10 ZUGRIFFSRECHTE

Jeder Benutzer weist sich gegenüber ORACLE durch seinen Benutzernamen und sein Paßwort aus.[1] Dem Benutzernamen werden vom Datenbankadministrator (DBA) Zugriffsrechte erteilt. Der DBA hat somit eine Schlüsselfunktion in der Verwaltung der Datenbank. In der Praxis fallen u.a. folgende Tätigkeiten in seinen Aufgabenbereich: Einrichten der Datenbank, Verwaltung der Benutzerrechte, Performance-Tuning der Datenbank, Durchführung von Datensicherungsmaßnahmen und "Beaufsichtigung" der Benutzer (Auditing).

Bei der Vergabe von Zugriffsrechten an die Benutzer wird meist nach der Regel verfahren, das jeder Benutzer so wenige Rechte wie möglich bekommt - damit er möglichst wenig Schaden in der Datenbank anrichten kann - und so viele Rechte wie nötig um seine Anwendungen durchführen zu können.

Schemakonzept. Die Organisation von Tabellen in ORACLE basiert auf dem *Schemakonzept*: Jedem Benutzer ist sein *Database Schema* zugeordnet, in dem er sich defaultmäßig aufhält und in dem er Objekte erzeugt und Anfragen an sie stellt. Die Bezeichnung der Tabellen geschieht somit global durch `<username>.<table>`, wird bei einer Anfrage durch einen Benutzer das Schema nicht angegeben, wird automatisch das entsprechende Objekt aus dem eigenen Schema angesprochen.

Die Zugriffsrechte teilen sich in zwei Gruppen: "Systemprivilegien", die z.B. zu Schemaoperationen berechtigen, sowie "Objekt"rechte, d.h., was man dann mit den einzelnen Objekten machen kann.

Systemprivilegien. In ORACLE existieren über 80 verschiedene Systemberechtigungen. Jede Systemberechtigung erlaubt einem Benutzer die Ausführung einer bestimmten Datenbankoperation oder Klasse von Datenbankoperationen. Auch bestimmte Schema-Objekte, wie z.B. Cluster, Indices und Trigger werden nur durch Systemberechtigungen gesteuert. Da Systemberechtigungen oftmals weitreichende Konsequenzen haben, darf sie in der Regel nur der DBA vergeben.

- `CREATE [ANY] TABLE/VIEW/TYPE/INDEX/CLUSTER/TRIGGER/PROCEDURE`: Benutzer darf die entsprechenden Schema-Objekte erzeugen,
- `ALTER [ANY] TABLE/TYPE/TRIGGER/PROCEDURE`: Benutzer darf die entsprechenden Schema-Objekte verändern,
- `DROP [ANY] TABLE/TYPE/VIEW/INDEX/CLUSTER/TRIGGER/PROCEDURE`: Benutzer darf die entsprechenden Schema-Objekte löschen.
- `SELECT/INSERT/UPDATE/DELETE [ANY] TABLE`: Benutzer darf in Tabellen Tupel lesen/erzeugen/verändern/entfernen.

`ANY` berechtigt dazu die entsprechende Operation in *jedem* Schema auszuführen, fehlt `ANY`, so bleibt die Ausführung auf das eigene Schema beschränkt. Die Teilnehmer des Praktikums sollten etwa mit den Privilegien `CREATE SESSION`, `ALTER SESSION`, `CREATE TABLE`, `CREATE TYPE`, `CREATE CLUSTER`, `CREATE SYNONYM`, `CREATE VIEW` ausgestattet sein. Dabei sind die Zugriffe und Veränderungen an den eigenen Tabellen nicht explizit aufgeführt (würde logischerweise `SELECT TABLE` heißen).

[1] Im Rahmen des Praktikums wird durch den Aufruf von `sqlplus /` der UNIX-Account zur Autorisierung verwendet.

Will man auf eine Tabelle zugreifen, deren Zugriff man nicht gestattet ist, etwa SELECT * FROM Kohl.Private, bekommt man nur die Meldung ORA-00942: Tabelle oder View nicht vorhanden.

Systemprivilegien werden mit Hilfe der GRANT-Anweisung erteilt:

```
GRANT <privilege-list>
TO <user-list> | PUBLIC [ WITH ADMIN OPTION ];
```

<privilege-list> ist dabei eine Aufzählung der administrativen Privilegien, die man jemandem erteilen will. Mit der <user-list> gibt man an, wem diese Privilegien erteilt werden; mit PUBLIC erhält jeder das Recht. Wird ein Systemprivileg zusätzlich mit ADMIN OPTION vergeben, so kann der Empfänger es auch weiter vergeben.

Informationen bzgl. der Zugriffsrechte werden in Tabellen des Data Dictionary gespeichert.[2] Mit SELECT * FROM SESSION_PRIVS erfährt man, welche generellen Rechte man besitzt.

Objektprivilegien. Jedes Datenbankobjekt hat einen Eigentümer, dies ist meistens derjenige der es angelegt hat (wozu er allerdings auch wieder das Recht haben muß). Prinzipiell darf niemand sonst mit einem solchen Objekt arbeiten, außer der Eigentümer oder der DBA erteilt ihm explizit entsprechende Rechte:

```
GRANT <privilege-list> | ALL [<column-list>]
ON <object>
TO <user-list> | PUBLIC
[ WITH GRANT OPTION ];
```

Hierbei wird das betreffende Objekt durch <object> beschrieben und kann von der Art TABLE, VIEW, PROCEDURE/FUNCTION/PACKAGE oder TYPE sein. <privilege-list> ist eine Aufzählung der Privilegien, die man erteilen will. Mögliche Privilegien sind CREATE, ALTER und DROP für die verschiedenen Arten von Schemaobjekten, DELETE, INSERT, SELECT und UPDATE für Tabellen[3] und Views, außerdem INDEX, und REFERENCES für Tabellen. Für INSERT, REFERENCES und UPDATE kann optional angegeben werden, für welche Spalten das Recht vergeben wird (bei DELETE ist das natürlich nicht erlaubt und resultiert auch in einer Fehlermeldung[4]. Für Prozeduren/Funktionen/Packages und Typen kann man EXECUTE vergeben. Mit ALL gibt man alle Privilegien die man an dem beschriebenen Objekt hat, weiter.

- SELECT, UPDATE, INSERT, DELETE klar, die entsprechenden Zugriffs- und Änderungsoperationen,
- REFERENCES: das Recht, Spalten einer Tabelle als Fremdschlüssel in einer CREATE TABLE-Anweisung zu deklarieren und referentielle Aktionen anzugeben,
- INDEX das Recht, Indexe auf dieser Tabelle zu erstellen.
- EXECUTE das Recht, eine Prozedur, Funktion oder einen Trigger auszuführen, oder einen Typ zu benutzen.

Mit der <user-list> gibt man an, wem diese Privilegien erteilt werden; mit PUBLIC erhält jeder das Recht. Wird ein Recht mit GRANT OPTION vergeben, kann derjenige, der es erhält, es auch weiter vergeben.

Rechte entziehen. Gelegentlich ist es erforderlich, jemandem ein Recht (oder nur die GRANT OPTION) zu entziehen; dies kann man natürlich nur machen, wenn man dieses Recht selbst vergeben hat:

[2] Zur Erinnerung: alle Tabellen, die im Data Dictionary enthalten sind, können mit SELECT * FROM DICTIONARY erfragt werden.

[3] dabei kann man mit UPDATE-Recht weder löschen noch einfügen.

[4] für Oracle 8.0.3 getestet.

Administrative Rechte:

```
REVOKE <privileg-list> | ALL
FROM <user-list> | PUBLIC;
```

Objekt-Rechte:

```
REVOKE <privileg-list> | ALL
ON TABLE <table>
FROM <user-list> | PUBLIC [CASCADE CONSTRAINTS];
```

`CASCADE CONSTRAINTS` sorgt dafür, daß alle referentiellen Integritätsbedingungen, die auf einem entzogenen `REFERENCES`-Privileg beruhen, wegfallen.

- Hat ein Benutzer eine Berechtigung von mehreren Benutzern erhalten, behält er sie, bis sie ihm jeder einzelne entzogen hat.
- Hat ein Benutzer eine Berechtigung mit `ADMIN OPTION` oder `GRANT OPTION` weitergegeben und bekommt sie entzogen, so wird sie automatisch auch allen Benutzern entzogen, an die er sie weitergegeben hat.

Mit

```
SELECT * FROM USER_TAB_PRIVS;
```

kann man sich die Informationen über Tabellen ausgegeben lassen, die einem gehören und auf die man irgendwelche Rechte vergeben oder die jemandem anderem gehören und man Rechte dafür bekommen hat. Hat man Rechte nur für bestimmte Spalten einer Tabelle vergeben oder erhalten, erfährt man dies mit

```
SELECT * FROM USER_COL_PRIVS;
```

Die Tabellen `USER_TAB/COL_PRIVS_MADE` und `USER_TAB/COL_PRIVS_RECD` enthalten im einzelnen die vergebenen oder erhaltenen Rechte für Tabellen bzw. Spalten.

Rollenkonzept. Neben der direkten Vergabe von Privilegien an einzelne Beutzer kann die Vergabe von Privilegien in ORACLE auch durch *Rollen* geregelt werden. Benutzer die über die gleichen Rechte verfügen sollen werden dann einer bestimmten Rolle zugeordnet. Das Rollenkonzept vereinfacht somit auch die Rechtevergabe für den DBA, da die Rechte nur einmal der Rolle zugeordnet werden, anstatt dies für jeden einzelnen Benutzer zu wiederholen. Rollen können hierarchisch strukturiert sein. Dabei muß darauf geachtet werden, daß keine Zyklen bei der Zuweisung entstehen. Die Zuordnung von Benutzern zu einer Rolle erhält man (falls man die DBA-Tabellen lesen darf) über folgende Anfrage

```
SELECT * FROM DBA_ROLE_PRIVS;
```

11 SYNONYME

Synomyme können verwendet werden, um ein Schemaobjekt unter einem anderen Namen als ursprünglich abgespeichert anzusprechen. Dies ist insbesondere sinnvoll, wenn man eine Relation, die in einem anderen Schema liegt, nur unter ihrem Namen ansprechen will.

```
CREATE [PUBLIC] SYNONYM <synonym>
  FOR <schema>.<object>;
```

Ohne den Zusatz `PUBLIC` ist das Synonym nur für den Benutzer definiert, der es definiert hat. Mit `PUBLIC` ist das Synonym systemweit verwendbar. Das dazu notwendige `CREATE ANY SYNONYM`-Privileg haben i.a. nur die SysAdmins.

Beispiel: Wenn ein Benutzer keine eigene Relation "City" besitzt, sondern stattdessen immer die Relation "City", aus dem Schema "dbis" verwendet, müßte er diese immer mit `SELECT * FROM dbis.City` ansprechen. Definiert man

```
CREATE SYNONYM City
  FOR dbis.City;
```

so kann man sie auch über `SELECT * FROM City` erreichen.

Man kann übrigens ein Synonym auch für eine Tabelle definieren, die (noch) nicht existiert und/oder auf die man noch kein Zugriffsrecht hat. Wird die Tabelle dann definiert und erhält man das Zugriffsrecht, tritt das Synonym in Kraft.

Synonyme werden mit `DROP SYNONYM <synonym>` gelöscht.

12 ZUGRIFFSEINSCHRÄNKUNG ÜBER VIEWS

Bei der Aufstellung der Objektprivilegien im vorhergehenden Abschnitt läßt sich feststellen, daß bei `GRANT SELECT` der Zugriff nicht auf Spalten eingeschränkt werden kann. Dies kann allerdings einfach über Views (vgl. Abschnitt 2.2) geschehen:

Der *nicht existierende* Befehl

```
GRANT SELECT [<column-list>]
ON <table>
TO <user-list> | PUBLIC
[ WITH GRANT OPTION ];
```

kann ersetzt werden durch

```
CREATE VIEW <view> AS
  SELECT <column-list>
  FROM <table>;
```

```
GRANT SELECT
ON <view>
TO <user-list> | PUBLIC
[ WITH GRANT OPTION ];
```

Beispiel: Der Benutzer `pol` ist Besitzer der Relation "Country". Er entscheidet sich, seinem Kollegen `geo` die Daten über die Hauptstadt und ihre Lage nicht zur Verfügung zu stellen. Alle restlichen Daten sollen von `geo` aber gelesen *und* geschrieben werden können.

```
CREATE VIEW pubCountry AS
SELECT Name, Code, Population, Area
FROM Country;
```

```
GRANT SELECT ON pubCountry TO geo;
```

Da das View über Basis *genau einer* Basisrelation definiert ist und der von Basisrelation und View übereinstimmt, sind Updates und Einfügungen erlaubt.

```
GRANT DELETE,UPDATE,INSERT ON pubCountry TO geo;
```

Zusätzlich wird `geo` sich ein Synonym definieren:

```
CREATE SYNONYM Country FOR pol.pubCountry;
```

Ärgerlich ist in diesem Zusammenhang nur, daß man keine Referenzen auf Views legen kann (diese haben keinen Primary Key): Um Country.Code zu referenzieren, muß `geo` trotzdem die *Basistabelle* `pol.pubCountry` verwenden:

```
pol : GRANT REFERENCE Code ON Country TO geo;
```

```
geo : ... REFERENCES pol.Country(Code);
```

13 OPTIMIERUNG DER DATENBANK

Dieser Abschnitt behandelt die in ORACLE vorhandenen Konstrukte, um Speicherung und Zugriff einer Datenbank zu optimieren. Diese Konstrukte dienen dazu, (a) effizienter bestimmte Einträge zu finden (Indexe) und (b) physikalisch schneller zugreifen zu könnnen (Cluster). (a) ist dabei ein algorithmisches Problem, betreffend die Suche nach Werten; (b) soll die Zugriffe auf den Hintergrundspeicher minimieren.

Diese Strukturen haben keine Auswirkung auf die Formulierung der restlichen SQL-Anweisungen!

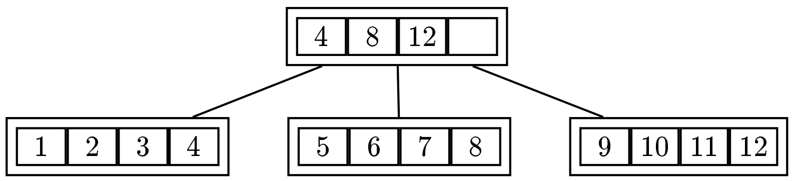
13.1 Indexe

Indexe unterstützen den Zugriff auf eine Tabelle über einen gegebenen Spaltenwert:

```
CREATE TABLE PLZ
  (City    VARCHAR2(35)
   Country VARCHAR2(4)
   ProvinceVARCHAR2(32)
   PLZ     NUMBER)
```

- Indexe in ORACLE als B-Baum organisiert,
- logisch und physikalisch unabhängig von den Daten der zugrundeliegenden Tabelle,
- keine Auswirkung auf die Formulierung einer SQL-Anweisung,
- mehrere Indexe für eine Tabelle möglich,
- bei sehr vielen Indexen auf einer Tabelle kann es beim Einfügen, Ändern und Löschen von Sätzen zu Performance-Verlusten kommen.

B-Baum:



- Logarithmische Höhe des Baumes,
- B-Baum: Knoten enthalten auch die Werte der Tupel,
- B*-Baum: Knoten enthalten *nur* die Weg-Information, daher ist der Verzweigungsgrad sehr hoch, und die Höhe des Baumes auch absolut gesehen gering.
- Schneller Zugriff (logarithmisch) versus hoher Reorganisationsaufwand (→ Algorithmentechnik-Vorlesung).

Indexarten:

- **eindeutige Indexe**
  - keine gleichen Werte in der Spalte des Indexe erlaubt,
  - bei einer `PRIMARY KEY`-Deklaration erzeugt ORACLE automatisch einen Index über diese Spalte(n).
- **mehrdeutige Indexe**
  - gleiche Werte sind erlaubt.
- **zusammengesetzte Indexe**
  - Index besteht aus mehreren Spalten (Attributen) einer Tabelle
  - sinnvoll, wenn häufig `WHERE`-Klauseln über mehrere Spalten formuliert werden (z.B. (L_ID,LT_ID)).

```
CREATE [UNIQUE] INDEX <name> ON <table>(<column-list>);
```

Im obigen Beispiel wäre ein Index über L_ID, PLZ sinnvoll:

```
CREATE INDEX PLZIndex ON PLZ (Country,PLZ);
SELECT *
  FROM plz
  WHERE plz = 79110 AND Country = ‘D’;
```

Indexe werden mit `DROP INDEX` gelöscht.

Wichtig: Indexe sind physikalische Strukturen, die in der Datenbank gespeichert sind, während Schlüssel ein rein logisches Konzept sind. Damit gehören Indexe *nicht* zur Modellierung (ER- und relationale Modellierung), sondern werden erst betrachtet, wenn das Datenbankschema entworfen wird.

13.2 Bitmap-Indexe

Die Motivation der oben genannten Indexe (insbesondere als `UNIQUE INDEX`) liegt in erster Linie darin, den Zugriff zu einzelnen Elementen zu verbessern indem nicht mehr die gesamte Relation linear durchsucht werden muß. Erst auf den zweiten Blick wird erkennbar, daß damit auch alle Werte mit demselben Indexwert effizient zugegriffen werden können.

Bei `BITMAP`-Indexen liegt das Hauptinteresse darauf, zu bestimmten Attributen, die nur wenige verschiedene Wert annehmen, *alle* Tupel mit einem bestimmten Attributwert optimal selektieren zu können: Für die entsprechenden Attribute wird eine Bitmap angelegt, die für jeden vorkommenden Wert ein bit enthält. Dieses wird für ein Tupel genau dann gesetzt, wenn das Tupel den entsprechenden Attributwert enthält:

```
CREATE BITMAP INDEX <name> ON <table>(<column-list>);
```

Beispiel 11 (Bitmap-Index) Die Relation *encompasses* kann vorteilhaft mit einem Bitmap-Index über Kontinent erweitert werden:

```
CREATE BITMAP INDEX continents_index ON encompasses(continent);
```

Country	**Perc.**	EUR	AS	AM	AF	AUS
D	100	×				
CH	100	×				
USA	100			×		
R	20	×				
R	80		×			
ZA	100				×	
NZ	100					×

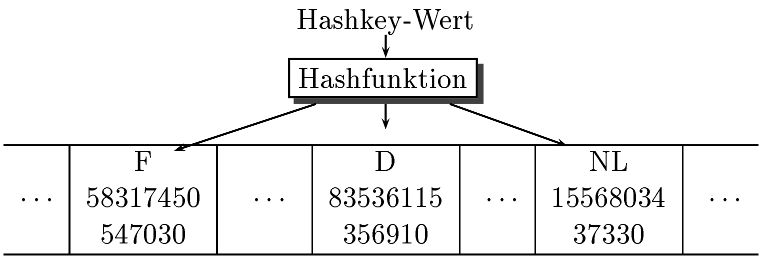
□

Bitmap-Indexe haben insbesondere dann Performanz-Vorteile, wenn eine `WHERE`-Klausel mehrere solche Spalten auswertet und damit auf eine Verknüpfung auf logischer (Bit)-Ebene auf diesen Indexen reduziert werden kann. Bitmap-Indexe sind damit maßgeschneidert für Data-Warehousing-Anwendungen.

13.3 Hashing

Hashing unterstützt Zugriff auf eine Tabelle über bestimmte Attributwerte (eben die, die als *Hashkey* definiert sind). Dabei wird aufgrund dieser Attributwerte *in konstanter Zeit* berechnet, wo die entsprechenden Tupel zu finden sind: Eine Hash-Funktion bildet jeden Wert des angegebenen Hashkeys auf eine Zahl innerhalb eines festgelegten Hash-Bereiches ab. Die Vorgehensweise ist damit ähnlich zu einem Index.

Z.B. wäre ein Hash-Zugriff sinnvoll, wenn um auf die Daten über ein bestimmtes Land gezielt zugreifen zu können: Hashkey ist *Country.Code*.



Vorteile der Hash-Methode:

- falls Anzahl der Hash-Werte ausreichend groß, genügt ein Blockzugriff, um einen Datensatz zu finden.
- kein Zugriff auf Indexblöcke erforderlich.

In Oracle 8 ist Hashing nur für Cluster implementiert.

13.4 Cluster

Cluster unterstützen Zugriff auf mehrere Tabellen mit gemeinsamen Spalten durch physikalisches Zusammenfassen der Tabellen nach diesen Spalten, um jeweils bei einem Hintergrundspeicherzugriff semantisch zusammengehörende Daten in den Hauptspeicher zu bringen.

- Zusammenfassung einer Gruppe von Tabellen, die alle eine oder mehrere gemeinsame Spalten (Clusterschlüssel) besitzen,

- sinnvoll bei referentiellen Integritätsbedingungen (Fremdschlüsselbeziehungen) oder bei häufigen JOIN-Anweisungen über dieselbe Spalte(n),
- auch zu einer einzelnen Tabelle kann ein Cluster erzeugt werden, z.B. um Städte nach Landesteilen geordnet abzuspeichern.

Vorteile eines Clusters:

- geringere Anzahl an Plattenzugriffen und schnellere Zugriffsgeschwindigkeit
- geringerer Speicherbedarf, da jeder Clusterschlüsselwert nur einmal abgespeichert wird

Nachteile:

- ineffizient bei häufigen Updates der Clusterschlüsselwerte, da dies eine physikalische Reorganisation bewirkt
- schlechtere Performance beim Einfügen in Cluster-Tabellen

Wird ein Cluster erstellt, muß zuerst dessen Cluster-Schlüssel angegeben werden, um die in diesem Cluster gespeicherten Tabellen geeignet zu organisieren.

Erzeugen eines Clusters:

1. Cluster und Clusterschlüssel definieren,
2. Tabellen erzeugen und mit der Angabe der Spalten für den Clusterschlüssel in den Cluster einfügen,
3. Clusterindex über den Clusterschlüssel-Spalte(n) definieren. Dies muß *vor* dem ersten DML-Kommando geschehen.

In Mondial gibt es einige Relationen (*Mountain*, *Lake*, ...), in denen zusammengehörende Daten über auf zwei Relationen verteilt sind: Z.B. enthält *Sea* die topographischen Informationen über ein Meer, während *geo_Sea* dessen Lage in bezug auf geopolitische Begriffe bezeichnet. Diese könnte man clustern. Ebenso ist es sinnvoll, die Relation *City* nach (*Country, Province*) zu clustern. Beide Cluster sind in Abb. 13.1 und 13.2 gezeigt.

Diese Speicherstruktur erreicht man folgendermaßen:

Zuerst wird ein Cluster erzeugt, und angegeben, welche Spalten (und ihre Datentypen) den Clusterschlüssel bilden sollen:

```
CREATE CLUSTER <name>(<col> <datatype>-list)
  [INDEX | HASHKEYS <integer> [HASH IS <funktion>]];
```

Als Default wird ein *indexed Cluster* erstellt, d.h. die Zeilen werden entsprechend ihren Clusterschlüsselwerten indiziert und geclustert.

Gibt man HASH und eine Hashfunktion an wird aus den Zeilen eines Tupels dessen Hashwert (mit <funktion> falls angegeben, sonst intern) berechnet, der dann modulo <integer> genommen wird. Nach diesem Wert wird dann geclustert (wobei vollkommen unzusammenhängende Zeilen zusammen geclustert werden können).

Die Tabellen werden durch ein weiteren Befehl in der CREATE TABLE-Definition einem Cluster zugeordnet. Dabei muß die Zuordnung der Spalten zum Clusterschlüssel angegeben werden:

```
CREATE TABLE <table>
  (<col> <datatype>,
    ⋮
```

Sea		
Mediterranean Sea	**Depth**	
	5121	
	Province	**Country**
	Catalonia	E
	Balearic Isl.	E
	Valencia	E
	Murcia	E
	Andalusia	E
	Corse	F
	Languedoc-R.	F
	Provence	F
	⋮	⋮
Baltic Sea	**Depth**	
	459	
	Province	**Country**
	Schleswig-H.	D
	Mecklenb.-Vorp.	D
	Szczecin	PL
	Koszalin	PL
	Gdansk	PL
	Olsztyn	PL
	⋮	⋮

Abbildung 13.1: Geclusterte Tabellen

```
    <col> <datatype>)
   CLUSTER <cluster>(<column-list>);
```

Zum Schluss muß noch der Clusterschlüsselindex erzeugt werden:

```
CREATE INDEX <name> ON CLUSTER <cluster>;
```

Mit den folgenden Befehlen kann man den oben beschriebenen Cluster *Cl_Sea* generieren:

```
CREATE CLUSTER Cl_Sea (Sea VARCHAR2(25));

CREATE TABLE CSea
  (Name    VARCHAR2(25) PRIMARY KEY,
   Depth   NUMBER)
  CLUSTER Cl_Sea (Name);

CREATE TABLE Cgeo_Sea
  (ProvinceVARCHAR2(35),
   Country VARCHAR2(4),
   Sea     VARCHAR2(25))
  CLUSTER Cl_Sea (Sea);
```

Country	**Province**			
D	Nordrh.-Westf.	**City**	**Population**	...
		Düsseldorf.	572638	...
		Solingen	165973	...
USA	Washington	**City**	**Population**	...
		Seattle	524704	...
		Tacoma	179114	...
⋮	⋮	⋮	⋮	⋮

Abbildung 13.2: Geclusterte Tabellen

```
CREATE INDEX ClSeaInd ON CLUSTER Cl_Sea;
```

Alle drei Methoden erfordern relativ hohen Aufwand, wenn sie reorganisiert werden müssen (Überlaufen von Indexknoten, Cluster-Bereichen, Hash-Speicherbereichen oder des Wertebereichs der Hash-Funktion).