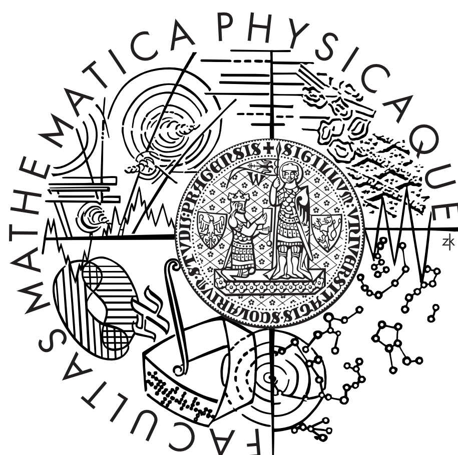
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Martina Krejčová

Index pro podobnostní vyhledávání ve vysokodimenzionálních prostorech

Katedra softwarového inženýrství

Vedoucí diplomové práce:	RNDr. Michal Kopecký, Ph.D.
Studijní program:	Informatika
Studijní obor:	Softwarové inženýrství

Praha 2012

Ráda bych poděkovala svému vedoucímu RNDr. Michalu Kopeckému, Ph.D. za vedení diplomové práce a čas strávený při konzultacích.
Také děkuji rodině za podporu při studiu a trpělivost, kterou se mnou měli.

Prohlašuji, že jsem tuto diplomovou práci vypracovala samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V .... dne .... Podpis autora

Název práce: Index pro podobnostní vyhledávání ve vysokodimenzionálních prostorech

Autor: Martina Krejčová

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Michal Kopecký, Ph.D., Katedra softwarového inženýrství

Abstrakt: V této práci se zabýváme indexováním a vyhledáváním vysokodimenzionálních dat pomocí metody Metrického indexu pro indexování a podobnostní vyhledávání v metrických prostorech. Použití této metody nám umožnilo vytvořit implementaci indexu vhodného pro indexaci obecných metrických prostorů. Díky tomuto indexu je kromě ukládání dat umožněno i jejich efektivní vyhledávání. Vnitřní struktura dat indexu zůstává skryta, index od uživatele vyžaduje pouze definici extrakční funkce pro získání vektoru, který data reprezentuje, a podobnostní funkce, která má být na indexovaná data aplikována.
V této práci vznikla implementace Metrického indexu jako data cartridge pro databázový server Oracle. Tato data cartridge rozšiřuje možnosti indexace v Oracle o vytváření doménových indexů nad nestrukturovanými daty, takzvanými LOBy.

Klíčová slova: podobnostní vyhledávání, metrický prostor, index, data cartridge

Title: Index Suitable for Similar Search in High-dimensional Spaces

Author: Martina Krejčová

Department: Department of Software Engineering

Supervisor: RNDr. Michal Kopecký, Ph.D., Department of Software Engineering

Abstract: In this paper, we focus on indexing and searching in high-dimensional data. To achieve the target we implemented the Metric Index, a model of the similarity search based on the metric spaces, that employs many of known principles of partitioning and filtering. The metric space is a general model of similarity, which enables the usage of implemented index for various data. With this index, stored data could be searched effectively. The internal structure of data is hidden, we just require an implementation of the function for feature extraction, which produces a vector representing data, and the metric function applicable to the given data.
The Metric Index was implemented as a data cartridge, the mechanism for extending the capabilities of the Oracle server. This data cartridge enables indexing of large unstructured data in the Oracle server known as LOBs.

Keywords: similarity search, metric space, index, data cartridge

Obsah

Úvod **4**

1 Vyhledávání v metrických prostorech **6**
- 1.1 Metrický prostor . . . 6
- 1.2 Dotazování v metrických prostorech . . . 6
- 1.3 Shlukování . . . 6
  - 1.3.1 Typy shluků . . . 7
- 1.4 Strategie vyhnutí se počítání vzdáleností . . . 7
  - 1.4.1 Object-pivot distance . . . 7
  - 1.4.2 Double-pivot distance . . . 8
  - 1.4.3 Range-pivot distance . . . 8

2 Metrický Index **10**
- 2.1 iDistance . . . 10
- 2.2 M-Index s jednou úrovní . . . 10
- 2.3 M-Index s více úrovněmi . . . 10
- 2.4 M-Index s dynamickým počtem úrovní . . . 13
  - 2.4.1 Strom shluků . . . 14
- 2.5 Volba pivotů . . . 14
  - 2.5.1 Kritérium efektivnosti . . . 14
  - 2.5.2 Incremental selection . . . 16

3 Vyhledávání v M-Indexu **17**
- 3.1 Hledání sousedů do určité vzdálenosti . . . 17
- 3.2 Hledání k nejbližších sousedů . . . 17

4 Design **21**
- 4.1 Použitý databázový server . . . 21
- 4.2 Extensible Indexing v Oracle . . . 21
  - 4.2.1 Interface ODCIIndex . . . 22
  - 4.2.2 Operátory . . . 23
- 4.3 Extensible Optimizer v Oracle . . . 25

5 Požadavky na uživatele **26**
- 5.1 Podporované datové typy . . . 26
- 5.2 Typ TChar . . . 26
- 5.3 Objektový typ mspace . . . 26
  - 5.3.1 Definice mspace . . . 27
  - 5.3.2 Příklad definice potomka mspace . . . 27

6 Implementace **29**
- 6.1 Uložení dat indexu . . . 29
  - 6.1.1 Strom shluků . . . 29
  - 6.1.2 Data ve shluku . . . 30
- 6.2 Metadata indexu . . . 30
  - 6.2.1 Tabulka mindex_data . . . 31

6.2.2 Tabulka _idx 31
6.3 Indexový typ mindex_type 31
6.3.1 ODCIIndexCreate 31
6.3.2 ODCIIndexDrop 32
6.3.3 ODCIIndexInsert 32
6.3.4 ODCIIndexDelete 33
6.3.5 ODCIIndexStart 33
6.3.6 ODCIIndexFetch 34
6.3.7 ODCIIndexClose 34

7 Existující řešení **35**
7.1 VPT 35
7.1.1 Hledání sousedů do určité vzdálenosti $R(q, r)$ 35
7.2 AESA 36
7.2.1 Hledání nejbližšího souseda $NN(q)$ 37
7.2.2 Hledání sousedů do určité vzdálenosti $R(q, r)$ 37
7.2.3 Hledání k nejbližších sousedů $kNN(q, k)$ 38
7.3 LAESA 38
7.3.1 Hledání sousedů do určité vzdálenosti $R(q, r)$ 38
7.4 GNAT 39
7.4.1 Hledání sousedů do určité vzdálenosti $R(q, r)$ 39
7.5 MESSIF 40
7.5.1 Metrické prostory 41
7.5.2 Kolekce a dotazy 41
7.5.3 Správa dat 41
7.5.4 Distribuované datové struktury 42
7.5.5 Podobnostní vyhledávání přes více vlastností 43
7.5.6 Uživatelské rozhraní 43

8 Měření výkonu **44**
8.1 Použitý hardware 44
8.2 Jednotka složitosti operací 44
8.3 Použitá data 44
8.3.1 Normální rozdělení 45
8.3.2 Upravené rovnoměrné rozdělení 45
8.4 Vytvoření indexu 45
8.4.1 Nalezení pivotů 46
8.4.2 Vložení všech prvků 46
8.5 Vyhledávání 47
8.5.1 Dotaz R(q,r) 47
8.5.2 Dotaz kNN(q,k) 49
8.5.3 Vliv počtu pivotů na efektivitu vyhledávání 49
8.6 Modifikace kolekce dat 54
8.6.1 Vkládání 54
8.6.2 Mazání 54

Závěr **58**

Seznam použité literatury **59**

Přílohy **61**

9 Uživatelská příručka **62**
9.1 Vytvoření indexu 62
9.2 Vkládání a mazání dat 63
9.3 Vyhledávání 63
9.4 Rušení indexu 64

Úvod

Informační technologie jsou používány v mnoha oborech lidského konání. S tím jde ruku v ruce i potřeba ukládat data a efektivně v nich vyhledávat. Asi nejběžnějším způsobem uchovávání dat je dnes jejich uložení do relační databáze. Jejich rozdělení do řádek - sad jednoduchých číselných či textových atributů a následné vložení do tabulky či tabulek v databázi.
Uživatel může takto uložená data později nalézt kladením dotazů. Nalezení odpovědi na dotaz pak nejčastěji vyžaduje nalezení řádků s obsahy atributů rovnými požadované hodnotě nebo patřící do požadovaného rozsahu.
Ne ve všech datech lze ale jednoduše najít strukturu. Řada úloh vyžaduje práci s nestrukturovanými, nebo jen semi-strukturovanými daty. Například multimediální data (obrázky, filmy a podobně) bychom jen těžko ukládali v podobě sady jednoduchých atributů.
Někdy nás také mohou zajímat odpovědi na dotazy kladené jiným způsobem. Mohli bychom se ptát, jaká data jsou nejvíce podobná nějakému vzoru. Takovémuto vyhledávání se říká vyhledávání podobnostní. Příkladem by mohlo být vyhledávání otisků prstů. Je velmi nepravděpodobné, že by se nalezly dva úplně stejné obrázky otisků prstů, a to i u jednoho člověka. Na každém obrázku budou papilární linie zdeformované jiným způsobem. V tuto chvíli bychom se asi rádi zeptali právě na otisky, které jsou nejvíce podobné našemu vzoru.
Aby se dalo v nestrukturovaných datech vyhledávat, je potřeba z nich vyextrahovat použitelná strukturovaná data nebo je ručně či strojově oanotovat. Z obrázků se dá extrahovat histogram barev, detekovat hrany a vytvářet obrysy. K filmům může být přiřazen autor, herecké obsazení a další vlastnosti. Ze zvukové stopy lze zrekonstruovat text. U otisků prstů mohou být hledány křížení a větvení papilárních linií, takzvané markanty, a měřeny vzdálenosti a úhlové odchylky mezi spojnicemi.
Výsledkem extrakce strukturovaných dat zpravidla bývá velké množství hodnot. Každý prvek je tak reprezentován bodem v mnohadimenzionálním prostoru. Dvojice vzniklých strukturovaných dat lze již pomocí nějaké podobnostní funkce porovnávat.
Podobnostní vyhledávání je implementovatelné spočítáním podobnosti zadaného vzoru se všemi prvky kolekce. Tento postup však vyžaduje velké množství vyhodnocování podobnostní funkce. Vzhledem k mnohadimenzionalitě dat však může být tato podobnostní funkce drahá na vyčíslení a spočtení všech podobností se neúměrně protahuje. Právě to bylo motivací k vývoji indexačních a vyhledávacích metod v metrických prostorech. Ty se snaží na základě vlastností prvků, některé z vyhodnocování vyřadit, nebo naopak rovnou zahrnout do výsledku, aniž by bylo nutné podobnost těchto prvků se vzorem počítat, a tím minimalizovat počet vyhodnocování podobnostní funkce.

V této práci se budeme zabývat indexováním vysokodimenzionálních dat, které nám umožní jejich efektivní podobnostní vyhledávání. Jako model dat jsme použili právě metrických prostorů, které nabízí efektivní metody pro indexování a podobnostní vyhledávání v mnohadimenzionálních doménách. Výhodou tohoto modelu je také jeho obecnost a z toho vyplývající použitelnost pro mnoho růz-

ných datových typů.
První kapitoly této práce seznamují s teoretickým základem práce. Nejdříve přiblížíme pojem metrického prostoru a jeho vlastnosti. Ve druhé a třetí kapitole se zaměříme na metodu indexování v metrických prostorech pomocí *Metrického indexu* a na vyhledávání pomocí tohoto indexu.
Následují kapitoly věnující se způsobu implementace vlastních doménových indexů pro databázový server Oracle s popisem vlastní implementace *Metrického indexu.*
Následuje kapitola věnující se dalším existujícím řešením indexování metrických prostorů a srovnání s Metrickým indexem, kapitola popisující testování naší implementace a kapitola s uživatelskou příručkou.

1. Vyhledávání v metrických prostorech

V této kapitole uvedeme definici metrického prostoru, nejběžnější typy dotazů v metrických prostorech a pravidla použitelná pro odhadování vzdálenosti dvou objektů.

1.1 Metrický prostor

Definice 1. *Metrika na množině $\mathcal{D}$ ($\mathcal{D} \neq \emptyset$) je zobrazení $d : \mathcal{D} \times \mathcal{D} \rightarrow [0; \infty)$ s vlastnostmi :*

1. $\forall x, y \in \mathcal{D}, d(x, y) = 0 \iff x = y$
2. $\forall x, y \in \mathcal{D}, d(x, y) = d(y, x)$
3. $\forall x, y, z \in \mathcal{D}, d(x, z) \leq d(x, y) + d(y, z)$

Definice 2. *Metrický prostor $\mathcal{M}$ je uspořádaná dvojice $\mathcal{M} = (\mathcal{D}, d)$, kde $\mathcal{D}$ je množina prvků a d je metrika na množině $\mathcal{D}$.*

Příkladem metrického prostoru $\mathcal{M} = (\mathcal{D}, d)$ je $\mathcal{D} = \mathbb{R}^2$ s metrikou $d(x, y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$.

1.2 Dotazování v metrických prostorech

V metrických prostorech jako modelu podobnosti dat jsou data typicky vyhledávána pomocí dotazování příkladem. Dotaz se skládá z příkladu objektu $q \in \mathcal{D}$ a nějakého omezení na vracená data z množiny dat $X \subseteq \mathcal{D}$. Hlavními typy dotazů jsou :

1. *range* $R(q, r)$, který vrací všechny objekty $o \in X$, které jsou od q vzdálené nejvýše r, tedy $o \in X, d(q, o) \leq r$.
2. *k nearest neighbours* $kNN(q, k)$, který vrací množinu S k objektů $o \in X$ s nejmenší vzdáleností od q, tedy takovou, že pro všechny $o' \in X - S$, $d(q, o) \leq d(q, o')$.

1.3 Shlukování

Pro zúžení procházeného prostoru při podobnostním vyhledávání se využívá shlukování [6]. Shlukování spočívá ve sdružování podobných prvků do skupin. Vzniklým skupinám se říká právě shluky či klastry.
Při podobnostním vyhledávání jsou pak navštíveny jen shluky, které obsahují prvky podobné vzoru z dotazu.
Shluk si můžeme například představit jako m-rozměrnou kouli určenou středem a poloměrem.

Často jsou používány hierarchické metody shlukování, kde jsou jednotlivé shluky obsahující velmi podobné prvky sdružovány do shluků již si navzájem méně podobných prvků, které mohou být opět sdružovány.

1.3.1 Typy shluků

Shluky mohou být různých typů

- disjunktní a
- nedisjunktní,

podle toho, zda jeden objekt může patřit zároveň do více shluků či ne.
Shluky mohou být konstruovány

- se stejnou velikostí, pokrývající stejně velkou část prostoru
- nebo s přibližně stejným počtem prvků.

My budeme dále používat disjunktní shluky vytvořené hierarchickou metodou shlukování. Ale budeme postupovat shora a dělit prostor na shluky a shluky, jejichž velikost přeroste nějakou danou hraniční hodnotu, budeme dále dělit. Jelikož prostor obecně nelze pokrýt disjunktními koulemi, budeme používat rozdělení na konvexní mnohostěny, kde každý prvek připadá k nejbližšímu pivotu ze sady předem vybraných pivotů.

1.4 Strategie vyhnutí se počítání vzdáleností

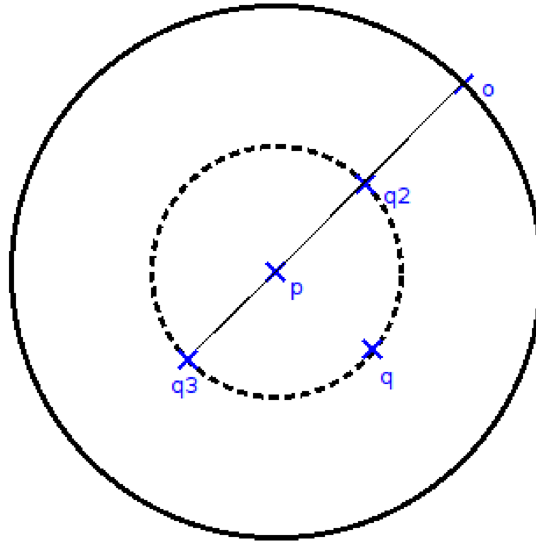
Aby byly algoritmy vyhledávání co nejefektivnější, jsou používána pravidla, která umožní odhadnout vzdálenost dvou objektů [3]. Tato pravidla umožní prořezat části prostoru tak, aby se neprocházely ty části, které nemohou obsahovat prvky z výsledku. Odhad vzdálenosti také umožní některé objekty vyloučit z výsledku, nebo je do výsledku zařadit, aniž by bylo potřeba počítat vzdálenost těchto objektů od vzorového.

1.4.1 Object-pivot distance

Věta 1. *Je-li $\mathcal{M} = (\mathcal{D}, d)$ metrický prostor a $q, p, o \in \mathcal{D}$ tři různé objekty platí, že*

$$|d(q,p) - d(p,o)| \leq d(q,o) \leq d(q,p) + d(p,o).$$

Známe-li vzdálenosti $d(q,p)$ dotazu q od daného pivota p a vzdálenost $d(p,o)$ tohoto objektu o, můžeme shora i zdola odhadnout vzdálenost $d(q,o)$ objektu od dotazu a případně se tak vyhnout jejímu výpočtu.
Pravidlo je ilustrováno v obrázku 1.1. Čerchovaná kružnice znázorňuje všechny prvky q se zadanou vzdáleností od p, $d(p,q)$, s extrémními případy pozic $q2$ a $q3$ ležícími nejblíže k bodu o, respektive nejdále od něj.



Obrázek 1.1: object-pivot distance

1.4.2 Double-pivot distance

Věta 2. *Je-li* $\mathcal{M} = (\mathcal{D}, d)$ *metrický prostor,* $q, p1, p2, o \in \mathcal{D}$ *a platí, že* $d(o, p1) \leq d(o, p2)$*, pak*

$$max\left\{\frac{d(q,p1) - d(q,p2)}{2}, 0\right\} \leq d(q,o).$$

Známe-li vzdálenosti $d(q, p1)$, $d(q, p2)$ dotazu od dvou pivotů a víme-li, že pro posuzovaný objekt platí $d(o, p1) \leq d(o, p2)$, pak můžeme zdola odhadnout vzdálenost $d(q, o)$ objektu od dotazu a opět se tak případně vyhnout jejímu výpočtu. Objekt o je podle zadání blíže k $p1$.

- Pokud je i dotaz q blíže k prvku $p1$, pak dolní hranicí vzdálenosti bude 0, což pro účely vyhledávání příliš nepomůže.

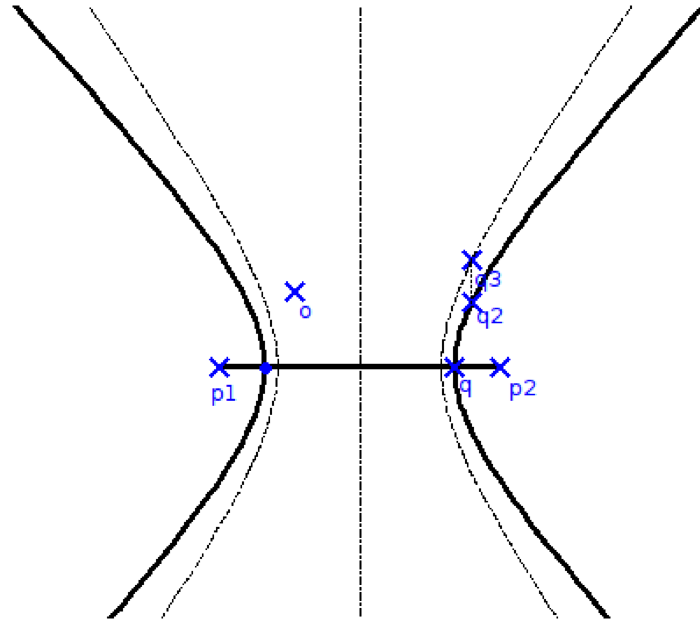
- Obrázek 1.2 znázorňuje situaci, kdy objekt q je naopak blíže k $p2$. Hyperbola znázorňuje všechny body q, které mají stejnou hodnotu $d(q, p1) - d(q, p2)$, posunem bodu q vertikálně dále od úsečky $p1p2$, se tato hodnota zmenší. Vzdálenost $d(q, o)$ musí být větší než vzdálenost q od osy úsečky $p1p2$ a tedy větší než $\frac{d(q,p1)-d(q,p2)}{2}$.

1.4.3 Range-pivot distance

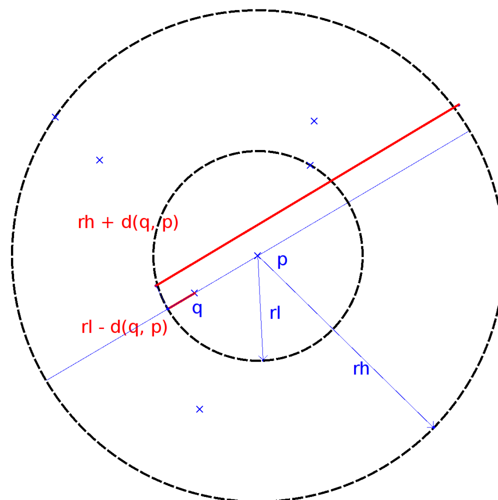
Věta 3. *Je-li* $\mathcal{M} = (\mathcal{D}, d)$ *metrický prostor,* $q, p, o \in \mathcal{D}$ *a platí, že* $r_l \leq d(o, p) \leq r_h$*, pak*

$$max\{d(q,p) - r_h, r_l - d(q,p), 0\} \leq d(q,o) \leq d(q,p) + r_h.$$

Známe-li vzdálenost $d(q, p)$ vzoru z dotazu q od daného pivota p, dolní hranici r_l a horní hranici r_h vzdálenosti $d(p, o)$ tohoto pivota p od objektu o, můžeme shora i zdola odhadnout vzdálenost objektu o od vzoru q.
Tento odhad vzdálenosti je pro jednu z možných pozic vzorového prvku q znázorněn červenou barvou v obrázku 1.3.



Obrázek 1.2: double-pivot distance



Obrázek 1.3: range-pivot distance

2. Metrický Index

V této kapitole popíšeme strukturu M-Indexu [1], který jsme implementovali a použili k indexování vysokorozměrných dat.
Indexovaná data jsou z nějaké domény D, indexovanou množinu zde budeme nazývat X, $X \subseteq D$.
Začneme popisem iDistance, indexu pro vyhledávání ve vektorových prostorech. M-Index je zobecněním tohoto indexu, vhodným pro libovolné metrické prostory. Od iDistance tak přejdeme k M-Indexu s jednou úrovní. Následuje popis M-Indexu s více úrovněmi, včetně způsobu jakým je prostor rozdělován na jednotlivé shluky a tyto shluky na další. K M-indexu s více úrovněmi je uvedena i varianta s dynamickým počtem úrovní.

2.1 iDistance

Máme-li množinu dat $X \subseteq D$, iDistance [2] rozdělí X do nějakého počtu n shluků a ke každému shluku C_i, $i \in \{0, ..., n-1\}$ určí referenční prvek p_i. Každému prvku $o \in C_i \subseteq X$ je určen klíč $iDist(o)$

$$iDist(o) = d(p_i, o) + i \cdot c, \tag{2.1}$$

kde c je konstanta dostatečně veliká, aby oddělila klíče prvků z různých klastrů. Klíče všech prvků z klastru C_i jsou pak z intervalu $[i \cdot c, (i+1) \cdot c)$ a tyto intervaly jsou disjunktní. Přiřazení $iDist$ klíčů je znázorněno v obrázku 2.1. Celý prostor je tak pomocí iDist promítnut do prostoru s jedinou dimenzí. Prvky tak mohou být podle jejich $iDist$ klíče ukládány do B^+stromu.

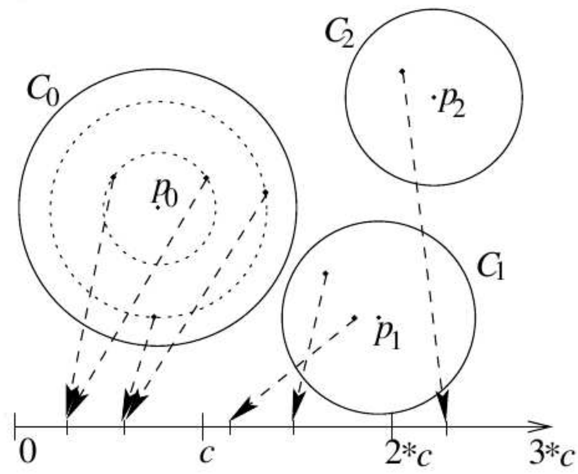
2.2 M-Index s jednou úrovní

M-Index zvolí množinu n pivotů $p_0, p_1, ..., p_{n-1}$ z předem dané množiny dat $X \subseteq \mathcal{D}$. Pomocí těchto pivotů je metrický prostor rozdělen, ke každému pivotu p_i náleží ty prvky, které mají k pivotu p_i menší vzdálenost než k jinému pivotu z množiny pivotů. Částem na které je prostor rozdělen budeme dále říkat shluky. Množinu prvků náležející k pivotu p_i budeme označovat jako shluk C_i.
Příklad rozdělení dvourozměrného prostoru se čtyřmi pivoty je znázorněn na obrázku 2.2.
Přiřazování klíčů k prvkům probíhá obdobně jako u iDistance. Díky požadavku M-Indexu na funkci vzdálenosti, aby nabývala hodnot jen z intervalu $[0, 1)$, zde konstanta c není potřeba, respektive může být vždy použita hodnota $c = 1$.

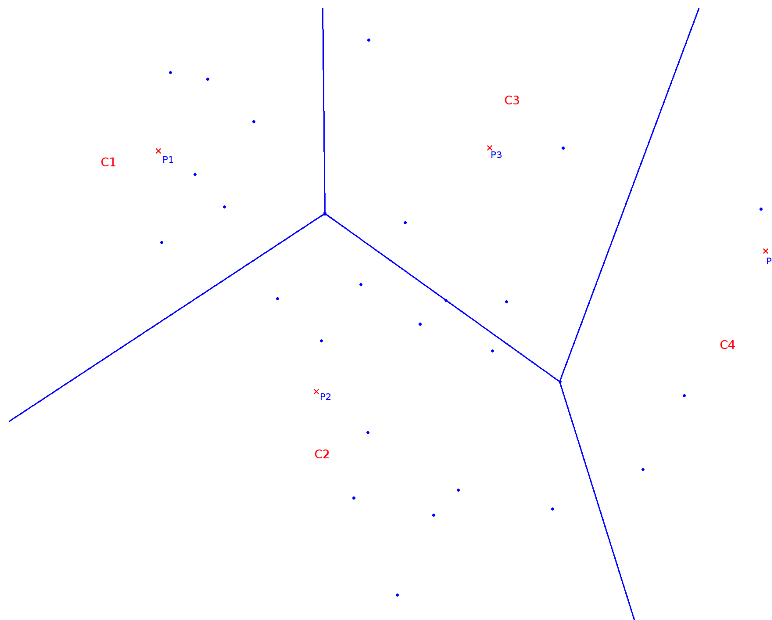
$$key(o) = d(p_i, o) + i, \tag{2.2}$$

2.3 M-Index s více úrovněmi

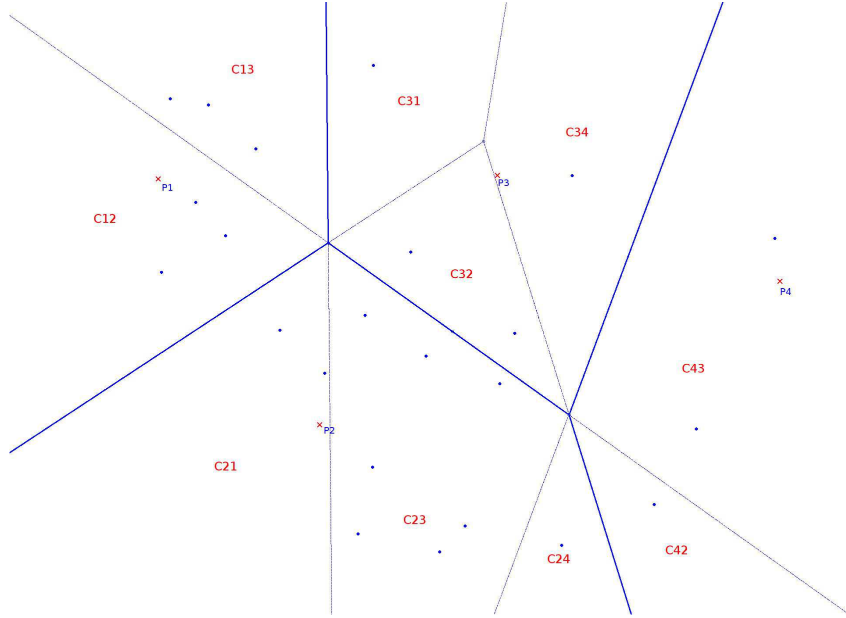
Další dělení shluku C_i proběhne stejně jako dělení celého prostoru, tentokrát ale pomocí zbývajících pivotů, tedy pivotů p_j, kde $j \in \{0, 1..., n-1\} \setminus \{i\}$. Druhou



Obrázek 2.1: iDistance [1]



Obrázek 2.2: M-Index s jednou úrovní [1]



Obrázek 2.3: M-Index s l = 2

úroveň tak určí index druhého nejbližšího pivota k danému bodu. Takto jsou shluky děleny i na dalších úrovních.

Definice 3. *Máme-li n pivotů $\{p_0, p_1, ..., p_{n-1}\}$ a prvek $o \in \mathcal{D}$,*

$$(\cdot)_o : \{0, 1, ..., n-1\} \rightarrow \{0, 1, ..., n-1\}$$

je permutace indexů taková, že

$$d(p_{(0)_o}, o) \leq d(p_{(1)_o}, o) \leq \cdots \leq d(p_{(n-1)_o}, o).$$

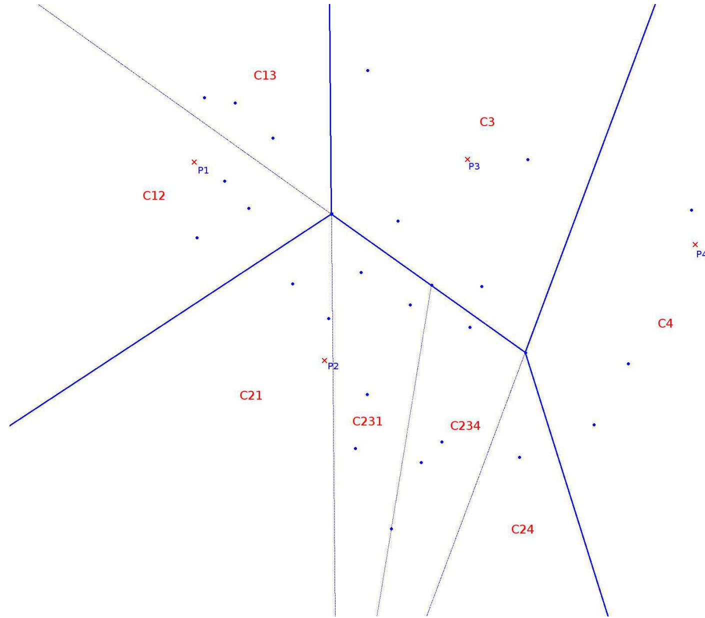
M-Index s l úrovněmi, kde l je celé číslo $1 \leq l \leq n$, rozděluje prostor $\mathcal{D}$ do $n \cdot (n-1) \cdot \cdots \cdot (n-l+1)$ částí následujícím způsobem.

- Na první úrovni jsou prvky přiřazeny k pivotu, od kterého mají nejmenší vzdálenost. Vzniknou shluky C_i takové, že $(0)_o = i$ pro objekt $o \in C_i$.
- Každý shluk C_i je rozdělen do $n-1$ shluků stejným způsobem za použití pivotů $\{p_0, p_1, ..., p_{i-1}, p_{i+1}, ..., p_{n-1}\}$. Vzniknou shluky $C_{i,j}$, kde j je index druhého nejbližšího pivotu k prvku $o \in C_{i,j}$, tedy $(1)_o = j$.
- ...

Na obrázku 2.3 je znázorněno rozdělení prostoru M-Indexem se dvěma úrovněmi. Na tomto obrázku je také vidět, že do některých shluků může spadat velké množství prvků a do některých naopak žádné. Je zde také patrné, že některé shluky (zde C14 a C41) ve skutečnosti neexistují.

Nechť $o \in \mathcal{D}$ patří do shluku $C_{i_0, i_1, ..., i_{l-1}}$, pak lze každému prvku o přiřadit číselný klíč podle vzorce

$$key_l(o) = d(p_{(0)o}, o) + \sum_{i=0}^{l-1} (i)_o \cdot n^{l-i-1}. \tag{2.3}$$



Obrázek 2.4: M-Index s dynamickým počtem úrovní

Celá část $key_l(o)$ odpovídá číslu $i_0 i_1 \ldots i_{l-1}$ zapsanému v číselné soustavě o základu n. Desetinná část čísla $key_l(o)$ pak odpovídá vzdálenosti prvku o od pivotu $p_{(0)_o}$. Klíč $key_l(o)$ je tedy sestaven tak, aby bylo možné zjistit do jakého shluku objekt o patří a jeho vzdálenost od nejbližšího pivotu.

2.4 M-Index s dynamickým počtem úrovní

Větší počet úrovní zlepšuje prořezávání prostoru při vyhledávání, ale na druhou stranu může příliš rozdělit prostor, který je nutné při vyhledávání projít. Aby se zachoval přínos přidání další úrovně a prostor se zbytečně nerozdroboval, M-Index umožňuje dynamický počet dělení jednotlivých částí prostoru. K dalšímu dělení shluku dojde jen v případě, že počet prvků uvnitř něj dosáhne nebo překročí předem dané hraniční hodnoty.

Pro M-Index s dynamickým počtem úrovní je nutné:

- určit maximální počet úrovní l_{max}, $1 \leq l_{max} \leq n$, kde n je počet pivotů,
- změnit výpočet hodnoty klíče daného prvku na

$$key_l(o) = d(p_{(0)_o}, o) + \sum_{i=0}^{l-1} (i)_o \cdot n^{l_{max}-i-1}, \tag{2.4}$$

- ukládat aktuální strukturu shluků do stromové struktury, které budeme říkat strom shluků.

Přidání další úrovně vyžaduje pouze lokální úpravy rozdělovaného shluku. Prostor rozdělený M-Indexem s dynamickým počtem úrovní je znázorněn na obrázku 2.4 (maximální počet prvků ve shluku je zde 5).

2.4.1 Strom shluků

Strom shluků uchovává aktuální strukturu metrického prostoru.
Vnitřní uzly stromu shluků obsahují:

- úroveň uzlu l, je to úroveň, ve které uzel vznikl dělením nadřazeného shluku,
- prefix permutace pivotů délky l, shluk $C_{i_0,i_1,\ldots,i_{l-1}}$ obsahuje tuto permutaci pivotů $(i_0, \cdots, i_{l-1})$,
- ukazatele na shluky s úrovní $l+1$, které vznikly jeho rozdělením $ptr_j l+1$.

$$\langle l, (i_0, \ldots, i_{l-1}), (ptr_0 l+1, \ldots, ptr_{n-1} l+1) \rangle$$

Kořen stromu shluků vypadá vždy následujícím způsobem.

$$\langle 0, (), (ptr_0 1, \ldots, ptr_{n-1} 1) \rangle$$

Listové uzly stromu shluků obsahují:

- úroveň uzlu l, je to úroveň, ve které vznikl dělením nadřazeného shluku,
- prefix permutace pivotů $(i_0, \cdots, i_{l-1})$ délky l,
- minimální a maximální hodnotu klíče key_{min}, key_{max} prvků $o \in X$, které jsou uvnitř tohoto shluku.

$$\langle l, (i_0, \ldots, i_{l-1}), [key_{min}, key_{max}] \rangle$$

Strom shluků k rozdělení prostoru z obrázku 2.4 je znázorněn na obrázku 2.5.

2.5 Volba pivotů

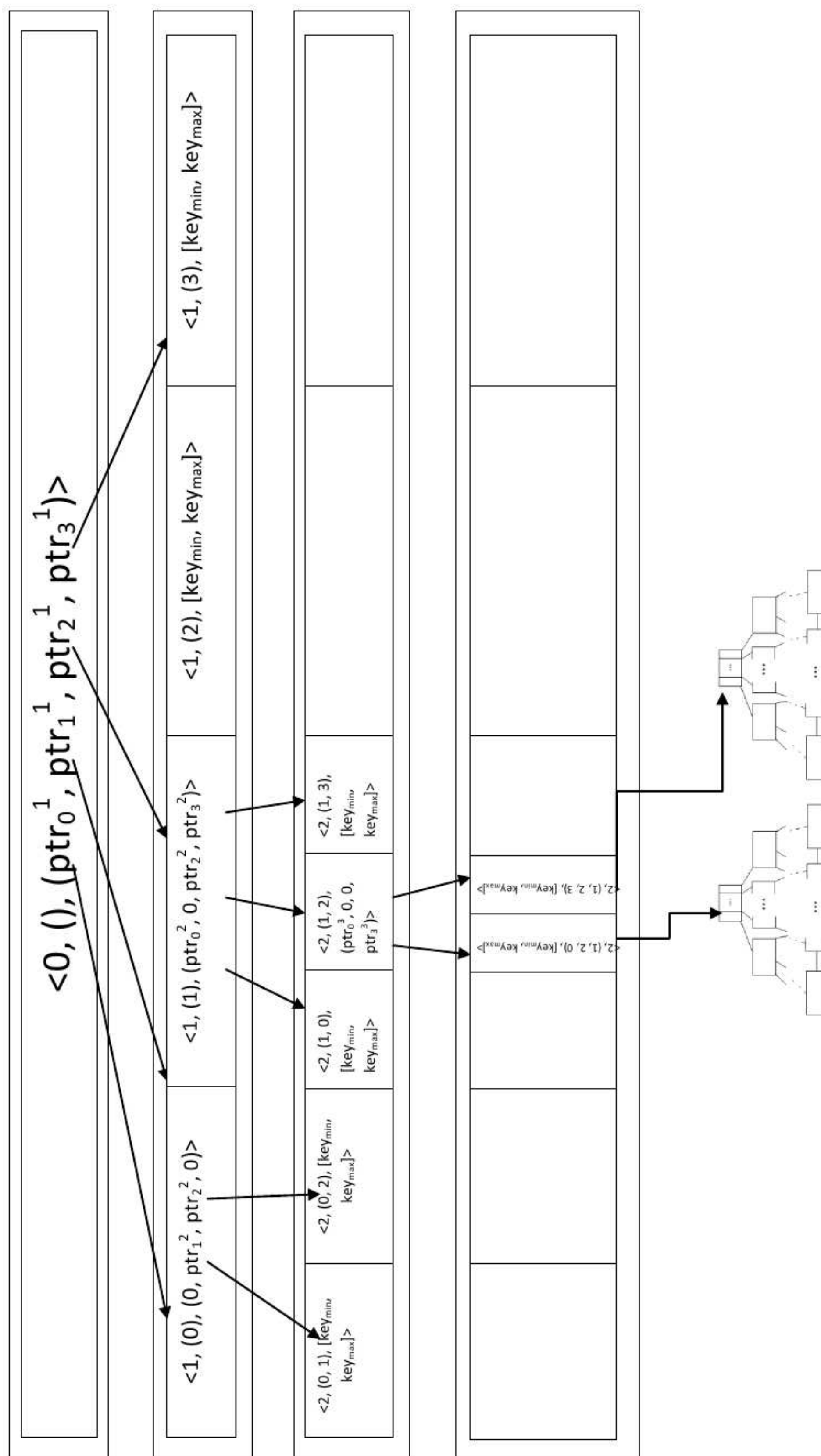
Efektivnost vyhledávání pomocí M-Indexu velmi záleží na rozdělení prostoru, tedy na volbě pivotů. Autoři M-Indexu zvolili pro výběr pivotů algoritmus *Incremental selection*. V algoritmu Incremental selection je podle určeného kritéria postupně z množiny kandidátů vybíráno v každém kole po jednom pivotu. Pivoty vybrané v předchozích kolech ovlivňují výběr pivota v kolech následujících.

2.5.1 Kritérium efektivnosti

Kritérium efektivnosti [5] má udávat, jak moc je daná sada prvků vhodná pro to, být sadou pivotů. Zvolené pivoty by měly mezi sebe prostor co nejlépe rozdělovat. Je očekáváno, že se díky nim podaří co nejlépe prořezat prostor a tím co nejvíce snížit množství potřebných výpočtů podobnosti dvou prvků.
Máme-li dotaz $R(q, r)$, můžeme podle věty z části object-pivot distance 1.4.1 z vyhodnocování dotazu, bez výpočtu vzdálenosti od vzoru, vyloučit prvky o, pro které platí:

$$|d(p_i, o) - d(p_i, q)| > r \tag{2.5}$$

pro nějaký pivot p_i.



Obrázek 2.5: Strom shluků, M-Index s dynamickým počtem úrovní

Definice 4. *Množina k pivotů* $\{p_1, p_2, \ldots, p_k\}, p_i \in \mathbb{U}$ *určuje prostor vzdáleností* $\mathbb{P}$ *mezi dvojicemi pivot a prvek z* $\mathbb{U}$*.* $[u]$ *je zobrazení do tohoto prostoru,* $[u] : \mathbb{U} \to \mathbb{P}$*,* $[u] = (d(u, p_1), d(u, p_2), \ldots, d(u, p_k))$*.*

Definice 5. *Metrika* $D_{\{p_1,\ldots,p_k\}}([x], [y]) = max_{1\leq i\leq k}|d(x, p_i) - d(y, p_i)|$*.*

Uspořádaná dvojice $(\mathbb{P}, D)$ je metrický prostor.
Podmínka pro odfiltrování prvků 2.5 se pro metriku D dá zapsat jako

$$D_{\{p_1,\ldots,p_k\}}([q], [u]) > r. \tag{2.6}$$

Aby se co nejvíce snížil počet prvků, pro které se bude muset jejich vzdálenost od vzoru počítat, musí se co nejvíce zvýšit pravděpodobnost jevu 2.6. Jednou možností je maximalizovat střední hodnotu μ_D rozdělení vzdáleností z D.

Věta 4. *Sada pivotů* $p_1, \ldots p_k$ *je lepší než sada pivotů* $p'_1, \ldots p'_k$*, jestliže*

$$\mu_{D_{\{p_1,\ldots,p_k\}}} > \mu_{D_{\{p'_1,\ldots,p'_k\}}}. \tag{2.7}$$

K μ_D je počítán pouze odhad. Náhodně se vybere A párů objektů z $\mathbb{U}$, $\{(a_1, a'_1), (a_2, a'_2), \ldots, (a_A, a'_A)\}$. Vezme se obraz těchto párů v prostoru $\mathbb{P}$, $\{D_1, D_2, \ldots, D_A\}$. μ_D je poté odhadnuto jako $\mu_D = \frac{1}{A}\sum_{1\leq i\leq A} D_i$.

2.5.2 Incremental selection

První pivot p_1 je vybrán ze sady kandidátů, prvků z $\mathbb{U}$, velikosti N tak, aby hodnota μ_D byla co největší. Další pivot p_2 je vybrán z jiné sady N kandidátů z $\mathbb{U}$ tak, aby $\{p_1, p_2\}$ mělo největší možnou hodnotu μ_D, p_1 je pevně daný pivot zvolený v minulém kroku algoritmu. Takto je postupně v každém kroku vybírán další pivot, který bude přidán k sadě pivotů, než bude mít sada pivotů požadovanou velikost.

3. Vyhledávání v M-Indexu

Dotazy podporované M-Indexem jsou již zmíněné $R(q,r)$ a $kNN(q,k)$ 1.2.
M-Index využívá vlastností funkce vzdálenosti dvou prvků v metrických prostorech a rozdělení prostoru na shluky. Při vyhledávání používá principů *double-pivot distance*, *object-pivot distance* a *range-pivot distance*. Podle nich jsou vyřazovány části prostoru, ve kterých se výsledek nemůže vyskytnout, čímž se snižuje množství výpočtů vzdálenosti dvou prvků a zefektivňuje se vyhledávání.

3.1 Hledání sousedů do určité vzdálenosti

Algoritmus pro vyhledání všech prvků, které jsou vzdálené od daného vzoru maximálně r [1], je zapsán v algoritmu 1.
Prostor je postupně procházen od kořene stromu shluků reprezentujícího celý prostor, postupně přes potomky, reprezentující podprostory svých rodičů, po jeho listy, které obsahují odkaz na strukturu s daty z příslušného podprostoru.
V každém běhu cyklu (v algoritmu na řádce 8) je vyhodnocován jeden uzel z fronty Q. Je nalezen pivot p_j, který je nejblíže k vzorovému prvku q a ještě nebyl po cestě od kořene k rodiči uzlu použit k rozdělení prostoru $(min\{j \geq 0; (j)_q \notin \{i_0, \ldots, i_{l-2}\}\})$, a pivot $p_{i_{l-1}}$, který byl použit k rozdělení prostoru rodiče na podprostor tohoto uzlu. Pokud je rozdíl vzdáleností vzoru od těchto pivotů větší než $2r$, potom podle pravidla *double-pivot distance* ve shluku odpovídajícím uzlu *node* nemohou být prvky do vzdálenosti r od q a uzel, ani jeho potomci, nebude dále prohledáván.
Když je právě zkoumaný uzel listový, použije se pravidlo *range-pivot distance* pro rozhodnutí, zda v něm mohou být obsaženy prvky z výsledku a zda má tedy smysl prvky v něm dále procházet (v algoritmu na řádce 24). Následně se dá pro některé prvky použít pravidlo *object-pivot distance* a o prvku p říci, že je od q vzdálený více nebo méně než o r, a to bez počítání vzdálenosti (v algoritmu na řádce 29). Ke zbylým prvků v listovém uzlu musí být jejich vzdálenost od vzoru spočítána.

3.2 Hledání k nejbližších sousedů

Při hledání k nejbližších sousedů [1], [2] se předpokládá, že prvky nejbližší k vzoru q budou pravděpodobně ve shluku, do kterého by patřil i prvek q. Dále pak ve shlucích, které vznikly rozdělením prostoru přiřazením jeho částí k pivotům, které měly co nejmenší vzdálenost od pivotů, ke kterým byl přiřazován shluk obsahující prvek q. Tyto shluky jsou prohledávány s větší prioritou.
Ke každému shluku, který odpovídá listovému uzlu stromu shluků, je spočítána penalizace, která odpovídá právě předpokládané pravděpodobnosti výskytu prvků nejbližších k q v tomto shluku. Čím větší priorita, tím nižší penalizace.
Penalizace je dána vzorcem 3.1.

Algorithm 1 R(q, r)

```
1: Set A:=∅; {množina A bude obsahovat výsledek}
2: for i := 0 to n − 1 do
3:    calculate d(p_i, q); {spočítá vzdálenost q ke všem pivotům}
4: end for
5: sort(p_0, ··· p_{n−1}) {setřídí pivoty podle vzdálenosti od q}
6: Queue Q:=∅
7: Q.enqueue(root) {root je kořen stromu shluků}
8: while Q not empty do
9:    node:=Q.dequeue();
10:   l:=node.level;
11:   {i_0, ... i_{l−2}}:=node.pivot_permutatiton;
12:   j:= min{j ≥ 0; (j)_q ∉ {i_0, ..., i_{l−2}}};
13:   if l > 0 ∧ d(p_{i_{l−1}}, q) − d(p_{(j)_q}, q) > 2r then
14:      Continue; {tento shluk podle pravidla double-pivot distance nemůže obsahovat data z výsledku, je přeskočen}
15:   end if
16:   if node is internal then
17:      for i := 0 to n − 1 do
18:         Q.enqueue(*ptr_i l + 1);
19:      end for
20:   else
21:      {node je listový uzel}
22:      r_min := node.key_min;
23:      r_max := node.key_max;
24:      if d(p_{i_0}, q) + r < r_min ∨ d(p_{i_0}, q) − r > r_max then
25:         continue; {tento shluk podle pravidla range-pivot distance nemůže obsahovat data z výsledku, je přeskočen}
26:      end if
27:      data:=node.data.getData(⌊r_min⌋ + d(p_{i_0}, q) − r, ⌊r_max⌋ + d(p_{i_0}, q) + r);
28:      for all p in data do
29:         if max_{i=0}^{n−1} |d(p_i, q) − d(p_i, p)| > r then
30:            Next p; {tento prvek je podle pravidla object-pivot distance od q vzdálený více než o r}
31:         end if
32:         if d(q, p) ≤ r then
33:            A.add(p);
34:         end if
35:      end for
36:   end if
37: end while
38: return A;
```

$$penalty(C_{i_0,\ldots,i_{l-1}}) = \frac{1}{l} \cdot \sum_{j=0}^{l-1} max \left\{ d(p_{i_j}, q) - d(p_{(j)_q}, q), 0 \right\}. \quad (3.1)$$

Postup hledání k nejbližších sousedů k prvku q je uveden v algoritmu 2.

Algorithm 2 kNN(q, r)

1: Set K:=∅; {množina K bude obsahovat dočasný výsledek}
2: Queue Q; {je prioritní fronta, řadí uzly podle penalizace}
3: enqueue_leaf_nodes(root); {do Q vloží všechny listové uzly stromu shluků s kořenem root}
4: leaf_flag[]; {obsahuje informaci, zda byl list i stromu shluků již navštíven}
5: leaf_left[]; {ukazatel na první nenavštívený prvek při vyhledávání vlevo}
6: leaf_right[]; {ukazatel na první nenavštívený prvek při vyhledávání vpravo}
7: **while** $K.size() < k \wedge i < Q.size()$ **do**
8:     leaf_flag[i] = true;
9:     $Q[i].findFirstK(q, k, K, leaf_left[i], leaf_right[i])$; {vloží k prvků nebo všechny ze shluku $Q[i]$ do K}
10:     ++i;
11: **end while**{navštíví po řadě prvky z Q, než nalezne prvních k prvků}
12: Float rk:=$max\{d(p, q), p \in K\}$; {max vzdálenost prvku z K od q}
13: Float r:=deltar; {vzdálenost v které se vyhledává}
14: **while** $r < rk$ **do**
15:     i:=0;
16:     **while** $i < Q.size()$ **do**
17:         $leaf_flag[i] := Q[i].findKInRange(q, k, r, K, leaf_left[i], leaf_right[i], leaf_flag[i])$; {nahradí prvky v K bližšími prvky ze shluku $Q[i]$}
18:         ++i;
19:     **end while**
20:     r+=deltar;
21:     rk:=$max\{d(p, q), p \in K\}$;
22: **end while**
23: **return** K;

Hledání k nejbližších sousedů začíná spočítáním penalizací všech shluků v listech stromu shluků a jejich vložení do prioritní fronty Q, právě podle hodnoty penalizace.
Do množiny K, která po čas vyhledávání obsahuje dočasný výsledek, je vloženo prvních k prvků. Tyto první prvky jsou vybírány nejdříve ze shluků s nejnižší penalizací. V každém shluku $C_{i_0,\ldots,i_{l-1}}$ se začne vkládáním prvků, které mají hodnotu klíče nejbližší hodnotě klíče, kterou by měl prvek q, kdyby byl prvkem tohoto shluku, tedy prvků majících podobnou vzdálenost od pivota p_{i_0} jako vzor q. Při navštívení shluku $Q[i]$ je nastaven $leaf_flag[i]$ na true a jsou nastaveny ukazatele $leaf_right[i]$ na následující prvek vpravo a $leaf_left[i]$ vlevo.
V dalších krocích jsou vyhledávány prvky v malé sféře kolem q. Postupně se zvětšuje poloměr r sféry kolem q. Pro každou hodnotu r jsou opět postupně procházeny všechny shluky $Q[i]$ z Q a jsou z nich vybírány prvky, které by podle pravidla *object-pivot distance* mohli být do vzdálenosti r od q. Pokud už byl shluk $Q[i]$

procházen, postupuje se dále od ukazatelů $leaf_left[i]$ a $leaf_right[i]$. Každému navštívenému shluku je opět aktualizována hodnota $leaf_flag[i]$, $leaf_left[i]$ a $leaf_right[i]$.
Vyhledávání končí, když poloměr vyhledávání r přeroste rk. rk obsahuje vzdálenost mezi q a prvkem z množiny K, který je od q nejvzdálenější. Do množiny výsledků K jsou přidávány další prvky a vyřazovány ty nejvzdálenější od q, tak aby velikost množiny K byla stále k. Hodnota rk tedy postupně klesá. Ve chvíli, kdy r překročí rk, množina K obsahuje právě k nejbližších sousedů.
Pokud nám stačí přibližné výsledky a záleží nám více na efektivnosti vyhodnocení, může být tento algoritmus ukončen i dříve, například při překročení nějakého množství výpočtu vzdáleností.

4. Design

Naším cílem bylo implementovat index pro vysokorozměrná data, který by byl co nejobecněji použitelný pro různé domény dat. V této kapitole popíšeme k tomu použité prostředky a návrh implementace.

4.1 Použitý databázový server

Pro implementaci metrického indexu přicházely v úvahu buď databázové servery s otevřeným zdrojovým kódem, nebo ty, které podporují implementaci uživatelem definovaných indexů.
Zvolen byl databázový server Oracle, a to díky možnosti rozšíření funkcionality pomocí takzvaných `data cartridge`, v našem případě se jednalo hlavně o dobře zdokumentované rozhraní pro rozšiřitelné indexování, které je podporováno ve všech konfiguracích databázového serveru počínaje volně šiřitelnou verzí XE.
Tímto způsobem se v Oracle dá pro data, která neumí sám indexovat, a tedy v nich dostatečně efektivně vyhledávat pomocí vestavěného indexu, implementovat vlastní doménový index.
Dalšími výhodami databázového systému Oracle je jeho výkonnost, možnost běhu na mnoha platformách a v neposlední řadě možnost implementovat a importovat funkce psané v C/C++ nebo jiném vyšším programovacím jazyku zkompilované do DLL knihovny.

4.2 Extensible Indexing v Oracle

Implementace vlastního doménového indexu obnáší vytvoření vlastního indexového typu (tzv. *indextype*).
K vytvoření indexového typu je zapotřebí

- naprogramovat metody specifikované rozhraním ODCIIndex,
- vytvořit operátory, které bude náš index podporovat.

Úkolem implementace indexového typu je postarat se o

- definici struktury indexu,
- uložení dat v indexu,
- údržbu indexu při manipulaci s daty a jejich navracení při vyhodnocování dotazu.

Databázový systém pak spolupracuje s naší implementací při stavění, správě a používání indexu.
Extensible Indexing je vhodné použít například pro

- indexování nestrukturovaných dat,
- implementaci vlastních operátorů,

- indexování částí objektů,
- indexování hodnot odvozených pomocí operací specifických pro danou doménu.

4.2.1 Interface ODCIIndex

V rozhraní `ODCIIndex` je definována sada funkcí, které musí typ implementovat jako své metody, aby z něj mohl být vytvořen indextype. Jsou to metody pro

- definici indexu,
- manipulaci s daty,
- procházení indexem a
- pro získávání metadat.

Metody pro definici indexu

Tyto metody jsou volány při požadavku uživatele na `CREATE`, `DROP`, `ALTER` a `TRUNCATE INDEX`.

- `ODCIIndexCreate`
- `ODCIIndexDrop`
- `ODCIIndexAlter`
- `ODCIIndexTruncate`

Pokud například uživatel zavolá `CREATE INDEX` ... `INDEXTYPE t PARAMETERS (`...`)` naimplementovaného indextypu, Oracle vyvolá jeho `ODCIIndexCreate` metodu.

Metody pro manipulaci s daty

Metody pro manipulaci s daty jsou volány vždy, když uživatel vykoná některou z aktualizačních operací `INSERT`, `DELETE`, nebo `UPDATE` nad tabulkou a sloupcem, nad kterými je index vytvořený.

- `ODCIIndexInsert`
- `ODCIIndexDelete`
- `ODCIIndexUpdate`

Metody pro procházení indexem

Tyto metody jsou volány při vyhodnocování predikátů, které obsahují operátory podporované daným indexem.

- `ODCIIndexStart`
- `ODCIIndexFetch`
- `ODCIIndexClose`

Procházení indexem začíná voláním metody `ODCIIndexStart`, tato metoda dostane informace o indexu a operátoru, které již nejsou předávány následně volaným metodám `ODCIIndexFetch` a `ODCIIndexClose`. V `ODCIIndexStart` jsou tedy do paměti alokované na dobu trvání vyhledávání ukládaná data, která je potřeba v průběhu procházení indexu sdílet. Tato data jsou vracena ve výstupním `sctx` parametru. `ODCIIndexFetch` a `ODCIIndexClose` se pak k těmto datům dostanou jako členské metody přes parametr `SELF`.
Metoda `ODCIIndexFetch` vrací identifikátory řádek, pro něž hodnoty vrácené operátorem splňují daný predikát. Konec vyhledávání je oznámen vrácením `NULL` identifikátoru řádky.
Metoda `ODCIIndexClose` vykonává úklid po vyhledávání.

Metody pro získávání metadat

Do této kategorie patří metoda

- `ODCIIndexGetMetadata`,

která je volitelná. Pokud je implementována, může být volána při exportu databáze a vracet informace o indexu, které nejsou uloženy v systémovém katalogu a budou zapsány do dumpu. Tyto informace jsou zapisovány v podobě anonymních PL/SQL bloků a budou vykonány při importu před vytvářením daného indexu.

4.2.2 Operátory

Definice operátorů

Uživatelsky definované operátory jsou podobné těm vestavěným, mohou být použity ve stejných situacích a slouží především k zjednodušení zápisu SQL příkazů. Operátory jsou identifikovány jménem a jsou napojeny na funkce, které udávají jejich chování v daném kontextu. Vytvoření a napojení na funkci může být provedeno následovně.

```
CREATE OPERATOR otisky.podobnost
BINDING
(BFILE, BFILE) RETURN NUMBER USING otisky.podobnostni_fce;
```

Operátor je pak zrušen klasicky pomocí `DROP OPERATOR`. K existujícímu operátoru je možné přidat napojení na funkci pomocí `ALTER OPERATOR` s klauzulí `ADD BINDING`. Tak jako vestavěné operátory může být i ten uživatelsky definovaný použit všude, kde se může objevit nějaký výraz:

- v `SELECT` části dotazu
- ve `WHERE` klauzuli
- v `ORDER BY` nebo `GROUP BY` klauzuli.

Když je operátor zpracováván, je jako výsledek operátoru vyhodnocena funkce, na kterou je operátor napojen. Pokud je operátor spojen s více funkcemi, je vybrána ta, jejíž argumenty mají odpovídající datové typy.

Operátory a Indextype

Ve chvíli, kdy je operátor nadefinován, může být vytvořen indexový typ, který bude tento operátor podporovat a který se může postarat o efektivní vyhledání odpovídajících řádek v indexu.

```
CREATE INDEXTYPE otisky.otiskyIdx FOR otisky.podobnost (BFILE, BFILE)
USING otisky.TypeName;
```

Indexu, který je vytvořen z implementovaného indexového typu, se pak říká `doménový index`.
Doménový index pak může být vytvořen nad sloupcem tabulky a může být použit při vyhodnocování operátorů specifických pro danou aplikaci.

Vyskytne-li se operátor mimo klauzuli `WHERE`, bývá jeho chování právě takové, jak je definováno ve funkci, na kterou je napojen. Je procházena celá tabulka a ke každému řádku spočtena hodnota operátoru.
Aby mohlo být použito vyhledávání řádek v indexu, musí se operátor vyskytovat v predikátu ve `WHERE` klauzuli a operovat nad sloupcem nebo atributem objektu, nad kterým je postaven daný index. V takovém případě ještě o použití funkční implementace operátoru nebo vyhledání v indexu rozhodne optimalizátor.
Když je k vyhodnocení predikátu s operátorem vybrán průchod indexu, jsou postupně volány metody indexového typu `ODCIIndexStart`, `ODCIIndexFetch` a `ODCIIndexClose`, jak bylo popsáno výše. `ODCIIndexStart` dostane mezi svými parametry i jméno a argumenty operátoru a dolní a horní hranici hodnoty z predikátu.

```
CREATE INDEX otisky_idx ON tabulka_otisku(otisk)
INDEXTYPE IS otisky.otiskyIdx;

SELECT * FROM tabulka_otisku
WHERE otisky.podobnostst(otisk, bfile) < 0.1;
```

Existuje i možnost použití vyhledávání v indexu i ve funkcionální implementaci operátoru, a tedy i použití indexu při použití operátoru i v `SELECT` části dotazu.
Operátor musí operovat nad sloupcem nebo atributem objektu, nad kterým je vytvořený index, a funkční implementace musí mít navíc parametry index context, scan context, scan flag. Index context obsahuje informace o daném doménovém indexu a identifikaci řádku, nad kterým je zrovna operátor vyhodnocován. Scan

context obsahuje data sdílená mezi jednotlivými voláními pro uchování stavu vyhledávání. Scan flag ukazuje, zda se jedná o poslední volání, které se má postarat o úklid po vyhledávání.

4.3 Extensible Optimizer v Oracle

V Oracle se dá rozšiřovat i optimalizátor a to implementací rozhraní *Extensible Optimizer Interface*. Vytvořené rozšíření optimalizátoru pak sbírá statistické informace, jako je selektivita a odhad ceny vykonání, a generuje plán pro vykonání SQL příkazu. Tyto informace jsou použity optimalizátorem při sestavování plánu dotazu.

```
ASSOCIATE STATISTICS WITH INDEXTYPE otisky.otiskyIdx USING statType;
```

5. Požadavky na uživatele

V této kapitole popíšeme návrh rozhraní, přes které uživatel nadefinuje, jakým způsobem má náš index pracovat s indexovanými daty.
Abychom mohli vyhledávat v nestrukturovaných nebo vysokodimenzinonálních datech z různých domén, je nutné od doménového specialisty požadovat, aby vytvořil nebo alespoň naspecifikoval funkci pro extrakci strukturovaných dat z objektů domény, kterou chce indexovat. Stejně tak by měl vytvořit funkci, která bude počítat podobnost dvou vzorků vyextrahovaných dat. Tato podobnostní funkce musí splňovat podmínky metriky tak, aby se vyextrahovaná data stala prvky metrického prostoru a mohli bychom použít pravidel platících pro metrické prostory.

5.1 Podporované datové typy

Díky možnosti použití funkcí pro extrakci strukturovaných dat a pro výpočet vzdálenosti dle uživatelovy definice, je náš index obecnější a použitelný pro více různých domén. Jsou však omezené datové typy, které uživatel může indexovat. Podporované datové typy jsou

- `BLOB`,
- `CLOB`,
- `VARCHAR2`.

Toto omezení vzniklo kvůli operátorům, které jsou vytvářeny s udáním typů parametrů, nad kterými budou operovat.

5.2 Typ TChar

`TChar` je námi vytvořený typ pro uložení pole celých čísel. Od uživatele je požadováno, aby data, která jsou výsledkem extrakce strukturovaných dat, byla právě tohoto typu. Definice TChar vypadá takto.

```
CREATE TYPE TChar AS VARRAY(100) OF INTEGER;
```

5.3 Objektový typ mspace

Abychom mohli provést extrakci strukturovaných dat tak, jak si ji uživatel bude představovat, a počítat vzdálenosti respektive podobnosti mezi jednotlivými prvky, vyžadujeme od uživatele, aby vytvořil svůj objektový typ, který tyto metody bude implementovat. Tento uživatelem definovaný datový typ, popisující převod vkládaných elementů na prvky metrického prostoru, by měl být vytvořen jako potomek námi definovaného objektového typu `mspace` a přetěžovat členské funkce `fe` a `distance` tak, jak uživatel potřebuje.

Při vytváření indexu pomocí našeho indexového typu pak uživatel jako parametr uvede jméno objektového typu pro převod prvků do metrického prostoru. Tím se dosáhne možnosti mít v jednu chvíli definováno více druhů indexů pod společným typem indexu a tím možnosti efektivně vyhledávat ve více nezávislých datových typech.

5.3.1 Definice mspace

Definice objektového typu `mspace` vypadá takto.

```
CREATE OR REPLACE
TYPE mspace AS OBJECT(
  name VARCHAR2(100),
  MEMBER FUNCTION fe(orig_val VARCHAR2) RETURN TChar,
  MEMBER FUNCTION fe(orig_val CLOB) RETURN TChar,
  MEMBER FUNCTION fe(orig_val BLOB) RETURN TChar,
  MEMBER FUNCTION distance(el1 TChar, el2 TChar) RETURN FLOAT
)
NOT FINAL NOT INSTANTIABLE;
```

Členská funkce s názvem `fe` je funkcí pro extrakci strukturovaných dat. Parametrem jsou data, ze kterých chceme extrahovat strukturovaná data ve formě `TChar`. Parametr funkce musí být jednoho z výše uvedených podporovaných typů.
Členská funkce `distance` počítá vzdálenost dvou prvků. Tyto prvky jsou v podobě již vyextrahovaných strukturovaných dat, tedy typu `TChar`.

5.3.2 Příklad definice potomka mspace

Příkladem uživatelem definovaného objektového typu, který je potomkem našeho objektového typu mspace a implementuje zadané metody, může být například typ `usr_mspace` z následujícího příkladu.

```
CREATE OR REPLACE
TYPE user_mspace UNDER mspace(
  OVERRIDING MEMBER FUNCTION fe(orig_val VARCHAR2) RETURN TChar,
  OVERRIDING MEMBER FUNCTION distance(el1 TChar, el2 TChar) RETURN FLOAT
);

CREATE OR REPLACE
TYPE BODY user_mspace AS
  OVERRIDING MEMBER FUNCTION fe(orig_val VARCHAR2) RETURN TChar IS
    pos integer := -1;
    num varchar2(2);
    res TChar := TChar(0,0);
  BEGIN
    pos := instr(orig_val, ',');
    num := substr(orig_val, 1, pos - 1);
    res(1) := to_number(num);
    num := substr(orig_val, pos + 1, length(orig_val) - pos);
    res(2) := to_number(num);
```

```
    RETURN res;
  END;

  OVERRIDING MEMBER FUNCTION distance(el1 TChar, el2 TChar) RETURN FLOAT IS
  suma float := 0;
  BEGIN
    suma := power(el1(1) - el2(1), 2) + power(el1(2) - el2(2), 2);
    RETURN sqrt(suma/20000);
  END;

END;
```

Zde je pro jednoduchost indexovaným typem `VARCHAR2`, přičemž každá hodnota obsahuje čárkou oddělená dvě čísla menší než sto. Funkce `fe` provede extrakci strukturovaných dat a navrací pole čísel požadovaného typu `TChar`. Funkce `distance` pak počítá vzdálenost mezi jednotlivými extrahovanými hodnotami, v tomto případě pomocí standardní Eukleidovské metriky, jejíž hodnota je následně vydělena 20000, aby hodnota vzdálenosti nepřesáhla číslo 1.

6. Implementace

Naším cílem bylo implementovat rozhraní `ODCIIndex` tak, aby jeho chování odpovídalo chování `Metric Indexu` tak, jak byl popsán v kapitole 2. Z implementovaného rozhraní a operátorů vytváříme indexový typ `mindex_type`. Indexový typ je pak svým chováním podobný indexům, které jsou přímo podporovány Oracle serverem. Například indexovému typu CTXSYS.CONTEXT pro vyhledávání v textových datech. Rozhraní a část implementace objektového typu, implementující metody rozhraní `ODCIIndex`, jsou zapsány v `PL/SQL`. Výpočetně náročnější části metod jsou však napsány v `C++`, přeloženy do DLL knihovny a volány z metod rozhraní jako externí funkce.

6.1 Uložení dat indexu

Podstatné části algoritmů pro manipulaci s daty v indexu jsou volané jako externí funkce. Externí funkce v Oracle jsou standardně bezstavové a všechny alokované ukazatele (handly) jsou na konci volání externí funkce implicitně uvolňovány.
Oracle navíc nabízí možnost alokace paměti na dobu trvání session, jejíž uvolnění si musí vývojář explicitně zavolat sám. Takto alokovaná paměť je přístupná i z dalších volání. Této možnosti využíváme například při ukládání dat při vyhledávání v indexu pro předání dat mezi `ODCIIndexStart` a jednotlivými voláními `ODCIIndexFetch` a paměť uvolňujeme až v `ODCIIndexClose`.
Pro uložení dat indexu, která musí být uchována perzistentně, nám ani doba trvání na celou session nepomůže. Náš index proto využívá tabulku v databázi `mindex_data`, v které je ke každému indexu uložen `BLOB` se serializovanou datovou strukturou stromu shluků a s daty k shlukům příslušejícími.

6.1.1 Strom shluků

Strom shluků, popsaný v 2.4.1, je struktura, která slouží k uložení aktuální podoby rozdělení metrického prostoru, a následně tedy k navigaci při vyhledávání konkrétních částí prostoru - shluků. Vyhledávání začíná v kořeni stromu shluků, dalo by se říci, že kořen reprezentuje celý metrický prostor. Od kořene je postupováno do odpovídajícího syna, reprezentujícího jeho podprostor připadající k příslušnému z pivotů. Vyhledávání pokračuje až k listům stromu shluků.
Jednotlivé uzly jsou adresovány pomocí id, které je jim přiřazeno při serializaci. Každý vnitřní uzel obsahuje pole s id jeho synů.
V listových uzlech stromu shluků jsou id B^{+}stromů, které obsahují prvky patřící do podprostoru reprezentovaného daným uzlem. B^{+}stromy jsou opět adresovatelné pomocí svých id.

Obrázek 6.1: Serializovaný strom shluků

Obrázek 6.2: Serializovaný B^{+}strom

6.1.2 Data ve shluku

Při vkládání nového prvku do indexované tabulky je vložen i prvek do některého ze shluků. Shluk, do kterého bude prvek vložen, je nalezen pomocí stromu shluků. Stejně je tomu i u mazání prvků. Vkládání do, či mazání z nalezeného shluku, pak odpovídá vkládání do, či mazání z B^{+}stromu obsahujícího prvky příslušející k danému shluku.
Vkládaným prvkům je vždy ze sekvence přiřazeno id, pomocí kterého je pak tento prvek reprezentován ve stromu shluků. Jinými slovy, hodnoty vkládané jako prvky do stromu shluků jsou právě tato přiřazená id.

Data indexu uložená v BLOBu v řádce tabulky mindex_data jsou serializovanou podobou stromu shluků s B^{+}stromy z jeho listů. Zjednodušeně si tuto serializovanou strukturu můžeme představit tak, jak je na obrázku 6.1.
Jelikož je velikost jednotlivých metadat předem známá, je dán maximální počet pivotů a maximální počet vrstev stromu shluků, je i pevně dána hranice, kolik uzlů bude strom shluků nejvýše mít a jejich celková velikost.
Součástí metadat každého uzlu stromu shluků je i pole s id uzlů, které jsou jeho syny, pakliže je to listový uzel, namísto pole potomků obsahuje id B^{+}stromu s daty ze shluku.
Serializovaný B^{+}strom je na obrázku 6.2.
Nejvyšší možná celková velikost B^{+}stromu je také předem známá, a to díky hraniční hodnotě počtu prvků, které mohou být do shluku vloženy. Když víme maximální počet prvků B^{+}stromu, víme i maximální počet listových uzlů a také nejvyšší možný počet jeho vnitřních uzlů. Každému B^{+}stromu je ponechán prostor právě o této maximální velikosti.

6.2 Metadata indexu

Ke každému vytvořenému indexu jsou držena metadata, podle kterých se řídí zpracování prvků uložených do indexované tabulky či metadata ukládaná do indexu, sloužící k následnému efektivnějšímu vyhledávání.

6.2.1 Tabulka mindex_data

Během instalace je mimo jiné také ve schématu vlastníka indexu vytvořena tabulka mindex_data, která slouží k uložení dat vytvořených indexů.
Tabulka mindex_data drží k metrickým indexům následující data

- v sloupci `index_name` jméno indexu,
- v sloupci `data` BLOB uchovávající serializovanou datovou strukturu stromu shluků daného indexu,
- v sloupci `usr_mspace` instanci objektového typu, který je potomkem objektového typu `mspace`.

6.2.2 Tabulka _idx

Tabulka s příponou _idx je vytvořena ke každému metrickému indexu při jeho vzniku. Tato tabulka obsahuje

- sloupec `orig_rowid` pro uložení `ROWID` řádku prvku z indexované tabulky,
- sloupec `id`, číselné id přiřazené danému prvku,
- sloupec `data`, obsahující hodnotu typu TChar, odvozenou funkcí pro extrakci strukturovaných dat a
- sloupec `pivot_flag`, indikátor zda je prvek pivotem či ne.

Umístění prvku ve struktuře je zcela závislé na vzdálenosti prvku od jednotlivých pivotů. Sada pivotů zůstává od vytvoření indexu stejná. Při vyhledávání se pak počítá vzdálenost prvku z dotazu od pivotů a využívá se rozdělení prvků do jednotlivých shluků. Vzdálenost se počítá z hodnot spočítaných při extrakci strukturovaných dat, uložených v tabulce _idx. Odvozené hodnoty vytvořené k pivotům tedy nesmí být z tabulky _idx smazány, při dotazování i úpravách dat budou potřeba. Atribut pivot_flag slouží právě k identifikaci, zda prvek je pivotem a zabrání smazání označených řádek z tabulky _idx. Z dat indexu, umístěných v BLOBu v sloupci data tabulky mindex_data, bude prvek smazán jako každý jiný.

6.3 Indexový typ mindex_type

V této části popíšeme kroky prováděné implementací indexového typu při volání jednotlivých metod.

6.3.1 ODCIIndexCreate

Typicky je index pomocí našeho indexového typu vytvářen následovně.

```
CREATE INDEX otisky_idx ON tabulka_otisku(otisk)
INDEXTYPE IS mindex_type PARAMETERS (:mspace otiskyMSpace);
```

V parametru mspace uživatel uvede jméno existujícího objektového typu, který je vytvořen nad objektovým typem mspace, s implementovanými metodami fe a distance. Tento typ je popsán v části 5.3.1.
Při vytvoření metrického indexu je do tabulky `mindex_data` vložen řádek s metadaty daného indexu, jménem vytvořeného indexu obsahujícím i jméno schématu, ve kterém je index vytvořen, prázdným BLOBem, který bude sloužit k uchování serializované datové struktury stromu shluků indexu a instancí objektového typu uvedeného v parametru mspace.
Také je ve schématu vlastníka indexu vytvořena tabulka pro data vzniklá z vkládaných prvků pomocí funkce fe pro extrakci strukturovaných dat. Tato tabulka je pojmenována názvem vytvořeného indexu s příponou _idx.
Ke každému prvku jsou ukládány `ROWID` řádku prvku z indexované tabulky, id, které bylo prvku přiřazeno ze sekvence, odvozenou hodnotu metodou pro extrakci strukturovaných dat a hodnota indikující, zda je prvek pivot či ne.

Vytvoření tabulky pro odvozené hodnoty a vložení dat týkajících se daného indexu jsou provedeny v `PL/SQL` části, která následně volá externí funkci `MIndexCreate`.
Funkce `MIndexCreate` volá funkci pro získání pivotů metodou incremental selection -viz. sekce 2.5. Vytvoří strom shluků, do nějž vloží všechny id přiřazená prvkům. Nakonec zapíše aktuální podobu stromu shluků do BLOBu.

6.3.2 ODCIIndexDrop

Metoda ODCIIndexDrop se stará o úklid po indexu. Maže řádek z tabulky mindex_data a ruší tabulku s odvozenými daty příslušející k dotyčnému indexu.

6.3.3 ODCIIndexInsert

Při vkládání prvku do tabulky, nad kterou je vytvořen náš index, dostaneme jako parametr `ROWID` vloženého řádku a hodnotu prvku ve sloupci, nad kterým je index vytvořen. Na vkládanou hodnotu je použita funkce pro extrakci strukturovaných dat fe, prvku je přiřazeno `id` a získané hodnoty jsou vloženy do tabulky s odvozenými daty. Následně je volána opět externí funkce, které je předáno `id` právě vloženého prvku.
V externí funkci je nalezen listový uzel stromu shluků, v kterém by měl prvek být. A to tak, že je spočtena vzdálenost prvku od jednotlivých pivotů. Pivoty jsou seřazeny od nejbližšího po nejvzdálenější, podle tohoto pořadí jsou z BLOBu čteny uzly stromu shluků, než se dojde k listovému uzlu stromu shluků, do jehož struktury bude prvek vložen. Po vložení jsou aktualizovány hodnoty v daném uzlu stromu shluků a aktualizovaná data jsou vložena na svá místa v BLOBu.
Při vkládání prvku může dojít i k přetečení mezní hodnoty maximálního počtu prvků ve shluku, pak dojde k dělení shluku. Při dělení shluku dojde jen k lokálním změnám. Původní uzel představující štěpený shluk se stane vnitřním uzlem stromu shluků a bude mít tedy i syny, mezi které budou původně jeho prvky rozděleny. V tomto případě je v BLOBu upravena hodnota původního uzlu a jsou do něj přidány uzly potomků.

6.3.4 ODCIIndexDelete

Mazání z indexu probíhá obdobně jako vkládání do něj. V parametru dostaneme hodnotu `ROWID` a hodnotu prvku ve sloupci, nad kterým je index vytvořen, z mazaného původního řádku.
K obdrženému `ROWID` je v tabulce odvozených hodnot nalezen odpovídající řádek a je zjištěno `id` mazaného prvku. Následuje opět volání externí funkce s parametrem `id` právě mazaného prvku.
V externí funkci je nalezen listový uzel stromu shluků, v kterém by měl prvek být, a prvek je smazán ze struktury. Hranice pro minimální počet prvků ve shluku není, při mazání se tedy rozdělení prostoru nemění.
Po návratu z externí funkce je smazán řádek z tabulky odvozených hodnoty odpovídající mazanému prvku. Pokud ale prvek byl pivotem, řádek v tabulce odvozených hodnot zůstane.
Odpovídá-li podmínce mazání více prvků, o vše se postará databázový server sám, naší metodě jsou postupně pouze předány `ROWID` a hodnota indexovaného sloupce jednotlivých mazaných řádků.

6.3.5 ODCIIndexStart

Metoda ODCIIndexStart je implementována celá v `C++`. Tato metoda inicializuje vyhledávání v indexu.
Následující příklad ukazuje, jak by mohlo vypadat vyhledávání v našem indexu.

```
SELECT * FROM tabulka_otisku WHERE knn(otisk, otisk_vzor) <= 10;
```

Kde otisk_vzor by mělo být nahrazeno nějakým vzorem otisku prstů. Tímto dotazem bychom se ptali na deset otisků nejpodobnějších danému vzoru.
Náš index ještě umožňuje dotaz na všechny otisky prstů, jejichž vzdálenost od našeho vzoru je menší než nějaká hodnota. Takový dotaz by vypadal následovně.

```
SELECT * FROM tabulka_otisku WHERE range(otisk, otisk_vzor) <= 0.05;
```

Jako parametry jsou metodě předány datová struktura s informacemi o operátoru, horní a dolní hranice a hodnota vzorového prvku. My budeme vyhledávat prvky do určité hodnoty, buď do určitého počtu nebo určité vzdálenosti, dolní hranice a operátory > či >= tedy v našem případě nemají smysl. Horní hranicí se myslí v případě operátoru `knn` přirozené číslo vyjadřující počet prvků, které mají být nalezeny a v případě `range` reálné číslo z intervalu $[0, 1)$ vyjadřující maximální vzdálenost prvků od vzorového prvku.
V metodě ODCIIndexStart je alokována paměť na dobu trvání session, do které bude ukládán kontext vyhledávání. Ve shlucích nalezneme všechna id, která patří do výsledku. Pole s těmito id uschováme do scan contextu. Ve volání metody ODCIIndexFetch pak do výsledku přidáváme id převedená na odpovídající ROWID řádek v indexované tabulce.
Pro vrácení výsledků jsme měli také možnost v ODCIIndexStart připravit vyhledávání a v průběhu ODCIIndexFetch postupně přidávat prvky do výsledku a držet si kontext, kde jsme s vyhledáváním skončili. Tato možnost by však byla komplikovanější a prostor použitý pro uchování kontextu vyhledávání by převýšil i prostor pro uchování id vyhledaných prvků.

6.3.6 ODCIIndexFetch

Tato metoda v parametru dostává počet ROWID, která může v tomto běhu vrátit.
Jak jsme již zmínili, metoda ODCIIndexFetch postupně ke každému id ze struktury předané z volání ODCIIndexStart vybere z tabulky odvozených hodnot ROWID původních řádek. Konec výčtu prvků patřících do výsledků oznámí přidáním `NULL` prvku do výsledku. Po takto ukončeném ODCIIndexFetch již není znovu voláno ODCIIndexFetch, ale metoda ODCIIndexClose.

6.3.7 ODCIIndexClose

Metoda ODCIIndexClose vykoná úklid po vyhledávání, uvolní například paměť alokovanou v době vyhledávání.

Při instalaci data cartridge s naším indexem jsou vytvořeny a uloženy ve schématu vlastníka

- indexový typ `mindex_type`,
- operátory `range` a `kNN`, které `mindex_type` podporuje
- základní objektový typ pro definici metrického prostoru `mspace`,
- tabulka `mindex_data` pro uchování dat o každém existujícím indexu vytvořeném z našeho indexového typu a
- knihovna s externími funkcemi napsanými v `C++`.

7. Existující řešení

V této kapitole si popíšeme některá existující řešení indexování vysokodimenzonáních dat.
Budeme se věnovat popisu dalších metod pro indexování v metrických prostorech, jako jsou algoritmy VPT, AESA, LAESA a GNAT. A frameworku MESSIF, který umožňuje sestavení vlastních indexů z již připravených modulů schopných efektivně vyhledávat v metrických prostorech či existující moduly rozšířit o vlastní.

7.1 VPT

Vantage Point Tree je metodou pro indexování metrických prostorů používající hierarchické dělení množiny indexovaných dat.
Množina dat S je rozdělena do dvou podmnožin S_1 a S_2. Dělení proběhne podle prvku p, takzvaného *vantage point* neboli pivota, a mediánu vzdáleností d_m od pivota p k prvkům z S. Do jedné množiny jsou umístěny prvky se vzdáleností od p menší než d_m a do druhé prvky se vzdáleností vetší. Na začátku máme celou indexovanou množinu dat X, na kterou je rekurzivně aplikováno toto dělení. Tímto dělením vzniká vyvážený binární strom. Metoda se dá pozměnit na použití střední hodnoty místo mediánu, to však může vést k nevyváženým stromům, což zhoršuje efektivitu vyhledávacích algoritmů.
Struktura VPT je znázorněna na obrázku 7.1.

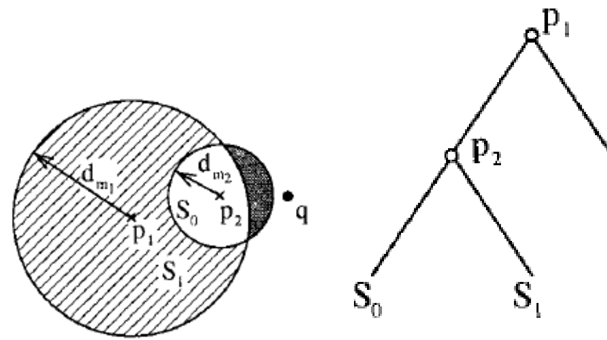
7.1.1 Hledání sousedů do určité vzdálenosti $R(q, r)$

Při vyhledávání je procházen VPT od kořene k listům. V každém vnitřním uzlu je vyhodnocena vzdálenost $d(q, p)$ mezi vzorem a pivotem p použitém k dělení v tomto uzlu. Pokud je vzdálenost $d(q, p) \leq r$, je pivot p zařazen do výstupu.
Ve vnitřních uzlech se pak algoritmus rozhoduje, do kterých podstromů má smysl přistupovat, k tomu je zapotřebí držet si dolní hranici vzdáleností od q. Pokud je poloměr vyhledávání r menší než tato dolní hranice, algoritmus tento podstrom prozkoumávat nebude. Je-li dolní hranice vzdálenosti levého podstromu $r_l = 0$ a horní hranice $r_h = d_m$, pak víme, že vzdálenost všech prvků z levého podstromu od q je nejméně $max\{d(q, p) - d_m, 0\}$. V pravém podstromu platí, že pokud je dolní hranice vzdálenosti prvků od q $r_l = d_m$ a horní hranice vzdálenosti prvků od q $r_h = \infty$, pak vzdálenost prvků v pravém podstromu je nejméně $max\{d_m - d(q, p), 0\}$. To znamená, že algoritmus navštíví levý podstrom, pokud $max\{d(q, p) - d_m, 0\} \leq r$ a pravý podstrom, pokud $max\{d_m - d(q, p), 0\} \leq r$. Navštíveny mohou být oba podstromy.

Při konstrukci VPT jsou spočteny všechny vzdálenosti mezi všemi prvky o v listech a každým pivotem p, který je po cestě od kořene do tohoto listu. Tyto spočítané vzdálenosti se dají uložit a využít v algoritmu pro vyhledávání sousedů do určité vzdálenosti.

- Pokud

$$|d(q, p) - d(p, o)| > r,$$



(a) VPT dělení množiny prvků

(b) VPT stromová reprezentace dělení

Obrázek 7.1: VPT [3]

potom nemusí být počítána vzdálenost $d(q, o)$ a prvek o může být vyřazen. Dolní odhad jeho vzdálenosti od vzoru je totiž větší než poloměr vyhledávání.

- Pokud
$$|d(q, p) + d(p, o)| \leq r,$$
můžeme podle pravidla *object-pivot distance* prvek o rovnou zařadit do výsledku bez počítání vzdálenosti $d(q, o)$. Z trojúhelníkové nerovnosti totiž platí, že $d(q, o) \leq r$.

VPT také jako Metrický index používá dělení metrického prostoru pomocí pivotů, které pak při vyhledávání používá k redukci oblastí, které musí projít k nalezení výsledku. K vynechání částí prostoru VPT používá pravidlo *object-pivot distance* k dolnímu odhadu vzdálenosti mezi prvky v daném podstromu a vzorovým prvkem. Metrický index rozděluje množinu pomocí více pivotů. Podstromy pak vynechává díky vlastnostem dělení na základě pravidla *double-pivot distance*, což umožňuje rozhodnout o vynechání podstromu ze znalosti pouze vzdálenosti mezi vzorem q a danými pivoty.
Oba algoritmy využívají pravidlo *object-pivot distance* pro nahrazení výpočtů vzdálenosti vzoru od prvků v listech, které jsou procházeny.

7.2 AESA

Approximating and Eliminating Search Algorithm je algoritmus pro podobnostní vyhledávání v datech, pro která je náročné právě spočítání podobnosti dvojic prvků. K minimalizaci výpočtů podobnosti dvou prvků během výpočtu slouží matice $n \times n$ obsahující vzdálenosti mezi všemi prvky databáze. Tato matice je spočtena během vytváření struktury AESA.
Při vyhledávání ve struktuře AESA jsou všechny prvky rovnocenné, nejsou zde prvky, které by sloužily jako pivoty a ostatní ne. Při vyhledávání se v jednotlivých krocích může stát pivotem libovolný prvek.

7.2.1 Hledání nejbližšího souseda $NN(q)$

Nejdříve byl algoritmus AESA určen k hledání nejbližšího souseda. Algoritmus jeho vyhledání $NN(q)$ vypadá takto. Dostaneme množinu prvků S, množinu pivotů $S‘ = S$, aktuální vzdálenost $d_{min} = \infty$ k nejbližšímu sousedu o_{nn}.

1. Vyber náhodně prvek p z množiny pivotů $S‘$.
2. Spočti vzdálenost p od vzoru q, $d(p, q)$, pokud je
$$d_{min} > d(q, p),$$
pak $d_{min} = d(q, p)$, $o_{nn} = p$ a $S‘ = S‘ - \{p\}$.
3. Z S vyřaď všechny prvky o, pro které
$$|d(q, p) - d(p, o)| > d_{min}.$$
4. Pokud je $S‘$ prázdná, je vrácen nejbližší soused o_{nn}, jinak je vybrán další pivot z $S‘$.
5. Algoritmus znovu přejde ke kroku 2., než se odfiltrují všechny objekty z S.

Jako o_{nn} je držen zatím vybraný pivot, který má minimální vzdálenosti od vzorového prvku a v d_{min} je uložena právě tato vzdálenost od vzoru q.

7.2.2 Hledání sousedů do určité vzdálenosti $R(q, r)$

Algoritmus probíhá následovně.

1. Vyber náhodně objekt p z prvků, v kterých se vyhledává.
2. Spočti vzdálenost objektu p od vzoru q, $d(p, q)$.
3. Z vyhledávání vyřaď všechny prvky o, pro které
$$|d(q, p) - d(p, o)| > r,$$
kde r je poloměr vyhledávání.
4. Opakuj kroky 1.-4. za použití prořezané množiny prvků, dokud není množina nevyřazených dat dostatečně malá.
5. Ke všem zbývajícím prvkům o je spočtena vzdálenost $d(q, o)$ a jsou vráceny prvky pro něž
$$d(q, o) \leq r.$$

Vzdálenost spočtená v kroku 2. je použita v kroku 3. pro prořezání objektů, ve kterých je vyhledáváno. Toto prořezávání je stejné jako prořezávání pomocí pravidla *object-pivot distance* v Metrickém indexu. Prořezávání nevyžaduje žádné další počítání vzdálenosti dvou prvků, použijí se již spočtené vzdálenosti uložené v matici $n \times n$.

7.2.3 Hledání k nejbližších sousedů $kNN(q,k)$

Hledání k nejbližších sousedů probíhá stejně jako hledání nejbližšího souseda, jen si pamatujeme k prvků, které mají aktuálně minimální vzdálenost od vzoru q, jako d_{min} je pak držena hodnota

$$d_{min} = min\{d_{min}, d_k\},$$

kde d_k je maximum ze vzdáleností $d(q,p)$ vzoru q od všech těch k prvků, které si držíme jako nejbližší k vzoru q.

Podle [3] je algoritmus AESA použitelný pouze pro malé kolekce dat. Při konstrukci vyžaduje čas $O(n^2)$, ke spočtení všech vzdáleností mezi prvky a po dobu existence struktury také prostor $O(n^2)$ k uchování matice $n \times n$. Při vyhledávání $R(q,r)$ s velkým poloměrem vyhledávání či vyhledávání k nejbližších sousedů s vysokým číslem k, vyžaduje čas až $O(n)$, který odpovídá i sekvenčnímu průchodu dat.

7.3 LAESA

Linear Approximating and Eliminating Search Algorithm [8] je obdobou algoritmu AESA vyžadující lineární čas pro předzpracování dat a má i lineární prostorové nároky.
Základní změnou od AESA je volba pevného počtu m pivotů a z toho vyplývající změna matice vzdáleností na matici $n \times m$. Musejí být však zvoleny pivoty. Pivoty jsou vybírány tak, aby od sebe byly co nejvzdálenější.
Vyhledávání je velmi podobné vyhledávání pomocí algoritmu AESA, jako pivoty jsou brány ale pouze prvky z m pevně daných pivotů.
Ukážeme si algoritmus pro vyhledávání sousedů do určité vzdálenosti.

7.3.1 Hledání sousedů do určité vzdálenosti $R(q,r)$

1. Pro všechny pivoty p z množiny pivotů opakuj 2.-3..
2. Spočti vzdálenost p od vzoru q, $d(p,q)$.
3. Z vyhledávání vyřaď všechny prvky o, pro které

   $$|d(q,p) - d(p,o)| > r,$$

   kde r je poloměr vyhledávání.
4. Ke všem zbývajícím prvkům o, je spočtena vzdálenost $d(q,o)$ a jsou vráceny prvky pro něž

   $$d(q,o) \leq r.$$

V tomto případě je již vyžadovaný čas na předzpracování dat $O(mn)$ a vyhledávání vyžaduje čas $m + O(1)$.

Srovnáme-li postup Metrického indexu s algoritmy AESA a LAESA, zjistíme, že používají stejnou metodu pro dolní odhad vzdálenosti dvou prvků, pomocí

kterého odfiltrovávají příliš vzdálené prvky, pravidlo *object-pivot distance*.
AESA a LAESA uchovávají v matici vzdáleností vzdálenosti všech prvků mezi sebou nebo všech prvků od pivotů. Metrický index uchovává pouze vzdálenosti prvků k jejich nejbližšímu pivotu, navíc od AESA a LAESA ale vyžaduje prostor pro uchování své struktury. V této struktuře uchovává aktuální podobu rozdělení prostoru, v kterém jsou prvky vyhledávány. Pomocí pravidla *double-pivot distance* pak určuje, v kterých částech prostoru má smysl vyhledávat. V těchto částech prostoru navíc vyhledává způsobem efektivnějším než AESA a LAESA v celém prostoru. Prvky uvnitř prohledávaného podprostoru jsou totiž uloženy setříděné podle vzdálenosti od jednoho pivota. Metrický index pak přistupuje pouze k prvkům z relevantního intervalu, určeného pomocí pravidla *object-pivot distance*, vzdáleností od tohoto pivota.

7.4 GNAT

Geometric Near-neighbour Access Tree [10] patří mezi takzvané hybridní indexační metody. Kombinuje totiž předpočítané hodnoty s dělením prostoru v hierarchické struktuře. Díky tomu, že nevyžaduje takové množství uložených předpočítaných dat, nemá již tak velké prostorové nároky jako metody používající pouze předpočítaných vzdáleností (AESA, LAESA).
Každý vnitřní uzel GNAT k dělení množiny používá m pivotů. Na začátku je dána indexovaná množina X, která je rozdělena do podmnožin $S_1, \ldots, S_m$, podle toho k jakému pivotu jsou prvky nejblíže. Toto dělení je na množinu aplikováno rekurzivně a je postaven m-ární strom. Ve skutečnosti se počet pivotů a potomků uzlu mění podle hloubky, v které uzel je.
GNAT také ukládá vzdálenosti prvků k jejich nejbližšímu pivotu, což umožňuje prořezávání během vyhledávání. Ke každému vnitřnímu uzlu je také uložena tabulka $m \times m$ s intervaly $[r_l, r_h]$ vzdáleností. Tabulka obsahuje prvky $[r_l^{ij}, r_h^{ij}], i, j = 1, \ldots, m$.

$$r_l^{ij} = min_{o \in S_j \cup \{p_j\}} d(p_i, o),$$

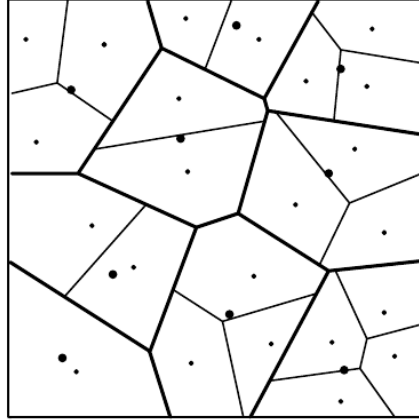
$$r_h^{ij} = max_{o \in S_j \cup \{p_j\}} d(p_i, o).$$

S_j je podmnožina příslušející k pivotu p_j, p_i jsou pivoty z množiny m pivotů. Horní a dolní meze intervalů jsou vlastně horní a dolní meze vzdáleností prvků příslušejících k pivotu p_j od ostatních pivotů p_i.
Možné rozdělení metrického prostoru při konstrukci GNAT je na obrázku 7.2.

7.4.1 Hledání sousedů do určité vzdálenosti $R(q, r)$

Při vyhledávání je v GNAT postupováno prohledáváním do hloubky od kořene stromu. V každém vnitřním uzlu N je spočtena vzdálenost mezi vzorem q a pivoty v N a podstromy, které podle intervalu vzdáleností neobsahují relevantní prvky, jsou vynechány. Pak algoritmus projde všechny podstromy, které zůstaly.

1. P je množina všech pivotů použitá k dělení aktuálního vnitřního uzlu.



Obrázek 7.2: GNAT [10]

2. Vyber prvek p_j z P (ne dvakrát ten stejný), spočti $d(q, p_j)$, pokud $d(q, p_j) \leq r$, zařaď p_j do výsledku.

3. Ke každému $p_i \in P, i \neq j$ zjisti průnik

$$[d(q,p) - r, d(q,p) + r] \cap [r_l^{ij}, r_h^{ij}],$$

pokud je průnik prázdný, pak odstraň p_i z P.

4. Opakuj kroky 2. a 3., než jsou použity všechny prvky z P.

5. Ke všem zbývající $p_i \in P$ rekurzivně projdi potomky k nim příslušející.

Prostorová náročnost GNAT je $O(nm^2)$, protože GNAT ukládá tabulku velikosti m^2 ke každému vnitřnímu uzlu [3]. Složitost vyhledávání nebyla analyzována, ale podle experimentů v [10] je algoritmus efektivnější než například VPT.

Struktura Metrického indexu a GNAT je podobná. Algoritmy se liší ve výběru pivotů, které dělí stávající podmnožiny indexovaných dat, GNAT vybírá stále nové pivoty z této podmnožiny, zatímco Metrický index používá stále stejné pivoty již jednou vybrané ze všech indexovaných prvků. GNAT pak ukládá intervaly vzdáleností prvků v podstromě od ostatních pivotů. Metrický index tento interval ukládá pouze pro listové uzly a jen interval vzdáleností od k nim nejbližšího pivota. Vyhledávání pomocí GNAT je již poměrně podobné vyhledávání v Metrickém indexu. Vzhledem k ukládaným datům, může Metrický index používat pravidlo *double-pivot distance* pro vynechání podstromu, zatímco GNAT může použít prořezávání pomocí znalosti intervalů vzdáleností prvků v podstromě ke všem pivotům v uzlu.
Vyhledávání pomocí Metrického indexu je efektivnější než pomocí GNAT [1].

7.5 MESSIF

Metric Similarity Search Implementation Framework [7] je framework v jazyce Java, který umožňuje vytváření indexů založených na metrických prostorech. Skládá se z několika modulů, z kterých lze sestavit index, který umí podobnostní vyhledávání v metrických prostorech i podle více různých vlastností, sbírá statistické

informace důležité pro výběr indexační metody a umí fungovat i distribuovaně. Tento framework by také měl být jednoduše rozšiřitelný o další moduly.

7.5.1 Metrické prostory

MESSIF byl navržen pro práci s obecnými metrickými prostory. Vnitřní struktura objektů je schována a nikde se nevyužívá. Jen každá třída objektů musí obsahovat implementaci funkce vzdálenosti aplikovatelnou právě na data této třídy. Kvůli snazšímu adresování je každému objektu přiřazeno unikátní `OID`. Pro případ, že by ukládaná data byla zjednodušením původních dat (jako vyextrahovaná strukturovaná data v naší verzi Metric Indexu), objekt obsahuje také `URI` původního objektu. Moduly vytvořené v MESSIF jsou

- `Vectors` pro vektory čísel,
- `Strings` pro řetězce proměnlivé délky,

obojí s implementovanými různými funkcemi pro výpočet vzdálenosti.

7.5.2 Kolekce a dotazy

Kolekce prvků, která je indexována se může měnit, může být zvětšována přidáváním prvků do kolekce, či zmenšována mazáním prvků z kolekce. Hlavním úkolem je pak vyhodnocovat dotazy nad takovouto měnící se databází.
Stejně jako náš index MESSIF podporuje podobnostní vyhledávání a dotazy $R(q, r)$ a $kNN(q, k)$. Moduly vytvořené v MESSIF jsou

- `Insert operation` pro vkládání prvků do databáze,
- `Delete operation` pro mazání prvku z databáze,
- `Range query operation` pro vyhledání sekvenčním průchodem všech prvků, které jsou do dané vzdálenosti od vzorového prvku,
- `kNN query operation` pro vyhledání sekvenčním průchodem všech prvků, které jsou k nejbližšími prvky ke vzorovému prvku,
- `Incremental kNN query operation` pro vyhledávání nejbližších sousedů opět sekvenčním způsobem, při každém zavolání vrací další prvky nejbližší k vzorovému prvku, jen vynechává prvky již vrácené.

7.5.3 Správa dat

Kolekce je podmnožina domény metrického prostoru $X \subseteq D$. Jednotlivé kolekce bývají ukládány do přihrádek, takzvaných `bucketů`. Každá přihrádka je adresována přes identifikátor `BID`.
Přihrádky podporují vykonávání jednotlivých operací, vložení jednoho či více prvků, jejich mazání, výběr všech prvků z přihrádky či jednoho konkrétního. Také mají metody pro vyhodnocování dotazů. Tyto metody sekvenčně procházejí celou přihrádku a vrací vyhovující prvky. Přihrádky mívají omezenou kapacitu, proto obsahují i podporu pro své dělení.
Každá přihrádka také spolupracuje při sběru statistik.
Moduly vytvořené v MESSIF pro ukládání dat jsou

- `Main memory bucket` pracující v paměti a implementovaný jako spojový seznam prvků,
- `Disk storage bucket` ukládající objekty perzistentně na disk.

Moduly vytvořené v MESSIF pro podporu dělení prostoru jsou

- `Random pivot chooser` pro náhodný výběr pivota z kolekce prvků,
- `Incremental pivot chooser` používající kritérium efektivnosti, pomocí kterého vybere ze dvou množin pivotů, tu lépe ohodnocenou,
- `On-fly pivot chooser`, který si pamatuje vkládané prvky, které by mohly být nejlepšími pivoty.

K rozdělenému prostoru je také nutné držet aktuální podobu struktury vnitřních uzlů navigujících k jednotlivým listům, přihrádkám. Moduly vytvořené v MESSIF jsou

- `M-Tree`,
- `PM-Tree`,
- `D-Index`,
- `aD-Index` a
- `VPT`.

Pro sběr statistik byly v MESSIF vytvořeny tyto moduly

- `Query operation statistics` pro sběr dat jako potřebný čas na zpracování odpovědi, množství počítaných vzdáleností mezi prvky, počet navštívených přihrádek či množství vrácených dat,
- `Bucket statistics` pro počet vkládání, mazání a přístupů v přihrádce a celková statistika dotazů vyhodnocovaných v dané přihrádce,
- `Algorithm statistics` pro sběr celkových statistik ze všech přihrádek a operací v těchto přihrádkách.

7.5.4 Distribuované datové struktury

K zvětšení možného množství uložených dat a rozdělení výpočetních nároků MESSIF podporuje i distribuované uložení datové struktury. Navíc pak výpočet v jednotlivých přihrádkách může probíhat paralelně.
K potřebám distribuovaného fungování MESSIF obsahuje moduly pro síťovou komunikaci a vyvažování výpočtové zátěže.

7.5.5 Podobnostní vyhledávání přes více vlastností

MESSIF také umožňuje podobnostně vyhledávat i přes více různých vlastností, neboli agregací více podobnostních funkcí.
Nechť D^m je doména objektů s více vlastnostmi, $o = (o^1, o^2, \ldots, o^m)$, $o^1 \in D^1$, $o^2 \in D^2 \ldots, o^m \in D^m$ jsou objekty. Když d^i jsou metriky, dvojice (D^i, d^i), $i = 1, \ldots, m$ jsou metrické prostory. Výpočet celkové podobnosti dvou objektů musí uživatel definovat $m-$ární agregační funkci t, která musí být monotónní a jejím výsledkem musí být nezáporné reálné číslo. Pro daný objekt q s vlastnostmi $q = (q^1, \ldots, q^m)$ se pak celková podobnost s objektem o spočítá jako $t(d^1(q^1, o^1), d^1(q^2, o^2), \ldots d^m(q^m, o^m))$.

7.5.6 Uživatelské rozhraní

Uživatelské rozhraní nabízí nastavení jednotlivých parametrů, jako jsou například velikosti datových struktur, dále nabízí výběr z operací, které jsou podporované zvoleným algoritmem či zobrazení nasbíraných statistik.
MESSIF nabízí celou škálu uživatelských rozhraní, od příkazových řádků po okenní aplikace zobrazující statistiky v podobě grafů.

MESSIF je framework v jazyce Java, který se dá dále rozšiřovat. Mohli bychom pomocí něj například sestavit i implementaci Metrického indexu, stejnou jako jsme začlenili do Oracle.
MESSIF obsahuje moduly například i pro indexaci různých datových struktur, uživatelská rozhraní či správu dat. Naše implementace Metrického indexu je data cartridge do databázového serveru Oracle, který rozšiřuje o možnost indexace dat typu `VARCHAR2`, `BLOB` a `CLOB`. Uživatelské rozhraní je pak tvořeno rozhraním, kterým je přistupováno k databázovému serveru. Stejně tak i správu dat, jejich perzistentní uložení, má naše implementace zprostředkované databázovým serverem.
MESSIF i naše implementace Metrického indexu umožňují indexaci obecných metrických prostorů stejným způsobem. K interpretaci dat pak oba vyžadují implementaci funkce pro výpočet podobnosti prvků.
Výhodou MESSIF je možnost distribuovaného uložení dat a paralelního vyhledávání. Metrický index byl navržen tak, aby mohl fungovat distribuovaně, naše implementace toho však nevyužívá.

8. Měření výkonu

V této kapitole uvedeme, jakým způsobem jsme měřili náročnost operací naší implementace Metrického indexu a výsledky našeho měření.

8.1 Použitý hardware

Konfigurace systému na kterém byla implementace vyvýjena a testována je uvedena v tabulce 8.1.

8.2 Jednotka složitosti operací

Předpokládá se, že nejnáročnější operací při vyhledávání v mnohodimenzionálních datech bude výpočet podobnosti dvou prvků. Jakékoliv jiné operace, i vstupně-výstupní operace, jsou považovány za méně nákladné a při udávání složitosti algoritmů jsou zanedbávány.
Všechny algoritmy, jak pro vytvoření indexu, tak i pro vyhledávání v něm, proto byly optimalizovány tak, aby vyžadovaly co nejmenší počet vyhodnocení podobnostní funkce. Nákladnost provedené operace budeme proto uvádět právě v počtu provedených vyhodnocení podobnostní funkce.

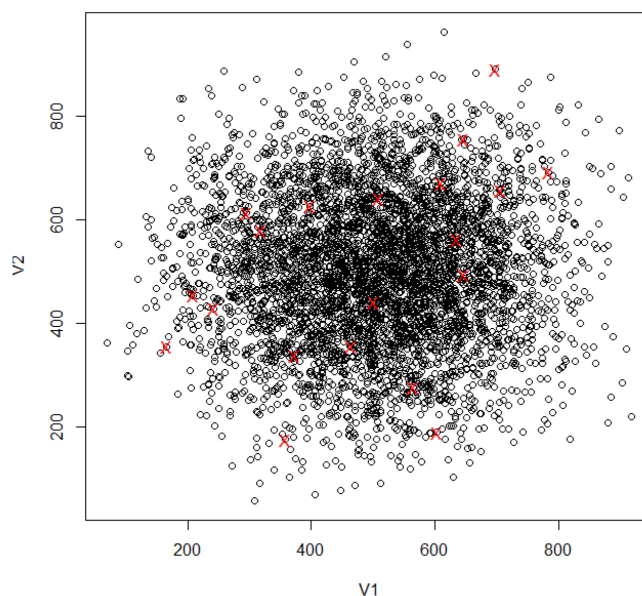
Typicky by práce s naším indexem měla vypadat tak, že je index vytvořen nad sloupcem tabulky, která již obsahuje velkou část všech prvků, v kterých bude vyhledáváno, tvořící co do rozložení reprezentativní vzorek. Následně jsou prováděny již jen menší modifikace kolekce dat a převažuje vyhledávání v indexu. Z tohoto důvodu se budeme nejvíce věnovat části týkající se náročnosti vyhledávání.

8.3 Použitá data

Abychom mohli jednoduše znázornit vygenerovaná data použitá k testování a ukázat na nich závislost nákladnosti operace na zvoleném vzoru, poloměru a množství prvků vybraných dotazem, byla vygenerována data z dvojrozměrného intervalu.

Verze Oracle	10.2.0.1.0
Operační systém	Windows 7 Ultimate SP1
Procesor	Intel Core2 Quad 2.33GHz
Paměť	4GB RAM

Tabulka 8.1: Použitý hardware



Obrázek 8.1: Normální rozdělení dat

8.3.1 Normální rozdělení

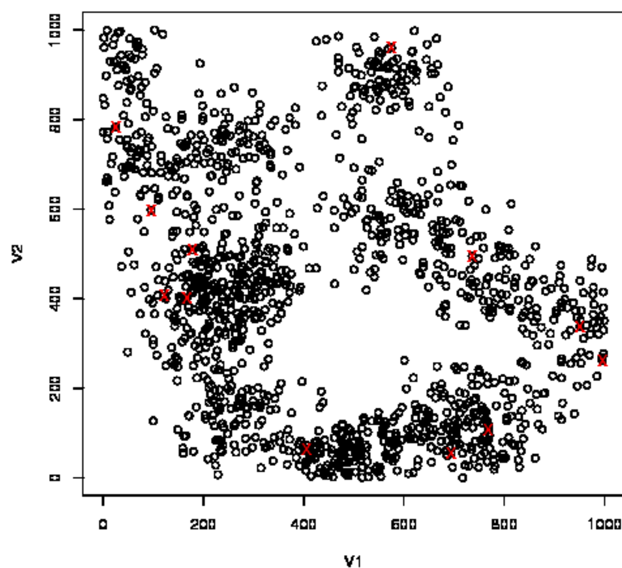
Jedna sada vygenerovaných dat byla generována z normálního rozdělení.
Tato data jsou zobrazena na obrázku 8.1. Křížkem jsou v obrázku znázorněny pivoty vybrané algoritmem incremental selection.
V tomto případě je velikost dat 1348 prvků, prostor je dělen mezi 15 pivotů, maximální velikost shluku je 100 prvků a maximální počet úrovní ve stromu shluků je 5.

8.3.2 Upravené rovnoměrné rozdělení

Další sadou testovaných dat byla data generovaná tak, aby tvořila shluky. Z rovnoměrného rozdělení byla vygenerována centra shluků, do jejich okolí pak byly pomocí normálního rozdělení dogenerovány prvky. Tato data by se měla více blížit reálným datům, která také nebývají zcela rovnoměrně rozdělená.
Tato data jsou znázorněna na obrázku 8.2.
Vygenerováno bylo 5000 prvků, prostor byl dělen mezi 20 pivotů, maximální velikost shluku byla nastavena na 200 prvků.

8.4 Vytvoření indexu

Nejnákladnějšími částmi operace vytvoření indexu jsou nalezení pivotů a následné vložení všech prvků do vytvořené struktury s případnou reorganizací struktury při přeplnění a dělení jednotlivých částí.



Obrázek 8.2: Upravené normální rozdělení dat

8.4.1 Nalezení pivotů

Vhodnost zvolených pivotů je vyhodnocována na základě kritéria efektivnosti 2.5.1 na sadě A náhodně zvolených párů prvků. V každém kroku je vybrán jeden z N kandidátů na dalšího pivota. Ke každému kandidátu musí být tedy spočtena efektivnost množiny pivotů vzniklé jeho přidáním mezi ně. Ze vzorce 2.7 je zřejmé, že

$$D_{\{p_1,\ldots,p_i\}}([a_r],[a'_r]) = max(D_{\{p_1,\ldots,p_{i-1}\}}([a_r],[a'_r]), D_{\{p_i\}}([a_r],[a'_r])), 1 \leq r \leq A. \tag{8.1}$$

V každém kroku je tedy potřeba $2NA$ výpočtů podobnostní funkce, pro výběr k pivotů to celkově znamená $2kAN$ vyhodnocení.
Nákladnost výběru pivotů se tedy při stejném počtu pivotů, kandidátů a párů nemění.

8.4.2 Vložení všech prvků

Některé metody si ke každému vloženému prvku ukládají i jeho vzdálenosti od jednotlivých pivotů, což vyžaduje prostor navíc. Ale při reorganizaci struktury, kterou se myslí rozdělování shluku do více menších, již není nutné znovu počítat vzdálenosti od pivotů. V tomto případě by nákladnost této operace byla při stejném počtu pivotů opět neměnná.
V našem řešení se tyto vzdálenosti k prvkům neukládají. Jedinými hodnotami vzdálenosti dvou prvků, které jsou v našem metrickém indexu ukládány, jsou vzdálenosti prvků od jejich nejbližšího pivota. Tyto hodnoty jsou součástí klíče prvku ve stromě shluků.

8.5 Vyhledávání

Požadavek na vyhledání prvků v našem indexu může být buď $kNN(q,k)$ na k nejbližších prvků k nějakému vzoru, či $R(q,r)$ na všechny prvky do určité vzdálenosti od vzoru (viz 1.2 a 3). K oběma případům uvedeme nákladnost jejich vyhodnocení v příkladu dat z normálního rozdělení a z upraveného rovnoměrného rozdělení. V každém grafu jsou křivky odpovídající jinému vzorovému prvku. Pro normální rozdělení byly vybrány vzorové prvky

- uprostřed shluku, kde je nejvíce prvků (prvek [500, 500], v grafu `cs`),
- na kraji shluku (prvek([350, 600]) v grafu `ks`)
- a v prostoru s řídkým výskytem prvků (prvek [200, 200], v grafu `v`).

Pro upravené rovnoměrné rozdělení byly vybrány vzorové prvky

- uprostřed většího shluku prvků v blízkosti pivota (prvek [400, 200], v grafu `cs`),
- v prostoru mezi více shluky (prvek [420, 600], v grafu `ms`)
- a v prostoru s řídkým výskytem prvků (prvek [800, 900], v grafu `p`).

8.5.1 Dotaz R(q,r)

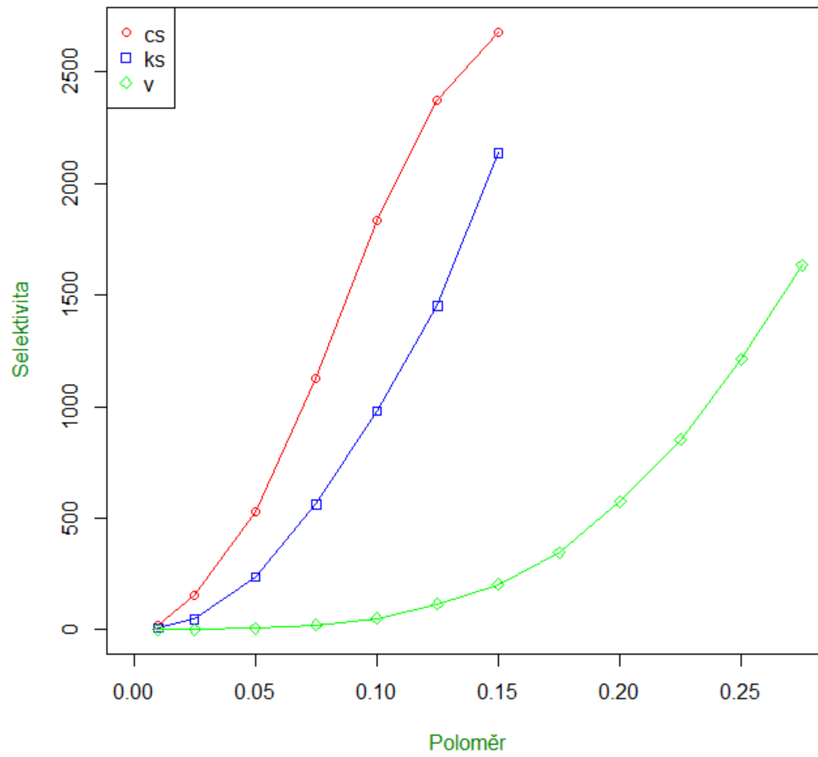
Naměřená data k dotazu $R(q,r)$ jsou v tabulce 8.2.

Závislost selektivity na poloměru vyhledávání

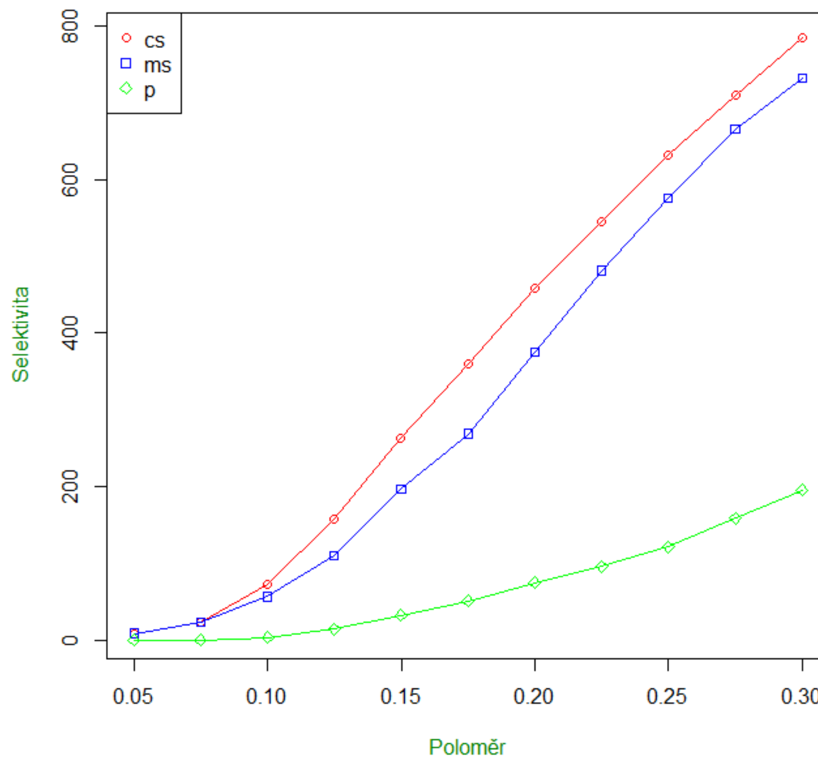
Prvním grafem pro dotaz $R(q,r)$ je graf závislosti počtu prvků vybraných dotazem na poloměru dotazu 8.3. Nevyjadřuje sice přímo nákladnost vykonání dotazu, ale naznačuje správnost vyhodnocení. V okolí vzorových prvků, které kolem sebe mají více prvků, je ve výsledku skutečně i více prvků vráceno. A při zvětšování poloměru vyhledávání se skutečně počet vrácených prvků zvyšuje.
V grafu pro normální rozdělení je vidět, že pro prvek `cs`, který leží v centru shluku, je již pro malé poloměry vráceno hodně prvků. Pro prvek `ms`, který se nachází na kraji shluku, je při malém poloměru nalezeno méně prvků, ale jejich počet se čím dál rychleji zvyšuje i s tím, jak se poloměrem vyhledávání přibližujeme k centru shluku. Vzoru `v`, který je v prostoru, kde nemá v blízkosti žádné prvky, přibývá vrácených prvků pomaleji, až s poloměrem, který více dosahuje k centru vygenerovaného shluku, se počet vrácených prvků zvyšuje rychleji.
Stejně je tomu i v grafu pro upravené rovnoměrné rozdělení se vzorovými prvky `cs` v prostoru s hodně prvky a v blízkosti pivota, `ms` v prostoru s méně prvky a mezi více shluky a vzor `p`, který leží v prostoru, kde nemá blízké prvky.

Závislost počtu vyhodnocení podobnostní funkce na selektivitě

Grafy 8.4 zobrazují závislost počtu vyhodnocení podobnostní funkce na počtu prvků vybraných dotazem.
Naměřená data ukazují, že s rostoucím poloměrem a rostoucím počtem prvků,



(a) Normální rozdělení



(b) Upravené rovnoměrné rozdělení

Obrázek 8.3: Závislost počtu vybraných prvků na poloměru dotazu

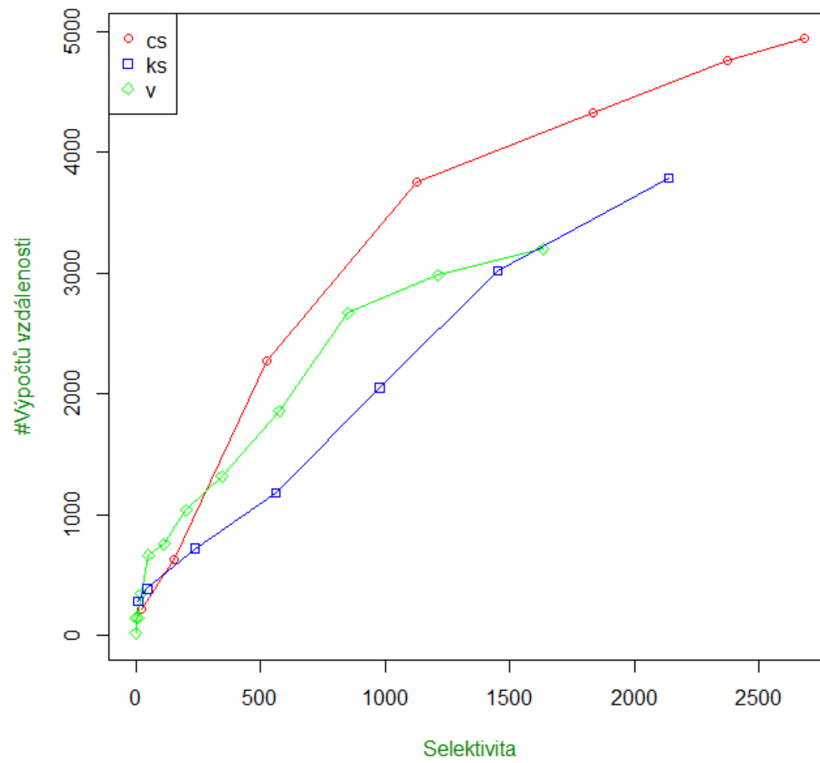
které odpovídají podmínce dotazu, roste i počet shluků, v kterých jsou prvky vyhledávány. Čím dál méně shluků je z vyhledávání vyloučeno pomocí pravidla *double-pivot distance* a čím dál méně prvků je při vyhledávání ve shlucích vyloučeno pomocí pravidla *object-pivot distance*.
V případě normálního rozdělení křivka pro vzorový prvek v centru prostoru *cs* ukazuje ze začátku menší nákladnost operace, která se postupně vůči zbylým vzorům zvyšuje. Je to tím, že pro vybrání malého počtu dat je zapotřebí malého poloměru dotazu, který pak dosáhne k menšímu počtu shluků. S rostoucím počtem prvků a rostoucím poloměrem dosáhne dotaz, díky poloze vzorového prvku, k více shlukům, ve kterých je potom nutné vyhledávat.
Porovnáme-li křivky `ks` pro vzor na kraji vygenerovaného shluku a *v* pro vzor vně vygenerovaného shluku, vidíme, že pro vybrání stejného množství prvků, potřebuje *v* větší poloměr, a následně tedy prohledání více shluků. To se ale změní ve chvíli, kdy vzorový prvek u kraje vygenerovaného shluku, díky tomu, že je blíže středu prostoru, dosáhne při zvyšujícím se poloměru vyhledávání do více shluků, od jejihž prvků je pak nutné počítat jeho vzdálenost.
V případě dat generovaných z upraveného normálního rozdělení je tomu podobně. Vzorovému prvku uvnitř oblasti s hodně prvky a v blízkosti pivota *cs* pro vyhledání nejbližších prvků stačil pouze malý poloměr, který nedosáhl k dalším shlukům. Další vývoj byl již velmi podobný s prvkem mezi více shluky *ms*, jejich poloha v prostoru se totiž moc neliší a oba brzy dosáhnou k dalším shluků a prvkům v nich. Opačně je na tom vzorový prvek *p*, který leží na okraji prostoru a nedosáhne tedy ani při větším poloměru k tolika shlukům. Pomocí tohoto vzorového prvku tedy není vybráno tolik prvků, ani nedojde k tolika výpočtům podobnosti, neboť většina shluků je vyloučena již pomocí pravidla *double-pivot distance*.

8.5.2 Dotaz kNN(q,k)

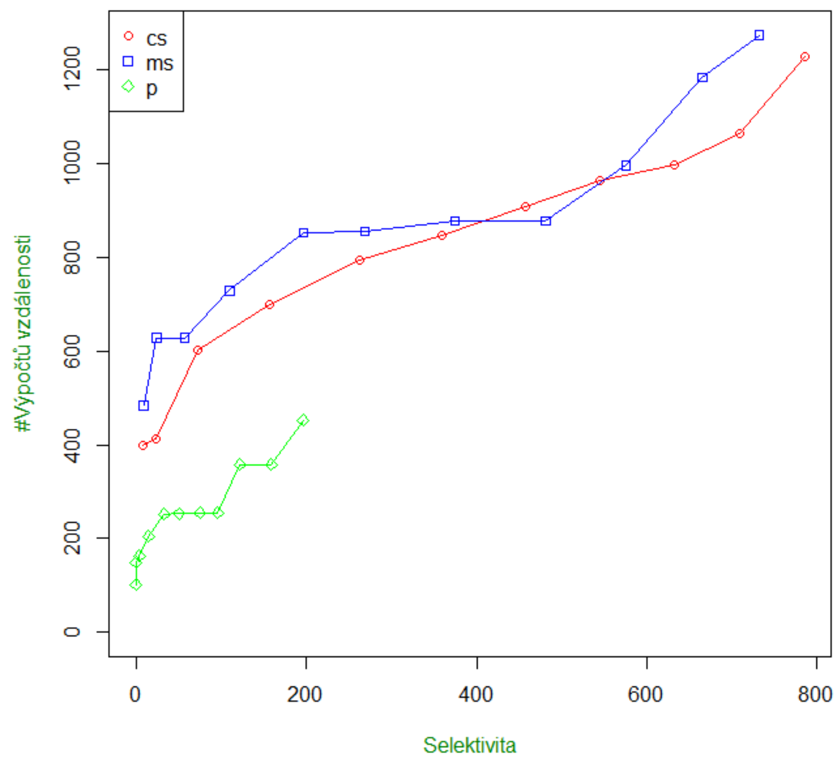
Naměřené hodnoty k dotazu $kNN(q,k)$ jsou v tabulce 8.3.
Grafy 8.5 zobrazují závislost vyhodnocení podobnostní funkce na požadovaném počtu k nalezených nejbližších prvků.
Z grafů je vidět, že množství potřebných výpočtů podobnostní funkce pro jednotlivé vzorové prvky se příliš neliší. Ve všech případech je zvolen dostatečně velký poloměr, ve kterém jsou prohledávány shluky.

8.5.3 Vliv počtu pivotů na efektivitu vyhledávání

Se zvyšujícím se počtem pivotů by se mělo vylepšovat i rozdělení prvků do jednotlivých shluků. Tato vlastnost byla testována na datech generovaných z upraveného rovnoměrného rozdělení. Bylo vygenerováno 30 shluků po zhruba 200 prvcích. Celkový počet prvků byl 5458.
Test byl proveden pro různý počet pivotů na jednom vzorovém prvku, měněny byly poloměr vyhledávání r a počet vyhledávaných nejbližších sousedů k.
Pro vyhledávání k nejbližších sousedů se skutečně počet vyhodnocení podobnostní funkce se zvyšujícím se počtem pivotů snižuje, u hledání všech sousedů do určité vzdálenosti tento trend viditelný není. Výsledky testů jsou na obrázku 8.6 a v tabulkách 8.4 a 8.5.



(a) Normální rozdělení



(b) Upravené rovnoměrné rozdělení

Obrázek 8.4: Závislost počtu vyhodnocení podobnostní funkce na počtu vybraných prvků

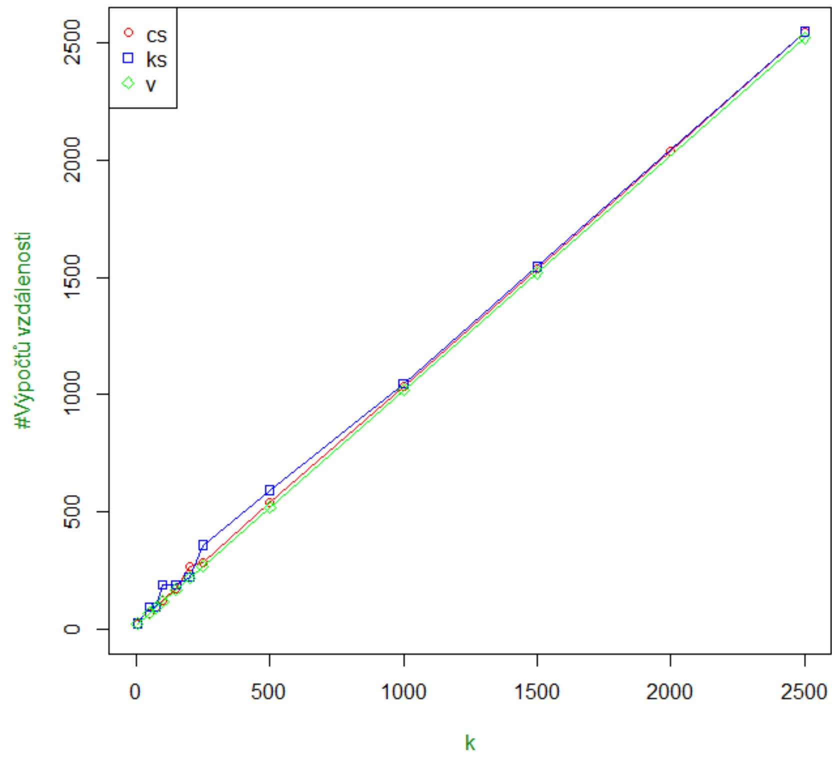
Normální rozdělení		
prvek [500, 500], cs		
poloměr	selektivita	#vyhodnocení
0.01	20	221
0.025	154	631
0.05	527	2273
0.075	1125	3752
0.1	1832	4330
0.125	2374	4756
0.15	2681	4949
0.175	2907	5020
prvek [350, 600], ks		
poloměr	selektivita	#vyhodnocení
0.01	9	287
0.025	46	387
0.05	237	721
0.075	561	1183
0.1	979	2052
0.125	1452	3018
0.15	2137	3787
prvek [200, 200], v		
poloměr	selektivita	#vyhodnocení
0.01	0	20
0.025	1	147
0.05	7	147
0.075	21	345
0.1	49	663
0.125	114	759
0.15	203	1038
0.175	345	1320
0.2	575	1861
0.225	852	2669
0.25	1212	2984
0.275	1633	3199

(a) Normální rozdělení

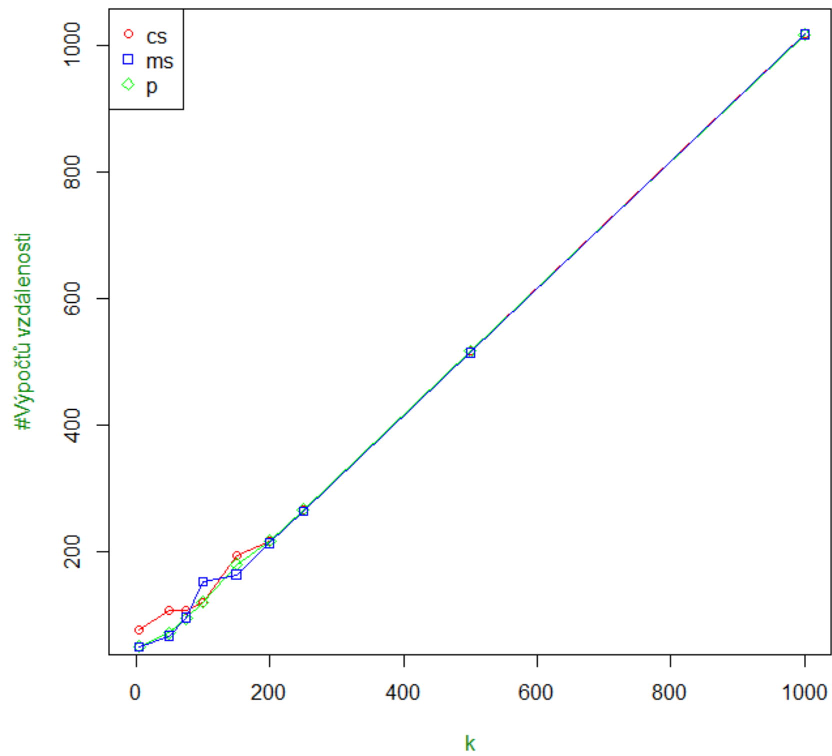
Upravené rovnoměrné rozdělení		
prvek [400, 200], cs		
poloměr	selektivita	#vyhodnocení
0.05	8	399
0.075	24	414
0.1	73	603
0.125	157	700
0.15	263	795
0.175	359	847
0.2	458	907
0.225	545	964
0.25	632	996
0.275	709	1065
0.3	785	1228
prvek [420, 600], ms		
poloměr	selektivita	#vyhodnocení
0.05	9	484
0.075	24	628
0.1	57	628
0.125	110	729
0.15	197	853
0.175	269	855
0.2	375	877
0.225	481	877
0.25	575	996
0.275	665	1183
0.3	732	1274
prvek [800, 900], p		
poloměr	selektivita	#vyhodnocení
0.05	0	101
0.075	0	149
0.1	4	163
0.125	15	205
0.15	33	252
0.175	51	253
0.2	75	255
0.225	96	255
0.25	122	358
0.275	159	358
0.3	196	452

(b) Upravené rovnoměrné rozdělení

Tabulka 8.2: Dotaz $R(q, r)$

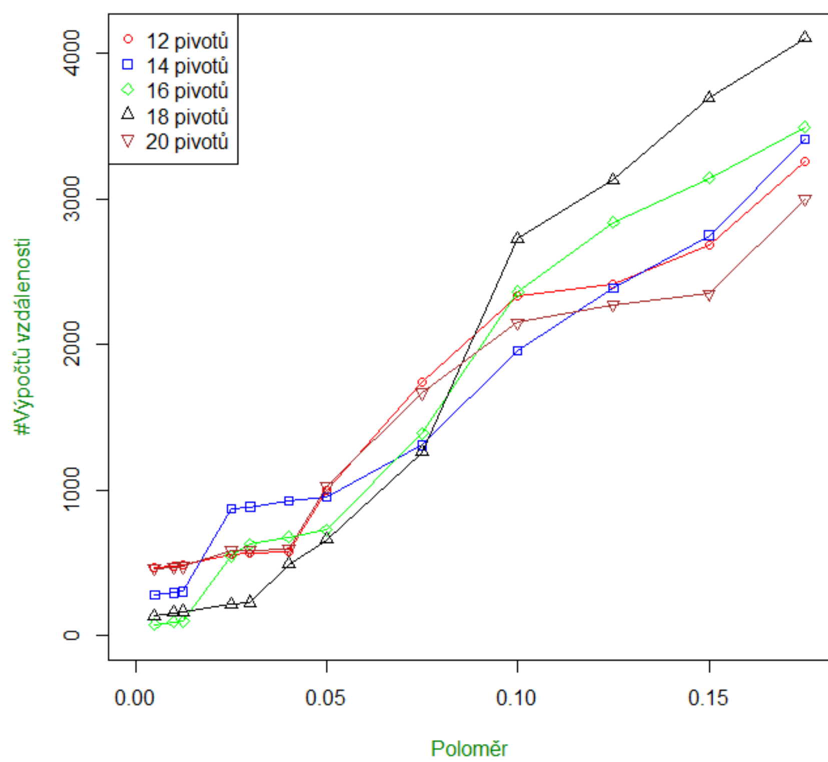


(a) Normální rozdělení

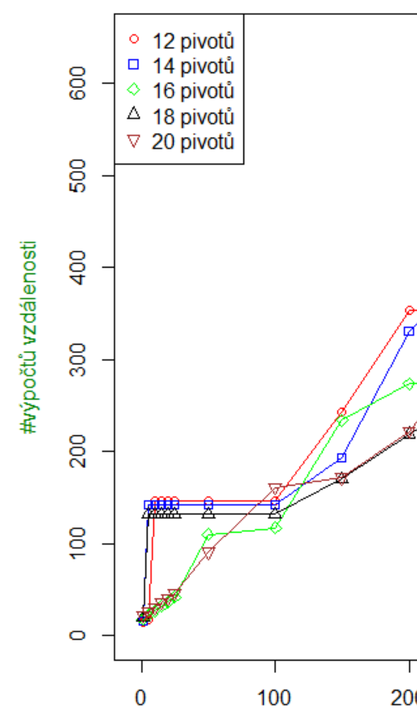


(b) Upravené rovnoměrné rozdělení

Obrázek 8.5: Závislost počtu vyhodnocení podobnostní funkce na počtu vybraných prvků k



(a) $R(q,r)$



(b)

Obrázek 8.6: Závislost počtu vyhodnocení podobnostní funkce na počtu pivotů

8.6 Modifikace kolekce dat

8.6.1 Vkládání

V případě vkládání prvků je vždy nalezen shluk, v kterém by měl daný prvek ležet, to vyžaduje vyhodnocení podobnostní funkce právě tolikrát, kolik je pivotů. K dalšímu vyhodnocování podobnostní funkce dojde jen v případě, že přidáním prvku dojde k překročení maximálního počtu prvků ve shluku. V takovém případě jsou spočteny vzdálenosti všech prvků v tomto shluku od pivotů, které ještě nebyly použity k dělení prostoru na jejich podprostor.

8.6.2 Mazání

Při mazání prvků k žádnému dělení ani slučování prostoru nedochází, stačí nám pouze najít shluk, v kterém by se měl prvek nacházet, což vyžaduje již zmíněný počet vyhodnocení podobnostní funkce stejný, jako je počet pivotů.

Při vkládání nebo mazání velkého množství prvků z kolekce indexovaných dat se nám může stát, že vyhledávání bude méně efektivní. Prostor byl totiž rozdělen pomocí pivotů, které byly vybrány z množiny dat, která se hodně změnila. Po modifikaci indexovaných dat, se nám tedy může vyplatit index vytvořit znovu nad aktuálními daty.

Normální rozdělení	
prvek [500, 500], cs	
k	#vyhodnocení
5	26
50	71
75	96
100	122
150	172
200	266
250	286
500	540
1000	1036
1500	1538
2000	2038
2500	2542
prvek [350, 600], ks	
k	#vyhodnocení
5	26
50	97
75	97
100	192
150	192
200	233
250	360
500	593
1000	1048
1500	1546
2500	2546
prvek [200, 200], v	
k	#vyhodnocení
5	25
50	97
75	97
100	192
150	192
200	223
250	360
500	593
1000	1048
1500	1546
2500	2546

(a) Normální rozdělení

Upravené rovnoměrné rozdělení	
prvek [400, 200], cs	
k	#vyhodnocení
5	76
50	108
75	108
100	119
150	195
200	216
250	266
500	518
1000	1020
prvek [420, 600], ms	
k	#vyhodnocení
5	50
50	67
75	96
100	153
150	164
200	214
250	264
500	515
1000	1018
prvek [800, 900], p	
k	#vyhodnocení
5	50
50	72
75	95
100	120
150	180
200	217
250	267
500	517
1000	1017

(b) Upravené rovnoměrné rozdělení

Tabulka 8.3: Dotaz $kNN(q, k)$

12 pivotů		14 pivotů	16 pivotů	18 pivotů	20 pivotů
k	#vyhodnocení	#vyhodnocení	#vyhodnocení	#vyhodnocení	#vyhodnocení
1	14	16	18	20	22
5	18	142	22	132	26
10	147	142	27	132	31
15	147	142	32	132	36
20	147	142	37	132	41
25	147	142	42	132	46
50	147	142	110	132	91
100	147	142	117	132	161
150	243	193	234	170	172
200	353	331	274	219	222
250	353	398	274	269	319
300	353	398	368	319	382
350	550	398	628	369	382
400	550	484	628	419	423
500	550	538	649	576	523

Tabulka 8.4: Dotaz $kNN(q,k)$

12 pivotů			14 pivotů	16 pivotů	18 pivotů	20 pivotů
poloměr	selektivita	#vyhodnocení	#vyhodnocení	#vyhodnocení	#vyhodnocení	#vyhodnocení
0.005	1	467	282	77	135	461
0.01	5	481	295	96	158	470
0.0125	7	487	304	100	163	475
0.025	30	559	870	546	214	583
0.03	48	569	885	628	228	589
0.04	75	576	924	677	490	598
0.05	123	993	954	726	660	1024
0.075	271	1744	1307	1388	2271	1667
0.1	476	2333	1957	2361	2275	2149
0.125	741	2414	2388	2838	3129	2272
0.15	994	2681	2748	3143	3693	2346
0.175	1324	3253	3407	3492	4102	2997

Tabulka 8.5: Dotaz $R(q, r)$

Závěr

Zadáním práce bylo umožnit indexování vysokorozměrných dat. Jako indexační metodu jsme zvolili Metrický index, tedy metodu pro indexaci a podobnostní vyhledávání v metrických prostorech, která patří mezi nejefektivnější. Metodu jsme implementovali jako data cartridge databázového serveru Oracle.
Použití metody pro indexaci metrických prostorů nám umožňuje vytvářet doménové indexy nad daty různé struktury. Data jsou pak interpretována pomocí funkcí definovaných uživatelem. To přináší možnost nahlížet na data různými pohledy, a to podle požadavků zvolených uživatelem.
Integrace do databázového serveru Oracle nám umožnila implementovat metody rozhraní, tak aby odpovídaly Metrickému indexu. Vše ostatní, jako je vkládání dat do indexované tabulky, jejich perzistentní ukládání či dotazování se nad daty, zprostředkuje databázový server.
Výhodou zvoleného řešení je, že indexy budou integrovány do prostředí databáze a v rámci dotazů mohou být kombinovány s běžnými indexy.
Zvolený způsob implementace umožňuje vyhledávání podle indexu. Implementace funkce svázané s operátorem, která by vyhledávala data bez použití indexu, by byla také technicky možná. Vyžadovala by však přidání dalších parametrů do operátorů a znesnadňovala použití.
Dalším možným zlepšením by bylo efektivnější načítání částí struktury indexu uložených v `BLOB`u v databázi. V současné době jsou tyto části čteny po kusech, které nemusí odpovídat blokům paměti. Také jsou data jednoho shluku načítána vcelku a je deserializován celý B^+strom, ačkoliv by třeba stačila jeho část.

Cílem řešení bylo zajistit efektivní podobnostní vyhledávání v databázi oproti sekvenčnímu procházení celé tabulky spojenému s vyhodnocováním podobnostní funkce pro každý záznam. Výsledky provedených testů prokázaly, že zvolená implementace je vhodná pro reálné použití.
Řešení jako celek zahrnuje analýzu problému, výběr vhodné metody, realizaci kódu data cartridge pro databázový server Oracle, testování s vyhodnocením výsledků, zpracování dokumentace a instalačního balíčku pro správce databáze.

Seznam použité literatury

[1] Novak, David, Batko, Michal, Zezula, Pavel. *Metric Index: An efficient and scalable solution for precise and approximate similarity search*, Elsevier, 2010.

[2] Jagadish, H. V., Ooi, B. C., Tan, K.-L., Yu, C., Zhang, R., *iDistance: An adaptive B^+-tree based indexing method for nearest neighbor search*, ACM Transactions on Databse Systems, 20YY, 1-34.

[3] Zezula, P., Amato, G., Dohnal, V., Batko, M. *The Metric Space Approach*, ISBN-13: 978-0387291468, 1.vydání, 2006.

[4] *Oracle Database Data Cartridge Developer's Guide*, [online], citováno 5.7.2012, url: http://docs.oracle.com/cd/B19306_01/appdev.102/b14289/toc.htm.

[5] Bustos, B., Navvaro, G., Chávez, E. *Pivot Selection Techniques for Proximity Searching in Metric Spaces*, Elsevier Science, 2003.

[6] Pokorný, J., Snášel, V., Kopecký, M., *Dokumentografické Informační systémy*, ISBN 80-246-1148-1 2. vydání. Univerzita Karlova v Praze, 2005.

[7] Batko, M., Novák, D., Zezula, P. *Metric Similarity Search Implementation Framework*, url: http://lsd.fi.muni.cz/trac/messif/, 2007.

[8] Micó, M.L., Oncina, J., Vidal, E. *Approximating and Eliminating Search Algorithm (AESA) with linear preprocessing time and memory requirements.* Elsevier Science B.V., 1994.

[9] Skopal, T. *Slidy k předmětu Vyhledávání multimediálního obsahu na webu.* [online], citováno 5.7.2012, url: http://siret.ms.mff.cuni.cz/members/skopal/home.

[10] Brin, S. *Near Neighbor Search in Large Metric Spaces*. 21th International Conference on Very Large Data Bases Zurich, Switzerland, 1995.

[11] Pokorný, J., *Dotazovací Jazyky*, ISBN 978-80-246-0497-8, Univerzita Karlova v Praze, 2007.

[12] Pokorný, J., Žemlička, M., *Základy implementace souborů a databází*, ISBN 80-246-0837-5, Univerzita Karlova v Praze, 2004.

[13] Koubková, A., Koubek, V., *Datové Struktury I*, ISBN 978-80-7378-166-8, Univerzita Karlova v Praze, 2011.

[14] Hoksza, D., Diplomová práce: *Vícerozměrné indexování pro relační SŘBD* Univerzita Karlova v Praze, 2006.

Seznam obrázků

1.1 object-pivot distance . . . 8
1.2 double-pivot distance . . . 9
1.3 range-pivot distance . . . 9

2.1 iDistance [1] . . . 11
2.2 M-Index s jednou úrovní [1] . . . 11
2.3 M-Index s l = 2 . . . 12
2.4 M-Index s dynamickým počtem úrovní . . . 13
2.5 Strom shluků, M-Index s dynamickým počtem úrovní . . . 15

6.1 Serializovaný strom shluků . . . 30
6.2 Serializovaný B^+strom . . . 30

7.1 VPT [3] . . . 36
7.2 GNAT [10] . . . 40

8.1 Normální rozdělení dat . . . 45
8.2 Upravené normální rozdělení dat . . . 46
8.3 Závislost počtu vybraných prvků na poloměru dotazu . . . 48
8.4 Závislost počtu vyhodnocení podobnostní funkce na počtu vybraných prvků 50
8.5 Závislost počtu vyhodnocení podobnostní funkce na počtu vybraných prvků k 52
8.6 Závislost počtu vyhodnocení podobnostní funkce na počtu pivotů 53

Přílohy

9. Uživatelská příručka

V této kapitole popíšeme použití naší implementace Metrického indexu. Instalace je popsána v souboru **\bin\install\README.txt** na přiloženém CD.
Máme-li již data cartridge s Metrickým indexem nainstalovanou, můžeme vytvářet doménové indexy nad daty typu `VARCHAR2`, `BLOB` či `CLOB`. Do indexované tabulky lze data vkládat, z tabulky je mazat a v datech vyhledávat.

9.1 Vytvoření indexu

Jak bylo popsáno v kapitole 5, uživatel musí vytvořit objektový typ, který bude dědit od objektového typu `mspace`, s vlastní implementací funkcí `fe` a `distance` pro typ, který chce indexovat.
Předpokládejme, že uživatel vytvořil potomka typu `mspace` nazvaný `user_mspace`, jak bylo popsáno v části 5.3.2, vytvořil tabulku, jejíž sloupec bude chtít indexovat pomocí našeho doménového indexu a vložil do ní nějaká data.

```
CREATE TABLE test(
  vector VARCHAR2(8)
);

INSERT INTO test VALUES( '10,20');
INSERT INTO test VALUES( '20,30');
INSERT INTO test VALUES( '80,30');
INSERT INTO test VALUES( '60,50');
INSERT INTO test VALUES( '25,25');
INSERT INTO test VALUES( '40,20');
INSERT INTO test VALUES( '85,90');
```

Pak již stačí znát postup pro vytvoření doménového indexu, třeba jak ukazuje následující příklad.

```
CREATE INDEX mindex_vector ON test(vector) INDEXTYPE IS mindex_idx
PARAMETERS(':mspace user_mspace :pcount 4 :dvolume 10');
```

Parametr `mspace` je nám již znám. Následující parametry `pcount` slouží ke zvolení počtu pivotů, které budou vybrány a bude se podle nich dělit množina indexovaných prvků, a parametr `dvolume`, který určuje maximální velikost dat v jednom shluku.
Typickými problémy, ke kterým může během vytváření indexu dojít, jsou

- příliš malé množství dat v tabulce pro vytvoření indexu,
- příliš mnoho dat či špatné rozložení dat v prostoru, pro zvolenou konfiguraci stromu shluků.

Příliš malé množství dat v tabulce pro vytvoření indexu značí situaci, kdy tabulka obsahuje méně prvků, než je počet pivotů zvolený uživatelem.
Příliš mnoho dat či špatné rozložení dat v prostoru znamená, že přetekla velikost

shluku, u kterého mělo dojít k jeho rozdělení. K rozdělení nedošlo nejčastěji z důvodu, že uzel stromu shluků, obsahující daný shluk, je již v maximální hloubce a další hladina již nemůže být přidána. Méně častou příčinou je zvolený příliš nízký počet pivotů, který již neumožňuje další dělení shluku. Například při existenci pouze dvou pivotů může dojít jen k jednomu dělení, v další úrovni by už nebylo podle čeho dělit.

O těchto situacích se uživatel dozví z chybových hlášení. Chybový kód těchto výjimek je 28400. K vyřešení situace vede zrušení doménového indexu a následný nový pokus o vytvoření indexu třeba s jinou konfigurací parametrů `pcount` či `dvolume`.
Napevno zvolené limity jsou maximální počet úrovní ve stromu shluků, nastavená na 5, a maximální počet pivotů nastavený na 20. Tyto hodnoty byly zvoleny podle doporučení v [1].

9.2 Vkládání a mazání dat

Vkládání dalších hodnot do tabulky se nijak nemění.

```
INSERT INTO test VALUES( '60,80');
```

V tomto případě může dojít k chybě při dělení uzlu a opět k vyvolání výjimek s informací o neprovedení přidání úrovně do stromu shluků. V takovém případě se nám nepovede daný prvek do tabulky vložit.
Řešením pak je index zrušit a vytvořit znovu, případně ho vytvořit opět s jinou konfigurací parametrů.
Mazání prvků z tabulky, jak ukazuje příklad, opět zůstává stejné.

```
DELETE FROM test WHERE vector = '60,80';
```

Při delete žádná předpokládaná výjimka nenastává.

9.3 Vyhledávání

Doménový index vytvořený na základě indexového typu `mindex_idx` podporuje vyhledávání pomocí operátorů `range` a `knn`.

```
SELECT * FROM test WHERE range(vector, TChar(10,10)) <= 0.2;
```

Dotaz vybere všechny řádky z tabulky `test`, které mají ve sloupci `vector` hodnotu, jejíž extrahovaná strukturovaná data jsou od prvku TChar(10,10) do vzdálenosti 0,2.

```
SELECT * FROM test WHERE knn(vector, TChar(10,10)) <= 5;
```

Dotaz vybere řádky z tabulky `test`, které mají ve sloupci `vector` hodnotu, jejíž extrahovaná strukturovaná data jsou mezi 5-ti s nejmenší vzdáleností od prvku `TChar(10,10)` ze všech hodnot sloupce `vector` z tabulky `test`.

Povolenými mezními hodnotami vyhledávání v dotazu s operátorem `range` je desetinné číslo z intervalu $(0, 1]$, v dotazu s operátorem `knn` je to přirozené číslo maximálně o velikosti počtu prvků v tabulce.
Při vyhledávání může dojít k výjimce při zadání nepřípustných mezních hodnot vyhledávání.

9.4 Rušení indexu

Ke zrušení indexu je třeba zavolat příkaz `DROP INDEX`.

```
DROP INDEX mindex_vector;
```