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:
Ricerca Operativa

Relatore
Marco Antonio Boschetti

Candidato
Leonardo Marcaccio

III Sessione di Laurea
Anno Accademico 2023-2024

Sommario

Questa Tesi esplora il processo di studio, progettazione e implementazione di un prototipo di *File System* basato sulla struttura dati *B-Tree*, sviluppato per la piattaforma *Internet of Things* (*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 *File System* come la scalabilità e l'ottimizzazione delle risorse.

Il lavoro comprende un'analisi dello stato dell'arte sui moderni *File System*, 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.

Alla mia famiglia, per il loro costante supporto e incoraggiamento.

Indice

Elenco delle figure

Elenco dei Codici

Capitolo 1

Introduzione

1.1 Struttura della Tesi

Questa Tesi è articolata in quattro capitoli principali.

- Questo primo capitolo introduce il problema affrontato, delineando il contesto di riferimento e gli obiettivi del lavoro.

- Il secondo capitolo fornisce le basi teoriche del lavoro, offrendo una panoramica sui moderni *File System* e sulla loro connessione con il mondo dell'*IoT*. Viene inoltre approfondita la struttura dati *B-Tree*, evidenziandone le proprietà principali e la sua rilevanza rispetto al problema affrontato.

- Il terzo capitolo si concentra sulla progettazione e sull'implementazione del prototipo di *File System*. 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 quarto 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.

1.2 Premesse e Contesto

1.2.1 Onit

Onit S.p.A. è un'azienda italiana all'avanguardia nel settore delle soluzioni IT e della consulenza, dedicata al miglioramento del management e dell'organizzazione aziendale.

L'azienda offre un portafoglio diversificato di soluzioni software che spaziano da *Operations & Supply Chain* a *Healthcare*, *Business Analytics*, *Web & Mobile* e *ICT System Integration*.

Si caratterizza per la sua capacità di integrare sistemi eterogenei e di supportare la trasformazione digitale nelle imprese, promuovendo innovazione, efficienza operativa e competitività nel mercato moderno.

1.2.2 IOtto

IOtto rappresenta la soluzione *IoT* sviluppata da Onit per favorire la connettività tra macchinari, servizi e persone, realizzando così l'ideale di Industria 4.0.

La piattaforma offre un controllo completo e in tempo reale degli impianti e dei macchinari, grazie a una serie di funzionalità integrate, tra cui:

- **IOtto Platform**: consente il monitoraggio e il controllo centralizzato degli impianti, supportando una completa digitalizzazione delle attività produttive.

- **IOtto Edge**: permette l'elaborazione e l'analisi dei dati direttamente "al margine" della rete, riducendo la latenza e ottimizzando il traffico informativo.

- **Asset Management System (AMS)**: facilita la gestione strategica della manutenzione, anticipando possibili anomalie e riducendo i tempi di intervento.

.

Questa soluzione consente di ottimizzare i processi produttivi, ridurre i costi operativi, offrendo alle aziende il controllo totale sui propri impianti e una gestione predittiva degli asset.

1.3 Descrizione del problema

L'attuale *File System* 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.

1.4 Obiettivo di Tesi

L'obiettivo della Tesi risulta essere quello di realizzare un prototipo di *File System* 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.

1.4.1 Domande

Segue ora un elenco di domande che sono state poste durante le prime fasi del progetto, al fine di identificare le esigenze e le aspettative del cliente.

- 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 File System 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?

1.4.2 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.

Capitolo 2

Stato dell'Arte

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[1] relativo alla creazione del network EPC.

Acronimo di Electronic Product Code, l'EPC è concepito come un identificativo universale che fornisce un'identità univoca per ogni oggetto fisico in qualsiasi parte del migliorando.

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.

[1] Auto-ID Labs è un gruppo di ricerca nel campo dell'identificazione a radiofrequenza (RFID) in rete.

Attraverso tecnologie come le *Etichette RFID* (Identificatori 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 nate inoltre organizzazioni come EPCglobal [2] e Open Geospatial Consortium (o OGC)[3] 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

I software, durante la loro esecuzione, si trovano di fronte alla sfida di dover gestire una grande quantità di informazioni. Per questo motivo, hanno bisogno di un sistema per archiviare temporaneamente i dati necessari al loro funzionamento. Una possibile soluzione è quella di utilizzare lo spazio di indirizzamento, un'area di memoria dedicata a questo scopo.

[2] organizzazione globale per la gestione dello standard EPC

[3] organizzazione internazionale no-profit, che si occupa di definire specifiche tecniche per i servizi geospaziali e di localizzazione.

Questo metodo di gestione dei dati però non è privo 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.

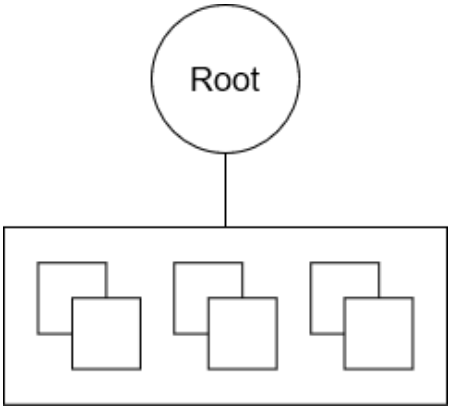

Figura 2.1: Esempio di Single Level Directory System

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 ciò che è presente all'interno del sistema è contenuto in una singola cartella, rendendo inutile un sistema di ricerca avanzato.

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 di base. 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.

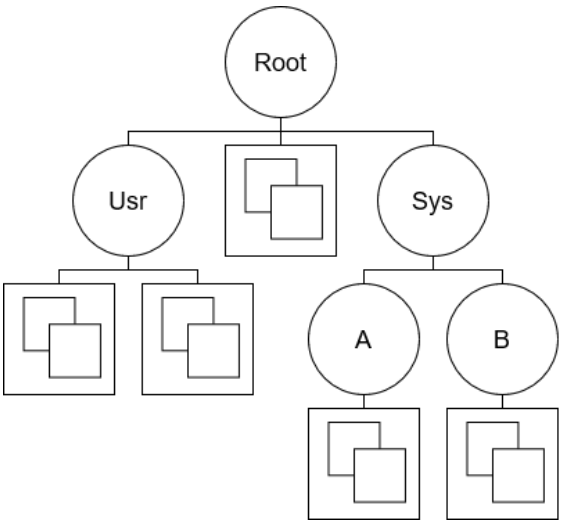

Figura 2.2: Esempio di Hierarchical Directory System

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

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 Le Strutture Dati

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.

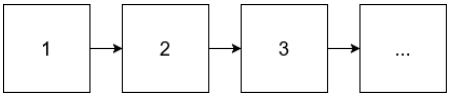

Figura 2.3: Esempio di Linked List

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

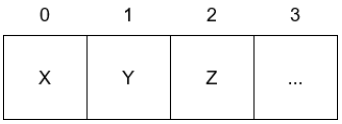

Figura 2.4: Esempio di Array

- ***Tree***: Una struttura gerarchica composta da nodi collegati tra loro in maniera aciclica. In essa, ogni nodo (eccetto la radice) ha un solo nodo genitore e può generare uno o più nodi figli.

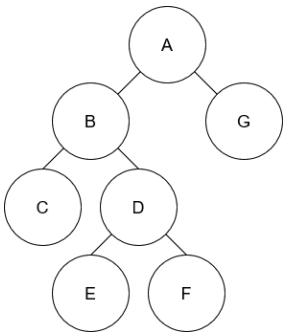

Figura 2.5: Esempio di Tree

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

Key	Value
A	1
B	2
C	3
...	...

I

Figura 2.6: Esempio di Hash Table

[CLRS22]

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* (abbreviato *BST*) 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, oltre a causare uno spreco di memoria, 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 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.

Capitolo 3

Design e Implementazione

3.1 Analisi

Come precedentemente menzionato nella sezione 1.2, 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 con 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 valutazione 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].

Inoltre, l'uso del *B-Tree* è supportato da una vasta letteratura nel campo dei *Database Systems*, come ad esempio lo studio [Mos20]. La loro efficacia è ampiamente riconosciuta e sfruttata in numerosi contesti, dai database relazionali ai file system.

In particolare, sono fondamentali per ottimizzare le prestazioni dei database relazionali come Oracle, Microsoft SQL Server, PostgreSQL, MySQL (con il motore InnoDB) e SQLite. Grazie alla loro struttura, consentono di eseguire ricerche e query su intervalli di dati in modo estremamente efficiente.

Anche a livello di file system, l'importanza dei B-tree è evidente. Sistemi come NTFS (Windows), ReFS, APFS (Apple) e alcuni file system Linux (ad esempio, ReiserFS e Ext4 con extent trees) utilizzano strutture simili per indicizzare i metadati.

In sintesi, l'adozione dei B-tree è una pratica consolidata sia nei database che nei sistemi di memorizzazione, poiché contribuisce a ottimizzare l'I/O e a garantire la scalabilità anche in presenza di grandi quantità di dati.

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 binaria 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, denominate *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.

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* rappresentano 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 corrispondenti alla specifica *Key*

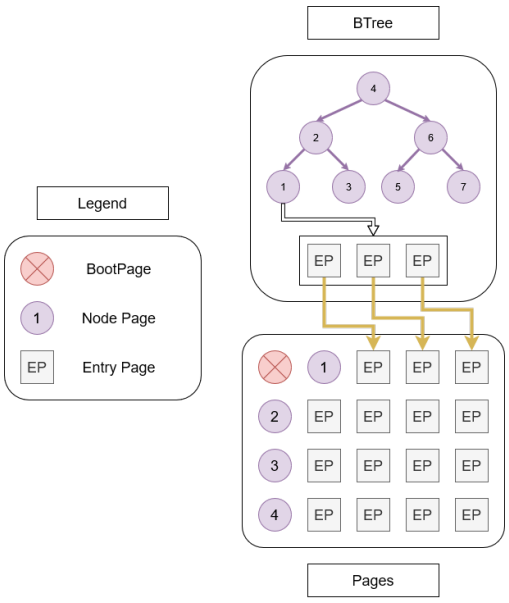

Figura 3.1: Rappresentazione schematica del prototipo

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.

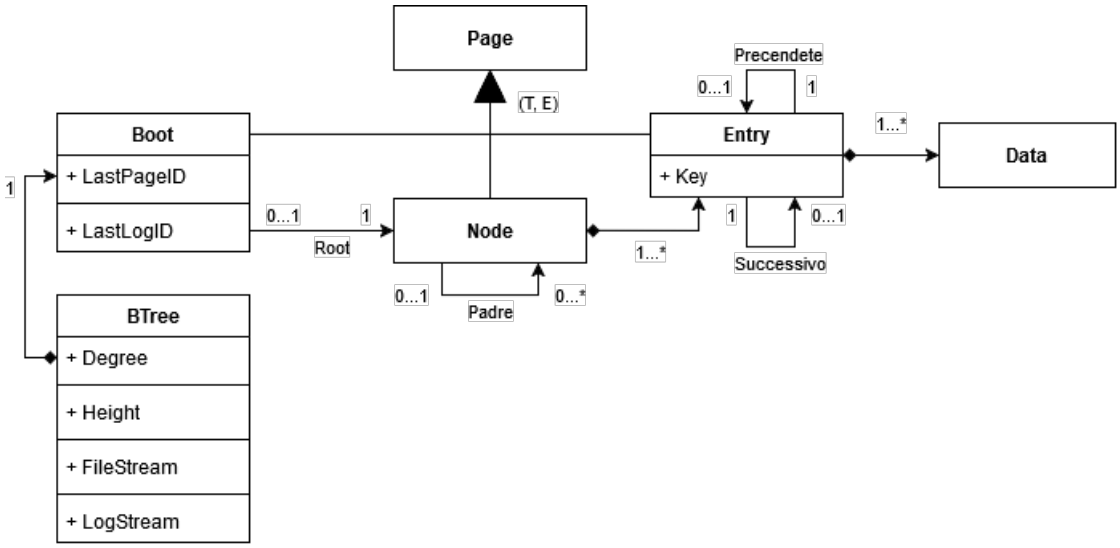

Figura 3.2: Diagramma UML del prototipo

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

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

È presente inoltre una relazione di composizione tra la classe `BTree` e la classe `Boot`, che rappresenta la pagina iniziale all'interno del file di sotricizzazione.

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`.

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.

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

3.4.2 Librerie

In aggiunta 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:

- **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.

- **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

Figura 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.

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

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** [CLRS22].

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.

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.

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

```
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;
    }
}
```

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.

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

```
1  public class BootPage<TK, TP> : Page where TK : IComparable<TK> where TP :
       IComparable<TP>
2  {
3      private BTree<TK, TP> _BTree;
4      public uint LastPageId { get; }
5      public ulong LastLogVersionId { get; }
6      public List<uint> EmptyPageIndexes { get; }
7      public uint RootPageID { get; }
8
9      public BootPage(BTree<TK, TP> BTree, uint Id) : base(Id, PageType.BOOT)
10     {
11         this._BTree = BTree;
12         this._lastPageId = Id;
13         Array.Copy(BitConverter.GetBytes(this._lastPageId), 0, this._buffer, 0,
               SysConstants.PAGE_ID_SIZE);
14         this._lastLogVersion = 0;
15         Array.Copy(BitConverter.GetBytes(this._lastLogVersion), 0, this._buffer,
               0, SysConstants.PAGE_ID_SIZE);
16         this._emptyPageIndexes = new List<uint>();
17     }
18
19     public BootPage(BTree<TK, TP> BTree, byte[] buffer) : base(buffer)
20     {
21         this._BTree = BTree;
22         Read();
23     }
24
25     public void AddID(uint ID)
26     {
27         this._emptyPageIndexes.Add(ID);
28         this._emptyPageIndexes.Sort();
29     }
30
31     public uint GetNewPageId()
32     {
33         if (this._emptyPageIndexes.Any())
34         {
35             uint index = this._emptyPageIndexes.First();
36             return index;
37         }
38
39         this._lastPageId++;
40         this.SetModified(true);
41         this._BTree._bufferedPages.TryAdd(this.ID, this);
42         this._BTree._loggedPages.TryAdd(this.ID, this);
43         Write();
44         return this.LastPageId;
45     }
46
47     public ulong GetNewLogId()
48     {
49         this.SetModified(true);
50         this._BTree._bufferedPages.TryAdd(this.ID, this);
51         this._BTree._loggedPages.TryAdd(this.ID, this);
52         return this._lastLogVersion++;
53     }
54 }
```

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.

La prima sezione 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.

Mentre la seconda sezione rappresenta 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.

```
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;
}
```

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`.

- `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.

```
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
        ();
}
```

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 successiva 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 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.

```
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 }
```

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 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.

```
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);
}
```

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.

```
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();
}
```

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`.

```
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 }
```

Metodi per la ricerca nel BTree Il metodo di ricerca in un `B-tree` permette di localizzare un elemento specifico all'interno dell'albero attraverso la sua chiave identificativa.

Questo processo si basa su una strategia ricorsiva che sfrutta l'organizzazione ordinata delle chiavi presenti nella struttura.

L'implementazione di tale metodo è strutturata in due metodi distinti: uno pubblico, che funge da punto di accesso esterno e avvia la ricerca a partire dal nodo radice dell'albero, e uno ricorsivo interno, incaricato di gestire l'atraversamento dei nodi e il percorso gerarchico all'interno della struttura.

La ricerca si articola in diverse fasi, ciascuna finalizzata a individuare la posizione corretta della chiave all'interno dell'albero.

Inizialmente, viene effettuata un'analisi del nodo corrente, l'obiettivo è determinare quante di queste chiavi presenti nel nodo risultano essere inferiori rispetto alla chiave che si sta cercando.

Tale conteggio permette di identificare l'indice del sottoalbero in cui la ricerca dovrà proseguire.

Contemporaneamente, si verifica se una delle chiavi presenti nel nodo corrente corrisponde esattamente alla chiave target. In caso affermativo, la ricerca si conclude con successo e l'elemento viene immediatamente restituito.

Qualora il nodo in esame sia una foglia, ovvero un nodo terminale dell'albero privo di ulteriori diramazioni, e la chiave non sia presente, la ricerca termina con un esito negativo, segnalando l'assenza dell'elemento attraverso la restituzione di un valore nullo.

Se, al contrario, il nodo non è una foglia, la ricerca prosegue in modo ricorsivo.

Viene selezionato il figlio corrispondente all'indice precedentemente calcolato e la funzione di ricerca viene richiamata su tale figlio, ripetendo il processo.

La natura ricorsiva dell'algoritmo garantisce una scansione completa dell'albero in profondità, fino a raggiungere un nodo foglia o a individuare una corrispondenza con la chiave cercata.

L'efficienza di questa operazione è notevole: grazie alla struttura bilanciata del B-tree, la ricerca ha una complessità logaritmica, proporzionale al logaritmo del numero totale di elementi presenti nell'albero.

La terminazione dell'algoritmo è sempre garantita: la ricerca raggiunge inevitabilmente un nodo foglia o identifica una corrispondenza, escludendo la possibilità di cicli infiniti.

```
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;
    }
}
```

Metodi di Utility del BTree Lo `Split`, e la sua controparte `Merge`, sono operazioni critiche attive durante l'inserimento (o l'eliminazione) di nuove chiavi, quando un nodo supera la capacità massima (o minima) definita dal *Grado* dell'albero.

Queste procedure preservano l'equilibrio strutturale del `B-tree`, condizione essenziale per le prestazioni logaritmiche.

A differenza delle operazioni di `Merge`, che sono state già integrate all'interno delle operazioni di eliminazione, lo `Split` è stato implementato come metodo autonomo.

```
1  private void SplitChild(NodePage<TK, TP> parentNode, int nodeToBeSplitIndex,
       NodePage<TK, TP> nodeToBeSplit)
2  {
3      NodePage<TK, TP> newNode = CreateNewNode();
4
5      parentNode.InsertEntry(nodeToBeSplitIndex, nodeToBeSplit.GetEntry(this.Degree
           - 1));
6      parentNode.InsertChild(nodeToBeSplitIndex + 1, newNode);
7
8      newNode.AppendEntries(nodeToBeSplit.GetEntriesRange(this.Degree, this.Degree -
            1));
9
10     // remove also _entries[this.Degree - 1], which is the one to move up to the
           parent
11     nodeToBeSplit.RemoveEntriesRange(this.Degree - 1, this.Degree);
12
13     if (!nodeToBeSplit.IsLeaf)
14     {
15         newNode.AppendChildren(nodeToBeSplit.GetChildrenRange(this.Degree, this.
               Degree));
16         nodeToBeSplit.RemoveChildrenRange(this.Degree, this.Degree);
17     }
18
19     parentNode.UpdateEntryReferences();
20     newNode.UpdateEntryReferences();
21     nodeToBeSplit.UpdateEntryReferences();
22 }
```

Il processo si articola in cinque fasi:

1. **Creazione di un nuovo nodo**: Viene allocato un nodo ausiliario per ospitare una porzione delle chiavi.

2. **Promozione della chiave mediana**: La chiave centrale del nodo saturo viene spostata nel nodo genitore, agendo da separatore.

3. **Collegamento del nuovo nodo**: Il nodo creato viene associato al genitore come figlio destro della chiave promossa.

4. **Ridistribuzione delle chiavi**: Le chiavi successive alla mediana vengono riposizionate all'interno del nuovo nodo, lasciando il nodo originale con *Grado - 1* elementi.

5. **Gestione dei figli (per nodi non foglia)**: Se il nodo splittato non è una foglia, i puntatori ai figli vengono ripartiti tra nodo originale e nuovo nodo, mantenendo la corretta gerarchia.

Al termine, i metadati dei nodi coinvolti (genitore, nuovo nodo e nodo originale) vengono aggiornati per riflettere la nuova configurazione.

Lo `Split` garantisce così una crescita controllata dell'albero, prevenendo sbilanciamenti e assicurando che tutte le operazioni mantengano una complessità logaritmica.

Metodi di Serializzazione e Deserializzazione del BTree L'atto di serializare ha lo scopo di rendere persistenti i dati, su un supporto di memorizzazione di massa (come un disco).

Questo processo avviene attraverso due fasi distinte: la prima fase riguarda la scrittura di un log, mentre la seconda fase si concentra sul salvataggio effettivo dei dati.

Nella fase iniziale, dedicata alla scrittura del log, il metodo itera sull'insieme delle `Page` contenute in una lista.

Per ciascuna di queste `Page`, viene creato un array di byte di dimensione fissa. Successivamente, il viene chiamato il metodo Write della pagina che provvede a popolare questo array con i dati specifici della pagina L'array così ottenuto viene poi scritto nel flusso di log.

Qualora si verifichi un errore durante questa fase di scrittura, viene generata un'eccezione, accompagnata dal messaggio "Unable to create a new log".

Una volta completata la scrittura di tutte le pagine di log, lo stream viene chiuso attraverso il metodo Dispose.

Successivamente, il codice tenta di aprire un nuovo file di log, utilizzando un percorso che combina la cartella di destinazione con un identificativo di log univoco, ottenuto attraverso la `BootPage`.

Dopo la scrittura e la riapertura del file di log, la lista di `Page` viene ricreata come una nuova istanza vuota.

Questa operazione ha lo scopo di liberare la memoria precedentemente occupata dalle pagine di log che sono state salvate.

La seconda fase del processo di serializzazione prevede la scrittura effettiva dei dati. In questa fase, il metodo itera sull'insieme delle pagine presenti nella lista di `Page` da serializzare.

Analogamente alla fase precedente, per ciascuna pagina viene creato un array di byte della dimensione predefinita, e la pagina stessa provvede a scrivere i propri dati all'interno di questo array.

Prima di scrivere definitivamente i dati nell'array, è fondamentale che il flusso di dati sia allineato con precisione a un punto specifico del file. Questo garantisce che le informazioni vengano memorizzate nella posizione corretta e che possano essere recuperate in modo accurato in seguito.

Qualora si verifichi un errore durante questa fase di scrittura, viene generata un'eccezione con il messaggio "Unable to create a new save".

Una volta completato il salvataggio di tutte le pagine, anche la lista di `Page` da serializzare viene ricreata come una nuova istanza vuota, preparandolo per future operazioni di buffering.

```
private void Serialize()
{
    try
    {
        foreach (var page in this._loggedPages.Values)
        {
            byte[] tmp = new byte[SysConstants.PAGE_SIZE];
            page.Write(tmp);
            this._logFileStream.Write(tmp);
        }
    }
    catch (Exception ex)
    {
        throw new Exception("Unable to create a new log");
    }
    this._logFileStream.Dispose();
    try
    {
        this._logFileStream = new FileStream(
            Path.Combine(this._folderPath, this._bootPage.GetNewLogId() + ".fslog"
                ),
            FileMode.OpenOrCreate,
            FileAccess.ReadWrite,
            FileShare.Read
        );
    }
    catch (Exception ex)
    {
        Console.WriteLine("Throwed exception : " + ex.Message + "in filestream or
            logstream opening");
    }
    this._loggedPages = new SortedDictionary<uint, Page>();
    try
    {
        foreach (var page in this._bufferedPages.Values)
        {
            byte[] tmp = new byte[SysConstants.PAGE_SIZE];
            page.Write(tmp);
            this._fileStream.Seek(page.ID * SysConstants.PAGE_SIZE, SeekOrigin.
                Begin);
            this._fileStream.Write(tmp);
        }
    }
    catch
    {
        throw new Exception("Unable to create a new save");
    }
    this._bufferedPages = new SortedDictionary<uint, Page>();
}
```

Nel caso della deserializzazione poniamo particolare interesse sulla deserializzazione delle `Entry`. Il metodo `DeserializeEntryFromID` ha proprio questo specifico compito. In altre parole, trasforma una sequenza di byte letta dal disco in un oggetto `EntryPage` utilizzabile in memoria.

Il processo inizia con il posizionamento del cursore di lettura all'interno del file.

Il metodo riceve un ID come parametro e lo utilizza per calcolare l'offset, ovvero la posizione precisa nel file dove si trovano i dati della pagina desiderata.

Una volta posizionati nel punto giusto del file, si procede con la lettura dei dati.

Viene creato un array di byte, chiamato buffer, con una dimensione pari a quella di una pagina.

Successivamente, il metodo legge esattamente un quantitativo di byte dal file pari alla dimensioni di una `Page`, a partire dalla posizione corrente del puntatore, e li memorizza all'interno del buffer.

Dopo aver recuperato i dati dal file, è necessario verificarne il contenuto per assicurarsi che corrispondano al formato di una pagina di tipo `Entry`. Se la verifica ha successo, il metodo crea una nuova istanza dell'oggetto `EntryPage`.

```
internal EntryPage<TK, TP>? DeserializeEntryFromID(uint ID)
{
    this._fileStream.Seek(ID * SysConstants.PAGE_SIZE, SeekOrigin.Begin);
    byte[] buffer = new byte[SysConstants.PAGE_SIZE];
    this._fileStream.Read(buffer, 0, SysConstants.PAGE_SIZE);
    if (buffer[4] == 3)
    {
        return new EntryPage<TK, TP>(
            this,
            this._entryKeyByteConverter,
            this._dataToBuffer,
            this._keyNullIndicator,
            buffer
        );
    }
    else
    {
        return null;
    }
}
```

3.4.6 Sviluppo dei Test

Come precedentemente accennato nella sezione 3.4.2, per la gestione e realizzazione dei test si è fatto utilizzo della libreria `XUnit`.

Grazie a questa libreria è stato possibile sviluppare una serie di test mirati e automatizzati, che hanno permesso di verificare la correttezza e l'efficienza del sistema.

In particolare, sono stati sviluppati test per verificare il corretto funzionamento delle operazioni di inserimento, ricerca ed eliminazione delle chiavi all'interno del `B-Tree`.

Per validare tali risultati sono stati realizzati diversi metodi di test, ciascuno focalizzato su un aspetto specifico del sistema. Esempio di questi controlli di validazione sono i test `ValidateTree` e `ValidateSubtree`.

Il metodo `ValidateTree` avvia la validazione dall'albero, raccogliendo le chiavi presenti in un dizionario e verificando che coincidano con quelle attese, senza duplicati.

Mentre `ValidateSubtree` opera in modo ricorsivo, controllando che ogni nodo, eccetto la radice, abbia:

- Il numero delle chiavi compreso nei limiti del BTree.
- Le chiavi ordinate correttamente entro intervalli specifici.
- I nodi interni contengano il numero minimo di figli richiesto.

In sostanza, il codice assicura che l'albero rispetti le proprietà strutturali e l'ordinamento tipico di un `B-Tree`.

```
public static void ValidateTree(NodePage<int, int> treeroot, int degree, params
    int[] expectedKeys)
{
    var foundKeys = new Dictionary<int, List<EntryPage<int, int>>>();
    ValidateSubtree(treeroot, treeroot, degree, int.MinValue, int.MaxValue,
        foundKeys);
    Assert.True(expectedKeys.Except(foundKeys.Keys).Count() == 0);
    foreach (var keyValuePair in foundKeys)
    {
        Assert.True(keyValuePair.Value.Count == 1);
    }
}

public static void ValidateSubtree(NodePage<int, int> root, NodePage<int, int>
    node, int degree, int nodeMin, int nodeMax, Dictionary<int, List<EntryPage<int
    , int>>> foundKeys)
{
    if (root != node)
    {
        Assert.True(node.Entries.Count >= degree - 1);
        Assert.True(node.Entries.Count <= (2 * degree) - 1);
    }
    for (int i = 0; i <= node.Entries.Count; i++)
    {
        int subtreeMin = nodeMin;
        int subtreeMax = nodeMax;
        if (i < node.Entries.Count)
        {
            var entry = node.Entries[i];
            UpdateFoundKeys(foundKeys, entry);
            Assert.True(entry.Key >= nodeMin && entry.Key <= nodeMax);

            subtreeMax = entry.Key;
        }
        if (i > 0)
        {
            subtreeMin = node.Entries[i - 1].Key;
        }
        if (!node.IsLeaf)
        {
            Assert.True(node.Children.Count >= degree);
            ValidateSubtree(root, node.Children[i], degree, subtreeMin, subtreeMax
                , foundKeys);
        }
    }
}
```

Inoltre, sono stati sviluppati test di stress per valutare le prestazioni del sistema in condizioni generiche, verificando la capacità del `B-Tree` di gestire qualsiasi combinazione di inserimenti e eliminazioni.

Il principale test per dimostrare ciò è il RunBruteForce. Il codice esegue un test brute force su un `B-Tree` del grado scelto. Procede con l'eseguire, per 1000 iterazioni, operazioni casuali di inserimento o cancellazione.

Dopo ogni operazione, viene chiamato il metodo `CheckNode`, che attraversa ricorsivamente ogni nodo per controllare che, se ci sono figli, il loro numero sia esattamente quello atteso e che né il numero di chiavi né quello dei figli superino i limiti imposti dal grado.

```
public static void RunBruteForce(int degree)
{
    var btree = new BTree<string, int>(degree);
    var rand = new Random();
    for (int i = 0; i < 1000; i++)
    {
        var value = (int)rand.Next() % 100;
        var key = value.ToString();
        if (rand.Next() % 2 == 0)
        {
            if (btree.Search(key) == null)
            {
                btree.Insert(key, value);
            }
            Assert.True(btree.Search(key).Pointer == value);
        }
        else
        {
            btree.Delete(key);
            Assert.Null(btree.Search(key));
        }
        CheckNode(btree.Root, degree);
    }
}

public static void CheckNode(NodePage<string, int> node, int degree)
{
    if (node.Children.Count > 0 &&
    node.Children.Count != node.Entries.Count + 1)
    {
        Assert.Fail("There are children, but they don't match the number of
            entries.");
    }
    if (node.Entries.Count > (2 * degree) - 1)
    {
        Assert.Fail("Too much entries in node");
    }
    if (node.Children.Count > degree * 2)
    {
        Assert.Fail("Too much children in node");
    }
    foreach (var child in node.Children)
    {
        CheckNode(child, degree);
    }
}
```

Capitolo 4

Conclusioni

4.1 Risultati e Valutazioni

Il prototipo sviluppato ha dimostrato di essere un sistema efficiente e scalabile per la gestione di grandi quantità di dati, garantendo prestazioni elevate e una gestione ottimale delle risorse.

Per dimostrazione, sono stati implementati vari benchmark che ci hanno permesso di ottenere le valutazioni desiderate. In ciascun benchmark, all'inizio di ogni iterazione viene creato un B-Tree di grado n contenente m chiavi.

Successivamente, per ogni combinazione di n e m vengono eseguite diverse operazioni—come inserimento, cancellazione e ricerca di un elemento—per valutarne le prestazioni.

Da questi benchmark sono stati poi estratti i seguenti risultati:

Method	Degree	InsertionNumber	Mean	Error	StdDev	Median	Rank	Gen0	Gen1	Gen2	Allocated
Benchmark	8	8	3.126 ms	0.2869 ms	0.8139 ms	2.942 ms	1	-	-	-	377.7 KB
Benchmark	2	8	6.016 ms	0.4778 ms	1.3936 ms	5.826 ms	2	-	-	-	574.38 KB
Benchmark	4	8	9.255 ms	1.8631 ms	5.4347 ms	7.987 ms	3	-	-	-	471.18 KB
Benchmark	8	64	27.447 ms	5.5836 ms	15.6571 ms	19.420 ms	4	-	-	-	4597.34 KB
Benchmark	4	64	32.565 ms	5.0935 ms	14.6143 ms	27.336 ms	5	-	-	-	5290.85 KB
Benchmark	2	64	40.528 ms	2.4527 ms	6.8371 ms	39.278 ms	6	1000.0000	-	-	7115.68 KB
Benchmark	8	512	235.938 ms	22.3300 ms	62.2471 ms	231.389 ms	7	7000.0000	1000.0000	-	43910.36 KB
Benchmark	4	512	279.948 ms	37.4608 ms	105.0442 ms	247.019 ms	8	7000.0000	1000.0000	-	47596.83 KB
Benchmark	2	512	472.972 ms	56.8627 ms	161.3101 ms	454.393 ms	9	10000.0000	2000.0000	1000.0000	63836.96 KB

Figura 4.1: Risultati del Benchmarking dell'inserimento

Method	Degree	InsertionNumber	Mean	Error	StdDev	Median	Rank	Gen0	Allocated
Benchmark	8	8	695.8 us	36.11 us	101.2 us	664.1 us	1	-	44.66 KB
Benchmark	4	8	895.7 us	46.65 us	130.0 us	843.5 us	2	-	102.14 KB
Benchmark	2	8	1,177.9 us	72.90 us	204.4 us	1,091.6 us	3	-	156.02 KB
Benchmark	8	64	8,027.8 us	508.37 us	1,442.2 us	7,827.1 us	4	-	614.98 KB
Benchmark	4	64	8,294.5 us	500.72 us	1,436.7 us	8,061.1 us	4	-	1007.61 KB
Benchmark	2	64	16,058.2 us	3,860.78 us	10,952.4 us	9,727.7 us	5	-	1966.74 KB
Benchmark	8	512	63,316.4 us	3,586.61 us	10,116.1 us	60,512.1 us	6	-	5662.84 KB
Benchmark	4	512	83,772.9 us	5,257.84 us	14,214.9 us	81,103.7 us	7	1000.0000	8228 KB
Benchmark	2	512	109,163.2 us	4,721.39 us	12,764.6 us	106,883.4 us	8	2000.0000	17338.63 KB

Figura 4.2: Risultati del Benchmarking dell'eliminazione

Method	Degree	InsertionNumber	Mean	Error	StdDev	Median	Rank	Allocated
Benchmark	8	8	6.733 us	0.7919 us	2.181 us	6.100 us	1	5.7 KB
Benchmark	2	8	10.752 us	0.6568 us	1.831 us	10.050 us	2	7.34 KB
Benchmark	4	8	11.010 us	0.7115 us	1.983 us	10.100 us	2	7.93 KB
Benchmark	8	64	29.535 us	1.4170 us	3.927 us	28.700 us	3	83.46 KB
Benchmark	4	64	63.826 us	10.0682 us	29.369 us	65.750 us	4	98.93 KB
Benchmark	2	64	68.759 us	11.8292 us	34.693 us	49.400 us	5	133.15 KB
Benchmark	8	512	328.730 us	14.5198 us	40.475 us	319.700 us	6	1048.49 KB
Benchmark	4	512	354.582 us	13.6596 us	38.303 us	353.100 us	7	1135.21 KB
Benchmark	2	512	590.573 us	28.0531 us	79.124 us	562.850 us	8	1784.37 KB

Figura 4.3: Risultati del Benchmarking della ricerca

I risultati evidenziano come la scelta del grado del B-Tree influenzi in modo significativo le prestazioni del sistema, con impatti misurabili sui tempi di esecuzione e sull'utilizzo della memoria.

L'analisi dimostra che, se calibrato con parametri ottimali, il sistema è capace di elaborare migliaia di operazioni al secondo mantenendo un elevato livello di affidabilità.

L'implementazione del `B-Tree` ha dimostrato quindi di essere robusta e affidabile, garantendo un bilanciamento ottimale tra prestazioni e persistenza dei dati.

I test sviluppati hanno confermato la correttezza e l'efficienza del sistema, dimostrando la capacità del B-Tree di gestire un elevato numero di operazioni in modo rapido e affidabile.

Inoltre, il prototipo ha dimostrato di essere altamente scalabile, consentendo di adattarsi facilmente a nuove esigenze e di gestire grandi quantità di dati in modo efficiente.

In conclusione, il prototipo sviluppato rappresenta una soluzione innovativa e affidabile per la gestione di dati complessi, garantendo prestazioni elevate e una gestione ottimale delle risorse.

4.2 Sviluppi Futuri

Il prototipo sviluppato rappresenta solo una prima fase di un progetto più ampio, che prevede lo sviluppo di nuove funzionalità e l'integrazione con altri sistemi.

Alcuni possibili sviluppi futuri includono:

- **Ottimizzazione delle Prestazioni**: Migliorare ulteriormente le prestazioni del sistema attraverso l'ottimizzazione del codice e l'introduzione di nuove tecniche di caching e buffering.
- **Implementazione di Nuove Funzionalità**: Aggiungere nuove funzionalità, come la gestione di transazioni, la replicazione dei dati e la gestione degli indici.
- **Realizzazione di Pagine più specializzate**: Sviluppare nuove tipologie di pagine specializzate per gestire dati specifici, come ad esempio pagine per la gestione di indici.

Bibliografia

[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

Tengo a ringraziare per il supporto e l'assistenza ricevuti durante la stesura di questa tesi da parte del mio relatore, il Prof. Marco Antonio Boschetti, del mio team, il Gabriele Monti, dei miei colleghi. Ringrazio inoltre Onit S.p.A. per avermi fornito la possibilità di lavorare a questo progetto.