Kleine Oracle Schulung

Inhaltsverzeichnis

1 PARTITIONING 1

1.1 Arten 1
1.1.1 Range 1
1.1.2 Hash 1
1.1.3 List 1
1.1.4 Composite Range-Hash 2
1.1.5 Composite Range-List 2

1.2 Vorteile 2
1.2.1 Partition-Pruning 2
1.2.2 Partitioned-Wise-Joins 2
1.2.3 Index 3

1.3 B*Tree Indexe 3
1.3.1 Local Index 3
1.3.2 Prefixed Index 3

1.4 Global Index 4
1.4.1 Global partitioned Index 4
1.4.2 Global nonpartitoned Index 4

1.5 Bitmap Index 5

1.6 Bitmap Join Index 5

1 Partitioning

Mit dieser Option können Tabellen und Indizes in kleinere Untereinheiten aufgeteilt werden. Dies erleichtert die Managebarkeit von grossen Datenmengen. Weiter gibt es so noch eine Reihe von weiteren Vorteilen.

1.1 Arten

Im Wesentlichen werden drei Grundarten von Partitioning unterschieden. Daneben existieren auch Kombinationen davon.

1.1.1 Range

Diese Art von Partitioning basiert auf dem Datenrange einer Spalte. Jede Partition ist definiert durch die Abgrenzung eines Wertes.
Eine typische Anwendung von Rangepartitioning ist die Unterteilung der Daten in Zeitabschnitte zum Beispiel Monate oder Quartale.

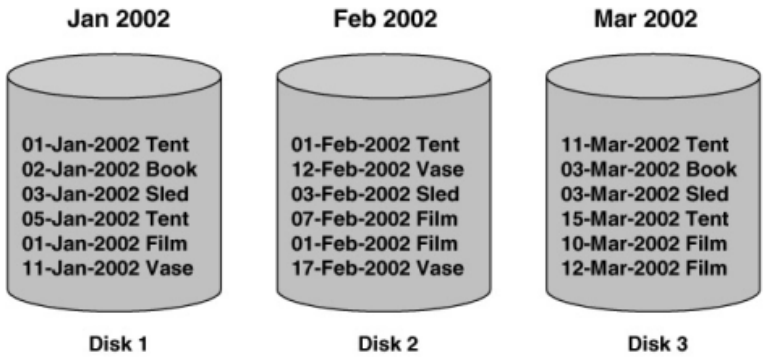


Abbildung 1: Range-Partitionierung

1.1.2 Hash

Diese Art von Partitioning basiert auf dem Hashwert eines Wertes einer Spalte.
Haspartitioning hat keine solche logische Bedeutung wie das Rangeopartitioning. Es dient vielmehr einer gleichmässigen Verteilung von Daten. Dabei muss die Anzahl der Partitionen einer 2er Potenz entsprechen. Hier funktioniert das Partition-Pruning in der Regel nur, wenn eine einzelne Zeile gemäss dem Partitionierungskey aus der Tabelle gelesen wird.
Typischerweise wird Hashpartitioning in Verbindung mit Rangepartitioning verwendet.

1.1.3 List

Hier werden die Daten gemäss einer Werteliste in Partitionen unterteilt. Man kann sich zum Beispiel eine Unterteilung nach Ländern oder Kontinenten vorstellen.

1.1.4 Composite Range-Hash

Dies ist eine Kombination von Rangepartitioning und Hashpartitioning. Typischerweise wird eine Datenmenge erst in logische Einheiten unterteilt, zum Beispiel nach Monaten und diese Untereinheiten werden nochmals über eine Hashalgorithmus weiter unterteilt. Dies ist dann sinnvoll, wenn die Rangepartitionen immer noch sehr gross sind. Dies bedeutet aber auch, dass die Anzahl der Partitionen sehr hoch werden kann. Zum Beispiel 48 Monatspartitionen unterteilt in 8 Subpartitionen ergeben 384 Partitionen.

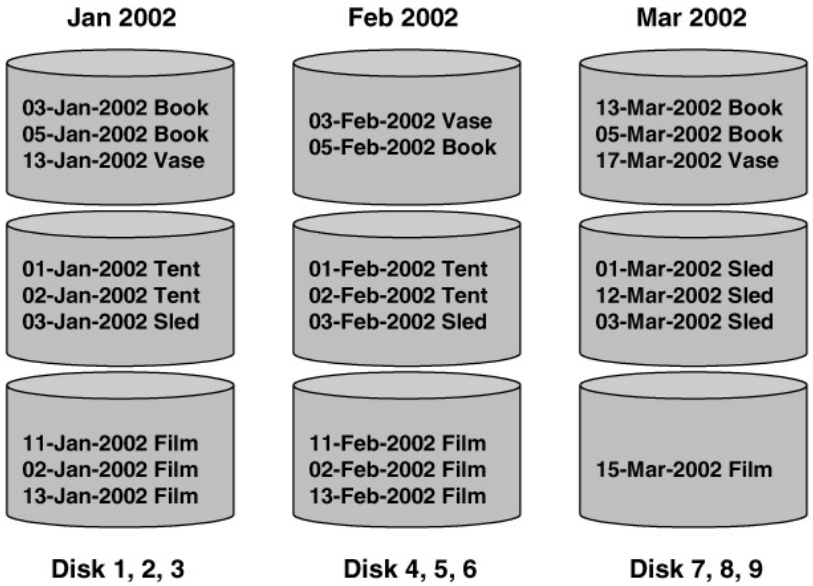


Abbildung 2: Compoite-Partitionierung (Range-Hash)

1.1.5 Composite Range-List

Dies ist eine Kombination von Rangepartitioning und Listpartitioning.

1.2 Vorteile

- Kleinere und damit einfacher zu managende Dateneinheiten
- Partitionen sind unabhängig voneinander
- Recovery funktioniert auf Partition-Level
- Transparent für den Benutzer
- Schnellerer Datenzugriff
- SQL-Optimierungen wie:
  - Partition-Pruning oder Partition-Elemination
  - Partitioned-Wise-Joins

1.2.1 Partition-Pruning

Dieses Feature kommt hauptsächlich bei Range-Partitionierung oder List-Partitionierung zum Einsatz. Der Optimizer kann anhand der ‚where'-Klausel einzelne Partitionen vom durchsuchen ausschliessen, weil er weiss, dass die gesuchten Zeilen in den jeweiligen Partitionen nicht vorkommen. Dies reduziert das physische Lesevolumen erheblich und erhöht damit die Performance.

1.2.2 Partitioned-Wise-Joins

Werden grosse partitionierte Tabellen gejoint, dann kann der Optimizer statt eines grossen Joins, die Join-Operation auf mehrere kleinere Joins aufteilen, nämlich eine Join-Operation pro Partition. Diese können entweder parallel oder seriell ausgeführt werden.

1.2.3 Index

Index sind sehr wichtig für eine effiziente Datenabfrage. Oracle unterstützt im wesentlichen drei Arten von Indexes:

- B*Tree
- Bitmap
- Bitmap Join

Diese drei Arten werden in den folgenden Kapiteln näher beschrieben. Zudem wird auch der Zusammenhang zur Partitioning-Option erläutert.

*1.3 B*Tree Indexe*

B*Tree Indexe sind hierarchisch struktriert und speichern Pointers, welche auf die Row in der Tabelle zeigen. Ein B*Tree-Index ist umso besser je höher die Kardinalität der Spalte ist. Ein B*Tree kann Unique oder Nonunique sein. Falls mehre Spalten zusammen in einer ‚where'-Klausel benutzt werden kann ein sogenannter ‚Composite'-Index erstellt werden. Das bedeutet der Index wird nicht über eine einzelne Spalte sondern über eine Gruppe von Spalten erzeugt. Ist der volle Informationsgehalt einer Abfrage im Index enthalten, kann der Optimizer über einen Index-Fast-Full-Scan[1] die Abfrage sehr effizient durchführen. Weiter besteht für den Optimizer auch die Möglichkeit Index-Range-Scans durchzuführen oder eine Sortierung über den Index vorzunehmen.

1.3.1 Local Index

Der Index ist gleich partitioniert wie die dazugehörige Tabelle. Dadurch wird die Administration gegenüber einem globalem Index wesentlich erleichtert. Administrationsaufgaben können isoliert auf nur einer Partition ausgeführt werden. Es werden weder die anderen Tabellenpartitionen noch die Indexpartititionen beeinträchtigt.

1.3.2 Prefixed Index

Bei diesem Index entspricht das erste Indexelement dem Partitionierungskey. Dieser Index kann als Unique-Index definiert werden. Das Indexprobing ist etwas günstiger, da nicht alle Indexbäume durchsucht werden müssen[2]. Man sagt diesem Index ein etwas schlechteres Verhalten gegenüber nonprefixed Index bezüglich Parallelisierung nach.

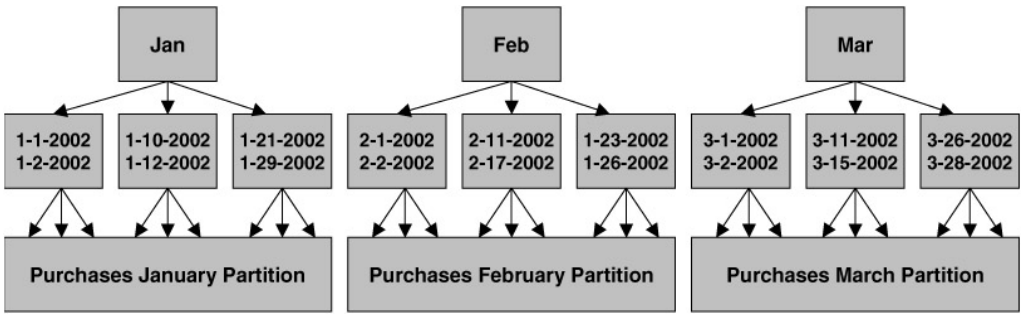


Abbildung 3: Prefixed-Local-Index

[1] Hint index_ffs
[2] Indexpartition-pruning

1.3.2.1.1 Nonprefixed Index

Bei diesem Index ist der Partitionierungskey nicht das erste Indexelement oder gar nicht vorhanden. Dieser Index kann nur Unique sein, wenn der Partitionierungskey Teil der indexierten Felder ist. Index-Partition-Pruning funktioniert wenn der Partitionierungskey teil der Query-Einschränkung ist.

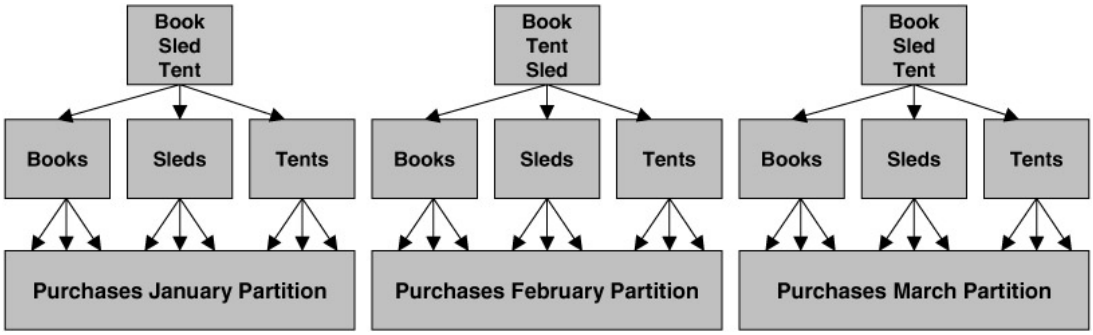


Abbildung 4: Nonprefixed-Local-Index

1.4 Global Index

Der Index ist nicht gleich partitioniert bzw. nicht partitioniert.
Hier geht der gewonnene administrative Vorteil zu einem grossen Teil wieder verloren.
Der Einsatz vom globalen Index muss also gut überlegt werden. Von Oracle werden nur die Global prefixed Indexe unterstützt.

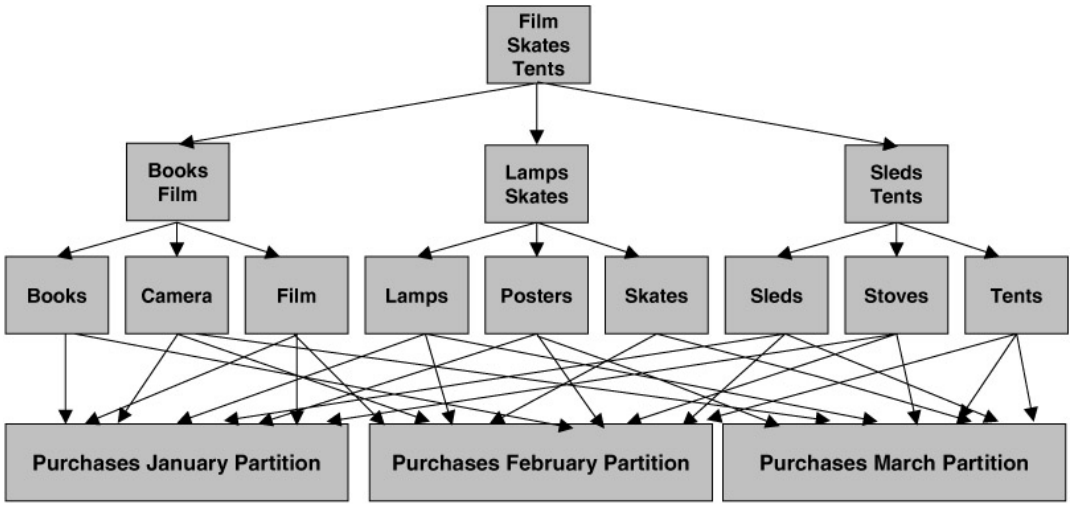


Abbildung 5: Globaler Index

1.4.1 Global partitioned Index

Der Index ist nach einem anderen Schlüssel partitioniert als die darunter liegende Tabelle. Jede Operation, welche auf nur einer Partition ausgeführt wird hat Einfluss auf den ganzen Index. So können Administrationsaufgaben nicht mehr isoliert auf nur einer Partition ausgeführt werden, da diese den ganzen Index invalidiert.

1.4.2 Global nonpartitoned Index

Der Index ist nicht partitioniert. Verhält sich im Wesentlichen gleich wie ein normaler Index. Aber jede Operation, welche auf nur einer Partition ausgeführt wird hat Einfluss auf den ganzen Index. So können Administrationsaufgaben nicht mehr isoliert auf nur einer Partition ausgeführt werden, da diese den ganzen Index invalidiert.

1.5 Bitmap Index

Bitmap Index verfolgen einen ganz anderen Ansatz als die B*Tree-Indexes. Ein Bitmap-Index ist dann gut wenn die Kardinalität klein ist. Beispielsweise für Spalten wie ‚Geschlecht', wo nur zwei unterschiedliche Werte vorkommen.

Row	Geschlecht	Männlich	Weiblich
1	M	1	0
2	W	0	1
3	M	1	0
4	M	1	0
5	M	1	0
6	W	0	1
7	W	0	1
8	M	1	0
9	W	0	1

Tabelle 1: Beispiel eines Bitmap-Indexes

Bitmap-Indexe sollten nicht aus zusammen gesetzten Spalten bestehen. Möchte man mehr als eine Spalte im Index haben, dann sollte man für jede einzelne Spalte je einen Index erstellen. Ein Single-Column Bitmap-Index ist schneller als ein Composite-Column Bitmap-Index. Der Optimizer ist in der Lage einzelne Bitmap-Indexe ‚AND' oder ‚OR' zu verknüpfen.
Ein Bitmap Index speichert für jede Rowid ein Bitmuster. Ein Bitmap-Index benötigt viel weniger Platz als ein B*Tree-Index. Ein Bitmap-Index kann nicht global sein, sondern nur local partitioniert.



Abbildung 6: Bitmap-Index

Ein grosser Nachteil des Bitmap-Index ist ihr Locking-Verhalten. Beim Bitmap-Index wird nicht auf Row-Level gelockt sondern auf Block-Level. Da nun in einem Block mehrere Rows referenziert sind, werden mehrere Rows gleichzeitig gelockt. Man spricht hier von Page-Locking. Der Bitmap-Index hat also ein schlechteres Verhalten bei mehreren parallelen Datenmanipulationen. Ein Bitmap-Index kann gleich wie ein B*Tree-Index auch compressed sein. Dies reduziert den benötigten Platz nochmals massiv, da ein Bitmap-Index typischerweise viele gleiche Werte beinhaltet. Diese geht natürlich auf Kosten der CPU und die Maintainance-Arbeiten werden wesentlich aufwendiger.

1.6 Bitmap Join Index

Dies ist der einzige Index, der über mehrere Tabellen geht. Dieser Index speichert das Resultat einer Join-Operation zwischen n Tabellen.

```
SELECT sum(p.purchase_price)
FROM purchase p, customer c
WHERE p.customer_id = c.customer_id
AND c.sex = 'F'
```

In diesem Beispiel wäre mal ein Bitmap-Index auf der Spalte ‚sex' schon sehr optimal. Die Join-Operation müsste aber dennoch durchgeführt werden. Hier springt der Bitmap-Join Index in die Lücke.
Man kann nun folgenden Index erstellen:

```
CREATE BITMAP INDEX JOIN_Pu_Cc
ON
purchase (customer.sex)
FROM purchase, customer
WHERE purchase.customer_id = customer.customer_id
```

Der Bitmap-Join-Index wird zwar auf der Tabelle ‚purchase' erstellt, aber über die Spalte ‚sex' der Tabelle ‚customer'. In diesem Index ist also das gesamte Resultat der ‚where'-Bedingung gespeichert[3].

[3] Eine Alternative dazu wäre ‚Materialized View'