Oracle Direct Seminar

ORACLE®

実践!! パフォーマンス・チューニング
- 索引チューニング編 - 【後編】

日本オラクル株式会社
Oracle Direct

Agenda

【前編】

- 索引構造の理解
- 索引を使用した検索
- オプティマイザによる索引走査/全表走査の判断
- ヒストグラムによる索引利用の効率化

【後編】

- 索引チューニングのポイント
  索引がうまく使われない
  4つのパターン
- 様々なタイプの索引

Agenda

- 索引チューニングのポイント
  索引がうまく使われない4つのパターン
  - 索引を使用する事で速くなる処理か？
  - 索引を利用できるSQL文か？
  - オプティマイザが索引利用を選択しているか？
  - 索引のメンテナンスをしているか？
- 様々なタイプの索引
  - ビットマップ索引
  - 複合索引
  - 逆キー索引
  - 索引構成表

索引がうまく使われない4つのパターン

索引を作成したのに検索が遅い！
＜索引チューニング　4つのチェックポイント＞

1. 索引を使用する事で速くなる処理か？
2. 索引を利用できるSQL文か？
3. オプティマイザが索引利用を選択しているか？
4. 索引のメンテナンスをしているか？

1. 索引を使用する事で速くなる処理か？
全表走査と索引走査

- 表のサイズや選択率（選択行数）によっては全表走査の方が速い可能性
  - 全表走査（フルスキャン）
    - 全てのデータを検索、比較する必要がある
    - マルチブロック READにより、少ないI/O回数でデータをみ込むことができる
  - 索引走査（インデックススキャン）
    - 索引から行アドレス(ROWID)を得て、単一のブロックを読み込むことができる

選択率が高い場合には全表走査が有利

単一行へのアクセスは索引走査が有利

	全表走査	索引走査
表のサイズ	小 表が大きいほど読み込むデータ量が増加するので、表が大きい場合には向かない	大 大量のデータがあっても読み込むデータは一部なので、大きい表にも有効
検索対象の割合	多 検索対象の割合が多い場合には、全てのブロックを読み込んだほうが効率的	少 大きな表から一部のデータを読み込む場合に効率的
読み込み方	マルチブロックREAD	シングルブロックREAD

1. 索引を使用する事で速くなる処理か？
【参考】索引作成の基準

- **索引作成の基準しきい値**
  注意：実際のしきい値はレコード長や各種条件によって異なるので、あくまでも参考値として考えてください

  ✓ レコード件数が1/50より絞り込める場合は索引をつける
  ✓ レコード数が1/50～1/10の絞り込みの場合は、速度の向上具合と、更新に伴う性能劣化を考慮して決める
  ✓ レコード数が1/10より絞り込めない場合は索引をつけない

厳密にチューニングするのであれば、全表走査時と索引走査時のタイムを測って、パフォーマンスの良い方を選択する

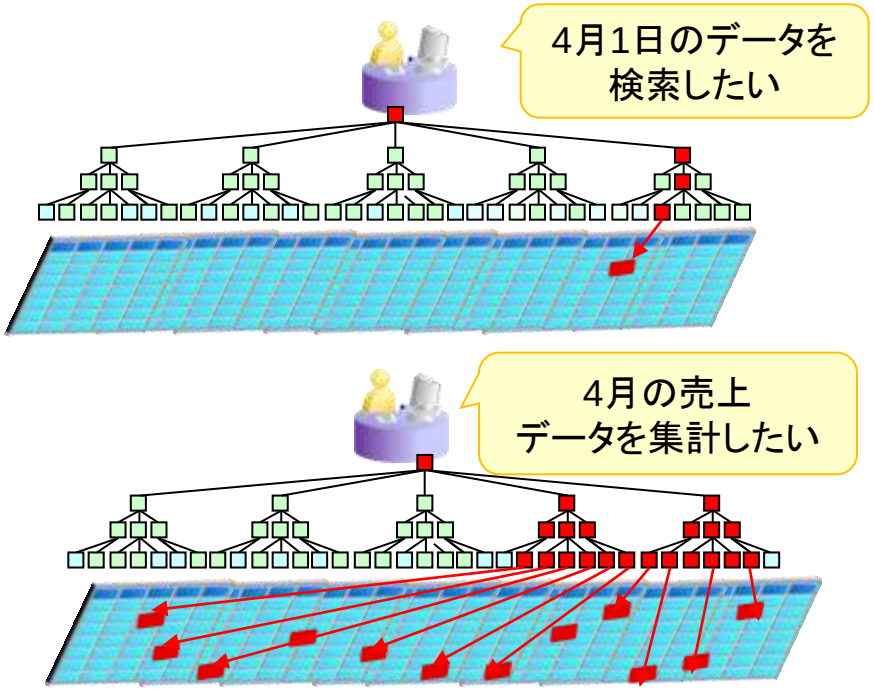
1. 索引を使用する事で速くなる処理か？

索引の限界

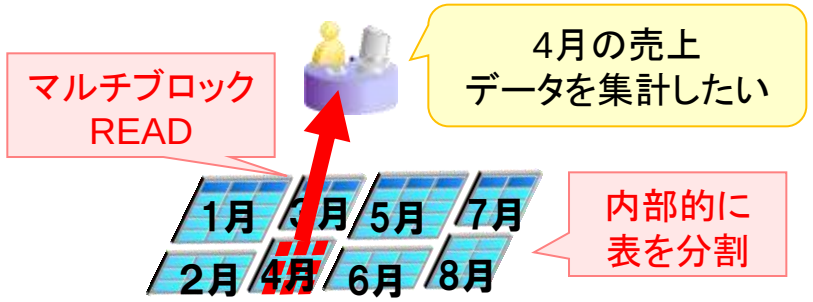
- **索引検索でもパフォーマンスが上がらないケース**
  - 表サイズが大きく、検索対象データも多い場合、索引と表に大量のアクセスが発生するため、パフォーマンスがあがらない

大量の索引読み込みが行われる検索では、索引を使ってもDISK I/Oがボトルネックになることも

パーティショニング機能で読み込みを効率化



パーティショニング機能とは
- 単一の表を内部的に複数の領域に分割して格納する機能
- 表名等は変わらないため、SQLの変更は不要



2. 索引を利用できるSQL文か？
索引が使われないSQL文

- SQLの構文によっては、索引があっても使用されない可能性
  - 索引列に対してNULL条件やNOT条件が使用されている場合
  - 条件式に計算を含む場合
  - LIKE条件を使った中間一致・後方一致検索をする場合

2. 索引を利用できるSQL文か？
NULL条件やNOT条件が使用されている場合

- IS NULLやIS NOT NULL条件を使用した検索例

COMM列に索引がある場合

```
SELECT NAME FROM EMP WHERE COMM = 0.1;
```

```
SELECT NAME FROM EMP WHERE COMM IS NULL ;
```

```
SELECT NAME FROM EMP WHERE COMM IS NOT NULL;
```

- NOT条件を使用した検索例

DEPTNO列に索引がある場合

```
SELECT NAME FROM EMP WHERE DEPTNO = 30 ;
```

```
SELECT NAME FROM EMP WHERE DEPTNO != 30 ;
```

2. 索引を利用できるSQL文か？
条件式に計算や関数を含む場合

- 列に計算式を含む検索例

SAL列に索引がある場合

```
SELECT NAME FROM EMP WHERE SAL*1.1 > 10000;
```

```
SELECT NAME FROM EMP WHERE SAL > 10000/1.1;
```

- 列に関数を使用する検索例

HIRE_DATE列に索引がある場合

```
SELECT NAME FROM EMP
WHERE TO_CHAR( HIRE_DATE,'DDMMYY')='010403';
```

```
SELECT ENAME FROM EMP
WHERE HIRE_DATE = TO_DATE('010403','DDMMYY');
```

2. 索引を利用できるSQL文か？
LIKE条件を使った検索をする場合

- 索引は昇順、降順にデータが並べられているため、前方一致には索引が使われるが、中間一致・後方一致では索引が使われない

- LIKE条件を使った検索例

SAL列に索引がある場合

<前方一致>

```
SELECT NAME FROM EMP WHERE NAME LIKE 'Su%';
```

<中間一致>

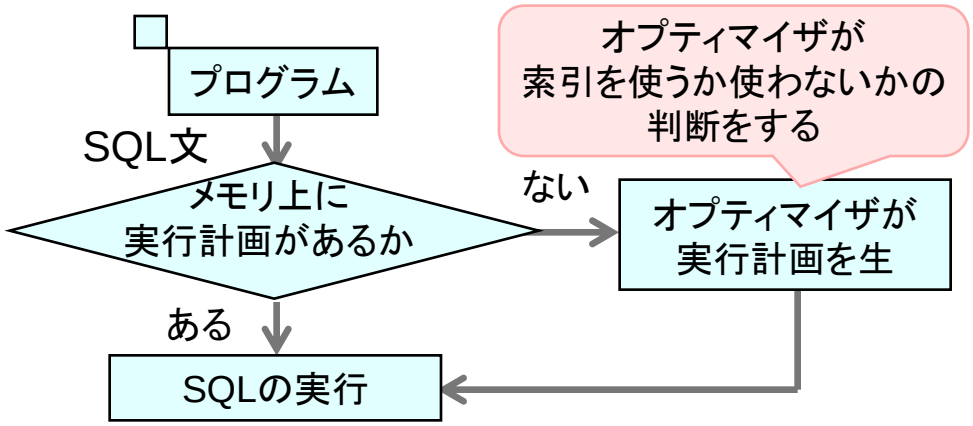
```
SELECT NAME FROM EMP WHERE NAME LIKE '%zu%';
```

<後方一致>

```
SELECT NAME FROM EMP WHERE NAME LIKE '%ki';
```

3. 適切な実行計画が選択されているか？
オプティマイザと実行計画

- オプティマイザが適切な実行計画を選択していない可能性
  - 実行計画を確認し、適切な実行方法を指定（ヒント）



参考　実行計画の調べ方
（SQL*PlusのAUTOTRACE機能）

1. SYSユーザでPLUSTRACEロールを作成し、SQLを実行するユーザに付与する。
   SQL> @%ORACLE_HOME%¥sqlplus¥admin¥plustrce.sql
   SQL> GRANT plustrace TO scott;
2. SQLを実行するユーザで実行計画を保存するための表(PLAN_TABLE)を作成する。
   SQL> connect scott/tiger
   SQL> @%ORACLE_HOME%¥rdbms¥admin¥utlxplan.sql
3. AUTOTRACE 機能を ON にし、SQL文を実行する。
   SQL> SET AUTOTRACE ON
   SQL> SELECT ...

3. 適切な実行計画が選択されているか？

効率的な実行計画が立てられない場合

- データの偏りなどにより、コストベース・オプティマイザが最適な実行計画を立てられない場合

```
SELECT name FROM emp WHERE deptno=20 AND job='営業';
```

<前提条件> EMPには10,000行
DEPTNO列とJOB列には索引がある
<統計情報> DEPTNO列のうちDEPTNO=20の占める割合は30%
JOB列の値のうち営業が占める割合は30%

コストベースオプティマイザの分析

予想されるI/O ⇒1万行×30%×30% = 900行（全体の9 %と判断）

⇒ コストを比較した結果JOB列の 索引走査 を選択

実際：営業のDEPTNOが20で、営業のほとんど人のDEPTNOが20だった場合
1万行 × 30% = 3000行（全体の30%）

⇒全表走査のほうが効率的

⇒ ヒントによるチューニング

3. 適切な実行計画が選択されているか？
ヒントによるチューニング（全表走査）

- ヒント：コストベースオプティマイザに、より良い実行方法を指定する手法
  - 適切な索引の使用や、結合順序の指定、処理の優先度（スループット重視 or OLTP向き）をSQL文の中で指定
  - /*+ と */ の間でヒントを指定し、SQLに直接埋め込む

```
SELECT name FROM emp WHERE deptno=20 AND job='営業';
```

実際：営業のDEPTNOが20で、営業のほとんど人の
DEPTNOが20だった場合
1万行 × 30% = 3000行（全体の30%）

⇒全表走査のほうが効率的

ヒントを使って全表走査を指定

全表走査を実行させる「ヒント」

```
SELECT /*+ FULL(emp)*/ NAME FROM emp
WHERE deptnp=20 AND job='営業';
```

3. 適切な実行計画が選択されているか？
ヒントによるチューニング（索引スキャン）

- /*+ INDEX(表 索引名) */
  - 特定の索引を使用するようオプティマイザに指示する
  - 一つの表に複数の索引がある場合、最適な索引を指示することができる

＜例＞ sales表の「cust_id_indx」索引を使用させる

```
SELECT /*+ INDEX(sales cust_id_indx) */
       sales_date,sales_amount
FROM   sales
WHERE  customer_id=100;
```

索引を指定する「ヒント」

3. 適切な実行計画が選択されているか？
ヒントによるチューニング（高速全索引スキャン）

- /*+ INDEX_FFS (表 索引…) */
  - 高速全索引スキャンを実行するようオプティマイザに指示する
  - 索引の中に求めるデータがすべてあるため、索引をマルチブロックREADで読み込むことで、必要なデータを取得することができる（表にアクセスする必要がない）

＜例＞ 高速全索引スキャン

（※）高速全索引スキャンは、索引キー内の列の少なくとも1つの列にNOT NULL制約がついている必要がある

```
CREATE TABLE emp
(empno  NUMBER(10)    PRIMARY KEY,
 name   VARCHAR2(30)  NOT NULL(※),
 salary NUMBER(10),
 deptno NUMBER(5));

CREATE INDEX name_idx
ON emp(name);
```

索引を読めば、NAMEのデータが取得できる（表へのアクセスは不要）

nameの一覧の問い合わせ

```
SELECT /*+ INDEX_FFS(emp name_idx) */ name
FROM emp;
```

3. 適切な実行計画が選択されているか？

ヒントを書くときの注意点

- ヒントの指定方法が間違っていた場合、ヒントは無視される
  - 索引名や表名が間違っていた場合
  - 表名に表別名が指定されていた場合

<例>

```
SELECT /*+ INDEX(e deptno_index) */
       name
FROM   emp e
WHERE  deptno=20;
```

ヒント内で表名「emp」を指定すると、ヒントが無視される

エラー出力はされないため、ヒント指定後は必ず実行計画を確認

- ヒント内で索引が複数指定されている場合は、最もコストが低い索引が使用される

- ヒントの使用により、実行計画が固定されてしまう
  - 最適な実行計画は、データの件数、値によって変わってくる可能性があるがヒントを指定することで実行計画が固定されてしまう

定期的なヒントの見直しが必要

4. 索引のメンテナンスをしているか？

索引メンテナンスの必要性

- 表のメンテナンスに伴う索引のメンテナンス
  - 表データを移動した場合（MOVEコマンド等）
  - 索引内のアドレスと行データのアドレスが異なり、索引が使用できない可能性
- 索引の断片化に伴うメンテナンス
  - データを追加、削除、変更した場合
  - 索引ブロックが断片化し、パフォーマンスが劣化する可能性
- 不要な索引のメンテナンス
  - 使われていない索引があることにより、DML文のパフォーマンスに悪影響を与える可能性

索引の定期的なメンテナンスが必要

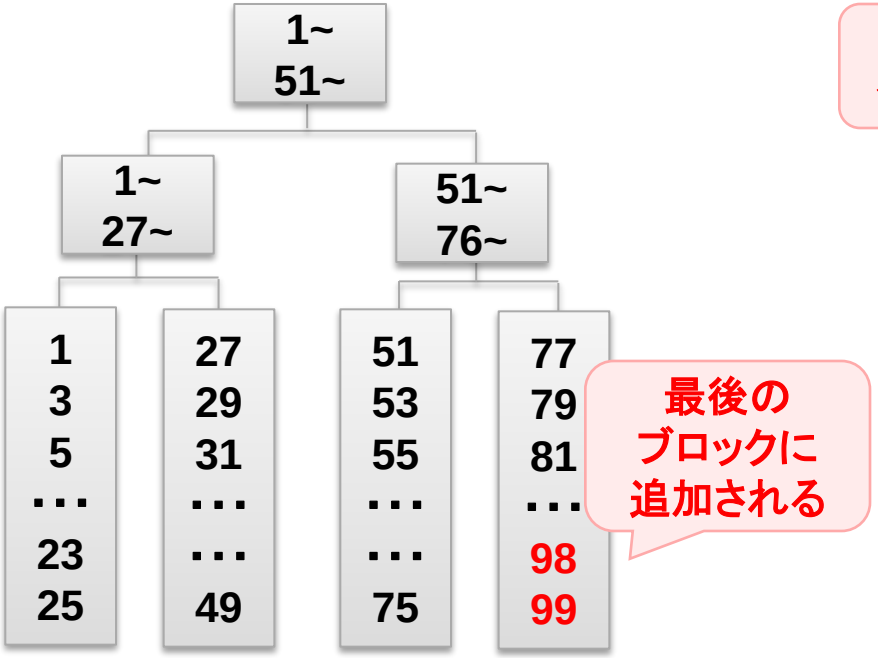
4. 索引のメンテナンスをしているか？

索引断片化の例（データを追加する場合）

- 昇順にデータを追加するよりもランダムに追加する方が断片化しやすい

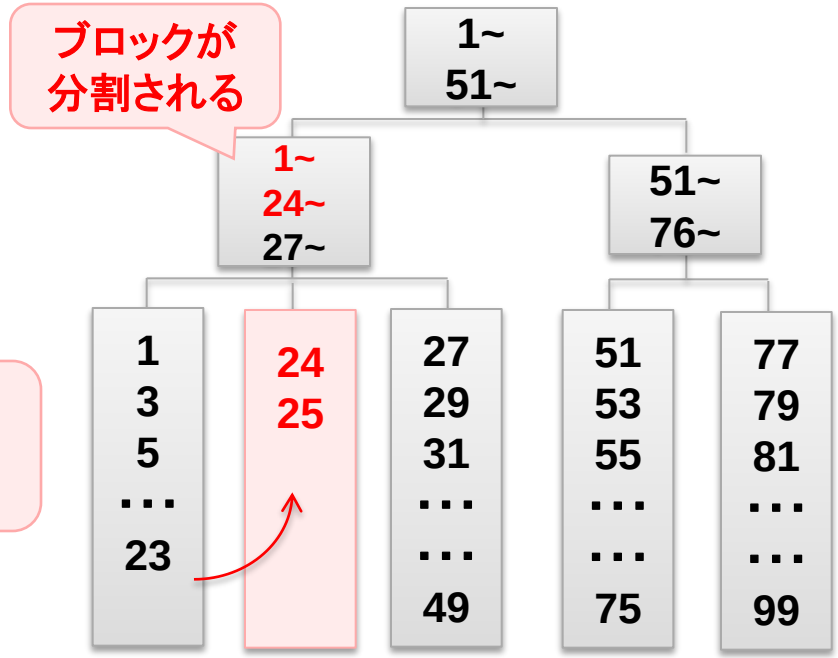
＜例＞昇順にデータを追加した場合

- INSERT 98 INSERT 99



＜例＞ランダムにデータを追加した場合

- INSERT 24



※スライドの図はイメージあり、厳密な索引構造ではありません

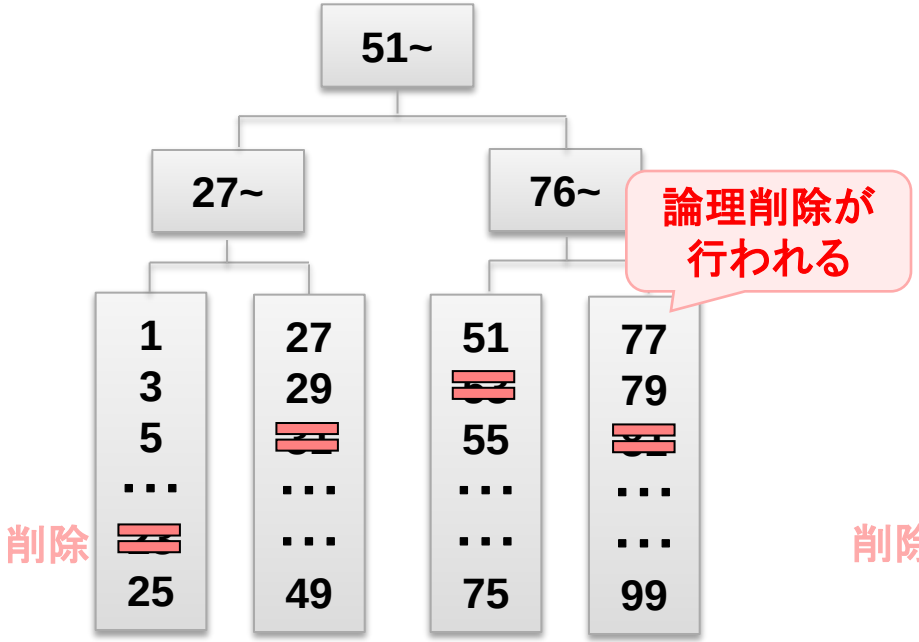
4. 索引のメンテナンスをしているか？

索引断片化の例（データを変更・削除する場合）

- データを更新・削除すると、削除済みエントリが残る
  - データを削除した場合：論理削除のみが行われる（領域は減らない）
  - データを更新した場合：論理削除とデータの追加が行われる（領域が増える）

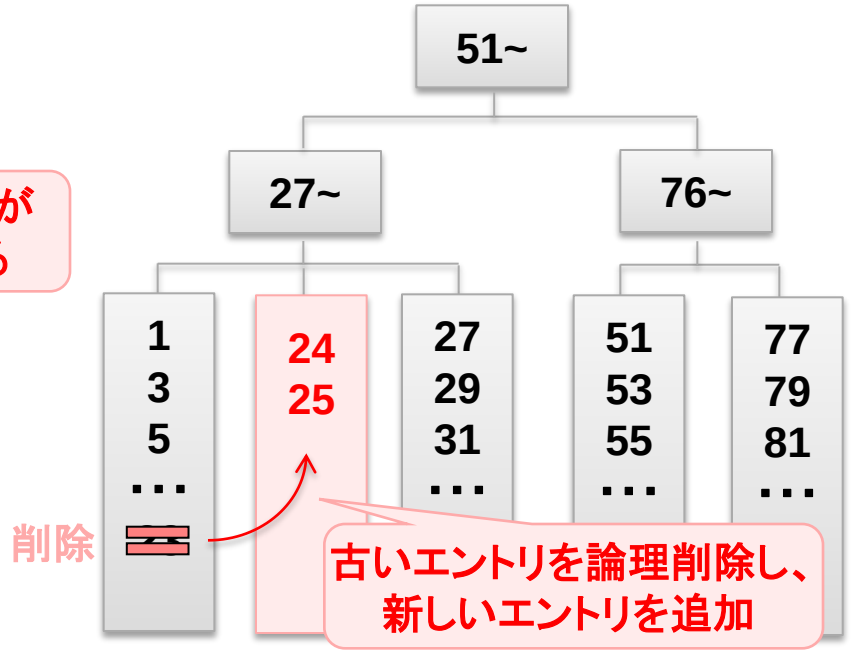
＜例＞データを削除した場合

- DELETE 23,31,53,81



＜例＞データを変更した場合

- UPDATE 23 ⇒ 24



※スライドの図はイメージあり、厳密な索引構造ではありません

4. 索引のメンテナンスをしているか？
索引構造情報の分析方法

- 索引構造の分析
  - 索引に含まれるブロック数や高さ
  - 索引に含まれている削除済みエントリ数

```
ANALYZE INDEX 索引名 VALIDATE STRUCTURE ;
```

INDEX_STATS データディクショナリに分析結果を格納

```
SQL> SELECT blocks, pct_used, lf_rows, del_lf_rows
  2    FROM index_stats;

      BLOCKS  PCT_USED LF_ROWS DEL_LF_ROWS HEIGHT
------------ -------- ------- ----------- ------
          56       54    4373         949      2
```

主な項目		
	BLOCKS	使用ブロック数
	PCT_USED	領域の使用比率(%)
	LF_ROWS	全リーフ行数（削除済み含）
	DEL_LF_ROWS	削除されたリーフの行数
	HEIGHT	Bツリーの高さ

4. 索引のメンテナンスをしているか？
索引再構築の目安

- 索引を定期的に分析し、メンテナンス
- メンテナンスの目安
  - 領域の使用比率(PCT_USED)を定期的に監視し、使用率が低下したら再構築を検討
  - 一般的に、下記の数値を超えたタイミングで再構築を検討
    - 階層の高さが4階層以上 （ HEIGHT => 4 ）
    - 削除された行エントリーの占める割合が20～30%を越える場合
      （ DEL_LF_ROW/LF_ROWS > 0.2 ）
  - 表に対する大量データ操作の後、再構築
    - 大量のデータを追加・削除した場合
      例：夜間に大量のデータをローディングした
    - 表のメンテナンスをした場合
      例：ALTER TABLE MOVEコマンドなどで表を移動した場合
  - ある程度傾向をみて定期バッチで再構築
    （データの更新量により、1回/1週間～1回/1ヶ月）

4. 索引のメンテナンスをしているか？
索引の再構築

- 索引の再構築方法
  - 索引の再作成（削除して再作成）
  - 索引のREBUILD

索引のREBUILD

```
ALTER INDEX 索引名 REBUILD;
```

- 既存の索引をもとに索引を作成し、新しい索引の作成後、古い索引を削除
- 索引のREBUILDのメリット
  - 既存の索引（ソート済みのデータ）をもとに、新しく索引を作成するため、一から索引を作成するよりも高速
  - 索引の再構築中も、古い索引を使用してアクセス（SELECT）可能
  - オンライン再構築（索引の再構築中にDML操作可能）が可能
    （注意：オンライン再構築はEnterprise Editionの機能です）
- 索引のREBILDのデメリット
  - 新しい索引を索引してから古い索引を削除するので、一時的に2倍の領域が必要

4. 索引のメンテナンスをしているか？
未使用索引の確認と削除

- 使用されていない索引を特定し、削除する
  - 使われていない索引が無駄な領域をとるのを防ぐため
  - 更新操作のボトルネックになるため
- 未使用索引の特定方法

未使用索引の監視

```
ALTER INDEX 索引名 MONITORING USAGE;
```

この間、索引が使われたかどうかを監視し、
使用状況をV$OBJECT_USAGEビューに格納

```
ALTER INDEX 索引名 NOMONITORING USAGE;
```

```
SELECT index_name,used FROM V$OBJECT_USAGE;

INDEX_NAME      USED
--------------- ---
DEPT_ID_IDX     YES
EMP_ID_IDX      NO
```

「USED」列が「NO」の索引は、
この期間使用されなかった

＜まとめ＞索引を作成したのに検索が遅い！

索引チューニング　4つのチェックポイント

1. **索引を使用する事で速くなる処理か？**
   - 表のサイズや選択率（選択行数）によっては全表走査の方が速い可能性
   - 大量の索引読み込みが行われる検索ではパーティショニングが有効
2. **SQLの構文によっては、索引があっても使用されない可能性**
   - 索引列に対してNULL条件やNOT条件が使用されている場合
   - 条件式に計算を含む場合
   - LIKE条件を使った中間一致・後方一致検索をする場合
3. **オプティマイザが適切な実行計画を選択していない可能性**
   - 実行計画を確認し、適切な実行方法を指定（ヒント）
4. **表のメンテナンスに伴う索引のメンテナンス**
   - 表データを移動した場合（MOVEコマンド等）
   - 索引の断片化に伴うメンテナンス
   - 不要な索引のメンテナンス

Agenda

- 索引チューニングのポイント
  索引がうまく使われない4つのパターン
  - 索引を使用する事で速くなる処理か？
  - 索引を利用できるSQL文か？
  - オプティマイザが索引利用を選択しているか？
  - 索引のメンテナンスをしているか？
- 様々なタイプの索引
  - ビットマップ索引
  - 複合索引
  - 逆キー索引
  - 索引構成表

ビットマップ索引

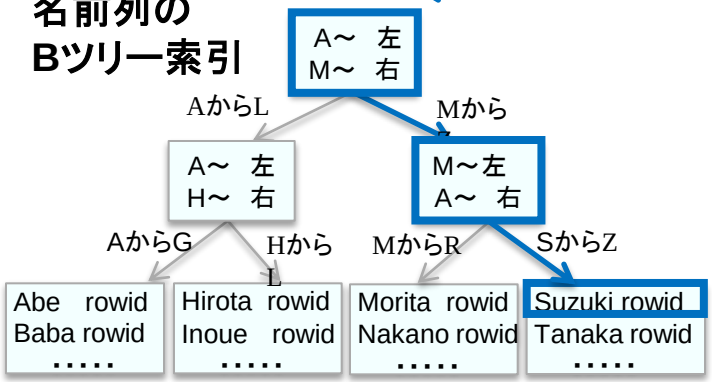
ビットマップ索引とは

- 列値と各レコードがその値に該当するか否かをビットで表した索引
- 検索処理ではビットの有無で条件に該当するか否かが判定される

社員表

	番号	名前	勤務地	性別
ROWID1	1	Tanaka	関東	男
ROWID2	2	Suzuki	関東	女
ROWID3	3	Yoshida	東北	男
ROWID4	4	Abe	関西	女
ROWID5	5	Inoue	関東	男

名前列の
Bツリー索引



勤務地列のビットマップ索引

ROWID	関東	関西	東北
ROWID1	1	0	0
ROWID2	1	0	0
ROWID3	0	0	1
ROWID4	0	1	0
ROWID5	1	0	0

性別列のビットマップ索引

ROWID	男	女
ROWID1	1	0
ROWID2	0	1
ROWID3	1	0
ROWID4	0	1
ROWID5	1	0

ビットマップ索引
ビットマップ索引の特徴と作成例

- カーディナリティの低い（値の種類が少ない）列に有効
  - 地域（東北・関東・中部・関西　など）
  - 性別（男・女）　年代（10代・20代・30代　など）
- 複数の列をWHERE条件に指定している場合、各列にビットマップ索引を作成しておくことで、ビット演算による高速な検索を行うことが可能
  - 関東に住んでいる20代男性
  - 製品Aを買った男性　など

関西に住んでいる
女性は？

ROWID	関東	関西	東北
ROWID1	1	0	0
ROWID2	1	0	0
ROWID3	0	0	1
ROWID4	0	1	0
ROWID5	1	0	0

ROWID	男	女
ROWID1	1	0
ROWID2	0	1
ROWID3	1	0
ROWID4	0	1
ROWID5	1	0

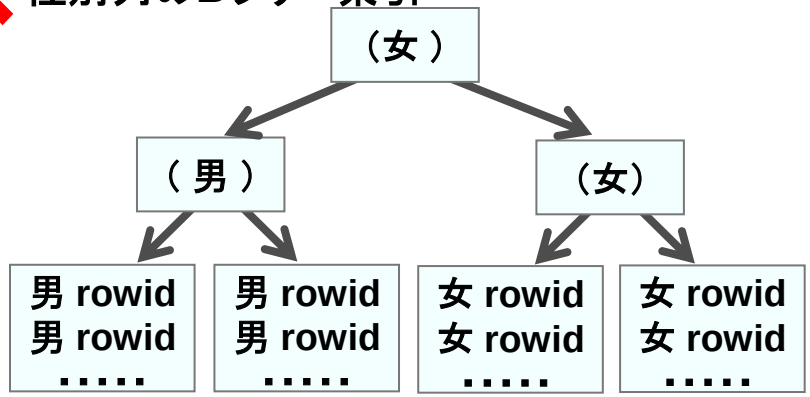
- ビットマップ索引の作成例
  - CREATE INDEX文で「BITMAP」を指定

```
CREATE BITMAP INDEX 索引名
ON 表名(列名);
```

ビットマップ索引
Bツリー索引とビットマップ索引の比較

Bツリー索引	ビットマップ索引
カーディナリティの高い （種類の多い）列に適している	カーディナリティの低い （種類の少ない）列に適している
ORを使用した検索には それほど適していない	OR条件などのビット演算による 高速な検索を行うことが可能
更新のオーバーヘッドが大きい	更新のオーバーヘッドが非常に大きい
OLTP向き	DWH向き
索引のサイズ比較的大きい	索引のサイズ比較的小さい

✕ 性別列のBツリー索引



✕ 名前列のビットマップ索引

ROWID	Tanaka	Suzuki	Yoshida	・・・
ROWID1	1	0	0	0
ROWID2	0	1	0	0
ROWID3	0	0	1	0
ROWID4	0	0	0	1
・・・	0	0	0	0

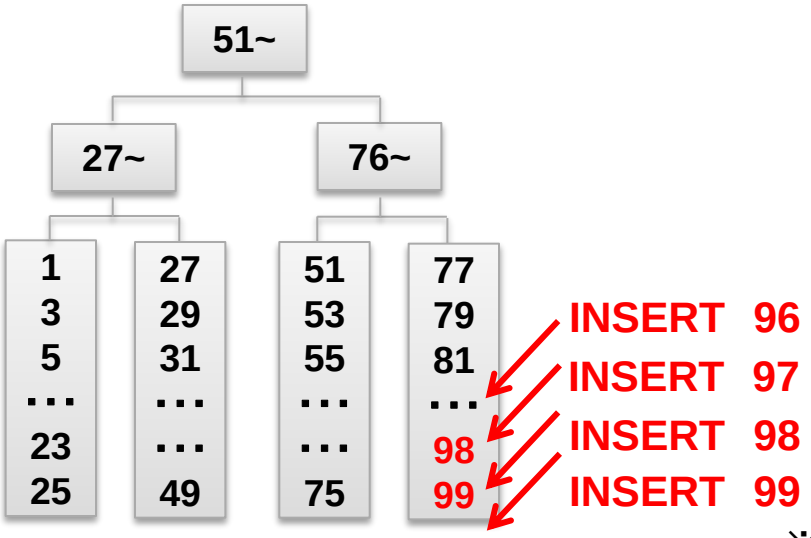
逆キー索引

逆キー索引とは

- 索引列のデータをビット単位で反転させ、反転させたデータをソートして索引に格納する索引
- 索引のキー値を逆にすることにより、挿入値を索引のリーフ・キー全体に分散させることができる

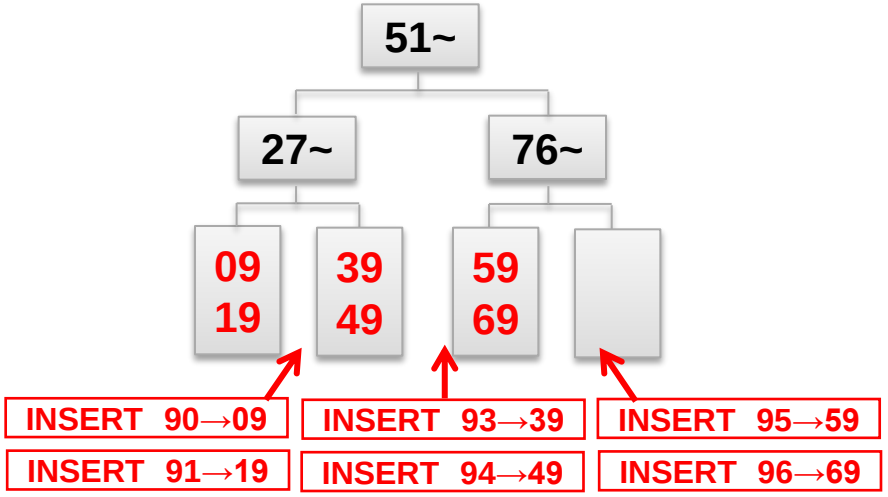
＜通常の索引＞

- 連続した値がINSERTされる環境では特定のブロックにアクセスが集中



＜逆キー索引＞

- 値を逆にすることによりアクセスを分散させることが可能



※スライドの図はイメージあり、厳密な索引構造ではありません

逆キー索引

逆キー索引の特徴と作成例

- 逆キー索引のメリット
  - 索引のキー値を逆にすることにより、挿入値を索引のリーフ・キー全体に分散させることができるため、特定の索引ブロックにアクセスが集中することを防ぐことが可能
- 逆キー索引のデメリット
  - 範囲検索（ <、>、betweenなど）に索引を使うことができない

- ビットマップ索引の作成例
  - CREATE INDEX文で「REVERSE」を指定

```
CREATE INDEX 索引名
ON 表名(列名) REVERSE;
```

複合索引(コンポジット索引)

複合索引とは

- 複数の列を指定した索引

以下のような製品表の索引を検討

```
CREATE TABLE 製品表
( 親コード number(30),
  分類    varchar2(30),
  種別    varchar2(30),
  製品名  number(10));
```

以下のような条件で検索を検討

- 親コードが1で、分類がAで、種別が「う」の製品を検索
- 親コードが3で、分類がCの製品を検索

親コード	分類	種別	製品名
1	A	あ	XXX
3	B	う	XXX
…	…	…	…
2	A	い	XXX
3	C	あ	XXX
…	…	…	…
3	C	か	XXX
2	A	さ	XXX
…	…	…	…
1	B	い	XXX
2	B	え	XXX
…	…	…	…

複合索引(コンポジット索引)

単一列索引と複合索引

- 単一列索引と複合索引の作成例

単一列索引
それぞれの列に索引を作成

```
CREATE INDEX 製品_親_idx
ON 製品表(親コード);

CREATE INDEX 製品_分類_idx
ON 製品表(分類);

CREATE INDEX 製品_種別_idx
ON 製品表(種別);
```

複合索引
複数列にまとめて一つの索引を作成

```
CREATE INDEX 製品_idx
ON 製品表(親コード,分類,種別);
```

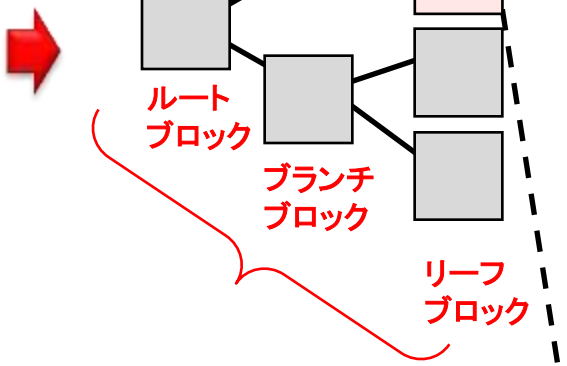
親コード	分類	種別	製品名
1	A	あ	XXX
3	B	う	XXX
…	…	…	…
2	A	い	XXX
3	C	あ	XXX
…	…	…	…
3	C	か	XXX
2	A	さ	XXX
…	…	…	…
1	B	い	XXX
2	B	え	XXX
…	…	…	…

複合索引(コンポジット索引)

複合索引の構造

- 複合索引の構造
  - 指定した索引列のデータが、指定した列の順にソートされて、リーフ・ブロックに格納される
  - 単一列索引を3つ読み、マージするよりも効率的

親コード	分類	種別	製品名
1	A	あ	XXX
3	B	う	XXX
…	…	…	…
2	A	い	XXX
3	C	あ	XXX
…	…	…	…
3	C	か	XXX
2	A	さ	XXX
…	…	…	…
1	B	い	XXX
2	B	え	XXX
…	…	…	…



各キー列値ごとに階層的にソートされて格納されている

親コード	分類	種別	
1	A	あ	ROWID
1	A	い	ROWID
1	A	う	ROWID
1	B	あ	ROWID
1	B	え	ROWID
1	C	あ	ROWID
2	A	あ	ROWID
2	A	う	ROWID
2	B	あ	ROWID
2	C	あ	ROWID
2	C	い	ROWID
3	A	い	ROWID
…	…	…	…

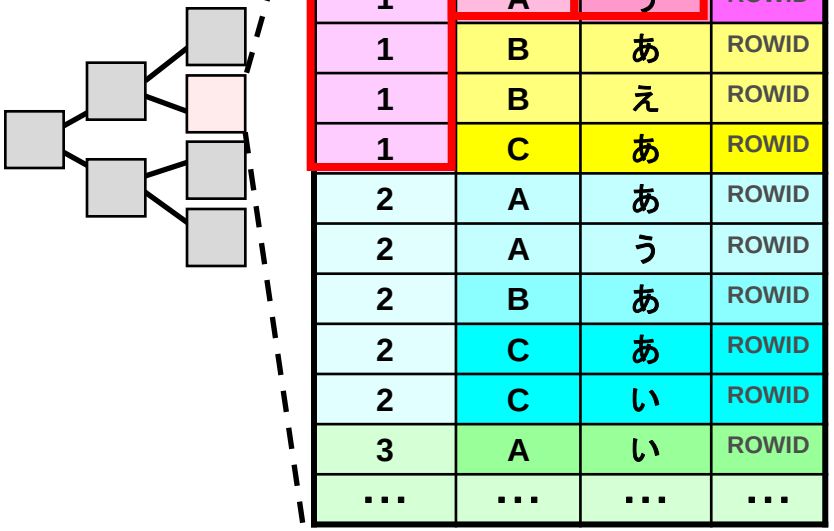
一つ目のキー列を元にB*Treeのツリー構造が構成

複合索引(コンポジット索引)

複合索引の検索イメージ

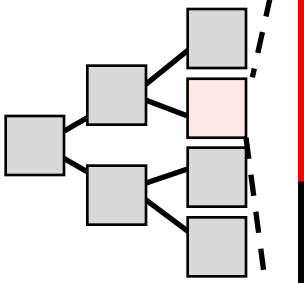
- 複合索引を使った検索例
  - 索引を使って、複数の検索条件にあった行を絞り込みが可能
  - 索引に含まれる全ての列を指定しなくても、複合索引をつかうことが可能

```
SELECT * FROM 製品表
WHERE  親コード=1
AND    分類 ='A'
AND    種別 ='う';
```



親コード	分類	種別	
1	A	あ	ROWID
1	A	い	ROWID
1	A	う	ROWID
1	B	あ	ROWID
1	B	え	ROWID
1	C	あ	ROWID
2	A	あ	ROWID
2	A	う	ROWID
2	B	あ	ROWID
2	C	あ	ROWID
2	C	い	ROWID
3	A	い	ROWID
…	…	…	…

```
SELECT * FROM 製品表
WHERE  親コード=1
AND    分類='A';
```



親コード	分類	種別	
1	A	あ	ROWID
1	A	い	ROWID
1	A	う	ROWID
1	B	あ	ROWID
1	B	え	ROWID
1	C	あ	ROWID
2	A	あ	ROWID
2	A	う	ROWID
2	B	あ	ROWID
2	C	あ	ROWID
2	C	い	ROWID
3	A	い	ROWID
…	…	…	…

複合索引(コンポジット索引)

複合索引を利用できる条件

- 複合索引の対応条件とパフォーマンス

```
SELECT * FROM 製品表
WHERE  親コード=?
AND    分類    =?
AND    種別    =?;
```

WHRER句の条件にどの列を使うかによって、索引を使用できるかどうか、および、パフォーマンスが変わる

親コード	分類	種別
○	○	○
○	○	×
○	×	○
○	×	×
×	○	○
×	○	×
×	×	○
×	×	×

○:WHERE句に条件あり
×:WHERE句に条件なし

・・・◎ 複合索引の使用がパフォーマンス最適

・・・○ 複合索引を使用可能
単一列索引より複合索引のほうが早いケースが多い
(列数や各列のサイズなどによって変わる)

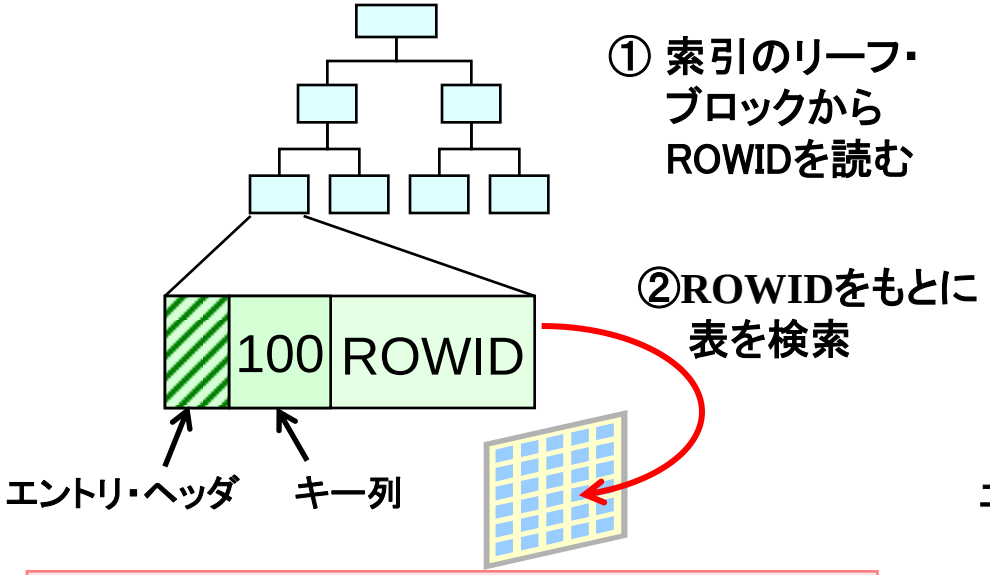
・・・△ 9i以降複合索引を使用可能
複合索引より単一列索引のほうが早いケースが多い
(パフォーマンスが最適かどうかはデータ構造などによる)

・・・× 複合索引を使用不可

索引構成表
索引構成表とは

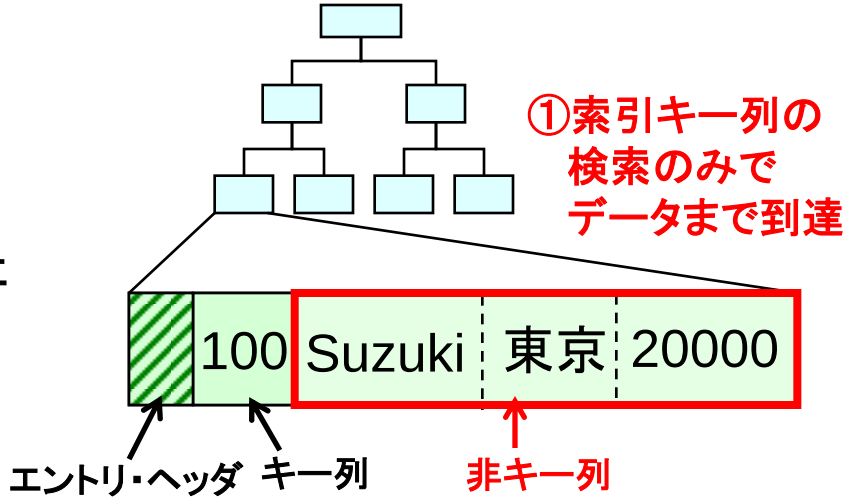
- データ全体をB*Tree索引に格納している索引
  - 索引のキー列だけでなく、その他の列（非キー列）もリーフブロックに格納している索引
  - 索引にアクセスするだけでデータが取得できるため、より高速な検索が可能

＜通常の索引を使用した検索＞



2段階の処理が必要で余分なI/Oが発生

＜索引構成表を使用した検索＞

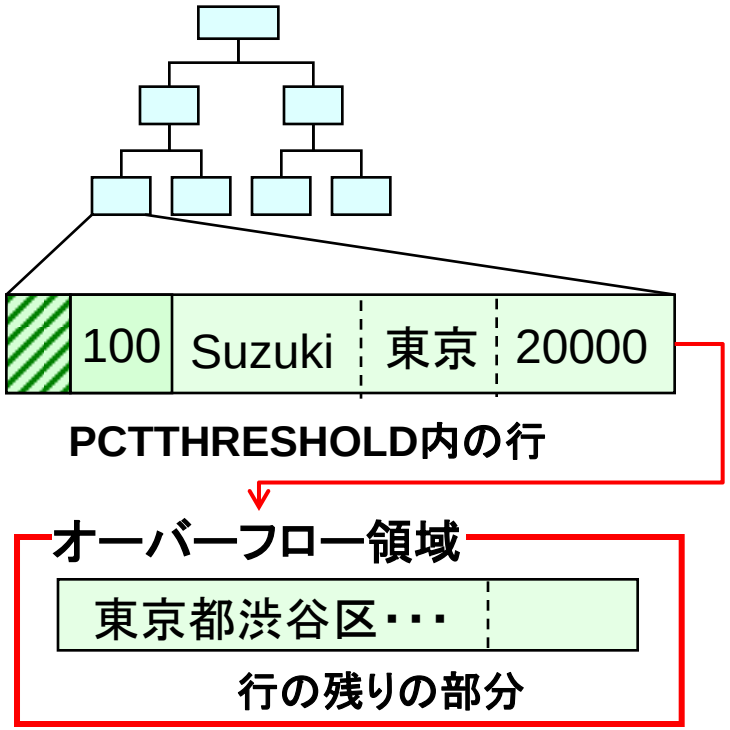


検索時のI/Oを減少させるので高速

索引構成表
索引構成表の作成例

- 索引構成表の作成例
  - CREATE TABLEで「ORGANIZATION INDEX」を指定
  - しきい値（PCTTHRESHOLD）を設定し、一部のデータを別の領域に格納することもできる

```
CREATE TABLE countries
( emp_id    NUMBER(2) PRIMARY KEY,
  name      VARCHAR2(40),
  ....
  address   VARCHAR2(25),
  ....                        )
ORGANIZATION INDEX TABLESPACE indx
PCTTHRESHOLD 20
OVERFLOW TABLESPACE data;
```



＜まとめ＞様々なタイプの索引

表の特性、検索条件に合わせて最適な索引を選択

- ビットマップ索引
  - 列値と各レコードがその値に該当するか否かをビットで表した索引
  - カーディナリティの低い列で、ビット演算による高速な検索を行うことが可能
- 複合索引
  - 索引列のデータをビット単位で反転させ、反転させたデータをソートして索引に格納する索引
  - 挿入値を索引のリーフ・キー全体に分散させ、特定の索引ブロックにアクセスが集中することを防ぐことが可能
- 逆キー索引
  - 複数の列を指定した索引
  - 索引を使って、複数の検索条件にあう行を絞り込みが可能
- 索引構成表
  - 索引のキー列だけでなく、その他の列（非キー列）もリーフブロックに格納している索引
  - 索引にアクセスするだけでデータが取得できるため、より高速な検索が可能

まとめ

- 索引チューニングのポイント
  索引がうまく使われない4つのパターン
  - 索引を使用する事で速くなる処理か？
  - 索引を利用できるSQL文か？
  - オプティマイザが索引利用を選択していないのではないか？
  - 索引のメンテナンスをしているか？
- 様々なタイプの索引
  - ビットマップ索引
  - 複合索引
  - 逆キー索引
  - 索引構成表

あなたにいちばん近いオラクル

Oracle Direct

まずはお問合せください

Oracle Direct　検索

システムの検討・構築から運用まで、ITプロジェクト全般の相談窓口としてご支援いたします。

システム構成やライセンス/購入方法などお気軽にお問い合わせ下さい。

Web問い合わせフォーム

専用お問い合わせフォームにてご相談内容を承ります。
http://www.oracle.co.jp/inq_pl/INQUIRY/quest?rid=28

※フォームの入力には、Oracle Direct Seminar申込時と同じ
　ログインが必要となります。
※こちらから詳細確認のお電話を差し上げる場合がありますので、ご登録されている連絡先が最新のものになっているか、ご確認下さい。

フリーダイヤル

0120－155－096

※月曜~金曜　9:00~12:00、13:00~18:00
（祝日および年末年始除く）

ORACLE

以上の事項は、弊社の一般的な製品の方向性に関する概要を説明するものです。また、情報提供を唯一の目的とするものであり、いかなる契約にも組み込むことはできません。以下の事項は、マテリアルやコード、機能を提供することをコミットメント（確約）するものではないため、購買決定を行う際の判断材料になさらないで下さい。オラクル製品に関して記載されている機能の開発、リリースおよび時期については、弊社の裁量により決定されます。