Oracle9i

Oracle実践研修2

INDEX活用

2007.10.18

カリキュラムの確認

- インデックス使用の目的 0.5時間
- 種類と特徴 1時間
- インデックスの使用状況と
  チューニングの基礎 2時間
- インデックスが使用される条件 0.5時間
- 断片化と再作成 1時間
- チューニング(基本)実習 1時間

インデックス使用の目的

- 表の行に高速アクセスするため
- 問い合わせの高速化
  更新については、その限りではない

インデックスの概要

Table

	EMPNO	ENAME	JOB	MGR	HIREDATE
1	7369	SMITH	CLERK	7902	1980/12/17
2	7499	ALLEN	SALESMAN	7698	1981/2/20
3	7521	WARD	SALESMAN	7698	1981/2/22
4	7566	JONES	MANAGER	7839	1981/4/2
5	7654	MARTIN	SALESMAN	7698	1981/9/28
6	7698	BLAKE	MANAGER	7839	1981/5/1
7	7782	CLARK	MANAGER	7839	1981/6/9
8	7788	SCOTT	ANALYST	7566	1987/4/19
9	7839	KING	PRESIDENT		1981/11/17
10	7844	TURNER	SALESMAN	7698	1981/9/8
11	7876	ADAMS	CLERK	7788	1987/5/23
12	7900	JAMES	CLERK	7698	1981/12/3
13	7902	FORD	ANALYST	7566	1981/12/3
14	7934	MILLER	CLERK	7782	1982/1/23

Index

	ENAME
11	ADAMS
2	ALLEN
6	BLAKE
7	CLARK
13	FORD
12	JAMES
4	JONES
9	KING
5	MARTIN
14	MILLER
8	SCOTT
1	SMITH
10	TURNER
3	WARD

インデックスの一般知識

- 一意索引と非一意索引
- コンポジット索引(複数列にまたがる索引)
- Nullは索引化されない
- すでにデータがある状態でインデックスを作成する時、ソート領域を使用(SORT_AREA_SIZE、一時表領域)

インデックスを作るとよいケース

- 大きな表で頻繁に検索される行の割合が15% 未満の場合
- 複数の表の結合に使用される列に
- インデックスの候補
  列の値が比較的一意
  値の範囲が広い→B*Tree索引
  値の範囲が狭い→ビットマップ索引

全表走査との比較

- 全表検索
  上から下まで
- インデックス検索
  インデックス部を検索し、見つかったレコードのROWIDで、テーブルを検索する。
- よって、小さい表ではインデックス検索より、全表検索が速い

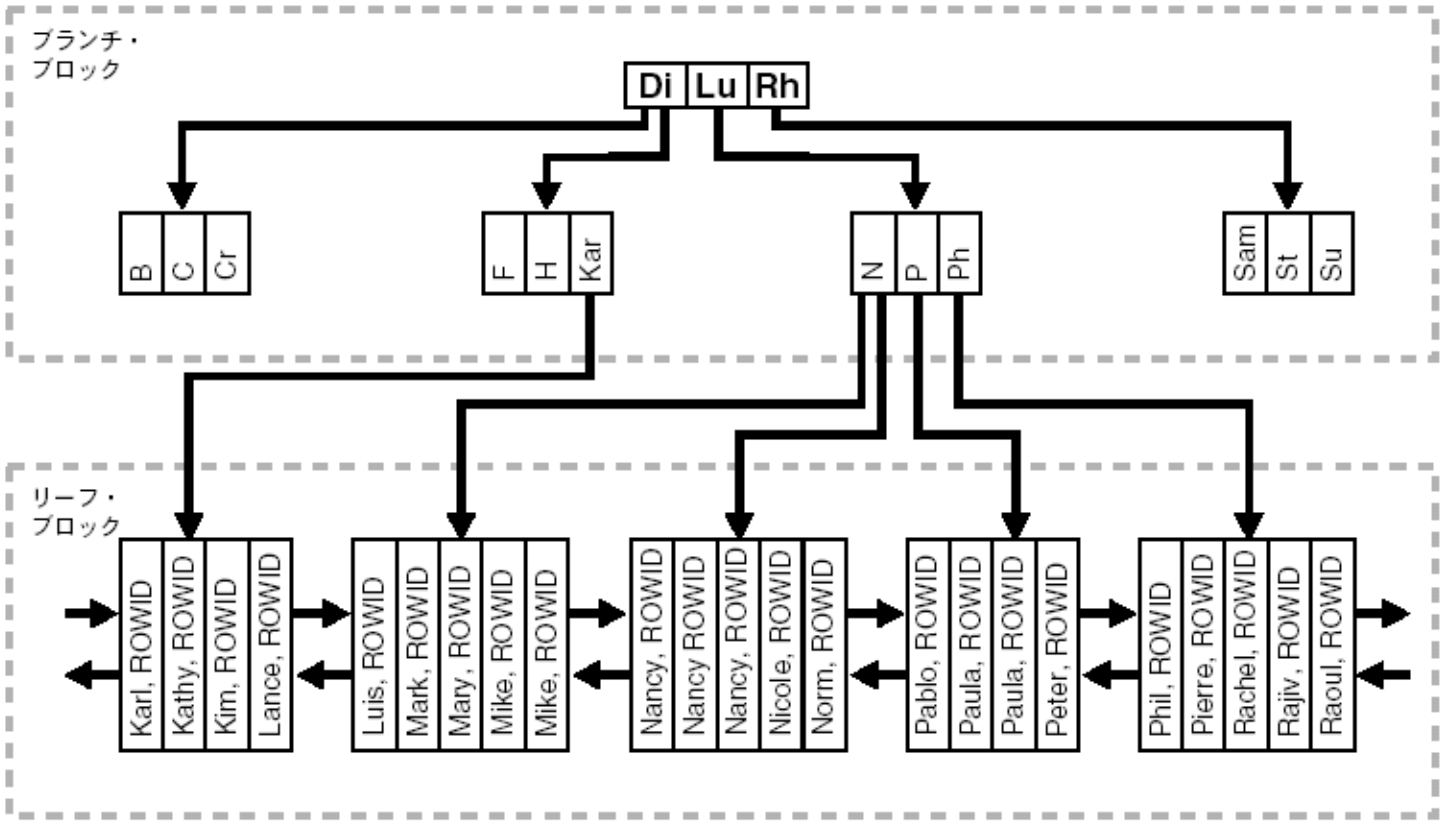
更新処理とのトレードオフ

- インデックスは順番に並んでいる
- 新規のレコードがテーブルにInsertされると、インデックスも更新される
- インデックスを多数持つテーブルでは、更新処理が重くなる
- 検索の速さをとるか、更新の速さをとるか

種類と特徴

B*Treeインデックス1

図 10-7 B ツリー索引の内部構造



B*Treeインデックス2

- ブランチ・ブロック
  下位レベルの索引ブロックを指す索引データが含まれる
- リーフ・ブロック
  すべての索引対象のデータ値と、実際の行を検出するためのROWID が含まれる

B*Treeインデックス3

- 索引内のどの位置のレコード検索も、所要時間はほぼ同じ
- 自動的にバランスが保たれる
- 平均して、すべてのブロックの4 分の3 が満たされる
- 完全一致や範囲検索など、広範囲な問合せに対して、優れた検索パフォーマンスを提供
- 挿入、更新および削除が効率的で、高速検索のためにキー順序が維持される
- 小さな表と大きな表のどちらでも優れているため、表のサイズが大きくなっても低下しない

B*Treeインデックス4

- 検索手順~Paula を検索する場合
  - ルート・ブロックで、x＜Paula＜＝y となるxを見つける。→Luを選ぶ
  - Luのリンクをたどって、ブランチ・ブロック（Mo、P、Ph）に到達。Ph が>= Paula となる最小キー。→Pを選ぶ
  - Pのリンクをたどって、リーフ・ブロック（Pablo、Paula、Paula、Peter）に到達。このブロックで、Paulaを探す。
  - 見つかった場合は、(KEY, ROWID) を戻す。

ビットマップインデックス1

- 各キー値のビットマップを使用
- Enterprise Editionだけで使用可

ビットマップインデックス2

- 大量データと非定型問合せを扱い、同時実行トランザクションのレベルは高くないデータ・ウェアハウス・アプリケーションに対して効果的
  -多くの種類の非定型問合せの応答時間が短縮
  -使用領域が実質的に縮小される
  -機能の低いハードウェアでもパフォーマンスが劇的に向上
  -パラレルDML とロードを効果的に実行

ビットマップインデックス3

- 大小の比較による問合せの対象とされる列には適さない
- AND、OR、NOT、等価問合せで有効
- カーディナリティの低い列(表内の行数に比べて個別値の数が少ない列)において、最も効果大
  -列の個別値の数が表内の行数の1% より少ない場合・・Yes/Noのみとか A/B/Cのどれかだけとか
- カーディナリティが高い列でも、WHERE 句で複雑な条件に含まれることが頻繁にある場合

ビットマップインデックス4

表 10-1 ビットマップ索引の例

CUSTOMER #	MARITAL_ STATUS	REGION	GENDER	INCOME_ LEVEL
101	single	east	male	bracket_1
102	married	central	female	bracket_4
103	married	west	female	bracket_2
104	divorced	west	male	bracket_4
105	single	central	female	bracket_2
106	married	central	female	bracket_3

- MARITAL_STATUS、REGION、GENDER 、INCOME_LEVEL は、カーディナリティの低い列。可能な値は、MARITAL_STATUS、REGION は3 つ、GENDERは2 つ、INCOME_LEVELは4 つ。そのため、これらの列にはビットマップ索引が有効。一方、CUSTOMER# はカーディナリティの高い列であるため、一意のB ツリー索引を使用する方が、検索が効率的。

ビットマップインデックス5

表 10-2 サンプル・ビットマップ

Customer	REGION='east'	REGION='central'	REGION='west'
101	1	0	0
102	0	1	0
103	0	0	1
104	0	0	1
105	0	1	0
106	0	1	0

- ビットマップの各ビットはCUSTOMER 表の1 つの行に対応しており、各ビットの値は対応する行の値に依存。Customer=101はREGIONがeastなのでビットマップREGION='east'のビットマップは1。REGION の値がeast の行は他にないため、ビットマップREGION='east' の残りのビットは0。

ビットマップインデックス6

図 10-11 ビットマップ索引を使用した問合せの実行

status = 'married'		region = 'central'		region = 'west'				
0		0		0		0		0
1		1		0		1		1
1	AND	0	OR	1	=	1	AND	1	=	1
0		0		1		0		1		0
0		1		0		0		1		0
1		1		0		1		1		1

- SELECT COUNT(*) FROM CUSTOMER WHERE MARITAL_STATUS = 'married' AND REGION IN ('central','west');

結果が"1"のレコードが該当

ビットマップインデックス7

- Null
  Null値があってもインデックスができる
  つまり、is Nullという条件でもインデックスが使用される
- B*TreeとBitmapの比較

B*TreeとBitmapの比較

テスト環境

K2：2種類の値だけ取るカラム(1、2)
K3：3種類の値だけ取るカラム(1、2、3)
K1K：1000種類の値を取るカラム
K500K：500000種類の値を取るカラム

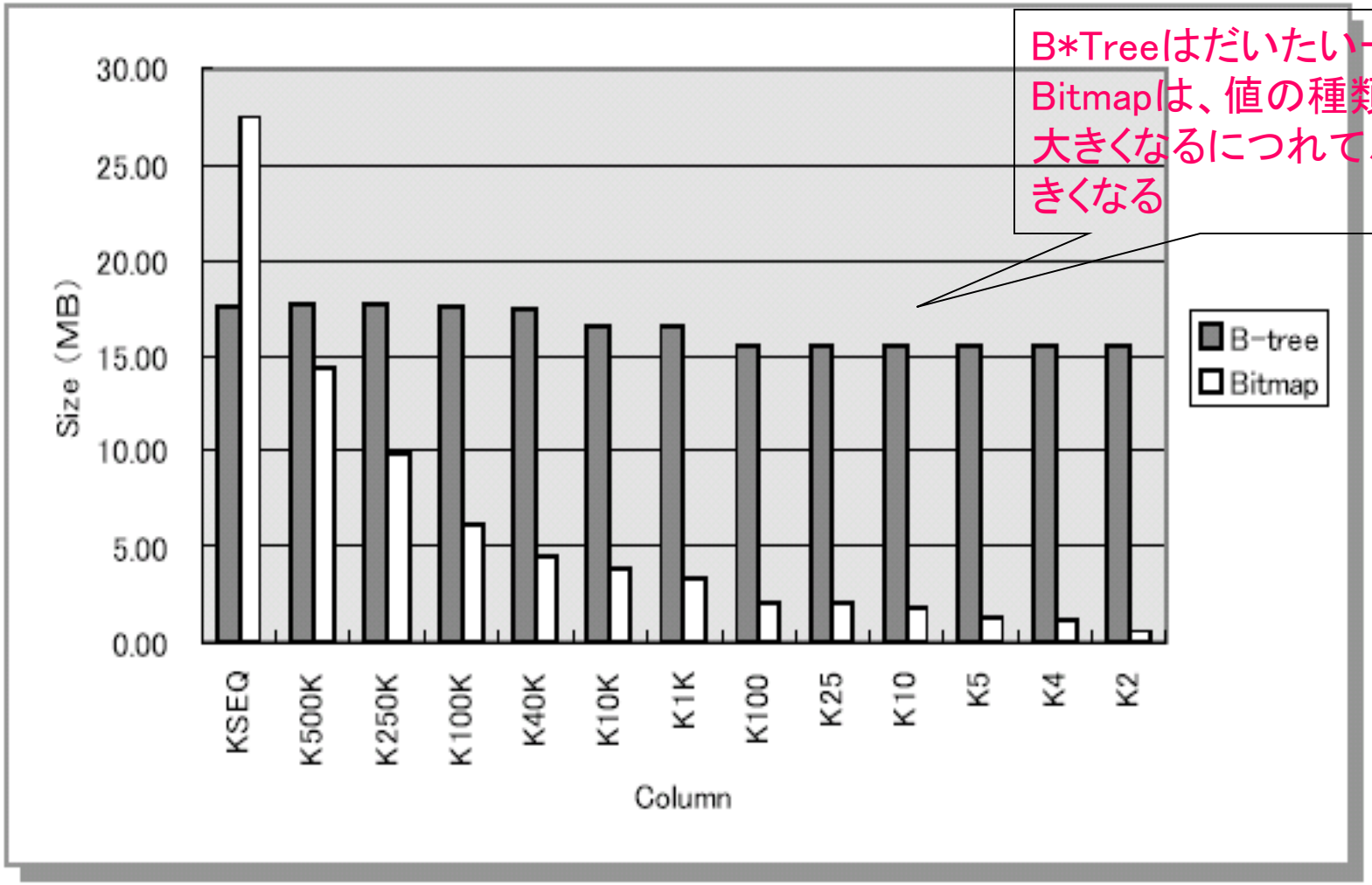
＜BENCHテーブルの定義＞

ビットマップインデックス作成

KSEQ	K500K	K250K	K100K	K40K	K10K	K1K	K100	K25	K10	K5	K4	K2	S1	.....	S8
1	16808	225250	50074	23659	8931	273	45	4	4	5	1	2	12345678	.....	12345678900987654321
2	484493	243043	7988	2504	2328	730	41	13	4	5	2	2	12345678	.....	12345678900987654321
3	129561	70934	93100	279	1817	336	98	2	3	3	3	2	12345678	.....	12345678900987654321
4	80980	129150	36580	38822	1968	673	94	12	6	1	1	2	12345678	.....	12345678900987654321
5	140195	186358	35002	1154	6709	945	69	16	5	2	3	2	12345678	.....	12345678900987654321
6	227723	204667	28550	38025	7802	854	78	9	9	4	3	2	12345678	.....	12345678900987654321
7	28636	158014	23866	29815	9064	537	26	20	6	5	2	2	12345678	.....	12345678900987654321
8	46518	184196	30106	10405	9452	299	89	24	6	3	1	1	12345678	.....	12345678900987654321
9	436717	130338	54439	13145	1502	898	72	4	8	4	2	2	12345678	.....	12345678900987654321
10	222295	227905	21610	26232	9746	176	36	24	3	5	1	1	12345678	.....	12345678900987654321

200B × 100万行 = 200MB

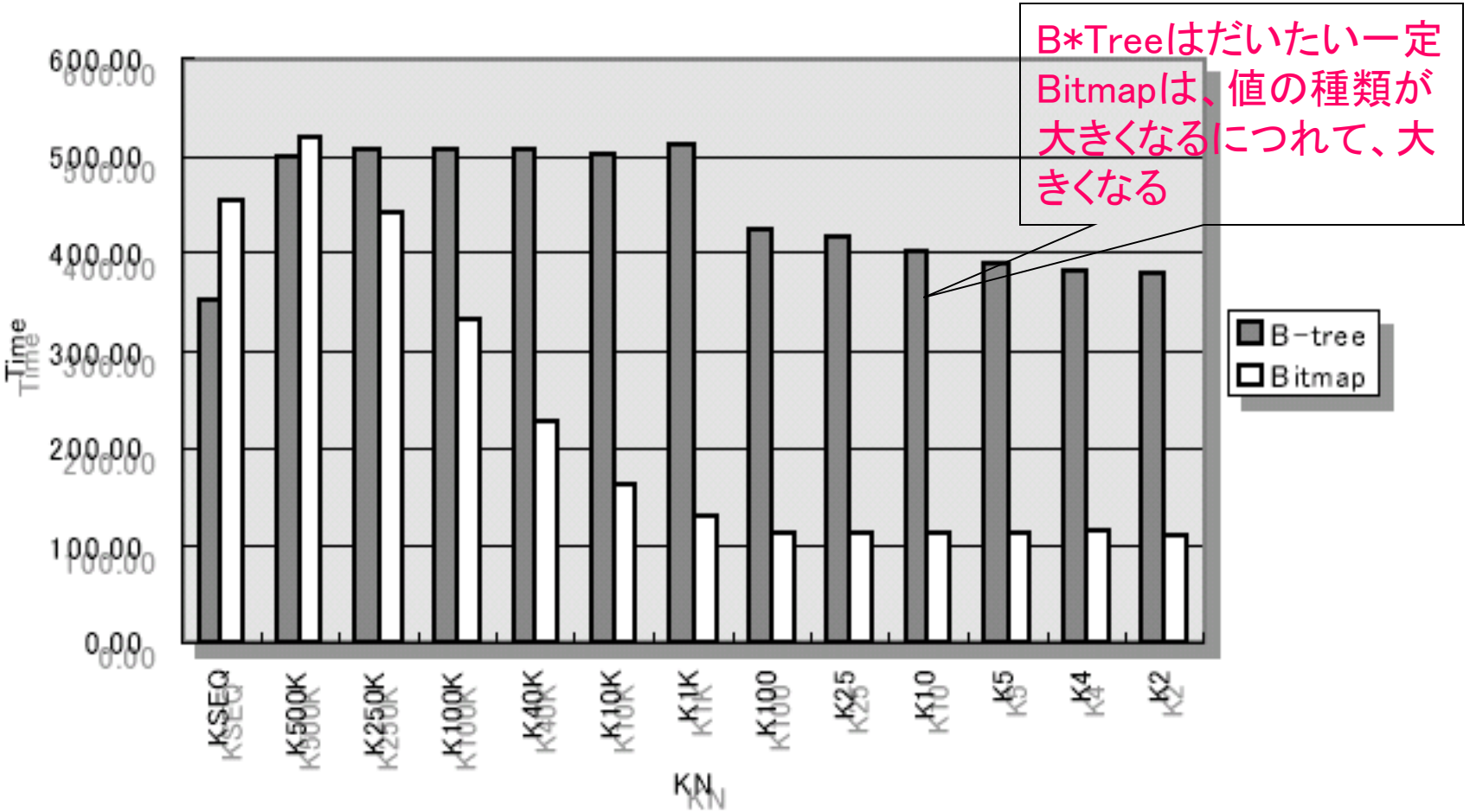
B-Tree vs Bitmap（Size）



(注) このデータはある特定の条件下でのテスト結果です
あくまでも1つの例としてお考え下さい

B-Tree vs Bitmap（作成時間）

create_bitmap_area_size = 8MB



(注) このデータはある特定の条件下でのテスト結果です
あくまでも1つの例としてお考え下さい
Timeは係数をかけた時間です。

Q1 カウント、単一条件一致検索

```
SELECT COUNT(*) FROM BENCH
WHERE KN = 2;
```

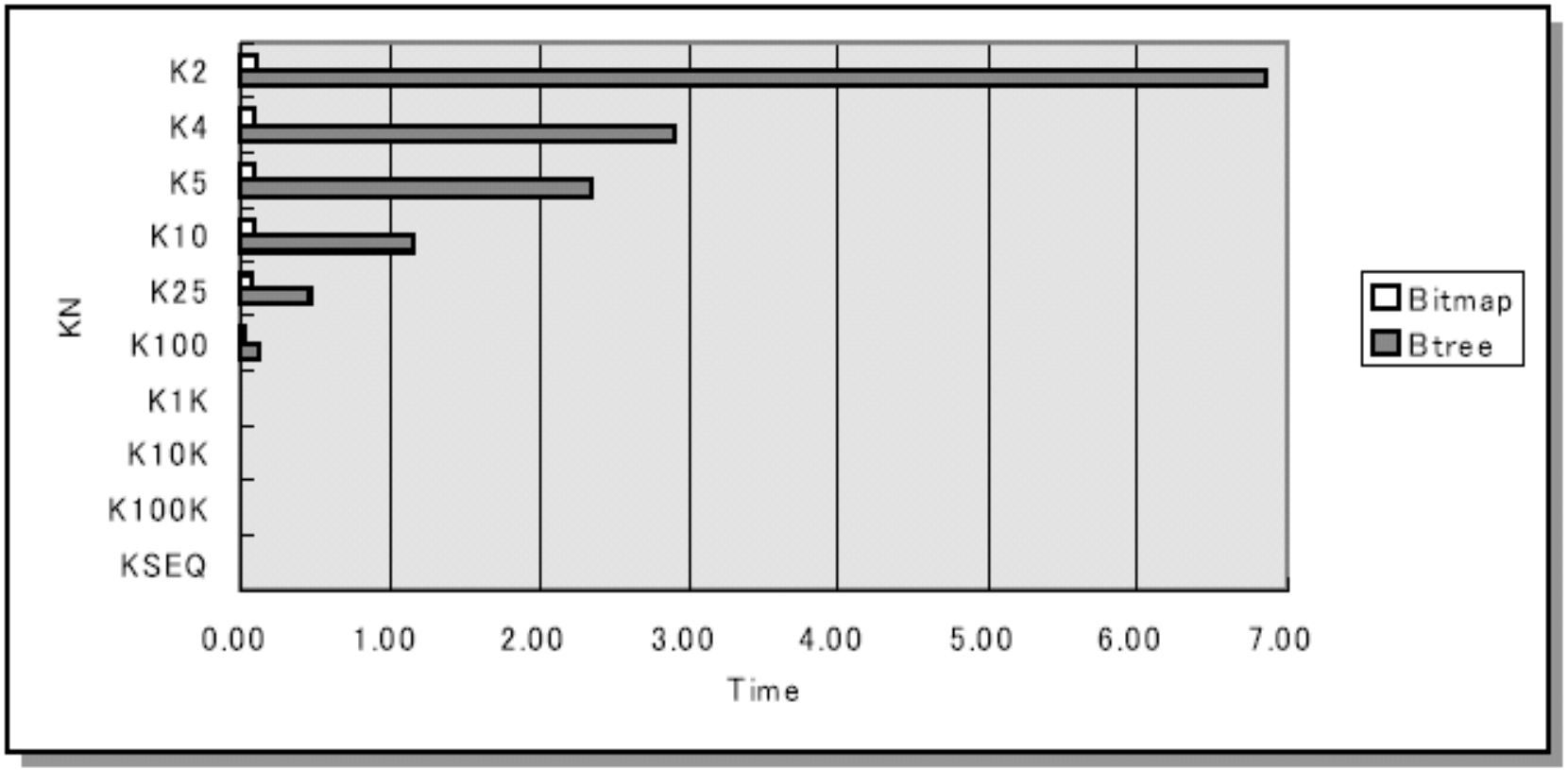
[各KNは{KSEQ, K100K, ..., K4, K2}]

Q1 測定結果

```
SELECT COUNT(*) FROM BENCH
WHERE KN = 2;

[各KNは[KSEQ, K100K, ..., K4, K2]]
```

	KSEQ	K100K	K10K	K1K	K100	K25	K10	K5	K4	K2
Btree	0.01	0.01	0.01	0.02	0.12	0.47	1.16	2.35	2.90	6.85
Bitmap	0.01	0.01	0.01	0.02	0.04	0.06	0.08	0.08	0.08	0.10



(注) このデータはある特定の条件下でのテスト結果です
あくまでも1つの例としてお考え下さい

Timeは係数をかけた時間です

Q2　一つの一致検索条件と一つの一致検索条件の否定(NOT)の論理積

```
SELECT COUNT(*) FROM BENCH
WHERE K2 = 2 AND NOT KN = 3;
```

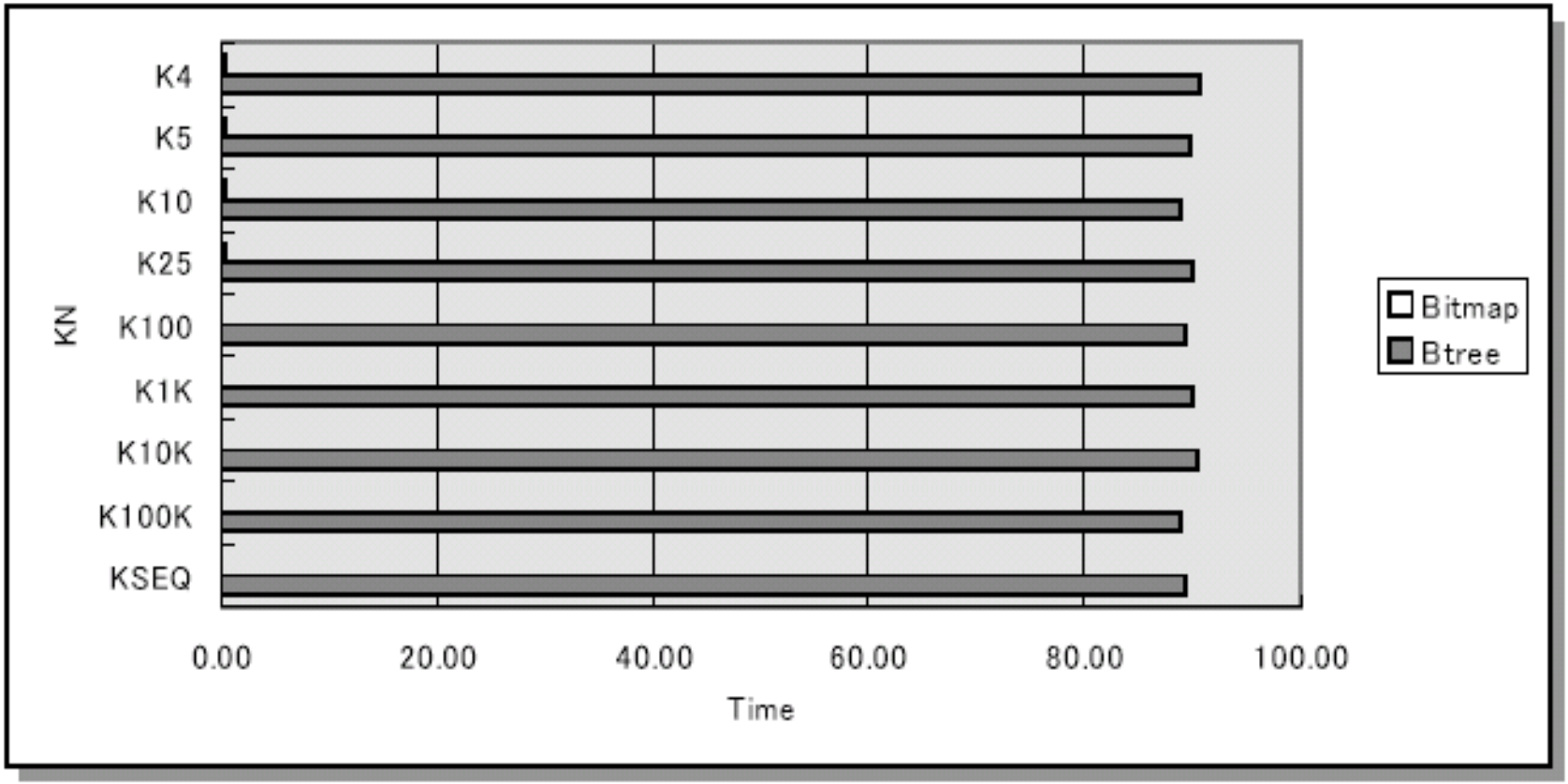
[各KNは{KSEQ, K100K, ..., K4}]

Q2　測定結果

```
SELECT COUNT(*) FROM BENCH
WHERE K2 = 2 AND NOT KN = 3;

[各KNは{KSEQ, K100K, ..., K4}]
```

	KSEQ	K100K	K10K	K1K	K100	K25	K10	K5	K4
Btree	89.24	88.91	90.60	90.13	89.22	90.18	88.78	89.65	90.63
Bitmap	0.10	0.24	0.28	0.29	0.31	0.36	0.40	0.41	0.41



(注) このデータはある特定の条件下でのテスト結果です　　Timeは係数をかけた時間です
あくまでも1つの例としてお考え下さい

Q3 五つの条件の論理積

SELECT KSEQ, K500K FROM BENCH
WHERE 条件1 AND 条件2 AND 条件3
AND　条件4 AND 条件5;

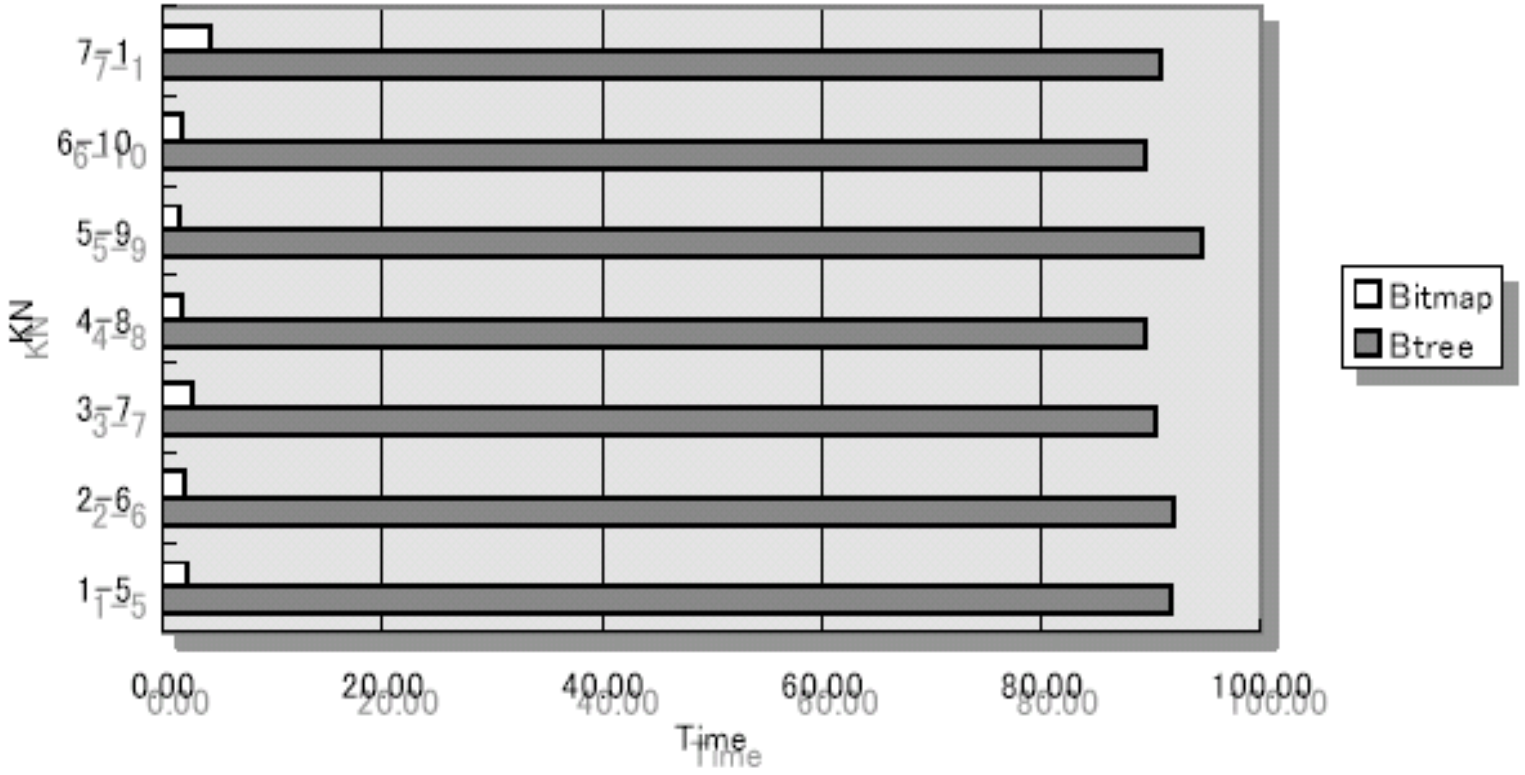
[各条件はリストから選択]

(1) K2 = 1
(2) K100 > 80
(3) K10K BETWEEN 2000 AND 3000
(4) K5 = 3
(5) (K25 = 11 OR K25 = 19)
(6) K4 = 3
(7) K100 < 41
(8) K1K BETWEEN 850 AND 950
(9) K10 = 7
(10) K25 BETWEEN 3 AND 4

Q3　測定結果

SELECT KSEQ, K500K FROM BENCH
WHERE 条件1 AND 条件2 AND 条件3
AND　条件4 AND 条件5;
[各条件はリストから選択]

	1-5	2-6	3-7	4-8	5-9	6-10	7-1
Btree	91.84	92.12	90.35	89.48	94.40	89.37	90.75
Bitmap	2.19	2.07	2.65	1.74	1.61	1.71	4.46



(注) このデータはある特定の条件下でのテスト結果です
あくまでも1つの例としてお考え下さい

Timeは係数をかけた時間です

ビットマップ・インデックス

〈長所〉

インデックスの格納領域の削減
検索速度の劇的な向上
論理オペレーション(ANDおよびOR)でも高いパフォーマンス
複数の索引に対する検索に効果大
〈 カーディナリティが少ないほど効果大 〉

〈短所〉

更新処理に弱い

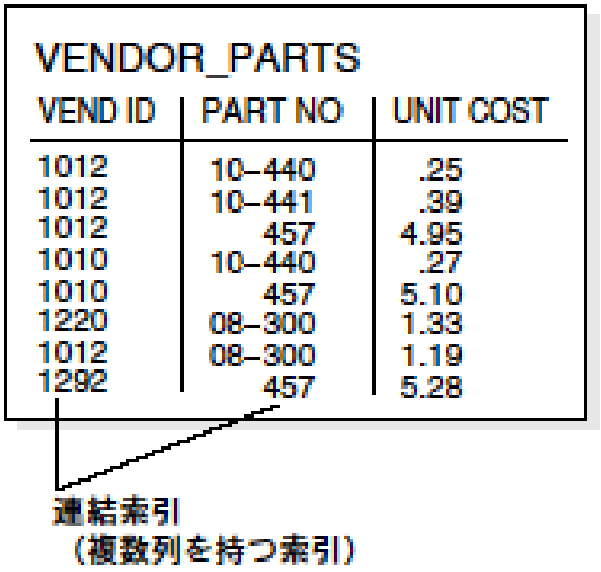
DSSシステムに有効 !!

インデックス(その他1)

- コンポジット索引
  - 複数列につける

B*Treeでも
Bitmapでも
作成可

図 10-6 コンポジット索引の例



その他インデックス2

- ファンクション・ベース索引
  - ファンクション、式の値が格納される
    - CREATE INDEX idx ON table_1 (a + b * (c - 1));
  - 次のような問合せを処理するときに、この索引を使用できる
    - SELECT a FROM table_1 WHERE a + b * (c - 1) < 100;

B*Treeでも
Bitmapでも
作成可

その他インデックス3

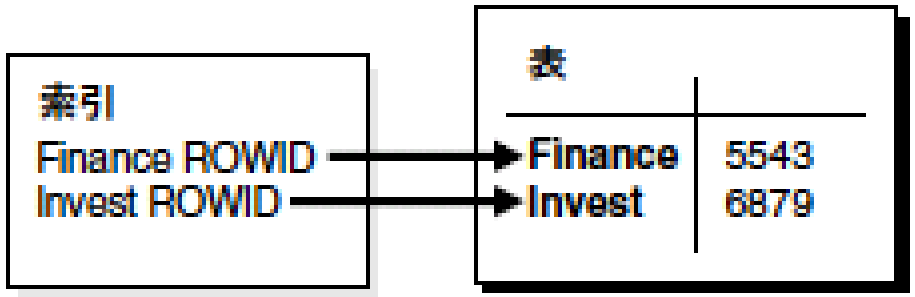
- 逆キー索引(B*Treeのみ)
  - 列の順序は保ちながら、索引の各列（ROWID を除く）のバイトを逆にする
  - 索引に対する変更が少数のリーフ・ブロックに集中するOracle9i Real Application Clusters では、この機能によりパフォーマンスの低下を防ぐことができる
  - キーを逆にすることにより、挿入値は索引のリーフ・キー全体に分散される
  - レンジスキャン使用不可

索引構成表

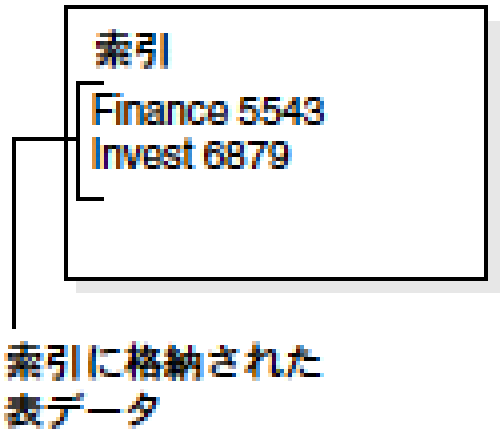
- 索引構成表のデータは主キーによるソート形式でB ツリー索引構造に格納される(通常はデータが順不同のコレクションとして格納)
- B ツリーの各索引エントリには、索引構成表の行の主キー列値のみでなく、非キー列値も格納される

索引構成表イメージ

通常の表と索引



索引構成表



インデックス作成1

B*Tree

CREATE INDEX インデックス名
ON テーブル名(カラム名)
TABLESPACE テーブルスペース名
[STORAGE (INTIAL xx,NEXT xx,PCTFREE yy,PCTUSED yy]);

Bitmap

CREATE BITMAP INDEX….

xx:サイズ
yy:割合

インデックス作成(補足1~テーブルも同じ)

- INITIAL、NEXT
  - ローカル管理では重要ではない
  - ディクショナリ管理では重要

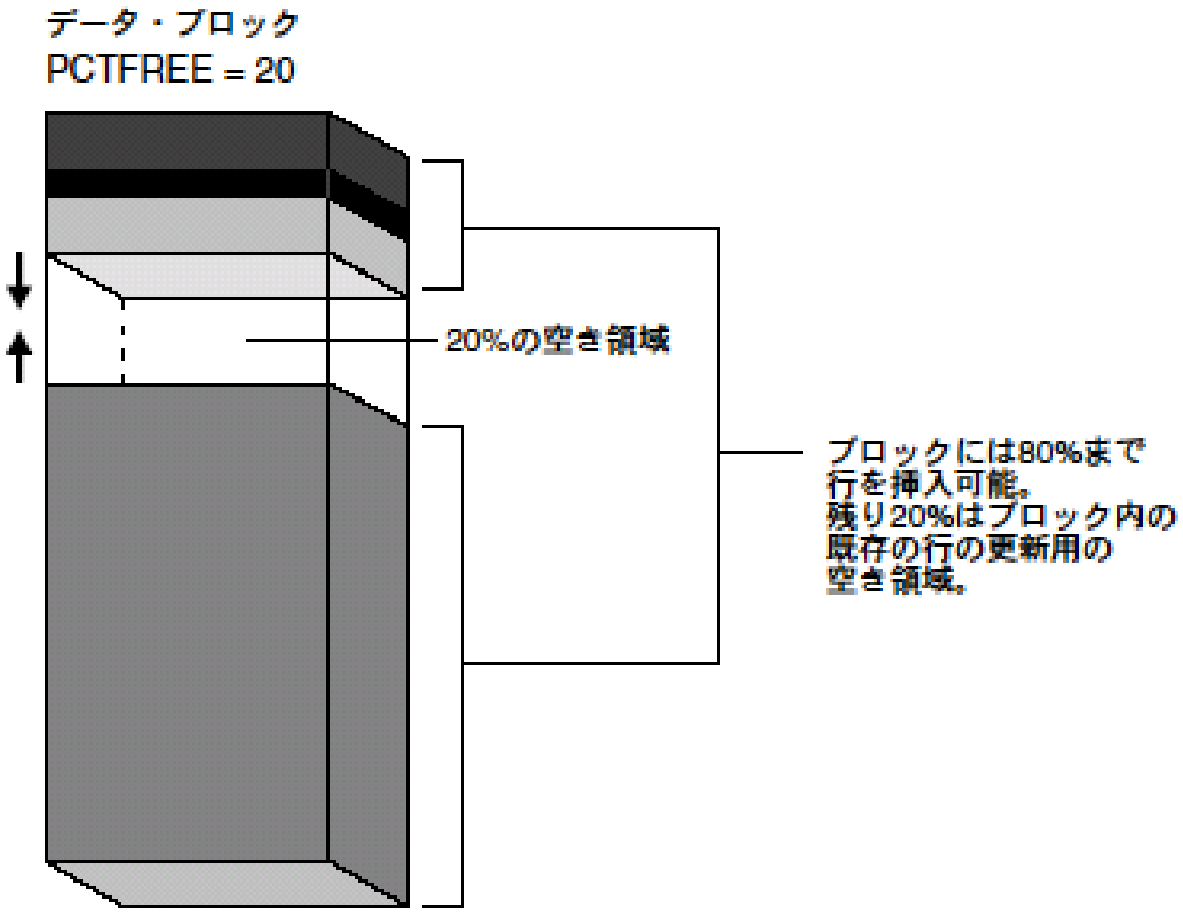
非推奨

- PCTUSED
  - 空き領域自動管理では不要
  - 空き領域手動管理では指定

非推奨

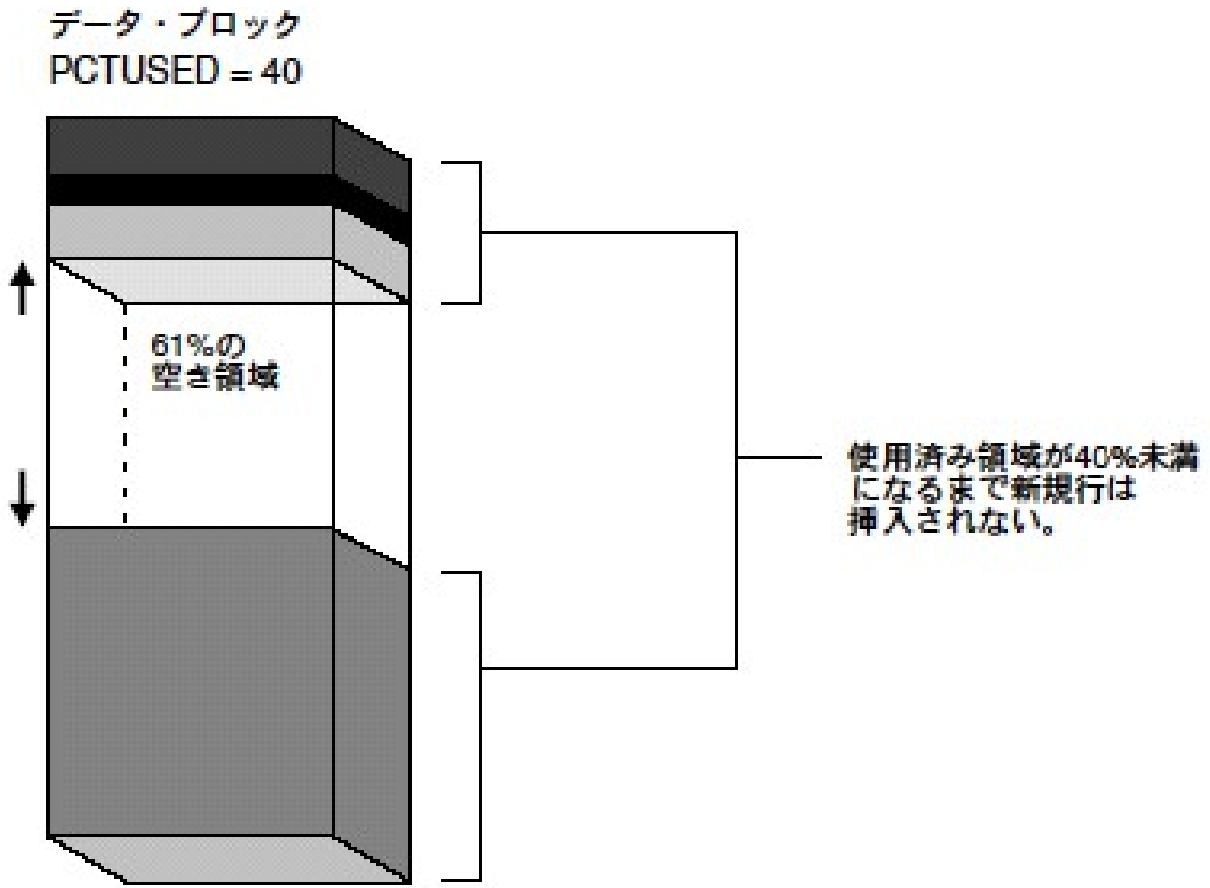
インデックス作成(補足2)

図 B-6 PCTFREE



インデックス作成(補足3)

図 B-7 PCTUSED



インデックス作成2

例

emp 表のename 列に対してemp_enameという名前の索引を作成

CREATE INDEX emp_ename ON emp(ename)

TABLESPACE users

STORAGE (INITIAL 20K NEXT 20k

PCTINCREASE 75) PCTFREE 0;

索引に記憶域オプション（INITIALやNEXTなど）を指定しない場合、デフォルトの表領域または指定された表領域のデフォルトの記憶域オプションが自動的に使用される(ディクショナリ管理の場合)→「領域見積」テキストP46参照

インデックス作成3

例

一意索引を作成するには、CREATE UNIQUE INDEX 文を使用する

```
CREATE UNIQUE INDEX dept_unique_index
ON dept (deptno,dname)
TABLESPACE indx;
```

explain plan

- 準備
  %ORACLE_HOME
  %\rdbms\admin\utlxplan.sqlを実行して
  PLAN_TABLEを作成する
- オプティマイザが選択した実行計画(どのように実行するか、Indexを使うか全件検索か)を表示
- 実習
  Excelシート

SQL Trace

- アプリケーションが実行するSQL 文の効率を正確に評価
- EXPLAIN PLAN の評価も同時にわかる
- TKPROFによって、判読可能なフォーマットに出力する
- 実習
  Excelシート

その他のツール

- 自動トレース
  - 実習Excelシート
- Oracle Trace
  - コマンドラインでotrccolを実行
  - 10gで廃止

インデックスが使用される条件

どんな場合にどのインデックスが使用されるか1

- SQLの書き方の問題
  - 索引の値とNULLの比較やNOTを使わない
  - 複合索引の先頭の列がないと複合索引は利用されない
  - 索引には計算をさせない
  - LIKE句を使った中間一致・後方一致検索

どんな場合にどのインデックスが使用されるか2

- オプティマイザ
  SQL文をどう実行するか、Oracleが判断し実行計画を作る～どのインデックスを使うか、全表走査をするか
- ルールベースとコストベースがある
  RBO；アクセスパスと優先順位による
  CBO；統計に基づきコストの低いもの
- インデックスが使われないケースがある

10gで廃止

ヒント

```
select /*+ INDEX(test_emp test_emp_idx) */
    count(*) from test_emp where deptno=99 and
    empno != 1111;
```

ALL_ROWS；スループットを優先した実行計画を選択させる

FIRST_ROWS；応答時間を優先させた実行計画を選択させる

FULL(table)；全表走査を選択させる

INDEX(table index)；指定された表に対して索引走査を選択させる

INDEX_DESC(table index)；索引走査を降順に行う

INDEX_COMBINE(table index)；ビットマップインデックスを選択させる

断片化1

1. 索引を使用しつづけると索引の構造バランスが崩れる
2. 領域が断片化する
3. パフォーマンスダウン
4. 索引の定期的な監視と保守が必要

断片化2

- 分析
  ANALYZE INDEX 索引名VALIDATE STRUCTURE;
  select * from index_stats;
  ※LF_ROWSに対する、DEL_LF_ROWの割合が高い場合、再作成を検討する。目安は5%。

インデックスの再作成

1. DROP INDEX→CREATE INDEX
2. ALTER INDEX 索引名REBUILD
   (10g)ALTER INDEX 索引名 COALESCE

・実習
→Excelシート

余裕があれば…

1. navis3_data_insert2.sqlでデータInsert
2. navis3_tuning1.sqlをチューニング
3. navis3_tuning3.sqlをチューニング

最後に

- OTN(http://otn.oracle.co.jp)を活用する!
- iSeminar(http://www.oracle.co.jp/direct/iseminar.html)を活用する!