Kapitel 5

Mehrdimensionale Suchstrukturen

5.1 Problemstellung

Ein Sekundär-Index ist in der Lage, alle Records mit $x_1 \leq a \leq x_2$ zu finden. Nun heißt die Aufgabe: Finde alle Records mit $x_1 \leq a_1 \leq x_2$ und $y_1 \leq a_2 \leq y_2, \ldots$

Beispiel für mehrdimensionale Bereichsabfrage:
Gesucht sind alle Personen mit der Eigenschaft

Alter	zwischen 20 und 30 Jahre alt
Einkommen	zwischen 2000 und 3000 DM
PLZ	zwischen 40000 und 50000

Im folgenden betrachten wir (wegen der einfacheren Veranschaulichung) nur den 2-dimensionalen Fall. Diese Technik ist auf beliebige Dimensionen verallgemeinerbar.

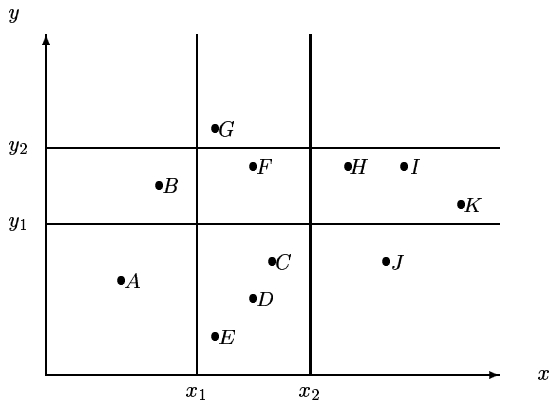

Abbildung 5.1: Fläche mit Datenpunkten

Abbildung 5.1 zeigt eine zweidimensionale Fläche mit Datenpunkten sowie ein Query-Rechteck, gegeben durch vier Geraden.

Die Aufgabe besteht darin, alle Punkte zu ermitteln, die im Rechteck liegen. Hierzu bieten sich zwei naheliegende Möglichkeiten an:

- Projektion durchführen auf x oder y mit binärer Suche über vorhandenen Index, danach sequentiell durchsuchen, d.h. zunächst werden G, F, C, D, E ermittelt, danach bleibt F übrig
- Projektion durchführen auf x und Projektion durchführen auf y, anschließend Durchschnitt bilden.

Es ist offensichtlich, daß trotz kleiner Trefferzahl ggf. lange Laufzeiten auftreten können. Dagegen ist für die 1-dimensionale Suche bekannt: Der Aufwand beträgt O($k + \log n$) bei k Treffern in einem Suchbaum mit n Knoten.

5.2 k-d-Baum

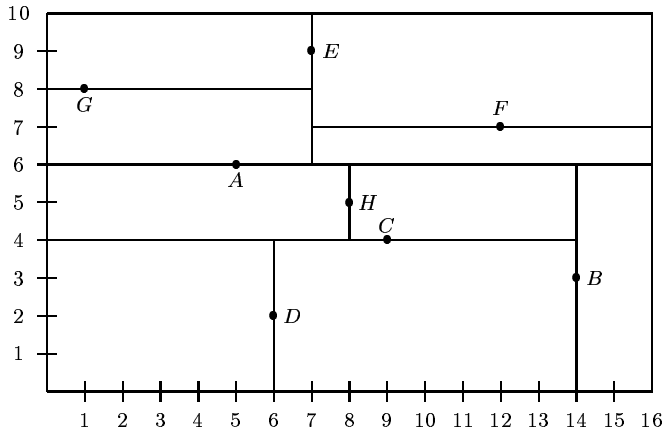

Abbildung 5.2: Durch die Datenpunkte A,B,C,D,E,F,G,H partitionierte Fläche

Eine Verallgemeinerung eines binären Suchbaums mit einem Sortierschlüssel bildet der *k-d-Baum* mit k-dimensionalem Sortierschlüssel. Er verwaltet eine Menge von mehrdimensionalen Datenpunkten, wie z.B. Abbildung 5.2 für den 2-dimensionalen Fall zeigt. In der homogenen Variante enthält jeder Baumknoten ein komplettes Datenrecord und zwei Zeiger auf den linken und rechten Sohn (Abbildung 5.3). In der inhomogenen Variante enhält jeder Baumknoten nur einen Schlüssel und die Blätter verweisen auf die Datenrecords (Abbildung 5.4). In beiden Fällen werden die Werte der einzelnen Attribute abwechselnd auf jeder Ebene des Baumes zur Diskriminierung verwendet. Es handelt sich um eine statische Struktur; die Operationen Löschen und die Durchführung einer Balancierung sind sehr aufwendig.

Im 2-dimensionalen Fall gilt für jeden Knoten mit Schlüssel $[x/y]$:

	im linken Sohn	im rechten Sohn
auf ungerader Ebene	alle Schlüssel $\leq x$	alle Schlüssel $> x$
auf gerader Ebene	alle Schlüssel $\leq y$	alle Schlüssel $> y$

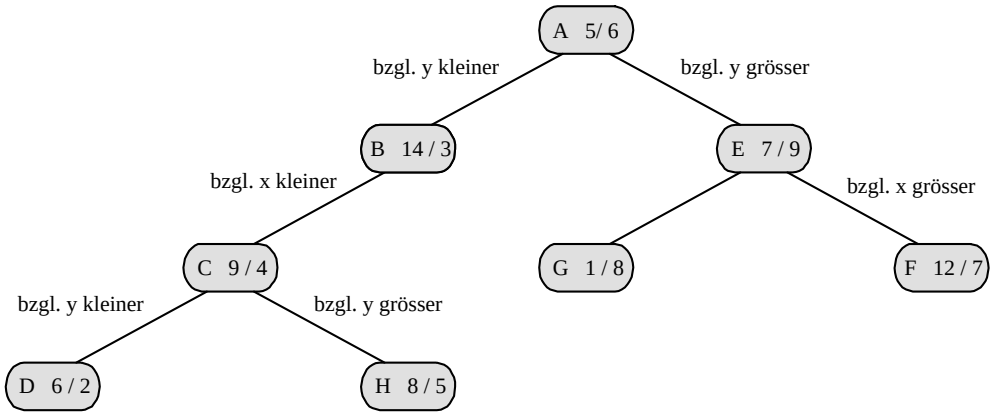

Abbildung 5.3: 2-d-Baum (homogen) zu den Datenpunkten A,B,C,D,E,F,G,H

Die Operationen auf einem $2-d$-Baum laufen analog zum binärem Baum ab:

- **Insert**:
  Suche mit Schlüssel $[x/y]$ unter Abwechslung der Dimension die Stelle, wo der $[x/y]$-Knoten sein müßte und hänge ihn dort ein.
- **Exakt Match** (z.B. finde Record $[15/5]$):
  Suche mit Schlüssel [x/y] unter Abwechslung der Dimension bis zu der Stelle, wo der [x/y]-Knoten sein müßte.
- **Partial Match** (z.B. finde alle Records mit $x = 7$):
  An den Knoten, an denen nicht bzgl. x diskriminiert wird, steige in beide Söhne ab; an den Knoten, an denen bzgl. x diskriminiert wird, steige in den zutreffenden Teilbaum ab.
- **Range-Query** (z.B. finde alle Records $[x, y]$ mit $7 \leq x \leq 13, 5 \leq y \leq 8$):
  An den Knoten, an denen die Diskriminatorlinie das Suchrechteck schneidet, steige in beide Söhne ab, sonst steige in den zutreffenden Sohn ab. Beobachtung: Laufzeit $k + \log n$ Schritte bei k Treffern!
- **Best-Match** (z.B. finde nächstgelegenes Record zu $x = 7, y = 3$):
  Dies entspricht einer Range-Query, wobei statt eines Suchrechtecks jetzt ein Suchkreis mit Radius gemäß Distanzfunktion vorliegt. Während der Baumtraversierung schrumpft der Suchradius. Diese Strategie ist erweiterbar auf k-best-Matches.

Bei der inhomogenen Variante enthalten die inneren Knoten je nach Ebene die Schlüsselinformation der zuständigen Dimension sowie Sohnzeiger auf weitere innere Knoten. Nur die Blätter verweisen auf Datenblöcke der Hauptdatei, die jeweils mehrere Datenrecords aufnehmen können. Auch die inneren Knoten werden zu Blöcken zusammengefaßt, wie auf Abbildung 5.5 zu sehen ist. In Abbildung 5.4 befinden sich z.B. die Datenrecords *C, B* und *D* in einem Block.

Abbildung 5.6 zeigt, daß neben der oben beschriebenen 2-d-Baum-Strategie eine weitere Möglichkeit existiert, den Datenraum zu partitionieren. Dies führt zu den sogenannten *Gitterverfahren.*

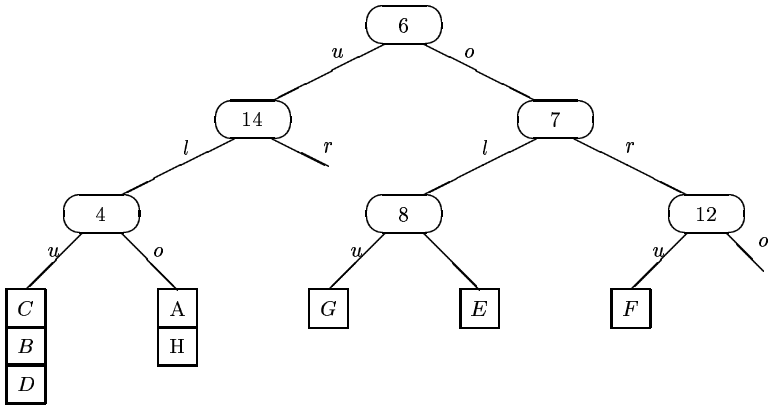

Abbildung 5.4: 2-d-Baum (inhomogen) zu den Datenpunkten A,B,C,D,E,F,G,H

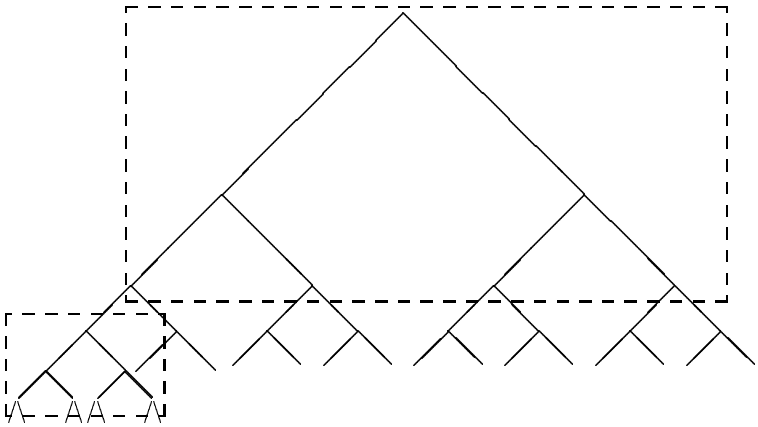

Abbildung 5.5: Zusammenfassung von je 7 inneren Knoten auf einem Index-Block

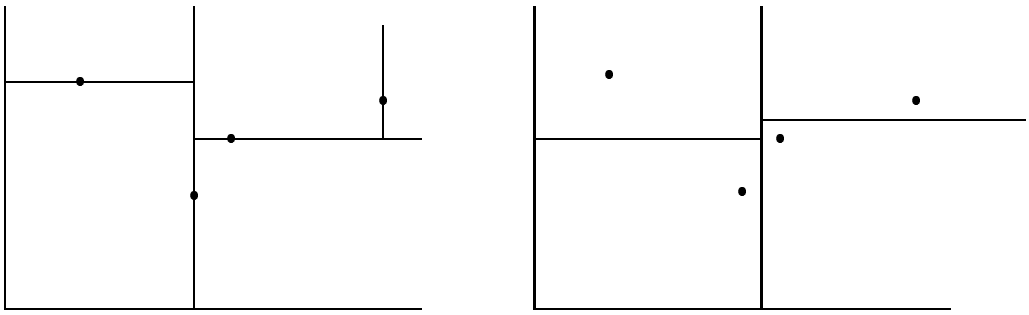

Abbildung 5.6: Partitionierungsmöglichkeiten des Raumes

5.3 Gitterverfahren mit konstanter Gittergröße

Gitterverfahren, die mit konstanter Gittergröße arbeiten, teilen den Datenraum in Quadrate fester Größe auf. Abbildung 5.7 zeigt eine Anordnung von 24 Datenblöcken, die jeweils eine feste Anzahl von Datenrecords aufnehmen können. Über einen Index werden die Blöcke erreicht. Diese statische Partitionierung lastet die Datenblöcke natürlich nur bei einer Gleichverteilung wirtschaftlich aus und erlaubt bei Ballungsgebieten keinen effizienten Zugriff.

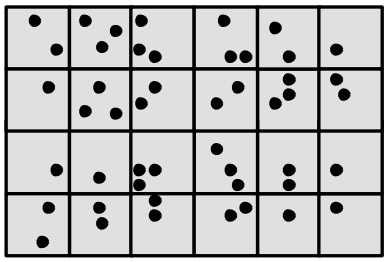

Abbildung 5.7: Partitionierung mit fester Gittergröße

5.4 Grid File

Als Alternative zu den Verfahren mit fester Gittergröße stellten Hinrichs und Nievergelt im Jahre 1981 das *Grid File* vor, welches auch bei dynamisch sich änderndem Datenbestand eine *2-Platten-Zugriffsgarantie* gibt.

Erreicht wird dies (bei k-dimensionalen Tupeln) durch

- k Skalen zum Einstieg ins Grid-Directory (im Hauptspeicher)
- Grid-Directory zum Finden der Bucket-Nr. (im Hintergrundspeicher)
- Buckets für Datensätze (im Hintergrundspeicher)

Zur einfacheren Veranschaulichung beschreiben wir die Technik für Dimension $k = 2$. Verwendet werden dabei

- **zwei eindimensionale Skalen**,
  welche die momentane Unterteilung der X- bzw. Y-Achse enthalten:

```
var X: array [0..max_x] of attribut_wert_x;
var Y: array [0..max_y] of attribut_wert_y;
```

- **ein 2-dimensionales Grid-Directory**,
  welches Verweise auf die Datenblöcke enthält:

```
var G: array [0..max_x - 1, 0..max_y - 1] of pointer;
```

D.h. `G[i, j]` enthält eine Bucketadresse, in der ein rechteckiger Teilbereich der Datenpunkte abgespeichert ist. Zum Beispiel sind alle Punkte mit $30 < x \leq 40, 2050 < y \leq 2500$ im Bucket mit Adresse `G[1,2]` zu finden (in Abbildung 5.8 gestrichelt umrandet). Achtung: mehrere Gitterzellen können im selben Bucket liegen.

- **mehrere Buckets**,
  welche jeweils eine maximale Zahl von Datenrecords aufnehmen können.

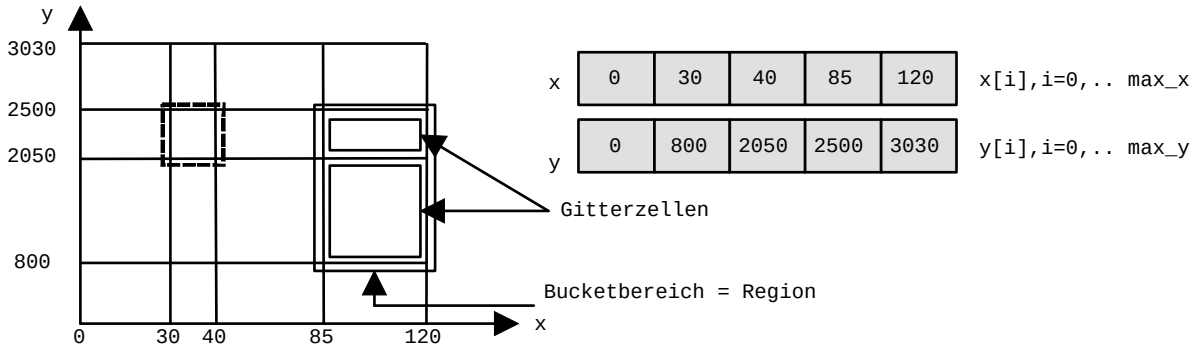

Abbildung 5.8: Skalen und resultierende Gitterzellen

Beispiel für ein Lookup mit $x = 100, y = 1000$:
Suche in Skala x den letzten Eintrag $< x$. Er habe den Index $i = 3$.
Suche in Skala y den letzten Eintrag $< y$. Er habe den Index $j = 1$.
Lade den Teil des Grid-Directory in den Hauptspeicher, der $G[3, 1]$ enthält.
Lade Bucket mit Adresse $G[3, 1]$.

Beispiel für den Zugriff auf das Bucket-Directory:
Vorhanden seien 1.000.000 Datentupel, jeweils 4 passen in einen Block. Die X- und die Y-Achse habe jeweils 500 Unterteilungen. Daraus ergeben sich 250.000 Einträge für das Bucket-Directory G. Bei 4 Bytes pro Zeiger und 1024 Bytes pro Block passen 250 Zeiger auf einen Directory-Block. Also gibt es 1000 Directory-Blöcke. D.h. $G[i, j]$ findet sich auf Block $2 \cdot j$ als i-te Adresse, falls $i < 250$ und befindet sich auf Block $2 \cdot j + 1$ als $(i - 250)$-te Adresse, falls $i \geq 250$

Bei einer *range query*, gegeben durch ein Suchrechteck, werden zunächst alle Gitterzellen bestimmt, die in Frage kommen, und dann die zugehörenden Buckets eingelesen.

5.5 Aufspalten und Mischen beim Grid File

Die grundsätzliche Idee besteht darin, bei sich änderndem Datenbestand durch Modifikation der Skalen die Größen der Gitterzellen anzupassen.

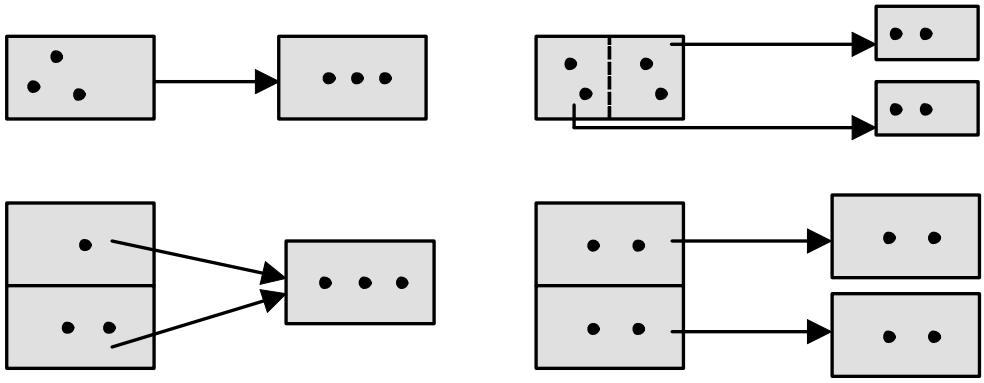

Abbildung 5.9: Konsequenzen eines Bucket-Überlauf (mit und ohne Gitterverfeinerung)

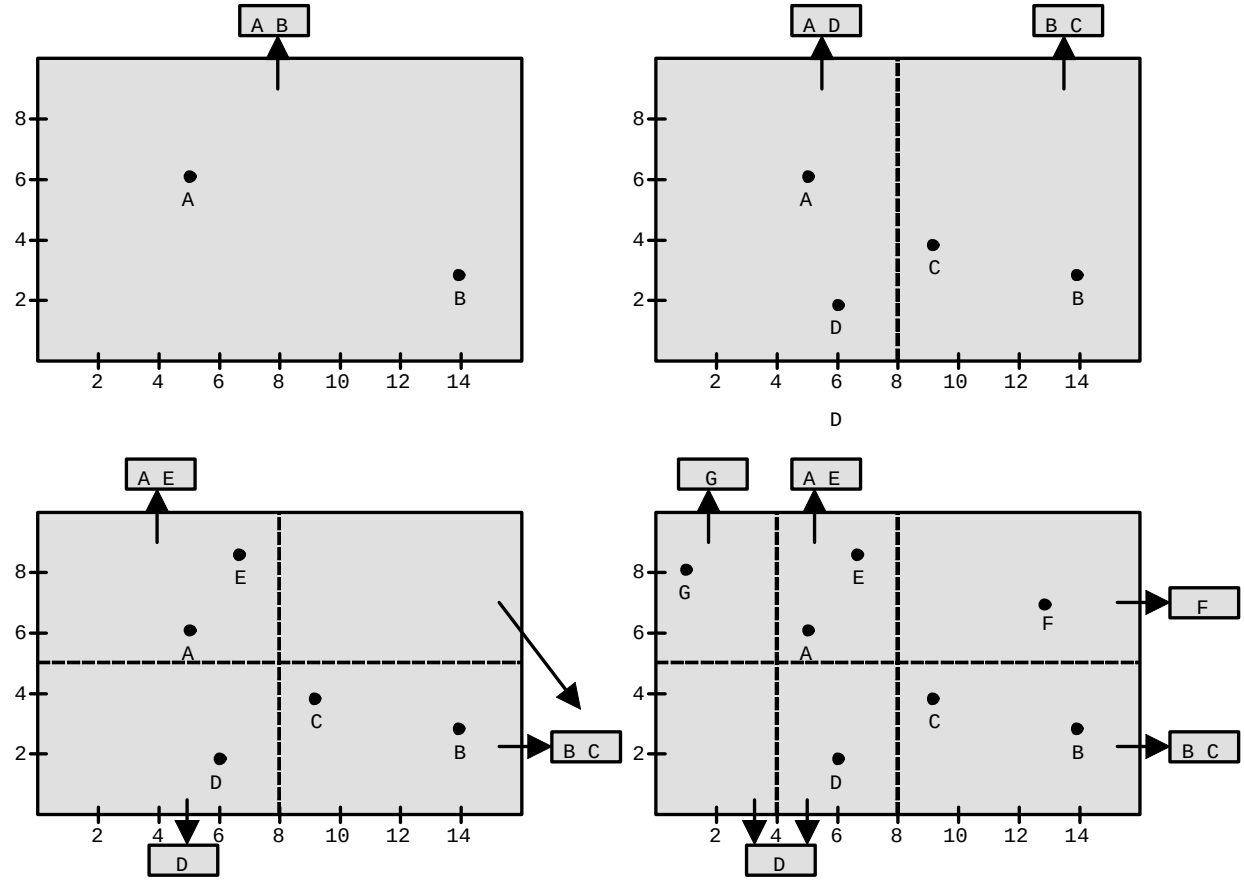

Abbildung 5.10: Aufspalten der Regionen für Datenpunkte A, B, C, D, E, F, G

Aufspalten von Regionen

Der Überlauf eines Buckets, dessen Region aus einer Zelle besteht, verursacht eine Gitterverfeinerung, die gemäß einer *Splitting Policy* organisiert wird. Im wesentlichen wird unter Abwechslung der Dimension die Region halbiert. Dieser Sachverhalt wird in der oberen Hälfte von Abbildung 5.9

demonstriert unter der Annahme, daß drei Datenrecords in ein Datenbucket passen. In der unteren Hälfte von Abbildung 5.9 ist zu sehen, daß bei Überlauf eines Buckets, dessen Region aus mehreren Gitterzellen besteht, keine Gitterverfeinerung erforderlich ist.

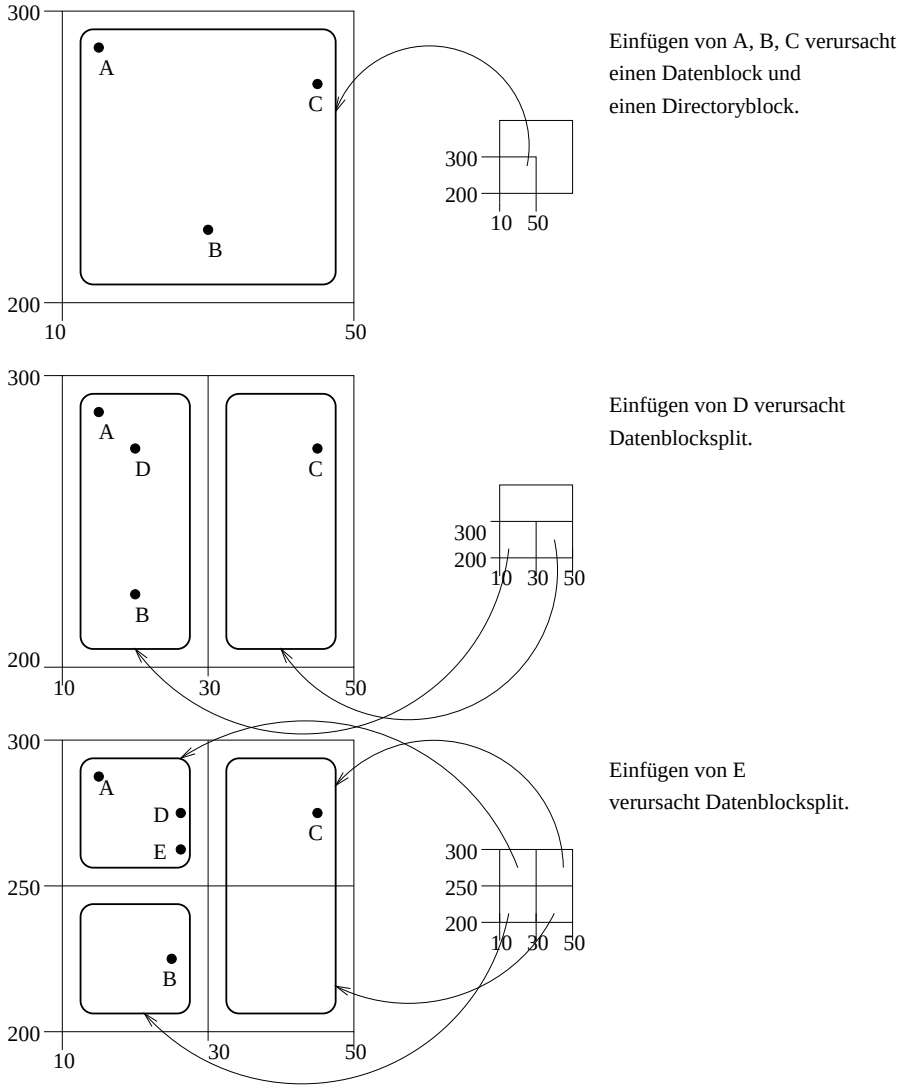

Abbildung 5.11: Dynamik des Grid File beim Einfügen der Datenpunkte A,B,C,D,E

Abbildung 5.10 zeigt die durch das sukzessive Einfügen in ein Grid File entwickelte Dynamik. Es handelt sich dabei um die in Kapitel 4 verwendeten Datenpunkte A, B, C, D, E, F, G. In dem Beispiel wird angenommen, daß 2 Datenrecords in einen Datenblock passen. Bei überlaufendem Datenblock wird die Region halbiert, wobei die Dimension abwechselt. Schließlich hat das Grid-Directory 6 Zeiger auf insgesamt 5 Datenblöcke. Die x-Skala hat drei Einträge, die y-Skala hat zwei Einträge.

Zu der dynamischen Anpassung der Skalen und Datenblöcke kommt noch die Buchhaltung der Directory-Blöcke. Dies wird in der Abbildung 5.11 demonstriert anhand der (neu positionierten) Datenpunkte A, B, C, D, E. Von den Directory-Blöcken wird angenommen, daß sie vier Adressen speichern können, in einen Datenblock mögen drei Datenrecords passen. Grundsätzlich erfolgt der Einstieg in den zuständi-

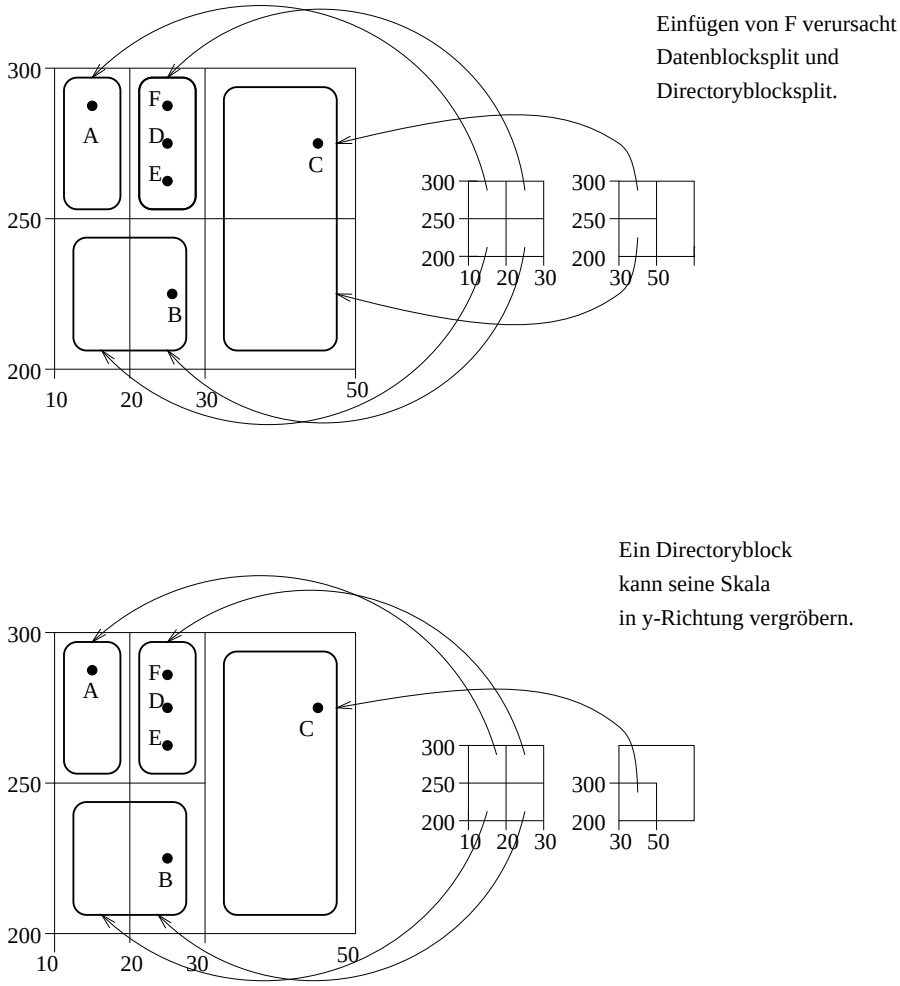

Abbildung 5.12: Vergröberung des Grid Directory nach Aufspalten

gen Directory-Block über das sogenannte *Root-Directory*, welches im Hauptspeicher mit vergröberten Skalen liegt. Die durch das Einfügen verursachte Aufspaltung eines Datenblocks und die dadurch ausgelösten Verfeinerungen der Skalen ziehen auch Erweiterungen im Directory-Block nach. Abbildung 5.12 zeigt, wie beim Überlauf eines Directory-Blockes dieser halbiert und auf zwei Blöcke verteilt wird. Dabei kommt es zu einer Vergröberung der Skala.

Mischen von Regionen

Die beim Expandieren erzeugte Neustrukturierung bedarf einer Umordnung, wenn der Datenbestand schrumpft, denn nach dem Entfernen von Datenrecords können Datenblöcke mit zu geringer Auslastung entstehen, welche dann zusammengefaßt werden sollten. Die *Merging Policy* legt den Mischpartner und den Zeitpunkt des Mischens fest:

- Mischpartner zu einem Bucket X kann nur ein Bucket Y sein, wenn die Vereinigung der beiden Bucketregionen ein Rechteck bildet (Abbildung 5.13). Grund: Zur effizienten Bearbeitung von Range-Queries sind nur rechteckige Gitter sinnvoll!

- Das Mischen wird ausgelöst, wenn ein Bucket höchstens zu 30 % belegt ist und wenn das vereinigte Bucket höchstens zu 70 % belegt sein würde (um erneutes Splitten zu vermeiden)

Abbildung 5.13: Zusammenfassung von Regionen

5.6 Verwaltung geometrischer Objekte

In der bisherigen Anwendung repräsentierten die Datenpunkte im k-dimensionale Raum k-stellige Attributkombinationen. Wir wollen jetzt mithilfe der Datenpunkte geometrische Objekte darstellen und einfache geometrische Anfragen realisieren.

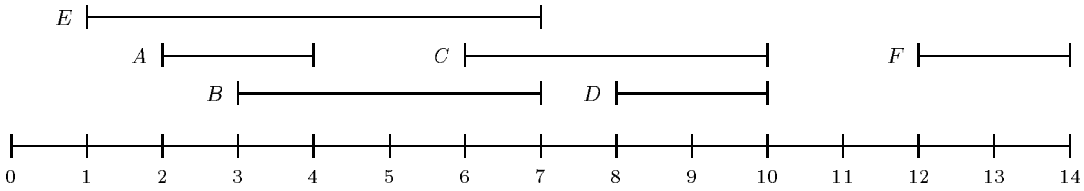

Abbildung 5.14: Intervalle A,B,C,D,E,F über der Zahlengeraden

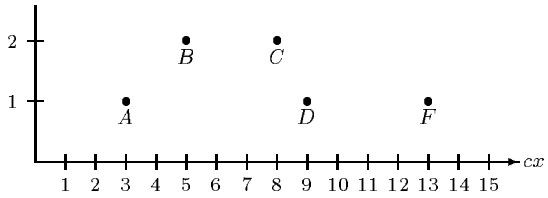

Abbildung 5.15: Repräsentation von Intervallen durch Punkte

Abbildung 5.14 zeigt eine Ansammlung von Intervallen, die zu verwalten seien. Die Intervalle sollen durch Punkte im mehrdimensionalen Raum dargestellt werden. Wenn alle Intervalle durch ihre Anfangs- und Endpunkte repräsentiert würden, kämen sie auf der Datenfläche nur oberhalb der 45-Grad-Geraden zu liegen.

Abbildung 5.15 präsentiert eine wirtschaftlichere Verteilung, indem jede Gerade durch ihren Mittelpunkt und ihre halbe Länge repräsentiert wird.

Typische Queries an die Intervall-Sammlung lauten:

- Gegeben Punkt P, finde alle Intervalle, die ihn enthalten.
- Gegeben Intervall I, finde alle Intervalle, die es schneidet.

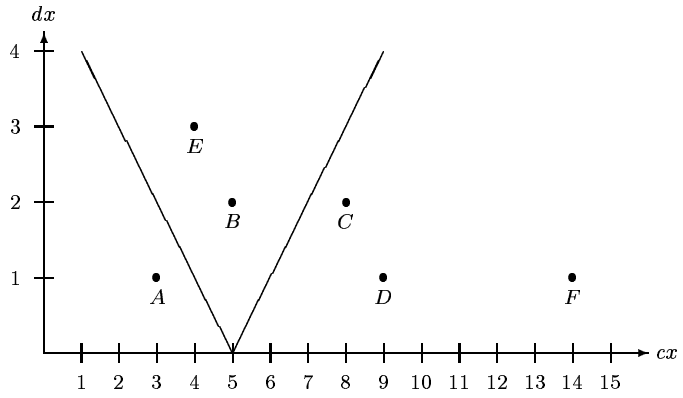

Abbildung 5.16: Abfragekegel zu Punkt p=5

Abbildung 5.16 zeigt den kegelförmigen Abfragebereich zum Query-Punkt p=5, in dem alle Intervalle (repräsentiert durch Punkte) liegen, die den Punkt p enthalten. Grundlage ist die Überlegung, daß ein Punkt P genau dann im Intervall mit Mitte m und halber Länge d enhalten ist, wenn gilt: $m - d \leq p \leq m + d$

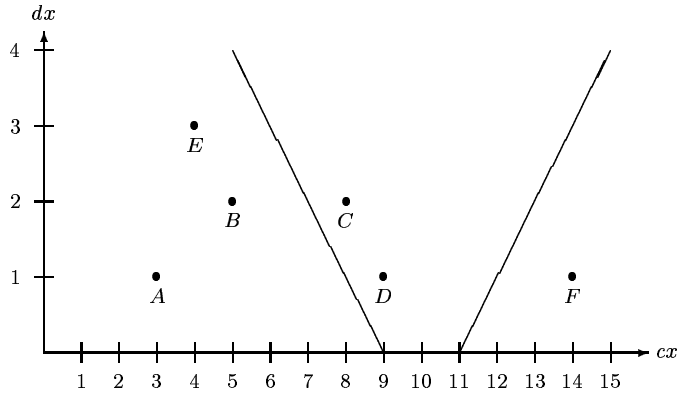

Abbildung 5.17: Abfragekegel zu Intervall mit Mitte s=10 und halber Länge t=1

Abbildung 5.17 zeigt den kegelförmigen Abfragebereich zu dem Query-Intervall mit Mittelpunkt s=10 und halber Länge t=1, in dem alle Intervalle (repäsentiert durch Punkte) liegen, die das Query-Intervall schneiden. Grundlage ist die Überlegung, daß ein Intervall mit Mitte s und halber Länge t genau dann ein Intervall mit Mitte m und halber Länge d schneidet, wenn gilt: $m-d \leq s+t$ und $s-t \leq m+d$

Abbildung 5.18 zeigt die Vorgehensweise bei der Bestimmung des nächstgelegenen Nachbarn (englisch: *nearest neighbor*). Suche zunächst auf dem Datenblock, der für den Query-Point Q zuständig

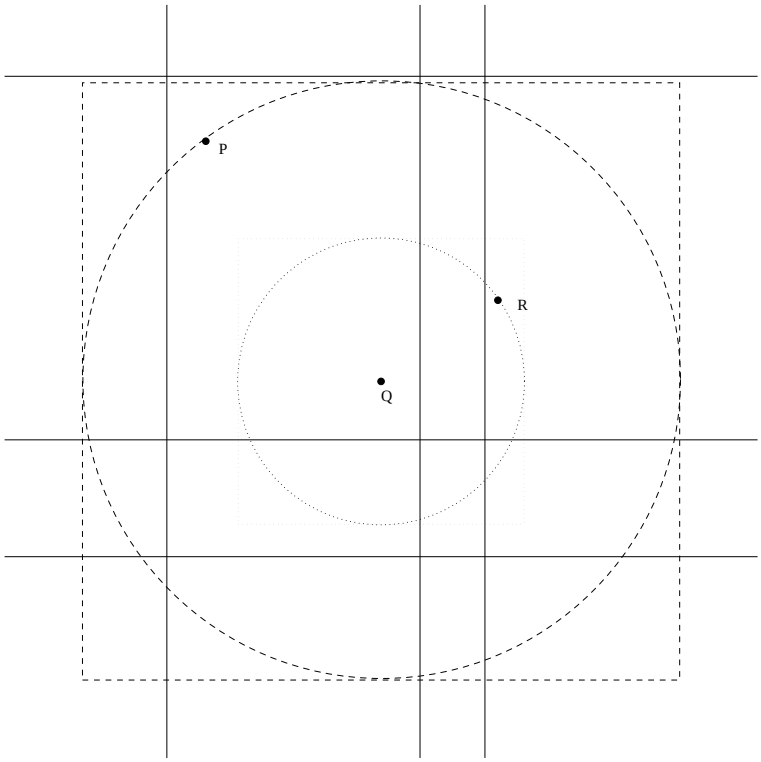

Abbildung 5.18: Nearest-Neighbor-Suche zu Query-Punkt Q

ist, den nächstgelegenen Punkt P. Bilde eine *Range-Query* mit Quadrat um den Kreis um Q mit Radius $|P - Q|$. Schränke Quadratgröße weiter ein, falls nähere Punkte gefunden werden.

Die erwähnten Techniken lassen sich auf höherdimensionierte Geometrie-Objekte wie Rechtecke oder Quader erweitern. Zum Beispiel bietet sich zur Verwaltung von orthogonalen Rechtecken in der Ebene folgende Möglichkeit an: Ein Rechteck wird repräsentiert als ein Punkt im 4-dimensionalen Raum, gebildet durch die beiden 2-dimensionalen Punkte für horizontale bzw. vertikale Lage. Zu einem Query-Rechteck, bestehend aus horizontalem Intervall P und vertikalem Intervall Q, lassen sich die schneidenden Rechtecke finden im Durchschnitt der beiden kegelförmigen Abfragebereiche zu den Intervallen P und Q.