Alma Mater Studiorum · Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea Triennale in Informatica per il Management

RASSEGNA DI STRUTTURE DATI E ALGORITMI PER L'INDICIZZAZIONE DI BASI DI DATI

Tesi di Laurea in Algoritmi e Strutture Dati

Relatore:
Chiar.mo Prof.
Moreno Marzolla

Presentata da:
Umberto Marini

Sessione II
Anno Accademico 2010/2011

Alla mia Famiglia

e a Lori.

Sommario

La frenetica evoluzione sociale e culturale, data dal crescente e continuo bisogno di conoscenza dell'uomo, ha portato oggi a navigare in un oceano sconfinato di dati e informazioni. Esse assumono una propria peculiare importanza, un valore sia dal punto di vista del singolo individuo, sia all'interno di un contesto sociale e di un settore di riferimento specifico e concreto. La conseguente mutazione dell'interazione e della comunicazione a livello economico della società, ha portato a parlare oggi di economia dell'informazione. In un contesto in cui l'informazione rappresenta la risorsa principale per l'attività di crescita e sviluppo economico, è fondamentale possedere la più adeguata strategia organizzativa per la gestione dei dati grezzi. Questo per permetterne un'efficiente memorizzazione, recupero e manipolazione in grado di aumentare il valore dell'organizzazione che ne fa uso. Un'informazione incompleta o non accurata può portare a valutazioni errate o non ottimali. Ecco quindi la necessità di gestire i dati secondo specifici criteri al fine di creare un proprio vantaggio competitivo.

La presente rassegna ha lo scopo di analizzare le tecniche di ottimizzazione di accesso alle basi di dati. La loro efficiente implementazione è di fondamentale importanza per il supporto e il corretto funzionamento delle applicazioni che ne fanno uso: devono garantire un comportamento performante in termini di velocità, precisione e accuratezza delle informazioni elaborate. L'attenzione si focalizzerà sulle strutture d'indicizzazione di tipo gerarchico: gli alberi di ricerca. Verranno descritti sia gli alberi su dati ad una dimensione, sia quelli utilizzati nel contesto di ricerche multi dimensionali (come, ad esempio, punti in uno spazio). L'ingente sforzo per implementare strutture

di questo tipo ha portato gli sviluppatori a sfruttare i principi di ereditarietà e astrazione della programmazione ad oggetti al fine di ideare un albero generalizzato che inglobasse in sé tutte le principali caratteristiche e funzioni di una struttura di indicizzazione gerarchica, così da aumentarne la riusabilità per i più particolari utilizzi. Da qui la presentazione della struttura GiST: Generalized Search Tree. Concluderà una valutazione dei metodi d'accesso esposti nella dissertazione con un riepilogo dei principali dati relativi ai costi computazionali, vantaggi e svantaggi.

Indice

1 Dato e informazione **1**

1.1 Caratteristiche dell'informazione 2

1.2 Management Information System 4

1.2.1 Il sistema informativo e la Business Intelligence 4

1.2.2 Introduzione alla gestione dei dati 6

2 Database Management System **7**

2.1 Architettura ANSI/SPARC per DBMS 8

2.2 Indipendenza dei dati e modelli concettuali 10

2.3 DBMS relazionali 11

2.4 Gestione dei dati 15

2.4.1 Metodi d'accesso 15

2.4.2 Definizione e classificazione di un indice 16

2.4.3 Strutture di indicizzazione 17

3 Metodi d'accesso gerarchici per dati unidimensionali **19**

3.1 Introduzione alle strutture ad albero 19

3.2 B-Tree 21

3.3 B^+-Tree 24

4 Metodi d'accesso gerarchici per dati multidimensionali **27**

4.1 Caratteristiche e requisiti delle basi di dati spaziali 28

4.2 Rappresentazione dei dati a più dimensioni 30

4.3 Tipo di query spaziali 31

4.4 Metodi d'accesso multidimensionali 34

4.4.1 Point Search Tree 34
k-d-B-Tree 36
4.4.2 Spatial Search Tree 38
R-Tree 39

5 Generalizzazione delle strutture gerarchiche 43
5.1 Generalized Search Tree 44
5.1.1 Key Methods 45
5.1.2 Tree Methods 46
Ricerca 47
Inserimento 49
Cancellazione 53
5.2 Implementazione di un R-Tree tramite GiST 53
5.2.1 R-Tree in spazi bidimensionali 54
5.2.2 PostgreSQL: rappresentazione grafica di un R-Tree utilizzando Gevel 57

Conclusioni 61

Bibliografia 65

Ringraziamenti 73

Elenco delle figure

1.1 Processo di creazione dell'informazione [59]. 2
1.2 Componenti di un generico sistema informativo [59]. 4
1.3 Aree di applicazione della BI [62]. 5

2.1 Architettura ANSI/SPARC a tre livelli. 8
2.2 Componenti principali di un DBMS relazionale [27]. 13
2.3 Esempio di funzione hash h(k) che mappa le chiavi k all'interno di N pagine primarie. 17

3.1 Nodo di un B-Tree di ordine d. Contiene $2d$ chiavi e $2d + 1$ puntatori ai sottoalberi. 21
3.2 Suddivisione tra indici e parte delle chiavi di un B^+-Tree [10]. 24
3.3 B-Tree e B^+-Tree a confronto [45]. 26

4.1 MBB rettangolare (MBR) e MBB approssimato al minimo poligono convesso. 30
4.2 Valutazione delle chiavi durante una query spaziale [19]. . . . 31
4.3 Rappresentazione grafica delle più comuni query in una base di dati multidimensionale [19]. 33
4.4 Spazio U^2 utilizzato per esemplificare graficamente le operazioni degli alberi descritti nelle sezioni successive [19]. 35
4.5 k-d-B-Tree [19]. 36
4.6 Organizzazione dello spazio U^2 da parte di un R-Tree [19]. . . 40
4.7 R-Tree [19]. 40

5.1 I punti sulla mappa rappresentano i villaggi presenti sul territorio della Grecia 57
5.2 Gevel: statistiche di composizione di un R-Tree. 58
5.3 Rappresentazione grafica dei livelli dell'albero R-Tree [41]. . . 59

Elenco delle tabelle

5.1 Metodi d'accesso gerarchici suddivisi per tipo di base di dati. . 61
5.2 Point Access Methods suddivisi per livello di memoria utilizzata [56]. 62
5.3 Spatial Access Methods [23]. 62
5.4 Tempo di esecuzione delle strutture dati viste nei Capitoli 3 e 4. 63

Capitolo 1

Dato e informazione

Il progresso dei mezzi di comunicazione ha alterato notevolmente la capacità di accesso alla conoscenza dal punto di vista del costo di recupero delle informazioni. Questo cambiamento è principale conseguenza della trasposizione dei contenuti da analogici a digitali, grazie alla conquista della società da parte di Internet. Un contesto economico caratterizzato da una sovrabbondanza di informazione ha causato una maggiore incertezza dal punto di vista organizzativo e decisionale. Da qui nasce la necessità di gestire i dati in maniera efficiente e selettiva, al fine di attivare un processo di creazione di conoscenza, utile a ricavare informazioni rilevanti per le operazioni di business di un'azienda.

In questo capitolo verranno brevemente esaminati i requisiti che un'informazione dovrebbe possedere per facilitarne il suo processo valutativo. Si introdurrà successivamente il tema della gestione dei processi informativi all'interno di un'organizzazione.

1.1 Caratteristiche dell'informazione

L'informazione è uno dei principali beni immateriali che aiuta l'azienda a generare risultati economici. Spesso questo concetto viene associato o utilizzato in maniera interscambiabile, ma errata, a quello di dato. Esso si riferisce alla più bassa astrazione del concetto di input elementare: quando parliamo di dato, infatti, facciamo riferimento ad un fatto non ancora sottoposto ad elaborazione. L'insieme di questi input, circoscritti in un dominio di appartenenza, acquistano un valore assumendo così la forma di informazione. Questo grazie ad un processo di applicazione di conoscenza, utilità, istruzioni e regole di rappresentazione specifiche. I dati, di per sé, non hanno un valore particolare: analizzati singolarmente o senza un modello di riferimento non esprimono nessun concetto di valore né quantitativo né qualitativo. Per esempio il numero 382910 è un dato di tipo numerico che non ci permette di inferire nessuna interpretazione sul significato; associato ad un nome di persona (questa volta un dato grezzo di tipo stringa), presente all'interno di un archivio di studenti dell'Università di Bologna, ci suggerisce che quel particolare individuo abbia come numero di matricola "382910" e che si sia immatricolato nell'anno accademico 2009/2010 piuttosto che l'anno successivo (questo incrociando una informazione aggiuntiva quale l'intervallo di numeri di matricola registrati quell'anno specifico). Anche l'integrazione dei dati da sorgenti multiple, quindi, è un'attività fondamentale per tutte le organizzazioni. Tutto ciò suggerisce che l'informazione sia legata al tipo di associazione definita tra i dati stessi; inoltre assume un'utilità specifica data dal processo di trasformazione logica e dall'applicazione di una particolare conoscenza (Figura 1.1).

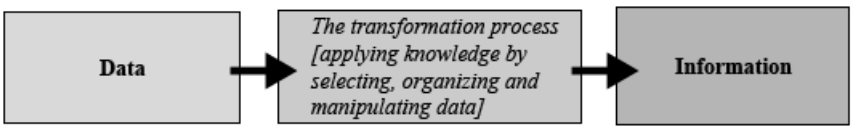

Figura 1.1: Processo di creazione dell'informazione [59].

Come suggeriscono Ralph M. Stair e George Reynolds [59], per valutare la rilevanza e la validità di un'informazione, questa dovrebbe possedere alcune caratteristiche rilevanti per l'organizzazione che ne fa uso.

- **Accessibile**: si parla di facilità d'accesso sia in termini di tempo che di correttezza nel formato di rappresentazione.
- **Accurata**: la presenza di dati imprecisi deve essere esclusa dal processo di trasformazione in informazione finale.
- **Completa**: un'informazione deve contenere in maniera integrale tutti i fatti rilevanti per la sua interpretazione.
- **Economica**: il costo di produzione dell'informazione deve sempre bilanciare il suo stesso valore.
- **Flessibile**: l'informazione deve adattarsi al mutare di situazioni temporali e aree di applicazione.
- **Rilevante**: si parla di pertinenza rispetto ad un contesto di riferimento.
- **Affidabile**: l'affidabilità dell'informazione è un concetto che può portare a decisioni aziendali improduttive o irrazionali causate da una scarsa attendibilità delle sorgenti dei dati.
- **Sicura**: sicurezza in termini di accesso garantito ad utenti autorizzati e fidati.
- **Semplice**: come citato nell'introduzione del capitolo, una sovrabbondanza di informazione o un eccessivo grado di dettaglio possono portare ad una difficile comprensione dei suo concetti rilevanti.
- **Tempestiva**: l'informazione deve essere reperibile e utilizzabile nell'effettivo momento di necessità.
- **Verificabile**: la non falsificabilità dell'informazione implica che qualsiasi soggetto possa verificare la sua correttezza semantica.

La criticità e l'importanza di questi fattori di valutazione dipendono necessariamente dal tipo di dati da cui derivano e dall'obiettivo per cui l'informazione è finalizzata.

1.2 Management Information System

Come vengono effettivamente gestite queste informazioni nell'ambito di un sistema in cui interagiscono risorse umane ed entrano in gioco strategie organizzative volte al raggiungimento di particolari obiettivi?

In questa sezione verrà esposto brevemente il concetto di sistema informativo, saranno enunciati i principali servizi per gestire in maniera strutturata e flessibile le informazioni all'interno di un processo aziendale. Successivamente verrà introdotto il tema delle basi di dati, argomento principale del capitolo due.

1.2.1 Il sistema informativo e la Business Intelligence

Un sistema informativo è un insieme organizzato di elementi o componenti interdipendenti tra loro che hanno come scopo la raccolta, la memorizzazione, la manipolazione e l'elaborazione di dati e informazioni. Queste procedure vengono accompagnate da un meccanismo di feedback continuo (Figura 1.2) per garantire l'allineamento tra il processo di produzione aziendale e gli obiettivi da raggiungere [59].

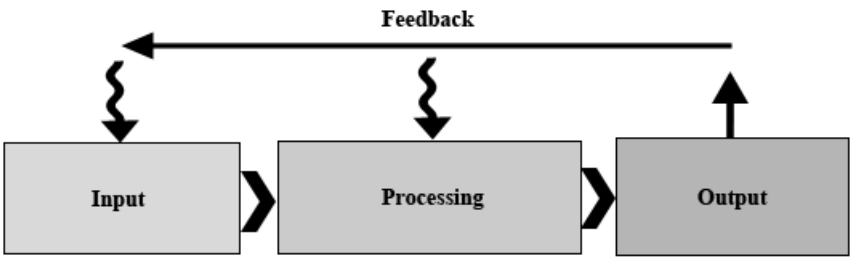

Figura 1.2: Componenti di un generico sistema informativo [59].

L'ingresso in gioco dell'Information Technology ha inevitabilmente portato al concetto di sistema informativo come infrastruttura tecnologica, organica e sistematica che ha lo scopo di inglobare in sé componenti hardware e software, metodologie e strumenti per la definizione delle strategie aziendali e l'apporto razionale e cognitivo delle risorse umane. Questo modello definisce l'architettura di un Management Information System (MIS), ossia l'insieme delle applicazioni volte alla risoluzione di problemi decisionali complessi. L'approccio organizzativo dei sistemi informativi si è lentamente combinato e amalgamato con l'idea di Business Intelligence, una delle principali chiavi di successo per incrementare il valore e la performance di un'organizzazione [62]. Il termine è stato propriamente coniato da Howard Dressner nei primi anni Novanta per identificare una catena di attività che interessano l'interpretazione ed elaborazione delle informazioni.

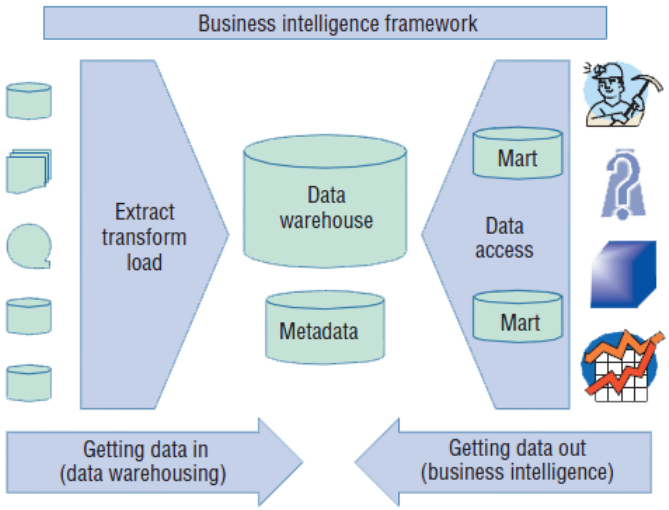

Figura 1.3: Aree di applicazione della BI [62].

Il processo che interessa la Business Intelligence (Figura 1.3) è suddiviso in due grandi aree applicative: la prima riguarda l'estrazione di dati da sorgenti multiple e la loro trasformazione in informazioni e metadati utili al supporto del processo decisionale. La seconda riguarda tutte quelle me-

todologie e applicazioni di accesso ai dati, precedentemente elaborati, per costruire un'analisi dettagliata e predittiva dello sviluppo aziendale.

1.2.2 Introduzione alla gestione dei dati

Una gestione errata o inefficiente dei dati, all'interno del sistema della Business Intelligence, può gravare pesantemente sui budget investiti dai manager nei loro progetti aziendali: più del 50 per cento dei costi inattesi di un processo produttivo sono causati da queste mancanze di precisione [62]. Inoltre, in un contesto in cui i servizi vengono spinti assiduamente verso le architetture distribuite del Web, cresce la necessità di azzerare il tempo di latenza per l'accesso alle informazioni. All'interno di un sistema informatico aziendale, la responsabilità di soddisfare questi requisiti di affidabilità, efficienza, efficacia e privatezza è attribuita al Database Management System (DBMS). Una base di dati è una collezione persistente e condivisa di record, gestita da un DBMS.

Nel capitolo successivo verrà illustrata l'architettura di un DBMS e introdotto il tema e le metodologie di accesso efficiente ai dati.

Capitolo 2

Database Management System

La gestione informativa di un'organizzazione rispecchia la progettazione della base di dati studiata dagli amministratori del sistema informatico. La realizzazione di questo lavoro viene resa possibile grazie al supporto e ai servizi offerti da un Database Management System. Le funzionalità di un DBMS sono di fondamentale importanza in qualsiasi contesto sia necessario memorizzare, manipolare e analizzare dati per la produzione di informazione: dal ricercatore universitario che deve registrare quotidianamente i progressi dei suoi studi, alla multinazionale quotata in borsa che deve tenere traccia di centinaia di transazioni al giorno.

In questo capitolo verrà data una visione d'insieme dei modelli strutturali di organizzazione dei dati e dei tre livelli di astrazione che identificano il loro processo descrittivo. Successivamente si illustrerà l'architettura di uno dei Database Management System più diffuso a livello aziendale e maggiormente studiato a livello accademico: il DBMS relazionale. Vedremo, infine, quali sono i criteri e le metodologie che utilizzano questi sistemi per l'accesso alle strutture di memorizzazione dei dati.

2.1 Architettura ANSI/SPARC per DBMS

La maggior parte dei DBMS commerciali adottano uno standard di progettazione astratta proposto nel 1975 dal gruppo di lavoro SPARC/DBMS [1]. L'idea architetturale nasce dallo studio dell'informazione come entità che può essere contestualizzata su tre diversi domini: (1) il mondo reale, (2) il complesso delle concezioni percepite e sviluppate dall'uomo sugli oggetti della realtà e (3) la rappresentazione di questi fatti su supporti di memorizzazione (da quelli cartacei a quelli digitali). In ciascuno di questi domini l'informazione assume delle peculiarità, che vengono delineate da appropriati schemi di descrittori dei dati. Tenendo conto del contesto appena illustrato, l'obiettivo dell'architettura ANSI/SPARC, è quello di discostare l'interpretazione dei concetti dell'utente dal livello di memorizzazione fisico e permanente dei dati.

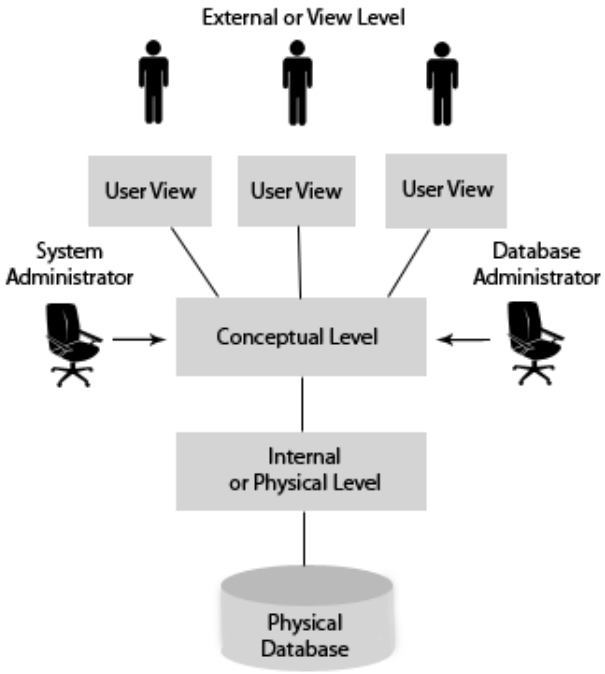

Figura 2.1: Architettura ANSI/SPARC a tre livelli.

La Figura 2.1 illustra i sopracitati livelli di progettazione, a loro volta descritti da tre diversi schemi di dati: esterno, logico o concettuale e interno (rispettivamente da quello più alto e più vicino all'utente, a quello più basso che individua le strutture di memorizzazione).

Schema esterno

Lo schema esterno identifica le possibili viste richieste dagli utenti e gestisce un meccanismo di sicurezza robusto, ma flessibile, per la visualizzazione dei soli dati per cui il gruppo di utenti detiene i permessi. Consente inoltre il controllo degli accessi concorrenti.

Schema concettuale

Lo schema concettuale rappresenta la struttura logica dell'intera base di dati progettata dall'amministratore del sistema ed è indipendente dall'organizzazione e dalle considerazioni relative alle strutture di memorizzazione sottostanti. In questo schema vengono definite le entità di interesse del mondo reale, le relazioni tra i dati, i loro attributi e i vincoli di integrità, sicurezza e correttezza semantica. Ciascuna vista dello schema esterno deriva dall'elaborazione degli oggetti al livello concettuale.

Schema interno

Lo schema interno interpreta lo schema concettuale attraverso la descrizione di record a livello di supporti di memorizzazione. L'implementazione fisica della base di dati deve garantire la maggiore efficienza sia dal punto di vista del tempo d'accesso alle strutture, sia da quello dell'allocazione di spazio in memoria.

2.2 Indipendenza dei dati e modelli concettuali

Una delle principali e più importanti conseguenze dell'architettura sopra descritta è quella dell'indipendenza dei dati, ossia la possibilità di modificare lo schema in uno dei livelli senza interessare le altri parti del sistema. Esistono due forme di indipendenza:

- **Logica**: implica la totale trasparenza delle modifiche allo schema concettuale, che lasciano inalterate le viste dell'utente. Al tempo stesso, cambiamenti e aggiornamenti delle applicazioni a livello esterno non vengono riflesse sulla logica della base di dati.
- **Fisica**: permette che l'organizzazione dei dati a livello fisico possa cambiare nel tempo per migliorare le prestazioni del sistema, senza il necessario apporto di modifiche allo schema logico o a quello esterno.

Le caratteristiche dei dati hanno permesso, quindi, di costruire degli schemi per la descrivere le relazioni che intercorrono tra essi, per rappresentare la loro semantica e definirne i loro vincoli di tipo. Questi sono: i modelli ad oggetti (Object-based Data Models), i modelli basati su record (Record-based Data Models) e i modelli fisici (Physical Data Models). I primi vengono utilizzati nelle fasi preliminari della progettazione della base di dati, consentendo di rappresentare i concetti derivanti dal mondo reale in maniera interconnessa tra loro. Tra questi vi sono:

- *Il modello entità-relazione*: proposto per la prima volta nel 1976 da Peter Pin-Shan Chen [8], permette la traduzione di un dominio d'interesse reale in rappresentazioni concettuali ad alto livello. Questo grazie a dei costrutti che descrivono classi di oggetti (entità), i loro legami (relazioni o associazioni) e il loro livello di dettaglio dei dati (attributi) [58].
- *I modelli orientati agli oggetti*: introdotti in letteratura nel 1990, dal Prof. Kim Won del MIT [34], si basano sui principi della programmazione ad oggetti (Object Oriented Programming, OOP). In questo

caso le applicazioni a livello utente e la base di dati, condividono lo stesso modello di rappresentazione: le informazioni e le relazioni tra esse, vengono memorizzate in una struttura logica ad oggetti in grado di lavorare con dati molto complessi.

- *I modelli basati sulla semantica dei dati*: a differenza del precedente modello, questi schemi concettuali sono orientati ai fatti, che vengono espressi in forma di relazione binaria tra coppie di dati. L'interpretazione del significato semantico del binomio appena citato, non è altro che la costruzione di proposizioni composte da un soggetto, un predicato ed un oggetto. Questo modello viene utilizzato soprattutto nel campo delle basi di dati distribuite, grazie alla sua alta integrabilità nel caso di più collezioni che condividono gli stessi tipi di relazione tra i contenuti. A questo proposito è stato effettuato uno studio da Michael Hammer e Dennis McLeod, i quali sono partiti dalla limitata capacità di rappresentazione del mondo reale da parte dei modelli relazionali e orientati agli oggetti, per introdurre la possibilità di completare queste mancanze tramite l'arricchimento semantico dei concetti [24].

La scelta di uno dei modelli concettuali suddetti, influenza notevolmente la successiva rappresentazione dei dati e dei loro attributi sotto forma di record. In questo senso, esistono diverse strutture logiche tra cui quella relazionale, gerarchica, reticolare e ad oggetti.

2.3 DBMS relazionali

Il modello logico maggiormente utilizzato e diffuso per la progettazione di un DBMS commerciale è quello relazionale (Relation Data Base Management System, RDBMS). Nonostante fosse stato presentato per la prima volta nel 1970 da Edgar Frank Codd [9], viene implementato all'interno di sistemi di gestione di basi di dati solamente undici anni dopo, a causa di notevoli problemi derivanti dall'efficienza e dall'affidabilità delle implementazioni. Il modello relazionale è la naturale trasformazione degli oggetti espressi dal

modello entità-relazione, in una collezione di record interconnessi tra loro tramite associazioni che sono identificate da tabelle in grado di diminuire le ridondanze e le inconsistenze dei dati. Il concetto fondamentale su cui si basano i DBMS relazionali è quello di relazione in senso matematico del termine, ovvero un insieme di tuple definite tramite il prodotto cartesiano tra più domini di interesse. I riferimenti tra dati di differenti relazioni sono ottenuti tramite valori uguali in ennuple (insieme di attributi di un record) diverse. L'identificazione univoca di questi attributi costituisce le chiavi d'accesso ai record e definisce le dipendenze funzionali tra le varie tabelle. Questa proprietà permette di costruire strutture di rappresentazione dell'informazione complesse. Consente inoltre di specificare dei vincoli d'integrità utili ad esprimere un'accurata descrizione dei record in termini di qualità dei dati e di correttezza rispetto al dominio di riferimento. I vincoli sono predicati booleani utilizzati nell'esecuzione delle interrogazioni da parte del DBMS e definiti in fase di progettazione della base di dati. Si suddividono in:

- Vincoli intra-relazionali: sono stabiliti sulle singole ennuple e sul dominio di uno o più attributi.
- Vincoli inter-relazionali: garantiscono l'integrità referenziale tra attributi di più tabelle, ovvero la coerenza nelle dipendenze funzionali tra diverse relazioni. Sono definiti tramite l'utilizzo di chiavi univoche.

Un DBMS relazionale è quindi un sistema progettato per essere gestito e controllato tramite specifici linguaggi di interrogazione, che prevedano sia operazioni di definizione (Data Definition Language, DDL), sia operazioni di manipolazione dei dati (Data Manipulation Language, DML). Ciò è reso possibile dall'architettura del sistema stesso, il quale, come si può vedere dalla Figura 2.2, si suddivide in cinque componenti principali [27].

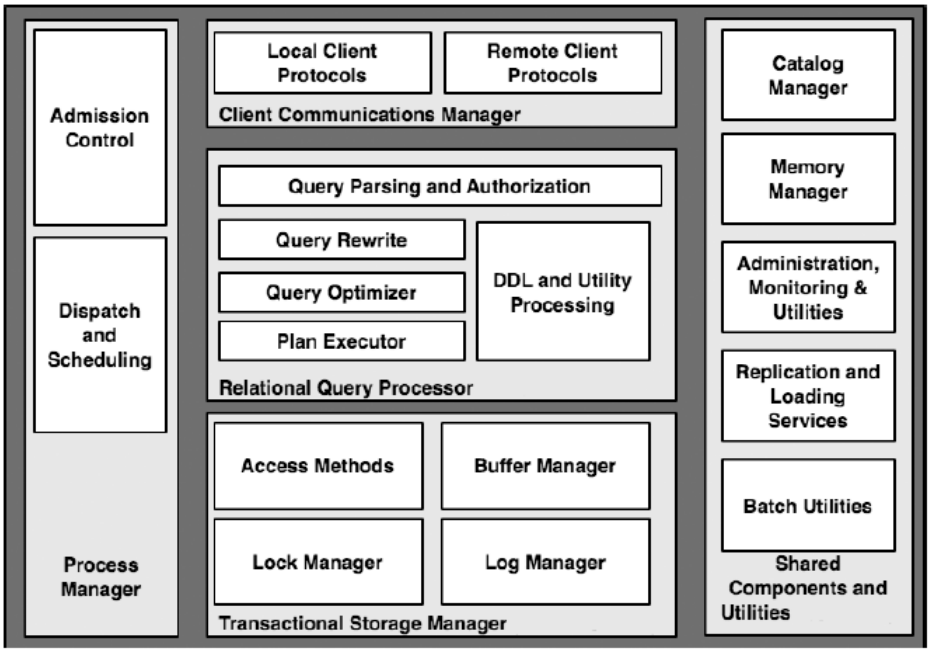

Figura 2.2: Componenti principali di un DBMS relazionale [27].

Di seguito si trova una breve illustrazione dei sottoinsiemi di un RDBMS. Nella prossima sezione del capitolo, invece, l'attenzione sarà focalizzata sul modulo dei metodi d'accesso facente parte del Transactional Storage Manager, che rappresenta l'input per il tema principale di questa rassegna: le strutture gerarchiche d'indicizzazione dei dati.

Client Communication Manager

Le richieste di comunicazione alla base di dati che provengono da API esterne, vengono gestite dal Client Communication Manager (CCM). Esso ha il compito di stabilire e ricordare lo stato della connessione; alla fine del processo di esecuzione della query, il CCM risponde al comando del chiamante con un messaggio appropriato contenente i dati richiesti o un stringa di controllo in caso di errore.

Process Manager

Dopo aver stabilito la connessione, il DBMS assegna un thread al comando che viene elaborato successivamente dal Process Manager. E' compito del modulo di Admission Control decidere, in base alla disponibilità delle risorse, quando la query potrà essere trasferita allo schedulatore per iniziare la sua elaborazione.

Relational Query Processor

Una volta allocate le risorse e ammessa l'esecuzione del thread, la richiesta iniziale del client passa al Relational Query Processor. L'insieme di questi moduli è responsabile della compilazione ed esecuzione dell'interrogazione in un query plan, ovvero una sequenza di operatori insiemistici e algoritmi relazionali definiti dal DDL.

Transactional Storage Manager

L'insieme delle operazioni stabilite dal Plan Executor per la computazione della query, richiedono i dati necessari per tale scopo al Transactional Storage Manager, il quale gestisce ogni accesso e modifica ai record. Questi moduli sono regolati da un gestore del buffer della memoria e includono algoritmi e strutture dati (quali tabelle e indici) per organizzare il contenuto su disco. Inoltre sono previsti dei moduli come il Lock e Log Manager per garantire la corretta esecuzione di più interrogazioni concorrenti tra loro. Una volta eseguito l'intero query plan, la risposta alla richiesta del client risale la catena dei componenti; il controllo tornerà al Client Communication Manager che si occuperà di spedire i risultati al chiamante.

Shared Components and Utilities

Esistono, infine, dei moduli condivisi che vengono invocati durante ogni transazione eseguita dal DBMS e che lavorano per l'affidabilità e l'ottimizzazione dell'intero sistema e dei suoi processi.

2.4 Gestione dei dati

In questa sezione, si forniranno i concetti e le informazioni preliminari necessarie a comprendere lo studio di rassegna delle strutture dati d'indicizzazione, che verranno ampiamente esposte nel capitolo successivo.

2.4.1 Metodi d'accesso

Come anticipato in precedenza parlando del modulo di gestione dei dati, un DBMS utilizza un insieme di strutture e algoritmi per recuperare le informazioni su disco, chiamati metodi d'accesso. Insieme alla gestione del buffer, che ha il compito di portare in memoria centrale i blocchi richiesti dalle operazione logiche, tali algoritmi di recupero rappresentano uno dei fattori critici per le prestazioni dell'intero sistema. L'organizzazione dei file (ovvero le collezioni di pagine contenti blocchi di record) all'interno di una base di dati relazionale, prevede nella maggior parte dei casi che ogni associazione (o tabella) sia memorizzata in un file appropriato. Esistono due principali approcci di base per l'organizzazione dei record:

- *File di tipo Heap*: non vi è un specifico ordine di memorizzazione dei dati. Questo file è particolarmente efficiente nel caso dell'inserimento di nuovi record, ma se non viene ottimizzato mediante l'utilizzo di una struttura ausiliaria, il recupero dei dati è particolarmente lento e costoso; è comunque possibile supportare la ricerca tenendo traccia delle pagine presenti nel file oppure memorizzando lo spazio libero nelle pagine o il numero di record contenuti in ognuno di esse;

- *File ordinati*: i record sono fisicamente organizzati in ordine sequenziale. Sfruttando questa proprietà è possibile utilizzare la ricerca binaria per un accesso più rapido ai dati; è comunque oneroso mantenere l'ordinamento in caso di operazioni di inserimento e cancellazione di record.

2.4.2 Definizione e classificazione di un indice

Per migliorarne l'efficienza, i metodi citati poc'anzi vengono affiancati da strutture addizionali che hanno lo scopo di aiutare la ricerca diretta dei dati al loro interno: è il caso degli indici. Attraverso la definizione di coppie, formate da una chiave di ricerca e da un puntatore ai dati (data entry) per ogni record, si possono ridurre gli accessi a disco ottimizzando il tempo di recupero di tutte le data entry con valore di chiave uguale a k. Se una tabella non possedesse un indice associato ad uno o più dei suoi attributi, il sistema sarebbe obbligato a leggere tutti i dati presenti in essa sino a quando non verrebbe trovato il record d'interesse. Un indice permette di ridurre l'insieme dei dati analizzati, velocizzando le operazioni di ricerca. La dimensione di questi file di indicizzazione è tipicamente minore rispetto a quelli dei dati su cui lavorano; possono essere sia integrati ai file che indicizzano, sia fisicamente separati e indicizzati a loro volta (indici a più livelli).
Esistono diverse classificazioni di indice: la prima definisce il concetto di indice primario e secondario in base al valore nei record, univoco o meno, su cui esso viene costruito. La seconda ammette l'esistenza di strutture clustered e unclestered: le prime riorganizzano il file dei dati secondo un criterio di ordinamento relativo ad un particolare valore di chiave; mentre le seconde, al contrario, permettono che due record con valore uguale risiedano in due blocchi fisicamente lontani (determinando così accessi meno efficienti). Un'ulteriore suddivisione riguarda la corrispondenza tra le etichette nell'indice e i valori del campo di ricerca nel file di dati: se le etichette coprono la totalità del valori dei record, allora si dice che un indice è denso, in caso contrario si chiama sparso. Infine il numero di campi che formano la chiave di ricerca, definisce la classificazione tra indici semplici (un solo campo) e composti. Questa differenza è importante in quanto se un indice è composto, supporta un più alto numero di query e permette l'estrazione di una quantità superiore di informazioni. Tuttavia, questa condizione richiede una maggiore attenzione nell'aggiornamento dell'indice dopo ogni modifica dei campi di un record. Vediamo ora quali sono le strutture generiche utilizzate per creare indici sulla base del tipo di query da soddisfare.

2.4.3 Strutture di indicizzazione

Un'interrogazione basata su un criterio di uguaglianza, in cui il valore di ogni campo della chiave è fissato, prende il nome di equality query. Al contrario, si parla di range query quando vengono richiesti dei record all'interno di un intervallo noto di valori. La necessità di soddisfare vincoli di predicato diversi, ha portato a sviluppare differenti strutture dati efficienti per l'uno piuttosto che per l'altro scopo.

Indici basati su hashing

Le data entry vengono organizzati in gruppi chiamati bucket, ognuno dei quali contiene una pagina primaria e, collegate ad essa, ulteriori possibili pagine di overflow. I puntatori ai dati sono assegnati ai bucket secondo una funzione di hash applicata alla chiave di ricerca, ma indipendente dalla loro numerosità. Se la distribuzione dei record nei bucket e' uniforme, si può affermare che è stata applicata una buona funzione di hash: infatti ciascun bucket mappa una data entry permettendo che il suo tempo medio di recupero sia di ordine costante. In Figura 2.3 è mostrato un esempio di indice basato su una funzione hash, calcolata sul modulo del numero di bucket della struttura. L'hashing è la tecnica più efficiente per accedere ai dati in base a criteri di uguaglianza ma risulta inadatta nel caso di interrogazioni su intervalli di valori. Non essendo il tema principale di questo studio, rimando ai testi di R. Fagin per una trattazione maggiormente dettagliata dei metodi di hashing lineare [35] e hashing estensibile [11].

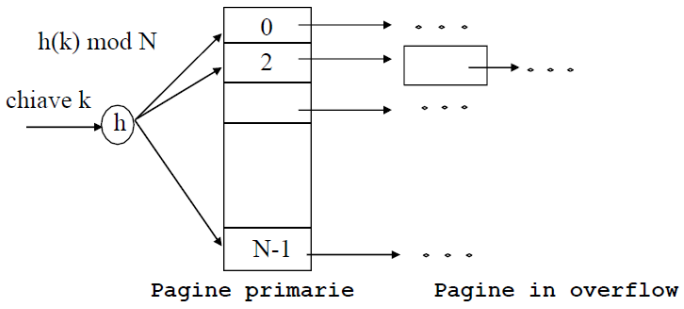

Figura 2.3: Esempio di funzione hash h(k) che mappa le chiavi k all'interno di N pagine primarie.

Indici basati su strutture ad albero

In questo caso, le etichette sono organizzate in base al valore del campo di ricerca e secondo una struttura dati gerarchica ad albero. Ogni etichetta coincide con una pagina diversa e viene mappata nell'ultimo livello delle foglie. La ricerca parte dalla radice e attraversa la struttura grazie ai puntatori tra nodi che collegano tra loro le pagine. Gli alberi permettono un efficiente accesso ai dati sia secondo predicati di uguaglianza, sia secondo la verifica di appartenenza a determinati intervalli di valori. Queste strutture saranno oggetto di discussione da qui al termine della rassegna e verranno disquisite in base alle caratteristiche dei dati che indicizzano.

Capitolo 3

Metodi d'accesso gerarchici per dati unidimensionali

3.1 Introduzione alle strutture ad albero

La scelta di un metodo d'indicizzazione dovrebbe essere valutata tenendo conto di diversi fattori, quali: (1) il costo della riorganizzazione periodica della struttura, (2) la frequenza delle operazioni di inserimento, cancellazione e recupero dei record, (3) il tempo d'accesso ai dati nel caso medio e nel caso peggiore, (4) il tipo di query più frequentemente processata dal sistema. La maggior parte dei DBMS implementano l'indicizzazione sia tramite strutture dati gerarchiche, sia attraverso funzioni di hash. In questo capitolo verranno discussi gli alberi di ricerca in quanto strumenti altamente scalabili e dinamici, in grado di offrire un efficiente supporto alle basi di dati su cui lavorano le numerose applicazioni di dominio geografico, multimediale, biochimico, sanitario, biometrico e così via. Queste strutture possiedono alcune proprietà interessanti: preservano l'ordine dei dati per supportare le interrogazioni basate su intervalli di valori, sono quasi del tutto indipendenti rispetto alla distribuzione degli input e, tranne in alcuni casi, garantiscono l'efficienza nell'occupazione di memoria.

Al fine di soddisfare ogni tipo di necessità a livello di gestione dei dati, sono stati sviluppati una grande varietà di metodi d'accesso. Pertanto, in questo e nei successivi capitoli, verrà suddivisa la trattazione delle strutture dati gerarchiche in base al loro dominio di interesse. Si inizierà con l'analisi di quelle specializzate nel campo delle ricerche su dati monodimensionali, applicabili quindi alle basi di dati relazionali. Si continuerà esponendo le strutture progettate per il recupero di informazioni multidimensionali, che utilizzano il concetto di spazio per integrare e analizzare grossi volumi di dati da diversi punti di vista. Infine, la rassegna verrà conclusa con lo studio delle caratteristiche dell'albero GiST (Generalized Search Tree), il quale definisce un'astrazione alle comuni operazioni di un metodo d'accesso gerarchico. Le transazioni di base che un gestore della memoria può eseguire su indici che memorizzano i record mediante una sequenza di chiavi ordinate sono: l'inserimento o rimozione di un nuovo record con chiave univoca, il suo recupero e la lettura sequenziale dell'intero file. I B-Tree [2] supportano tutte le operazioni appena elencate e sono di fatto diventati il metodo standard di indicizzazione all'interno di un DBMS [10]. Oltre a questi, sono state progettate numerose strutture ad albero, sia per risolvere problemi legati al tipo di dato da gestire (per esempio, Prefix B^+-Tree [3] e String B-Tree [12] nel caso delle stringhe), sia per ottimizzare l'utilizzo di memoria e minimizzare il numero di accessi a disco (B^*-Tree e B^+-Tree [10]). Vedremo di seguito la struttura e le proprietà di base di un generico B-Tree e di una delle sue principali varianti: il B^+-Tree.

3.2 B-Tree

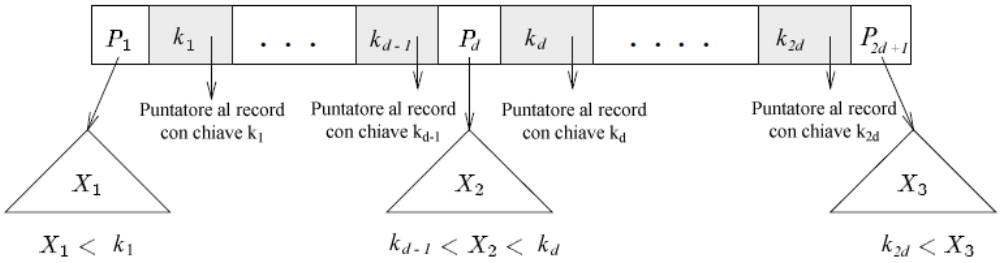

Figura 3.1: Nodo di un B-Tree di ordine d. Contiene $2d$ chiavi e $2d + 1$ puntatori ai sottoalberi.

I B-Tree [2] sono strutture ad albero ordinate e dinamiche. L'ordinamento è mantenuto grazie alla proprietà per la quale il valore di ogni chiave, appartenente al sottoalbero sinistro di un nodo, è strettamente inferiore rispetto ad quelli delle chiavi appartenenti ai sottoalberi alla sua destra. La dinamicità della struttura permette al numero di livelli dell'albero di variare in base alla dimensione del file dei record da indicizzare; le foglie si trovano tutte allo stello livello. I B-Tree sono caratterizzati da un parametro d che definisce l'ordine dell'albero e il grado di saturazione (fanout) dei suoi nodi n (esclusa la radice). Questo implica che il numero di chiavi e puntatori contenuti in ciascuno di essi, debba essere compreso rispettivamente tra $[d, 2d]$ (chiavi) e $[d+1, 2d+1]$ (puntatori); ogni nodo, quindi, risulta sempre pieno per almeno la metà del blocco in cui è memorizzato fisicamente. L'ultima proprietà dei B-Tree è quella per cui, ogni operazione di inserimento o cancellazione mantiene costantemente bilanciata la struttura dell'albero, permettendo così che il cammino più lungo dalla radice alle foglie sia pari a $log_d n$ nodi (si rimanda a [10] per la dimostrazione). Per una lettura più approfondita delle tecniche di bilanciamento si suggeriscono gli articoli di Foster [13] e Karlton [33]. In Figura 3.1 si trova la rappresentazione del nodo generico di un B-Tree.

Di seguito sono descritti gli algoritmi delle operazioni di base relativi alla ricerca sequenziale, all'inserimento, cancellazione e recupero di una chiave.

Inoltre, verrà data una stima dei costi computazionali in termini di accessi alla memoria secondaria, ossia il numero di nodi visitati dell'albero, e in base al tempo di esecuzione dell'algoritmo da parte della CPU.

- *Ricerca di una chiave k*: partendo dalla radice dell'albero (1) viene portato in memoria il nodo N. Se k è presente in N allora la ricerca è conclusa. In caso contrario si procede iterativamente da (1) analizzando il nodo identificato dal puntatore in N associato ad una chiave k_1, che rispetta il predicato di ordinamento di k: se $k < k_1$ si procede nel sottoalbero di sinistra, mentre se $k > k_1$ il sottoalbero selezionato è quello di destra. Questo processo viene eseguito sin quando viene trovata la chiave corrispondente a k oppure un puntatore nullo. Il numero di nodi visitati durante la ricerca di k è $O(log_d n)$. Grazie all'ordine delle chiavi, per ogni nodo visitato è possibile effettuare una ricerca binaria, con costo pari a $O(logd)$; ne va da sé che il costo complessivo della ricerca di una chiave k è $O(logd \times log_d n)$, che, per il cambiamento di base dei logaritmi, equivale a $O(logn)$.

- *Inserimento di una chiave k*: viene eseguito in due fasi. La prima, richiede la ricerca di una foglia F, candidata all'inserimento di k. Una volta trovata F, la seconda parte dell'operazione riguarda l'effettivo inserimento di k e il ribilanciamento dell'albero. Qualora F avesse abbastanza spazio, sarebbe sufficiente inserire la nuova chiave. Invece, nel caso in cui F fosse piena, bisognerebbe eseguire uno split della foglia, in maniera tale che le d chiavi più piccole e le d chiavi più grandi venissero memorizzate rispettivamente in due nodi separati. In questo modo, il valore mediano, verrebbe spinto verso il livello superiore, fungendo così da separatore delle due foglie. Nel peggiore dei casi la propagazione dello split arriverebbe alla radice, incrementando di un livello l'altezza dell'albero. Il costo, in termini di accessi a disco, è valutato in base al numero di nodi visitati durante la ricerca ed è pari a $O(h)$, dove $h = log_d n$ rappresenta l'altezza dell'albero. Il tempo di esecuzione dell'algoritmo, invece, è legato alle singole operazioni effettuate dalla CPU. Sapendo che l'operazione di split ha costo costante

di $O(1)$ e l'altezza dell'albero corrisponde a h, si conclude che il tempo di CPU è pari a $O(d \times h)$.

- *Rimozione di una chiave k*: anche questa operazione richiede una ricerca preliminare per localizzare il nodo N, candidato alla rimozione di k. N può essere (1) una foglia F oppure (2) un nodo interno. Nel caso (1), dopo aver eliminato k, è necessario controllare che lo slot vuoto, non abbia portato il numero di chiavi di F sotto la soglia d. Se così fosse (condizione di underflow), sarebbe necessario distinguere le due seguenti possibilità: se i fratelli adiacenti ad F avessero almeno $d + 1$ chiavi (serve solo una chiave da rimpiazzare in F), si effettuerebbe un'operazione di redistribuzione delle chiavi; in caso contrario si dovrebbe applicare un'operazione di fusione/concatenazione dei nodi. Nel caso (2) invece, il valore di k verrebbe sovrascritto da k_a, ossia la sua chiave adiacente (o successore). k_a risiede nella foglia F_s più a sinistra del sottoalbero destro di N ed è la chiave con valore minore più grande di k. Dopo aver effettuato questa operazione ed aver lasciato uno slot libero in F_s, il procedimento si ricongiunge al caso (1). Nel peggiore dei casi, le concatenazioni dei nodi adiacenti possono interessare anche i livelli superiori arrivando alla radice, decrementando di un livello l'altezza dell'albero. Le considerazioni fatte per l'inserimento di una chiave valgono anche per la sua cancellazione: le operazioni di I/O su disco hanno costo $O(h)$; il tempo di esecuzione dell'algoritmo è, invece, $O(d \times h)$, in quanto anche le concatenazioni e redistribuzione delle chiavi, come visto per gli split dei nodi, costano $O(1)$.

- *Ricerca sequenziale*: i B-Tree sono una struttura inefficiente nel caso di ricerche in cui vi è la necessità di estrarre tutte le chiavi dell'indice in maniera ordinata. Infatti, richiedono uno spazio minimo di $log_d(n + 1)$ nodi in memoria principale per memorizzare il percorso effettuato ed evitare di leggere gli stessi blocchi più volte. Una possibile implementazione degli algoritmi relativi alla ricerca, inserimento e cancellazione di una chiave si possono trovare in [2].

3.3 B^+-Tree

Per correggere il difetto relativo all'eccessivo utilizzo di memoria della ricerca sequenziale relativa ai B-Tree, Bayer e McCreight [2] hanno proposto un'efficiente variante: i B^+-Tree. Sono una struttura in cui esiste una netta separazione logica tra indice e insieme delle chiavi (Figura 3.2). Infatti, quest'ultime sono memorizzate interamente nelle foglie dell'albero e i livelli superiori servono a mapparle per una loro rapida localizzazione. In questo modo è possibile creare una sequenza collegata (e ordinata) tramite gli ultimi puntatori di ciascuna foglia, in maniera tale da permettere una più facile ricerca sequenziale.

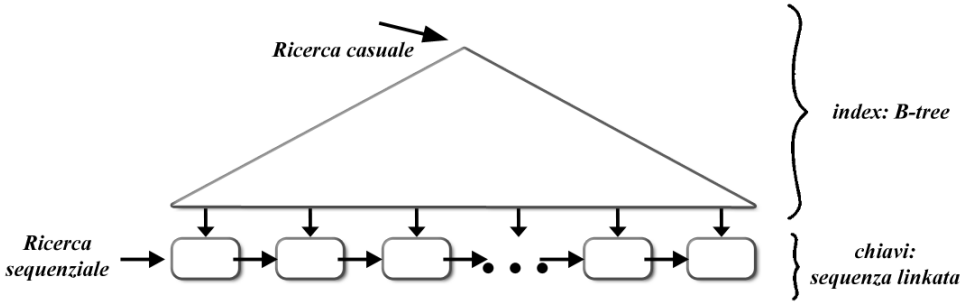

Figura 3.2: Suddivisione tra indici e parte delle chiavi di un B^+-Tree [10].

Vediamo in dettaglio quali cambiamenti subiscono le operazioni di base viste per i B-Tree:

- *Ricerca*: la ricerca procede sempre dalla radice alla corretta foglia, passando attraverso la mappatura indicizzata e servendosi del principio di ordinamento dei sottoalberi visto per i B-Tree. Infatti, nonostante la chiave del valore di ricerca della query possa essere presente nei nodi indice, l'algoritmo non si ferma e procede in ogni caso sino all'ultimo livello dell'albero.

- *Eliminazione di una chiave k*: il procedimento di cancellazione viene semplificato grazie alla proprietà secondo cui il valore k da eliminare,

si trova sempre nell'ultimo livello dell'albero. Sino a quando ciascuna foglia rimane piena almeno per metà, non vi è il bisogno di aggiornare i nodi interni che fungono da indici. In caso di underflow, invece, si procede con le redistribuzioni o con le concatenazioni appropriate, impostando i corretti valori ai livelli superiori.

- *Inserimento di una chiave k*: stesso metodo visto per i B-Tree.
- *Ricerca sequenziale*: richiede tempo logaritmico per raggiungere la chiave di valore minimo memorizzata nella foglia più a sinistra dell'albero. Inoltre, l'accesso ai suoi successori nello stesso blocco o a quelli nella foglia adiacente, avviene in tempo costante $O(1)$.

Il costo computazionale di tutte le operazioni messe a disposizione da un B^+-Tree è $O(log_d n)$, ovvero il singolo attraversamento dell'albero dalla radice sino alle foglie.

In Figura 3.3 si trova un esempio grafico delle diverse strutture d'indicizzazione appena esposte. Viene rappresentata la sequenza di chiavi 6, 8, 10, 11, 12, 20, 25, 30, 35, 50, 54, 56, 70, 71, 76, 80, 81, 89. Da notare come nel caso del B^+-Tree sia possibile che il valore di queste chiavi possa trovarsi sia nei nodi interni, sia nelle foglie dell'albero (11 e 50).

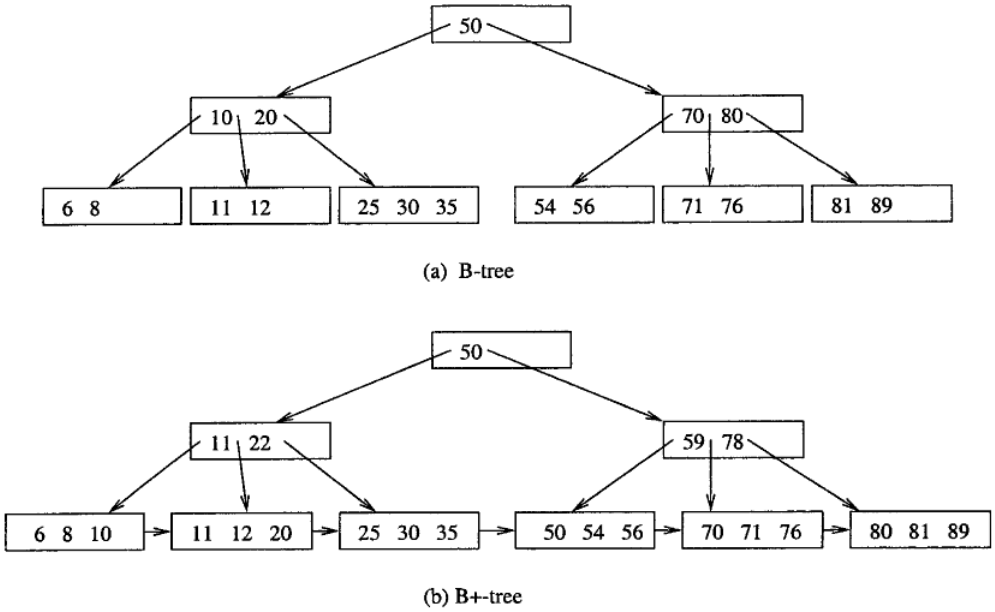

Figura 3.3: B-Tree e B^+-Tree a confronto [45].

Capitolo 4

Metodi d'accesso gerarchici per dati multidimensionali

Le basi di dati multidimensionali nascono da due necessità fondamentali: la prima è quella di riuscire a gestire i dati sotto forma di oggetti nello spazio, in maniera tale da supportare le numerose applicazioni che ne fanno uso, come ad esempio le geoscienze, il disegno CAD, le applicazioni radiografiche sanitarie, le aree della robotica, navigazione autonoma, protezione ambientale, e così via. La seconda necessità è legata al tentativo di organizzare le informazioni in maniera più vicina e subito interpretabile dall'utente, così da poterle analizzare sotto diverse prospettive e studiarne l'evoluzione temporale. I campi che fanno uso di questo principio riguardano l'analisi finanziaria, lo studio della profittabilità di un'organizzazione e tutte le applicazioni di natura economico-aziendale che utilizzano un approccio OLAP (Online Analytical Processing). Le basi di dati spaziali contengono oggetti in formato vettoriale con informazioni esplicite sulla loro estensione e posizione nello spazio. Per capire meglio i requisiti di questa categoria di sistemi è utile descrivere le proprietà relative al tipo di dati con cui lavorano.

4.1 Caratteristiche e requisiti delle basi di dati spaziali

I dati spaziali hanno una struttura complessa: possono rappresentare un singolo punto, un poligono o un poliedro a più dimensioni arbitrariamente distribuiti nello spazio. Non è quindi possibile memorizzare questi oggetti in una singola tabella relazionale, in quanto non vi è una dimensione fissa dei record. Nonostante alcune operazioni, come l'intersezione, siano più comuni rispetto ad altre, non esiste un'algebra comune per gli oggetti multidimensionali, poiché i dati spaziali dipendono fortemente dal dominio dell'applicazione. Da qui l'importanza di avere una classe di operatori geometrici supportati al livello fisico (sezione 4.3), che non si basino solamente sull'utilizzo di alcuni attributi alfanumerici, ma che facciano riferimento anche ad attributi posizionali nello spazio. Per supportare per esempio le operazioni di ricerca, non è possibile utilizzare le strutture di indicizzazione viste per le basi di dati tradizionali: il problema principale sta nella notevole difficoltà di mappare in una tabella relazionale oggetti a due o più dimensioni. Un approccio comune, ma totalmente inefficiente, è quello di gestire le ricerche multidimensionali tramite l'applicazione consecutiva di più strutture a chiave singola, una per ogni dimensione. In questo modo, però, si perde in concetto di alta selettività e integrazione dinamica dei dati; nasce così la necessità di progettare strutture d'indicizzazione ad hoc per dati spaziali. Esistono alcuni requisiti che un metodo d'accesso multidimensionale dovrebbe possedere [19]:

- *Dinamicità*: l'ordine delle operazioni non deve influenzare la struttura dei metodi d'accesso, che devono tenere traccia di ogni cambiamento avvenuto ai dati;

- *Gestione della memoria secondaria e terziaria*: garantire l'integrazione della memoria di secondo e terzo livello senza rallentamenti nella struttura;

- *Ampio supporto alle operazioni*: la struttura deve offrire compatibilità al maggior tipo di operazioni;
- *Indipendenza dei dati in input*: la distribuzione e la quantità dei dati in ingresso non deve influenzare l'efficienza della struttura;
- *Semplicità*: la struttura deve essere robusta, evitando di avere un'alta complessità dovuta alla sua progettazione per soddisfare contemporaneamente il maggior numero di applicazioni;
- *Scalabilità*: alto adattamento alla crescita della base di dati;
- *Efficienza nel tempo di esecuzione delle operazioni*: i metodi d'accesso dovrebbero garantire, nel caso peggiore, un costo logaritmico di ricerca per qualsiasi distribuzione di dati in ingresso;
- *Efficienza nello spazio di memorizzazione*: la struttura dovrebbe assicurare l'utilizzo minimo di memoria;
- *Concorrenza*: accessi multipli e concorrenziali ai dati, devono essere gestiti senza gravare sulla performance delle transazioni;
- *Impatto minimo*: l'impatto dell'integrazione di un metodo d'accesso in un DBMS deve essere minimo per ogni componente del sistema.

Queste caratteristiche dovrebbero offrire dei vantaggi sia in termini di efficienza nella presentazione e navigazione dei dati, sia nella facilità di manutenzione del sistema. Gli oggetti sono memorizzati nello stesso modo con cui vengono visualizzati ad alto livello, abbassando così il tempo di traduzione della richiesta dell'utente ad una specifica interrogazione per il gestore delle transazioni.

4.2 Rappresentazione dei dati a più dimensioni

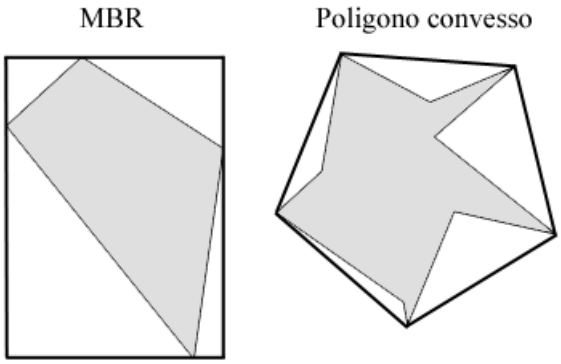

Figura 4.1: MBB rettangolare (MBR) e MBB approssimato al minimo poligono convesso.

I metodi d'accesso multidimensionali [19] si classificano in PAMs (Point Access Methods) e SAMs (Spatial Access Mehods). I primi sono progettati per basi di dati che memorizzano oggetti privi di estensione spaziale. I secondi invece sono in grado di gestire linee, poligoni o altri poliedri in spazi a più dimensioni; tali poliedri sono identificati tramite le coordinate dei vertici (che li caratterizzano univocamente) e possiedono sia attributi relativi alla loro estensione nello spazio, sia attributi di tipo non spaziale. Gli indici sono più efficienti se tutti gli oggetti hanno la stessa dimensione e, per questo motivo, spesso gli oggetti vengono approssimati ad una forma semplice come una sfera o un riquadro delimitativo (detto bounding boxo semplicemente box). Distribuito ogni oggetto o in uno spazio euclideo d-dimensionale E^d, dato un intervallo minimo di delimitazione $I_i(o) = [l_i, u_i](l_i, u_i \in E^1)$ che rappresenta l'estensione spaziale di o lungo le dimensioni i, definiamo il minimum bounding box (MBB) come $I^d(o) = I_1(o) \times I_2(o) \times \ldots \times I_d(o)$. Il MMB è un descrittore che approssima un generico solido a più dimensioni con un rettangolo (MBR) o con un poligono convesso. La Figura 4.1 mostra un esempio di MBB su due differenti figure geometriche.

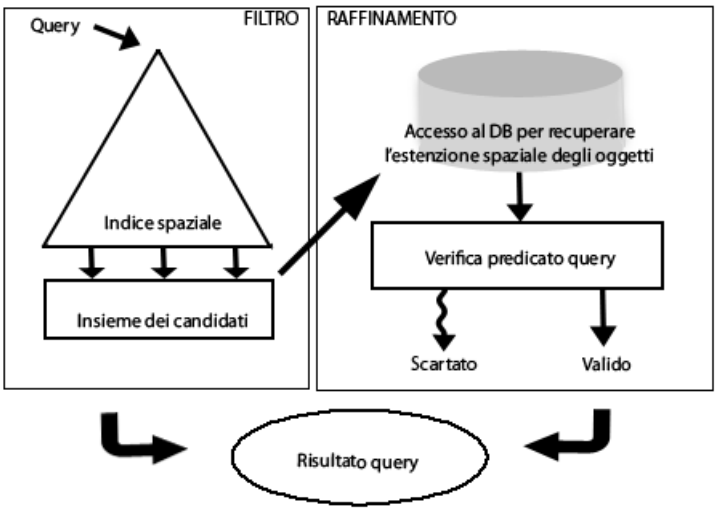

Figura 4.2: Valutazione delle chiavi durante una query spaziale [19].

Un indice gestisce solo il MBB di ciascun oggetto, associando un puntatore alla sua posizione all'interno della base di dati. In questo modo vengono filtrati un insieme di candidati alla soluzione; per ognuno di essi si controlla se il MBB sia sufficiente a soddisfare il predicato di ricerca. In caso positivo gli oggetti andrebbero direttamente aggiunti ai risultati dell'interrogazione; al contrario, se si avesse bisogno di maggiori informazioni per soddisfare la richiesta, si dovrebbe effettuare un processo di raffinamento dei candidati accedendo alla forma dell'oggetto in memoria secondaria: sarebbe, quindi, possibile verificarne la condizione di consistenza oppure rilevare un falso allarme. Questo processo viene mostrato nella Figura 4.2.

4.3 Tipo di query spaziali

Come detto in precedenza, non esiste né un algebra di base né un linguaggio di interrogazione standard per gli oggetti multidimensionali. Si possono comunque definire alcuni operatori comuni alla maggior parte delle basi di dati spaziali, utili a implementare le operazioni di ricerca, inserimento e cancellazione dei dati. Vediamo alcuni esempi che vengono trattati in [19].

Definiamo o gli oggetti presenti nella base di dati; ciascuno ha una propria estensione $o.G$.

1. Exact Match Query (EQM) o Object Query: dato un oggetto o', trova tutti gli oggetti o con estensione $o.G = o'.G$;

2. Point Query (PM): dato un punto p, trova tutti gli oggetti o che contengono p;

3. Window Query (WQ) o Range Query: dato un intervallo d-dimensionale I^d, trova tutti gli oggetti o che hanno almeno un punto in comune con I^d;

4. Intersection Query (IQ), Region Query o Overlap Query: dato un o' di estensione $o'.G$, trova tutti gli oggetti o che hanno almeno un punto in comune con o';

5. Encolsure Query (EQ): dato un oggetto o', trova tutti gli oggetti o che racchiudono o';

6. Containment Query (CQ): dato un oggetto o', trova tutti gli oggetti o che sono racchiusi in o';

7. Adiacent Query (AQ): dato un oggetto o', trova tutti gli oggetti o vicini (contigui) ad o';

8. Nearest-Neighbor Query (NNQ): dato in oggetto o', trova i k oggetti o che hanno minima distanza da o', ovvero la più piccola distanza euclidea tra i punti che chiudono il loro perimetro;

9. Spatial Join: date due collezioni R e S di oggetti e un predicato Ω, trova tutte le coppie (o, o') che verificano Ω. Il più importante tra i predicati di join è l'intersezione [20].

In Figura 4.3 si trova una rappresentazione grafica delle query appena descritte, con una loro formale descrizione matematica.

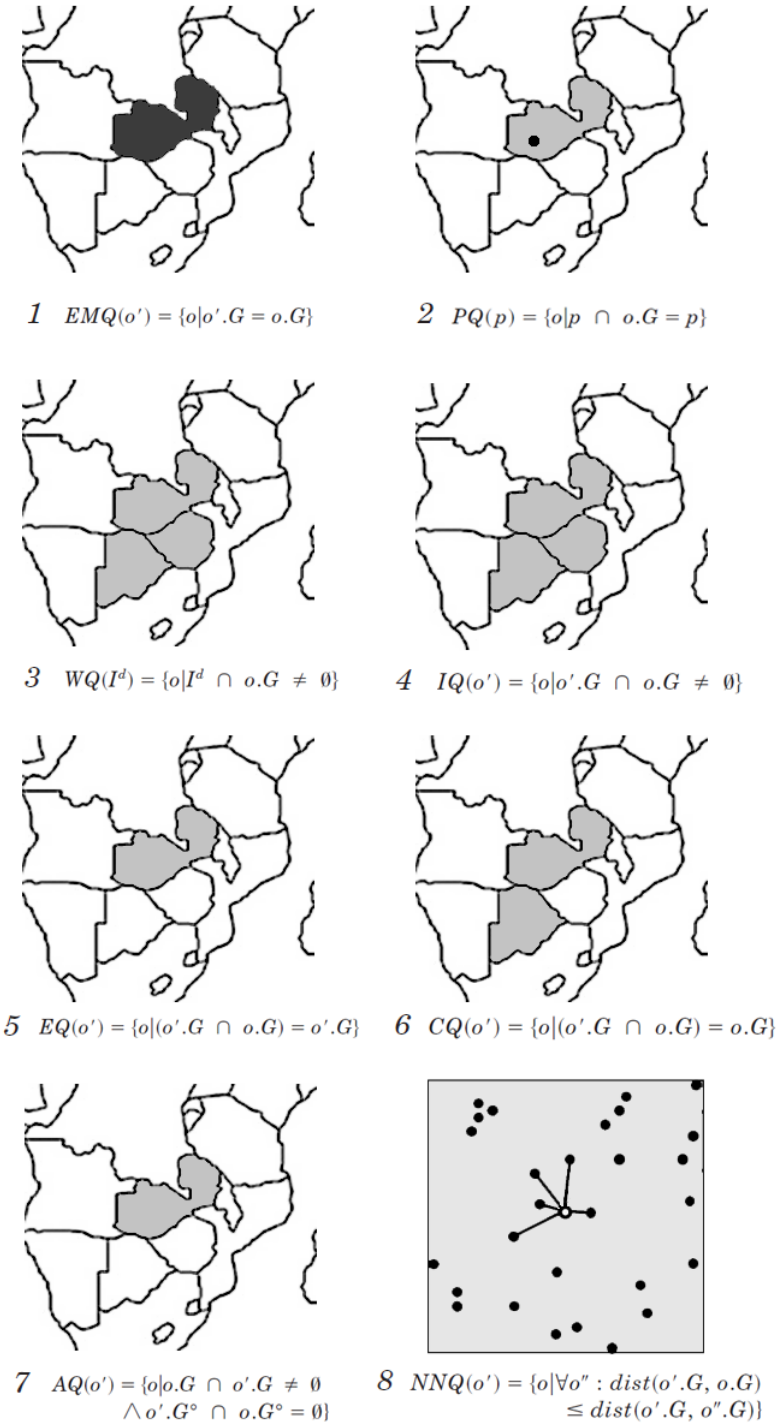

Figura 4.3: Rappresentazione grafica delle più comuni query in una base di dati multidimensionale [19].

A differenza dell'approccio basato sui B-Tree, l'efficienza delle strutture che manipolano oggetti con forme complesse, non dipende solamente dal numero di accessi a disco. Infatti, nella fase di raffinamento della ricerca spaziale, può capitare che il costo dell'utilizzo della CPU modifichi il bilanciamento delle operazioni di I/O [18] [29]. Nella pratica, tuttavia, come parametri di valutazione dell'efficienza si utilizzano sempre le informazioni relative al tempo di completamento di una ricerca in termini di accesso alle risorse.

4.4 Metodi d'accesso multidimensionali

A scapito della loro scalabilità, i primi metodi d'accesso multidimensionali sono nati per lavorare in memoria principale, senza la necessità di accedere al disco. Tra queste strutture dati fondamentali vi sono il k-d-Tree [5] [6] e le sue varianti (adaptive k-d-Tree [7], bintree [60]), il BSP-Tree [17] [16], il BD-Tree [39] e il quadtree [50] [51] [52]. In molte applicazioni, la quantità di dati spaziali da gestire è comprensibilmente ampia. Nelle prossime sezioni verranno descritte due strutture dati progettate per la memoria secondaria, ottimizzate attraverso il coordinamento delle loro operazioni da parte del sistema operativo.

4.4.1 Point Search Tree

I metodi d'accesso gerarchici per basi di dati spaziali sono, solitamente, la proiezione su più dimensioni dell'albero B-Tree visto in precedenza. In generale, i punti sono organizzati attraverso sottospazi dell'universo U^d chiamati regioni (o bucket), che corrispondono ognuno ad una pagina del disco e ad una foglia dell'albero con cui vengono indicizzati. I nodi interni, invece, sono utilizzati come mappa per guidare la ricerca. Le principali differenze tra queste strutture si basano principalmente sulle caratteristiche delle regioni; nei principali PAMs i bucket allo stesso livello dell'albero sono mutuamente suddivisi tra i nodi e la loro unione rappresenta l'intero spazio universo U^d.

Tra questi: k-d-B-Tree [48], LSD-Tree [28], hB-Tree [36] [37], BV-Tree [15]. Esistono inoltre strutture ibride, combinate con i principi degli schemi dinamici basati su hashing, come il BANG File [14] e il Buddy Tree [56]. Come rappresentante dei PAMs, verrà esaminato nel dettaglio il k-d-B-Tree. Per l'illustrazione grafica dei suoi metodi si farà riferimento al seguente spazio bidimensionale U^2 (Figura 4.4), in cui risiedono 10 punti pi e 10 poligoni ri con centro in ci (baricentro o centroide); il loro MMB è rappresentato da mi.

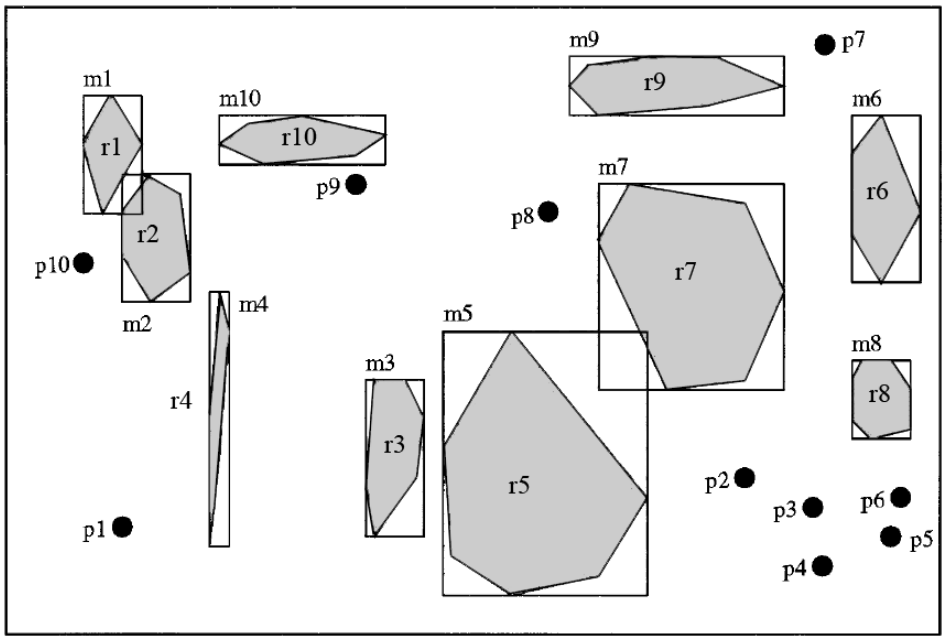

Figura 4.4: Spazio U^2 utilizzato per esemplificare graficamente le operazioni degli alberi descritti nelle sezioni successive [19].

k-d-B-Tree

Il k-d-B-Tree (k-dimensional B-Tree) [48] [19] combina le caratteristiche di bilanciamento dei B-Tree e quelle di partizionamento degli adaptive k-d-Tree per gestire oggetti sotto forma di punti. L'universo U^d viene, infatti, suddiviso in sottospazi mediante degli iperpiani $(d-1)$-dimensionali, posizionati lungo le d possibili direzioni (verticale ed orizzontale in U^2). I nodi indice dell'albero contengono la dimensione e le coordinate di queste partizioni, che condividono circa lo stesso numero di elementi al loro interno (almeno uno). Dato che i k-d-B-Tree gestiscono solamente dati di tipo punto, i poligoni vengono rappresentati mediante il loro baricentro *ci*. Come detto in precedenza, anche questa struttura presenta la proprietà di mutua esclusione tra lo spazio delle regioni allo stesso livello dell'albero: la loro unione forma l'intero insieme U^2. Le partizioni così suddivise, fanno sì che i corrispondenti punti vengano mappati nei rispettivi nodi foglia. Di seguito, in Figura 4.5, una rappresentazione grafica del partizionamento dell'universo U^2 e della struttura dell'albero.

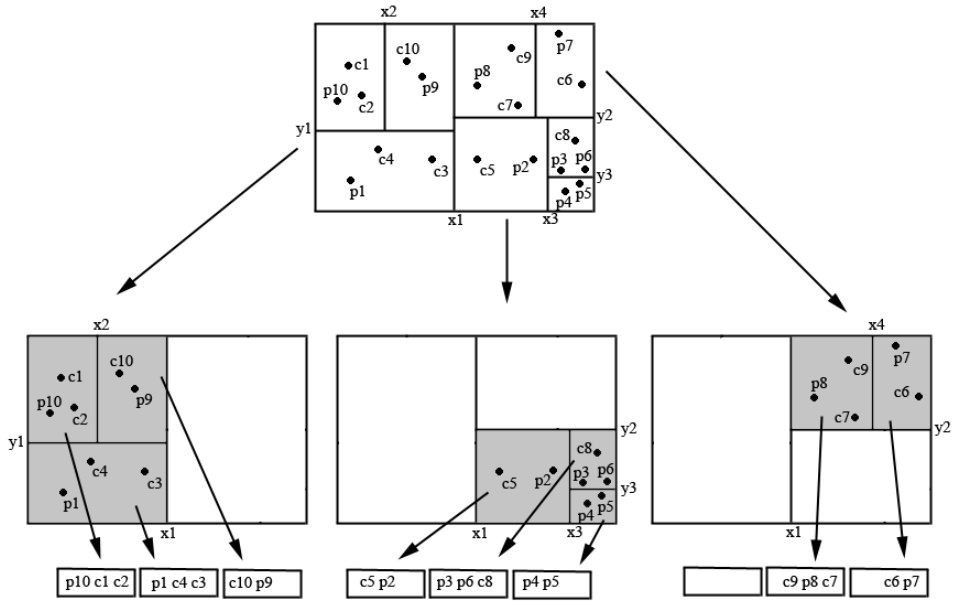

Figura 4.5: k-d-B-Tree [19].

Vediamo ora alcune operazioni che interessano i k-d-B-Tree. Data la varietà del tipo di query realizzabili sui dati spaziali, prendiamo come esempio la ricerca per regione. L'inserimento e la cancellazione di un punto, possono causare la riorganizzazione della struttura.

- *Ricerca (region query)*: la ricerca di una determinata regione dell'albero presenta due casi, diversificati in base al tipo di nodo che rappresenta la pagina cercata: (1) se la pagina è una foglia, vengono restituiti tutti i record a cui puntano gli elementi da cui è composta; (2) se la pagina è un nodo interno, si effettuano iterativamente tante ricerche quante sono le partizioni al suo interno, sin quando non si ricade nel caso (1).

- *Inserimento*: in primo luogo, l'inserimento di un generico punto p, prevede l'esecuzione di una Exact Match Query, che individua la foglia F in cui esso andrebbe inserito. Dopo la sua aggiunta, se la pagina non va in overflow il processo termina. In caso contrario si effettua uno split di F in base ad un'euristica di partizionamento [48], suddividendo i punti in due nuove pagine. Si procede poi con l'aggiornamento del nodo padre e, se anche lui dovesse finire in overflow, si ripete nuovamente lo split utilizzando un iperpiano opportuno.

- *Cancellazione*: come nel caso dell'inserimento, l'eliminazione di un punto avviene dopo una ricerca di tipo EMQ. Se il numero di elementi all'interno della pagina scende sotto una determinata soglia limite, si procede come nel caso dei B-Tree, ovvero con la concatenazione delle regioni adiacenti. Allo stesso modo dello split, anch'essa può propagarsi ai livello superiori interessando diversi nodi.

Grazie all'adattamento dell'albero alle varie distribuzioni di dati e alla proprietà per la quale tutte le foglie si trovano allo stesso livello, è possibile utilizzare l'analisi descritta per i B^+-Tree, per affermare che le operazioni di inserimento e cancellazione di un punto costano $O(logn)$ accessi a disco. Per quanto riguarda l'operazione di ricerca, lo studio sui k-d-Tree di Hemant M. Kakde [31], applicabile anche al dominio dei k-d-B-Tree, dimostra che è

possibile effettuare una ricerca di tipo Range Query in tempo $O(n^{1-\frac{1}{d}} + k)$, dove k è il numero di punti del risultato dell'interrogazione.

4.4.2 Spatial Search Tree

Con i metodi d'accesso visti finora, non è possibile realizzare la gestione di oggetti caratterizzati da estensione spaziale. Esistono, però, opportune tecniche per modificare e adattare i PAMs a questa necessità. Tra queste vi è la trasformazione (detta anche object mapping), con la quale è possibile riorganizzare la rappresentazione degli elementi nello spazio. Questa operazione può adattare ogni forma geometrica con d vertici ad un punto d-dimensionale [55], oppure può trasformare l'oggetto in un insieme di intervalli mediante curve di riempimento dello spazio (space-filling curves), che mappano gli oggetti in una lista di celle di una griglia (come per lo z-ordering [47]). Oltre ai PAMs, visti nella precedente sezione, fanno parte di questa categoria anche i zkdB+-Tree [46] e i P-Tree [30]. Un'altra tecnica di adattamento dei PAMs, è la sovrapposizione dei bucket (od object bounding). L'idea è di prendere come riferimento il Minimum Bounding Box degli oggetti per permettere che le regioni possano contenere l'intera figura geometrica; questo porta di conseguenza ad avere delle possibili sovrapposizioni di sottospazi all'interno dei nodi dell'albero. Esse causano, inevitabilmente, l'incremento del numero di percorsi di ricerca dei dati all'interno dell'albero. In termini di prestazioni, questo è un problema importante che potrebbe portare addirittura all'inefficienza dell'intera struttura. Esistono perciò degli studi per sviluppare tecniche euristiche di minimizzazione delle sovrapposizioni [49]. Tra le strutture che fanno uso di object bounding ci sono gli R-Tree [22], R*-Tree [4], skd-Tree [44], GBD-Tree [40], Hilbert R-Tree [32], Buddy Tree con overlapping [54], sphere-Tree [61], P-Tree [53], KD2B-Tree [61]. Un'ulteriore tecnica, infine, fa uso del ritaglio degli oggetti (object duplication), nel caso in cui siano eccessivamente ampi per essere memorizzati all'interno di un bucket dell'albero. In questo modo si evitano le sovrapposizioni tra regioni e vengono utilizzati più bucket per un solo oggetto, causando però in alcuni casi la duplicazione delle sue informazioni e in altri la dispersione dei dati su

più pagine di memoria. Di conseguenza, nonostante esista un unico percorso di ricerca tra i nodi dell'albero, si ha un aumento dei tempi medi di recupero degli oggetti e della frequenza di overflow dei bucket. Inoltre, sempre a causa dell'impossibilità di sovrapposizione tra le regioni, può accadere che sia molto complicato allargare lo spazio per un nuovo elemento. Gli overflow delle pagine possono quindi causare il blocco dell'intera struttura. Sono affetti da questi problemi gli extended k-d-Tree [38], R^+-Tree [57], Buddy Tree con clipping [54] e i cell Tree [21].

In questa sede verrà trattata la struttura degli R-Tree come rappresentante dei SAMs.

R-Tree

Un R-Tree (Rectangular Tree) [22] è un albero bilanciato in altezza che suddivide lo spazio in un insieme di regioni innestate gerarchicamente e con possibili sovrapposizioni. Ogni nodo corrisponde ad una pagina di memoria e ad un intervallo di MBB: quelli interni identificano lo spazio che occupano i nodi discendenti, mentre le foglie (che si trovano tutte allo stesso livello) contengono il MBB degli elementi memorizzati al loro interno. Oltre al MBB, ogni nodo salva un riferimento alla descrizione completa dell'oggetto. Il numero di entry (e quindi di puntatori) di ciascun nodo varia da un minimo m ad un massimo M (esclusa la radice); il primo assicura un'efficiente utilizzo dello spazio di memoria mentre il secondo identifica la dimensione di una pagina, definendo il limite superiore sopra il quale il nodo deve essere suddiviso. La divisione di una regione può avvenire in diversi modi e secondo diversi algoritmi con valutazioni di costo differenti [22], ma l'obiettivo principale è quello di causare il minor ampliamento dei confini dei rettangoli (Minimum Bounding Rectangular, MBR) in maniera tale da minimizzare le sovrapposizioni tra i vari bucket. Le figure 4.6 e 4.7 rappresentano rispettivamente l'organizzazione dello spazio U^2 descritto in Figura 4.4 e la struttura del relativo albero R-Tree.

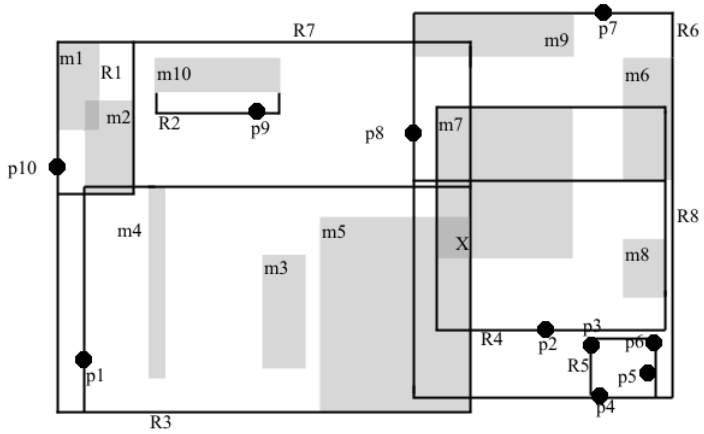

Figura 4.6: Organizzazione dello spazio U^2 da parte di un R-Tree [19].

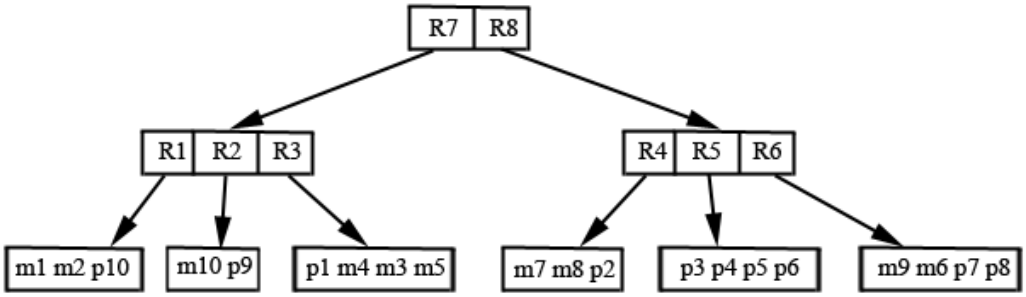

Figura 4.7: R-Tree [19].

- Ricerca (Point Query e Window Query): l'operazione di ricerca è simile a quella vista per i B-Tree. Partendo dalla radice, viene visitato il nodo verificando quali data entry rispondono positivamente al predicato d'intersezione con il MBR passato come input. La ricerca procede quindi in maniera ricorsiva sino alle foglie in cui si sono memorizzati i puntatori ai record. Nel caso peggiore, quando tutti gli MBB sono sovrapposti, potrebbero essere visitati tutti i nodi indice interni all'albero con un costo pari a $O(n)$. Al contrario, nel caso ottimo in cui non esistono sovrapposizioni, si ha un numero di accessi a disco di $O(log_m n)$.

- Inserimento: l'inserimento di un oggetto o, prevede la visita ricorsiva dell'albero per identificare la regione che richiede il minimo allargamento per ospitare o. A parità di estensione, viene scelto il nodo con dimensione minima. Questo assicura che non ci sia dispersione dell'oggetto su più pagine. Una volta raggiunto il nodo foglia corretto, viene aggiunto o. Se il bucket rispetta il limite superiore M, il processo viene terminato, altrimenti si procede con l'operazione di split che potrebbe propagarsi ai livello superiori dell'albero.

- Cancellazione: la cancellazione avviene in maniera subordinata ad una EMQ, che trova o meno l'oggetto da eliminare. Se l'operazione rispetta il limite inferiore m dello spazio della pagina, viene controllata la possibilità di ridurre la dimensione della regione interessata. In caso di underflow, invece, il nodo viene copiato in un nodo temporaneo e quello originale è rimosso dall'albero. Successivamente le entry vengono redistribuite tra le foglie. In entrambe i casi, i cambiamenti causano la riorganizzazione dei livelli superiori alla foglie.

Esistono varie metodologie per minimizzare le sovrapposizioni durante l'inserimento di nuovi oggetti. Come detto in precedenza, Guttman propone diversi algoritmi tra cui uno con costo quadratico ed uno più semplice con complessità lineare rispetto al numero dei nodi dell'albero [22]. Inoltre, per far ciò, esistono anche delle varianti del R-Tree che utilizzano una gerarchia di forme sferiche d-dimensionali (sphere tree [61]) oppure calcolano una partizione ottimale per un universo spaziale conosciuto a priori (packed R-Tree [49]) oppure combinano le sovrapposizioni tra le regioni con la tecnica del riempimento delle curve nello spazio (Hilbert R-Tree [32]).

Capitolo 5

Generalizzazione delle strutture gerarchiche

La scarsa flessibilità delle strutture presentate sinora, risulta spesso problematica quando si ha la necessità di estendere o modificare le funzionalità degli alberi di ricerca. Per esempio, partendo dallo schema e dalle proprietà dei B-Tree, è possibile costruire un indice di dati alfanumerici a chiavi ordinate, che supporta solamente ricerche in base a predicati di uguaglianza o secondo un intervallo di valori. In maniera simile, un indice basato su un R-Tree, potrebbe esclusivamente mettere a disposizione ricerche secondo predicati di contenimento, sovrapposizione o distanza tra elementi spaziali. L'adattamento dei metodi d'accesso al determinato dominio d'interesse e ai nuovi tipi di dato da gestire, deve garantire l'efficienza delle basi di dati per cui sono implementati. Pertanto, per superare il limite degli schemi specializzati e minimizzare lo sforzo di progettazione di strutture dati ad hoc, Hellerstein, Naughton e Pfeffer hanno progettato una struttura astratta, ampiamente estendibile da chi ne fa uso: il Generalized Search Tree (GiST) [26].

5.1 Generalized Search Tree

Il GiST riunisce i principi dei B^+-Tree e R-Tree con lo scopo di semplificare la definizione dei metodi d'accesso gerarchici e offrire un concreto supporto sia a nuovi tipi di dato definiti dall'utente, sia alle relative operazioni di base quali la ricerca, l'inserimento e la cancellazione di un record. Uno dei principi su cui si basa il funzionamento della struttura è quello relativo alla generalizzazione del concetto di query, che viene tradotta in un arbitrario predicato q. In questo modo, è l'utente stesso a specificare il metodo per determinare la consistenza tra i dati; viene creata così una categorizzazione delle chiavi di ricerca, per far sì che l'accesso al sottoalbero identificato da un puntatore, possa avvenire solamente se la chiave ad esso associato è consistente rispetto a q (proprietà di monotonicità del predicato q). Ogni nodo è formato da un insieme di entry $E = (p, ptr)$, che si distinguono in base al nodo di cui fanno parte: se E è un'entry di un nodo interno, allora p è il predicato che identifica la chiave di ricerca e ptr è il puntatore al nodo di livello inferiore; se, al contrario, l'entry è contenuta in una foglia, p è un valore di chiave che identifica una tupla nella base di dati, puntata da ptr. Esistono, inoltre, alcune proprietà di invarianza che la struttura deve sempre rispettare: il GiST è un albero paginato e perfettamente bilanciato in altezza, per cui tutte le foglie si trovano allo stesso livello. Ogni nodo è caratterizzato da un fattore minimo di riempimento k, diversamente dalla radice che deve avere almeno due nodi figli. Nel caso di entry a lunghezza fissa, il fattore varia da kM ad un massimo di M entry, (per cui $\frac{2}{M} \leq k \leq \frac{1}{2}$); nel caso di entry a lunghezza variabile si usano metriche relative alla loro dimensione.

L'interfaccia del GiST offre una serie di funzioni per la gestione dei valori di chiave (Key Methods), che vengono definiti prima di istanziare la classe ed invocati dai metodi che mette a disposizione l'albero per le operazioni di ricerca, inserimento e cancellazione (Tree Methods).

5.1.1 Key Methods

La classe delle chiavi è aperta alla ridefinizione da parte dell'utente ed formata dai sei metodi. Per ognuno di questi si fornirà una breve descrizione dello scopo, dei parametri necessari come input e l'output restituito come risultato.

Consistent(E,q)

- Input: Entry $E = (p, ptr)$, Predicato q
- Output: Boolean

Lavora con predicati arbitrari per valutare lo spazio di ricerca relativo ad un query. Restituisce *true* se e solo se il predicato del sottoalbero identificato da $E.p$ risulta consistente con il predicato di query q.

Union(P)

- Input: Lista di entry $P = (p_1, ptr_1), \ldots, (p_n, ptr_n)$
- Output: Predicato r

Restituisce il predicato r comune a tutte le tuple raggiungibili dai puntatori ptr_i, identificando di fatti il predicato che caratterizza la entry padre delle P.

Compress(E)

- Input: Entry $E = (p, ptr)$
- Output: Entry $E' = (p', ptr)$

Fornisce una rappresentazione p' compressa e più efficiente, del predicato $E.p$. In maniera implicita, i dati vengono compressi al momento della scrittura su disco e decompressi in fase di lettura.

Decompress(E)

- Input: Entry $E' = (p', ptr)$
- Output: Entry $E = (r, ptr)$

E' l'operazione inversa a *Compress(E)*: si ottiene un predicato r che, anche in caso di perdita di dati nella precedente compressione, soddisfa $p \rightarrow r$.

Penalty(E1,E2)

- Input: Entry $E_1 = (p_1, ptr_1)$, Entry $E_2 = (p_2, ptr_2)$
- Output: Numeric

Calcola un valore rappresentante la penalità di una determinata scelta nell'alternativa tra due sottoalberi. Viene utilizzata nei tree methods di *Split* e *Insert*.

PickSplit(P)

- Input: Lista di entry $P = (p_1, ptr_1), \ldots, (p_{m+1}, ptr_{m+1})$
- Output: Lista di Entry P^1, Lista di Entry P^2

Suddivide le entry in P, in due sottoliste P^1 e P^2. Ciascuna di queste liste ha cardinalità maggiore o uguale a kM.

5.1.2 Tree Methods

I metodi descritti in questa sezione sono specificati direttamente dal GiST ed fanno uso dei Key Methods per implementare i relativi algoritmi. Verranno esposte le operazioni di ricerca, inserimento e cancellazione di un entry nell'albero, che fanno uso, a loro volta, di ulteriori metodi quali la scelta di un sottoalbero tra più alternative disponibili, la suddivisione del nodo in caso di overflow e la riorganizzazione delle chiavi e della struttura dell'albero. Per alcuni di questi verrà riportato, altresì, lo pseudocodice descritto in [26]; per la cancellazione, invece, si rimanda al documento completo di Hellerstein, Naughton e Pfeffer [29] nel caso si voglia approfondirne lo studio.

Ricerca

La ricerca attraversa l'albero G partendo dalla radice R; utilizza il metodo *Consistent(E,q)* per determinare l'esistenza di un percorso da effettuare sino al raggiungimento dell'ultimo livello di G. Una volta individuata la foglia corretta, l'algoritmo restituisce tutte le entry consistenti rispetto a q, che puntano ai record dei dati. Da notare che il predicato q può essere basato su criteri di uguaglianza, come nel caso dei B^+-Tree, oppure può essere soddisfatto da più valori, con riferimento agli R-Tree, alle Range Query e Window Query.

```
algorithm search(Node R, Predicate q) > tuples
tuples := ∅
if !(R is a leaf) then
  for each entry E do
   if Consistent(E,q) then
     search(E.ptr,q)
   end if
  end for
else
   for each entry E do
    if Consistent(E,q) then
     tuples := E.ptr
    end if
   end for
end if
return tuples
```

Per ricerche nel campo di domini lineari ordinati, è possibile migliorare l'efficienza dell'algoritmo di ricerca appena visto attraversando la struttura una sola volta. Per far ciò si recupera l'entry consistente più a sinistra dell'albero

(quella con chiave minima) e si esaminano sequenzialmente le entry successive. Questa tecnica richiede, però, alcuni accorgimenti ed ulteriori metodi da definire in fase di creazione del GiST: è necessario istanziare a *true* una variabile booleana per esprimere la presenza di un ordine nelle chiavi; bisogna creare un Key Methods addizionale *Compare(E^1,E^2)* che valuta il confronto tra due entry; infine, si deve garantire che i metodi mantengano la proprietà di ordinamento durante le varie operazioni offerte dal GiST ed assicurino che due chiavi nello stesso nodo non vengano mai sovrapposte. Di seguito si descrivono gli algoritmi utili ad implementare una ricerca di tipo sequenziale.

```
algorithm find_min(Node R, Predicate q) > tuple
if !(R is a leaf) then
  find the first entry E that match Consistent(E,q)
   if (E exist) then
   find_min (E.ptr,q)
  else
   return NULL
  end if
else
  find the first entry E1 that match Consistent(E1,q)
  if (E1 exist) then
   return E1
  else
   return NULL
  end if
end if
```

Una volta trovata l'entry $E1$ con chiave minima dell'albero, si procede con l'analisi della consistenza dei nodi successivi a quello in cui si trova $E1$.

```
algorithm next(Node R, Predicate q, Current_entry E) > tuple
if !(E rightmost entry in R) then
  N := next entry to E
else
  P := next node to R on the same level
  if !(P exist) then
   return NULL
  else
   N := leftmost entry on P
  end if
end if
if (Consistent(N,q)) then
  return N
else
  return NULL
end if
```

Inserimento

Il GiST è un albero bilanciato in altezza. Questa proprietà deve essere mantenuta invariata anche dopo operazioni di inserimento di una entry. L'algoritmo descritto successivamente, permette di specificare il livello al quale si vuole inserire l'elemento specifico E ed è simile al procedimento visto per gli R-Tree. In [26] si assume che il numero dei livelli parta da zero, ovvero quello delle foglie. Una volta trovato il nodo L in cui aggiungere E, si controlla se vi è abbastanza spazio per procedere alla sua memorizzazione. In caso positivo, l'algoritmo riorganizza le chiavi ai livelli superiori, dopo di che termina; al contrario, se la pagina di memoria non può contenere E, si suddivide il nodo L prima della riorganizzazione dell'albero. Questo procedimento è descritto dal seguente pseudocodice.

```
algorithm insert(Node R, Entry E, Level l) > GiST
//find and store in L where E should go
L := ChooseSubtree(R,E,l)
if !(L+E in overflow) then
  L := E
else
  Split(R,L,E)
end if
return AdjustKeys(R,L)
```

L'algoritmo *choose_subtree* sceglie il miglior nodo in cui inserire E, rispettando, se specificato, l'ordinamento delle chiavi nelle foglie.

```
algorithm choose_subtree(Node R, Entry E, Level l) > Node
if (R.level == l) then
  return R
else
  //F1,F2: first and second entry on R
  min_penalty := Penalty(F1,F2)
  Fa := F1
  Fb := F2
  for each entry Fi from F3 to FM-1 on R do
   if(Penalty(Fi,Fi+1) < min_penalty) then
     min_penalty := Penalty(Fi,Fi+1)
     Fa := Fi
     Fb := Fi+1
   end if
  end for
 return choose_subtree(Fa.ptr,Fb,l)
```

```
end if
```

Lo split utilizza il metodo *PickSplit* per suddividere le entry nei nodi appropriati, inserirli nell'albero e aggiornare le chiavi al livello inferiore.

```
algorithm split(Node R, Node N, Entry E) > GiST
P := ∅
for each entry E of N do
  P := E
end for
N1 := new Node()
L[] := new Array(2)
L := PickSplit(P)
//put partitions in nodes
N := L[0]
N1 := L[1]
//insert entry EN1 for N1 in parent
for each entry E1 of N1 do
  P1 := E1
end for
q := Union(P1)
ptr1 := pointer to N1
EN1 := (q, ptr1)
if !(Parent(N)+EN1 in overflow) then
  Parent(N) := EN1 //there is room
else
  Split(R,Parent(N), EN1) ////there is no room
end if
//modify entry F in Parent(N) which points to N
qN := Union(P)
```

```
F.p := qN
return R
```

L'aggiornamento delle chiavi durante il procedimento di *Split* deve essere propagato (come specificato da *insert*) ai livelli superiori dell'albero. Questo è compito dell'algoritmo *adjust_keys*, il quale assicura la correttezza dei predicati all'interno del GiST, che devono caratterizzare ognuno il loro rispettivo sottoalbero. Quando le chiavi sono disposte mediante un criterio di ordinamento, il metodo *adjust_keys* tipicamente non è necessario. Infatti, i predicati all'interno dei nodi partizionano il dominio in base a intervalli di valori e non richiedono di essere aggiornati successivamente a semplici inserimenti o cancellazioni di una entry.

```
algorithm adjust_keys(Node R, Node N) > GiST
for each entry EN in N do
  P := EN
end for
rN := Union(P)
//pN is entry E predicate which points to N
if (N is root OR pN implies rN) then
  N := next entry to E
else
  pN := implies rN
  adjust_keys(R, Parent(N))
end if
```

Cancellazione

Come per l'inserimento, anche la cancellazione di una entry deve mantenere l'albero bilanciato. Rimandiamo a [25] per una trattazione completa sulle due tecniche principali, una basata sulle redistribuzioni e le concatenazioni utilizzate nello schema dei B^+-Tree e l'altra riferita al concetto di reinserimento visto per gli R-Tree.

Nel caso di strutture ad albero bilanciate, che non presentano al loro interno alcuna sovrapposizione degli spazi, è possibile definire il limite superiore di nodi visitati durante la ricerca di una chiave: esso esamina infatti $O(logn)$ nodi. Le sovrapposizioni, possibili sia per gli R-Tree che per i GiST, causano molteplicità di percorsi all'interno dell'albero e condizionano notevolmente le prestazioni del metodo d'accesso. Possono essere: sovrapposizioni sui dati, che creano un'inefficienza di indicizzazione a causa della collisione delle chiavi, oppure sovrapposizioni date dalla perdita di informazione durante la compressione imprecisa dei dati, che rendono le chiavi consistenti ad ogni tipo di query.

5.2 Implementazione di un R-Tree tramite GiST

L'interfaccia del GiST prevede un alto livello di astrazione, che esula dalla conoscenza dell'architettura e della logica della base di dati sottostante. Essa combina in un'unica struttura i principi di riusabilità del codice, estendibilità e generalizzazione dei domini di applicazione, fornendo così un modello d'implementazione di indici definiti dall'utente. In questo senso, è bensì possibile ricostruire lo schema e le operazioni messe a disposizione sia dai B+-Tree, sia dagli R-Tree.

In questa sezione si modificheranno i Key Methods, visti nella sezione 5.1.1, per simulare il comportamento di un R-Tree. Teodor Sigaev and Oleg Bartunov sono i responsabili dell'implementazione GiST all'interno dei metodi d'indicizzazione offerti dal PostgreSQL RDBMS [43]. Grazie al loro

lavoro, è disponibile un'ampia documentazione e un notevole supporto a livello di moduli addizionali. Tra questi, verrà utilizzato il Gevel Module [42] per analizzare le statistiche di un indice R-Tree, costruito su un campione di dati relativo alle città presenti sul territorio della Grecia [41].

5.2.1 R-Tree in spazi bidimensionali

Lo scopo di questo paragrafo è mostrare un esempio di implementazione dei Key Methods, per la creazione di un R-Tree che mappa oggetti a due dimensioni in uno spazio rappresentato da un piano cartesiano [26]. Le figure possono essere modellate tramite bounding box rettangolari e memorizzate nell'albero attraverso chiavi composte da 4 valori $(x_{ul}; y_{ul}; x_{lr}; y_{lr})$ e una variabile libera v: la coppia $(x_{ul}; y_{ul})$ rappresenta le coordinate dell'angolo superiore sinistro dell'MBR mentre la coppia di valori $(x_{lr}; y_{lr})$ si riferisce a quelle dell'angolo inferiore destro. I predicati di ricerca supportati da queste chiavi sono *Contains(box,v)*, *Overlap(box,v)* e *Equal(box,v)*.

Contains(box,pbox)

- Input: 4-tuple $box = (x^1_{ul}; y^1_{ul}; x^1_{lr}; y^1_{lr})$,
  Predicato $pbox = (x^2_{ul}; y^2_{ul}; x^2_{lr}; y^2_{lr})$
- Output: Boolean

Restituisce *true* se e solo se $(x^1_{lr} \geq x^2_{lr}) \wedge (x^1_{ul} \leq x^2_{ul}) \wedge (y^1_{lr} \geq y^2_{lr}) \wedge (y^1_{ul} \geq y^2_{ul})$.

Overlap(box,pbox)

- Input: 4-tuple $box = (x^1_{ul}; y^1_{ul}; x^1_{lr}; y^1_{lr})$,
  Predicato $pbox = (x^2_{ul}; y^2_{ul}; x^2_{lr}; y^2_{lr})$
- Output: Boolean

Restituisce *true* se e solo se $(x^1_{ul} \leq x^2_{lr}) \wedge (x^2_{ul} \leq x^1_{lr}) \wedge (y^1_{lr} \leq y^2_{ul}) \wedge (y^2_{lr} \leq y^1_{ul})$.

Equal(box,pbox)

- Input: 4-tuple $box = (x^1_{ul}; y^1_{ul}; x^1_{lr}; y^1_{lr})$,
  Predicato $pbox = (x^2_{ul}; y^2_{ul}; x^2_{lr}; y^2_{lr})$
- Output: Boolean

Restituisce *true* se e solo se $(x^1_{lr} = x^2_{lr}) \wedge (x^1_{ul} = x^2_{ul}) \wedge (y^1_{lr} = y^2_{lr}) \wedge (y^1_{ul} = y^2_{ul})$.

Grazie a questi predicati, è possibile definire i Key Methods di un R-Tree. A causa della varietà di scelta di un criterio di *PickSplit*, questa trattazione viene rimandata a [22, 4].

Consistent(E,q)

- Input: Entry $E = (p, ptr)$, Predicato q
- Output: Boolean

Dato $p = Contains((x^1_{ul}; y^1_{ul}; x^1_{lr}; y^1_{lr}), v)$ e $q = (x^2_{ul}; y^2_{ul}; x^2_{lr}; y^2_{lr})$ un predicato Constains, Overlap o Equal, il metodo restituisce *true* se e solo se è verificato *Overlap(p,q)*.

Union(P)

- Input:
  Lista di entry $P = ((x^1_{ul}; y^1_{ul}; x^1_{lr}; y^1_{lr}), ptr_1), \ldots, ((x^n_{ul}; y^n_{ul}; x^n_{lr}; y^n_{lr}), ptr_n)$
- Output: $(x^{MIN}_{ul}; y^{MAX}_{ul}; x^{MAX}_{lr}; y^{MIN}_{lr})$

Restituisce le coordinate dell'angolo superiore sinistro e di quello inferiore destro del MBR che verifica la condizione espressa in output.

Compress(E)

- Input: Entry $E = (p, ptr)$
- Output: Entry $E' = (p', ptr)$

Dato un poligono memorizzato come un insieme di linee $l^i = (x_1^i, y_1^i, x_2^i, y_2^i)$ viene costruita una rappresentazione compatta di
$p' = (\forall_i MIN(x_{ul}^i), \forall_i MAX(y_{ul}^i), \forall_i MAX(x_{lr}^i), \forall_i MIN(y_{lr}^i))$.
Anche in questo caso, i dati vengono compressi in maniera implicita al momento della scrittura su disco e decompressi in fase di lettura.

Decompress(E)

- Input: Entry $E' = (p', ptr)$
- Output: Entry $E = (r, ptr)$

E' l'operazione inversa a *Compress(E)*, per esempio la funzione identità.

Penalty(E1,E2)

- Input: Entry $E_1 = (p_1, ptr_1)$, Entry $E_2 = (p_2, ptr_2)$
- Output: Numeric

Calcola $q = Union(E_1, E_2)$ e restituisce un valore di penalità uguale alla differenza $area(q) - area(E_1.p)$ [22].

Grazie alla sua estendibilità e all'interfaccia facilmente comprensibile, il GiST può essere implementato per la costruzione sia di indici tradizionali, sia di metodi d'accesso spaziali come gli R-Tree (comprese le sue varianti), con costi computazionali identici a quelli esaminati nei capitoli precedenti.

5.2.2 PostgreSQL: rappresentazione grafica di un R-Tree utilizzando Gevel

La distribuzione sorgente dell'RDBMS di PostgreSQL comprende alcuni esempi di metodi d'indicizzazzione implementati tramite GiST. La valutazione e l'analisi delle statistiche di queste strutture è realizzabile grazie ad un modulo aggiuntivo creato da Teodor Sigaev and Oleg Bartunov: il Gevel Module [42]. Grazie al Gevel è, inoltre, possibile visualizzare la struttura degli indici e dei livelli che compongono l'albero. Di seguito, si riporta il test di partizionamento, effettuato in [41], su una collezione di punti rappresentanti i villaggi sul territorio della Grecia (Figura 5.1). Verrà costruita su di essa una struttura R-Trre implementata con interfaccia GiST e ne saranno analizzate graficamente le caratteristiche.

Figura 5.1: I punti sulla mappa rappresentano i villaggi presenti sul territorio della Grecia .

E' possibile trovare un tutorial approfondito sui comandi e sui dettagli della procedura tramite il riferimento [41] .
Inizialmente, è necessario creare un nuovo database e caricarvi i moduli relativi all'albero R-Tree e Gevel. Successivamente, devono essere importati i dati di test e su questi creato l'indice GiST. Come detto in precedenza, Gevel ha l'utilità di estrarre numerose informazioni sulle strutture che analizza. In Figura 5.2 vengono riportate alcune di queste statistiche, le quali suggeriscono che sono stati sufficienti due livelli dell'albero per indicizzare 6782 tuple.

```
Number of levels:          2
Number of pages:           64
Number of leaf pages:      63
Number of tuples:          6782
Number of leaf tuples:     6719
Total size of tuples:      298408 bytes
Total size of leaf tuples: 295636 bytes
Total size of index:       524288 bytes
```

Figura 5.2: Gevel: statistiche di composizione di un R-Tree.

I comandi di stampa grafica degli indici e quelli di normalizzazione dei dati sono rimandati a [41]. Si riporta, invece, la partizione in regioni effettuata dall'R-Tree sullo spazio in Figura 5.1. I bounding box di colore nero rappresentano le regioni presenti nella radice dell'albero, mentre, quelli delimitati di rosso, sono le entry contenute al livello delle foglie (Figura 5.3).

E' possibile notare come nella parte laterale sinistra dell'immagine, dove si registra un maggior numero di città e villaggi, si verifica anche un'alta frequenza di sovrapposizioni tra le regioni. Questo causa un peggioramento delle prestazioni della struttura, che sarebbe possibile migliorare ottimizzando la partizione dello spazio universo, applicando per esempio la tecnica dei packed R-Tree [49], una delle varianti degli R-Tree.

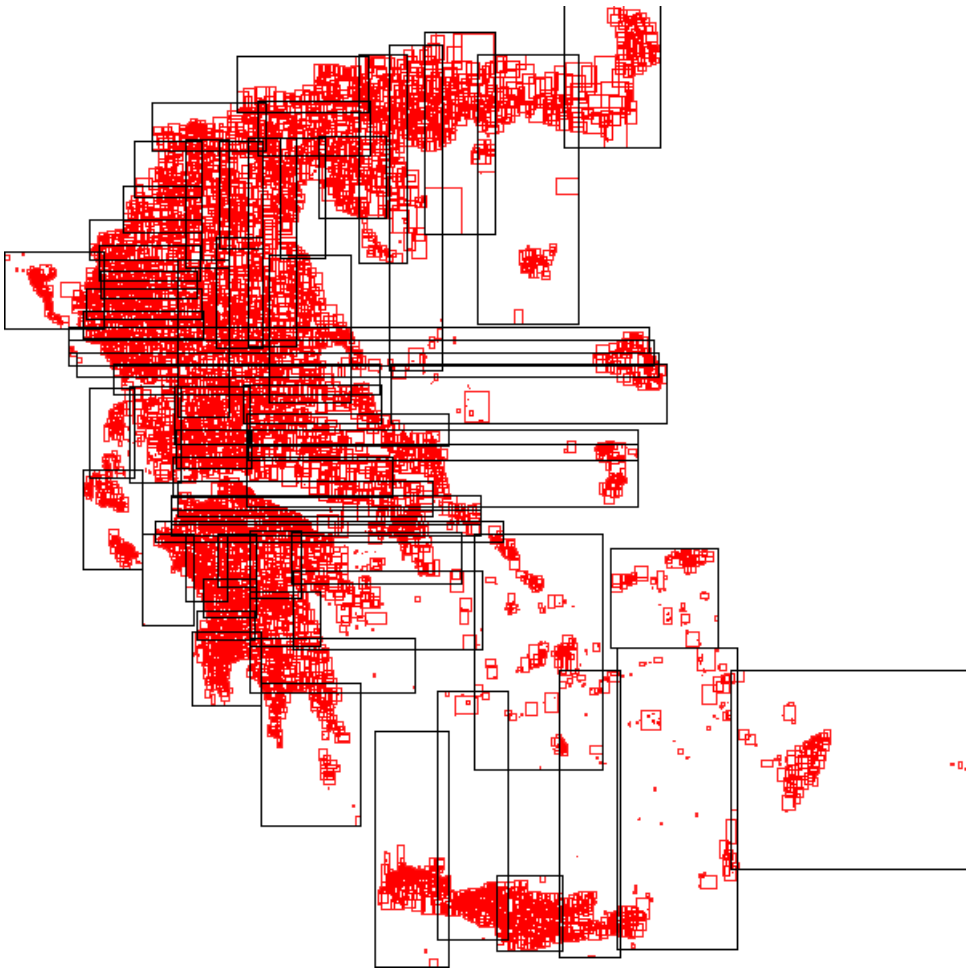

Figura 5.3: Rappresentazione grafica dei livelli dell'albero R-Tree [41].

Conclusioni

Il supporto ai metodi d'accesso è uno dei fattori critici per le prestazioni di un DBMS. Nelle tradizionali basi di dati relazionali, i B^+-Tree [10] garantiscono un alto livello di prestazioni delle applicazioni che utilizzano tipi di dato standard, come per esempio i formati numerici, le stringhe e le date. Queste strutture, però, non sono efficienti nel campo di sistemi che fanno uso di dati spaziali a più dimensioni (tra cui rientrano anche gli oggetti multimediali). In riferimento a ciò, viene richiesta la progettazione di nuovi metodi d'accesso di specifico dominio, nei quali si collocano per esempio gli alberi k-d-B-Tree [48] e R-tree [22]. La continua necessità di costruire dei metodi d'indicizzazione adatti a nuovi tipi di dati e l'evidente difficoltà di implementarli partendo da zero, hanno portato ad incapsulare le funzioni offerte dai metodi d'accesso standard in una struttura generalizzata, astratta, flessibile e indipendente sia dai dati gestiti, sia dal suo dominio di applicazione: il GiST [25]. Nelle tabelle 5.1, 5.2 e 5.3 vengono riassunti i metodi d'accesso visti durante la rassegna.

Tipo di base di dati		
	Tradizionale (dati ad una dimensione)	B-Tree [1] [9], B^+-Tree [9], B*-Tree [9], Prefix B+-tree [2], String B-tree [11]
	Mutidimensionale (Spaziale, multimediale, ecc.)	Point Access Methods (Tab.2), Spatial Access Methods (Tab.3)

Tabella 5.1: Metodi d'accesso gerarchici suddivisi per tipo di base di dati.

			PROPRIETA'		
			Forma: scatola[1]	U^d completo[2]	Regioni disgiunte[3]
POINT ACCESS METHODS	*Mem. Principale*	k-d-Tree [4] [5]	×	×	×
		adaptive k-d-Tree [6]	×	×	×
		bintree [56]	×	×	×
		quadtree [46] [47] [48]	×	×	×
		BSP-Tree [15] [16]		×	×
		BD-Tree [28]		×	×
	Mem. Secondaria	k-d-B-Tree [44]	×	×	×
		LSD-Tree [27]	×	×	×
		Buddy Tree [52]	×		×
		hB-Tree [35] [36]		×	×
		BANG File [13]		×	×
		BV-Tree [14]		×	×

Tabella 5.2: Point Access Methods suddivisi per livello di memoria utilizzata [56].

		Tipo di dato spaziale			
		Griglia	Scatola	Sfera	Poliedro
Tecniche di modifica del PAM	Trasformazione	BANG File [13] hB-Tree [35][36] zkdB$^+$-Tree [42]	k-d-B-Tree [44] LSD-Tree [27] BV-Tree [14] Buddy Tree [52]		P-Tree [29]
	Sopvrapposizione		R-tree [21] R*-Tree [3] skd-Tree [40] GBD-Tree [39] Hilbert R-Tree [31] Buddy Tree con overlapping [50]	sphere-Tree [57]	P-Tree [49] KD2B-Tree [57]
	Ritaglio		extended k-d-Tree [37] R$^+$-Tree [53] Buddy Tree con clipping [30]		cell Tree [20]

Tabella 5.3: Spatial Access Methods [23].

[1] Vengono barrati i metodi d'accesso che gestiscono le regioni sottoforma di scatola rettangolare.

[2] Lo spazio universo è coperto interamente.

[3] Metodi d'accesso che non permettono la sovrapposizione di regioni diverse.

Nella Tabella 5.1 vi è una schematica suddivisione secondo il tipo di base di dati su cui lavorano le strutture dati gerarchiche. La Tabella 5.2, invece, riassume la tassonomia dei PAM proposta da Seeger e Kriegel [56], che li classificano mediante le proprietà condivise dai bucket all'interno dell'albero: le regioni (1) possono avere figura rettangolare oppure essere rappresentate mediante una arbitraria forma di poliedro minimo, (2) possono coprire l'intero spazio universo o solamente una parte di questo e, infine, (3) possono essere soggette a sovrapposizioni o mutualmente separate. La Tabella 5.3 raggruppa i SAM secondo le tecniche di modifica dei PAM viste nel Capitolo 4 e aggiunge un'ulteriore dimensione alla classificazione [23], specificando il tipo di dato spaziale principalmente supportato dai vari metodi d'accesso.

		Operazione			
		Ricerca		Inserimento	Cancellazione
		Casuale	Sequenziale		
Strutture dati	B-Tree [1] [9]	O(log n)	O(n)	$O(d \times \log_d n)$	$O(d \times \log_d n)$
	B^+-Tree [9]	$O(\log_d n)$	$O(\log_d n)$	$O(\log_d n)$	$O(\log_d n)$
	k-d-B-Tree [44] [18]	$O(n^{1-1/d}+k)$*	-	O(log n)	O(log n)
	R-tree [21]	$O(\log_m n)$**	-	Costo fortemente legato al numero di regioni sovrapposte	

* Range Query
** Point Query e Window Query

Legenda:
- n Chiavi memorizate nell'albero
- d Grado dell'albero/dimensione dello spazio
- k Punti consistenti con la query di ricerca
- m Limite inferiore di entry in un nodo

Tabella 5.4: Tempo di esecuzione delle strutture dati viste nei Capitoli 3 e 4.

Le strutture gerarchiche trattate in dettaglio durante la rassegna, si differenziano sia per il campo di applicazione, sia in termini di prestazioni e tempo di esecuzione degli algoritmi relativi alle principali funzioni da loro offerte. La Tabella 5.4 riassume i costi di queste operazioni. Si ricorda che per quanto riguarda i SAM, il tempo d'inserimento e cancellazione di un oggetto spaziale, dipende fortemente dalla quantità di sovrapposizioni delle regioni a cui è soggetta la struttura.

In conclusione, l'efficienza dei metodi d'accesso alle basi di dati tradizionali, è rappresentata dall'utilizzo di strutture B^+-Tree. Questa è la tecnica gerarchica unidimensionale che viene implementata da tutti i principali DBMS in commercio, tra cui PostgreSQL, MySQL, Microsoft SQL Server e Oracle. Il supporto alla gestione di oggetti spaziali è fornita, invece, attraverso diversi criteri e metodologie organizzative dei dati: Microsoft utilizza i principi dei B-Tree per mappare gli oggetti multidimensionali in griglie gerarchiche di sottospazi; MySQL e Oracle Spatial implementano la struttura degli R-Tree descritta nel Capitolo 4, mentre PostgreSQL, come visto, include un supporto nativo per la definizione di alberi GiST, utilizzabili altresì per la costruzione di metodi d'accesso gerarchici a più dimensioni.

Bibliografia

[1] ANSI/X3/SPARC. Interim report: Ansi/x3/sparc study group on data base management systems 75-02-08. *FDT - Bulletin of ACM SIGMOD 7*, 2 (1975), 1–140.

[2] Bayer, R., and McCreight, E. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control* (New York, NY, USA, 1970), SIGFIDET '70, ACM, pp. 107–141.

[3] Bayer, R., and Unterauer, K. Prefix b-trees. *ACM Trans. Database Syst. 2* (March 1977), 11–26.

[4] Beckmann, N., Kriegel, H.-P., Schneider, R., and Seeger, B. The r*-tree: an efficient and robust access method for points and rectangles. *SIGMOD Rec. 19* (May 1990), 322–331.

[5] Bentley, J. L. Multidimensional binary search trees used for associative searching. *Commun. ACM 18* (September 1975), 509–517.

[6] Bentley, J. L. Multidimensional binary search trees in database applications. *IEEE Trans. Softw. Eng. 5* (July 1979), 333–340.

[7] Bentley, J. L., and Friedman, J. H. Data structures for range searching. *ACM Comput. Surv. 11* (December 1979), 397–409.

[8] Chen, P. P.-S. *The entity-relationship model: toward a unified view of data*. Springer-Verlag New York, Inc., New York, NY, USA, 2002, pp. 311–339.

[9] Codd, E. F. A relational model of data for large shared data banks. *Commun. ACM 13* (June 1970), 377–387.

[10] Comer, D. Ubiquitous b-tree. *ACM Comput. Surv. 11* (June 1979), 121–137.

[11] Fagin, R., Nievergelt, J., Pippenger, N., and Strong, H. R. Extendible hashing: a fast access method for dynamic files. *ACM Trans. Database Syst. 4* (September 1979), 315–344.

[12] Ferragina, P., and Grossi, R. The string b-tree: a new data structure for string search in external memory and its applications. *J. ACM 46* (March 1999), 236–280.

[13] Foster, C. C. Information retrieval: information storage and retrieval using avl trees. In *Proceedings of the 1965 20th national conference* (New York, NY, USA, 1965), ACM '65, ACM, pp. 192–205.

[14] Freeston, M. The bang file: A new kind of grid file. *SIGMOD Rec. 16* (December 1987), 260–269.

[15] Freeston, M. A general solution of the n-dimensional b-tree problem. *SIGMOD Rec. 24* (May 1995), 80–91.

[16] Fuchs, H., Abram, G. D., and Grant, E. D. Near real-time shaded display of rigid objects. *SIGGRAPH Comput. Graph. 17* (July 1983), 65–72.

[17] Fuchs, H., Kedem, Z. M., and Naylor, B. F. On visible surface generation by a priori tree structures. *SIGGRAPH Comput. Graph. 14* (July 1980), 124–133.

[18] Gaede, V. Optimal redundancy in spatial database systems. In *Proceedings of the 4th International Symposium on Advances in Spatial Databases* (London, UK, 1995), SSD '95, Springer-Verlag, pp. 96–116.

[19] Gaede, V., and Günther, O. Multidimensional access methods. *ACM Comput. Surv. 30* (June 1998), 170–231.

[20] Gaede, V., and Riekert, W.-F. Spatial access methods and query processing in the object-oriented gis godot. In *IN PROC. OF THE AGDM'94 WORKSHOP* (1994), pp. 40–52.

[21] Günther, O. *Efficient structures for geometric data management*. Springer-Verlag New York, Inc., New York, NY, USA, 1988.

[22] Guttman, A. R-trees: a dynamic index structure for spatial searching. *SIGMOD Rec. 14* (June 1984), 47–57.

[23] H P Kriegel, P Heep, S. H. M. S. R. S. An access method based query processor for spatial database systems. In *Geographic Database Management Systems* (New York, NY, USA, 1991), Springer-Verlag New York, Inc., pp. 273–292.

[24] Hammer, M., and McLeod, D. The semantic data model: a modelling mechanism for data base applications. In *Proceedings of the 1978 ACM SIGMOD international conference on management of data* (New York, NY, USA, 1978), SIGMOD '78, ACM, pp. 26–36.

[25] Hellerstein, J. M., Naughton, J. F., and Pfeffer, A. Generalized search trees for database systems. In *Proceedings of the 21th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1995), VLDB '95, Morgan Kaufmann Publishers Inc., pp. 562–573.

[26] Hellerstein, J. M., Naughton, J. F., and Pfeffer, A. Generalized search trees for database systems. In *Readings in database systems (3rd ed.)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998, pp. 101–112.

[27] Hellerstein, J. M., Stonebraker, M., and Hamilton, J. *Architecture of a Database System*. Now Publishers Inc., Hanover, MA, USA, 2007.

[28] Henrich, A., Six, H. W., and Widmayer, P. The lsd tree: spatial access to multidimensional and non-point objects. In *Proceedings of the 15th international conference on Very large data bases* (San Francisco,

CA, USA, 1989), VLDB '89, Morgan Kaufmann Publishers Inc., pp. 45–53.

[29] Hoel, E. G., and Samet, H. Benchmarking spatial join operations with spatial output. In *Proceedings of the 21th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1995), VLDB '95, Morgan Kaufmann Publishers Inc., pp. 606–618.

[30] Jagadish, H. V. Linear clustering of objects with multiple attributes. *SIGMOD Rec. 19* (May 1990), 332–342.

[31] Kakde, H. M. Range searching using kd tree, 2005.

[32] Kamel, I., and Faloutsos, C. Hilbert r-tree: An improved r-tree using fractals. In *Proceedings of the 20th international conference on Very large data bases* (1994), pp. 500–509.

[33] Karlton, P. L., Fuller, S. H., Scroggs, R. E., and Kaehler, E. B. Performance of height-balanced trees. *Commun. ACM 19* (January 1976), 23–28.

[34] Kim, W. *Introduction to object-oriented databases.* MIT Press, Cambridge, MA, USA, 1990.

[35] Litwin, W. Linear hashing: a new tool for file and table addressing. In *Proceedings of the sixth international conference on Very Large Data Bases - Volume 6* (1980), VLDB Endowment, pp. 212–223.

[36] Lomet, D., and Salzberg, B. A robust multi-attribute search structure. In *Data Engineering, 1989. Proceedings. Fifth International Conference on* (feb 1989), pp. 296 –304.

[37] Lomet, D. B., and Salzberg, B. The hb-tree: a multiattribute indexing method with good guaranteed performance. *ACM Trans. Database Syst. 15* (December 1990), 625–658.

[38] Matsuyama, T., Hao, L. V., and Nagao, M. A file organization for geographic information systems based on spatial proximity. *Computer Vision, Graphics, and Image Processing* (1984), 303–318.

[39] Ohsawa, Y., and Sakauchi, M. The bd-tree - a new n-dimensional data structure with highly efficient dynamic characteristics. In *IFIP Congress* (1983), pp. 539–544.

[40] Ohsawa, Y., and Sakauchi, M. A new tree type data structure with homogeneous nodes suitable for a very large spatial database. In *Proceedings of the Sixth International Conference on Data Engineering* (Washington, DC, USA, 1990), IEEE Computer Society, pp. 296–303.

[41] Oleg Bartunov, T. S. Analyzing gist: R-tree index page.

[42] Oleg Bartunov, T. S. Gevel wiki page.

[43] Oleg Bartunov, T. S. Gist for postgresql home page.

[44] Ooi, Mcdonell, K. J., and Davis, S. R. Spatial kd-tree: An Indexing Mechanism for Spatial Database. *COMPSAC conf.* (1987), 433–438.

[45] Ooi, B. C., and Tan, K.-L. B-trees: bearing fruits of all kinds. *Aust. Comput. Sci. Commun. 24* (January 2002), 13–20.

[46] Orenstein, J. A. Spatial query processing in an object-oriented database system. *SIGMOD Rec. 15* (June 1986), 326–336.

[47] Orenstein, J. A., and Merrett, T. H. A class of data structures for associative searching. In *Proceedings of the 3rd ACM SIGACT-SIGMOD symposium on Principles of database systems* (New York, NY, USA, 1984), PODS '84, ACM, pp. 181–190.

[48] Robinson, J. T. The k-d-b-tree: a search structure for large multidimensional dynamic indexes. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 1981), SIGMOD '81, ACM, pp. 10–18.

[49] Roussopoulos, N., and Leifker, D. Direct spatial search on pictorial databases using packed r-trees. *SIGMOD Rec. 14* (May 1985), 17–31.

[50] Samet, H. The quadtree and related hierarchical data structures. *ACM Comput. Surv. 16* (June 1984), 187–260.

[51] Samet, H. *Applications of spatial data structures: Computer graphics, image processing, and GIS*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[52] Samet, H. *The design and analysis of spatial data structures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[53] Schiwietz, M. *Speicherung und Anfragebearbeitung komplexer Geo-Objekte*. PhD thesis, Ludwig-Maximilian-Universität München, 1993.

[54] Seeger, B. Performance comparison of segment access methods implemented on top of the buddy-tree. In *Proceedings of the Second International Symposium on Advances in Spatial Databases* (London, UK, 1991), SSD '91, Springer-Verlag, pp. 277–296.

[55] Seeger, B., and Kriegel, H.-P. Techniques for design and implementation of efficient spatial access methods. In *Proceedings of the 14th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1988), VLDB '88, Morgan Kaufmann Publishers Inc., pp. 360–371.

[56] Seeger, B., and Kriegel, H.-P. The buddy-tree: An efficient and robust access method for spatial data base systems. In *Proceedings of the 16th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1990), VLDB '90, Morgan Kaufmann Publishers Inc., pp. 590–601.

[57] Sellis, T. K., Roussopoulos, N., and Faloutsos, C. The r+-tree: A dynamic index for multi-dimensional objects. In *Proceedings of the 13th International Conference on Very Large Data Bases* (San

Francisco, CA, USA, 1987), VLDB '87, Morgan Kaufmann Publishers Inc., pp. 507–518.

[58] Song, I.-Y., and Froehlich, K. Entity-relationship modeling. *Potentials, IEEE 13*, 5 (1994/jan 1994), 29 –34.

[59] Stair, R., and Reynolds, G. *Principles of Information Systems (with Printed Access Card)*, 10th ed. Course Technology Press, Boston, MA, United States, 2011.

[60] Tamminen, M. Comment on quad- and octtrees. *Commun. ACM 27* (March 1984), 248–249.

[61] van Oosterom, P. *Reactive data structures for geographic information systems*. Oxford University Press, Inc., New York, NY, USA, 1994.

[62] Watson, H. J., and Wixom, B. H. The current state of business intelligence. *Computer 40* (September 2007), 96–99.

Ringraziamenti

Ringrazio in primo luogo il Prof. Moreno Marzolla, che mi ha seguito nella gratificante esperienza verso la laurea triennale, in maniera costante e con grande professionalità e disponibilità. Un legittimo ringraziamento è dovuto anche a tutti gli altri docenti del corso di studi in Informatica per il Management, che, in questi tre anni di percorso accademico, mi hanno permesso di maturare sotto il profilo culturale, sociale e relazionale. In particolare, voglio precisare quanto sia stato importante trovare delle persone serie e diligenti come i miei "compagni di viaggio", nonché colleghi laureandi, Carlo e Fady; un doveroso ringraziamento anche a loro per il sostegno reciproco e i singolari momenti da studenti universitari vissuti insieme.

Il più forte e profondo grazie, però, va alla mia Famiglia e soprattutto ai miei genitori Renata e Maurizio, i quali hanno reso possibile con tanti sforzi questo difficile e oneroso percorso di studi. Chi in un modo o chi nell'altro, avete sempre creduto in me e sono orgoglioso di poter dedicare a voi, oltre che alla mia crescita personale, questo importante momento.

Come giusto ringraziamento alla persona che mi ha incoraggiato, sostenuto e sopportato con grande pazienza durante questi intensi mesi di lavoro, voglio dedicare queste righe conclusive a Lei. Grazie di cuore Lori.