ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Corso di Laurea in Ingegneria e Scienze Informatiche

B-Tree File System per piattaforme IoT

Tesi di laurea in:
Supervisor's course name

Relatore
Marco Antonio Boschetti

Candidato
Leonardo Marcaccio

I Sessione di Laurea
Anno Accademico 2023-2024

Abstract

Questa Tesi esplora il processo di studio, progettazione e implementazione di un prototipo di File System (FS) basato sulla struttura dati B-Tree, sviluppato per la piattaforma IoT IOtto di Onit S.p.A.

L'obiettivo principale è migliorare l'efficienza nella gestione e nel recupero dei dati, affrontando le problematiche tipiche di un FS come la scalabilità e l'ottimizzazione delle risorse.

Il lavoro comprende un'analisi dello stato dell'arte sui moderni FS, con particolare attenzione alla loro interazione con i sistemi IoT. Viene approfondito il principio di funzionamento del B-Tree, dimostrando come questa struttura dati possa essere sfruttata per realizzare una struttura performante e affidabile.

Inoltre, vengono descritte le fasi di progettazione e implementazione, evidenziando le soluzioni adottate per adattare il prototipo alle esigenze specifiche della piattaforma IOtto.

I risultati preliminari mostrano che il prototipo proposto garantisce significativi miglioramenti in termini di velocità di accesso ai dati e utilizzo delle risorse rispetto alle alternative tradizionali.

Optional. Max a few lines.

Contents

List of Figures

List of Listings

Chapter 1

Introduction

1.1 Background

1.1.1 Onit

1.1.2 IOtto

AGGIUNGI PRESENTAZIONE DELL'AZIENDA E DEL PRODOTTO

1.2 Descrizione del problema

L'attuale FS utilizzato dall'applicativo risulta non essere più ottimale nel contesto attuale, data la crescita e la mole dei dati gestiti. Risulta dunque necessario trovare un'alternativa che soddisfi le necessità attuali e future, in ambito di scalabilità e ottimizzazione.

1.3 Obiettivi di Tesi

L'obbiettivo della Tesi risulta essere quello di realizzare un prototipo di FS che sia scalabile, in grado di effettuare rapide letture e scritture, strutturalmente solido e ottimizzato nell'allocazione di spazio.

In particolare, il lavoro si concentra sulla progettazione e realizzazione di un prototipo alternativo al sistema preesistente, con l'intento non solo di mantenere

le prestazioni attuali, ma, ove possibile, di migliorarle.

1.3.1 Domande

- Quali tipi di dati devono essere memorizzati e gestiti?
- Qual è la quantità stimata di dati da gestire a regime?
- Quali saranno le dimensioni massime dei file e la granularità delle operazioni sui dati?
- Come garantire un basso consumo di risorse?
- Qual è la struttura di archiviazione più adatta?
- Il FS dovrà scalare per gestire un numero crescente di dispositivi IoT?
- Qual è la strategia per l'espansione dello spazio di archiviazione?
- Come gestire guasti hardware o interruzioni improvvise di alimentazione?
- È necessario prevedere meccanismi di backup o snapshot per i dati?
- Come si possono introdurre nuove funzionalità senza compromettere i dati esistenti?

1.3.2 Scopo

L'obiettivo finale è quello di apportare un significativo miglioramento al software esistente, sfruttando l'hardware già disponibile per ottenere prestazioni più elevate. Questo intervento mira a incrementare le capacità del prodotto, garantendo una maggiore efficienza e un'esperienza d'uso più soddisfacente per i clienti.

In particolare, il miglioramento si traduce nell'ottimizzazione delle funzionalità attuali e nell'ampliamento delle possibilità offerte dal sistema, lavorando direttamente sul software. Questo approccio consente di massimizzare il valore del prodotto, andando incontro alle crescenti aspettative degli utenti e mantenendo un vantaggio competitivo sul mercato.

1.4 Struttura della Tesi

Questa Tesi è strutturata in tre capitoli principali.

Il primo capitolo fornisce le basi teoriche del lavoro, offrendo una panoramica sui moderni FS e sulla loro connessione con il mondo dell'Internet of Thing (IoT). Viene inoltre approfondita la struttura dati B-Tree, evidenziandone le proprietà principali e la sua rilevanza rispetto al problema affrontato.

Il secondo capitolo si concentra sulla progettazione e sull'implementazione del prototipo di FS. Vengono descritte le scelte effettuate durante il processo di sviluppo e spiegato come il B-Tree sia stato utilizzato per soddisfare le esigenze specifiche della piattaforma IOtto.

Infine, il terzo capitolo discute i risultati ottenuti dalla valutazione del prototipo. Vengono inoltre presentate le conclusioni tratte dallo studio e delineate le possibili direzioni per future modifiche e sviluppi relativi al prodotto.

Chapter 2

State of the art

2.1 L'Internet of Things

Il concetto di Internet of Things, o in italiano Internet delle Cose, rappresenta un'evoluzione nell'utilizzo della rete Internet, in cui gli oggetti, o "cose", diventano riconoscibili e acquisiscono la capacità di comunicare dati attraverso la rete e di accedere a informazioni aggregate da altre fonti.

Uno dei primi utilizzi di questo termine risale al 2001, in un documento del centro Auto-ID relativo alla creazione del network EPC, concepito per tracciare automaticamente il flusso di beni nella catena di fornitura.

Tuttavia, non esiste un'origine univoca o un creatore effettivo del termine, poiché il concetto si è sviluppato progressivamente attraverso contributi di diversi ricercatori e innovatori nell'arco di tempo che va dai primi anni '90 fino al 2010.

Gli oggetti connessi, che costituiscono il cuore pulsante dell'IoT, sono definiti "oggetti intelligenti" (Smart Objects) e si caratterizzano per proprietà distintive come la capacità di identificarsi, connettersi, localizzarsi, elaborare dati e interagire con l'ambiente circostante.

Attraverso tecnologie come le etichette RFID (Identificazione a radiofrequenza) o i codici QR, questi oggetti e luoghi possono comunicare informazioni tramite la rete o dispositivi mobili.

L'obiettivo principale dell'IoT risulta essere dunque il tentativo di creare una mappatura elettronica del mondo fisico, attribuendo un'identità digitale agli oggetti e ai luoghi che lo compongono.

Sono nati inoltre organizzazioni come EPCglobal e Open Geospatial Consortium (o OGC) che lavorano per creare standard globali per la visibilità dei dati e l'utilizzo di sensori connessi via Web.

In conclusione, le applicazioni dell'IoT sono molteplici e variano dai processi industriali alla logistica, dalla logistica all'efficienza energetica, fino all'assistenza remota e alla salvaguardia ambientale.
[Uck11][HBE11][Cha13]

2.2 File System

Tutti i processi e le applicazioni informatiche necessitano di archiviare e recuperare informazioni, ed è possibile, durante la loro esecuzione di memorizzare una quantità limitata di informazioni all'interno del proprio spazio di indirizzamento.

Questo metodo di gestione dei dati però non viene sprovvisto di problematiche:

- Capacità limitata, in quanto lo spazio di indirizzamento virtuale limita la quantità di dati archiviabili, rendendolo inadeguato per applicazioni che richiedono un'ampia archiviazione di dati, come sistemi bancari o database.
- Le informazioni memorizzate nello spazio di indirizzamento di un processo sono altamente volatili e vengono perse al termine del processo stesso.
- Non è permesso l'accesso concorrente. Spesso è necessario che più processi possano accedere contemporaneamente alle stesse informazioni, ma il confinamento dei dati all'interno di un unico processo lo impedisce.

Per rispondere a queste sfide, l'archiviazione a lungo termine richiede il rispetto di tre requisiti fondamentali:

- La possibilità di memorizzare grandi quantità di informazioni.

- La persistenza delle informazioni, anche dopo la terminazione del processo.
- L'accesso simultaneo ai dati da parte di più processi.

Per affrontare queste problematiche, è stato sviluppato il concetto di File System, una componente fondamentale dei sistemi informatici progettata per permettere la gestione completa dei dati, inclusa la loro creazione, modifica, archiviazione e recupero all'interno di un sistema.

Un file system, al suo livello più alto, fornisce dunque un modo organizzato per archiviare, recuperare e gestire informazioni su supporti di archiviazione permanenti come i dischi.

Esistono diversi approcci alla gestione dell'archiviazione permanente, poiché il problema è ampio e offre numerose soluzioni possibili. [Gia98][TB15]

2.3 Categorizzazione dei File System

I file system possono essere suddivisi in diverse categorie in base a specifiche caratteristiche, tra cui il livello di organizzazione gerarchica. Di seguito, analizziamo due tipologie principali di file system in relazione alla loro struttura gerarchica.

2.3.1 Single-Level Directory Systems

Questo tipo di file system rappresenta la forma più basilare e semplice di organizzazione dei dati. Consiste in una singola directory, spesso denominata root, che contiene tutti i file del sistema.

Non esistono ulteriori livelli o sottodirectory: tutti i file sono archiviati in un unico spazio condiviso.

L'approccio a livello unico offre alcuni vantaggi distinti. In primo luogo, la semplicità della struttura lo rende facile da implementare e utilizzare, poiché non sono necessari meccanismi complessi per navigare tra directory o sottodirectory. Inoltre, la localizzazione dei file è immediata, poiché tutto è contenuto all'interno di una singola cartella, rendendo inutile un sistema di ricerca avanzato.

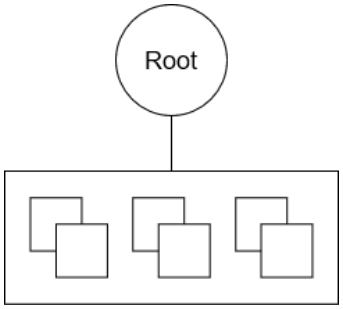

Figure 2.1: Esempio di Single Level Directory System

Tuttavia, questo modello ha limitazioni significative, soprattutto per applicazioni più complesse o ambienti che richiedono la gestione di un elevato numero di file.

Per questo motivo, i sistemi a directory singola sono utilizzati principalmente in dispositivi embedded o sistemi con esigenze limitate, come fotocamere digitali o dispositivi elettronici di consumo basilari. In tali contesti, la semplicità e la leggerezza dell'architettura superano le necessità di una maggiore organizzazione.

2.3.2 Hierarchical Directory Systems

A differenza del modello a singolo livello, i sistemi a directory gerarchica sono progettati per rispondere alle esigenze dei moderni ambienti informatici, caratterizzati dalla gestione di grandi volumi di dati e file.

In questo tipo di file system, è possibile creare una struttura organizzata composta da directory e sottodirectory. Questa struttura consente di suddividere i file in categorie o gruppi logici, favorendo una gestione più efficiente e un accesso più rapido alle informazioni.

L'organizzazione gerarchica risulta particolarmente utile quando il sistema deve gestire elevate quantità di file. La possibilità di classificare i file in directory dedicate consente di mantenere ordine e chiarezza, riducendo la complessità che deriverebbe dall'avere tutto in un'unica directory.

Un ulteriore vantaggio di questo approccio emerge nei contesti multiutente, dove più utenti condividono lo stesso file server. In questi casi, ogni utente può

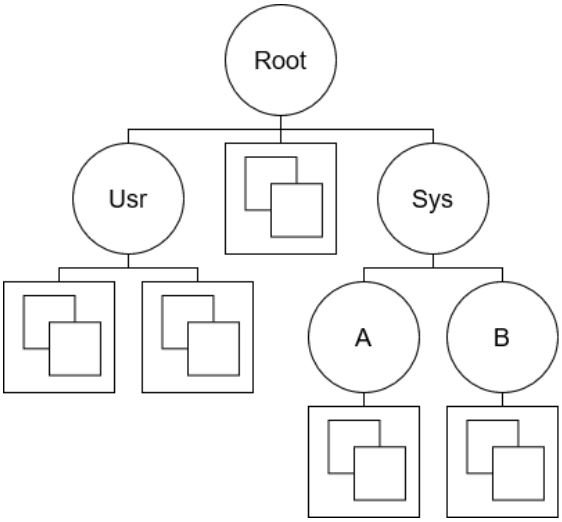

Figure 2.2: Esempio di Hierarchical Directory System

disporre di una directory root personale, che funge da punto di partenza per una gerarchia dedicata. Questa configurazione garantisce non solo la separazione dei dati tra gli utenti, ma anche un livello di personalizzazione che si adatta alle esigenze individuali.

In conclusione, mentre i sistemi a livello singolo sono ideali per applicazioni semplici e specifiche, i sistemi gerarchici rappresentano una soluzione robusta e scalabile per ambienti complessi e moderni.

[TB15]

2.4 Data Structures

Una struttura dati è un modo per memorizzare e organizzare i dati al fine di facilitare l'accesso e le modifiche.

Sono elementi fondamentali nella progettazione e nell'implementazione dei file system, poiché determinano come i dati vengono organizzati, memorizzati e recuperati, influenzando direttamente l'efficienza e le prestazioni delle operazioni sui dati.

Nessuna struttura dati singola è ottimale per tutti gli scopi, quindi è importante

conoscere i punti di forza e le limitazioni delle diverse strutture.

La scelta della struttura dati appropriata dipende da vari fattori, tra cui le esigenze specifiche del sistema, le prestazioni richieste e le limitazioni hardware.

Tra le strutture dati più comuni utilizzate nei file system, si possono citare:

- Linked List: una sequenza di nodi collegati tra loro, in cui ogni nodo contiene un riferimento al nodo successivo.

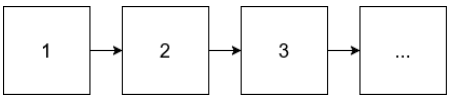

Figure 2.3: Esempio di Linked List

- Array: una collezione di elementi disposti in una sequenza lineare, in cui ogni elemento è accessibile tramite un indice numerico.

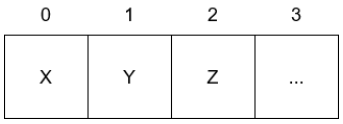

Figure 2.4: Esempio di Array

- Tree: una struttura gerarchica composta da nodi collegati tra loro, in cui ogni nodo può avere uno o più nodi figli.

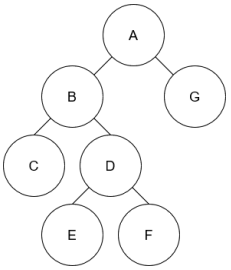

Figure 2.5: Esempio di Tree

- Hash Table: una struttura dati che associa chiavi a valori, consentendo un accesso rapido e diretto ai dati.

[CLRS22]

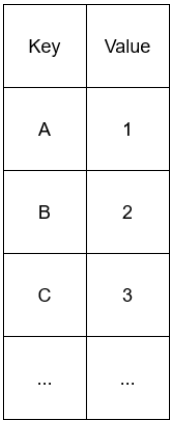

Figure 2.6: Esempio di Hash Table

2.5 Strutture Dati Basate sui Tree

Le strutture dati come Linked List e Array sono strumenti potenti per rappresentare relazioni lineari, dove gli elementi sono disposti in una sequenza.Tuttavia, non tutte le relazioni nel mondo reale sono lineari.

Molti problemi richiedono la rappresentazione di relazioni gerarchiche o ramificate, dove un elemento può essere connesso a più elementi in modo non sequenziale.

Un Tree è una struttura dati non lineare composta da nodi collegati tra loro in modo gerarchico.

Ogni albero ha un Nodo Radice denominato Root da cui partono uno o più Nodi denominati Figli, e ogni Figlio può a sua volta avere altri Figli, formando una struttura ramificata.

In questo capitolo esploreremo le principali tipologie di alberi, le loro caratteristiche, i vantaggi e gli svantaggi.

2.5.1 Binary Trees

Un Binary Tree è la variante più basilare dei Tree in cui ogni nodo può avere al massimo due Figli, chiamati Figlio Sinistro e Figlio Destro, ed è rappresentato tramite tre attributi per nodo: ‘Key‘ (Il Valore), ‘Left‘ (Figlio Sinistro) e ‘Right‘

(Figlio Destro).

Questa struttura è semplice, versatile e ampiamente utilizzata per realizzare strutture dati più avanzate.

Un Binary Tree risulta essere semplice, efficiente e flessibile, ma può risultare inefficiente in caso di sbilanciamento e non è adatto a rappresentare relazioni con più di due figli.

2.5.2 Binary Search Trees

Un Binary Search Tree è un tipo di Binary Tree in cui ogni nodo rispetta un ordine: il valore del nodo è maggiore di tutti i valori nel suo sottoalbero sinistro e minore di tutti quelli nel sottoalbero destro.

Questa struttura è utile per operazioni efficienti di ricerca, inserimento e cancellazione, e mantiene i nodi ordinati, semplificando operazioni come la ricerca del minimo o del massimo.

Tuttavia, se l'albero non è bilanciato, può degenerare in una lista collegata perdendo così tutti i suoi benefici.

2.5.3 Balanced Trees

I Balanced Trees, come AVL e Red-Black, sono Binary Search Trees che mantengono l'altezza logaritmica rispetto al numero di nodi per garantire efficienza.

Sia gli AVL che i Red-Black Trees utilizzano algoritmi e regole specifiche per mantenere l'equilibrio.

Entrambi offrono prestazioni garantite come ricerca, inserimento e cancellazione in tempi brevi, rendendoli ideali per applicazioni che richiedono velocità e prevedibilità.

Tuttavia, la gestione del bilanciamento richiede algoritmi più complessi rispetto ai BST standard, con un maggiore overhead computazionale dovuto alle operazioni di bilanciamento.

2.5.4 N-ary Trees

Un N-ary Tree è una generalizzazione dei Binary Trees in cui ogni nodo può avere fino a N Figli, con N come costante.

Può essere rappresentato tramite array o liste di puntatori ai figli ed è utile per relazioni con un numero fisso di figli, come negli alberi di decisione nei giochi di scacchi.

Il problema principale con gli N-ary Tree è la scelta del valore di N.

Se il valore è troppo piccolo, le operazioni risultano efficienti, ma può verificarsi uno spreco di memoria quando molti nodi hanno meno di N figli, e la struttura risulta rigida per relazioni con un numero variabile di figli.

Mentre se il valore è troppo grande, la struttura tende a diventare troppo simile ad una Linked List, con un aumento della complessità computazionale.

[CLRS22]

2.6 B-Tree

Un B-Tree è un Balanced Search Tree progettato per mantenere i dati ordinati e permettere operazioni di ricerca, inserimento e cancellazione in tempo logaritmico.

A differenza di altri Balanced Trees come i Red-Black Trees, i B-Tree sono ottimizzati per ridurre al minimo il numero di accessi al disco, rendendoli ideali per applicazioni che gestiscono grandi quantità di dati su dispositivi di memorizzazione secondaria.

Indicando con n il numero massimo di Nodi Figli a cui un Nodo può puntare, un B-Tree ha le seguenti proprietà:

- Il numero massimo di Chiavi per Nodo è $n - 1$.

- Il Nodo Root è o un Nodo Foglia o un Nodo Interno con un numero di Figli compreso tra 2 e n e un numero di chiavi compreso tra 1 e $n - 1$.
- Tutti gli altri Nodi Interni hanno un numero di Figli compreso tra $n/2$ e n.
- Tutte le Foglie si trovano alla stessa profondità.

Queste proprietà garantiscono che il B-Tree rimanga bilanciato, mantenendo un'altezza relativamente bassa anche con un grande numero di elementi.

Chapter 3

Design and Implementation

3.1 Analisi

Come precedentemente menzionato in section 1.1, la piattaforma IOtto rappresenta una soluzione IoT progettata per garantire il controllo completo degli impianti e dei macchinari ad essa connessi, ottimizzandone la manutenzione e la gestione energetica all'interno del contesto aziendale.

Tra le numerose funzionalità offerte dal sistema, spiccano quelle relative al monitoraggio e all'analisi dei dati raccolti, operazioni che richiedono un elevato numero di accessi alla memoria fisica del sistema stesso.

Queste funzionalità sono strettamente correlate alla scalabilità del sistema, che varia in base alle esigenze del cliente, con un numero di accessi alla memoria che può oscillare da poche centinaia a diverse decine di migliaia al giorno.

Il sistema attuale, sebbene sia in grado di gestire volumi elevati di operazioni di lettura e scrittura, non risulta ottimizzato sotto diversi aspetti. Ad esempio, il formato utilizzato per la storicizzazione dei dati, denominato MessagePack, si rivela inefficiente in termini di consumo di risorse, specialmente quando si tratta di memorizzare dati di piccole dimensioni.

Un ulteriore punto critico è rappresentato dal sistema di storicizzazione dei dati, basato su una struttura gerarchica tradizionale composta da cartelle e file.

Questo approccio comporta tempi di ricerca, inserimento e rimozione che crescono in modo lineare, una caratteristica che, sebbene accettabile per un file system generico, risulta inadeguata per gestire il volume elevato di dati che il sistema potrebbe essere chiamato a gestire.

Il software da sviluppare consiste in un prototipo che rappresenta una versione migliorata e ottimizzata dell'attuale file system. L'obiettivo del prototipo è introdurre ottimizzazioni che non alterino la struttura esistente, con particolare attenzione ai processi di gestione e storicizzazione dei dati.

3.1.1 Requisiti Funzionali

- Il sistema deve supportare un formato di serializzazione che, rispetto a soluzioni esistenti come MessagePack, sia ottimizzato per la gestione di dati di piccole dimensioni.
- È prevista l'implementazione di un sistema di indicizzazione il cui obiettivo è ridurre la complessità computazionale delle varie operazioni, passando da un algoritmo con complessità lineare a uno con complessità logaritmica.
- Il progetto prevede interventi strutturali finalizzati a migliore delle operazioni di ricerca, inserimento e rimozione, migliorando l'efficienza complessiva del sistema e riducendo i tempi di risposta alle richieste.
- Il sistema deve essere progettato per gestire un carico variabile di operazioni. Tale requisito implica che il sistema debba garantire performance costanti, attraverso una scalabilità sia verticale che orizzontale.
- Devono essere implementati meccanismi affidabili per il backup e il ripristino dei dati. Questi meccanismi devono assicurare l'integrità e la disponibilità delle informazioni, minimizzando il rischio di perdita di dati e garantendo un rapido recupero in caso di malfunzionamenti o incidenti.
- Il prototipo sviluppato dovrà integrarsi la struttura attualmente in uso. Tale integrazione deve avvenire senza introdurre impatti negativi sull'architettura

esistente, consentendo una transizione fluida e minimizzando eventuali disservizi.

3.1.2 Requisiti Non Funzionali

- Per garantire una manutenzione efficace e una facile estendibilità del sistema, è necessario implementare una documentazione esaustiva.

- Viene richiesta una struttura modulare dell'architettura, che consenta di aggiornare o sostituire singoli componenti senza dover riprogettare l'intero sistema.

- L'interoperabilità tra diverse piattaforme è un requisito fondamentale per garantire un'ampia adozione del sistema. La compatibilità con ambienti Linux e Windows, unitamente al supporto per architetture hardware sia x86 che ARM, assicura che il software possa essere implementato in contesti eterogenei.

- Una caratteristica essenziale dell'architettura è la capacità di integrare dinamicamente nuovi nodi di storage in un ambiente di produzione, senza interrompere il servizio.

- Il sistema deve prevedere un livello di configurabilità dei vari parametri, permettendo agli amministratori di ottimizzare il comportamento in funzione delle specifiche esigenze applicative e dei requisiti di performance.

3.2 Design

L'intera fase di design è avvenuta attraverso un processo iterativo, che ha coinvolto me e il team manager, con l'obiettivo di garantire che il prototipo rispondesse in modo efficace ai requisiti identificati.

Dopo alcune ricerche preliminari, abbiamo deciso di utilizzare un B-Tree come struttura dati principale per il prototipo, poiché, sulla base di un'attenta valu-

tazione delle prestazioni attese in scenari reali, esso si è dimostrato particolarmente adatto a gestire grandi volumi di dati garantendo elevate prestazioni.

Le caratteristiche che hanno reso questa struttura ottimale includono:

- **Bilanciamento Automatico**: garantisce complessità O(log n) per ricerche, inserimenti e cancellazioni, anche con dataset in crescita dinamica.
- **Efficienza nell'I/O su disco**: la struttura ad albero, con nodi di grado elevato, riduce il numero di accessi al disco, cruciale per dataset di dimensioni superiori alla RAM disponibile.
- **Scalabilità**: capacità di gestire milioni di record senza perdita prestazionale.

Un confronto con alternative come i Binary Trees (Soggetti al problema dello Sbilanciamento) o alle Hash Table (Inefficienti in operazioni di Range Query) ha confermato la superiorità del B-Tree nel contesto del progetto.

Inoltre, l'uso del B-Tree è supportato da letteratura consolidata in ambito di Database Systems, rafforzandone la validità come scelta progettuale, come ad esempio [Mos20].

Per ottimizzare l'uso della memoria e ridurre gli sprechi, è stato implementato un sistema customizzato di serializzazione/deserializzazione binaria, in sostituzione di MessagePack.

Le motivazioni includono:

- **Controllo Granulare**: la serializzazione manuale ha permesso di definire formati compatti per ogni dato, eliminando metadati superflui, come le Chiavi JSON, e utilizzando tipi a lunghezza fissa dove possibile, ad esempio interi a 32 bit invece di valori variabili.
- **Riduzione dell'Overhead**: rispetto a MessagePack, il formato custom riduce la dimensione dei dati serializzati, soprattutto grazie all'assenza di header descrittivi.
- **Flessibilità nelle modifiche**: la struttura binaria è progettata per essere estendibile senza impattare sulla backward compatibility, ad esempio riservando bit per flag futuri.

3.3 Architettura

Il sistema proposto si basa sulla rappresentazione di un intero Dataset all'interno di un singolo file di storicizzazione, strutturato in pagine (page) logicamente organizzate in una gerarchia ad albero. Questo approccio mira a coniugare efficienza nell'accesso ai dati, scalabilità e persistenza, adottando una filosofia simile a quella dei B-tree, ma con ottimizzazioni specifiche per scenari di scrittura sequenziale e query complesse.

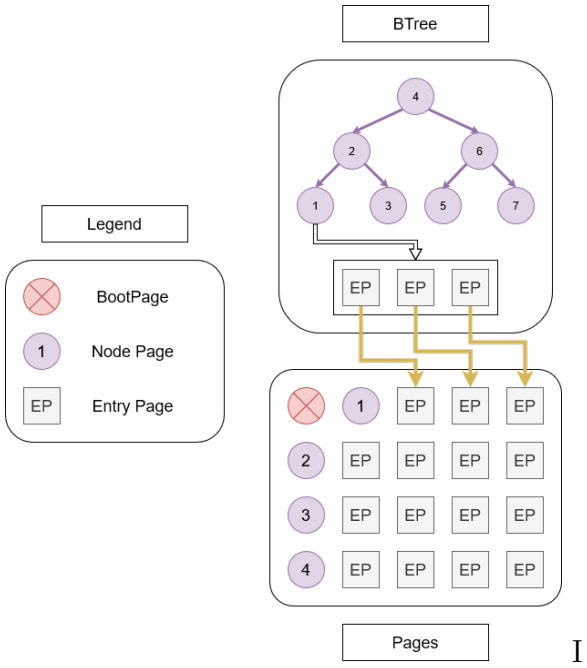

Figure 3.1: Rappresentazione schematica del prototipo

3.3.1 Componenti

- **Boot Page**

  La Boot Page rappresenta il punto di partenza per qualsiasi operazione sul sistema. Contiene metadati critici come informazioni relative allo stato dell'albero e un puntatore alla Root di quest'ultimo.

- **Node Page (e Root Page)**

Le Node Page incarnano i nodi interni dell'albero di indicizzazione. Ogni Node Page include:

  - Una lista ordinata di entry, dove ogni entry associa una chiave a un puntatore.
  - Riferimenti a pagine figlie, permettendo la navigazione verso livelli inferiori.
  - Diversi Metadati locali.

- **Entry Page**

  Le Entry Page contengono i dati effettivi associati alle chiavi. Ogni entry è strutturata come:

  - Identificatore univoco della Key.
  - Riferimenti a le Entry correlate.
  - Una lista dei dati corrispettivi alla specifica Key

3.3.2 Schema UML

Il sistema descritto precedentemente è stato modellato in un diagramma UML che ne riflette l'architettura logica e le relazioni tra componenti.

Un diagramma UML è un tipo di diagramma strutturale che rappresenta un sistema o un processo attraverso una serie di elementi grafici, come classi, interfacce e relazioni.

Nel nostro diagramma, la classe *BTree* funge da mediatore tra la logica ad albero e la storicizazione dei dati, gestendo le operazioni di ricerca, inserimento e cancellazione.

La classe *BTree* è contiene inoltre riferimenti anche al file di Log il quale conterrà le modifiche di riferimento per i controlli di sicurezza.

È presente in'oltre una relazione di composiziqone tra la classe *BTree* e la classe *Boot*, che rappresenta la pagina iniziale all'interno del file di sotricizazione.

La classe *Page* è una classe astratta che rappresenta una generica pagina all'interno del file di storicizzazione, e viene estesa dalle classi *Boot*, *Node* e *Entry*.

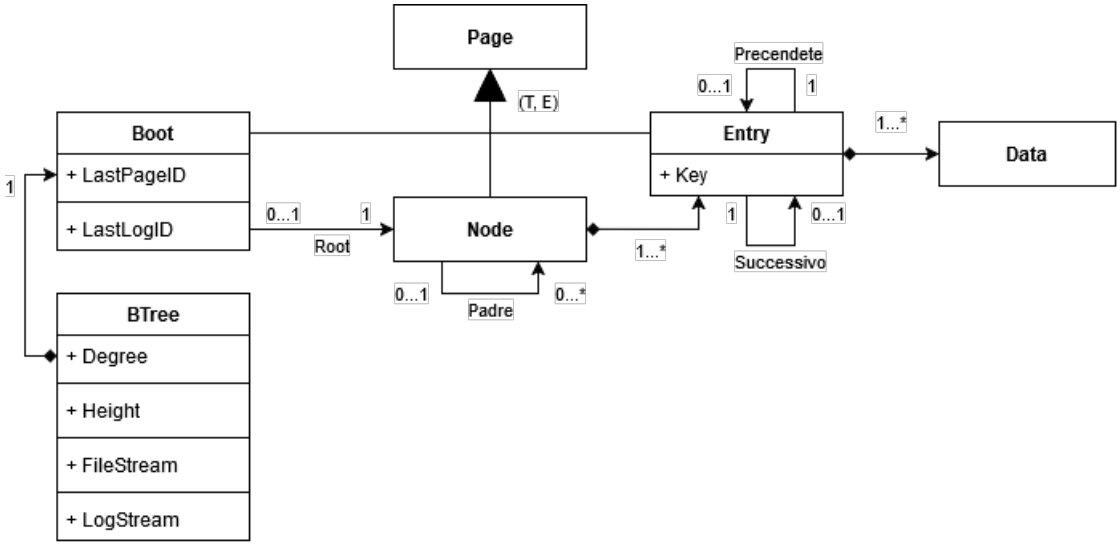

Figure 3.2: Diagramma UML del prototipo

Questa gerarchia favorisce il riuso del codice e l'isolamento delle responsabilità, riducendo l'accoppiamento tra logica di storage e operazioni sull'albero.

In conclusione, l'adozione del BTree come mediatore centralizza il flusso di controllo, semplificando l'aggiunta di nuove funzionalità, mentre la separazione tra Page e le sue implementazioni favorisce il riuso del codice e l'isolamento delle responsabilità, riducendo l'accoppiamento tra logica di storage e operazioni sull'albero.

3.4 Sviluppo

Segue ora una breve panoramica delle tecnologie utilizzate per lo sviluppo del prototipo.

3.4.1 Linguaggio di Programmazione

La scelta del linguaggio di programmazione è ricaduta su C#, in quanto è il linguaggio che è già stato usato per il prodotto.

C# è un linguaggio versatile e performante, con un'ampia base di utenti e una vasta libreria standard.

C# è particolarmente adatto per applicazioni di sistema, grazie alla sua portabilità, alla gestione della memoria automatica e alla robustezza.

3.4.2 Librerie

In supporto al linguaggio di programmazione è stato di notevolmente supporto l'impiego di diverse librerie, selezionate per implementare funzionalità specifiche e ottimizzare il flusso di lavoro.

In particolare, tra le librerie della famiglia `System` si evidenziano:

- **Gestione dei File Binari:** È stata impiegata la libreria `System.IO`, che fornisce un insieme di funzionalità avanzate per la lettura e la scrittura di file in formato binario.

  Questo strumento si è rivelato fondamentale per operazioni che richiedono un elevato grado di precisione nella manipolazione dei dati non testuali, garantendo un accesso efficiente e sicuro alle informazioni archiviate.

- **Gestione delle Strutture Dati:** La libreria `Collections.Generic` è stata utilizzata per sfruttare un'ampia gamma di strutture dati predefinite, quali liste, code e dizionari.

  Queste strutture permettono una gestione dinamica e performante dei dati, facilitando l'implementazione di algoritmi complessi e contribuendo a mantenere l'ottimizzazione delle risorse di sistema.

Per la gestione e realizzazione dei test, invece si è fatto utilizzo della libreria `BenchmarkDotNet` e `XUnit`.

La libreria `BenchmarkDotNet` è particolarmente apprezzata per le sue funzionalità avanzate nella misurazione delle performance del codice, offrendo strumenti di analisi dettagliata dei risultati.

Ciò ha permesso di identificare in maniera accurata eventuali colli di bottiglia e di ottimizzare ulteriormente l'efficienza computazionale dell'applicazione.

3.4.3 Strumenti di Sviluppo

Per la realizzazione del prototipo, è stato adottato un insieme di strumenti di sviluppo che hanno contribuito significativamente alla qualità e all'efficienza del processo di implementazione. In particolare, si evidenziano:

- **Visual Studio**: È stato scelto come ambiente di sviluppo integrato per C#. Visual Studio offre un ambiente completo e funzionale, caratterizzato da potenti strumenti per il debugging, l'analisi del codice e la gestione dei progetti.

  L'integrazione con altre tecnologie Microsoft ha permesso di accelerare le fasi di codifica e testing, contribuendo in modo determinante alla qualità del software prodotto.

- **Git**: Per il controllo versione del codice, si è fatto ricorso a Git, un sistema distribuito che consente di tracciare tutte le modifiche apportate durante lo sviluppo.

  Git ha facilitato la gestione della cronologia del progetto, permettendo di sperimentare modifiche in modo sicuro e coordinato, e di supportare un lavoro collaborativo efficace all'interno del team.

- **GitHub**: Come piattaforma di hosting per progetti basati su Git, GitHub ha offerto un ambiente collaborativo per la condivisione del codice e la gestione delle versioni.

L'integrazione di questi strumenti ha permesso di adottare un approccio allo sviluppo più dinamico e assicurando una gestione ottimale del ciclo di vita del software.

3.4.4 Sviluppo del Sistema di Serializzazione e Deserializzazione

La prima fase dello sviluppo del prototipo ha riguardato la progettazione e l'implementazione di un sistema dedicato alla serializzazione e deserializzazione dei dati.

L'obiettivo era quello di sostituire il formato MessagePack, attualmente in uso, con un formato binario ottimizzato, capace di ridurre la dimensione dei dati e di semplificare le operazioni di lettura e scrittura.

In una fase preliminare era già stata affrontata la problematica dell'eterogeneità dei dati attraverso la definizione di un formato univoco per i dati da storicizzare.

Pertanto, il focus di questa fase si è concentrato sulla riduzione dell'overhead e sul perfezionamento del processo di serializzazione.

Per raggiungere questi obiettivi sono state adottate le seguenti strategie:

1. **Calcolo della Dimensione degli Elementi:** È stato effettuato un calcolo accurato della dimensione di ogni singolo elemento del pacchetto dati, in modo da poter dimensionare in modo preciso l'allocazione dello spazio su di uno Stream binario.

2. **Utilizzo di Tipi a Lunghezza Fissa:** Dove possibile, sono stati impiegati tipi a lunghezza fissa per rappresentare i dati, standardizzando la codifica e minimizzando l'impatto sullo spazio occupato.

Il risultato di questo approccio è stato la definizione di un formato binario altamente efficiente, capace di ridurre la dimensione complessiva dei dati serializzati di circa il 60% rispetto a MessagePack, pur mantenendo prestazioni comparabili in termini di tempistiche di lettura e scrittura.

A dimostrazione dell'efficienza di questo formato è stato condotto un test di benchmarking, che ha confermato la riduzione della dimensione dei dati serializzati e l'ottimizzazione delle prestazioni.

Method	_contentEntryVersion	_oldSize	_newSize	Mean	Error	StdDev	Median	Rank	Allocated
Benchmark	BINARY	1	1	34.14 ms	1.585 ms	4.366 ms	32.95 ms	1	662.23 KB
Benchmark	BINARY	10	5	40.32 ms	0.923 ms	2.574 ms	40.26 ms	2	1168.36 KB
Benchmark	BINARY	100	1	40.33 ms	0.804 ms	2.090 ms	40.56 ms	2	1502.82 KB
Benchmark	BINARY	100	5	40.65 ms	0.810 ms	1.861 ms	40.56 ms	2	1518.12 KB
Benchmark	BINARY	50	100	40.81 ms	0.808 ms	2.071 ms	40.89 ms	2	1694.25 KB
Benchmark	BINARY	50	50	41.03 ms	0.945 ms	2.649 ms	40.89 ms	2	1502.38 KB
Benchmark	BINARY	10	1	41.09 ms	0.820 ms	2.102 ms	41.14 ms	2	1151.21 KB
Benchmark	MESSAGE_PACK	1	1	41.13 ms	0.925 ms	2.625 ms	41.14 ms	2	656.81 KB
Benchmark	MESSAGE_PACK	50	50	41.15 ms	0.864 ms	2.394 ms	41.40 ms	2	1502.38 KB
Benchmark	BINARY	100	10	41.26 ms	0.793 ms	1.112 ms	41.41 ms	2	1538.2 KB
Benchmark	BINARY	5	100	41.29 ms	0.826 ms	0.732 ms	41.51 ms	2	1518.45 KB
Benchmark	MESSAGE_PACK	10	10	41.55 ms	1.031 ms	2.924 ms	41.47 ms	2	1187.41 KB
Benchmark	BINARY	5	10	41.61 ms	1.469 ms	3.995 ms	41.13 ms	2	1168.33 KB
Benchmark	MESSAGE_PACK	100	1	41.83 ms	1.119 ms	3.175 ms	41.39 ms	2	1504.1 KB
Benchmark	MESSAGE_PACK	100	5	41.94 ms	0.941 ms	2.592 ms	41.68 ms	2	1518.2 KB
Benchmark	MESSAGE_PACK	5	50	42.08 ms	1.094 ms	3.087 ms	41.69 ms	2	1325.84 KB
Benchmark	MESSAGE_PACK	10	5	42.19 ms	1.274 ms	3.614 ms	41.56 ms	2	1171.17 KB
Benchmark	BINARY	100	100	42.22 ms	1.019 ms	2.841 ms	41.81 ms	2	1891.91 KB
Benchmark	MESSAGE_PACK	50	10	42.24 ms	1.056 ms	2.996 ms	41.82 ms	2	1343.82 KB
Benchmark	BINARY	100	50	42.29 ms	0.862 ms	2.403 ms	41.99 ms	2	1696.83 KB
Benchmark	BINARY	10	10	42.46 ms	1.048 ms	2.955 ms	42.10 ms	2	1191.63 KB
Benchmark	MESSAGE_PACK	5	10	42.71 ms	1.049 ms	2.943 ms	42.50 ms	2	1172.75 KB
Benchmark	MESSAGE_PACK	100	50	42.77 ms	1.019 ms	2.891 ms	42.36 ms	2	1696.48 KB
Benchmark	MESSAGE_PACK	10	100	42.94 ms	1.082 ms	3.034 ms	42.86 ms	2	1540.13 KB
Benchmark	BINARY	5	50	43.12 ms	0.888 ms	2.461 ms	42.77 ms	2	1323.48 KB
Benchmark	BINARY	50	10	43.13 ms	1.446 ms	4.125 ms	43.47 ms	2	1344.61 KB
Benchmark	MESSAGE_PACK	100	10	43.32 ms	1.072 ms	2.970 ms	42.97 ms	2	1543.03 KB
Benchmark	BINARY	10	100	43.33 ms	1.300 ms	3.646 ms	43.73 ms	2	1539.81 KB
Benchmark	BINARY	10	50	43.39 ms	1.095 ms	3.071 ms	42.99 ms	2	1347.08 KB
Benchmark	MESSAGE_PACK	10	50	43.40 ms	1.195 ms	3.391 ms	42.46 ms	2	1344.16 KB
Benchmark	BINARY	5	1	43.64 ms	1.421 ms	3.865 ms	43.11 ms	2	1137.55 KB
Benchmark	MESSAGE_PACK	10	1	43.85 ms	1.402 ms	4.001 ms	43.39 ms	2	1159.44 KB
Benchmark	BINARY	5	5	44.00 ms	1.057 ms	2.981 ms	44.21 ms	2	1153.23 KB
Benchmark	BINARY	50	5	44.04 ms	1.143 ms	3.223 ms	43.79 ms	2	1324.45 KB
Benchmark	BINARY	1	10	44.46 ms	1.667 ms	4.507 ms	44.57 ms	2	1155.88 KB
Benchmark	MESSAGE_PACK	50	1	44.55 ms	1.426 ms	3.999 ms	43.62 ms	2	1312.42 KB
Benchmark	MESSAGE_PACK	50	100	44.69 ms	1.537 ms	4.336 ms	44.08 ms	2	1696.36 KB
Benchmark	MESSAGE_PACK	100	100	44.70 ms	1.430 ms	4.011 ms	43.58 ms	2	1893.19 KB
Benchmark	MESSAGE_PACK	5	100	44.77 ms	1.258 ms	3.569 ms	44.80 ms	2	1525.65 KB
Benchmark	BINARY	50	1	45.28 ms	0.890 ms	1.462 ms	45.36 ms	2	1307.57 KB
Benchmark	BINARY	1	100	45.74 ms	1.696 ms	4.528 ms	45.03 ms	2	1505.45 KB
Benchmark	MESSAGE_PACK	1	10	45.97 ms	1.004 ms	2.848 ms	45.59 ms	2	1150.26 KB
Benchmark	MESSAGE_PACK	5	5	46.00 ms	1.667 ms	4.730 ms	46.16 ms	2	1150.95 KB
Benchmark	BINARY	1	5	46.34 ms	2.071 ms	5.806 ms	46.66 ms	2	1136.77 KB
Benchmark	MESSAGE_PACK	5	1	46.90 ms	1.172 ms	3.325 ms	46.69 ms	2	1131 KB
Benchmark	MESSAGE_PACK	1	5	46.94 ms	1.307 ms	3.707 ms	46.44 ms	2	1131.09 KB
Benchmark	MESSAGE_PACK	1	100	48.23 ms	1.040 ms	2.915 ms	48.51 ms	3	1503.09 KB
Benchmark	MESSAGE_PACK	1	50	48.50 ms	1.035 ms	2.869 ms	48.14 ms	3	1310.16 KB
Benchmark	BINARY	1	50	48.52 ms	2.174 ms	5.876 ms	47.37 ms	3	1309.99 KB
Benchmark	MESSAGE_PACK	50	5	49.10 ms	2.132 ms	6.049 ms	49.51 ms	3	1329.11 KB

Figure 3.3: Risultati del Benchmarking

3.4.5 Realizzazione del Btree

La fase successiva dello sviluppo ha riguardato la progettazione e la realizzazione del B-Tree, che ha rappresentaTo il nucleo centrale del prototipo sviluppato.

Questo capitolo descrive le soluzioni implementate per garantire un bilanciamento efficiente tra prestazioni e persistenza dei dati.

3.4.6 Analisi Preliminare e Riferimenti Teorici

Il processo di sviluppo è iniziato con un'analisi approfondita dello pseudocodice del B-Tree, tratto dal testo accademico *Introduction to Algorithms* [**?**].

Lo studio ha permesso di comprendere e integrare concetti fondamentali quali:

- Le politiche di Suddivisione(o Splitting) e Fusione (o Merging) dei nodi.
- Le strategie di ricerca delle chiavi all'interno della struttura.
- I vincoli tecnici imposti dall'albero per garantire un bilanciamento efficiente.

A partire da questa analisi, sono state identificate le classi e i metodi necessari per la realizzazione del B-Tree, delineando le interfacce e le relazioni tra i vari componenti del sistema.

Una volta definite le caratteristiche principali della struttura dati, si è passati alla fase di implementazione.

3.4.7 Implementazione e adattamento del codice

In questa fase sono state sviluppate le classi e i metodi necessari per la gestione delle operazioni fondamentali del B-Tree, tra cui ricerca, inserimento e cancellazione delle chiavi, assicurando il rispetto delle proprietà strutturali dell'albero.

Come base per l'implementazione, è stato adottato il codice open-source btree-dotnet [rsd], un'implementazione in C# che forniva lo scheletro delle classi e delle operazioni principali.

Tuttavia, la libreria presentava alcune criticità: non era compatibile con le versioni più recenti del linguaggio e conteneva errori nel meccanismo di bilanciamento dell'albero durante la fase di rimozione.

Per superare queste problematiche, è stato necessario apportare modifiche significative al codice, convalidandole successivamente attraverso una serie di test di controllo per garantire la correttezza e la stabilità del sistema.

Per risolvere queste problematiche, è stato necessario intervenire con modifiche sostanziali al codice, verificandone l'affidabilità attraverso una serie di test mirati.

Questi test hanno garantito la correttezza delle operazioni e la stabilità complessiva del sistema.

3.4.8 Implementazione delle Page

Successivamente, si è proceduto allo sviluppo delle Page e delle loro specifiche estensioni: BootPage, NodePage e EntryPage.

La classe Page rappresenta una pagina di memoria persistente, un elemento tipico dei sistemi di storage su file, come quelli utilizzati nei database. Questa classe è progettata per rispettare una serie di caratteristiche fondamentali, indispensabili per garantire la corretta gestione dei dati.

Attributi:

- **ID**: Un identificatore univoco che indica a quale page rappresentaa all'interno del file.
- **Type**: Un codice identificativo che specifica il tipo della pagina, distinguendo le varie specializzazioni.
- **Buffer**: Un array di byte che contiene il contenuto memorizzato nella pagina.
- **FreePageSize**: Un indicatore della quantità di spazio libero disponibile all'interno della pagina.
- **IsModified**: Un flag di controllo che segnala eventuali modifiche apportate alla pagina, fondamentale per le operazioni di storicizzazione e sincronizzazione dei dati.

Metodi:

- **Read**: Metodo per la lettura dei dati dal buffer della pagina, che aggiorna contestualmente i vari campi per garantire la coerenza delle informazioni.

```
public class Page
{
    public uint ID { get; protected set; }
    public byte Type { get; protected set; }
    public byte[] Buffer { get; protected set; }
    public int FreePageSize { get; protected set; }
    public bool IsModified { get; protected set; }

    public Page(uint Id, byte Type)
    {
        this._buffer = new byte[SysConstants.PAGE_SIZE];
        this.ID = Id;
        Array.Copy(
            BitConverter.GetBytes(ID),
            0,
            this._buffer,
            0,
            SysConstants.PAGE_ID_SIZE
        );
        this.Type = Type;
        this._buffer[4] = this.Type;
        this.FreePageSize = SysConstants.PAGE_SIZE - SysConstants.HEADER_SIZE;
        this.SetModified(true);
    }

    public Page(byte[] buffer)
    {
        this._buffer = buffer;
        this.FreePageSize = SysConstants.PAGE_SIZE - SysConstants.HEADER_SIZE;
        this.SetModified(true);
    }

    public virtual void Write()
    {
        Write(this._buffer);
    }

    public virtual void Write(byte[] buffer)
    {
        this.SetModified(false);
        Array.Copy(BitConverter.GetBytes(ID), 0, buffer, 0, SysConstants.
            PAGE_ID_SIZE);
        buffer[SysConstants.PAGE_ID_SIZE] = this.Type;
    }

    public virtual void Read()
    {
        this.FreePageSize = SysConstants.PAGE_SIZE - SysConstants.HEADER_SIZE;
        this.ID = BitConverter.ToUInt32(_buffer, 0);
        Type = _buffer[SysConstants.PAGE_ID_SIZE];
    }

    public void SetModified(bool val)
    {
        this.IsModified = val;
    }
}
```

- **Write**: Metodo responsabile della scrittura dei dati nel buffer, assicurando che le modifiche vengano correttamente registrate e mantenute.

3.4.9 Sviluppo della BootPage

La classe `BootPage`, introdotta come primo componente specializzato del sistema, funge da interfaccia primaria per la gestione delle pagine all'interno di un file.

Tale Page inoltre si occupa in modo centrale di orchestrare il riutilizzo degli spazi liberi presenti all'interno del file, garantendo così una gestione efficiente della memoria e un'ottimizzazione delle operazioni di I/O.

Attributi:

- **LastPageID**: Un indice che indica l'ultima pagina aggiunta alla fine del file, fungendo da riferimento per l'espansione del file stesso.
- **LastLogVersionID**: Un Indice che rappresenta la versione corrente del log del sistema, fondamentale per il tracciamento delle operazioni eseguite.
- **Buffer**: Un array di byte che contiene la trascrizione del contenuto della pagina, utilizzato per la lettura e la scrittura dei dati.
- **EmptyPageIndexes**: Una lista degli indici relativi alle pagine vuote presenti nel file. Questi spazi vengono prioritariamente riutilizzati, evitando così la creazione non necessaria di nuove pagine.
- **RootPageID**: L'indice della radice del nostro albero, che rappresenta il punto iniziale di riferimento per l'organizzazione delle altre pagine.

Metodi:

- **AddID**: Metodo responsabile dell'aggiunta di un nuovo ID alla lista di indici liberi, aggiornando così il set di pagine disponibili per il riutilizzo.
- **GetNewPageID**: Metodo che gestisce la generazione di un nuovo PageID. Se sono presenti pagine vuote (già indicate in `EmptyPageIndexes`), il metodo ne effettua il riutilizzo. In assenza di queste, ne viene creato uno nuovo.

```
public class BootPage<TK, TP> : Page where TK : IComparable<TK> where TP :
    IComparable<TP>
{
    private BTree<TK, TP> _BTree;
    public uint LastPageId { get; }
    public ulong LastLogVersionId { get; }
    public List<uint> EmptyPageIndexes { get; }
    public uint RootPageID { get; }

    public BootPage(BTree<TK, TP> BTree, uint Id) : base(Id, PageType.BOOT)
    {
        this._BTree = BTree;
        this._lastPageId = Id;
        Array.Copy(BitConverter.GetBytes(this._lastPageId), 0, this._buffer, 0,
            SysConstants.PAGE_ID_SIZE);
        this._lastLogVersion = 0;
        Array.Copy(BitConverter.GetBytes(this._lastLogVersion), 0, this._buffer,
            0, SysConstants.PAGE_ID_SIZE);
        this._emptyPageIndexes = new List<uint>();
    }

    public BootPage(BTree<TK, TP> BTree, byte[] buffer) : base(buffer)
    {
        this._BTree = BTree;
        Read();
    }

    public void AddID(uint ID)
    {
        this._emptyPageIndexes.Add(ID);
        this._emptyPageIndexes.Sort();
    }

    public uint GetNewPageId()
    {
        if (this._emptyPageIndexes.Any())
        {
            uint index = this._emptyPageIndexes.First();
            return index;
        }

        this._lastPageId++;
        this.SetModified(true);
        this._BTree._bufferedPages.TryAdd(this.ID, this);
        this._BTree._loggedPages.TryAdd(this.ID, this);
        Write();
        return this.LastPageId;
    }

    public ulong GetNewLogId()
    {
        this.SetModified(true);
        this._BTree._bufferedPages.TryAdd(this.ID, this);
        this._BTree._loggedPages.TryAdd(this.ID, this);
        return this._lastLogVersion++;
    }
}
```

- **GetNewLogID**: Metodo dedicato alla generazione di un nuovo LogID, aggiornando la versione del log e contribuendo al mantenimento della consistenza del sistema.

3.4.10 Sviluppo della NodePage

Successivamente, la classe `Node` dell'albero è stata riadattata nella forma di Page nella classe `NodePage`, apportando modifiche sia ai metodi che agli attributi per integrarsi al meglio nel nuovo sistema basato su pagine.

Questa trasformazione ha portato alla suddivisione del codice in due sezioni distinte:

- **Gestione dei nodi figli**: La sezione mostrata nel listing section 3.4.10 illustra come vengono gestiti i nodi figli all'interno di una `NodePage`. Oltre alle operazioni classiche di aggiunta, rimozione e ricerca, è stata implementata la funzione di controllo validità `CheckChildPage`. Questo metodo esegue controlli specifici per verificare la correttezza dei puntatori ai nodi figli, prevenendo così errori di accesso e garantendo la coerenza strutturale.

- **Gestione delle entry**: La sezione rappresentata nel listing section 3.4.10 descrive il sistema di gestione delle entry contenute nella `NodePage`. Analogamente a quanto avviene per i nodi figli, anche qui è stata integrata una funzione di controllo validità, `CheckEntryPage`, che verifica l'integrità delle entry e assicura la correttezza dei dati gestiti.

3.4.11 Sviluppo della EntryPage

Infine, è stata sviluppata la classe `EntryPage`, che rappresenta l'unità fondamentale dello storage del sistema.

Questa classe è progettata per gestire in maniera efficiente la memorizzazione e la manipolazione dei dati, garantendo che ogni operazione sullo storage venga eseguita in modo ordinato e affidabile.

I principali elementi di interesse di questa classe sono le 3 funzioni di gestione dei dati all'interno della EntryPage.

```
public class ChildWithPage
{
    public uint ChildPageId { get; set; }
    internal NodePage<TK, TP> _nodePage;
    public NodePage<TK, TP> NodePage
    {
        get
        {
            byte[] buffer = this._nodePage.Buffer;
            this._nodePage.Write(buffer);
            return new NodePage<TK, TP>(
                this._nodePage._BTree,
                this._nodePage._entryKeyByteConverter,
                this._nodePage._dataToBuffer,
                this._nodePage._keyNullIndicator,
                buffer
            );
        }
    }

    public ChildWithPage(uint childPageId)
    {
        this.ChildPageId = childPageId;
    }

    public ChildWithPage(NodePage<TK, TP> nodePage) : this(nodePage.ID)
    {
        this._nodePage = nodePage;
    }
}

private NodePage<TK, TP> CheckChildPage(ChildWithPage child)
{
    if (child._nodePage != null)
    {
        return child._nodePage;
    }
    if (this._BTree.BufferedPages.TryGetValue(child.ChildPageId, out var
        bufferedNodePage))
    {
        child._nodePage = bufferedNodePage as NodePage<TK, TP>;
    }
    else
    {
        var childNodePageOrBuffer = _BTree.GetPage(child.ChildPageId);
        child._nodePage = childNodePageOrBuffer.Page != null
            ? childNodePageOrBuffer.Page as NodePage<TK, TP>
            : new NodePage<TK, TP>(
                this._BTree,
                this._entryKeyByteConverter,
                this._dataToBuffer,
                this._keyNullIndicator,
                childNodePageOrBuffer.PageBuffer
            );
        this._BTree.BufferedPages[child.ChildPageId] = child._nodePage;
    }
    return child._nodePage;
}
```

```
public class EntryWithPage
{
    public TK Key { get; set; }
    public uint EntryPageID { get; set; }
    internal EntryPage<TK, TP> _entryPage;
    public EntryPage<TK, TP> EntryPage
    {
        get
        {
            return new EntryPage<TK, TP>(
                this._entryPage._BTree,
                this._entryPage._entryKeyByteConverter,
                this._entryPage._dataToBuffer,
                this._entryPage._keyNullIndicator,
                this._entryPage.Buffer
            );
        }
    }

    public EntryWithPage(uint entryPageId, TK key)
    {
        EntryPageID = entryPageId;
        Key = key;
    }

    public EntryWithPage(EntryPage<TK, TP> entryPage) : this(entryPage.ID,
        entryPage.Key)
    {
        _entryPage = entryPage;
    }
}

private EntryPage<TK, TP> CheckEntryPage(EntryWithPage entryWithPage)
{
    if (entryWithPage._entryPage != null)
    {
        return entryWithPage._entryPage;
    }
    if (_BTree.BufferedPages.TryGetValue(entryWithPage.EntryPageID, out var
        bufferedEntryPage))
    {
        entryWithPage._entryPage = bufferedEntryPage as EntryPage<TK, TP>;
    }
    else
    {
        var entryPageOrBuffer = _BTree.GetPage(entryWithPage.EntryPageID);
        entryWithPage._entryPage = entryPageOrBuffer.Page != null
            ? entryPageOrBuffer.Page as EntryPage<TK, TP>
            : new EntryPage<TK, TP>(
                this._BTree,
                this._entryKeyByteConverter,
                this._dataToBuffer,
                this._keyNullIndicator,
                entryPageOrBuffer.PageBuffer
            );
        _BTree.BufferedPages[entryWithPage.EntryPageID] = entryWithPage._entryPage
            ;
    }
    return entryWithPage._entryPage;
}
```

```
internal void AddData(TP data, byte[] dataBuffer, int position)
{
    if (dataBuffer.Length > this.FreePageSize)
    {
        throw new ArgumentException($"dataBuffer.Length > FreePageSize, dataBuffer
            .Length: {dataBuffer.Length}, FreePageSize: {this.FreePageSize}");
    }
    this._data.Insert(position, data);
    UpdateDataInBuffer();
}

internal void SetData(IEnumerable<TP> enumerableData)
{
    List<TP> dataToInsert = enumerableData.ToList();
    if (dataToInsert.Count * this._dataToBuffer.SizeInBytes() >
        GetMaxtFreePageSize())
    {
        throw new ArgumentException(
            $"dataBuffer.Count * SysConstants.PAGE_SIZE > FreePageSize, dataBuffer
                .Length * SysConstants.PAGE_SIZE: {dataToInsert.Count *
                SysConstants.PAGE_SIZE}, FreePageSize: {this.FreePageSize}"
        );
    }
    this._data = dataToInsert;
    UpdateDataInBuffer();
}

internal int FindPositionToInsert(TP data)
{
    return _data.TakeWhile(dataInEntry => data.CompareTo(dataInEntry) >= 0).Count
        ();
}
```

- **AddData**: Metodo dedicato all'aggiunta di nuovi dati nella `EntryPage`. Questo metodo assicura che i dati vengano inseriti correttamente all'interno della struttura, aggiornando opportunamente lo stato della pagina.
- **SetData**: Metodo che, a partire da una struttura enumerabile, consente la modifica dei dati presenti nella `EntryPage`. Questa funzione garantisce che le modifiche vengano applicate in modo coerente e affidabile, mantenendo la consistenza dei dati.
- **FindPositionToInsert**: Metodo incaricato di determinare la posizione ottimale in cui inserire un nuovo dato. Questo approccio garantisce una distribuzione equilibrata dei dati all'interno della pagina e contribuisce a ottimizzare l'utilizzo dello spazio disponibile.

3.4.12 Sviluppo della classe BTree

Come ultima classe del sistema è stata sviluppata la classe BTree, che rappresenta il mediatore tra la logica ad albero e la storicizzazione dei dati, gestendo le operazioni di ricerca, inserimento e cancellazione.

Creazione degli Elementi del BTree La precedente sezione di codice illustra come vengono creati gli elementi fondamentali del BTree: la radice, i nodi e le entry, che costituiscono la struttura portante del sistema.

- **CreateNewRoot**: Crea una nuova radice dell'albero generando una nuova istanza di `NodePage` con un ID univoco ottenuto dalla BootPage. Dopo la creazione, aggiorna l'ID della radice nel BootPage e aggiunge quest'ultimo alle collezioni di pagine loggate e bufferizzate.
- **CreateNewNode**: Genera un nuovo nodo non radice creando un'istanza di `NodePage` con un nuovo ID. Anche in questo caso, il BootPage viene registrato nelle collezioni di log e buffer prima della creazione del nodo.
- **CreateNewEntry**: Crea una nuova entry (istanza di `EntryPage`) per memorizzare una chiave specificata. Viene ottenuto un nuovo ID dalla BootPage, e il nuovo oggetto viene inizializzato con i parametri necessari, mentre alcuni

```
1  private void CreateNewRoot()
2  {
3      this._rootPage = new NodePage<TK, TP>(this, this.Degree, this.
           _entryKeyByteConverter, this._dataToBuffer, this._keyNullIndicator, this.
           _bootPage.GetNewPageId());
4      UpdateBootRootID(this._rootPage.ID);
5      this._loggedPages.TryAdd(this._bootPage.ID, this._bootPage);
6      this._bufferedPages.TryAdd(this._bootPage.ID, this._bootPage);
7  }
8
9  private NodePage<TK, TP> CreateNewNode()
10 {
11     this._loggedPages.TryAdd(this._bootPage.ID, this._bootPage);
12     this._bufferedPages.TryAdd(this._bootPage.ID, this._bootPage);
13     uint nodeId = this._bootPage.GetNewPageId();
14     return new NodePage<TK, TP>(this, this.Degree, this._entryKeyByteConverter,
           this._dataToBuffer, this._keyNullIndicator, nodeId);
15 }
16
17 private EntryPage<TK, TP> CreateNewEntry(TK key)
18 {
19     this._loggedPages.TryAdd(this._bootPage.ID, this._bootPage);
20     this._bufferedPages.TryAdd(this._bootPage.ID, this._bootPage);
21     uint entryId = this._bootPage.GetNewPageId();
22     return new EntryPage<TK, TP>( this, this._entryKeyByteConverter, this.
           _dataToBuffer, this._keyNullIndicator, entryId, key, null, null);
23 }
```

valori sono impostati su `null`. Anche qui, la BootPage viene aggiunta alle strutture di log e buffer.

Questi metodi garantiscono la corretta generazione e registrazione di nuovi elementi (radice, nodi e entry) all'interno della struttura, mantenendo la coerenza delle collezioni di pagine loggate e bufferizzate.

Metodi per l'inserimento nel BTree Questo frammento di codice mostra l'implementazione del metodo di inserimento di un nuovo elemento all'interno del BTree seguendo questi passaggi principali:

- **Inserimento nella radice non piena**: Se il nodo radice ha ancora spazio, la chiave e i dati vengono inseriti direttamente usando un metodo ricorsivo che, nel caso di nodo foglia, crea una nuova entry, oppure, se il nodo non è foglia, individua il figlio appropriato per continuare l'inserimento.
- **Gestione del nodo radice pieno**: Se la radice è piena, viene creata una nuova radice; il vecchio nodo radice viene aggiunto come figlio della nuova

```
public void Insert(TK newKey, IEnumerable<TP> data)
{
    if (!this._rootPage.HasReachedMaxEntries)
    {
        InsertNonFull(this._rootPage, newKey, data);
        UpdateLog(this._rootPage);
        WriteAllBuffersContents();
        Serialize();
        return;
    }
    NodePage<TK, TP> oldRoot = this._rootPage;
    CreateNewRoot();
    this._rootPage.AddChild(oldRoot);
    SplitChild(this._rootPage, 0, oldRoot);
    this._bufferedPages[this._rootPage.ID] = _rootPage;
    this._bufferedPages[oldRoot.ID] = oldRoot;
    InsertNonFull(this._rootPage, newKey, data);
    this.Height++;
    UpdateLog(this._rootPage);
    WriteAllBuffersContents();
    Serialize();
}

private void InsertNonFull(NodePage<TK, TP> node, TK newKey, IEnumerable<TP> data)
{
    int positionToInsert = node.GetPosition(newKey, true);
    if (node.IsLeaf)
    {
        EntryPage<TK, TP> entry;
        if (positionToInsert < node.GetEntriesCount())
        {
            var entryToBeMoved = node.GetEntry(positionToInsert);
            entry = CreateNewEntry(newKey);
            entryToBeMoved.InsertNewEntryBetweenEntryAndPrevious(entry);
        }
        else
        {
            var previousEntry = positionToInsert - 1 >= 0 ? node.GetEntry(
                positionToInsert - 1) : null;
            entry = CreateNewEntry(newKey);
            if (previousEntry != null)
            {
                previousEntry.InsertNewEntryBetweenEntryAndNext(entry);
            }
        }
        entry.SetData(data);
        node.InsertEntry(positionToInsert, entry);
        this._bufferedPages[node.ID] = node;
        this._bufferedPages[entry.ID] = entry;
        return;
    }
    NodePage<TK, TP> child = node.GetChild(positionToInsert);
    if (child.HasReachedMaxEntries)
    {
        this.SplitChild(node, positionToInsert, child);
        if (newKey.CompareTo(node.GetEntry(positionToInsert).Key) > 0)
        {
            positionToInsert++;
        }
    }
    InsertNonFull(node.GetChild(positionToInsert), newKey, data);
}
```

```
public void Delete(TK keyToDelete)
{
    this.DeleteInternal(this._rootPage, keyToDelete);
    if (this._rootPage.GetEntriesCount() == 0 && !this._rootPage.IsLeaf)
    {
        this._rootPage = this._rootPage.GetChild(0);
        this.Height--;
    }
    UpdateLog(this._rootPage);
    WriteAllBuffersContents();
    Serialize();
}
```

radice e suddiviso (split) per mantenere l'equilibrio dell'albero. Successivamente, si procede con l'inserimento ricorsivo nella nuova struttura.

- **Aggiornamenti e persistenza**: Dopo ogni inserimento, vengono aggiornati il log, la cache dei nodi modificati e viene eseguita la serializzazione dell'albero per garantire consistenza e persistenza dei dati.

Questo approccio assicura che l'albero rimanga bilanciato, consentendo operazioni di ricerca e inserimento efficienti.

Metodi per la rimozione nel BTree Segue ora una descrizione dei metodi implementati per la rimozione degli elementi dal BTree.

Il metodo Delete avvia il primo passo del processo di eliminazione a partire dalla radice dell'albero.

Dopo aver eseguito l'eliminazione, verifica se la radice è diventata vuota e non è una foglia; in tal caso, la sostituisce con il suo primo figlio e decrementa l'altezza dell'albero.

Successivamente, aggiorna il log, scrive i contenuti dei buffer e serializza la struttura per garantire la persistenza dei dati.

Il metodo DeleteInternal cerca la posizione della chiave da eliminare nel nodo corrente confrontandola con le chiavi presenti.

Se la chiave viene trovata nel nodo, chiama il metodo DeleteKeyFromNode per gestire l'eliminazione.

Se il nodo non è una foglia e la chiave non è presente, propaga l'eliminazione al sottoalbero appropriato, corrispondente all'intervallo di chiavi.

```
private void DeleteInternal(NodePage<TK, TP> node, TK keyToDelete)
{
    int i = node._entries.TakeWhile(entry => keyToDelete.CompareTo(entry.Key) > 0)
        .Count();
    if (i < node._entries.Count && node._entries[i].Key.CompareTo(keyToDelete) ==
        0)
    {
        this.DeleteKeyFromNode(node, keyToDelete, i);
        return;
    }
    if (!node.IsLeaf)
    {
        this.DeleteKeyFromSubtree(node, keyToDelete, i);
    }
}
```

Infine, il metodo DeleteKeyFromNode gestisce diversi scenari.

Se il nodo è una foglia, rimuove direttamente la chiave.

Se è un nodo interno, tenta di sostituire la chiave con il predecessore, la chiave massima del sottoalbero sinistro, se il figlio sinistro ha un numero di elementi maggiore o uguale al grado dell'albero.

Se ciò non è possibile, prova con il successore, la chiave minima del sottoalbero destro, se il figlio destro ha sufficienti elementi.

Se entrambi i figli hanno meno di 'Degree' elementi, fonde il nodo corrente con i suoi figli e propaga l'eliminazione nel nodo risultante, mantenendo così le proprietà del B-Tree.

Metodi per la ricerca e lo split nel BTree Infine, il metodo di ricerca è stato implementato per consentire la ricerca di una chiave all'interno del BTree.

Questo metodo inizia la ricerca dalla radice dell'albero, scendendo lungo i nodi interni fino a raggiungere una foglia.

Se la chiave è presente in una foglia, restituisce i dati associati.

Se la chiave non è presente, restituisce un valore nullo.

Questo approccio garantisce una ricerca efficiente e una gestione ottimale delle operazioni di lettura.

```
1  private void DeleteKeyFromNode(NodePage<TK, TP> node, TK keyToDelete, int
       keyIndexInNode)
2  {
3      if (node.IsLeaf)
4      {
5          node._entries.RemoveAt(keyIndexInNode);
6          return;
7      }
8      NodePage<TK, TP> predecessorChild = node._children[keyIndexInNode]._nodePage;
9      if (predecessorChild._entries.Count >= this.Degree)
10     {
11         EntryPage<TK, TP> predecessorEntry = GetLastEntry(predecessorChild);
12         DeleteInternal(predecessorChild, predecessorEntry.Key);
13         node._entries[keyIndexInNode] = new NodePage<TK, TP>.EntryWithPage(
               predecessorEntry);
14     }
15     else
16     {
17         NodePage<TK, TP> successorChild = node._children[keyIndexInNode + 1].
               _nodePage;
18         if (successorChild._entries.Count >= this.Degree)
19         {
20             EntryPage<TK, TP> successorEntry = GetFirstEntry(successorChild);
21             DeleteInternal(successorChild, successorEntry.Key);
22             node._entries[keyIndexInNode] = new NodePage<TK, TP>.EntryWithPage(
                   successorEntry);
23         }
24         else
25         {
26             predecessorChild._entries.Add(node._entries[keyIndexInNode]);
27             predecessorChild._entries.AddRange(successorChild._entries);
28             predecessorChild._children.AddRange(successorChild._children);
29             node._entries.RemoveAt(keyIndexInNode);
30             node._children.RemoveAt(keyIndexInNode + 1);
31             this.DeleteInternal(predecessorChild, keyToDelete);
32         }
33     }
34 }
```

```
public EntryPage<TK, TP>? Search(TK key)
{
    return this.SearchInternal(this._rootPage, key);
}

private EntryPage<TK, TP>? SearchInternal(NodePage<TK, TP> node, TK key)
{
    int i = node.Entries.TakeWhile(entry => key.CompareTo(entry.Key) > 0).Count();
    if (i < node.Entries.Count && node.Entries[i].Key.CompareTo(key) == 0)
    {
        return node.Entries[i]._entryPage;
    }
    if (node.IsLeaf)
    {
        return null;
    }
    else
    {
        EntryPage<TK, TP>? entryPage;
        foreach (var item in node.Children)
        {
            entryPage = this.SearchInternal(node.Children[i]._nodePage, key);
            if (entryPage != null)
            {
                return entryPage;
            }
        }
        return null;
    }
}
```

Bibliography

[Cha13] Hakima Chaouchi. *The internet of things: Connecting objects to the web*. John Wiley & Sons, 2013.

[CLRS22] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[Gia98] Dominic Giampaolo. *Practical file system design with the Be file system*. Morgan Kaufmann Publishers Inc., 1998.

[HBE11] Olivier Hersent, David Boswarthick, and Omar Elloumi. *The internet of things: Key applications and protocols*. John Wiley & Sons, 2011.

[Mos20] Salama A Mostafa. A case study on b-tree database indexing technique. *Journal of Soft Computing and Data Mining*, 2020.

[rsd] rsdcastro. Btree .net: Libraries for btrees in c#. `https://github.com/rsdcastro/btree-dotnet`.

[TB15] Andrew S Tanenbaum and Herbert Bos. *Modern operating systems*. Pearson Education, Inc., 2015.

[Uck11] D Uckelmann. *Architecting the Internet of Things*. Springer, 2011.

Acknowledgements

Optional. Max 1 page.