo meccanismo è essenziale per mantenere l'integrità referenziale e permettere operazioni di join.
- **Valori NULL**: indicano l'assenza di un valore definito in un campo. I valori NULL sono utilizzati per rappresentare informazioni mancanti o non applicabili e

devono essere gestiti con attenzione per garantire la correttezza delle query e delle operazioni di aggregazione.

Questi concetti sono alla base della normalizzazione e dell'organizzazione dei dati in un database relazionale, contribuendo a garantire la consistenza e l'efficienza delle operazioni di interrogazione.

2.3 Pregi e limiti del linguaggio SQL

SQL (Structured Query Language) si configura come il linguaggio standard per la gestione e l'interrogazione dei database relazionali. Progettato per operare su tabelle organizzate secondo il modello relazionale, SQL consente di eseguire operazioni di definizione, manipolazione e controllo dei dati attraverso una sintassi relativamente semplice, ma estremamente potente. Le istruzioni SQL, come `SELECT, INSERT, UPDATE e DELETE`, permettono di eseguire query complesse su dataset di grandi dimensioni, garantendo integrità e coerenza attraverso l'uso di transazioni e vincoli. Per ulteriori dettagli sulle specifiche si rimanda alla documentazione ufficiale (18; 19).

Uno dei principali pregi di SQL risiede nella sua capacità di tradurre in modo diretto le richieste dell'utente in operazioni sui dati, sfruttando ottimizzazioni interne che consentono di minimizzare i tempi di risposta, anche su volumi di dati significativi. La struttura del linguaggio, basata su una sintassi dichiarativa, permette all'utente di specificare il risultato desiderato senza dover indicare esplicitamente i dettagli dell'algoritmo di esecuzione, delegando al motore di database il compito di trovare il piano di esecuzione più efficiente.

SQL, pur essendo un linguaggio potente e specificamente progettato per la gestione di database relazionali, presenta alcune limitazioni quando si tratta di operazioni dinamiche e trasformazioni complesse che richiedono logiche di programmazione avanzate. Una delle principali difficoltà risiede nella sua sintassi limitata, che è orientata alla manipolazione di dati attraverso un insieme predefinito di operazioni standard (`SELECT, INSERT, UPDATE, DELETE, JOIN`, ecc.). Queste operazioni, per quanto flessibili e ampie, sono comunque vincolate a uno schema rigido, rendendo più complesso descrivere problemi che esulano dalla classica manipolazione tabellare.

Un'altra limitazione riguarda la mancanza di programmazione a runtime: in SQL, le query sono statiche e vengono interpretate ed eseguite dal database in un'unica fase. Non è possibile modificarle dinamicamente durante l'esecuzione o applicare logiche iterative e condizionali complesse come si farebbe in un linguaggio di programmazione completo. Questo significa che, mentre SQL è ottimizzato per interrogazioni efficienti su

grandi quantità di dati, non è altrettanto adatto a flussi di elaborazione che richiedono un controllo più granulare e adattivo sulle operazioni.

Nel confronto con strumenti come Pandas, che operano direttamente sulla memoria e permettono un'elaborazione più dinamica, emerge chiaramente la differenza tra un sistema pensato per la gestione strutturata di dati e uno che offre maggiore libertà di manipolazione, grazie a un'interfaccia più vicina alla programmazione tradizionale.

2.4 Settori di applicazione di SQL

Il linguaggio SQL, grazie alla sua affidabilità e alla maturità delle implementazioni, è largamente impiegato in numerosi settori che richiedono la gestione di grandi volumi di dati e l'esecuzione di operazioni transazionali complesse. In ambito finanziario, ad esempio, SQL è fondamentale per il controllo delle transazioni, la gestione dei portafogli e la reportistica. Le banche e le istituzioni finanziarie si affidano a database relazionali per garantire la sicurezza e la precisione delle operazioni quotidiane, con sistemi che gestiscono milioni di transazioni al giorno (13).

Nel campo della ricerca scientifica, SQL è ampiamente utilizzato per gestire grandi dataset sperimentali e osservazionali. Ad esempio, il Sloan Digital Sky Survey (SDSS) raccoglie petabyte di dati astronomici raccolti da telescopi avanzati, dati poi interrogati dagli astronomi tramite SQL per identificare, ad esempio, oggetti celesti con caratteristiche specifiche e studiare l'evoluzione delle galassie (15). Analogamente, il GenBank è un archivio fondamentale per il settore delle scienze della vita, dove simulazioni molecolari e sequenziamenti genetici sono organizzati in un database open-access per studi biomedici (11).

Nell'ingegneria spaziale, la gestione e l'analisi di dati sono cruciali per il monitoraggio di missioni e satelliti. La NASA, ad esempio, utilizza database SQL per archiviare e analizzare dati provenienti dai sistemi di telemetria dei veicoli spaziali. I dati raccolti dai sensori di bordo, che monitorano temperatura, pressione, consumi energetici e stato dei sistemi, vengono memorizzati in grandi database e analizzati per prevedere guasti o ottimizzare le operazioni di volo. Inoltre, SQL viene impiegato per l'analisi di dati scientifici raccolti da missioni come il Mars Rover e le sonde interplanetarie, supportando la gestione di immagini ad alta risoluzione e dati ambientali provenienti da altri pianeti (12).

Anche nel settore della sanità, la gestione dei dati clinici e delle informazioni relative ai pazienti si basa su sistemi SQL, che permettono di mantenere elevati standard di sicurezza e di integrità dei dati. Inoltre, in ambito governativo e nelle grandi aziende, SQL

rappresenta il cuore dei sistemi informativi, con applicazioni che spaziano dalla gestione delle risorse umane alla logistica, fino al monitoraggio delle performance operative. Le statistiche confermano l'ampia adozione del modello relazionale: secondo recenti report di mercato (13), oltre il 70% dei database aziendali si basa su tecnologie SQL2, evidenziando come la solidità e la versatilità di questo approccio continuino a garantire un ruolo di primo piano nella gestione dei dati.

L'adozione di SQL è anche favorita dalla vasta comunità di sviluppatori e dalla presenza di standard internazionali (ad esempio, ISO/IEC 9075) che ne garantiscono l'*interoperabilità* e la *portabilità* tra differenti piattaforme (in campo informatico con interoperabilità s'intende la capacità di un sistema o di un prodotto di cooperare e di scambiare informazioni o servizi con altri sistemi o prodotti in maniera più o meno completa e ottimizzata; mentre per portabilità s'intende la capacità di adattamento di un software ad essere eseguito e performare in un ambiente diverso da quello in cui è stato sviluppato inizialmente). Ciò rende il linguaggio non solo uno strumento di gestione dati, ma anche un pilastro fondamentale per la costruzione di sistemi informativi integrati e interoperabili.

2.5 Modalità operative di SQL

Negli ambienti tradizionali, l'utilizzo di SQL si fonda su sistemi di gestione di database relazionali (RDBMS) quali MySQL, PostgreSQL, Oracle e SQLite. In questi sistemi, i dati vengono memorizzati su supporti di memoria esterna, come dischi rigidi o SSD, e non vengono caricati integralmente in RAM. Questo approccio, pur garantendo la robustezza, l'integrità e la scalabilità necessarie per gestire dataset di dimensioni estremamente elevate, comporta un inevitabile *overhead* legato agli accessi I/O e alla gestione di transazioni complesse. Per operare con SQL è necessario un ambiente di programmazione dedicato, in cui le query, pur essendo espresse in una sintassi relativamente semplice e dichiarativa, si affidano a motori di ottimizzazione interni che selezionano il piano di esecuzione più efficiente. Tali caratteristiche hanno reso SQL lo strumento standard per la gestione dei dati in numerosi settori, soprattutto in quelli in cui la sicurezza e la consistenza delle informazioni sono prioritarie.

In contrasto, alcune soluzioni emergenti si focalizzano su un approccio in-memory, che permette di caricare l'intero dataset nella memoria RAM e di eseguire operazioni di manipolazione e analisi in maniera estremamente rapida. Questa modalità offre, oltre a tempi di risposta notevolmente ridotti, una maggiore flessibilità programmativa e un'interazione più dinamica con i dati. Questi aspetti hanno stimolato lo sviluppo di

strumenti alternativi, tra cui Pandas, una libreria del linguaggio Python, che propongono un paradigma diverso rispetto all'approccio tradizionale basato su SQL.

Il quadro complessivo evidenzia come il modello relazionale e SQL continuino a rappresentare un punto fermo nella gestione dei dati, garantendo affidabilità e consistenza su larga scala. Tuttavia, l'evoluzione delle esigenze applicative e la crescente importanza dell'elaborazione in-memory hanno aperto la strada a nuove soluzioni, capaci di integrare l'efficienza operativa con una maggiore flessibilità. Tale dinamica costituirà il fulcro delle sezioni successive, nelle quali verrà approfondito lo stato dell'arte della libreria Pandas, offrendo un confronto diretto e mirato tra i paradigmi tradizionali e le metodologie alternative da essa proposte.

3. Gestione di dati tabulari con Pandas

3.1 Origine di Pandas

Pandas (Python Data Analysis Library) è una libreria open source per il data analysis e la manipolazione dei dati, sviluppata in Python e ampiamente adottata dalla comunità scientifica e industriale (16). Originariamente creata da Wes McKinney intorno al 2008, Pandas è nata dalla necessità di colmare una lacuna nel panorama degli strumenti Python: la mancanza di strutture dati efficienti e flessibili per il trattamento di dati eterogenei, tipici dei dataset reali. La sua introduzione ha permesso di operare in modo semplice e interattivo su dati tabellari, offrendo agli utenti la possibilità di trasformare, filtrare e aggregare dati con una sintassi espressiva e intuitiva. Pandas permette di prendere dati da file di vario formato da database SQL, provvedendo poi alla loro gestione tramite un oggetto Python apposito, detto *dataframe*. Tale flessibilità, unita all'integrazione completa con l'ambiente Python (in particolare con librerie come NumPy e matplotlib), ha reso Pandas una valida alternativa ai tradizionali sistemi basati su SQL, soprattutto in contesti in cui i dati possono essere interamente caricati in memoria.

3.2 Principali applicazioni

Oggi Pandas è impiegato in numerosi settori grazie alla sua capacità di semplificare le operazioni di *data wrangling*[1] e analisi statistica.

Numerosi studi e articoli evidenziano come Pandas sia divenuto uno strumento fondamentale per la data analysis in ambienti accademici, industriali e finanziari. Ad esempio, McKinney (1) illustra come l'introduzione di strutture dati efficienti in Pandas abbia rivoluzionato il modo di gestire il calcolo statistico in Python, rendendo possibile l'elaborazione rapida di dataset di grandi dimensioni (anche grazie al supporto della libreria Numpy). Nello studio (4), gli autori mostrano come l'utilizzo di Pandas per visualizzare le trasformazioni delle tabelle supporti efficacemente l'insegnamento e l'applicazione pratica della data science; mentre l'importanza della profilazione continua dei dati per l'analisi interattiva è evidenziata nella ricerca (8), dove si sottolinea come tecniche avanzate di data profiling possano supportare analisi dinamiche nei process di data science. Lo studio sperimentale di Nahrstedt et al. (5) mette in luce i vantaggi in termini di consumo energetico e performance di Pandas rispetto ad altre librerie,

[1] processo di trasformazione e strutturazione dei dati da una forma grezza a un formato desiderato con l'intento di migliorare la qualità dei dati e renderli più fruibili e utili per l'analisi o il machine learning.

sottolineando la sua rilevanza nell'ottimizzazione delle risorse computazionali. Zhuang e Lu (6) affrontano invece il problema del type checking nei DataFrame, evidenziando come l'architettura di Pandas contribuisca a garantire l'integrità dei dati. Infine, Li et al. (7) discutono della convergenza tra *data integration* e *machine learning*, dimostrando come Pandas costituisca un ponte cruciale tra fonti di dati eterogenee e modelli analitici avanzati.

In ambito statistico, la capacità di aggregare e trasformare dataset eterogenei ha trovato applicazioni diffuse nella ricerca scientifica, nell'analisi di dati clinici e nei modelli predittivi nel settore finanziario. Ad esempio, McKinney (1) evidenzia come l'introduzione di strutture dati efficienti in Pandas abbia rivoluzionato l'elaborazione statistica in Python, permettendo agli utenti di gestire dataset complessi in maniera intuitiva e performante. Inoltre, Lau et al. (4) sottolineano come la curva di apprendimento relativamente breve e la documentazione esaustiva di Pandas abbiano facilitato la sua adozione diffusa, contribuendo a numerose collaborazioni accademiche e progetti open source. Questi elementi, supportati dalla documentazione ufficiale (16), hanno reso Pandas uno strumento essenziale per data scientist e ingegneri del software, affermandone la versatilità e utilità in vari ambiti applicativi.

3.3 Funzionamento e integrazione

Pandas si installa facilmente tramite gestori di pacchetti come pip o conda, integrandosi perfettamente in ambiente Python (ad esempio, dal prompt dei comandi è sufficiente digitare "pip install pandas"). Una volta installata, la libreria rende disponibili strutture dati come DataFrame e Series, che rappresentano rispettivamente tabelle e vettori di dati eterogenei. Queste strutture consentono di effettuare operazioni di indicizzazione, filtraggio, aggregazione e join in maniera altamente ottimizzata grazie alla programmazione vettoriale fornita da NumPy (17). È interessante notare come, in ambiente Python, si possano integrare query SQL tramite strumenti quali sqlite3 o SQLAlchemy, permettendo di confrontare direttamente l'approccio basato su database esterni con l'elaborazione in-memory offerta da Pandas.

Nella pratica, un utilizzo concreto di Pandas in ambiente Python ha inizio con il caricamento dei dati da file esterni, che possono essere in vari formati quali CSV, TSV, Excel, JSON, oppure provenienti da database SQL. Durante questa fase, il file presente su disco viene letto e trasferito in memoria, operazione che comporta un tempo di esecuzione non trascurabile e che incide significativamente sui benchmark sperimentali,

soprattutto con dataset di grandi dimensioni. Una volta caricati, i dati vengono convertiti in una struttura dati fondamentale di Pandas: il DataFrame. Questo oggetto, simile a una tabella di Excel, permette una manipolazione fluida ed efficiente dei dati, offrendo funzionalità avanzate per il filtraggio, l'aggregazione e la trasformazione delle informazioni.

Il fatto che Pandas operi in-memory, come descritto nella documentazione ufficiale (16) consente tempi di risposta estremamente rapidi, sebbene limiti la dimensione dei dati gestibili alla quantità di RAM disponibile. Tale caratteristica è uno degli elementi chiave che differenzia l'approccio di Pandas da quello dei tradizionali sistemi SQL, che operano direttamente sui dati memorizzati su supporti di memoria di massa, rendendo la manipolazione e l'analisi dei dati più agile, come evidenziato da Aditya Parameswaran in (14).

3.4 Lavorare in-memory

Il concetto di elaborazione in-memory si basa sul caricamento dei dati direttamente nella memoria RAM, anziché utilizzare supporti di memoria di massa come dischi rigidi o SSD. La RAM é sì caratterizzata da una capacità inferiore rispetto alla memoria di massa, ma offre tempi di accesso notevolmente ridotti, consentendo operazioni di manipolazione e analisi dei dati molto più rapide. Pandas sfrutta questa caratteristica per permettere operazioni interattive e immediate sui dataset; grazie anche all'ottimizzazione derivante dall'utilizzo di algoritmi vettorializzati e di librerie C sottostanti, come BLAS e LAPACK (3). Al contrario, i database SQL tradizionali, progettati per operare su dati memorizzati esternamente, devono gestire complessi processi di I/O e caching che, sebbene garantiscano affidabilità e scalabilità, possono introdurre un overhead non trascurabile quando si tratta di interrogazioni veloci su dataset di dimensioni moderate. Si rammenta che con il termine *overhead* s'intendono tutte le risorse necessarie per eseguire un'operazione, pertanto è sempre intenzione del programmatore cercare di ridurre tale carico e di ottimizzarlo il più possibile.

3.5 Confronto con SQL

Le differenze operative tra Pandas e SQL si evidenziano soprattutto nella modalità di accesso ai dati. Pandas, operando in-memory, esegue operazioni grazie a funzioni vettorializzate, che sfruttano il calcolo a basso livello offerto da NumPy per operare in tempi estremamente rapidi.

Pandas	Sql
Libreria Open Source del linguaggio Python	Linguaggio di programmazione
Usato per lo più per leggere dati da file CSV	Legge dati dai database relazionali
Potrebbe non essere efficiente quando chiamato a gestire grandi volumi di dati	E' ottimizzato per gestire ed interrogare grandi volumi di dati
Grande vastità di funzioni integrate	Sono integrate solo le funzioni base
E' relativamente semplice performare operazioni complesse	Performare operazioni complesse è decisamente più difficoltoso

Figura 3.1. Tabella riassuntiva delle principali differenze tra Pandas e SQL

Ad esempio, il calcolo del valore massimo in una colonna di un DataFrame è implementato in modo tale da minimizzare il numero di iterazioni a livello di Python, avvalendosi di funzioni C ottimizzate per l'elaborazione numerica.
Al contrario, le query SQL vengono eseguite da motori di database che, pur essendo altamente ottimizzati per dataset di grandi dimensioni, devono spesso gestire la complessità degli accessi su memoria di massa e l'ottimizzazione del piano di esecuzione.
Inoltre, la natura statica delle query SQL – dove le istruzioni non possono essere modificate dinamicamente a runtime – limita la programmabilità rispetto all'approccio più fluido e iterativo di Pandas, in cui è possibile combinare, modificare e concatenare operazioni in maniera interattiva. Questo aspetto rende Pandas particolarmente interessante per scenari di analisi esplorativa, dove la flessibilità e la rapidità di sviluppo sono fondamentali. Un utile approfondimento su queste differenze (schematizzate a Fig. 3.1) pratiche è offerto da Paras Yadav, il cui articolo (21) evidenzia come l'approccio in-memory di Pandas consenta un'efficienza superiore per dataset che rientrano nelle capacità della RAM, a fronte delle limitazioni intrinseche dei database tradizionali.

3.6 Differenze pratiche con SQL

Di seguito vengono mostrate alcune operazioni di base (che prendono spunto da quelle prese in esame da Paras Yadav in (21)), ponendo in evidenza le differenze tra la sintassi dichiarativa con cui nel linguaggio SQL si interrogano database su disco, e l'azione diretta di Pandas su un oggetto *dataframe*, caricato direttamente in memoria.

1. **Selezione di righe con una condizione**

   - SQL

   ```
   SELECT *
   FROM my_table
   WHERE age > 50;
   ```

   - Pandas

   ```
   df_filtered = df.loc[df['age'] > 50]
   ```

   In questo esempio, `loc` seleziona tutte le righe della tabella in cui *age* è maggiore di 50.

2. **Selezione di specifiche colonne**

   - SQL

   ```
   SELECT nome, cognome
   FROM my_table;
   ```

   - Pandas

   ```
   df_subset = df[['nome', 'cognome']]
   ```

   Pandas consente di selezionare un sottoinsieme di colonne passando una lista di stringhe corrispondenti ai nomi delle colonne, caratteristica utile nell'ottica di una programmazione dinamica, tuttavia non dissimile a quanto proposto dal linguaggio SQL. Tuttavia, se si volessero selezionare delle colonne, o delle righe, senza conoscerne il nome, in SQL diverrebbe estremamente difficile da realizzare. In ambiente Pandas questo è reso possibile grazie al comando `iloc`:

   ```
   df_subset = df.iloc[10:20, 0:3]
   ```

   La funzione `iloc` rende possibile selezionare righe e colonne in base agli indici, in questo esempio vengono considerate le righe partendo da quella di indice 10 fino a quella di indice 19 e le colonne dalla prima alla terza (indice dallo 0 al 2).
   Ricapitolando, Pandas mette a disposizione due funzioni per la selezione di righe colonne:

- `loc` (label-based indexing) : selezione basata sui nomi di righe e colonne;
- `iloc` (integer-based indexing : selezione basata sugli indici di righe e colonne.

Questa differenza tra le due funzioni rende Pandas decisamente flessibile in ambito di accesso ai dati, colmando inoltre un possibile *blind spot* del linguaggio SQL.

3. **Selezione delle prime dieci righe**

   - SQL

   ```
   SELECT *
   FROM my_table
   LIMIT 10
   ```

   - Pandas

   ```
   first_ten_rows = df.head(10)
   ```

   In Pandas è sufficiente ricorrere al comando 'head()' per selezionare le prime n righe di una tabella.

4. **Calcolo del Valore Massimo**

   - SQL

   ```
   SELECT MAX(age)
   FROM my_table;
   ```

   - Pandas

   ```
   max_age = df['age'].max()
   ```

   Operazioni come quella di somma, sono estremamente simili a livello sintattico, ma presente differenze sul piano algoritmico. In Pandas, questo tipo di operazioni matematiche, vengono eseguite in memoria sfruttando funzioni vettorializzate ed ottimizzazioni interne (grazie alla libreria NumPy), che riducono il numero di iterazioni a livello Python e migliorano le prestazioni.

3.7 Prestazioni su dataset tabellari

Diversi studi hanno esaminato le prestazioni di Pandas nell'elaborazione di dataset tabellari, evidenziando come l'adozione di operazioni vettorializzate consenta di gestire

in modo efficiente anche dataset di dimensioni rilevanti, purché rientrino nella memoria disponibile. Ad esempio, ricerche pubblicate su ACM Digital Library (22) e IEEE Xplore (23) hanno messo a confronto il consumo di memoria e i tempi di esecuzione di Pandas rispetto ad altre librerie di data processing. In particolare, lo studio (5) dimostra che, per operazioni quali il sorting o l'aggregazione dei dati, Pandas sfrutta algoritmi ottimizzati (come Timsort) che garantiscono prestazioni costanti anche al variare della dimensione del dataset. Questo articolo conferma che, sebbene Pandas possa comportare un consumo di memoria considerevole con dataset molto grandi, rimane estremamente efficiente e flessibile per la maggior parte delle applicazioni di data analysis in ambiente Python. Inoltre, la capacità di un motore di gestire in modo unificato operazioni di business intelligence e calcoli algebrici è stata dimostrata in (9), rafforzando l'idea che strumenti ottimizzati possano garantire prestazioni costanti anche in contesti di elaborazione dati complessi.

Il quadro delineato evidenzia come Pandas rappresenti uno strumento fondamentale per l'analisi e la manipolazione dei dati in-memory, grazie alla sua capacità di combinare facilità d'uso, elevata programmabilità e prestazioni notevolmente ottimizzate. Pur riconoscendo i punti di forza dei sistemi basati su SQL per la gestione di dati su larga scala, l'approccio offerto da Pandas si configura come una valida alternativa per scenari in cui la rapidità e la flessibilità operativa sono determinanti. In particolare, le differenze in termini di performance d'esecuzione delle query saranno la base su cui si innesteranno le prove sperimentali messe in atto ed esposte in questa tesi.

4. Progettazione

Il presente capitolo ha l'obiettivo di definire e motivare la progettazione delle prove sperimentali volte a confrontare le prestazioni di Pandas, libreria di Python per l'analisi dei dati in-memory, con quelle di SQL, il linguaggio standard per l'interrogazione dei database relazionali. Di seguito, verranno illustrati i principali obiettivi della tesi, la metodologia adottata per la costruzione dei database di prova e una breve descrizione delle tipologie di test scelti, così da fornire una solida base concettuale per le successive fasi di analisi tecnica.

4.1 Obiettivi della tesi

Nel presente lavoro è perseguito l'obiettivo di analizzare in maniera approfondita le prestazioni della libreria Pandas in operazioni di complessità variabile su dati tabulari, al fine di valutarne i tempi di esecuzione. Tale analisi di prestazioni avverrà tramite il confronto con SQL, il linguaggio standard per l'interrogazione dei database relazionali.

L'approccio operazionale proposto si fonda sul confronto diretto tra le caratteristiche intrinseche dei due strumenti: mentre SQL, con la sua sintassi dichiarativa e la struttura operativa statica, garantisce robustezza, consistenza e affidabilità nei contesti in cui i dati risiedono su supporti di memoria di massa, Pandas opera in modalità in-memory, consentendo così di eseguire manipolazioni e aggregazioni in maniera estremamente rapida e dinamica. Tale differenza operativa è di particolare interesse, in quanto evidenzia la possibilità di adottare Pandas come alternativa vantaggiosa per l'elaborazione dei dati in ambiente Python, soprattutto in applicazioni in cui il volume dei dati rientra nelle capacità della memoria disponibile. Oltretutto, il linguaggio SQL è confezionato per operare con velocità e consistenza con volumi di dati di gran lunga maggiori di dataset caricabili in memoria RAM; pertanto, il confronto su dati di queste dimensioni potrebbe rivelarsi particolarmente significativo.

Attualmente, la letteratura accademica disponibile, in particolare su piattaforme come IEEE Xplore (23) e ACM Digital Library (22), non fornisce confronti diretti e sistematici tra Pandas e SQL in termini di tempo di esecuzione per operazioni comuni su dataset interamente caricabili in memoria. Numerosi studi si concentrano invece sull'efficienza computazionale in contesti di Big Data e framework distribuiti, lasciando in ombra il confronto diretto tra questi due approcci per dataset di dimensioni moderate.

A conferma di questa lacuna, si possono trovare diverse discussioni informali e post in forum, come quelli su Stack Overflow, Reddit e Medium, in cui gli utenti condividono esperienze pratiche e osservazioni riguardo alla velocità e alla flessibilità di Pandas

rispetto a SQL. Tali fonti, pur essendo di natura meno formale e certificata, indicano che in ambienti Python l'approccio in-memory offerto da Pandas può talvolta risultare superiore a SQL, grazie ai rapidi tempi di accesso alla RAM e alla maggiore dinamicità delle operazioni, ribadendo saltuariamente come per piccoli volumi di dati le differenze potrebbero essere impercettibili e supponendo in linea di massima che per operazioni che coinvolgono un volume di dimensione considerate, Pandas potrebbe risultare più rapido, in termini di prestazioni. Tuttavia, queste osservazioni non sempre sono supportate da benchmark rigorosi e generalizzabili, nè tantomeno provengono da fonti affidabili, evidenziando l'esigenza di un'analisi comparativa approfondita.

Questa mancanza di studi sistematici e generalizzabili può essere attribuita a diversi fattori, tra le quali si annoverano:

- **variabilità delle dimensioni dei dataset**
  le prestazioni di Pandas e SQL dipendono fortemente dalla quantità di dati elaborati. Un'operazione su un piccolo dataset potrebbe essere più veloce in Pandas grazie alla gestione in-memory, mentre su dataset molto grandi SQL potrebbe risultare più efficiente per la sua ottimizzazione nei sistemi di archiviazione. La scalabilità è quindi un aspetto chiave, e ogni confronto dovrebbe tener conto di diversi ordini di grandezza nei dataset.

- **dipendenza dall'hardware**
  le prestazioni di Pandas sono strettamente legate alle risorse della macchina su cui viene eseguito il codice. Processori più potenti, maggiore quantità di RAM e velocità del disco possono influenzare significativamente i risultati. SQL, invece, è spesso eseguito su database ottimizzati per l'hardware specifico, il che rende difficile fare confronti diretti senza standardizzare le condizioni di test.

- **ottimizzazioni interne e configurazioni**
  SQL beneficia di ottimizzazioni automatiche da parte del motore di database (come caching, parallelizzazione, indici), mentre le performance di Pandas possono variare in base a come i dati sono strutturati e al codice specifico utilizzato. Una stessa query SQL potrebbe essere eseguita in modo molto diverso a seconda del motore utilizzato (MySQL, PostgreSQL, SQLite), così come un'operazione Pandas può essere ottimizzata con tecniche come vectorization o uso di moduli esterni (es. Modin, Dask), che possano sì migliorare le prestazioni in meri termini di tempo, ma possono impattare in maniera più significativa l'uso della memoria.

- **mancanza di uno standard di benchmarking**
  Ogni studio dovrebbe definire chiaramente il tipo di test eseguiti, il contesto e

le metriche utilizzate. L'assenza di una metodologia condivisa per confrontare Pandas e SQL complica la possibilità di trovare studi coerenti e comparabili tra loro.

L'obiettivo della ricerca consiste, dunque, nel verificare se l'impiego di Pandas possa rappresentare una soluzione efficace per gestire e interrogare dataset in-memory, individuando al contempo eventuali limiti o punti di forza rispetto all'approccio SQL.

4.2 Una variabile locale: la Macchina

Un aspetto cruciale nella valutazione delle prestazioni di strumenti come Pandas e SQL è rappresentato dall'hardware su cui vengono eseguiti i test. La variabilità delle prestazioni in relazione alle dimensioni dei dataset, alle ottimizzazioni interne (come caching, parallelizzazione e indicizzazione) e alla configurazione hardware rende difficile avere un benchmark standard a livello globale. Infatti, operazioni eseguite in-memory dipendono fortemente dalla quantità di RAM, dalla potenza del processore e da altri fattori specifici dell'hardware; queste variabili possono determinare differenze significative nei tempi di esecuzione. Nel presente studio, i test sono stati condotti su un computer HP Pavilion 15-eh3006nl Notebook, dotato di un processore AMD Ryzen 7 7730U con Radeon Graphics (2.00 GHz) e una memoria RAM DDR4 da 16 GB (di cui 15,3 GB utilizzabili). Tale configurazione è stata scelta in quanto rappresenta una tipica postazione di lavoro in ambito accademico e consente di analizzare le performance in un contesto reale.

Un ulteriore fattore da considerare riguarda le condizioni operative durante l'esecuzione dei test. Oltre alle caratteristiche tecniche dei componenti hardware, l'uso effettivo a runtime riveste un ruolo altrettanto importante nel garantire la coerenza dei risultati. Per minimizzare eventuali interferenze, durante le prove verrà adottato un approccio controllato: tutte le applicazioni non essenziali saranno chiuse, e l'unico programma attivo in primo piano sarà l'editor Visual Studio, con cui verranno lanciati gli script Python. Rimangono attivi solo i processi di background indispensabili e il software antivirus standard di Windows, il quale, per operare in una condizione verosimile alla realtà, non verrà disabilitato. Tale configurazione operativa è stata scelta per garantire che i tempi di esecuzione misurati riflettano in maniera attendibile le performance dei test, eliminando quanto più possibile eventuali variazioni dovute ad altre attività del sistema.

Pur riconoscendo che i risultati potrebbero variare in presenza di hardware e condizioni differenti, occorre porre un passo dopo l'altro per giungere a destinazione: se si

raccoglieranno sempre più valutazioni sperimentali, eseguite in contesti differenti, sarà possibile ottenere una visione d'insieme e generalizzata delle performance di Pandas su dataset tabellari.

4.3 Metodologie di confronto

Per verificare se l'impiego di Pandas possa rappresentare una soluzione efficace per gestire e interrogare dataset in-memory, è stato progettato un confronto diretto tra le operazioni eseguite con Pandas e quelle realizzate tramite SQL. Il confronto si basa su una serie di test che prevedono l'esecuzione di operazioni tipiche – quali ricerca di stringhe, aggregazioni (calcolo di valori massimi, minimi e somme), raggruppamenti e join – su dataset tabellari di prova opportunamente costruiti. Per ciascuna operazione verranno effettuate diverse prove su sezioni via via sempre più grandi dei dataset di prova, fino a considerarli nella loro interezza. I tempi d'esecuzione dell'operazione Pandas e di quella SQL saranno quindi misurati e salvati man mano in appositi file. I risultati verranno rielaborati e affinati tramite indici statistici, e infine rappresentati mediante grafici, in modo da permettere un'osservazione rapida ed evidente, allo scopo di poter trarre le relative conclusioni.

In questa ricerca verranno valutate le performance sotto l'aspetto prettamente esclusivo del tempo d'esecuzione. Non verranno quindi presi in considerazione altri fattori, come ad esempio, l'impatto che l'esecuzione delle operazioni ha in memoria. Sarà, invece, valutato il tempo di caricamento del dataset in memoria, necessario ai fini del funzionamento delle operazioni Pandas. Questo tempo potrà essere aggiunto a quelli delle performance d'esecuzione delle operazioni, al fine di ottenere un quadro più generale e completo su cui elaborare le osservazioni.

4.4 Costruzione dei dataset

Per eseguire i test sperimentali, è stato necessario costruire dataset versatili e possibilmente rappresentativi delle tipologie di dati comunemente presenti nelle applicazioni reali. In particolare, i dataset utilizzati sono costituiti da 27 colonne e 10^6(1 milione) di righe, in particolare:

- le prime 25 colonne sono popolate con stringhe casuali di 10 caratteri, scelte per simulare il tipo di dati testuali che si riscontrano frequentemente in ambienti reali (ad esempio, codici, dati identificativi o brevi informazioni). Questa modalità di popolazione permette di ottenere un campione eterogeneo e versatile, che rispecchia

sì la varietà dei dati presenti in contesti aziendali e di ricerca, ma al contempo offre una diversità incline all'ambiente sperimentale.

- La 26ª colonna, denominata *Numbers*, contiene valori numerici interi a 6 cifre. Questa colonna è impiegata per valutare le operazioni aritmetiche, come il calcolo del valore massimo, minimo, la somma. (La dimensione dei numeri è inferiore se paragonata alle stringhe casuali delle altre colonne, tuttavia la scelta delle 6 cifre è stata resa necessaria per performare alcuni tipi di operazioni sui dati, come la sopracitata somma).

- La 27ª colonna, denominata *Fruit*, viene popolata casualmente con nomi di frutti. Nei test, sono utilizzati dataset differenti: il dataset di base contiene 10 frutti (apricot, peach, plum, pear, banana, kiwi, orange, apple, grapes, strawberry), mentre per altre prove vengono considerati dataset con 30 e con 50 frutti. Questi casi permettono di valutare l'efficacia delle operazioni di raggruppamento (`GROUP BY`) su scale diverse.

Figura 4.1. Sezione del dataset con 10 frutti

In questo lavoro i dataset sono salvati in formato CSV, largamente diffuso e particolarmente comodo da utilizzare in ambiente Python, e salvati localmente per garantire la maggior velocità possibile. La dimensione dei dataset non supera i 300 MB, assicurando così il loro caricamento in memoria per l'elaborazione. Non verranno quindi presi in esame file di altri formati, come ad esempio il TSV, JSON o dati provenienti direttamente da database SQL. Solo in alcune prove ci si avvarrà, al bisogno, di dataset di dimensione maggiore, con un numero di righe di 10^7 e una dimensione complessiva di circa 2.70 GB, per riscontrare la veridicità dei risultati.

4.5 Progettazione dei test

Per valutare le prestazioni di Pandas e confrontarle con quelle di SQL, sono stati progettati otto test rappresentativi di operazioni tipiche nell'ambito del data processing:

1. **Ricerca di una stringa**

   - *Scopo*: cercare una stringa di dieci caratteri generata casualmente all'interno di tutte le celle del dataset.
   - *Output atteso*: restituzione di un valore booleano che assume *True* qualora la stringa sia presente in almeno una delle celle del dataset.
   - *Dataset*: viene impiegato il dataset di base (10 *Fruit*).

2. **Raggruppamento e conteggio degli elementi**

   - *Scopo*: valutare l'efficacia delle operazioni di `GROUP BY`, raggruppando i dati in base alla colonna *Fruit* e contando il numero di occorrenze per ciascun gruppo.
   - *Output atteso*: verrà restituito un insieme di coppie (*Fruit*, n° di elementi per ciascun gruppo)
   - *Dataset*: la prova verrà eseguita tre volte, ognuna avverrà considerando un diverso dataset (prima quello da 10*Fruit*, poi quello da 30, infine quello da 50).

3. **Raggruppamento e selezione dei gruppi che rispettano una condizione**

   - *Descrizione*: si procederà con un raggruppamento per *Fruit*, come nella prova precedente, per poi considerare solo quei gruppi il cui numero di elementi è almeno l'11% del numero degli elementi totali della colonna.
   - *Scopo*: verificare l'utilizzo di filtri avanzati (`HAVING`) selezionando gruppi che soddisfino la condizione imposta.
   - *Output atteso*: un sottoinsieme dei gruppi il cui conteggio supera la soglia prestabilita.
   - *Dataset*: come per la prova precedente, si utilizzano tutti e tre i dataset (con 10, 30 e poi 50 *Fruit*).

4. **Ricerca del valore massimo sulla colonna *Numbers***

   - *Scopo*: confrontare le performance nell'aggregazione aritmetica tra Pandas e SQL (verrà inclusa anche un'operazione SQL che utilizza l'indicizzazione).
   - *Output atteso*: il massimo valore presente nella colonna *Numbers*.
   - *Dataset*: il dataset di base con 10 *Fruit* (qualora le differenze dovessero essere considerevolmente minime, si procederà effettuando anche qualche test con il dataset da $10^7 righe$).

5. **Ricerca dela valore minimo sulla colonna Numbers**

   - *Scopo*: come il test 4, ma per il valore minimo.
   - *Output atteso*: il minimo valore presente nella colonna *Numbers*
   - *Dataset*: come per il test 4.

6. **Calcolo della somma dei valori della colonna *Numbers***

   - *Scopo*: valutare le operazioni di aggregazione aritmetica (somma) e le relative performance.
   - *Output atteso*: la somma totale dei valori della colonna *Numbers*.
   - *Dataset*: come per il test 4. Ai fini di operare tutti i test sugli stessi dataset e mantenere le loro esecuzioni lineari e correlate, la colonna *Numbers* dei dataset è stata popolata con numeri interi casuali di sole 6 cifre, differendo dalle lunghezza di dieci caratteri delle stringhe casuali che albergano nella quasi totalità delle altre celle. Questo garantisce che la somma di 1 milione di numeri da 6 cifre sia al massimo pari a 999'999'000'000. Questo valore causerebbe overflow qualora il tipo di dato utilizzato fosse l'int32 (il cui massimo valore è 2'147'483'647), ma rientra nei limiti del formato int64 (BIGINT - limite 9'223'372'036'854'775'807). Optare per numeri a 6 cifre permette quindi di mantenere il valore finale entro i limiti accettabili per int64, evitando *overflow* ed errori di calcolo.

7. **JOIN tra due dataset con indicizzazione**

   - *Scopo*: valutare le performance dell'operazione di `JOIN` tra due dataset sulla colonna *Numbers* che verrà preventivamente indicizzata.
   - *Output atteso*: risultato del `JOIN`, ovvero l'unione dei record corrispondenti sulla base della colonna *Numbers*.

- *Dataset*: l'operazione di `JOIN` avviena tra il dataset di base con 10 *Fruit* e il dataset da 30 *Fruit*.

8. **JOIN tra due dataset senza indicizzazione**

   - *Scopo*: confrontare le performance del JOIN eseguito senza indicizzazione rispetto al test 7 dove invece era avvenuta indicizzazione preventiva, osservando ed evidenziando il notevole impatto di questa tecnica.
   - *Output atteso*: come per il test 7 il risultato del `JOIN`, ma con una tempistica che potrà risultare significativamente più lenta.
   - *Dataset*: come per il test 7, l'operazione di `JOIN` viene effettuata tra il dataset di base e quello con 30 frutti.

Per ogni test è stato definito un numero di esecuzioni variabile (con incremento progressivo a seconda della dimensione del dataset) per garantire la ripetibilità e l'affidabilità delle misurazioni. La scelta di operazioni diversificate (dalla semplice ricerca di stringhe alle complesse operazioni di join) è finalizzata a fornire un quadro esaustivo delle performance in differenti scenari applicativi, evidenziando i punti di forza e le eventuali criticità dei due diversi approcci.

4.6 Breve sintesi della progettazione

In sintesi, il presente capitolo ha definito e motivato il percorso progettuale seguito per confrontare le prestazioni di Pandas e SQL nell'elaborazione e interrogazione di dataset interamente caricabili in memoria. Gli obiettivi della tesi sono stati esplicitati attraverso l'intento di verificare se, per operazioni comuni quali ricerche, aggregazioni, indicizzazione, raggruppamenti e join, l'approccio in-memory offerto da Pandas possa risultare più rapido e flessibile rispetto al tradizionale paradigma SQL, nonché individuare eventuali limiti nelle sue prestazioni.
La metodologia adottata ha previsto la costruzione di dataset tabellari versatili – composti da 27 colonne e 1 milione di righe, con dati testuali, numerici e categoriali, per una dimensione inferiore ai 300 MB – e l'esecuzione di otto tipologie di test, ciascuno mirato a valutare operazioni specifiche. In questo contesto, è stato fondamentale definire una strategia di misurazione basata sulla raccolta e sul confronto dei tempi di esecuzione, integrando anche il tempo di caricamento in memoria per operazioni Pandas. Una volta che i dati saranno stati raccolti si procederà con la rappresentazione grafica per la fase d'osservazione.

Un ulteriore elemento rilevante è rappresentato dalle condizioni operative e dalla configurazione hardware: i test verranno condotti su un notebook HP Pavilion con un processore AMD Ryzen 7 7730U e 16 GB di RAM DDR4, in un ambiente controllato per minimizzare interferenze esterne e incongruenze accidentali. Tale impostazione permetterà di garantire la ripetibilità e la coerenza delle misurazioni, fornendo così una base solida per l'analisi comparativa.

Questa impostazione progettuale, ispirata dalla classica metodologia scientifica e dalle fonti disponibili riguardanti le prestazioni di Pandas, costituisce il fondamento per la successiva fase di implementazione, in cui verranno illustrati dettagliatamente il codice e le soluzioni tecniche adottate.

5. Implementazione

In questo capitolo verranno introdotti e discussi i dettagli implementativi che porteranno alla realizzazione degli obiettivi delineati nel capitolo precedente. Verranno esposti gli script Python che gestiscono la generazione dei dataset, elaborano i dati realizzando i grafici e che traducono gli otto test elencati in *progettazione*. Gli script dedicati alle prove si compongono di parti comuni al fine di garantire ripetibilità e continuità sia nell'esecuzione che nella comprensibilità del codice; queste parti verranno quindi discusse in un discorso d'insieme. Obiettivo di questo capitolo è quello di fornire una visione chiara e comprensibile del funzionamento degli script e dell'implementazione adottata, mettendo in evidenza i ragionamenti alla base delle scelte progettuali e i meccanismi logici che garantiscono il confronto tra le operazioni eseguite con Pandas e quelle con SQL. Senza entrare nel dettaglio specifico di ogni riga di codice, si avrà comunque una visione adeguatamente dettagliata e comprensiva, che supporterà la fase successiva di validazione sperimentale e osservazione dei risultati ottenuti.

5.1 Generazione dei tre dataset

Il primo step dell'implementazione consiste nella creazione dei dataset di prova. Lo script dedicato a questo compito definisce funzioni per generare stringhe casuali, numeri interi a 6 cifre e valori categoriali (nomi di frutti). Queste funzioni vengono utilizzate per popolare un *DataFrame* con 27 colonne, di cui:

- le prime 25 vengono popolate con stringhe di dieci caratteri casuali, usando la seguente funzione per costruire ogni stringa:

```
def random_string(length = n):
    return ''.join(random.choices(string.ascii_letters +
        string.digits, k=length))
```

- la 26ª colonna *Numbers* viene popolata con numeri casuali di sei cifre, usando la seguente funzione per generare ciascun numero compreso tra 1 compreso e 10^6 escluso:

```
def random\_number():\\
    return random.randint(1, 10**6 - 1)
```

- la 27ª e ultima colonna del *DataFrame* viene popolata scegliendo, sempre randomicamente, un frutto da un elenco predefinito (vengono generati tre *DataFrame*, rispettivamente, con un pool di 10, 30 e 50 frutti da cui scegliere).

Definite queste funzioni, esse vengono richiamate in una funziona collettiva per la creazione vera e propria del *DataFrame*. Il *DataFrame* risultante viene quindi esportato in un dataset in formato CSV di una dimensione inferiore ai 300 MB. Questo procedimento viene ripetuto per un totale di tre volte, ottenendo così tre dataset che differiscono solo per il numero di frutti presenti nella colonna *Fruit*. Il dataset di base a cui si fa riferimento se non viene specificato nulla in particolare è quello con 10 frutti, quelli da 30 e 50 vengono chiamati in causa solo nelle prove di raggruppamento e di join.

5.2 Struttura generale dei test

Per ogni test descritto nella parte di Progettazione si realizza uno script Python dedicato. Lo script tradurrà l'intento della prova con apposite funzioni per le query per Pandas e per SQL, procedendo poi a descrivere varie funzioni integrative per garantire un lavoro ordinato e metodico. Ogni script eseguirà un determinato numero di prove (vedi la sottostante "funzione che genera il numero di esecuzioni") con diverse configurazioni di dati, partendo dalle prime prove eseguite su una porzione minima del dataset, procedendo quindi considerando porzioni sempre maggiori del dataset, fino ad operare sul volume di dati per intero. Il tempo impiegato per portare a termine ogni singolo giro di operazioni (per Pandas e per SQL) verrà misurato e salvato. I risultati saranno quindi organizzati sulla base del volume di dati sul quale sono state svolte le operazioni, poi esportati su un file CSV esterno (che verrà utilizzato successivamente nella realizzazione dei grafici). Gli script composti per ogni test si basano su un modello comune, così da avere il medesimo codice per tradurre operazioni condivise e differire poi solo nelle parti specifiche, diverse per ogni singola prova. Ogni script si compone di:

- **Import & funzioni specifiche**
  All'inizio di ogni script si trova l'elenco delle librerie Python utilizzate e trovano spazio le funzioni per performare le operazioni specifiche del singolo test che verranno illustrate successivamente.

- **Creazione del numero di righe da considerare**
  Per ogni test, le operazioni verranno eseguite su un numero di righe crescente, fino al raggiungimento del volume totale del dataset (10^6 di righe). Queste

configurazioni vengono impartite tramite l'array *domain*, che verrà popolato in maniera alternata con potenze di 10 a partire da 10^1 e il valore intermedio tra la potenza del 10 attuale e quella successiva (ad esempio: 10, 55, 100, 550, 1000, ...). Viene scelta questa scala per avere un numero di righe considerato che cresce in maniera graduale, ma comunque rapida, così da evidenziare eventuali effetti sulle performance di Pandas e SQL. Il campo *nFields* indica il numero di configurazioni, per tutti i test saranno 11 (tranne per il test 8, dove saranno 10), ovvero i numeri secondo la successione sopra spiegata, dove l'ultimo elemento è 10^6 (verrà considerato il dataset nella sua interezza di 1 milione di righe).

```
nFields = 11
domain = []
power = 2
while(len(domain) < nFields):
    if(len(domain) > 0):
        if((len(domain) % 2) != 0):
            domain.append(int((pow(10, power)/2)*1.1))
        else:
            domain.append(pow(10, power))
        power += 1
    else:
        domain.append(10)
```

Con questa struttura dinamica e facilmente editabile sarà inoltre possibile modificare al bisogno la capacità delle configurazioni, per ogni evenienza o per qualsiasi altro test futuro.

- **Funzione che genera il numero di esecuzioni**
  Per ogni configurazione verrà eseguito un determinato numero di prove (a partire da 31). Affinché si possano avere dei risultati sui quali operare con indici statistici e realizzare quindi grafici che rispecchino l'andamento dei test, è necessario avere un determinato numero di tempi registrati che non sia né troppo basso né eccessivamente elevato. E' anche importante tenere in considerazione il fatto che aumentando il volume delle righe considerate ad ogni configurazione, anche il numero di prove eseguite debba tenere conto di questo andamento. A seguito di alcuni tentativi mirati ad individuare quali potessero essere dei numeri validi che garantissero solidità nei risultati, ma al contempo non rendessero l'esecuzione dei test troppo lenta, si è optato per una serie che inizia con 31 esecuzioni e così rimane fino a che le righe considerate non sono 1000. Dalla configurazione

successiva (5500 righe) in poi, il numero di esecuzioni delle prove aumenta ogni volta di 5.

```
def getNRuns(fieldChanges):
    nRuns = []
    for i in range(fieldChanges):
        if(i < 4):
            nRuns.append(31)
        else:
            nRuns.append((i+2)*5 + 1)
    return nRuns
nRuns = getNRuns(nFields)
```

L'array risultato della funzione, contenente i numeri delle prove da eseguire per ogni configurazione del test, verrà appropriatamente assegnato al campo *nRuns* a runtime (esempio: 31, 31, 31, 31, 31, 36, 41, 46).

- **Import del dataset & gestione della connessione SQL**
  In ogni script avviene il caricamento dei dati importando uno dei tre dataset, solitamente quello di default da 10 frutti. Questa operazione di *import* del dataset in un DataFrame, è l'atto di caricamento dei dati in memoria RAM idiosincratico di Pandas. Pertanto è fondamentale misurare il tempo di quest'operazione, che sarà determinata in fase di osservazione dei risultati ottenuti.

```
t1_import = time.time()
df = pd.read_csv('dfTest.csv', sep = ',')
t2_import = time.time()
pandas_import_time = t2_import - t1_import
```

Ogni script apre inoltre una connessione SQL, sfruttando la libreria sqlite3. E' quindi possibile per le operazioni SQL, caricare il DataFrame nel database mediante il metodo `to_sql()`, permettendo quindi di eseguire le query su una tabella temporanea.

- **Esecuzione delle query & registrazione del tempo**
  Un blocco di codice comune a tutti gli script lancia le prove seguendo il numero di esecuzioni per ogni configurazione. Per ogni configurazione cambia il volume dei dati preso in esame, pertanto in base al numero delle righe da considerare, viene considerata la relativa porzione del DataFrame Pandas e caricata su SQL come tabella.

Al fine di garantire performance comparabili o evidenziare differenze ritenute significative, è stato implementato un meccanismo per la creazione di indici (sia in SQL che nella struttura dati Pandas) nei test che lo richiedono. Ad esempio, lo script 4 per il calcolo del valore massimo include la creazione di una tabella indicizzata e l'applicazione di un indice sulla colonna *Numbers*, in modo da evidenziare l'impatto dell'indicizzazione sulle performance.

- **Salvataggio ed esportazione dei risultati**
  Ogni script dispone di una parte di codice in cui i risultati vengono ordinati e salvati in file esterno in formato CSV. Il file tabellare dispone sulle colonne la metodologia usata (ad esempio: SQL, Pandas, Pandas+Import), disponendo poi i tempi registrati in ordine crescente rispetto al volume di dati considerato. In questo file vengono anche annotate tutte le informazioni relative alla prova (obiettivo del test, dataset utilizzati e la loro dimensione, volume dei dati considerati nella varie configurazioni e numero di esecuzioni per ogni configurazione). Il file CSV conterrà quindi i risultati ordinati, garantendo una collezione di dati accessibile e facilmente interpretabile per la rappresentazione grafica.

Questi blocchi comuni rappresentano la spina dorsale dell'implementazione e consentono di ridurre la ridondanza nel codice, facilitando la manutenzione e la comprensione delle operazioni specifiche di ogni test. Tale struttura modulare è essenziale per assicurare la coerenza dei confronti e la ripetibilità delle misurazioni, elementi fondamentali per trarre conclusioni affidabili.

5.3 Specifiche dei test

Si provvede ora a descrivere le parti di ogni script relative alle specifiche operazioni dei test. Sebbene questa sezione sia peculiare per ogni script, alcune di esse avvengono in funzione e in correlazione con altre, al fine di raggruppare i test in insieme logici che condividono o le medesime operazioni o le cui operazioni operano in un ambiente correlato.

5.3.1 Ricerca di una stringa - Script 1

Il primo test ha lo scopo di verificare se una stringa casuale è presente all'interno del dataset. L'operazione viene eseguita sia utilizzando Pandas che SQL, e il risultato atteso è un valore booleano che indichi se la stringa è stata trovata.

– **Pandas**

```
def pd_searchString(df, string):
    return df.map(lambda x: string in str(x)).any().any()
```

La funzione `pd_searchString()` sfrutta il metodo `map()` per applicare una funzione lambda a ciascun elemento del DataFrame. La funzione converte ogni elemento in stringa e verifica se contiene la stringa ricercata. La doppia applicazione di `.any().any()` garantisce che, se almeno una cella (in una qualsiasi colonna) restituisce `True`, il risultato finale sarà `True`.

– **SQL**

```
def sql_searchString(conn, df, string):
    conditions = " OR ".join([f'"{col}" LIKE ?' for col in
        df.columns])
    query = f"SELECT * FROM df_table WHERE {conditions}"
    cursor = conn.cursor()
    cursor.execute(query, tuple([f'%{string}%'] * len(df.columns)))
    return cursor.fetchone() is not None
```

La funzione `sql_searchString()` costruisce dinamicamente una query che verifica la presenza della stringa in tutte le colonne della tabella. Ogni colonna è sottoposta a un'operazione `LIKE ?`, e la query combina le condizioni con l'operatore OR. Se almeno una riga soddisfa la condizione, la funzione restituisce `True`.

5.3.2 Raggruppamenti - Script 2 e 3

Questi test mirano a valutare le operazioni di aggregazione sui dati categoriali, raggruppando i dataset in base alla colonna `Fruit` e contando il numero di occorrenze per ciascun gruppo. In particolare, il test 2 esegue il conteggio di occorrenze per ogni gruppo, mentre il test 3 seleziona solo quei gruppi il cui numero di elementi supera una soglia prestabilita (11% del totale degli elementi).

- **script 2 - GROUP BY COUNT**
  Il test 2 esegue il `GROUP BY` sulla colonna *Fruit* e conta il numero di elementi per ciascun gruppo.

  – **Pandas**

```
def group_by_fruit(df, col_name):
```

```
    grouped = df.groupby(col_name).size()
    return grouped
```

Il metodo `groupby()` suddivide il DataFrame in base ai valori della colonna *Fruit*, e `size()` restituisce il numero di righe per ciascun gruppo.

– **query SQL**

```
SELECT "{col_name}", COUNT(*)
FROM df_table
GROUP BY "{col_name}"
ORDER BY "{col_name}"
```

La query SQL esegue l'aggregazione raggruppando per la colonna *Fruit* e conta il numero di occorrenze per ciascun gruppo, ordinando i risultati.

- **script 3 - GROUP BY HAVING COUNT**
  Il test 3 esegue sempre il `GROUP BY` per frutto e poi seleziona solo i gruppi il cui numero di occorrenze supera la soglia dell'11% del totale degli elementi.

  – **Pandas**

```
def count_having(df, col_name, threshold):
    filtered = df.groupby(col_name).filter(lambda x: len(x) >
        threshold)
    return filtered
```

In questo caso si utilizza `groupby().filter()` per mantenere solo quei gruppi la cui dimensione supera il valore di soglia. La scelta di `filter()` rispetto alla possibile alternativa fornita da Pandas `transform('size')`, è stata motivata da una maggiore efficienza riscontrata nell'applicazione della condizione; `filter()` elimina direttamente i gruppi non conformi, evitando operazioni di trasformazione intermedie e guadagnando un lieve vantaggio sui tempi misurati, nell'ordine di $2\text{x}10^{-5}$.

– **query SQL**

```
SELECT "Fruit", COUNT(*)
FROM df_table
GROUP BY "Fruit"
```

```
    HAVING COUNT(*) > {threshold}
```

- **note comuni ai due script**
  Entrambi i test vengono eseguiti su dataset con variabilità nel numero di categorie presenti nella colonna *Fruit*, per valutare l'impatto della cardinalità sulla performance delle operazioni di aggregazione. Questi test verranno quindi ripetuti per le tre dimensioni diverse della colonna dei frutti, eseguiti una prima volta per il dataset da 10, poi quello da 30, ed infine quello da 50.

5.3.3 Operazioni aritmetiche - Script 4, 5 e 6

I test 4, 5 e 6 sono orientati a valutare le operazioni di aggregazione aritmetica eseguite sulla colonna *Numbers*, specificamente il calcolo del valore massimo, minimo e della somma. Queste operazioni vengono eseguite sia in Pandas che in SQL, includendo un'ulteriore variante in cui SQL opera su una tabella indicizzata, per valutare l'impatto dell'indicizzazione sulle prestazioni. Verrà mostrato nel dettaglio solo il codice dello script 4, in quanto gli script 5 e 6 sono implementati secondo la medesima logica, modificando solo le specifiche operazioni.

- **script 4 - calcolo del valore massimo sulla colonna *Numbers***
  - **Pandas**

```
    def find_max_value_pd(Numbers):
        return Numbers.max()
```

    La funzione utilizza il metodo `max()` di Pandas, che sfrutta operazioni vettorializzate per trovare il valore massimo in maniera estremamente efficiente.

  - **query SQL**

```
    SELECT MAX(Numbers) FROM df_table
```

    La query SQL calcola il massimo valore della colonna *Numbers*. L'assenza di un indice implica una scansione completa della tabella e il tempo d'esecuzione risulterà sicuramente maggiore.

  - **SQL con indicizzazione**

```
    def create_indexed_table(conn):
        query = 'DROP TABLE IF EXISTS df_table_indexed'
        conn.execute(query)
```

```
    query = """
    CREATE TABLE df_table_indexed AS
    SELECT * FROM df_table;
    """
    conn.execute(query)


def index_table(conn):
    query = "CREATE INDEX IF NOT EXISTS
        index_Numbers_on_df_table_indexed ON
        df_table_indexed(Numbers);"
    conn.execute(query)
    conn.commit()

def index_find_max_sql(conn):
    query = f'SELECT MAX(Numbers) FROM df_table_indexed'
    cursor = conn.cursor()
    cursor.execute(query)
    result = cursor.fetchone()
    return result[0] if result else None
```

La creazione di una tabella duplicata indicizzata permette di confrontare l'effetto dell'indicizzazione. L'indice migliora la velocità di accesso e l'esecuzione della query.

- **script 5 e 6 - calcolo del valore minimo e somma**
  I test per il valore minimo e la somma seguono una logica analoga a quella del test 4, modificando il tipo di operazione:

  – Minimo: utilizzo di `min()` in Pandas e `SELECT MIN(Numbers)` in SQL.

  – Somma: utilizzo di `sum()` in Pandas e `SELECT SUM(Numbers)` in SQL.

I tre test condividono la struttura operativa, differendo soltanto nella funzione aggregata utilizzata. È fondamentale verificare come la modalità in-memory di Pandas influenzi le prestazioni rispetto alla gestione su memoria di massa in SQL, in particolare quando viene applicata l'ottimizzazione tramite indicizzazione. Presso questa ultima configurazione è atteso un risultato simile tra le varie operazioni.

5.3.4 Join tra due dataset - Script 7 e 8

Questi due test si propongono di confrontare le prestazioni delle operazioni di join tra due dataset sulla colonna *Numbers*, valutando l'impatto dell'indicizzazione. Il test 7 esegue il join su dataset in cui la colonna Numbers è stata precedentemente indicizzata, mentre il test 8 esegue il join senza indicizzazione, evidenziando le differenze in termini di tempi d'esecuzione. Per le prove verranno usati il dataset di base con 10 frutti e quello da 30.

- **script 7 - JOIN tra due dataset con indicizzazione**
  - **Pandas**

```
def join_pd(df1, df2):
    df1_indexed = df1.set_index('Numbers')
    df2_indexed = df2.set_index('Numbers')
    return pd.merge(df1_indexed, df2_indexed, left_index=True,
        right_index=True, how='inner')
```

    Le funzioni `set_index()` e `merge()` vengono utilizzate per impostare la colonna Numbers come indice e per eseguire il join, sfruttando l'ottimizzazione degli indici in Pandas.

  - **SQL**

```
def create_index_sql(conn, tb):
    query = f"""CREATE INDEX IF NOT EXISTS index_Numbers_on_{tb}
        ON {tb}(Numbers)"""
    cursor = conn.cursor()
    cursor.execute(query)
    conn.commit()

def join_sql(conn, tb1 = "df_table1", tb2 = "df_table2"):
    create_index_sql(conn, tb1)
    create_index_sql(conn, tb2)

    query = """
    SELECT t1.*, t2.*
    FROM df_table1 AS t1
    INNER JOIN df_table2 AS t2
    ON t1.Numbers = t2.Numbers
    """
    cursor = conn.cursor()
```

```
    cursor.execute(query)
    result = cursor.fetchall()
    return result
```

Il join viene eseguito in maniera standard, con la garanzia che il vantaggio dell'indicizzazione si rifletta sui tempi d'esecuzione. Si attende che la creazione di indici su entrambe le tabelle prima del join migliori notevolmente le prestazioni, consentendo un accesso rapido ai dati corrispondenti e un tempo d'esecuzione inferiore alla prova 8.

- **script 8 - JOIN tra due dataset senza indicizzazione**
  - **Pandas**

```
def join_pd(df1, df2):
    return df1.join(df2, on='Numbers', how='inner',
        lsuffix='_df1', rsuffix='_df2')

def merge_pd(df1, df2):
    return df1.merge(df2, on='Numbers', how='inner')
```

In questo caso, il join viene eseguito senza prima impostare la colonna *Numbers* come indice, permettendo di confrontare l'effetto della mancata indicizzazione. Oltre alla funzione `join()`, viene eseguita una prova anche con la funzione `merge()`, così da evidenziare possibile differenze tra i due metodi.

  - **query SQL**

```
SELECT t1.*, t2.*
FROM df_table1 AS t1
INNER JOIN df_table2 AS t2
ON t1.Numbers = t2.Numbers
```

La query SQL per il join eseguito senza indicizzazione si basa sulla stessa logica di quella usata nel test 7, ma in assenza della creazione preventiva degli indici, il motore di database deve eseguire una scansione completa delle tabelle, pertanto è atteso un aumento dei tempi di esecuzione.

L'obiettivo dei test 7 e 8 è quello di evidenziare il vantaggio apportato dall'indicizzazione su operazioni di join. Nei test con indicizzazione (script 7) si attende un notevole

miglioramento dei tempi d'esecuzione, sia in Pandas che in SQL, rispetto a un'operazione senza tale ottimizzazione (script 8). La scelta di implementare due versioni per l'operazione di join consente di isolare l'impatto dell'indicizzazione e di valutare come questa tecnica possa ridurre il carico computazionale, in particolare quando il volume di dati considerato inizia ad avere dimensioni considerevoli.

5.4 Plotting dei risultati

Un altro componente fondamentale dell'implementazione è lo script per il plotting, che si occupa di elaborare i file CSV contenenti i risultati dei test, calcolare gli indici statistici e visualizzare i dati. La parte relativa agli indici statistici considerati e le motivazioni che hanno portato alla loro scelta, verrà discussa nel capitolo successivo di validazione sperimentale. Dopo aver letto i dati e applicato i filtri statistici, avvalendosi della libreria matplotlib (20), vengono creati i grafici. I grafici riportano i tempi d'esecuzione (per le operazioni eseguite con SQL, Pandas e la versione di Pandas che include il tempo di importazione, più eventuali misurazioni peculiari del test) in relazione al volume di dati su cui è avvenuta l'operazione. Lo script infine, esegue l'esportazione del grafico generato, sottoforma di immagine PNG. L'obiettivo è quello di tradurre i dati numerici in rappresentazioni visive che consentano l'interpretazione dei risultati e la valutazione delle performance, mantenendo una continuità logica con le metodologie descritte nei capitoli precedenti e garantendo leggibilità.

5.5 Sunto conclusivo dell'implementazione

In questo capitolo sono stati illustrati nel dettaglio gli script utilizzati per eseguire le prove sui dataset. In particolare, lo script di generazione dati ha prodotto tre dataset composti da 1'000'000 di righe e 27 colonne: le prime 25 contengono stringhe casuali di dieci caratteri, la 26ª (*Numbers*) ospita numeri interi casuali a sei cifre, mentre la 27ª (*Fruit*) è popolata con nomi di frutti selezionati casualmente da un pool rispettivamente, di 10, 30 e 50 elementi nei tre dataset. Successivamente, sono stati descritti gli otto script dedicati ai test. Ciascuno di essi è strutturato a grandi linee in due parti: una sezione comune, condivisa tra tutti gli script, e una sezione specifica, che implementa la logica della singola prova, analizzata in modo approfondito. L'esecuzione di questi script genera file CSV contenenti i tempi di esecuzione registrati, che verranno poi elaborati dallo script di plotting (dove vengono applicati i filtri statistici) per la realizzazione di grafici. Questi ultimi saranno fondamentali per l'analisi dei risultati, che verrà affrontata nel prossimo capitolo, dedicato alla validazione sperimentale.

6. Validazione Sperimentale

Nel capitolo precedente sono stati presentati in dettaglio gli script di implementazione, descrivendo sia le parti comuni sia le specifiche logiche di ciascun test. Ora si passa alla validazione sperimentale, in cui verranno discusse le modalità con cui sono state effettivamente condotte le prove, l'ambiente operativo adottato, la scelta degli indici statistici e le metriche utilizzate per rappresentare i risultati. Le analisi riguarderanno gli otto test descritti in precedenza, raggruppati in modo coerente (ad esempio, le operazioni di join con o senza indicizzazione verranno confrontate insieme), così da consentire una lettura più agevole dei risultati e un confronto immediato tra le diverse configurazioni. Al termine della presentazione dei grafici, sarà proposta una discussione finale che metterà in relazione i risultati sperimentali con le aspettative teoriche, fornendo spunti sulle potenziali applicazioni e sui limiti di ciascun approccio (Pandas o SQL).

6.1 Organizzazione delle prove

Per ognuna delle otto prove sperimentali è stato realizzato uno script Python che traduca i test in operazioni Pandas e SQL. Le prove vengono eseguite su un notebook HP (16 GB di RAM, Processore: AMD Ryzen 7 7730U)in un ambiente controllato, per minimizzare le interferenze e garantire risultati quanto più possibile coerenti, come discusso nel capitolo di progettazione. Si ricorda che questo accorgimento è necessario al fine di ridurre ogni rischio d'interferenza esterna, che potrebbe alterare i tempi d'esecuzione.

Ciascun test (tradotto in una o più operazioni Pandas e una o più operazioni SQL) viene ripetuto su un volume di dati via via crescente, partendo da un piccolo sottoinsieme del dataset e arrivando progressivamente a includere l'intero milione di righe (per i dataset di riferimento). A ogni passaggio, il numero di esecuzioni effettuate è inizialmente pari a 31 fino a un massimo di 1000 righe; oltre tale soglia, aumenta di 5 unità per ogni nuova configurazione (arrivando, ad esempio, a 61 run sulla configurazione massima da 1.000.000 di righe). Per ogni blocco di esecuzioni, il primo risultato non viene salvato, poiché risente di eventuali operazioni iniziali (caching, warm-up) e risulta statisticamente poco significativo. Analogamente, ulteriori *outliers* verranno prontamente rimossi in fase di analisi, grazie allo script di plotting. Al termine di ogni esecuzione, i tempi rilevati vengono salvati in un file CSV, così da poter essere successivamente letti e rielaborati per la rappresentazione grafica. I test sono stati eseguiti più volte in condizioni analoghe, scegliendo infine la serie di misurazioni considerata più attendibile, cioè priva di anomalie evidenti o di picchi di carico esterni all'ambiente dei test.

6.2 Script di plotting & Indici statistici

Una volta concluse le esecuzioni dei test, i risultati vengono raccolti in file CSV che contengono i tempi di esecuzione per ogni configurazione sperimentale (ad esempio, diverse dimensioni del dataset e numero di ripetizioni). Per elaborare questi dati in modo sistematico e rappresentarli graficamente, è stato sviluppato uno script di plotting che si occupa di:

- **Lettura e raggruppamento dei dati:**
  Lo script legge i file CSV e associa i tempi di esecuzione (sia per Pandas che per SQL, ed eventuali varianti come Pandas+Import) alla relativa configurazione di test. Questo passaggio consente di costruire una struttura dati omogenea, in cui ogni configurazione sperimentale (numero di righe, numero di run, ecc.) è associata ai tempi corrispondenti.

- **Rimozione degli outlier:**
  Gli outlier sono valori che si discostano in modo significativo dalla maggior parte dei dati e possono dipendere da fattori contingenti (come processi di sistema improvvisi o ritardi casuali nella gestione della memoria). Nel nostro script, la rimozione avviene applicando una soglia di 3 deviazioni standard:

  $$|x - \mu| > 3\sigma \tag{6.1}$$

  dove μ è la media dei tempi di esecuzione e σ è la deviazione standard. Quest'ultima misura quanto i valori si discostano, in media, dalla media stessa. In pratica, se un valore è più lontano di 3σ dalla media, viene considerato anomalo e rimosso.

- **Calcolo della media, della varianza e dell'intervallo di confidenza:**

  - *Media* (μ): è la somma di tutti i tempi di esecuzione divisa per il numero di valori considerati, e rappresenta una stima del "tempo medio" per quel tes, definita come:

    $$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{6.2}$$

  - *Varianza* (σ^2): è la media dei quadrati degli scarti dalla media, ovvero:

    $$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2 \tag{6.3}$$

Indica quanto i dati si distribuiscono intorno alla media; una varianza elevata segnala che i valori sono molto dispersi.

- *Deviazione standard* (σ): è la radice quadrata della varianza, $\sigma = \sqrt{\sigma^2}$. Se i valori dei tempi si distribuiscono in maniera stretta intorno alla media, la deviazione standard sarà piccola; in caso contrario, sarà più elevata.

- *Intervallo di confidenza*: per ogni media, lo script calcola un intervallo di confidenza al 99% basato sulla distribuzione t di Student, che fornisce una stima dell'incertezza legata alla media campionaria. Se l'intervallo è ampio, significa che la stima della media potrebbe variare sensibilmente; se è stretto, i tempi di esecuzione si concentrano intorno a un valore ben definito.

- **Produzione dei grafici:**
  Lo script utilizza infine la libreria matplotlib per tracciare i valori medi dei tempi di esecuzione (dopo la rimozione degli outlier) in funzione delle varie configurazioni sperimentali (numero di righe, numero di run, ecc.). Sull'asse delle ordinate (Y) vengono riportati i tempi di esecuzione, mentre sull'asse delle ascisse (X) compaiono le configurazioni di test. A fianco di ogni punto, può essere mostrata una barra d'errore corrispondente all'intervallo di confidenza, rendendo immediata la visualizzazione dell'incertezza statistica associata ai risultati.

L'adozione di queste tecniche statistiche (rimozione degli outlier, calcolo di media, varianza e intervalli di confidenza) viene impiegata per ottenere una visione quanto più obiettiva e robusta dei dati raccolti, mitigando l'effetto di eventuali anomalie e fornendo al contempo una misura dell'affidabilità delle medie stimate. I grafici prodotti consentono, quindi, di confrontare già visivamente le prestazioni di Pandas e SQL, evidenziando possibili differenze o definendo eventuali similitudini al variare delle dimensioni del dataset e delle tipologie di operazione.

6.3 Formato dei grafici

La metrica principale considerata è il tempo di esecuzione (in secondi) per ciascun test, misurato con il modulo `time` di Python. Sull'asse delle ascisse (X) vengono riportate le varie configurazioni, corrispondenti ai differenti volumi di dati analizzati: tipicamente si parte da poche decine di righe fino a raggiungere il milione di righe del dataset. Poiché i passaggi di scala non sono uniformi (si possono avere configurazioni come 100, 1000, 10'000, 100'000, ecc.), l'asse X potrebbe non riflettere una progressione lineare, fattore che non è strettamente rilevante ai fini di confrontare le prestazioni. A fianco di ogni

valore sull'asse X, talvolta, è riportato anche il numero di esecuzioni effettuate tra parentesi, per indicare quante volte si è ripetuto il test a quella specifica configurazione.

Sull'asse delle ordinate (Y) viene rappresentato il tempo medio (dopo la rimozione degli `outliers`) per completare l'operazione, con una barra di errore che corrisponde all'intervallo di confidenza calcolato. In questo modo, se due linee (ad esempio, Pandas e SQL) risultano vicine e i loro intervalli di confidenza si sovrappongono, significa che le differenze di performance potrebbero non essere significative dal punto di vista statistico. Al contrario, se un intervallo non si sovrappone all'altro, è probabile che esista una reale divergenza tra i due metodi.

Nella maggior parte dei grafici riportati, si preferisce non includere la variante Pandas + Import nel medesimo piano, in quanto il tempo aggiuntivo di caricamento del DataFrame può distorcere la scala, rendendo meno visibili le differenze tra SQL e il tempo di esecuzione "puro" di Pandas. Tali valori, comunque, sono stati registrati e possono essere esaminati separatamente, laddove si voglia valutare l'eventuale impatto di una fase di caricamento in memoria su un flusso di lavoro complessivo, e verranno infatti presi in esame al momento della discussione finale.

In ultima nota, le misurazioni e i test vengono effettuati su un volume di dati crescente per ogni configurazione, fino a comprendere i dataset per intero (1'000'000 di righe). Nella rappresentazione grafica è possibile però che non tutte le misurazioni trovino spazio, come spesso accadrà, ad esempio, per le ultime 60 esecuzioni relative ai dataset considerati per intero, le quali non verranno tracciate, poiché presentano spesso elevati problemi di consistenza dei dati (si osserva un fenomeno d'appiattimento dei tempi registrati, che non rispecchia il tasso d'incremento suggerito dalle misurazioni delle configurazioni precedenti, probabilmente dovuto ad un numero eccessivo di `outliers`). Alcuni grafici inoltre, non si spingeranno oltre la configurazione da 100'000 righe, al fine di evidenziare le differenze tra i tempi di Pandas e SQL in modo accessibile, qualora una delle due metodologie inizi a divergere dall'altra e il ritardo ad essere consistentemente più grande, si perderebbe precisione e dettaglio nella scala. In questi casi, si mostrerà quindi solo la porzione dei risultati che verrà ritenuta più rilevante da osservare ed interessante in ottica di trarre le relative conclusioni.

6.4 Risultati delle prove

In questo paragrafo vengono presentati e analizzati i risultati sperimentali ottenuti dall'esecuzione dei vari test sviluppati. Ogni test è stato valutato attraverso la raccolta dei tempi di esecuzione in varie configurazioni, misurati in condizioni controllate. Test che condividono similitudini (come test 7 e test 8, entrambi focalizzati sul join tra dataset) verranno analizzati insieme quanto più possibile, come già spiegato all'inizio del capitolo.
I dati ottenuti sono stati rielaborati mediante indici statistici (media, varianza, deviazione standard e intervallo di confidenza al 99%) e verranno esposti tramite grafici, consentendo così di confrontare in maniera immediata le performance di Pandas e SQL. Si ricoda che i grafici riportano, sull'asse delle ascisse, le diverse configurazioni in termini di volume di dati (numero di righe) e, sull'asse delle ordinate, il tempo medio di esecuzione (espresso in secondi). Le barre verticali d'errore rappresentano gli intervalli di confidenza, offrendo una misura dell'incertezza associata alle misurazioni. In questo modo, è possibile confrontare non solo le performance medie, ma anche la stabilità delle misurazioni tra i due approcci, alla luce delle aspettative teoriche e delle evidenze raccolte tra gli articoli e le ricerche disponibili a riguardo.
Il tempo di caricamento del dataset in memoria verrà discusso e integrato nelle conclusioni - anche sotto forma di grafico, ma non verrà mostrato nel grafico di principale riferimento di ogni test, poiché causerebbe un allargamento della scala di rappresentazione e una conseguente perdita di precisione, rilevante ai fini delle osservazioni basate sui grafici.

6.4.1 Test 1 - Ricerca di una stringa

Il Test 1 si focalizza sull'efficienza nel cercare una stringa casuale all'interno del dataset, utilizzando sia l'approccio in-memory di Pandas sia l'esecuzione di una query SQL.

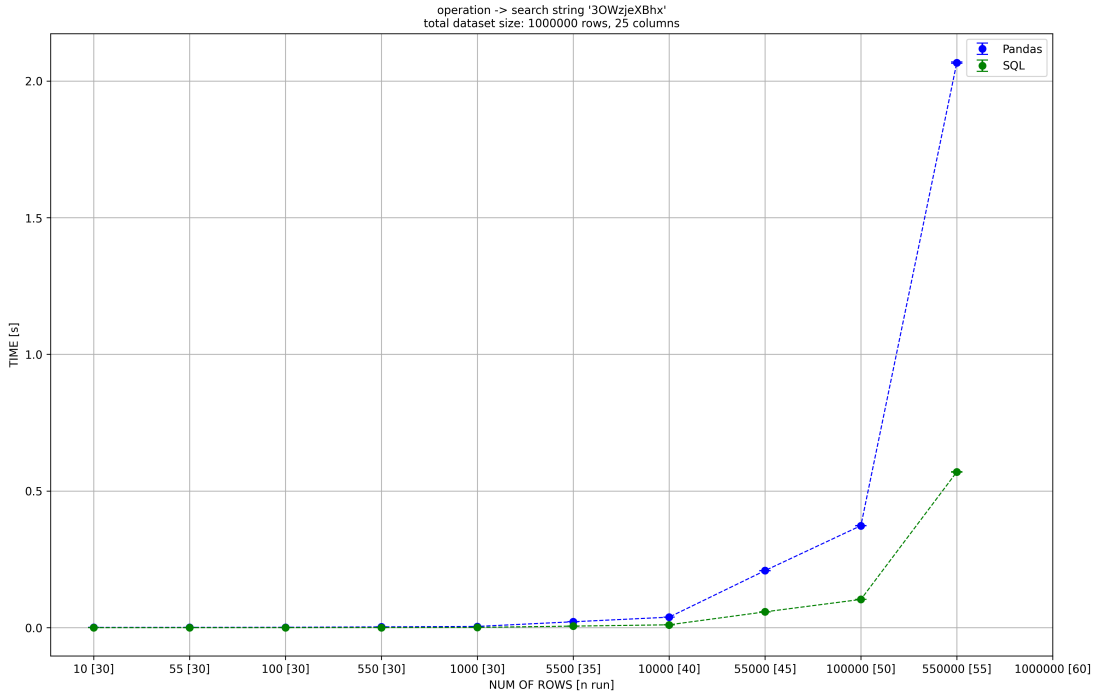


Figura 6.1. Grafico del test 1

Dall'analisi dei dati raccolti, si osserva che per dataset di piccole dimensioni (nell'ordine di poche centinaia di righe) la differenza di tempi è praticamente trascurabile (circa 10^{-4} secondi). Tuttavia, con l'aumentare del volume dei dati, la differenza diventa rilevante: ad esempio, per una configurazione in cui sono state considerate circa 550'000 righe, il tempo medio d'esecuzione della ricerca tramite Pandas supera quello registrato con SQL di oltre 1 secondo netto.

La funzione di ricerca implementata in Pandas, che utilizza `map()` e il doppio `.any().any()`, richiede di scorrere ogni singolo elemento del DataFrame per verificare la presenza della stringa, il che comporta un incremento sostanziale dei tempi quando il numero di righe cresce significativamente. Al contrario, la funzione SQL, che costruisce dinamicamente una query con condizioni `LIKE` su ogni colonna, beneficia delle ottimizzazioni interne del motore del database e mostra tempi di esecuzione più scalabili.

Un ulteriore elemento evidenziato dai risultati riguarda il tempo di importazione del dataset in memoria (misurato separatamente e riportato in appendice, nella sezione dedicata ai grafici). In questo caso particolare, il tempo impiegato per caricare il dataset in memoria è stato di 6.56 secondi, un contributo notevole rispetto al tempo totale sull'utilizzo di Pandas che aggrava chiaramente il divario rispetto all'approccio SQL.

In sintesi, i dati sperimentali indicano chiaramente che, per la ricerca di stringhe su dataset di grandi dimensioni, l'approccio SQL risulta significativamente più efficiente in termini di tempo di esecuzione rispetto a Pandas, confermando così le ipotesi iniziali basate sulla letteratura e sulle evidenze raccolte.

6.4.2 Test 2 e 3 - Raggruppamenti per frutto

- **Test 2 - Raggruppamento e conteggio**
  Il Test 2 valuta l'operazione di raggruppamento per la colonna *Fruit* e il conteggio degli elementi per ciascun gruppo (in SQL: `GROUP BY e COUNT`). Nei grafici, l'asse delle ascisse rappresenta il numero di righe considerate (che va da 10 a 550'000) e, sull'asse delle ordinate, vengono riportati i tempi medi di esecuzione.

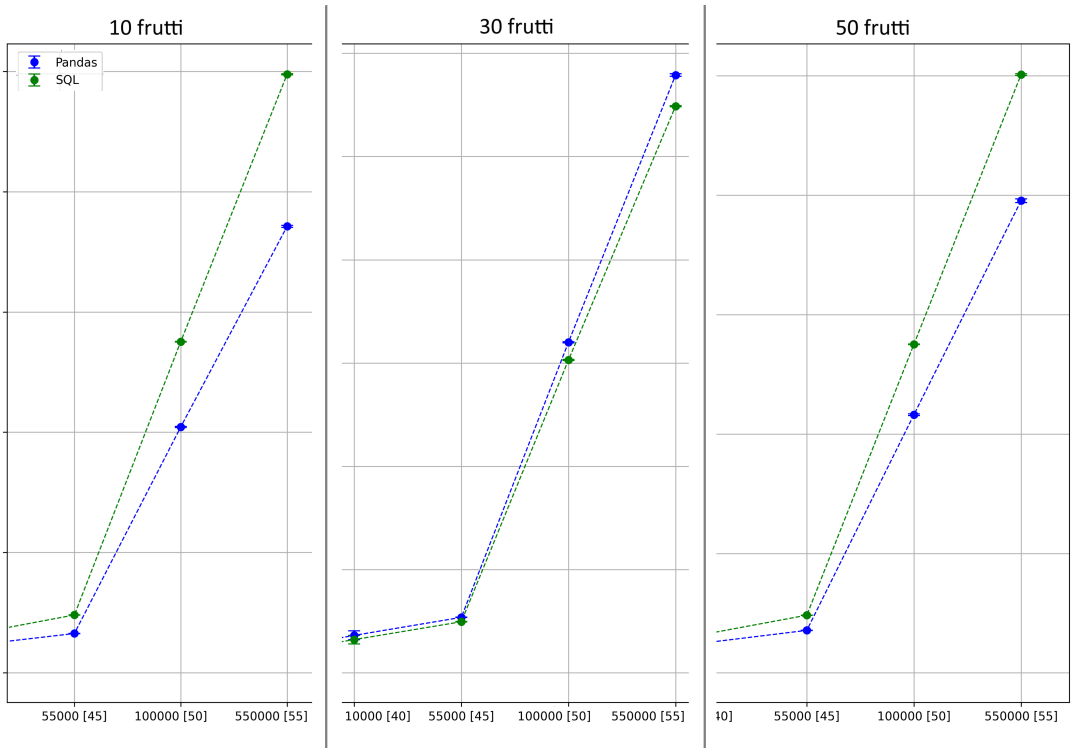


Figura 6.2. Highlight sulla parte finale dei tre grafici del test 2

I risultati mostrati evidenziano un andamento interessante:

- **Configurazioni con 10 frutti e con 50 frutti**:
  In queste configurazioni, soprattutto per volumi di dati elevati (da 10'000 righe in poi), i tempi medi registrati con Pandas risultano inferiori a quelli misurati con SQL. Questo indica che, per pool di categorie molto bassi o

molto elevati, l'approccio in-memory di Pandas sfrutta in modo efficace le sue ottimizzazioni interne, garantendo prestazioni migliori.

– **Configurazione con 30 frutti**:
In questo caso, i grafici indicano che Pandas è leggermente più lento di SQL, con una differenza che si manifesta nell'ordine dei millesimi di secondo. Tale comportamento potrebbe suggerire l'esistenza di una soglia critica intermedia, in cui il meccanismo di raggruppamento in-memory di Pandas subisce una lieve flessione in termini di ottimizzazione, rendendo SQL leggermente più performante.

- **Test 3 - Raggruppamento con filtraggio condizionale**
Il Test 3 estende l'operazione di raggruppamento introducendo una condizione che seleziona solo i gruppi il cui numero di occorrenze supera l'11% del totale degli elementi (in SQL: `GROUP BY e HAVING COUNT`). Anche qui sono state testate le tre configurazioni in termini di pool di frutti (10, 30 e 50).

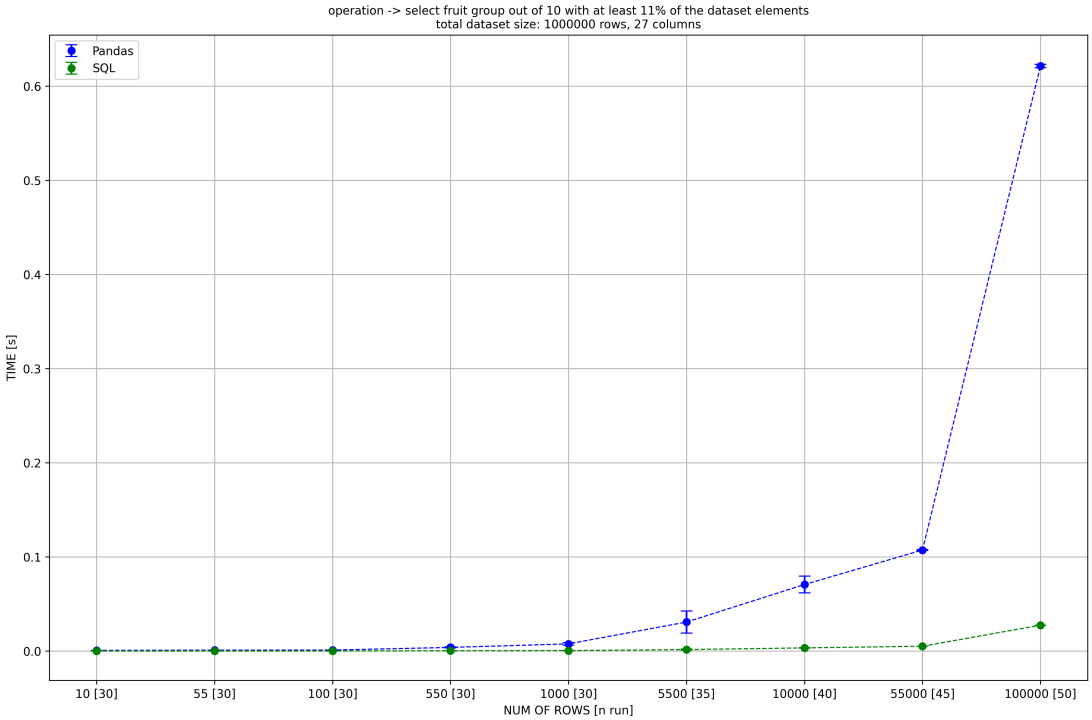


Figura 6.3. Grafico del test 3 eseguito con 50 frutti

In questo caso i risultati in tutte le configurazioni (10, 30 e 50 frutti), mostrano come SQL risulti costantemente più veloce di Pandas. In particolare, per il caso

con 30 frutti – in cui, nel Test 2, la differenza era minima – l'operazione con filtro evidenzia un gap significativo, con tempi medi per SQL inferiori di un termine pari a 0.5 secondi rispetto a quelli misurati con Pandas, in maniera consistente lungo le tre diverse configurazioni del pool di frutti. Questi risultati suggeriscono che l'applicazione del filtro (con la clausola `HAVING` in SQL) permette al motore di database SQL di eseguire in modo estremamente efficiente l'operazione, mentre l'approccio in-memory di Pandas, che richiede di iterare sui gruppi per applicare il filtro, risente maggiormente dell'incremento della complessità operativa.

Considerazioni Finali e Confronto Complessivo Nei risultati del Test 2, che valuta il raggruppamento per la colonna *Fruit* e il conteggio degli elementi per ciascun gruppo senza applicare ulteriori filtraggi, si osserva che, con dataset di dimensioni crescenti (da 10 a 550'000 righe), l'approccio in-memory di Pandas si dimostra generalmente competitivo, anzi in alcune configurazioni (quali quelle con 10 o 50 frutti) i tempi medi risultano inferiori rispetto a SQL. Tuttavia, occorre precisare che nella configurazione con 30 frutti, si evidenzia una lieve flessione nelle prestazioni di Pandas, con tempi medi leggermente superiori a quelli di SQL, in termini di una differenza dell'ordine dei millesimi di secondo. Questo comportamento suggerisce l'esistenza di una soglia critica intermedia in cui l'ottimizzazione interna di Pandas subisce un lieve impatto, pur mantenendo comunque prestazioni efficienti.

Nel Test 3, che estende l'operazione di raggruppamento includendo un filtro condizionale (selezionando solo i gruppi con un numero di occorrenze superiore all'11% del totale), il grafico mostra una situazione in cui SQL risulta sistematicamente più veloce di Pandas. In particolare, per configurazioni con volumi elevati (fino a 550'000 righe) la differenza può arrivare a circa 0.5 secondi, con SQL che sfrutta in modo più efficiente la clausola `HAVING` e le ottimizzazioni interne del motore di database, mentre l'approccio di Pandas – che deve iterare sui gruppi per applicare il filtro – subisce un ritardo notevole.

Un ulteriore elemento di confronto riguarda il tempo di importazione del dataset in memoria, che nei test effettuati si attesta mediamente intorno ai 6.75 secondi. Sebbene questo costo sia determinante se si considera una singola operazione, è bene ricordare che esso rappresenta un costo fisso che si ammortizza nel caso in cui si operino numerose esecuzioni sullo stesso DataFrame, in modo da ridurre così il suo impatto sul confronto complessivo delle operazioni.

In sintesi, questi test evidenziano come l'efficacia dei due approcci dipenda fortemente dalla complessità dell'operazione di raggruppamento e filtraggio. Se il raggruppamento è semplice, Pandas può risultare competitivo – anzi, nella configurazione qui testata si dimostra più rapido di SQL (eccezion fatta per una lieve depressione, attestata

intorno ai 30 gruppi, dovuta probabilmente all'interessamento di una soglia critica intermedia del meccanismo di raggruppamento). L'aggiunta di condizioni di filtro vede SQL attestarsi come metodologia più rapida e in grado di garantire prestazioni significativamente superiori; tuttavia, è bene evidenziare come nel raggruppamento e nel semplice conteggio degli elementi per gruppo da un pool di massimo 50 elementi, Pandas abbia registrato performance notevoli e più veloci di SQL, confermando quindi la necessità di scegliere lo strumento in funzione della complessità operativa richiesta al bisogno.

6.4.3 Test 4, 5, e 6 - Operazioni aritmetiche sulla colonna *Numbers*

Nei Test 4, 5 e 6 sono state valutate tre operazioni aritmetiche fondamentali sulla colonna *Numbers*: il calcolo del massimo, del minimo e della somma totale. Per ciascuna operazione, i grafici sono stati costruiti utilizzando come asse delle ascisse le configurazioni del numero di righe (10, 55, 100, 550, 1000, 5500, 10000, 55000, 100000, 550000) e come asse delle ordinate il tempo medio d'esecuzione.

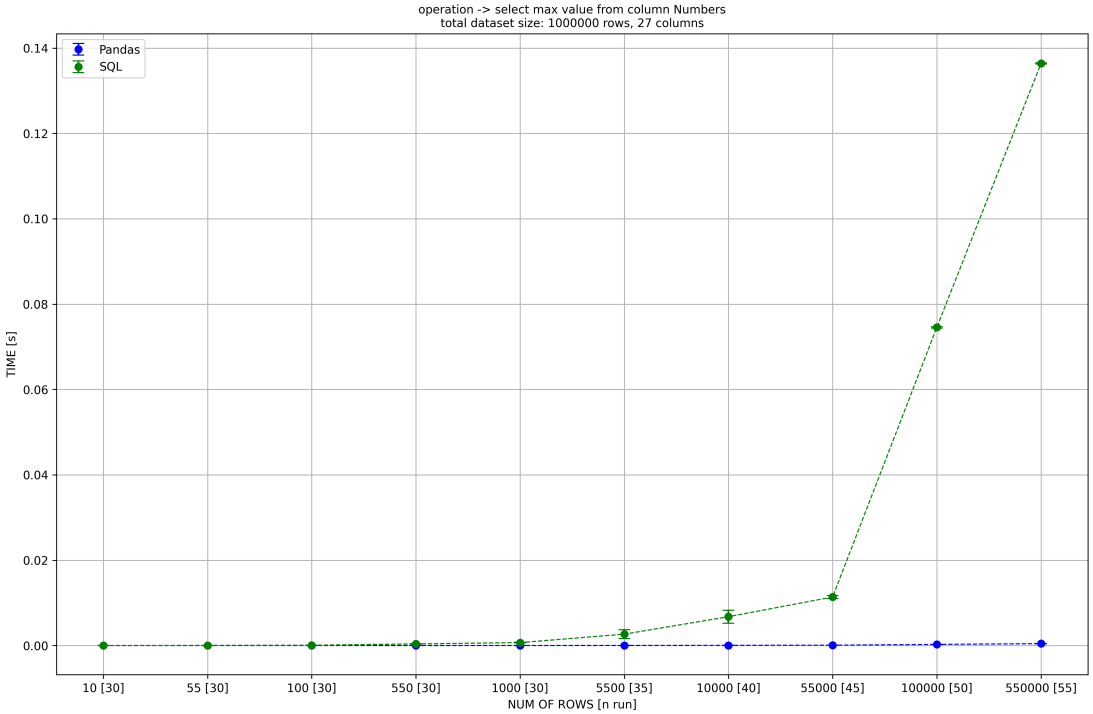


Figura 6.4. Grafico dei risultati del test 4 - Pandas e SQL

Analizzando i risultati relativi a Pandas (senza includere il tempo di importazione) e SQL (senza indicizzazione), si evidenzia quanto segue:

- **Test 4 - calcolo del massimo**
  In tutte le configurazioni, i tempi medi di esecuzione di Pandas sono molto bassi (nell'ordine di 10^{-5} secondi) e aumentano in modo molto contenuto al crescere del volume dei dati. SQL, invece, presenta tempi iniziali comparabili, ma con l'aumentare del numero di righe, la sua esecuzione risulta sensibilmente più lenta, con tempi che raggiungono, in alcune configurazioni, decimi di secondo. Questo ritardo può essere attribuito alla necessità per il motore SQL di effettuare una scansione completa della tabella su supporti di memoria di massa, senza poter sfruttare le ottimizzazioni in-memory tipiche di Pandas.

- **Test 5 e 6 - calcolo del minimo e della somma**
  Analogamente al calcolo del valore massimo, i risultati del calcolo del minimo e della somma mostrano come Pandas mantenga tempi d'esecuzione stabili e molto contenuti, con variazioni minime al variare del volume dei dati; contrariamente SQL, eseguendo la stessa operazione senza alcuna indicizzazione preventiva, registra tempi crescenti con l'incremento del numero di righe, con una differenza che si spinge fino ad alcuni decimi di secondo nelle configurazioni con un volume di dati considerato più elevato. I risultati due test risultano estremamente simili a quelli del test 4, pertanto per queste osservazioni si può tranquillamente fare riferimento allo stesso grafico (Fig. 6.4).

Nel complesso, confrontando queste due metodologie (Pandas senza import vs SQL senza indicizzazione), i grafici mostrano che SQL risulta notevolmente più lento. I ritardi, espressi in decimi di secondo, diventano sempre più evidenti man mano che il volume dei dati aumenta; ciò conferma l'ipotesi che l'accesso diretto a supporti di memoria di massa, senza ottimizzazioni specifiche, penalizzi l'esecuzione delle operazioni aritmetiche in SQL rispetto all'approccio in-memory di Pandas. La lentezza di SQL in termini di performance è sicuramente dovuta la fatto che per ricercare i valori viene scansionata ogni volta l'intera tabella.
Per arginare questo fenomeno, si rieseguono le prove, provvedendo, per quanto riguarda la metodologia SQL, ad eseguire un'indicizzazione preventiva sulla colonna *Numbers*, così da facilitare il lavoro del motore SQL e ottenere un miglioramento dei risultati. In questo scenario si osserva quanto segue:

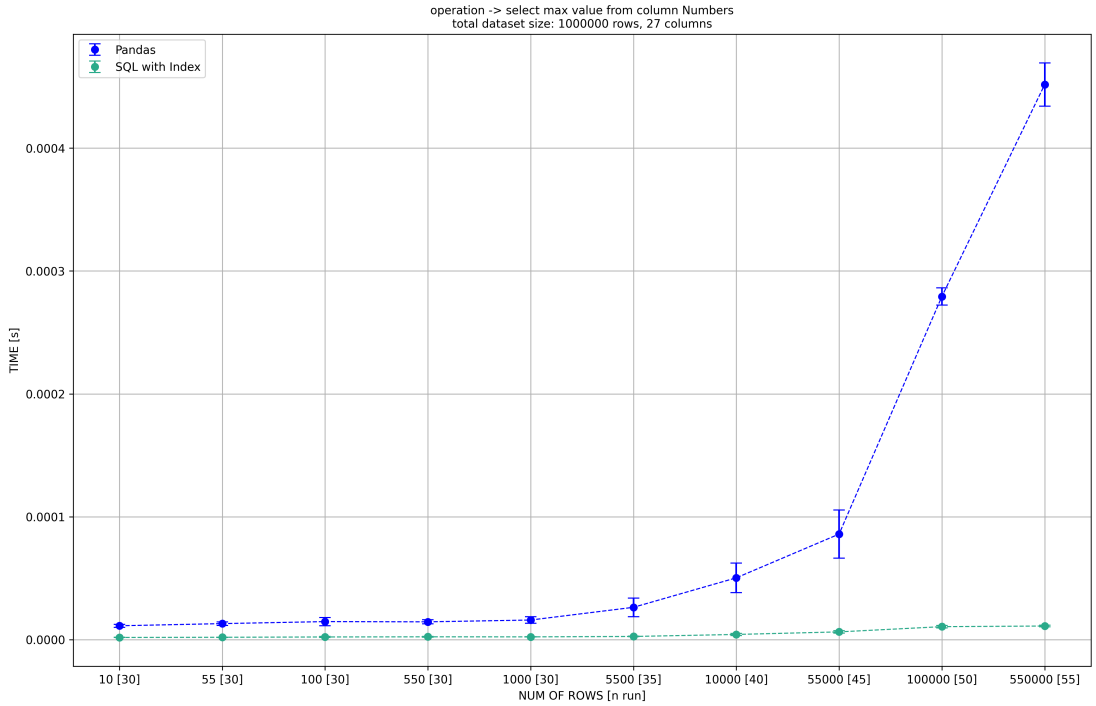


Figura 6.5. Test 4 - Pandas e SQL con indicizzazione

- **Test 4 - calcolo del massimo**
  L'indicizzazione porta SQL a eseguire l'operazione in tempi dell'ordine dei microsecondi (10^{-6} secondi), addirittura risultando più veloce di Pandas. Una possibile ipotesi per questo risultato è che il motore SQL, grazie all'indice, possa accedere direttamente all'ultimo elemento ordinato (dato che l'indice è tipicamente strutturato in modo ordinato), evitando così una scansione completa della tabella. In molti motori SQL, gli indici B-Tree sono strutturati in modo tale da permettere un accesso diretto ai valori estremi (min/max) con una complessità $O(1)$ o $O(\log n)$, a differenza di una scansione completa che sarebbe $O(\mathrm{n})$.

- **Test 5 e 6 - calcolo del minimo e della somma**
  Queste due operazioni condividono risultati estremamente simili e per questo verranno discussi insieme, facendo riferimento al grafico del calcolo del minimo (Fig. 6.6). Come per la prova precedente, anche in questi test l'uso dell'indicizzazione migliora notevolmente le prestazioni di SQL rispetto alla versione non indicizzata, riducendo i tempi d'esecuzione in maniera consistente. Tuttavia, in entrambi questi casi, a differenza del test 4, Pandas continua a risultare più veloce. Anche se la differenza di tempo si riduce notevolmente rispetto allo scenario senza

indicizzazione, rimane comunque rilevante, attestandosi sui decimi di secondo.

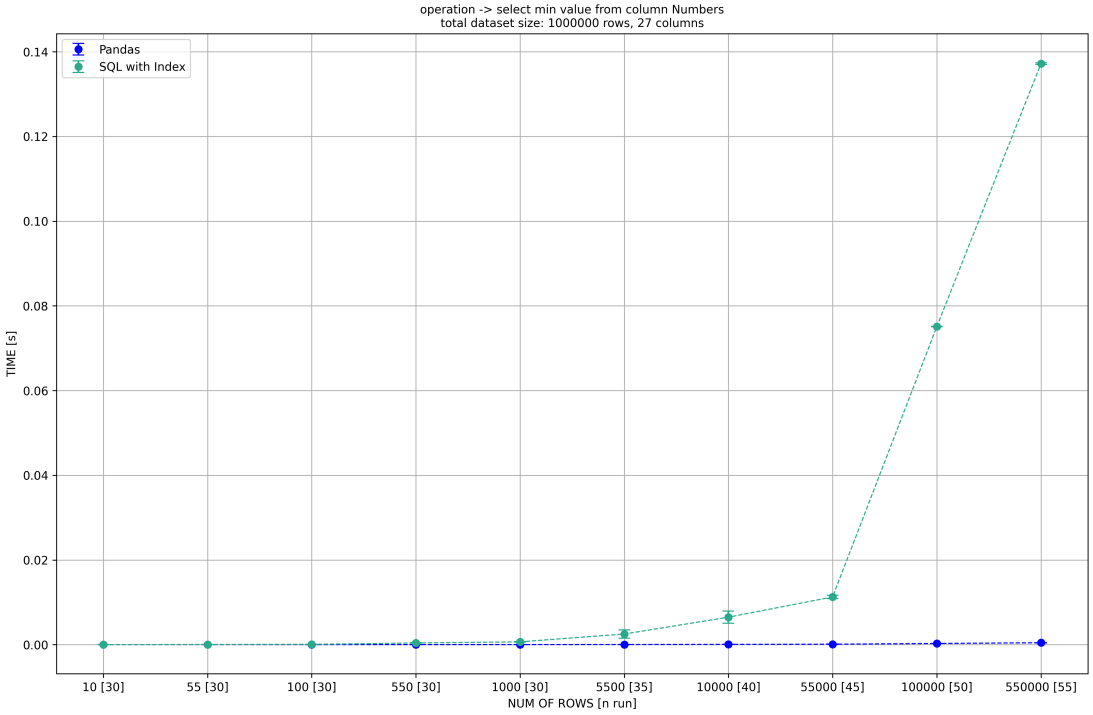


Figura 6.6. Test 5 - Pandas e SQL con indicizzazione

Queste prove dimostrano come l'indicizzazione migliori incredibilmente le prestazioni di SQL, rendendole competitive, ma non del tutto superiori a quelle di Pandas, ad eccezione del caso del calcolo del massimo.

Considerazioni Finali e Confronto Complessivo In generale, i grafici derivanti dai test evidenziano come, per operazioni aritmetiche su dataset ampi, l'approccio in-memory di Pandas mantenga tempi di esecuzione molto stabili e praticamente invarianti al crescere del volume dei dati. In contrasto, SQL, senza ottimizzazioni, tende a rallentare sensibilmente a causa della necessità di eseguire scansioni complete del dataset. L'applicazione dell'indicizzazione preventiva sulla colonna *Numbers* migliora notevolmente le prestazioni di SQL: in particolare, nel calcolo del massimo SQL indicizzato risulta addirittura più veloce di Pandas, mentre per il calcolo del minimo e la somma, sebbene SQL migliori rispetto alla versione non indicizzata, rimane ancora leggermente in ritardo rispetto a Pandas. Queste osservazioni suggeriscono che l'efficacia dell'approccio SQL dipenda fortemente dalla possibilità di sfruttare strutture dati ordinate (come gli

indici) per operazioni specifiche; mentre Pandas si dimostra estremamente efficiente per operazioni aritmetiche grazie alle sue funzioni ottimizzate, che non subiscono variazioni sostanziali man mano che aumenta il numero di righe. L'approccio vettoriale e l'uso di librerie C sottostanti (come BLAS e LAPACK) permettono di mantenere tempi di esecuzione quasi costanti.

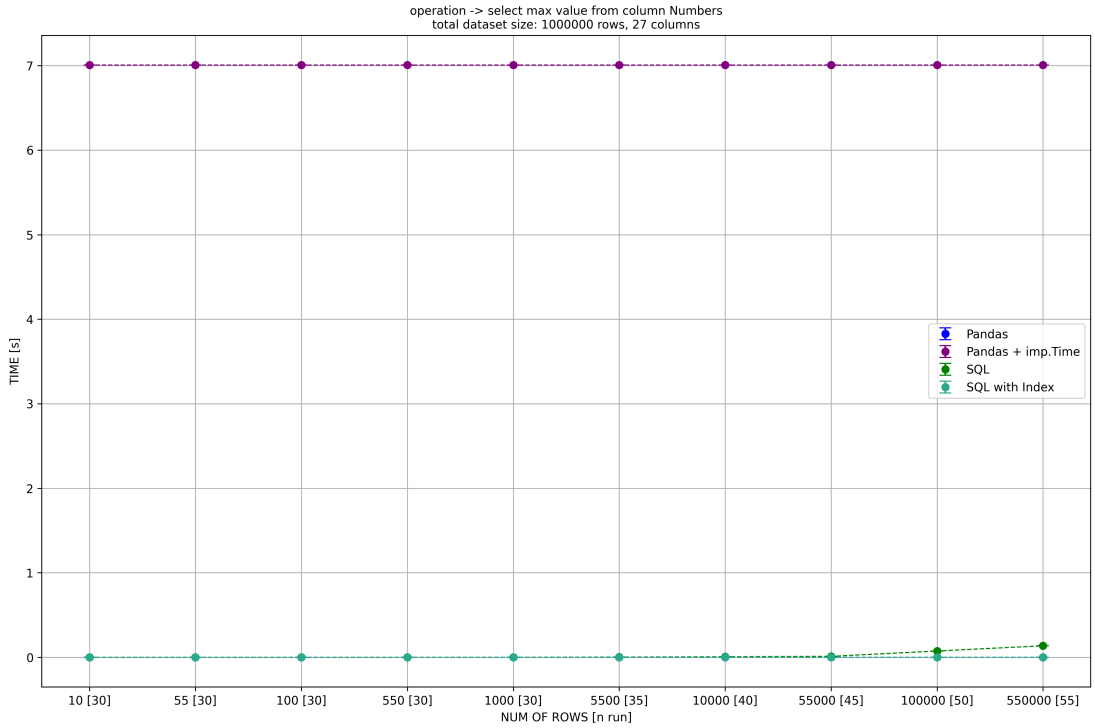


Figura 6.7. Test 4 - Pandas import time a confronto

Infine, va ricordato che il tempo di importazione del dataset (mediamente intorno ai 7 secondi), non incide direttamente nei confronti delle operazioni aritmetiche vere e proprie. Si tratta di un costo fisso che, sebbene vada considerato, viene ammortizzato quando si eseguono numerose operazioni sullo stesso DataFrame, e non impatta direttamente sulle performance delle singole operazioni aritmetiche.

6.4.4 Test 7 e 8 - Operazioni di join tra due dataset

I Test 7 e 8 sono stati progettati per analizzare l'impatto dell'indicizzazione sulla velocità di esecuzione delle operazioni di join tra due dataset, eseguite sulla colonna *Numbers*. In questi test, il join viene effettuato tra il dataset di base e il dataset con 30 frutti, al fine di valutare come il motore SQL e le metodologie offerte da Pandas (tramite le funzioni `join()` e `merge()`) reagiscano al crescente volume di dati. I risultati riportati nei grafici si concentrano sulle configurazioni fino a 100'000 righe, poiché oltre tale soglia i tempi d'esecuzione diventano ingenti e i dati raccolti non garantirebbero una rappresentazione grafica precisa.

- **Test 7 – Join con indicizzazione preliminare**
  In questo test, prima di eseguire l'operazione di join, viene applicata un'indicizzazione preventiva sulla colonna *Numbers* di entrambe le tabelle. L'obiettivo è osservare come l'uso dell'indice influenzi le prestazioni di SQL e confrontarle con le due metodologie Pandas. In questo caso si ottenuti i seguenti risultati:

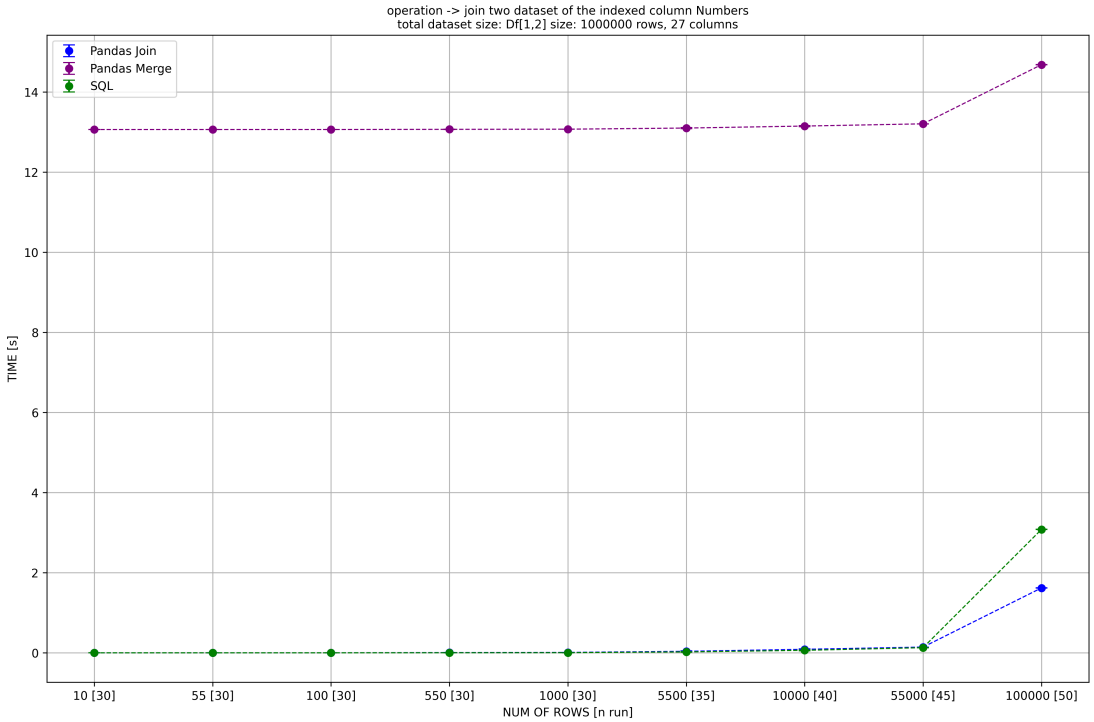


Figura 6.8. Test 7 - Risultati relativi alle tre metodologie

  – **Pandas `join()`**: Questa funzione imposta la colonna *Numbers* come indice per entrambi i DataFrame e poi esegue un merge basato sugli indici. I

risultati mostrano tempi estremamente rapidi, con medie che variano da un ordine di 10^{-3} secondi per configurazioni basse, fino a circa 1.62 s per la configurazione da 100'000 righe.

– **Pandas `merge()`**: Utilizzata come alternativa, questa funzione di Pandas, esegue l'operazione di join tra i due dataset, ma non imposta esplicitamente gli indici, di conseguenza i tempi di performance risultano essere in media notevolmente superiori, attestandosi attorno ai 13 secondi, segnalando un costo computazionale molto elevato. Tale lentezza potrebbe derivare dall'ulteriore complessità di allineamento e copia dei dati che `merge()`, essendo un'operazione che offre maggior versatilità, comporta rispetto a `join()`.

– **SQL (con indicizzazione)**: L'applicazione preventiva degli indici sulla colonna *Numbers* permette al motore SQL di accedere direttamente ai valori estremi grazie alla struttura ordinata dell'indice (tipicamente un B-Tree), consentendo tempi d'esecuzione che partono da un ordine di 10^{-6} secondi per le configurazioni più basse e raggiungono circa 3.08 s per la configurazione da 100'000 righe.

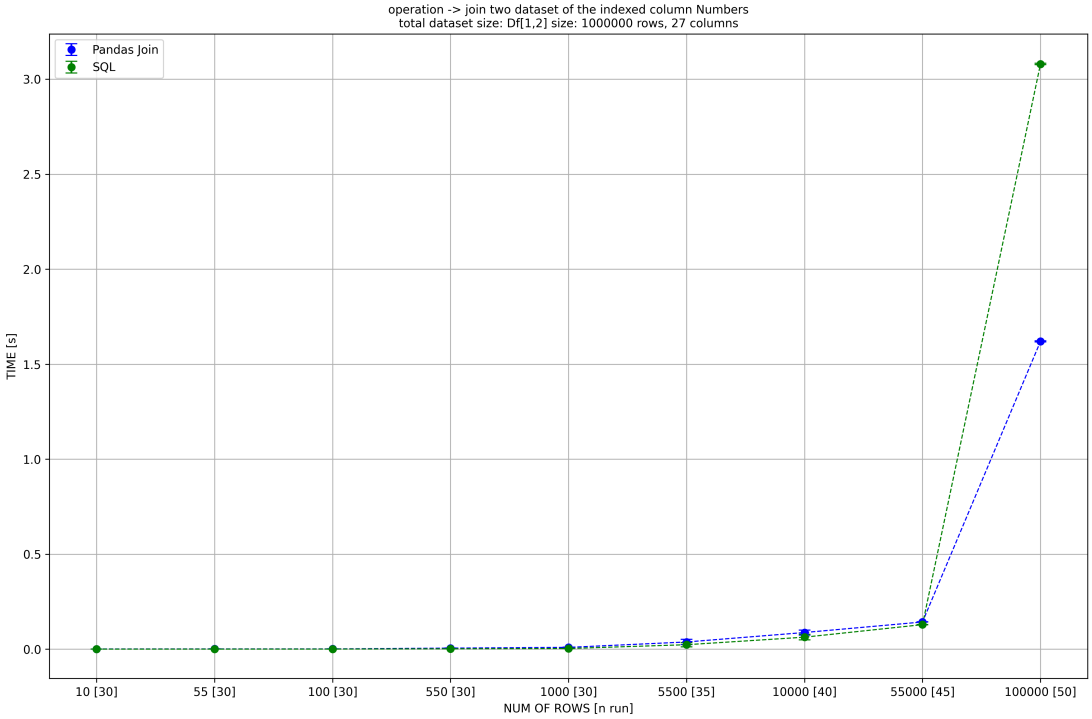


Figura 6.9. Test 7 - Risultati di Pandas `join()` e SQL

I grafici relativi al Test 7 evidenziano come, in presenza di indicizzazione, il metodo SQL mostri prestazioni notevolmente migliorate, come si era visto anche nel test 4, dedicato all'operazione di calcolo del massimo, dove SQL si è dimostrato addirittura superiore a Pandas, in termini di prestazione. L'indicizzazione consente al motore SQL di evitare una scansione completa della tabella, accedendo direttamente al valore desiderato con complessità $O(1)$ o $O(\log\ n)$, ottimizzando l'interrogazione dei dati e impiegando tempi sensibilmente più brevi. Al contrario, l'operazione `merge()` in Pandas si rivela molto meno performante rispetto al metodo `join()`, che è più indicato per operazioni basate sugli indici.

- **Test 8 - Join senza indicizzazione preliminare**
  Nel Test 8, l'operazione di join tra i due dataset viene eseguita senza attuare alcuna indicizzazione preventiva sulla colonna *Numbers*. I risultati mostrano una rimarchevole differenza rispetto al Test 7:

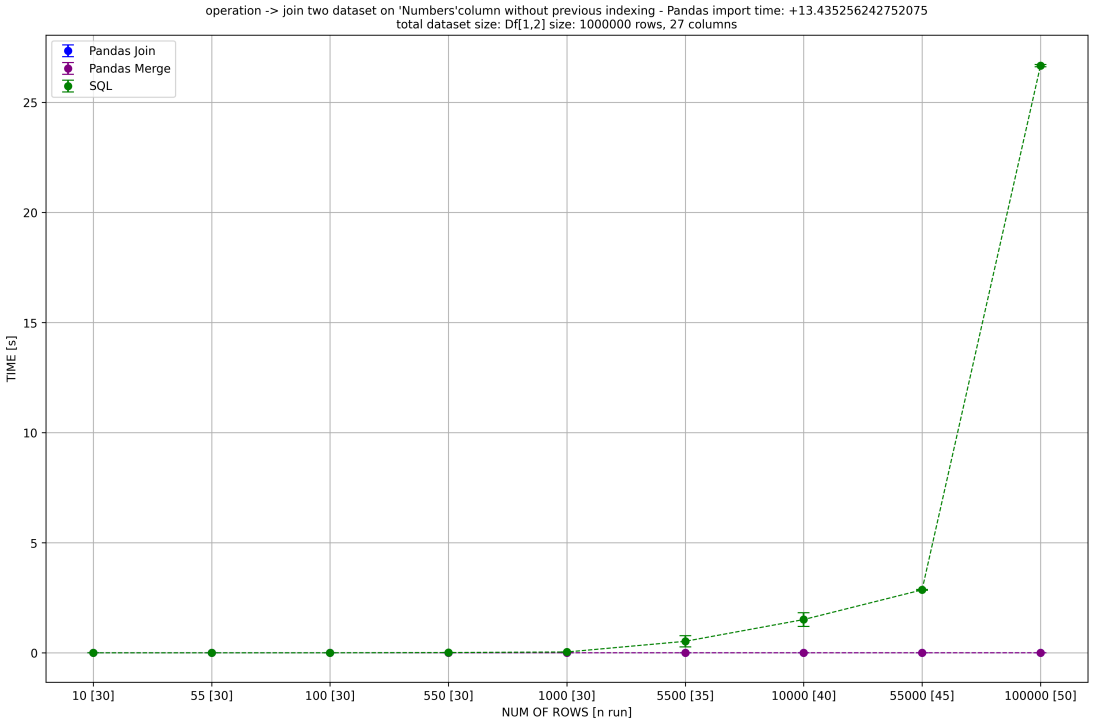


Figura 6.10. Test 8 - Risultati relativi alle tre metodologie

  – **SQL (senza indicizzazione)**: In questo test, senza indicizzazione preventiva, i tempi di esecuzione per SQL iniziano a crescere in maniera esponenziale con l'aumento del volume dei dati. In configurazioni inferiori a 55'000 righe, SQL risulta solo leggermente più lento rispetto al test precedente; tuttavia,

per configurazioni maggiori, come quella da 100'000 righe, il tempo medio raggiunge oltre 26 secondi, evidenziando il costo elevato di eseguire un join senza poter usufruire degli indici.

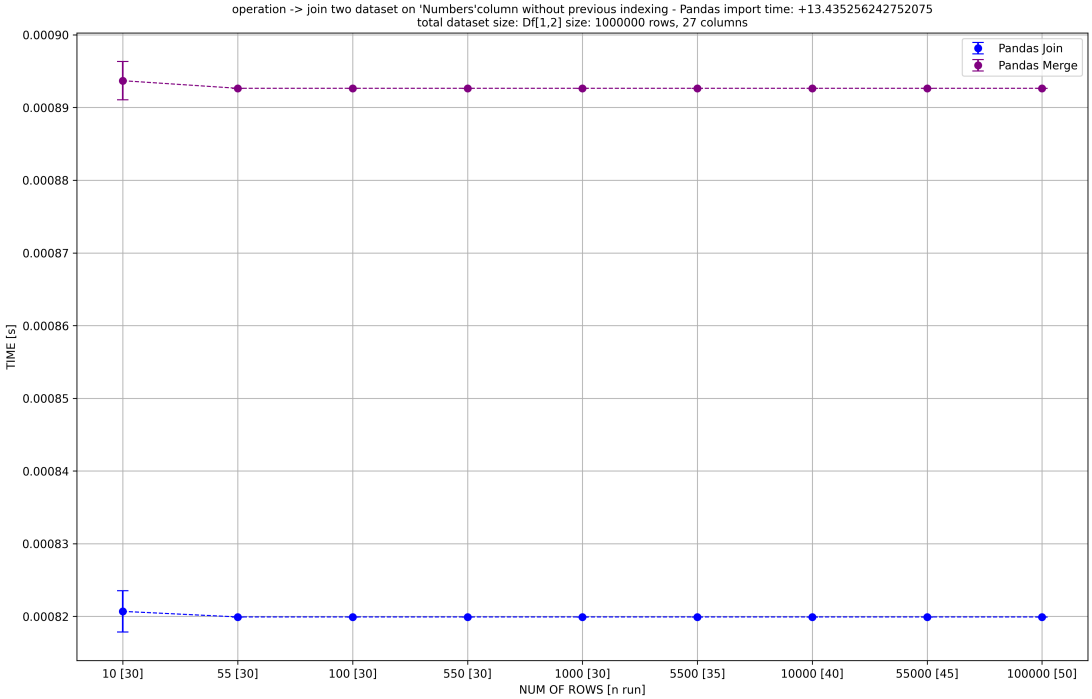


Figura 6.11. Test 8 - Risultati di Pandas `join()` e `merge()`

– **Pandas `join()` e `merge()`**: Entrambi i metodi di Pandas riportano, al contrario, tempi particolarmente rapidi, estremamente stabili e prossimi tra loro (circa 0.00082 s per `join()` e 0.00089 s per `merge()`) lungo tutte le configurazioni, evidenziando una prestazione solida ed efficiente, che non risente in maniera significativa della variazione del volume dei dati.

Questi dati sottolineano l'importanza dell'indicizzazione: senza di essa, SQL deve scansionare l'intera tabella per individuare le corrispondenze sulla colonna *Numbers*, il che comporta un aumento notevole del tempo d'esecuzione con l'espansione del dataset. In contrasto, le operazioni in Pandas, basate su strutture dati in-memory, non subiscono variazioni rilevanti, rimanendo costanti su tutte le configurazioni e mantenendo tempi d'esecuzione estremamente bassi. Al contrario di SQL, entrambe le funzioni di Pandas godono già di ottimizzazioni interne e si dimostrano più performanti senza un'operazione di indicizzazione preliminare.

Considerazioni Finali e Confronto Complessivo I grafici relativi ai Test 7 e 8 mostrano chiaramente che l'indicizzazione preventiva sulla colonna *Numbers* ha un impatto decisivo sulle prestazioni di SQL. Con l'indicizzazione (Test 7), SQL ottiene tempi significativamente inferiori rispetto allo scenario senza indicizzazione (Test 8), riducendo il gap di tempo in modo notevole (intorno a 1.5 secondi) e rendendo l'operazione più competitiva. Per quanto riguarda le metodologie di join in Pandas, il metodo `join()` risulta estremamente più performante rispetto a `merge()`, che registra tempi d'esecuzione nettamente superiori (mediamente oltre 13 s nel Test 7). Nel Test 8, invece, entrambi i metodi di Pandas si comportano in maniera quasi identica, evidenziando una stabilità eccezionale che non varia con l'aumentare del volume dei dati. Questo comportamento è dovuto al fatto che Pandas lavora interamente in-memory e utilizza strutture dati ottimizzate per l'accesso diretto, a differenza di SQL che, senza indice, deve eseguire una scansione completa della tabella per trovare i valori corrispondenti. La comparazione complessiva tra SQL e Pandas evidenzia che, in operazioni di join su dataset ampi, la mancanza di indicizzazione penalizza fortemente SQL, portandolo ad aumentare esponenzialmente i tempi di esecuzione. L'indicizzazione, infatti, riduce drasticamente il costo operativo del join, permettendo al motore SQL di sfruttare strutture dati ordinate per l'accesso rapido.
Viene infine citato che il tempo di importazione del dataset in Pandas supera mediamente i 13 secondi per il caricamento dei due dataset necessari all'operazione di join. Esso rappresenta un costo fisso in termini di tempo e, sebbene sia rilevante se considerato singolarmente, esso potrebbe andare incontro ad una progressiva ammortizzazione nel contesto di numerose operazioni sugli stessi DataFrame caricati in memoria. Nel caso qui preso in esame, del confronto tra i tempi d'esecuzione delle operazioni di join, esso non incide direttamente sui risultati e non intacca le performance dei due metodi.

In definitiva, i risultati dei test 7 e 8 dimostrano che l'operazione di indicizzazione è un elemento chiave per ottimizzare le prestazioni di SQL e di fondamentale importanza all'aumentare del volume dei dati; mentre Pandas, grazie alla gestione in-memory, garantisce performance elevate a velocità costante quando in assenza di indici. In particolare, tra le due metodologie di Pandas osservate, il `join()` si conferma come la scelta più efficiente per operazioni su dataset con chiavi indicizzabili rispetto al `merge()`.

7. Conclusione e Sviluppi Futuri

Discussi ed esaminati in dettaglio i risultati delle varie prove sperimentali nel capitolo precedente, in questa sezione verranno tratte le somme del lavoro svolto e delineate le conclusioni generali. Saranno quindi evidenziati i principali punti di forza e le criticità emerse nel confronto tra Pandas e SQL su dataset tabellari, per poi proporre possibili percorsi di ricerca e sviluppi futuri, garantendo così la continuità del percorso intrapreso con questa tesi.

7.1 Sinossi della Tesi

Nel corso della presente tesi è stata condotta un'analisi comparativa delle prestazioni di Pandas e SQL nell'ambito dell'elaborazione e interrogazione dei dati su dataset tabellari. Il lavoro si è articolato in diverse fasi:

- Nella parte relativa allo **Stato dell'Arte** (capitoli 2 e 3) sono state presentate le basi teoriche e gli ambiti di applicazione sia di SQL – il linguaggio standard per la gestione di database relazionali – sia di Pandas, una libreria sviluppata per l'analisi dei dati in-memory in Python, con particolare riferimento alle sue capacità di integrazione con NumPy e altre librerie scientifiche.

- E' stata quindi discussa la fase di **Progettazione**, che ha definito gli obiettivi della tesi e ha descritto l'architettura dei test, illustrando la costruzione dei dataset (basati su file CSV, costituiti da 1.000.000 di righe e 27 colonne), e delineando le metodologie adottate per confrontare le operazioni comuni (come la ricerca di una stringa, raggruppamenti e operazioni aritmetiche) eseguite sia con Pandas che con SQL.

- Nella sezione dedicata all'**Implementazione** sono stati descritti nel dettaglio gli script prodotti, rispettivamente è stato introdotto lo script di creazione dei dataset, chiarendo anche su quali tipi di dati tabellari si sarebbero poi eseguite le prove; si sono quindi illustrati gli script relativi ai test, spiegando il codice e la logica implementativa in maniera approfondita e discutendo le scelte tecniche adottate, evidenziando dapprima le parti comuni a tutti i test e in secondo luogo, le differenze specifiche nelle operazioni svolte; infine è stato brevemente introdotto lo script dedicato alla costruzione dei grafici, mediante rielaborazione dei risultati prodotti dai test e applicazione di indici statistici.

- Il capitolo dedicato alla **Validazione Sperimentale** ha quindi sancito l'esecuzione degli script e lo svogimento effettivo dei test, presentando l'analisi dei risultati raccolti mediante il dispiegamento degli appositi grafici, i quali confrontano i tempi di esecuzione in diverse configurazioni, evidenziando come le performance varino in relazione al volume dei dati e alla complessità dell'operazione. L'intera collezione dei grafici rappresentanti i risultati ottenuti è allegata per intero nella sezione **Grafici**, in appendice. Sono quindi state tratte osservazioni e considerazioni specifiche per ogni gruppo di test, che verranno di seguito raccolte in una conclusione collettiva.

Gli obiettivi iniziali – valutare le prestazioni di Pandas e la sua competitività nei confronti del linguaggio SQL, metro standard nell'ambiente del *data analysis*, in termini di prestazioni e tempi d'esecuzione, identificando i punti di forza e le criticità di ciascun approccio – sono stati confermati e approfonditi attraverso una serie di otto test che hanno preso in esame alcune operazioni specifiche - ricerca di una stringa, raggruppamenti con e senza condizione, operazioni aritmetiche e operazioni di join - che hanno permesso di mettere in luce il comportamento differenziato delle due metodologie, in relazione al tipo di operazione eseguita.

7.2 Conclusioni relative ai Risultati Raccolti

I risultati sperimentali hanno evidenziato le seguenti considerazioni principali:

- Per un'operazione di semplice ricerca di una stringa, SQL risulta più efficiente, soprattutto in presenza di grandi volumi di dati, dovute con ogni probabilità alle ottimizzazioni interne del motore del database. Tuttavia, l'approccio in-memory di Pandas, sebbene considerevolmente penalizzato se si considera anche il tempo di importazione, garantisce performance stabili e tempi di esecuzione competitivi. Appare evidente che per questo tipo di operazioni sia SQL a garantire le performance più efficienti.

- Nei test relativi ai raggruppamenti e conteggi con o senza condizione (Test 2 e Test 3), i risultati hanno mostrato che Pandas può essere competitivo fintanto che si tratta solo di operazioni di raggruppamento con conteggio. Per le configurazioni di pool testate Pandas si è comportato bene, registrando ottime performance nelle prove eseguite su dataset con 10 frutti e con 50 frutti; tuttavia, nella configurazione intermedia qui testata(con 30 frutti), si osserva una lieve flessione delle prestazioni, suggerendo probabilmente una soglia critica in cui le ottimizzazioni in-memory

non riescono a compensare completamente la complessità dell'operazione. Le performance di Pandas subiscono invece un drastico rallentamento quando si introduce una condizione di filtraggio, identificando nel linguaggio SQL la metodologia che più si adatta a questo tipo di operazioni più complesse.

- Per le operazioni aritmetiche (Test 4, 5 e 6), Pandas dimostra un'elevata stabilità e costanza nei tempi di esecuzione, grazie alle sue funzioni vettorializzate ottimizzate e all'impiego di librerie C sottostanti (BLAS, LAPACK). SQL, al contrario, senza indicizzazione preventiva, registra un incremento dei tempi notevole man mano che cresce il volume dei dati, a causa della necessità di effettuare una scansione completa dei dati su supporti di memoria di massa. L'applicazione dell'indicizzazione sulla colonna *Numbers* si rivela infatti determinante per le prestazioni di SQL. In particolare, per il calcolo del massimo, SQL indicizzato raggiunge tempi dell'ordine dei microsecondi, risultando addirittura più veloce di Pandas. Per il calcolo del minimo e della somma, sebbene l'indicizzazione migliori sensibilmente le performance, SQL resta leggermente più lento rispetto a Pandas. Queste differenze evidenziano come l'efficacia degli approcci dipenda fortemente dalla natura dell'operazione e dalla possibilità di sfruttare strutture dati ordinate.

- Nei test 7 e 8, relativi ad operazioni di join tra due dataset, Pandas garantisce performance solide e rapide, sia nel metodo `merge()`, che `join()`, con quest'ultimo che, nel caso dei dataset qui presi in considerazione, emerge più rapido e consistente, grazie alla sua capacità di lavorare su una colonna con chiavi indicizzabili. Le prestazioni di Pandas si rivelano efficienti anche nel confronto con SQL, il quale senza indicizzazione preventiva delle colonne registra tempi non competitivi, ma anche nel test con avvenuta indicizzazione preliminare registra comunque tempi d'esecuzione peggiori di quelli di Pandas.

Un ulteriore aspetto rilevante nelle prove prese in esame è il costo di importazione del dataset in memoria. Esso rappresenta un costo fisso che ammonta mediamente a 7 secondi per il caricamento di un singolo dataset (nei casi presi in esame, 1'000'000 di righe x 27 colonna, per una dimensione di crica 300 MB) e per le operazioni in cui i dataset caricati sono stati due, si aggira invece intorno a 13 secondi. Questo costo, sebbene rilevante in una singola operazione, può subire un importante ammortizzazione nel contesto in cui si debbano svolgere numerose operazioni sullo stesso DataFrame importato. Il tempo di import, pur rimanendo un elemento da considerare nella scelta dello strumento in funzione del contesto applicativo, non incide direttamente

sul confronto delle performance operative e non intacca le prestazioni specifiche delle operazioni in Pandas.

7.3 Sviluppi Futuri

Il lavoro svolto apre numerose prospettive di approfondimento e sviluppo:

- **Espansione del confronto di query**
  In questa tesi sono state testate alcune tipologie di query fondamentali, quali la ricerca, il raggruppamento, le operazioni aritmetiche e i join. Un ulteriore approfondimento potrebbe estendere il confronto a una gamma più ampia di operazioni, sia quelle relativamente semplici che quelle che, aggiungendovi complessità, potrebbero evidenziare ulteriori limiti e potenzialità. Ad esempio, si potrebbero considerare:

  – **Query di aggregazione complesse**:
    oltre al semplice calcolo di massimo, minimo e somma, si potrebbe valutare l'applicazione di funzioni aggregative multiple (come media, mediana, deviazione standard) in un'unica query, o l'utilizzo di funzioni di finestra (`window functions`: istruzioni che permettono di associare informazioni aggregate a tutte le righe di una tabella, senza impattare il numero di righe e di colonne nel risulto in output) per ottenere statistiche mobili o cumulative. Tali operazioni potrebbero mettere in luce la capacità di Pandas di gestire aggregazioni complesse grazie al supporto vettorializzato di NumPy, mentre SQL potrebbe soffrire in presenza di subquery e join interni.
  – **Join multipli e join condizionali**:
    mentre i nostri test si sono concentrati su un singolo join tra due dataset, operazioni che coinvolgono più tabelle o join basati su condizioni complesse (ad esempio, join condizionali o join con più chiavi) potrebbero evidenziare ulteriori differenze di prestazione. In questi scenari, l'efficienza dell'indicizzazione e la capacità di SQL di ottimizzare piani di esecuzione complessi potrebbero rivelarsi determinanti, mentre l'approccio in-memory di Pandas potrebbe risentire della necessità di ricalcolare più volte gli indici.

  Questi ulteriori test permetterebbero di verificare se il vantaggio competitivo di Pandas, evidente finché la situazione rimane semplice, si mantenga anche in contesti di maggiore complessità, o se invece l'inefficienza nelle operazioni non ottimizzate si acuisca. Inoltre, un confronto con tali query complesse offrirebbe

la possibilità di delineare in maniera più precisa i limiti e i punti di forza di ciascun approccio, fornendo spunti interessanti per applicazioni in ambito machine learning, data analysis e, in generale, in ogni contesto in cui le query dinamiche e complesse rappresentino un elemento cruciale.

- **Diversificazione dei formati dei dati**:
  Il presente studio si è concentrato su dataset in formato CSV. Per un'analisi più completa, sarebbe interessante eseguire test anche su formati alternativi, quali JSON, TSV o dati provenienti direttamente da database NoSQL, per valutare se le differenze di prestazione riscontrate siano confermate anche in questi contesti o per trovare ambienti applicativi specifici in cui Pandas possa essere impiegato consistentemente e sostituirsi al classico SQL.

- **Confronti con ambienti NoSQL**:
  Considerando l'espansione dei sistemi NoSQL e il crescente interesse per le soluzioni di data management non relazionali, un possibile sviluppo futuro potrebbe consistere nell'estendere il confronto a questi ambienti, verificando come Pandas si confronti non solo con SQL, ma anche con motori NoSQL, nei termini di velocità, flessibilità e scalabilità.

- **Analisi su differenti configurazioni hardware**:
  Un ulteriore aspetto di interesse riguarda il ruolo dell'hardware. I test condotti in questa tesi sono stati eseguiti su una singola configurazione (HP Pavilion con AMD Ryzen 7 7730U e 16 GB di RAM). Sarebbe utile replicare gli esperimenti su macchine con caratteristiche differenti per studiare la correlazione tra la dimensione del dataset, la memoria disponibile e le performance dei vari strumenti.

- **Impatto di Pandas sul consumo di memoria**:
  Oltre alla valutazione delle prestazioni in termini di tempo di esecuzione, un importante ambito di sviluppo futuro consiste nello studio dell'impatto di Pandas sul consumo di memoria. L'approccio in-memory, pur garantendo rapidità nell'elaborazione, comporta un notevole utilizzo di risorse RAM, il che può diventare un fattore critico in scenari in cui si gestiscono dataset particolarmente ampi o quando le risorse hardware sono limitate. Future ricerche potrebbero integrare metriche di profilazione della memoria, al fine di quantificare il footprint di Pandas durante l'esecuzione di operazioni complesse come raggruppamenti, join e calcoli aritmetici. In particolare, si potrebbe valutare come il consumo di memoria vari al variare del volume dei dati e confrontare questi risultati con altri sistemi di gestione dati, come SQL e motori NoSQL. Tale studio permetterebbe di

identificare eventuali compromessi tra velocità di esecuzione e efficienza nell'uso della memoria, contribuendo a definire linee guida per la scelta dello strumento più adeguato in base al contesto applicativo. Infine, potrebbe essere interessante indagare scenari in cui il tempo di esecuzione non rappresenta il parametro critico, ma l'ottimizzazione del consumo di memoria diventa determinante, soprattutto in ambienti con risorse limitate o in applicazioni che richiedono operazioni ripetute su dataset di grandi dimensioni.

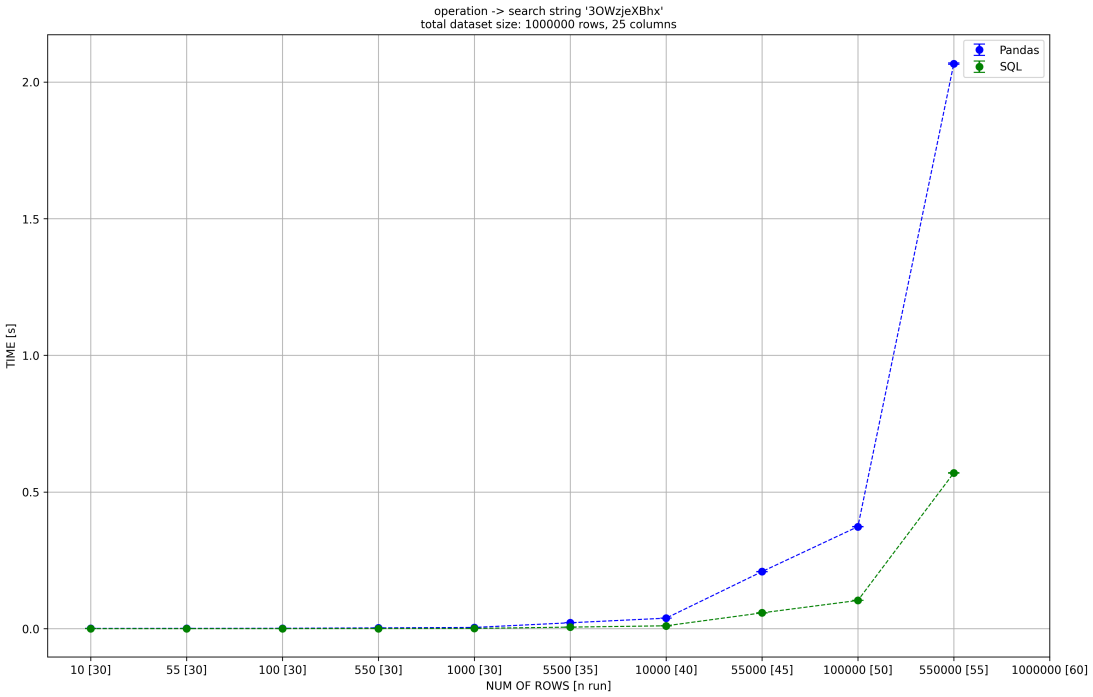
- **Confronto con altre librerie di Data Analysis**:
  Un ulteriore sviluppo futuro interessante riguarda l'estensione del confronto di prestazioni di Pandas a librerie emergenti e alternative, come ad esempio Polars. Mentre Pandas è da tempo lo standard per l'analisi dei dati in ambiente Python – grazie alla sua integrazione con NumPy, alla vasta documentazione e al supporto diffuso in ambito machine learning – sono emerse nuove soluzioni che promettono un'elevata efficienza, in particolare per operazioni su dataset di grandi dimensioni. Polars, ad esempio, è una libreria sviluppata in Rust che sfrutta il parallelismo e ottimizzazioni a basso livello per offrire performance notevolmente migliorate rispetto a Pandas in alcuni scenari. In contesti applicativi quali il machine learning e l'analisi dei Big Data, dove la scalabilità e la rapidità di elaborazione sono cruciali, un confronto approfondito tra Pandas e Polars potrebbe fornire spunti interessanti. Tali studi potrebbero indagare, ad esempio, come Polars gestisca operazioni di join, raggruppamento e aggregazione rispetto a Pandas, e come la sua architettura basata su Rust incida sul consumo di memoria e sul tempo di esecuzione. L'analisi comparativa potrebbe inoltre estendersi a scenari specifici di applicazione, quali elaborazioni in tempo reale in ambito economico o scientifico, in cui la scelta della libreria più performante può tradursi in un vantaggio competitivo. In questo modo, si aprirebbe la possibilità di definire linee guida chiare per la selezione degli strumenti di data analysis in base alle esigenze operative, contribuendo a un panorama di ricerca in continua evoluzione. A questo riguardo è stato pubblicato uno studio interessante su ACM Digital Library intitolato "An Empirical Study on the Energy Usage and Performance of Pandas and Polars Data Analysis Python Libraries" (5), da cui sono stati presi numerosi spunti anche per la stesura di questa tesi.

Questi sviluppi futuri non solo integrerebbero e fornirebbero continuità al lavoro qui svolto, ma fornirebbero ulteriori spunti per l'applicazione pratica di strumenti di data analysis in contesti reali, contribuendo a definire linee guida più precise per la scelta dello strumento migliore in base alle specifiche esigenze operative. In un mondo

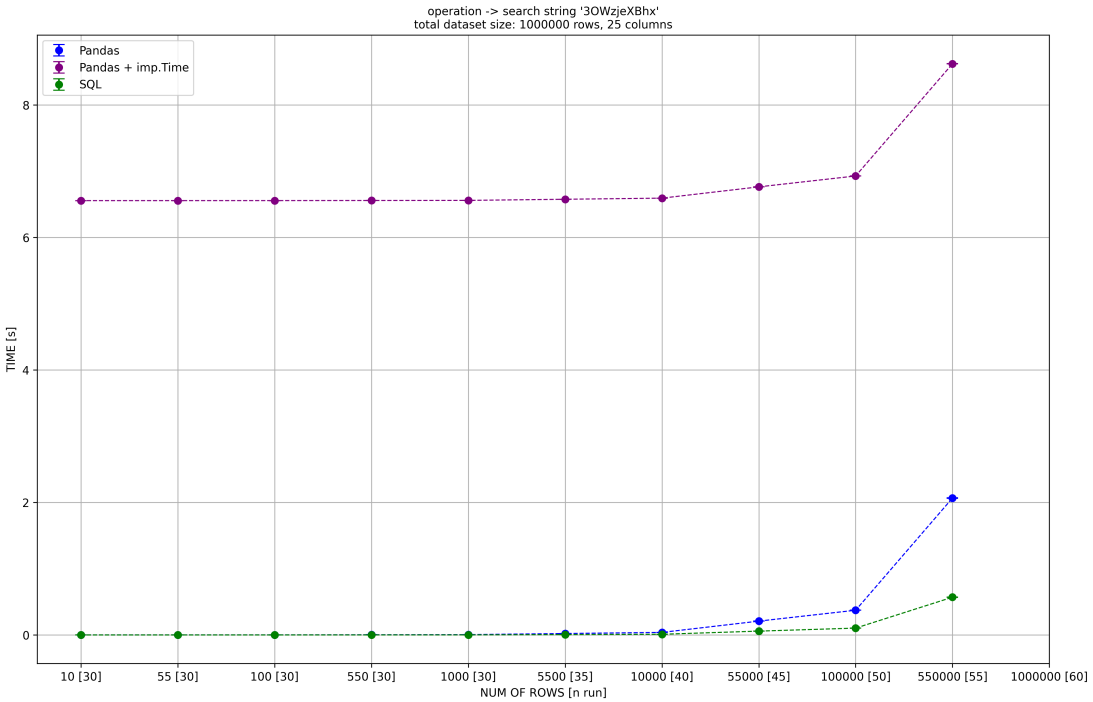
che costantemente si evolve e che sempre più è permeato dalla tecnologia, appare fondamentale non considerarsi mai giunti a destinazione e porre costantemente in discussione anche gli elementi più fidati.

Grafici

Test 1 - Ricerca stringa

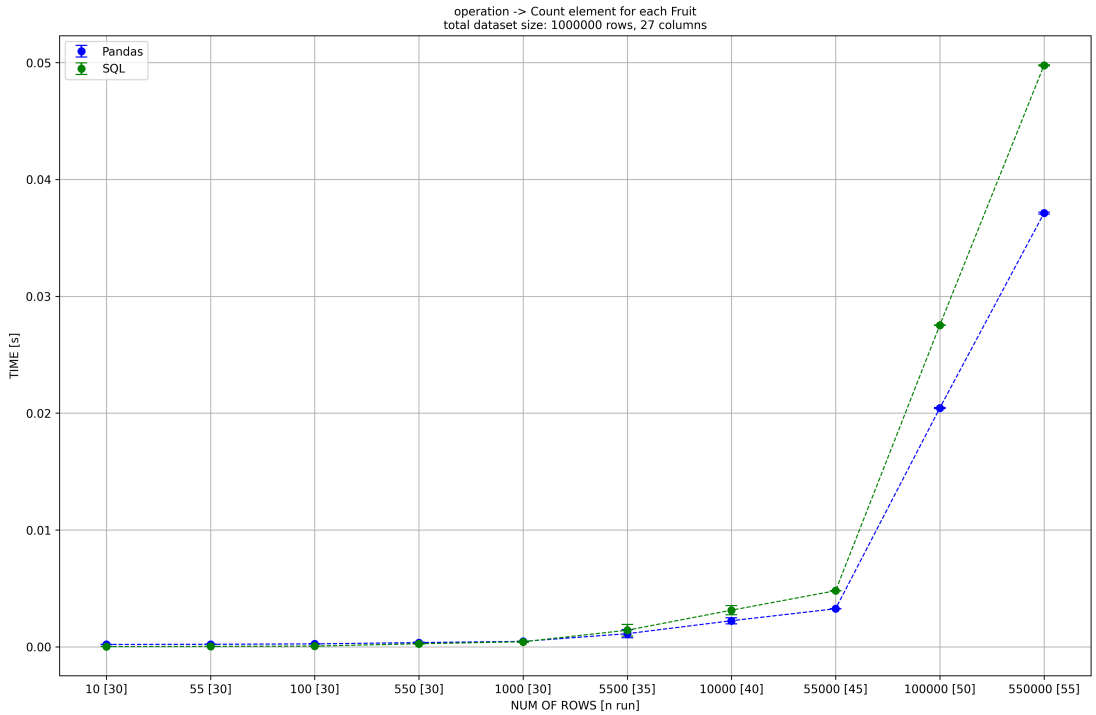


Risultati Pandas e SQL

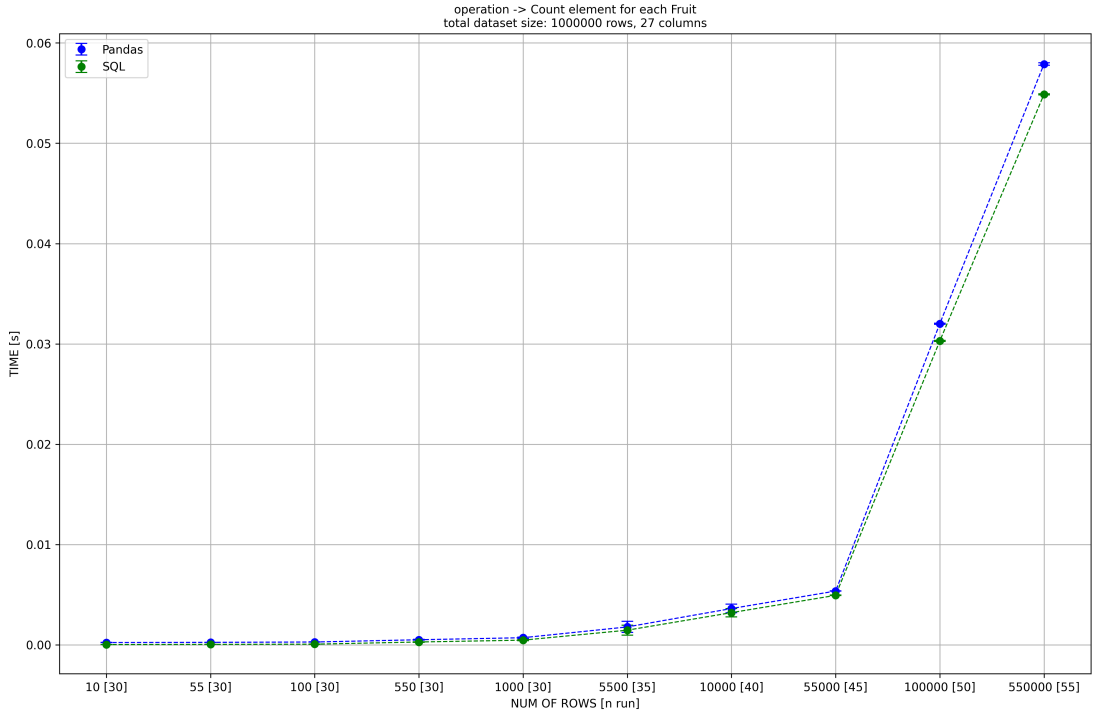


Risultati Pandas, Pandas + Import e SQL

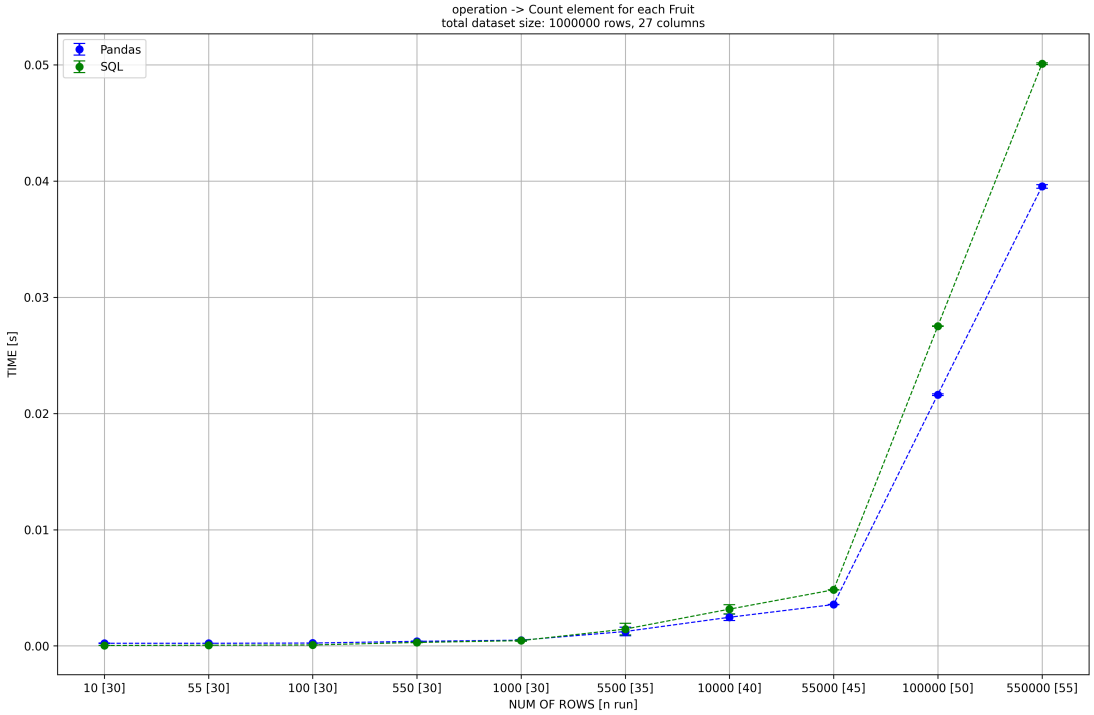
Test 2 - Raggruppamenti e conteggio



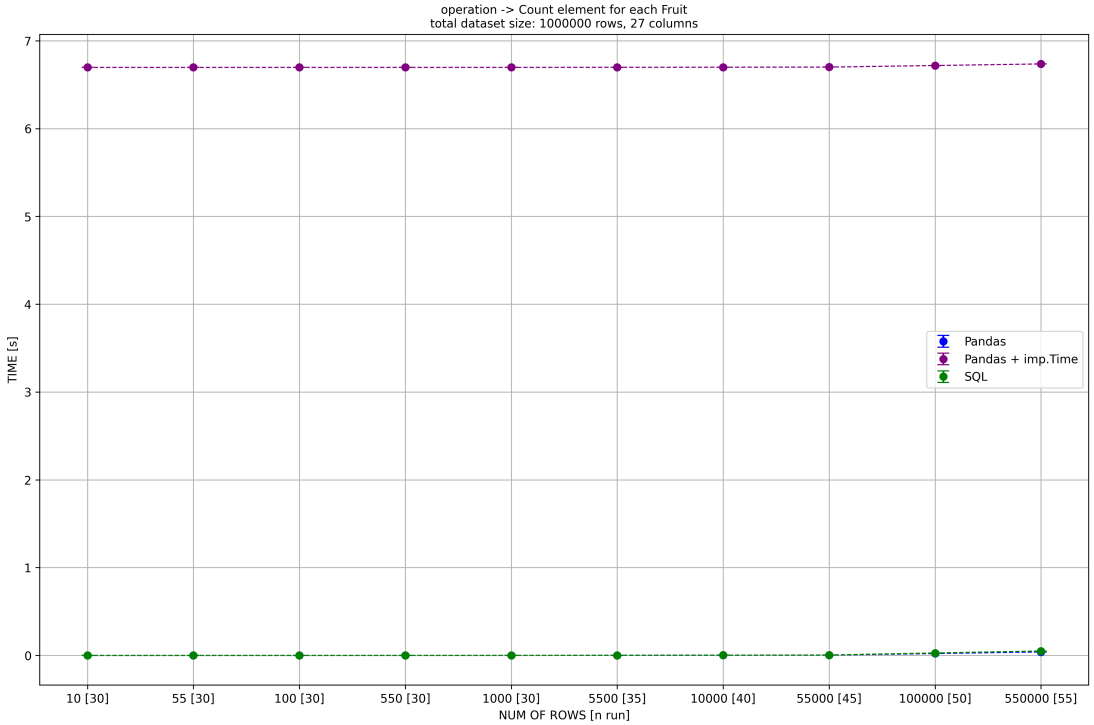
Risultati Pandas e SQL con 10 frutti



Risultati Pandas e SQL con 30 frutti

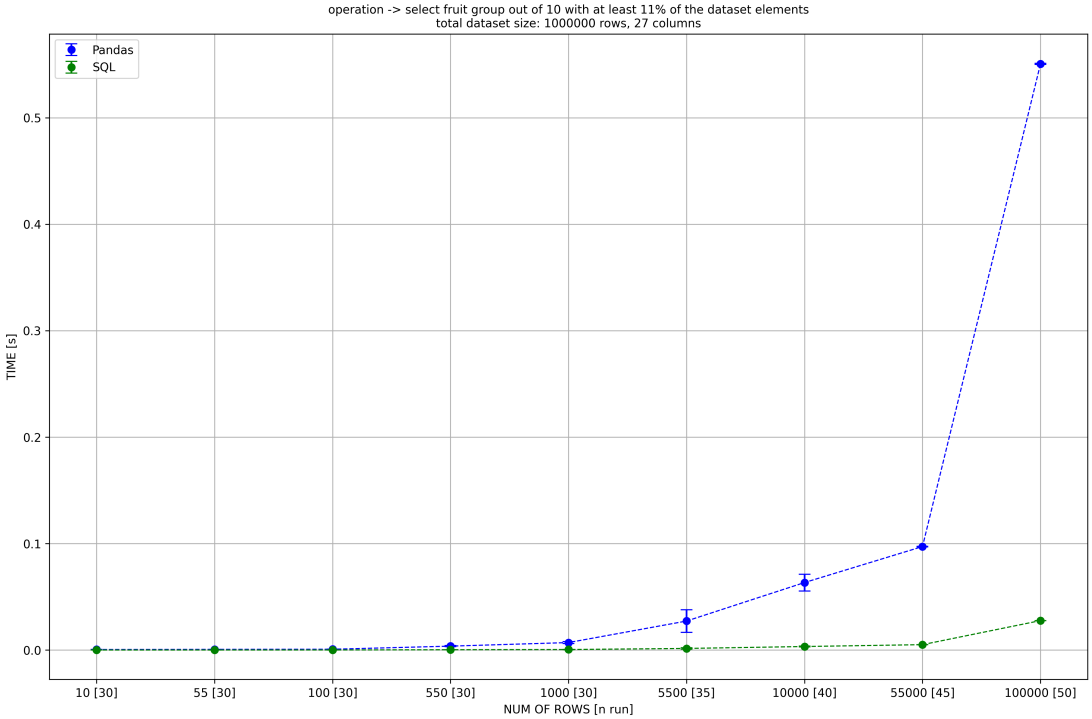


Risultati Pandas e SQL con 50 frutti

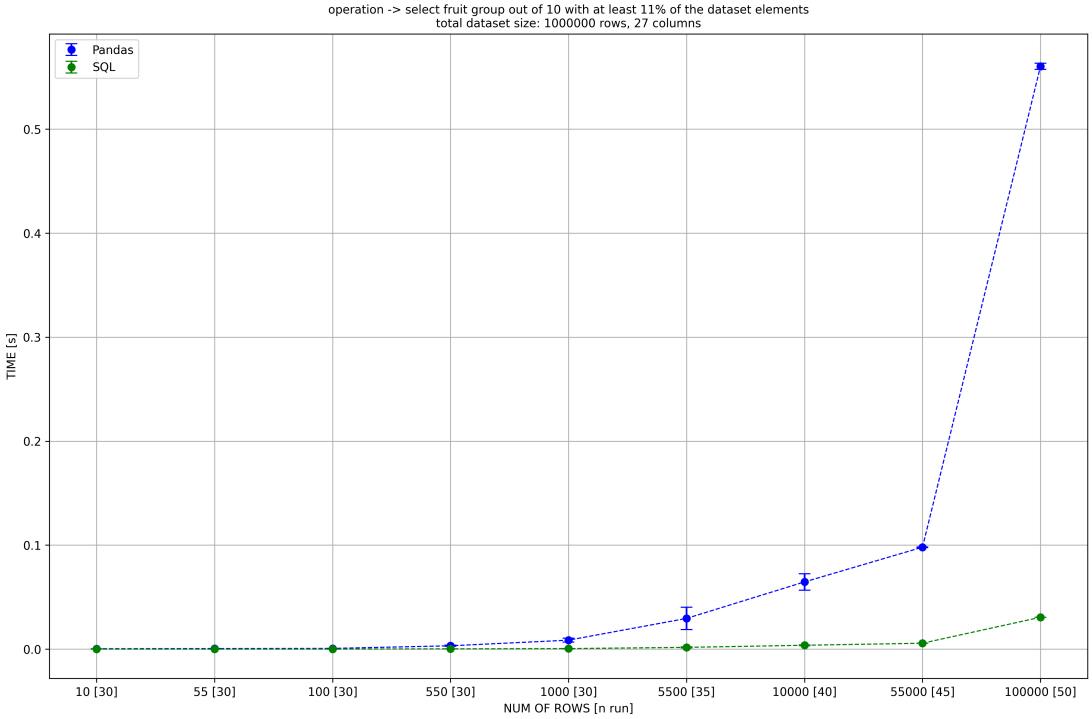


Risultati Pandas, Pandas + import e SQL con 50 frutti

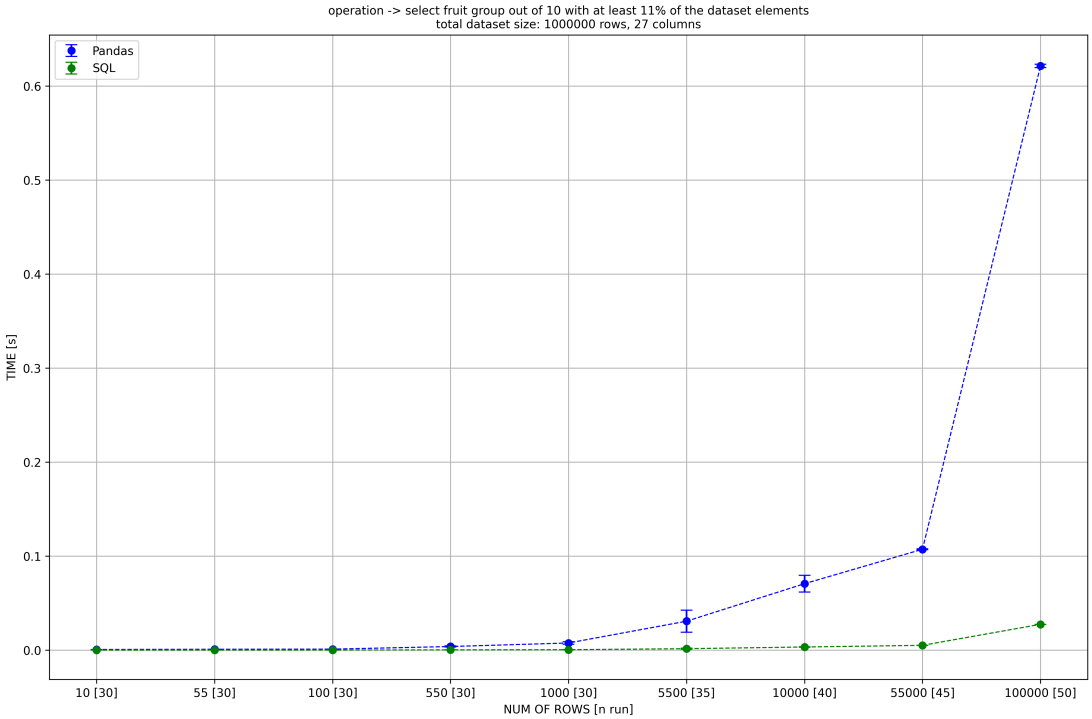
Test 3 - Raggruppamenti e filtraggio condizionale



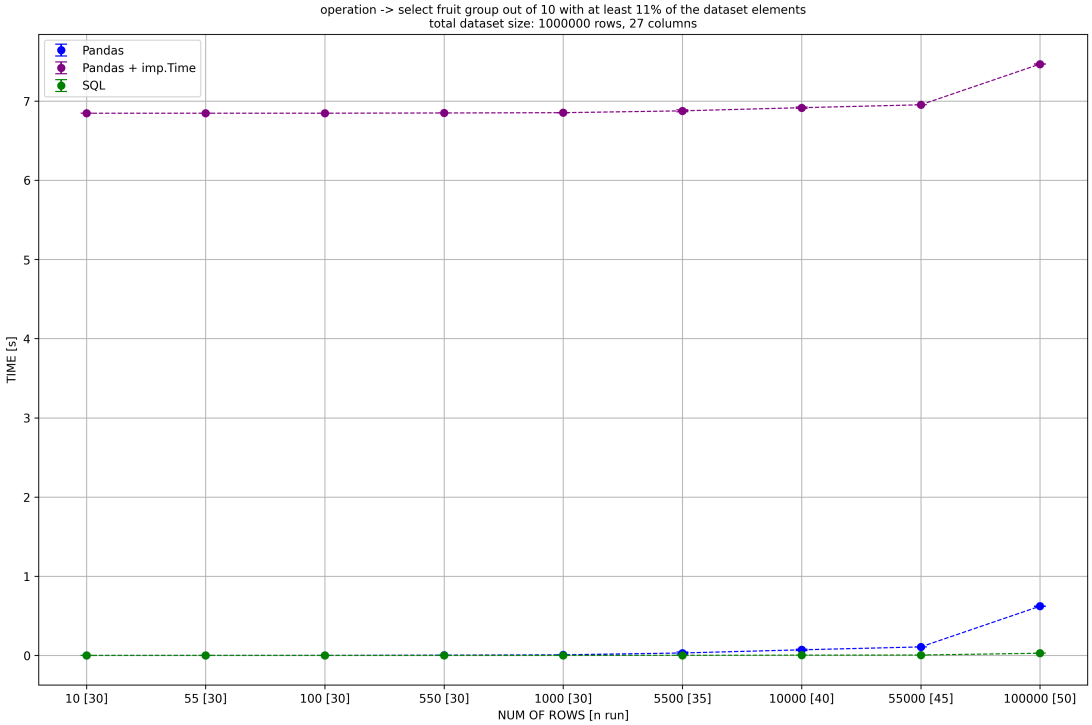
Risultati Pandas e SQL con 10 frutti



Risultati Pandas e SQL con 30 frutti

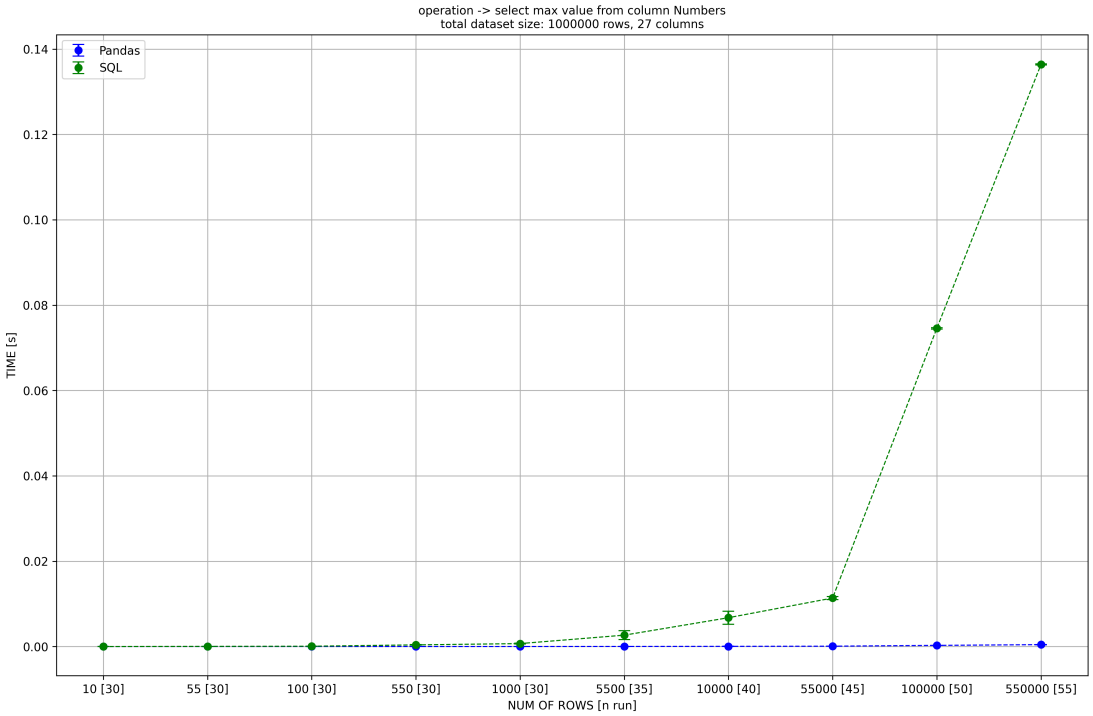


Risultati Pandas e SQL con 50 frutti

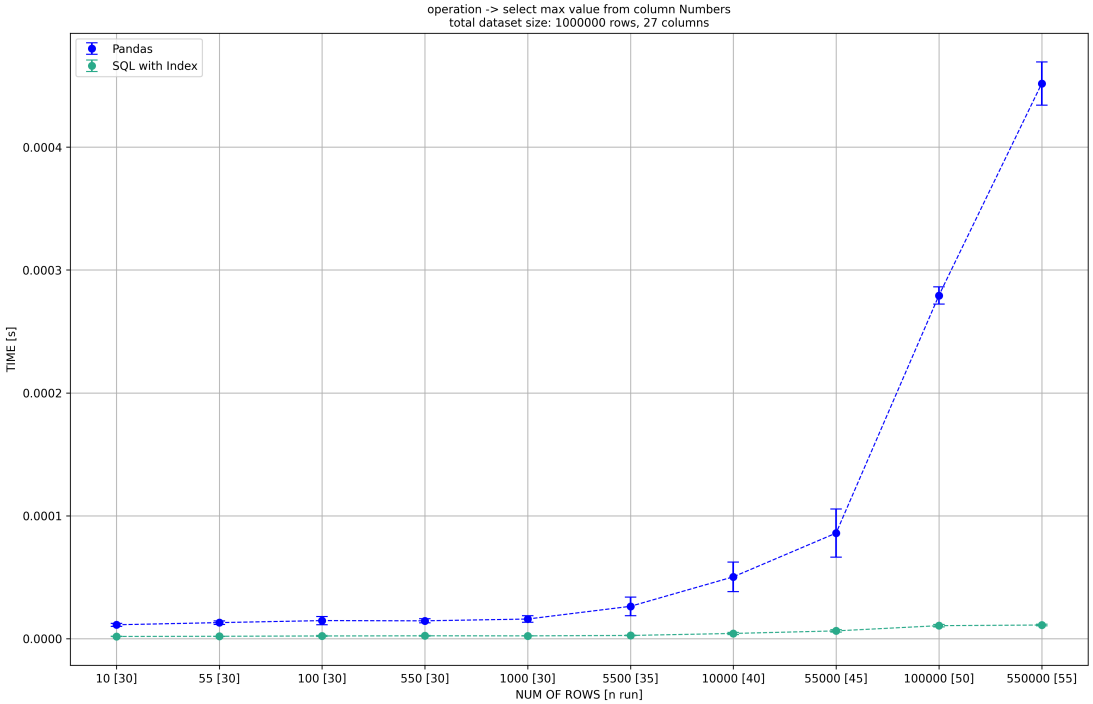


Risultati Pandas, Pandas + import e SQL con 50 frutti

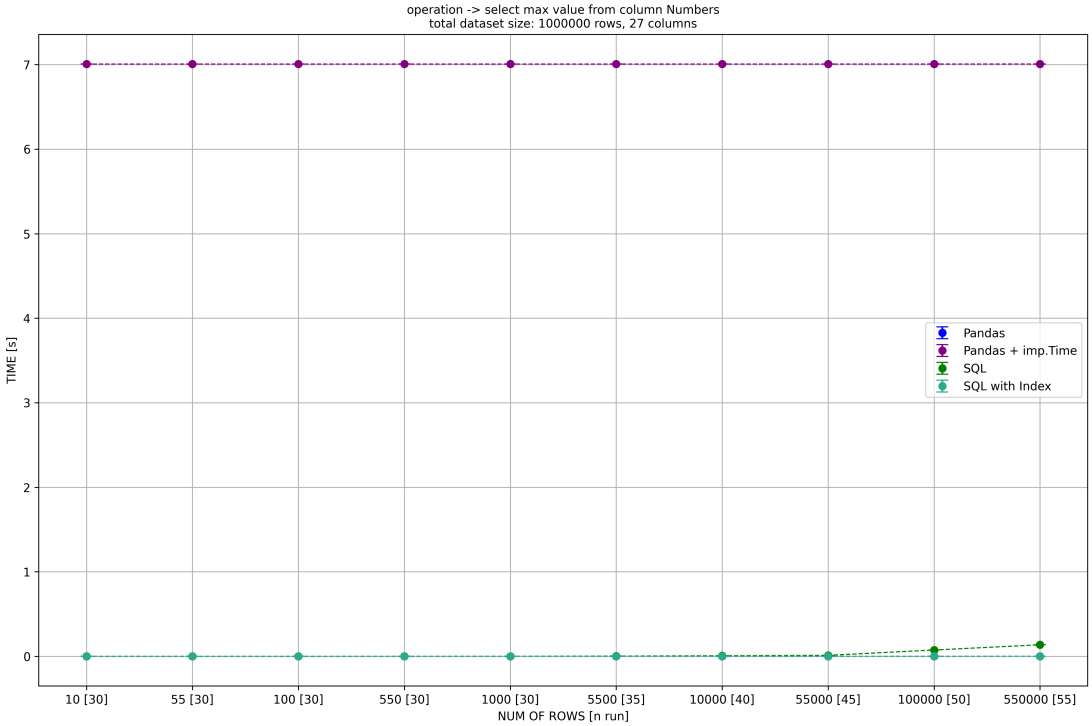
Test 4 - Calcolo del massimo



Risultati Pandas e SQL

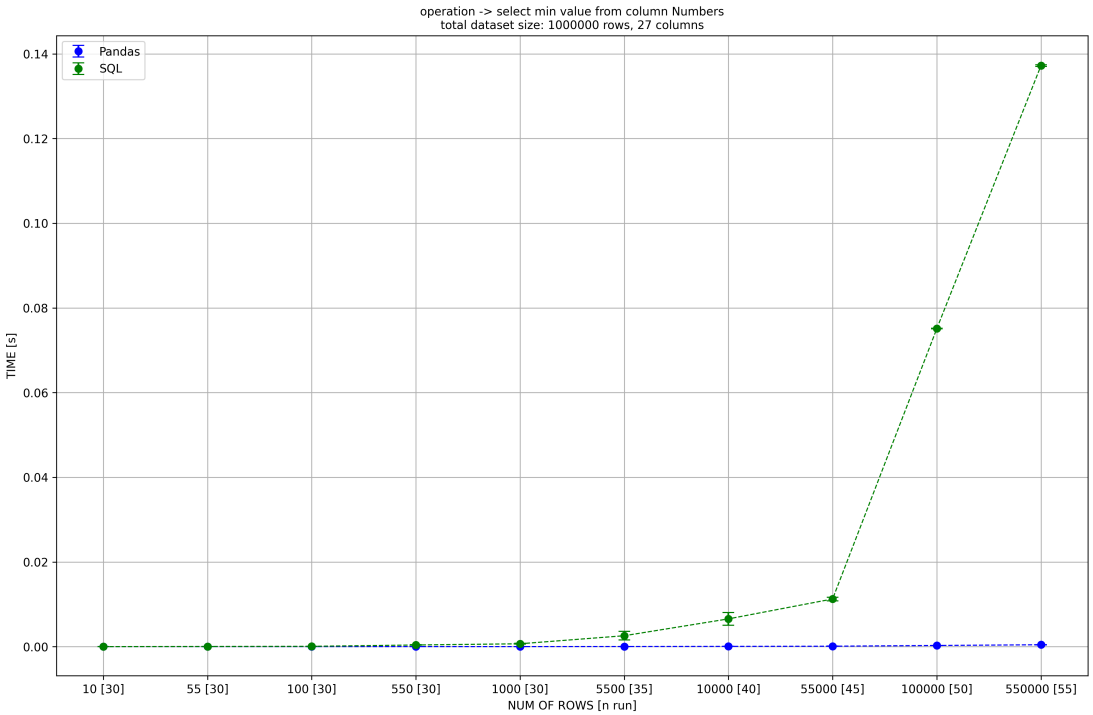


Risultati Pandas e SQL con indicizzazione

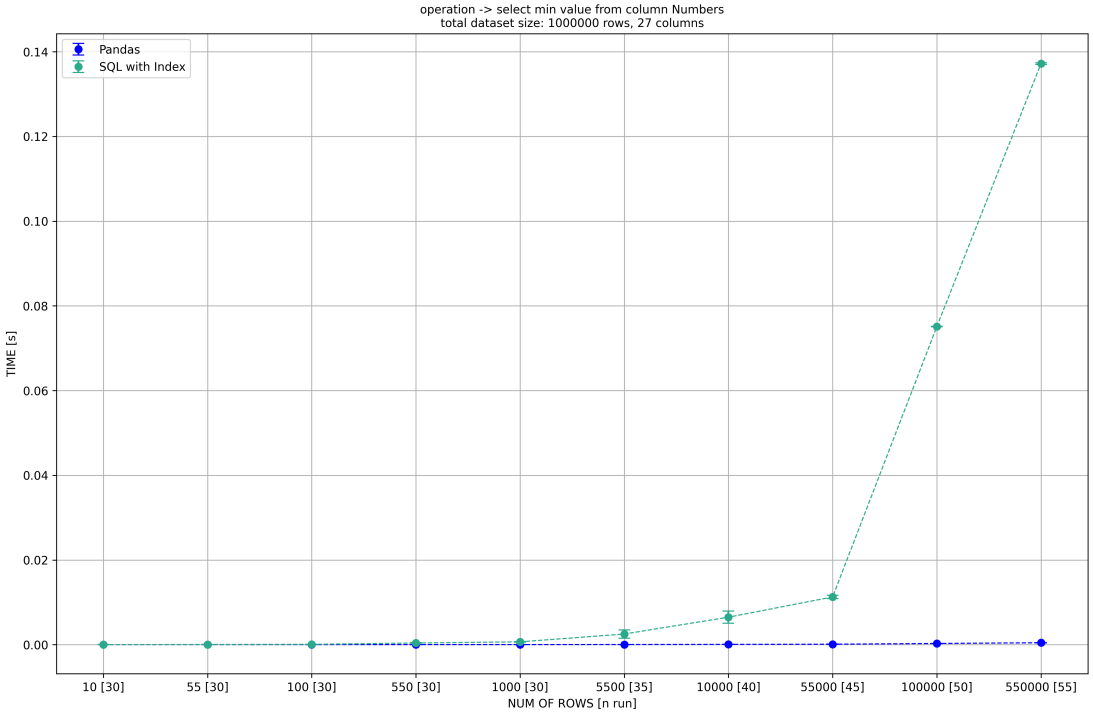


Risultati Pandas, Pandas + import, SQL e SQL index

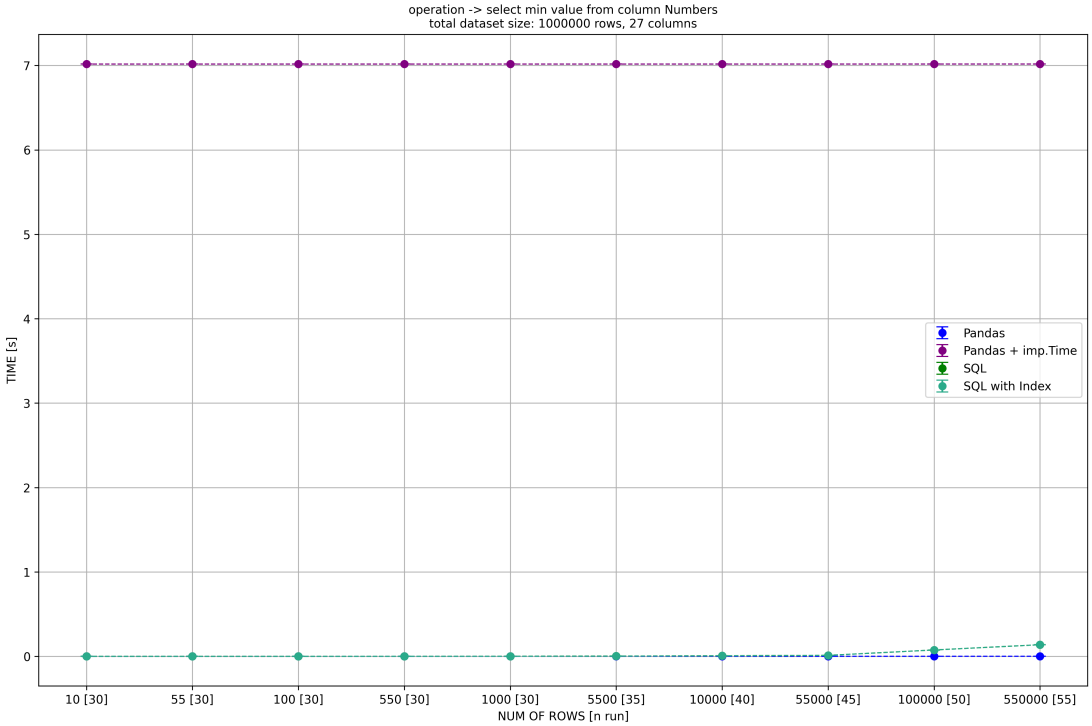
Test 5 - Calcolo del minimo



Risultati Pandas e SQL

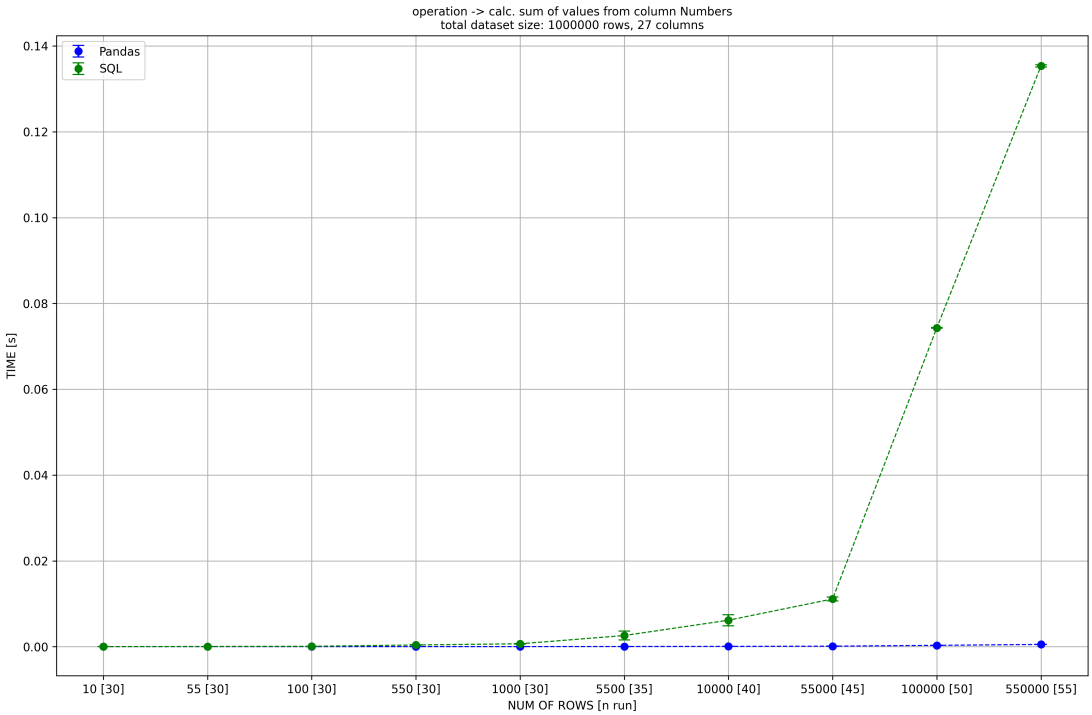


Risultati Pandas e SQL con indicizzazione

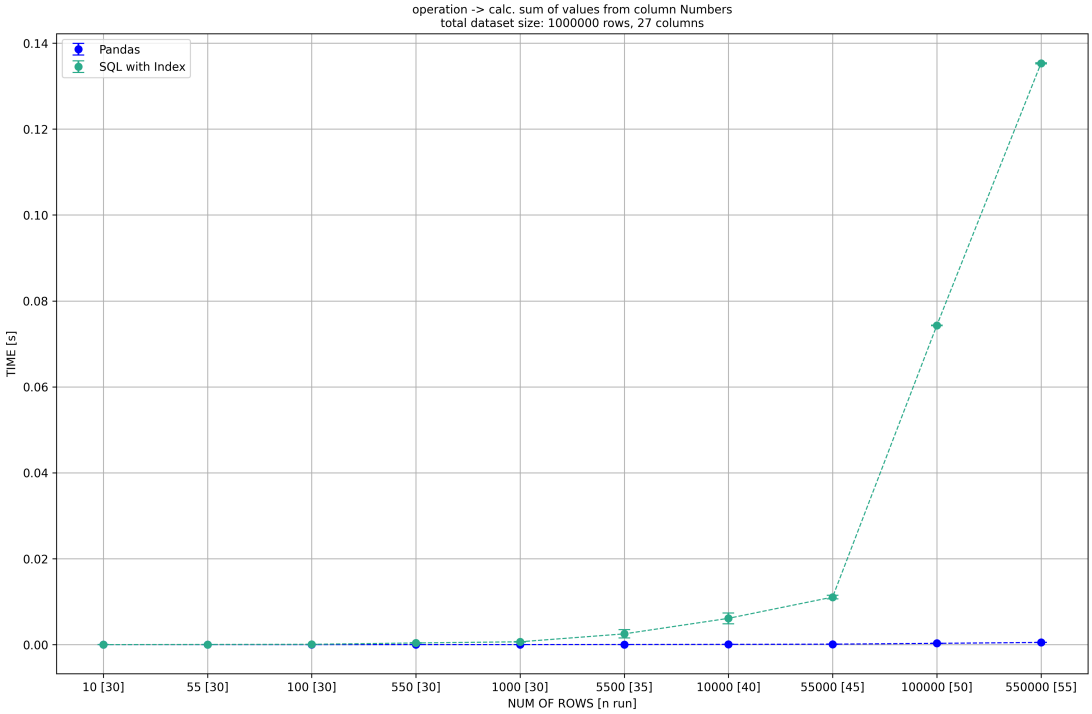


Risultati Pandas, Pandas + import, SQL e SQL index

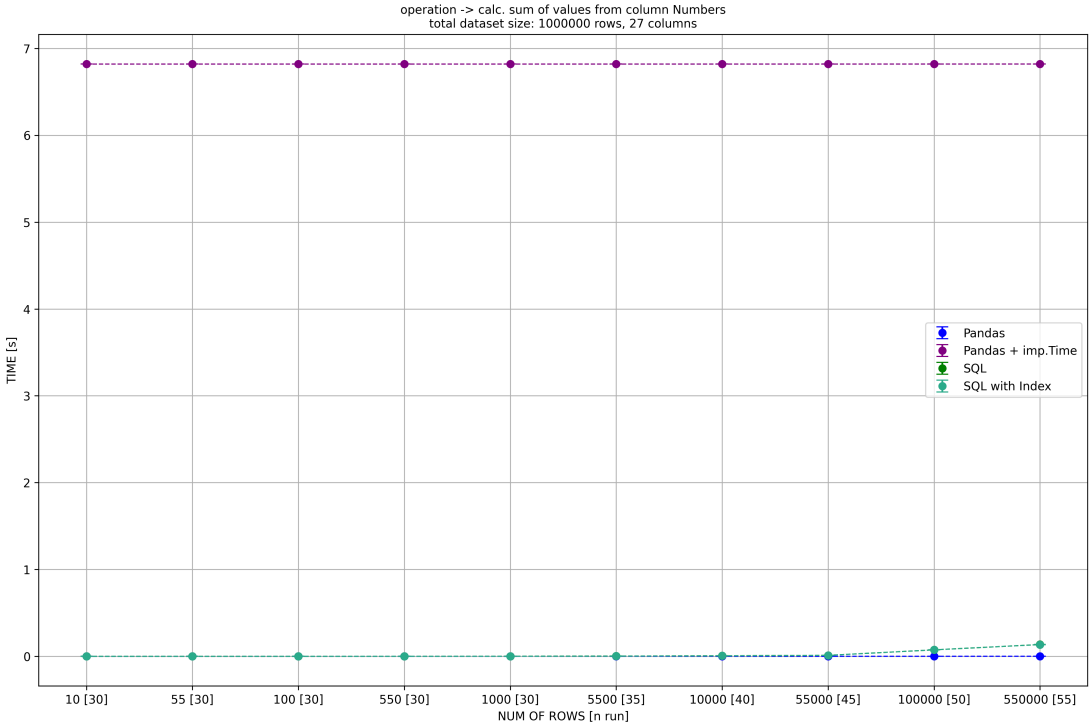
Test 6 - Calcolo della somma



Risultati Pandas e SQL

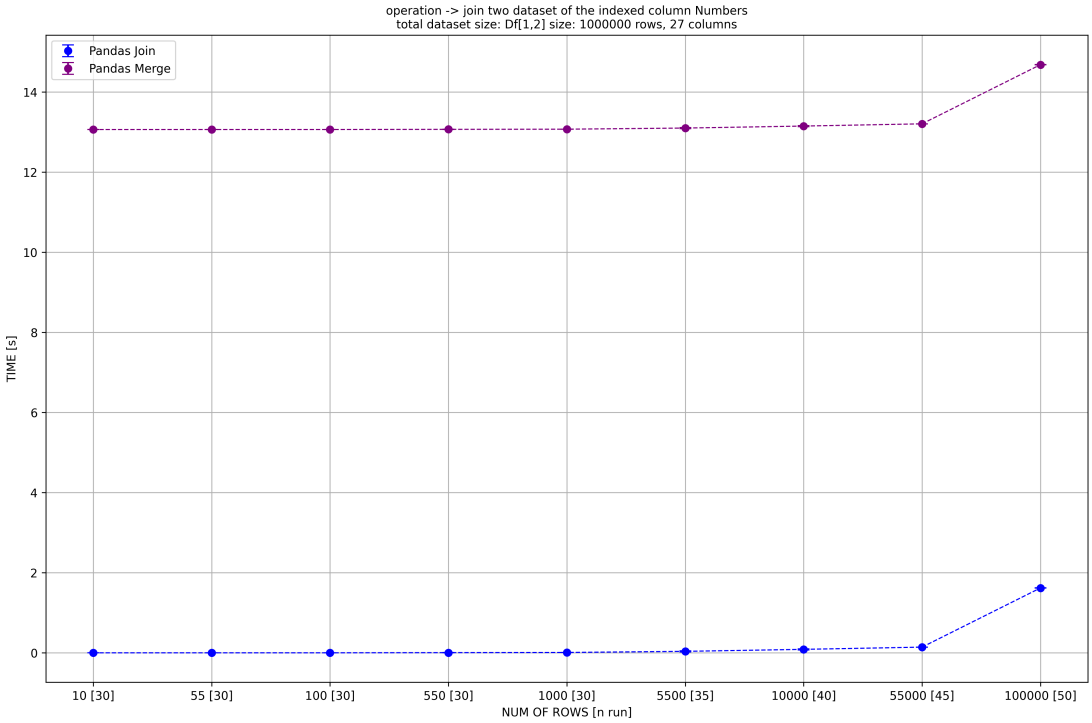


Risultati Pandas e SQL con indicizzazione

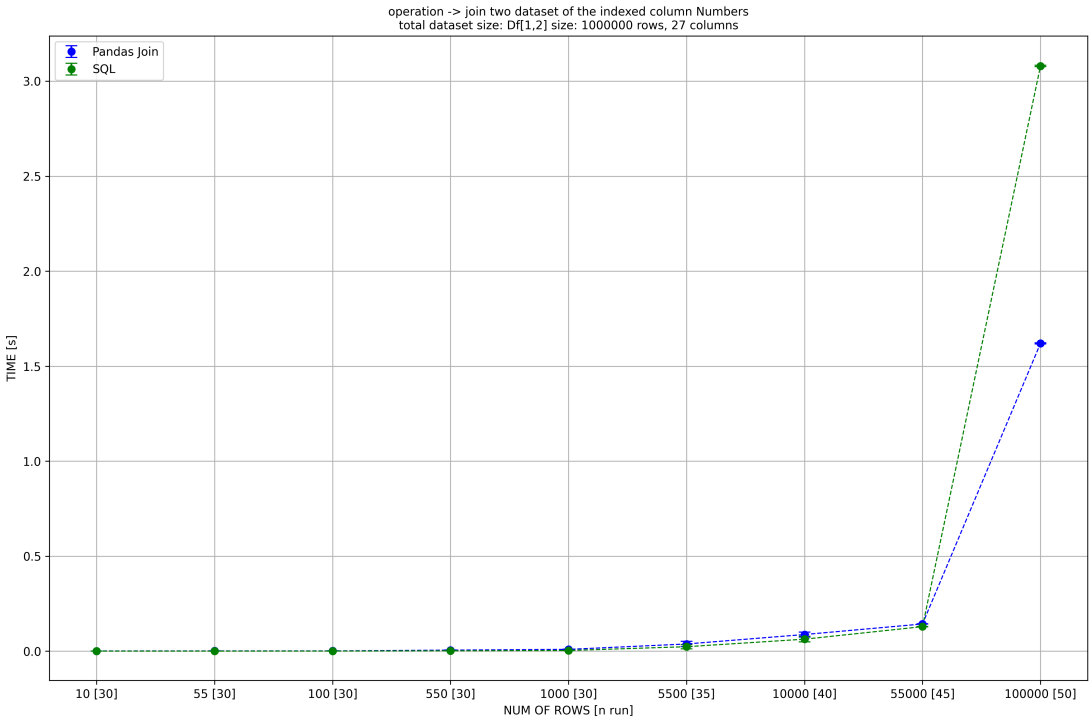


Risultati Pandas, Pandas + import, SQL e SQL index

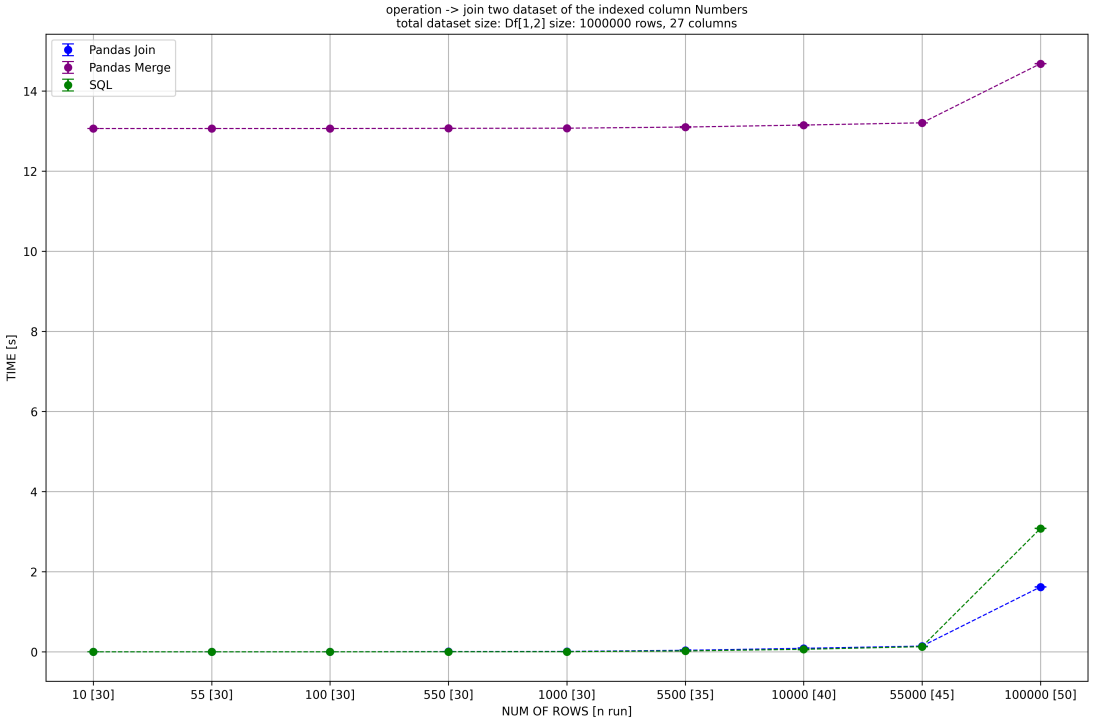
Test 7 - Join con indicizzazione



Risultati Pandas `join()` e Pandas `merge()`

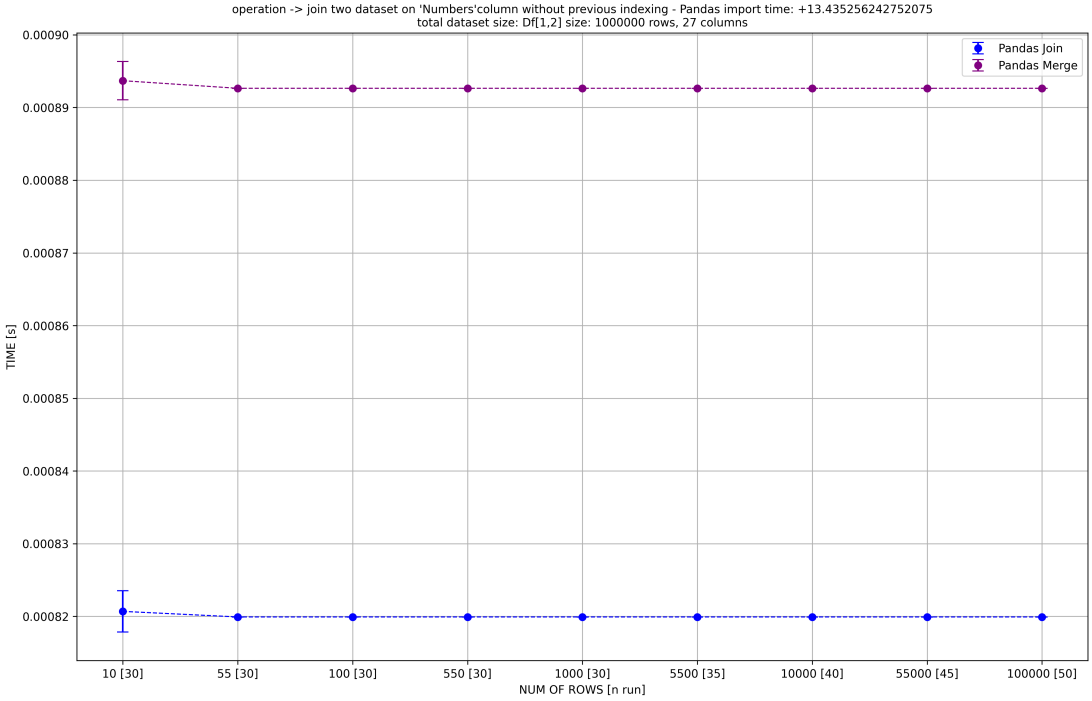


Risultati Pandas `join()` e SQL

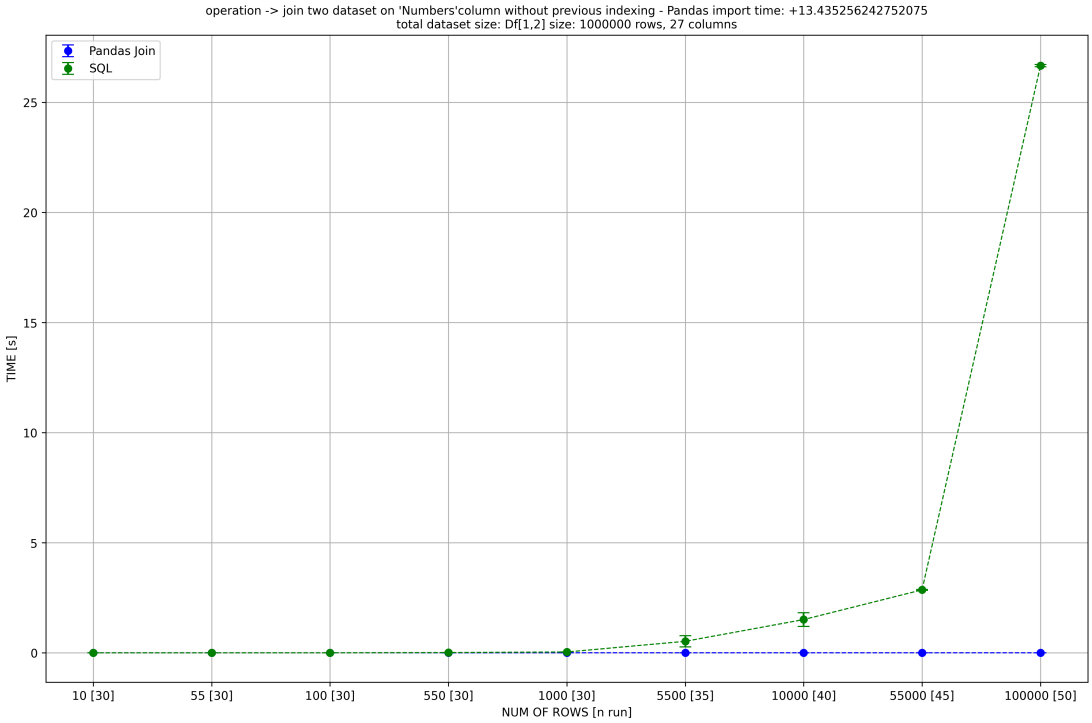


Risultati Pandas `join()`, Pandas `merge()` e SQL

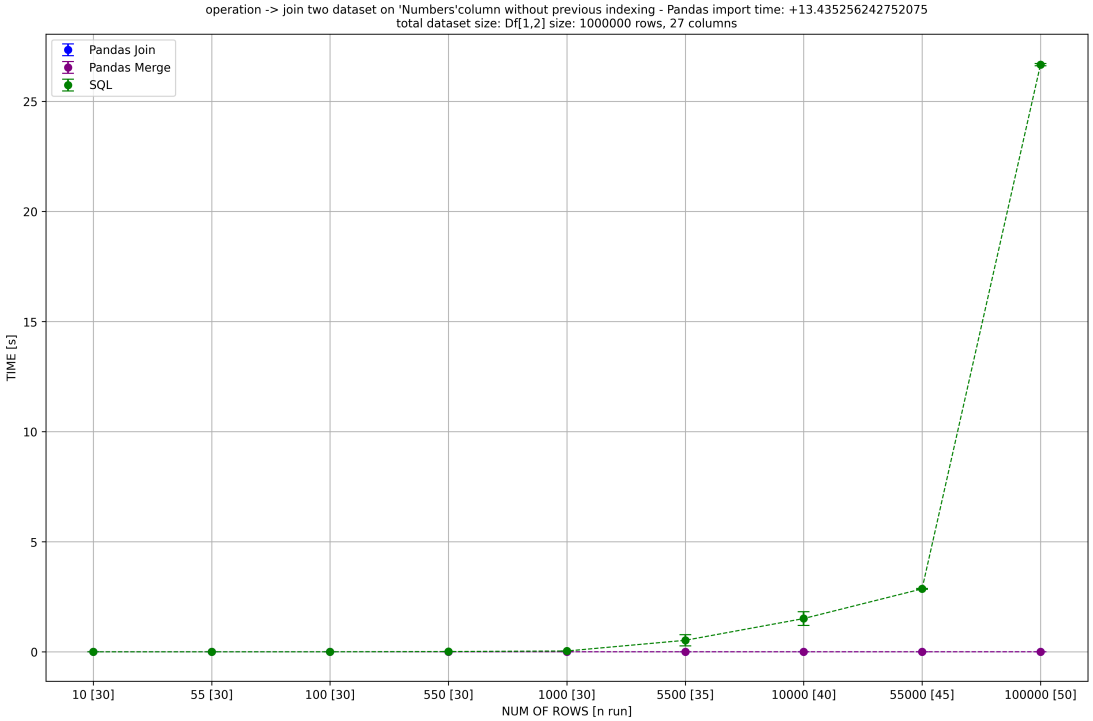
Test 8 - Join senza indicizzazione



Risultati Pandas `join()` e Pandas `merge()`



Risultati Pandas `join()` e SQL



Risultati Pandas `join()`, Pandas `merge()` e SQL

Bibliografia
&
Sitografia

[1] MCKINNEY, W. *Data Structures for Statistical Computing in Python.* Proceedings of the 9th Python in Science Conference, 2010, pp. 51-56.

[2] CODD, E. F. *A Relational Model for Large Shared Data Banks.* Communications of the ACM, 13(6), 1970, pp. 377-387.

[3] ANDERSON E., BAI Z., BISCHOF C., BLACKFROD S., DEMMEL J., DONGARRA J., *ET AL. LAPACK Users' Guide, 3rd Edition.* SIAM, 1999.

[4] LAU S., KROSS S., WU E., GUO P. J. *Teaching Data Science by Visualizing Data Table Transformations: Pandas Tutor for Python, Tidy Data Tutor for R, and SQL Tutor.* In 2nd International Workshop on Data Systems Education, June 23, 2023, Seattle, WA, USA. ACM, New York, NY, USA, pp. 50-54.

[5] F. NAHRSTEDT, M. KARMOUCHE, K. BARGIEL, P. BANIJAMALI, A. N. PRADEEP KUMAR, I. MALAVOLTA *An Empirical Study on the Energy Usage and Performance of Pandas and Polars Data Analysis Python Libraries.* IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS, VOL. 30, NO. 1, January 2024, pp. 197-205.

[6] YUNGYU ZHUANG, MING-YANG LU,*Enabling Type Checking on Columns in Data Frame Libraries by Abstract Interpretation.* Department of Computer Science and Information Engineering, National Central University, Taiwan. February 9, 2022, pp. 14418-14426.

[7] Z. LI, W. SUN , D. ZHAN, Y. KANG, L. CHEN, AL. BOZZON. R. HAI,*Amalur: The Convergence of Data Integration and Machine Learning.* IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, VOL. 36, NO. 12, DECEMBER 2024, pp. 7353-7355.

[8] EPPERSON W., GORANTLA V., MORITZ D., PERER A. *Dead or Alive: Continuous Data Profiling for Interactive Data Science.* IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS, VOL. 30, NO. 1, JANUARY 2024 pp. 197-205.

[9] C. ABERGER, A. LAMB, K. OLUKOTUN AND C. RE *LevelHeaded: A Unified Engine for Business Intelligence and Linear Algebra Querying.* IEEE 34th International Conference on Data Engineering (ICDE), Paris, France, 2018, pp. 449-460.

[10] D. LANEY *3D Data Management: Controlling Data Volume, Velocity, and Variety*. META Group Research Note, February 2001.

[11] GENBANK DATABASE.
https://www.ncbi.nlm.nih.gov/genbank/GenBankOverview.html

[12] NASA OPEN DATA PORTAL, *Mars orbital image (HiRISE) labeled data set version 3*, Planetary Data System Imaging Node, January 31 2023.
https://data.nasa.gov/Space-Science/Mars-orbital-image-HiRISE-labeled-data-set-version/egmv-36wq/about_data

[13] STATISTA *Global Database Market Report, 2022.*
https://www.statista.com/topics/2981/database-management-systems/

[14] ADITYA PARAMESWARAN *Pandas vs. SQL — Part 4: Pandas Is More Convenient*. Published in TDS Archive, Nov 24, 2022.
https://medium.com/data-science/pandas-vs-sql-part-4-pandas-is-more-convenient-8e9744e2cd10

[15] SLOAN DIGITAL SKY SURVEY (SDSS) DATABASE.
https://www.sdss.org/dr17/

[16] PANDAS OFFICIAL DOCUMENTATION.
https://pandas.pydata.org/docs/

[17] NUMPY OFFICIAL DOCUMENTATION.
https://numpy.org/doc/

[18] MYSQL OFFICIAL DOCUMENTATION.
https://dev.mysql.com/doc/

[19] SQLITE3 OFFICIAL DOCUMENTATION.
https://docs.python.org/3/library/sqlite3.html

[20] MATPLOTLIB OFFICIAL DOCUMENTATION.
https://matplotlib.org/stable/index.html

[21] P. YADAV *SQL vs Pandas*, SCALER TOPICS, 8 Dec 2022.

[22] ACM DIGITAL LIBRARY.
https://dl.acm.org/

[23] IEEE EXPLORE.
https://ieeexplore.ieee.org/

Ringraziamenti

La stesura di questa tesi sperimentale coincide con la fine del mio cammino universitario e sancirà inevitabilmente l'inizio di un nuovo ciclo della mia vita.
E' stato un percorso tumultuoso e scosceso, dal qual mai mi son lasciato cullare e che pur senza conoscer naufragio, m'ha sospinto a riva sulla più imperiosa delle onde.
Come nelle storie più avvincenti, questa somma conclusione mai avrebbe visto luce senza l'ausilio e l'affetto donatomi da persone preziose, che qui ringrazio senza fronzoli.
Grazie ai miei genitori Marcella e Valerio, a mio fratello Marco e a Gaia, per aver in fondo mai smesso di credere in me;
grazie ai miei zii Monica e Ercole, e alla nonna Rosanna, il cui appoggio ha sempre saputo raggiungermi;
grazie a tutti gli amici e le amiche che nonostante sapessero quanto per me non significasse nulla tutto questo, sapessero quanto per me significasse tutto questo.
Ringrazio infine il professor Marco Di Felice, per avermi concesso l'opportunità di svolgere sia il tirocinio che questa tesi sotto la sua attenta e cordiale supervisione, nonostante la mia non sempre ineccepibile disposizione.
E sul finir di queste pagine, sono grato allo spirito che dentro di me ancor s'agita e brucia e che indomito mai s'assopisce.

Enrico

LADY TERYL CORPORATION