Oracle Direct Seminar

ORACLE®

実践!!パフォーマンス・チューニング
～SQLチューニング編～

日本オラクル株式会社

Oracle Direct

Oracle Direct Seminar

アジェンダ

- 本セミナーの目的
- SQLチューニングの流れとチューニング例

無償技術サービスOracle Direct Concierge

・Oracle Databaseパフォーマンス・クリニック
・ Webシステム　ボトルネック診断サービス
・Oracle 構成相談
・Oracle Database バージョンアップ支援
・WebLogic Server バージョンアップ支援
・Oracle Developer/2000 Webアップグレード相談
・SQL Serverからの移行アセスメント
・MySQLからの移行相談
・PostgreSQLからの移行相談
・Accessからの移行アセスメント
・システム連携アセスメント
・システムセキュリティ診断
・簡易業務診断
・メインフレーム資産活用

http://www.oracle.com/lang/jp/direct/services.html

SQLチューニングの必要性
本セミナーの目的

- SQLチューニングのスキルが求められている背景

多くの環境で起こる問題
- データベースのパフォーマンス問題の多くが SQLパフォーマンス問題に帰着する
- 何をどのように調査、解決すればよいのかわからない

SQL性能問題が解決しにくい理由
- 同一の結果に対して、複数の記述方法、処理方法が可能であり最適な方法は環境によって異なる
- 処理方法はデータベースに任されている
- 「必ず早くなる」という正解がない

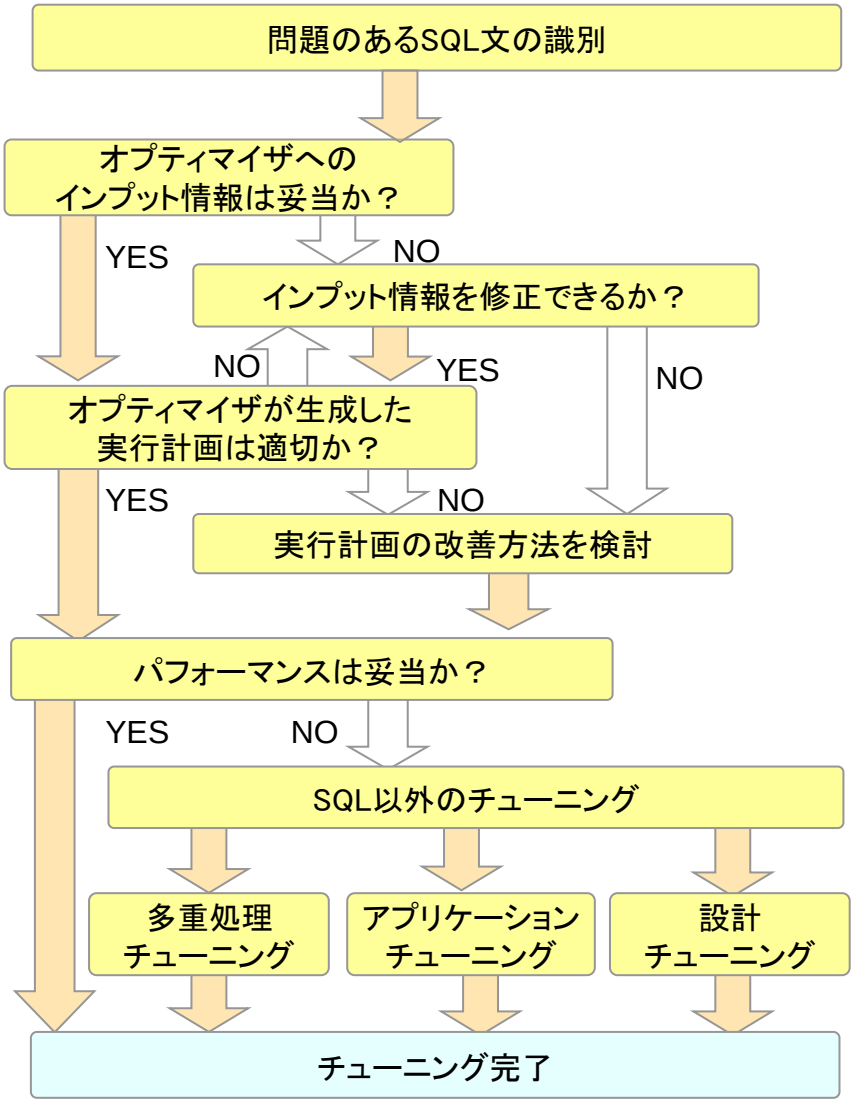
本セミナーの目的

Oracle Database内部でのSQL処理の流れから
チューニングに最低限必要なチェック項目と対策を考える

SQLチューニングの流れ

- SQLチューニングの流れ
  - 問題のあるSQL文を識別する
  - 前提条件を確認する
    - オプティマイザ統計は適切か
  - 実行計画を読み解く
  - 読み解いた実行計画から、コストの高い処理の改善方法を検討する
    - 結合順序、方法は変更できるか
    - 効率的な索引を作成できるか
    - SQLの構文を変更できるか
    - ヒント句を利用できるか

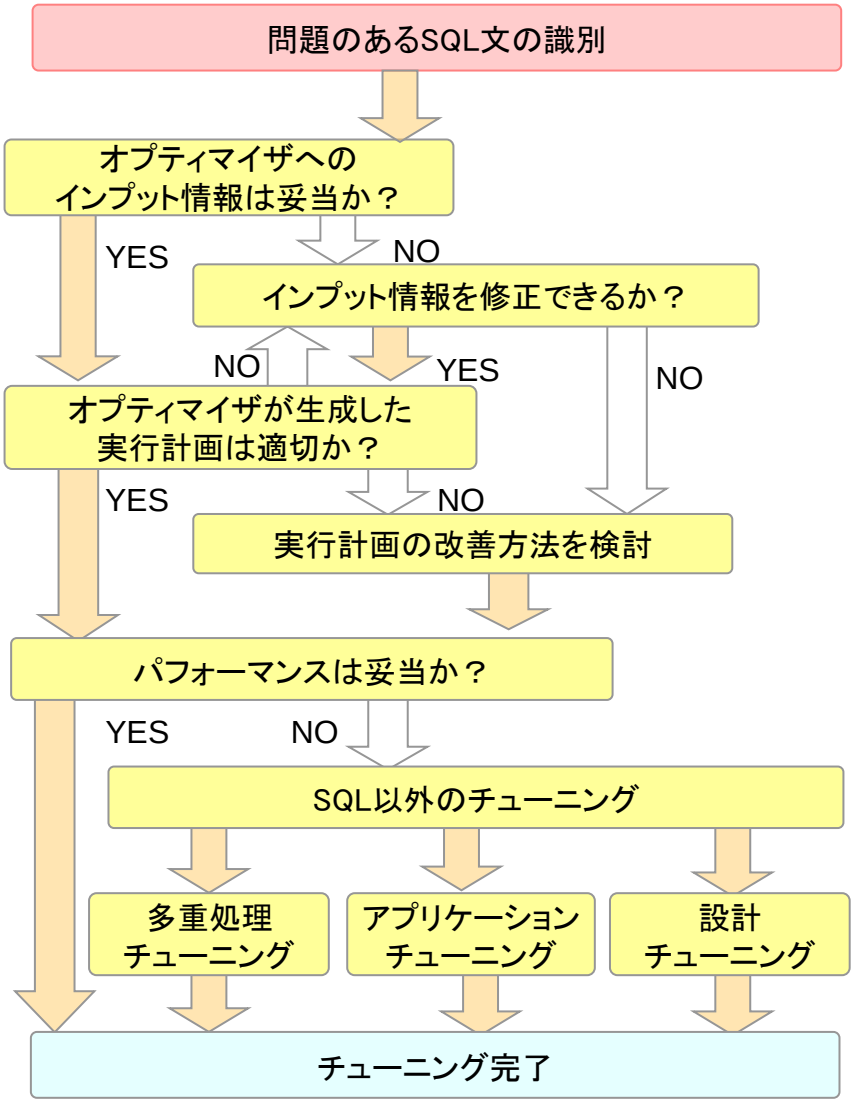
- 解決できない場合には、SQLチューニング以外の方法を検討する



SQLチューニングの流れ
問題のあるSQL文を識別する

- SQLチューニングの流れ
  - 問題のあるSQL文を識別する
  - 前提条件を確認する
    - オプティマイザ統計は適切か
  - 実行計画を読み解く
  - 読み解いた実行計画から、コストの高い処理の改善方法を検討する
    - 結合順序、方法は変更できるか
    - 効率的な索引を作成できるか
    - SQLの構文を変更できるか
    - ヒント句を利用できるか

- 解決できない場合には、SQLチューニング以外の方法を検討する



パフォーマンス問題のあるSQL文の識別
問題を識別する手法

パフォーマンス問題のあるSQL文とは
- 1実行当たりの実行時間が長いSQL
- 使用頻度が高く、実行中に大量のシステム・リソースを使用するSQL

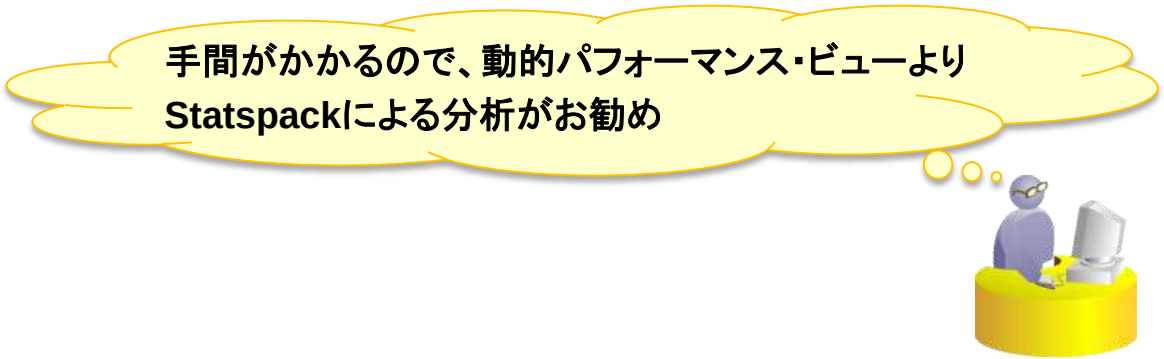
- パフォーマンス問題のあるSQL文の識別手法
  - 従来の手法
    - 動的パフォーマンス・ビュー
    - Statspack
  - Oracle Database 10g以降（Enterprise Edition + Diagnostics Pack）
    - Oracle Enterprise Manager の 上位SQL
    - Automatic Database Diagnostic Monitor（ADDM）レポート

Enterprise Managerを使ったSQL文の監視とチューニングは、【補足】で紹介します。

パフォーマンス問題のあるSQL文の識別
動的パフォーマンス・ビュー

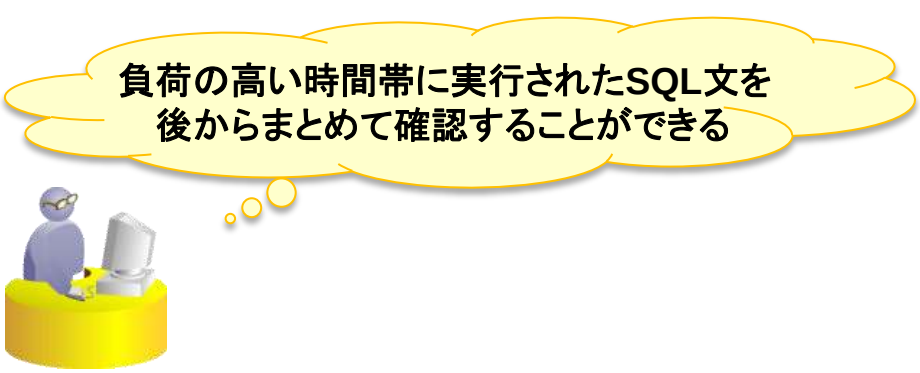
- 動的パフォーマンス・ビューを使った問題のあるSQL文の識別方法
  - V$SYSSTATビューからのシステム統計
    - インスタンスが起動以降のシステム統計情報の累積値
      （アクセスしたデータブロックの累積値など）
  - V$SQLAREAビューからのSQL統計
    - 共有プールに存在するすべてのSQL文のリソース使用情報

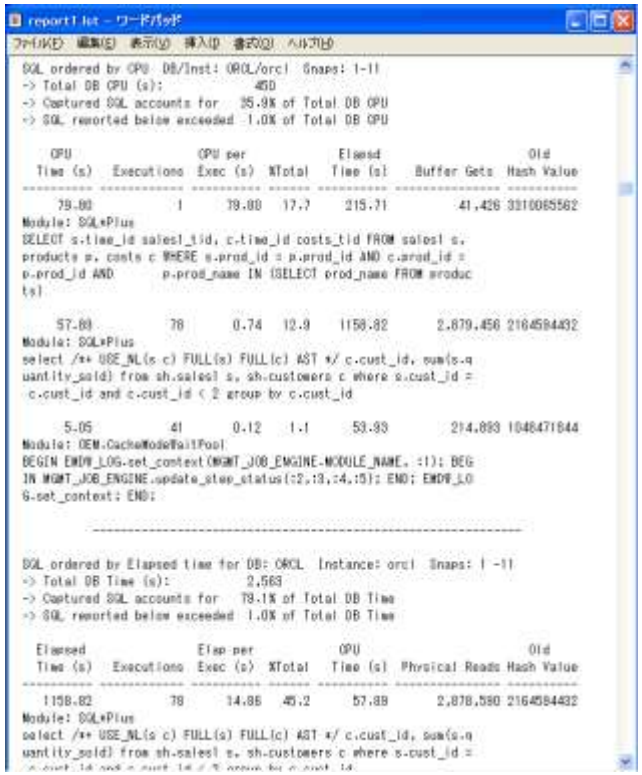
```
SELECT sql_text,disk_reads,sorts,cpu_time,elapsed_time
FROM   v$sqlarea
WHERE  upper(sql_text) like '%ORDERS%'
ORDER BY sql_text;
```



パフォーマンス問題のあるSQL文の識別
Statspack

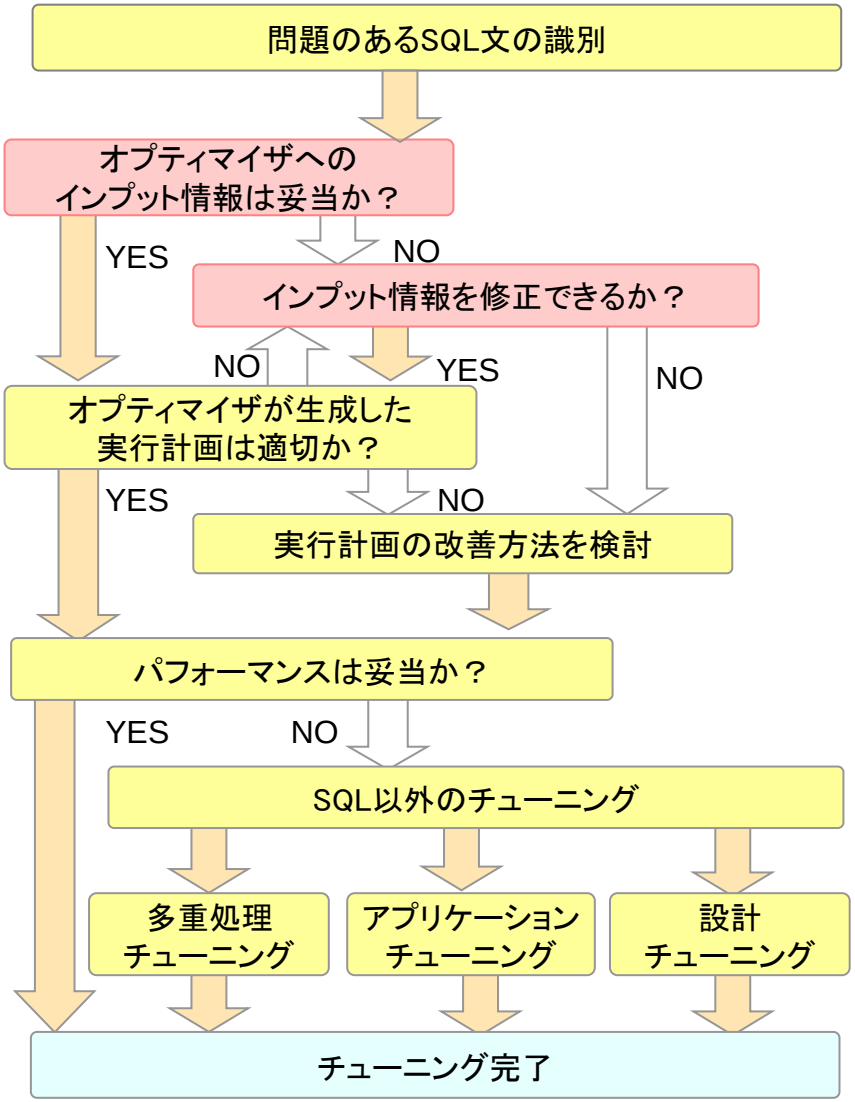
- Statspackを使った問題のあるSQL文の識別方法
  - パフォーマンス統計情報を収集するパッケージ
  - 2時点間の統計情報の差分をとることで、その時間帯におけるパフォーマンス解析を行うユーティリティ
  - データベースのパフォーマンス統計と「上位SQL」を確認
    - SQL ordered by CPU
    - SQL ordered by Elapsed time
    - SQL ordered by Gets
    - SQL ordered by Reads





SQLチューニングの流れ
前提を確認する

- SQLチューニングの流れ
  - 問題のあるSQL文を識別する
  - 前提条件を確認する
    - オプティマイザ統計は適切か
  - 実行計画を読み解く
  - 読み解いた実行計画から、コストの高い処理の改善方法を検討する
    - 結合順序、方法は変更できるか
    - 効率的な索引を作成できるか
    - SQLの構文を変更できるか
    - ヒント句を利用できるか
- 解決できない場合には、SQLチューニング以外の方法を検討する

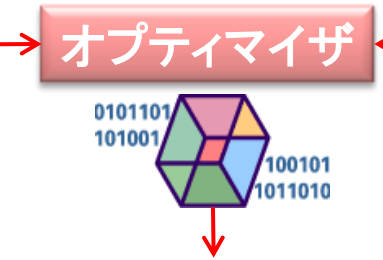


コストベース・オプティマイザと実行計画

コストベース・オプティマイザとは

- SQL文の実行とコストベース・オプティマイザ（CBO）
  - オプティマイザは、ユーザーが指定したSQL文に対して最も効率的な（最もコストの低い）アクセスパスを選択
  - コストの見積もりには、オプティマイザ統計情報や初期化パラメータを利用
  - コスト：DISK I/O、CPU使用量、メモリー使用量から算出される使用リソース

```
SELECT empno FROM emp
WHERE  ename = 'Tanaka'
AND    sal > 2000;
```



オプティマイザ統計情報
- 表統計（行数、ブロック長、平均行長）
- 列統計（列内のデータ種類数、列内のNULL数）
- 索引統計（リーフブロック数、ツリーの高さ）
- システム統計（I/O、CPUパフォーマンス）

初期化パラメータ
- DB_FILE_MULTIBLOCK_READ_COUNT
- OPTIMIZER_MODE

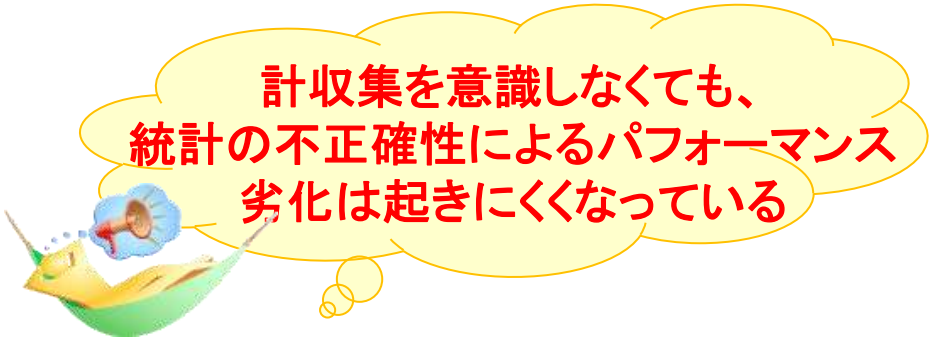
```
-------------------------------------------------------------------------------------------
| Id  | Operation                    | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             |     1 |    16 |     2   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES    |     1 |    16 |     2   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | EMP_NAME_IX |     1 |       |     1   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------
```

コストベース・オプティマイザと実行計画
オプティマイザ統計管理の重要性

コストベース・オプティマイザは統計情報に基づいて実行計画を決定

- 正確な情報が収集されていれば、最適な実行計画を選択できる
- 正確な統計が収集されていなければ、最適な実行計画を選択できない

- Oracle Database 10g 以降、デフォルトで夜間に統計が取得されるようにスケジューリングされている
- 必要に応じてヒストグラム等の追加統計情報も取得される



コストベース・オプティマイザと実行計画
【補足】統計情報取得方法の進化

- Oracle 9i Release1 までの統計収集
  - 管理者が手動で統計情報を収集
    - DBMS_STATSパッケージ
    - ANALYZEコマンド
- Oracle 9i Release2 での統計収集
  - 動的サンプリングによる統計収集
    - 統計情報が存在しない場合、SQLの実行時（ハードパース時）に動的に統計情報をサンプリングし、その結果を元に実行計画を生成
- Oracle Database 10g以降の統計収集
  - 事前定義スケジュール「GATHER_STATS_JOB」により統計を自動で収集
  - 以下のようなオブジェクトに対して、事前定義スケジュールに従って統計情報を収集
    - 統計情報をまだ収集していないオブジェクトやデータ
    - 前回の統計取得から10%以上更新されたオブジェクト

オプティマイザ統計の詳細は
「実践！！SQLチューニング
Oracle Databaseオプティマイザ120%活用術」
で説明しています

オプティマイザ統計の取得方法
DBMS_STATSによる手動統計取得

以下のような場合には統計を収集することが望ましい
- 自動統計収集が行われる時間帯から外れたタイミングで、表や索引のメンテナンスが行われる場合
- 大量データのINSERT/UPDATE/DELETEが行なわれた場合
- 統計収集の最適な頻度や対象オブジェクトを特定し、オブジェクトレベルできめ細かな管理をしたい場合

- 統計収集の方法
  - DBMS_STATSパッケージ
    - 表ごと、スキーマごと、データベース全体の統計情報を収集

```
EXECUTE DBMS_STATS.GATHER_TABLE_STATS('スキーマ','表名');
```

```
EXECUTE DBMS_STATS.GATHER_SCHEMA_STATS('スキーマ');
```

```
EXECUTE DBMS_STATS.GATHER_DATABASE_STATS();
```

オプティマイザ統計の取得方法
【補足】手動による詳細な統計情報の取得例

統計収集の詳細はマニュアルをご確認ください
「Oracle Databaseパフォーマンス・チューニング・ガイド」
オプティマイザ統計の管理

- 以下のように統計を取得することにより、オプティマイザが条件の選択性をより正確に計算することが可能になる
  - ヒストグラム：偏りのある列のデータ分布状況を取得

```
EXECUTE DBMS_STATS.GATHER_TABLE_STATS('スキーマ名','表名',
        METHOD_OPT =>'FOR ALL COLUMNS SIZE AUTO');
```

  - 複数列統計：列同士に相関関係がある場合、グループ化して統計を取得

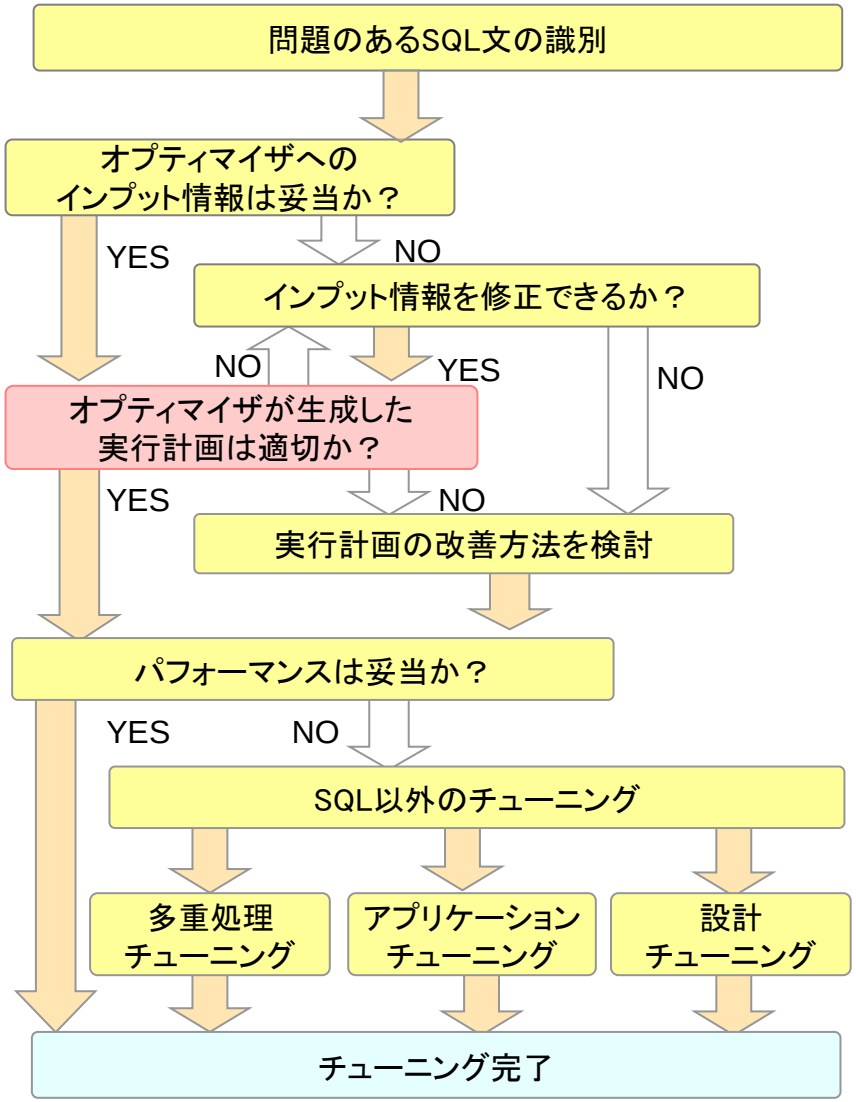
```
EXECUTE DBMS_STATS.GATHER_TABLE_STATS('スキーマ名','表名',
        METHOD_OPT =>'FOR ALL COLUMNS SIZE AUTO
        FOR COLUMNS (列1,列2) SIZE AUTO ');
```

  - 式統計：条件列に関数を使用している場合、式の結果に関する統計を取得

```
EXECUTE DBMS_STATS.GATHER_TABLE_STATS('スキーマ名','表名',
        METHOD_OPT =>'FOR ALL COLUMNS SIZE AUTO
        FOR COLUMNS (関数(列)) SIZE AUTO ');
```

SQLチューニングの流れ
実行計画を読み解く

- SQLチューニングの流れ
  - 問題のあるSQL文を識別する
  - 前提条件を確認する
    - オプティマイザ統計は適切か
  - 実行計画を読み解く
  - 読み解いた実行計画から、コストの高い処理の改善方法を検討する
    - 結合順序、方法は変更できるか
    - 効率的な索引を作成できるか
    - SQLの構文を変更できるか
    - ヒント句を利用できるか
- 解決できない場合には、SQLチューニング以外の方法を検討する



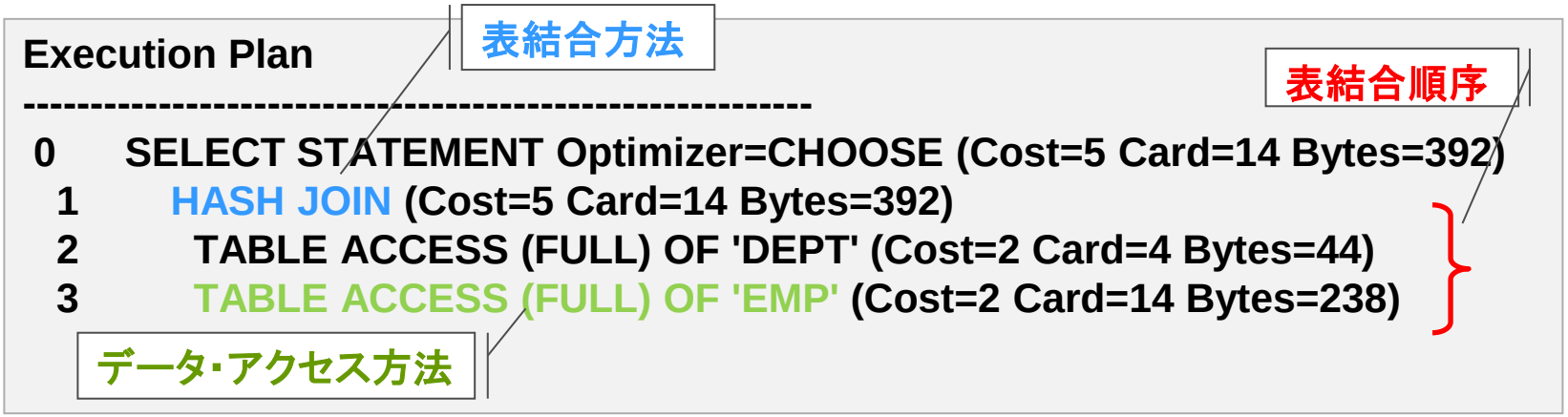
実行計画の読み方
実行計画とは

- 実行計画とは
  - Oracle DatabaseがそのSQL文を実行するために行う一連の処理
  - 表からどのようにデータを取り出し、どういう順番で結合し、どういう結合方法を選択するかといった手順
- 実行計画の読み方
  - インデントで整形されたツリー構造になっている
  - ツリー構造の深いオペレーションから実行される
  - 同一のレベルであれば、上に表示されているものから
  - 結合方法の次のレベルに結合対象の表が表示される

```
SQL> SELECT d.dname , e.empno , e.ename , e.job
 2 > FROM   emp e , dept d
 3 > WHERE  e.deptno = d.deptno;

順序    Execution Plan
---- ----------------------------------------------------------
  4      0       SELECT STATEMENT Optimizer=CHOOSE (Cost=5 Card=14 Bytes=392)
  3      1         HASH JOIN (Cost=5 Card=14 Bytes=392)
  1      2       ➡ TABLE ACCESS (FULL) OF 'DEPT' (Cost=2 Card=4 Bytes=44)
  2      3           TABLE ACCESS (FULL) OF 'EMP' (Cost=2 Card=14 Bytes=238)
```

実行計画の読み方
実行計画を読む3つのポイント



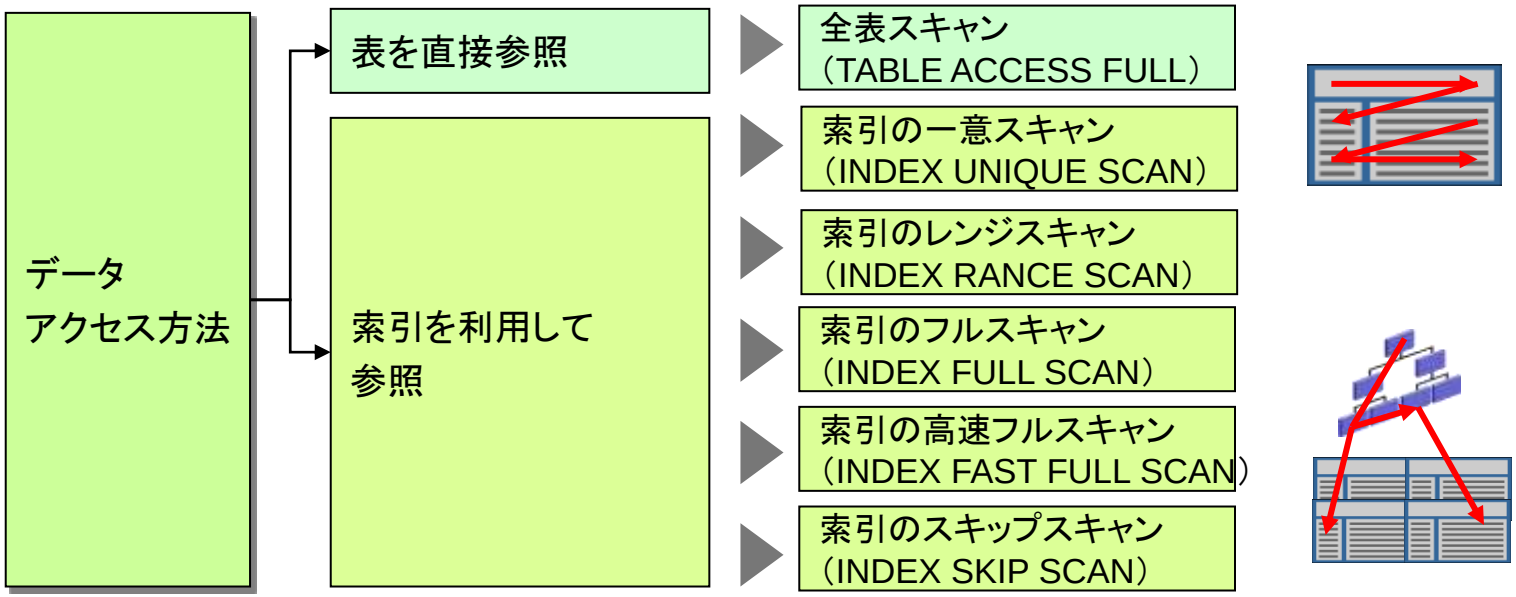
データ・アクセス方法 ▶ FULL SCANかINDEX SCANか
INDEX SCANの場合、索引をどのようにスキャンするか

表結合方法 ▶ どの結合方法が最適か

表結合順序 ▶ どのような順序で結合するか

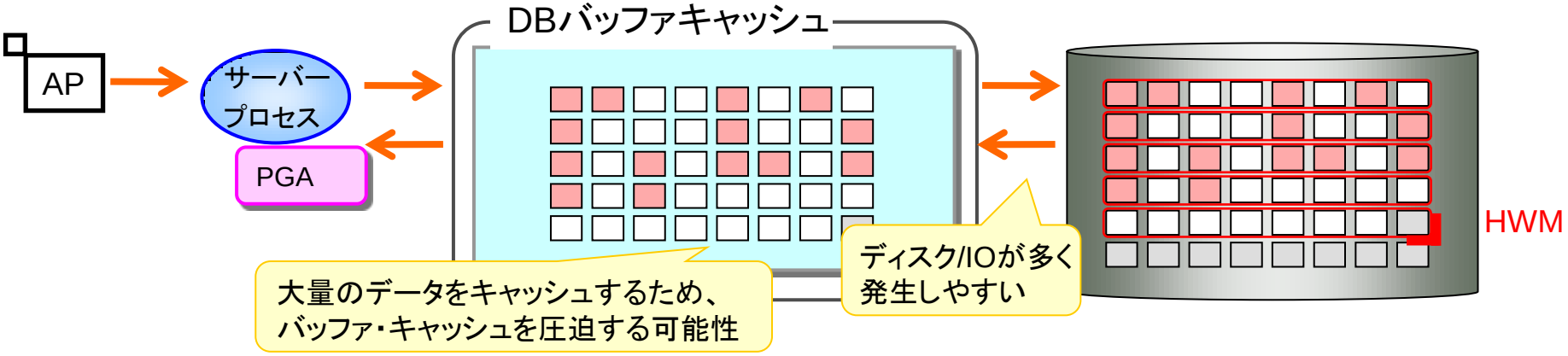
実行計画を読み解くポイント1
データ・アクセス方法

- データ・アクセス方法の種類
  - 表を直接参照（全表スキャン）
    - 索引を使わずに、表のすべてのデータにアクセスする
  - 索引を利用して参照（索引スキャン）
    - 索引を利用して特定したデータにのみアクセスする
    - いくつかの索引スキャン方法が素材する



データアクセス方法
全表スキャン

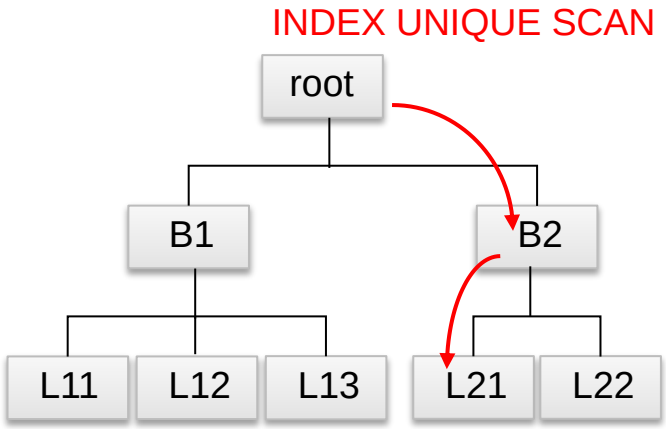
- 全表スキャン（TABLE ACCESS FULL）
  - HWM(High Water Mark)までの全ブロックを読み込む
    HWMとは：過去にデータが格納されたことのある一番高い位置を示す指標



全表スキャンが選択されていたら・・・
- 大規模な表に対する全表スキャンの実行回数が多いほど、データの取得にかかる時間が長くなる傾向があるため、全表スキャンが適切かどうかを確認
- 必ずしも全表走査が非効率というわけではなく、小さい表の場合や、多数の行を検索する場合には有効である場合もある
  - 1回の読み込みで複数ブロックを読むことができるため、小さいコールを何度も実行する場合のコストよりも低くなりやすい

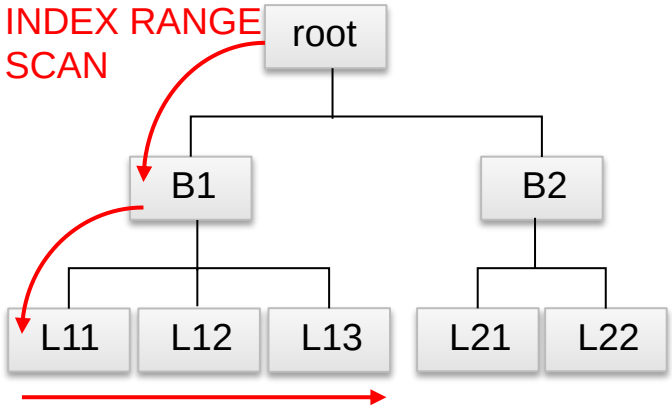
データアクセス方法
索引一意スキャン

- 索引一意スキャン
  - 単一のROWIDを戻す検索
  - UNIQUE制約またはPRIMARY KEY制約の指定された列に対して、等価条件を使用している場合に選択される



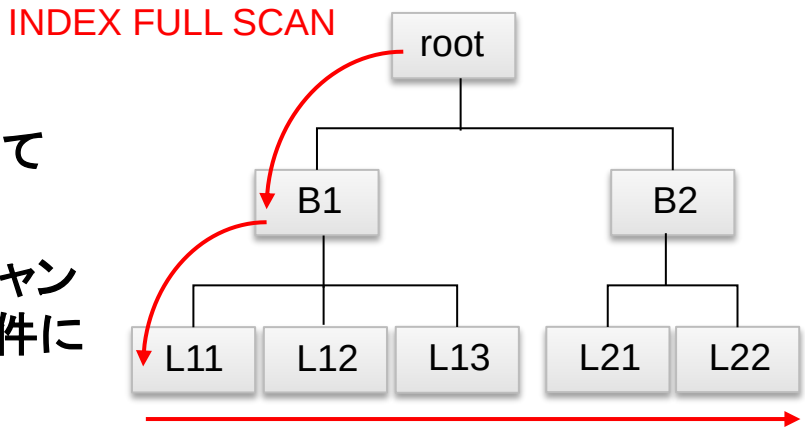
データアクセス方法
索引レンジスキャン

- 索引レンジスキャン
  - 選択したデータにアクセスするための一般的な処理
  - キー値の範囲でリーフ・ブロックをスキャン(シングルブロック・アクセス)して条件に該当する複数エントリを返す
  - データは、索引列の昇順で戻される



索引フルスキャンが選択されていたら・・・
- シングルブロック・アクセスであるため、検索量が多いと、全表スキャンのほうが効率が良い可能性もある
- リンク順にスキャンするため、キー値でソートされた順にエントリを返すことができ、その後のソート処理を省略できる場合がある

データアクセス方法
索引フルスキャン

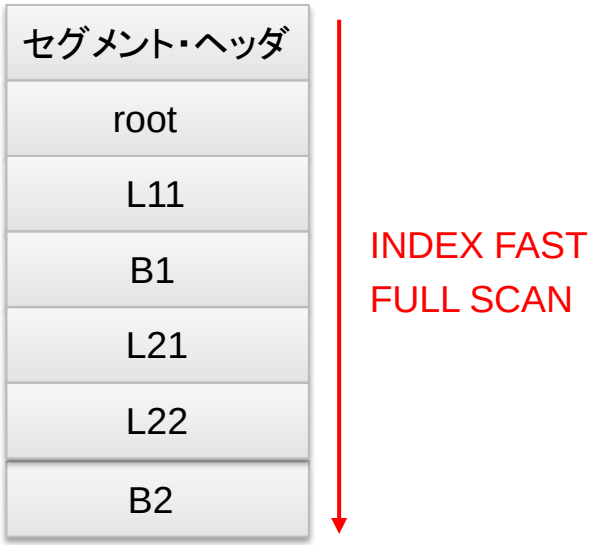


- 索引フルスキャン
  - 条件が索引のいずれかの列を参照している場合に使用される
  - 全てのリーフ・ブロックを1つずつスキャン(シングルブロック・アクセス)して、条件に該当するエントリを返す
  - データは、索引列の昇順で戻される
  - 参照列がすべて索引列である場合、表へのアクセスが排除され、高速化される可能性がある

索引フルスキャンが選択されていたら・・・
- シングルブロック・アクセスであるため、検索量が多いと、全表スキャンのほうが効率が良い可能性もある
- リンク順にスキャンするため、キー値でソートされた順にエントリを返すことができ、その後のソート処理を省略できる場合がある

データアクセス方法
索引高速フルスキャン

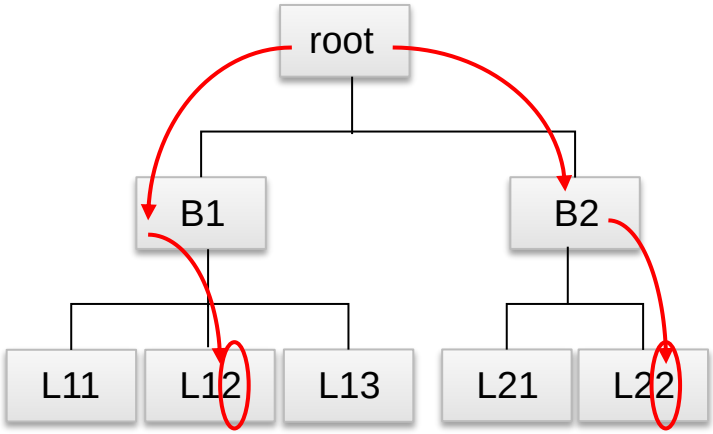
- 索引高速フルスキャン
  - 問合せに必要なすべての列が索引に含まれており、索引キーの1つ以上の列にNOT NULL制約が指定されている場合に全表スキャンのかわりに使用される
  - ツリー構造を意識せず、セグメント・ヘッダから順にブロックをフルスキャンする
  - マルチ・ブロック・リードやパラレル実行を行うことができる



索引高速フルスキャンが選択されていたら・・・
- 全表スキャンのようにマルチブロックI/Oが使用でき、パラレル化できるため、通常、索引フルスキャンよりも高速に検索を行うことができる
- データは索引キーによって並び替えられないため、ソート操作は省略できない

データアクセス方法
索引スキップスキャン

- 索引スキップスキャン
  - 複合索引の第1列目に対する条件指定が無く、2列目以降の列に対して条件指定があった場合に使用される可能性がある

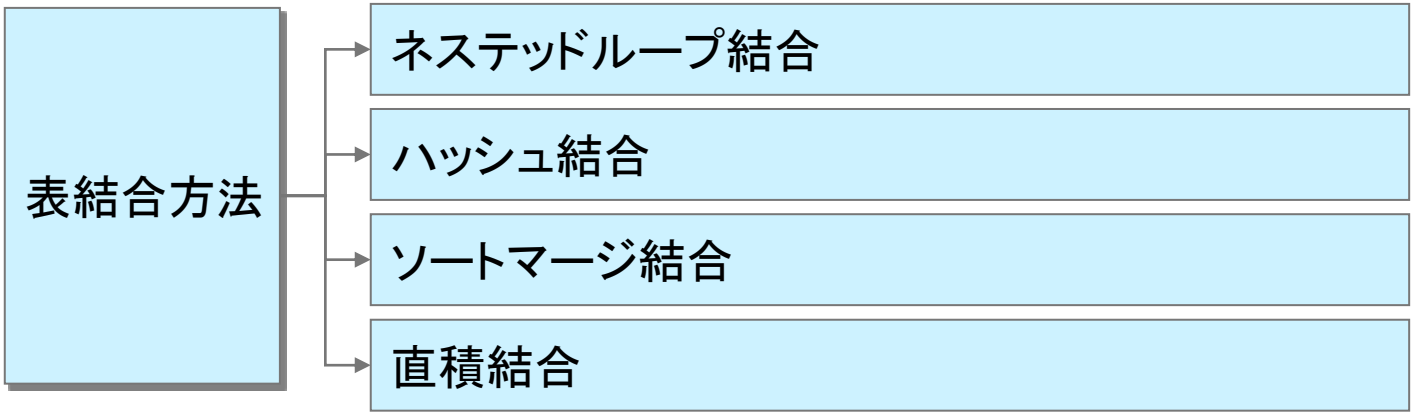


INDEX SKIP SCAN

索引スキップ・スキャンが選択されていたら・・・
- 一般的には、INDEX RANGE SCANと比較すると、効率が悪くなる可能性がある
- 複合索引の先頭の列のカーディナリティが低い場合は効果を得やすい

実行計画を読み解くポイント2
結合方法

- 結合方法の種類



結合方法
ネステッド・ループ結合

- 表の一部分を結合する場合に有効な結合方法
- 結合条件とは関係なく、どのような結合要件でも処理できる
- 先に一方の表（外部表）から行を取り出し、その行と結合する行を、もう一方の表（内部表）から取り出す方法
  - カーディナリティが小さい表を外部表とする（実レコード件数ではなく、カーディナリティが小さい表）

ネステッド・ループが選択されていたら・・・
- 外部表の各行に対して、内部表の全表スキャンが繰り返し実行されるとパフォーマンスが低下しやすい
- 内部表のアクセス効率を上げるため内部表の結合列に索引があると効率的

外部表
DEPT

DEPT_NO	DEPT_NAME
10	SALES
20	RESEARCH
30	SUPPORT

WHERE dept_name= ‘RESEARCH’

DEPT_NO	DEPT_NAME
20	RESEARCH

内部表
EMP

EMP_NO	EMP_NAME	DEPT_NO
1	Tanaka	30
2	Suzuki	20
3	Yoshida	10
4	Watanabe	20
5	Endo	30

外部ループ
```
while {
    外部表から絞込条件に合致する1行を取得する
    1行もなければループから抜ける
    while {                                    内部ループ
        内部表から絞込条件、結合条件に合致する1行を
        取得し、取得した行を結果として返す
        1行もなければループから抜ける
        }
}
```

結合方法
【補足】ネステッド・ループ結合の実行計画例

- ネステッド・ループの実行計画

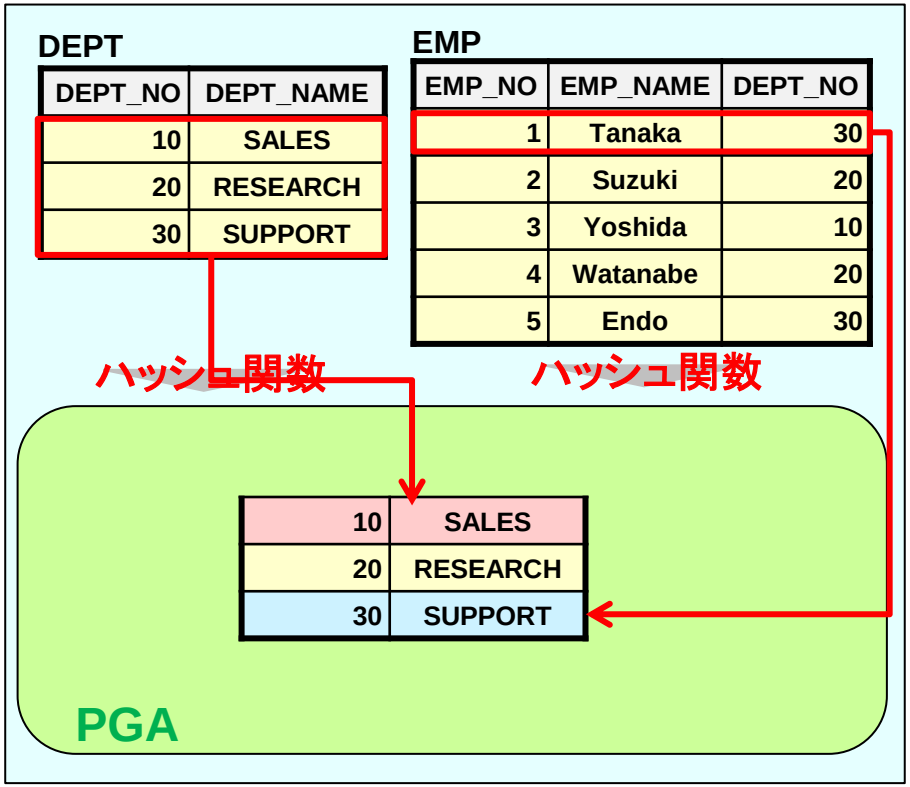
操作	オブジェクト	順序	行	バイト	コスト	CPU (%)	時間	問合せブロック名/オブジェクトの別名	述語	フィルタ	予測
SELECT STATEMENT		6			4	100					
NESTED LOOPS		5						SEL$1			"EMPNO"[NUMBER,22], "ENAME"[VA...
NESTED LOOPS		3	4	136	4	0	0:0:1				"E".ROWID[ROWID,10]
TABLE ACCESS FULL	DEPT	1	1	13	3	0	0:0:1	SEL$1 / D@SEL$1		"D"."DNAME"='RESEARCH'	"D"."DEPTNO"[NUMBER,22]
INDEX RANGE SCAN	EMP_IX2	2	4		0			SEL$1 / E@SEL$1	"E"."DEPTNO"="D"."DEPTNO"		"E".ROWID[ROWID,10]
TABLE ACCESS BY INDEX ROWID	EMP	4	4	84	1	0	0:0:1	SEL$1 / E@SEL$1			"EMPNO"[NUMBER,22], "ENAME"[VA...

1. DEPT表を外部表、EMP表を内部表とする
2. DEPT表をフルスキャンし、条件（DNAME= ‘RESEARCH’ ）に合致する行をフェッチする
3. 2で取得する行数分4～5を繰り返す
4. EMP表の索引EMP_IX2を索引レンジスキャンし、フェッチした行と結合する行を特定する
5. EMP表にアクセスして特定した行を取得する

結合方法
ハッシュ結合

- 表の大部分のデータの等価結合に有効な結合方法
- メモリ上にハッシュテーブルを作り、ハッシュ値を利用したマッチングを行なう
- カーディナリティが小さい方を先に処理する
  - PGA上に作成されるハッシュ表が小さくなるため、結合処理が効率的になる

ハッシュ結合が選択されていたら・・・
- ハッシュ結合はソート/マージ結合より一般的にパフォーマンスが良い（ただし等価結合でのみ使用できる）
- ハッシュ表がメモリ内に収まらない場合、一時表領域を使用するため、ディスクI/Oの発生により性能が劣化しやすい



1. EMP表から抽出条件に合致する結果セットを取り出し、結合条件列のキーをもとにハッシュ表をPGA内に作成する
2. EMP表の結合条件列をハッシュ関数にかけ、結合できるかをハッシュ・テーブルで確認する
3. ハッシュ値が等しいレコードを結合して結果を返す

結合方法
【補足】ハッシュ結合の実行計画例

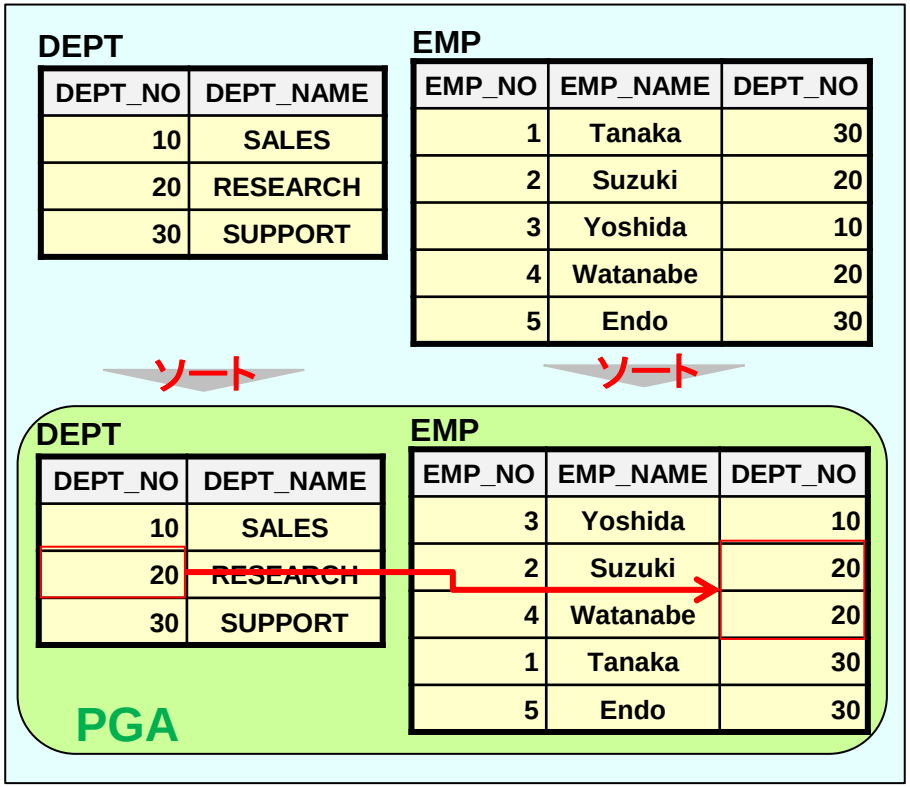
- ハッシュ結合の実行計画例：

操作	オブジェクト	順序	行	バイト	コスト	CPU (%)	時間	問合せブロック名/オブジェクトの別名	述語	フィルタ	予測
SELECT STATEMENT		4			7	100					
HASH JOIN		3	4	136	7	14	0:0:1	SEL$1	"E"."DEPTNO"="D"."DEPTNO"		(#keys=1) "EMPNO"[NUMBER,22], ...
TABLE ACCESS FULL	DEPT	1	1	13	3	0	0:0:1	SEL$1 / D@SEL$1		"D"."DNAME"='RESEARCH'	"D"."DEPTNO"[NUMBER,22]
TABLE ACCESS FULL	EMP	2	12	252	3	0	0:0:1	SEL$1 / E@SEL$1			"EMPNO"[NUMBER,22], "ENAME"[VA...

1. DEPT表をフルスキャンし、絞込条件（DNAME= ‘RESEARCH’ ）に合致するデータを取り出す
2. 結合キーの値をハッシュし、ハッシュ表の対応するパーティションに値を格納する
3. EMP表をフルスキャンする
4. 結合キーの値をハッシュし、該当するパーティションに行があるか確認し、行がある場合には値を適切なパーティションに格納する
5. 結合した結果を返す

結合方法
ソート/マージ結合

- 表の大部分を結合する場合に有効な結合方法
  - 主に結合条件が等価条件ではない場合に使用される
- 結合する双方の表を結合条件列でソートし、結果をマージすることでデータを取り出す



1. DEPT表の結果セットを結合列でPGA内でソートする
2. EMP表の結果セットを結合列でPGA内でソートする
3. ソート結果をPGA内でマージして結果を返す

ソート/マージ結合が選択されていたら・・・

- 一般的に、ハッシュ結合やネステッド・ループのほうが効率的であるため、結合方法を見直す
- 行ソースが前の操作でソート済みである場合、ソートをスキップすることができるため効率的な場合もある
- ソート操作がメモリー内のみで実行できない場合、ディスクI/Oの発生により性能が劣化しやすい

結合方法
【補足】ソートマージ結合の実行計画例

- ソート/マージ結合の実行計画例:

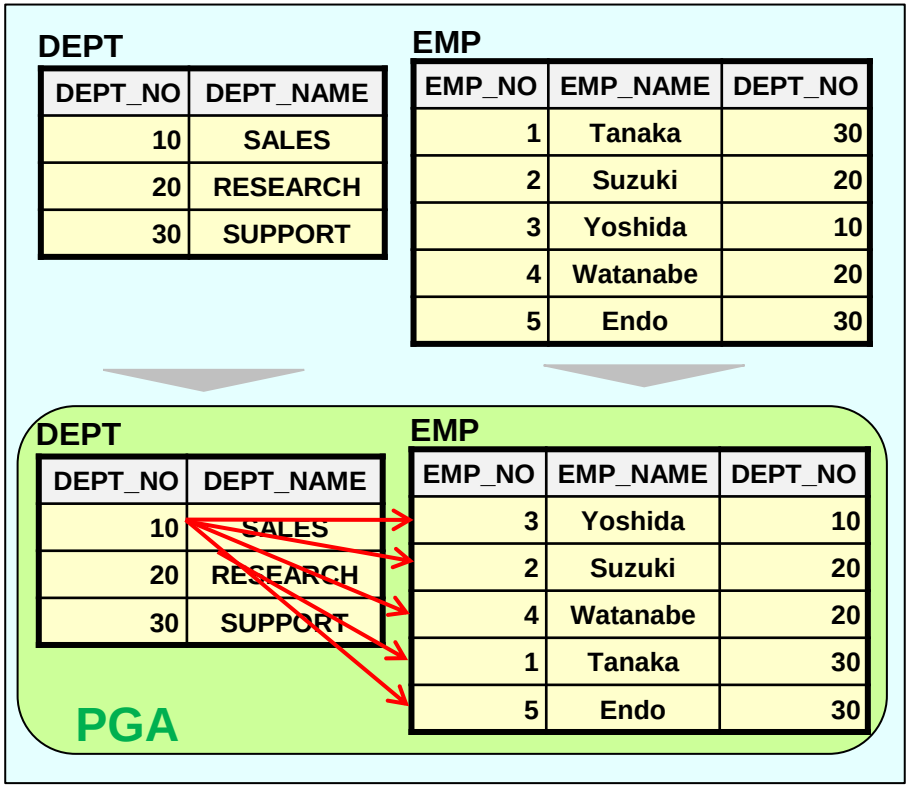
操作	オブジェクト	順序	行	バイト	コスト	CPU (%)	時間	問合せブロック名/オブジェクトの別名	述語	フィルタ	予測
SELECT STATEMENT		6			8	100					
MERGE JOIN		5	4	136	8	25	0:0:1	SEL$1			"EMPNO"[NUMBER,22], "ENAME"[VA...
SORT JOIN		2	1	13	4	25	0:0:1				(#keys=1) "D"."DEPTNO"[NUMBER,...
TABLE ACCESS FULL	DEPT	1	1	13	3	0	0:0:1	SEL$1 / D@SEL$1		"D"."DNAME"='RESEARCH'	"D"."DEPTNO"[NUMBER,22]
SORT JOIN		4	12	252	4	25	0:0:1		"E"."DEPTNO"="D"."DEPTNO"	"E"."DEPTNO"="D"."DEPTNO"	(#keys=1) "E"."DEPTNO"[NUMBER,...
TABLE ACCESS FULL	EMP	3	12	252	3	0	0:0:1	SEL$1 / E@SEL$1			"EMPNO"[NUMBER,22], "ENAME"[VA...

1. DEPT表をフルスキャンする
2. DEPT表の結果セットを結合列（DEPTNO）でソートする
3. EMP表をフルスキャンする
4. EMP表の結果セットを結合列（DEPTNO）でソートしながら2の結果と結合する

結合方法
直積演算

- 表の結合条件がない場合に使用される結合方法
- ２つの表の結果セット全行を直積（掛け算）する
  - m行×n行

直積演算が選択されていたら・・・
- 一般的には非効率な結合方法であるため、検索条件の指定方法を確認する
- スター・スキーマ構造となっているDWHシステムでは有効な場合もある



1. DEPT表をフルスキャンする
2. EMP表をフルスキャンする
3. 2の結果セットをソートする
4. 2と4の結果を全て組み合わせる

結合方法
【補足】直積結合の実行計画例

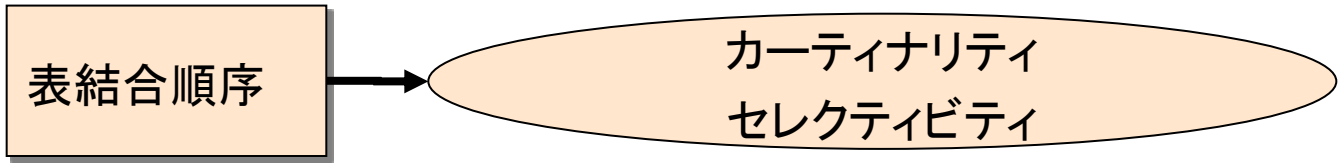
- 直積演算の実行計画例:

操作	オブジェクト	順序	行	バイト	コスト	CPU (%)	時間	問合せブロック名/オブジェクトの別名	予測
SELECT STATEMENT		5			9	100			
MERGE JOIN CARTESIAN		4	48	864	9	0	0:0:1	SEL$1	"EMPNO"[NUMBER,22], "ENAME"[VA...
TABLE ACCESS FULL	DEPT	1	4		3	0	0:0:1	SEL$1 / D@SEL$1	
BUFFER SORT		3	12	216	6	0	0:0:1		(#keys=0) "EMPNO"[NUMBER,22], ...
TABLE ACCESS FULL	EMP	2	12	216	2	0	0:0:1	SEL$1 / E@SEL$1	"EMPNO"[NUMBER,22], "ENAME"[VA...

1. DEPT表をフルスキャンする
2. EMP表をフルスキャンする
3. 2の結果セットをソートする
4. 2と4の結果をマージ

実行計画を読み解くポイント3
結合順序

- 結合順序の決定方法
  - 登場するオブジェクトを抽出し、列の関係を整理する
  - 開始点（どの表から結合するか）を考える
  - 結合方法と結合処理のアクセス回数を考える



- カーディナリティ：C
  - 行ソースから戻される行の予測数
  - 結合順序を考えるために使用
    - 選択率（割合）が低い（=絞り込むことができる）表から結合する
- セレクティビティ（選択率）：S
  - SQLの条件（条件の組み合わせ）にヒットする行の割合
  - 結合方法を考えるために使用
    - 選択率が高い時にはハッシュ結合等
    - 選択率が低い時にはネステッド・ループ結合　等

結合順序

表の関係と結合順序を整理する

- サンプルで使用するSQL文と統計情報

[SQL文]

```
SELECT count(*)
FROM   tab1 t1, tab2 t2, tab3 t3 , tab4 t4 , tab5 t5
WHERE t1.id = t2.id
AND t1.id = t3.id
AND t2.class = t5.class
AND t3.class = t4.class
AND t4.flag = 'Y'
AND t5.num = TO_NUMBER(:b1)
AND t4.code = TO_NUMBER(:b2)
AND t1.start_date <=
    (TO_DATE(:b3,'yyyymmdd')+1)
AND t1.end_date >
    TO_DATE(:b3,'yyyymmdd')
```

[表の統計情報]

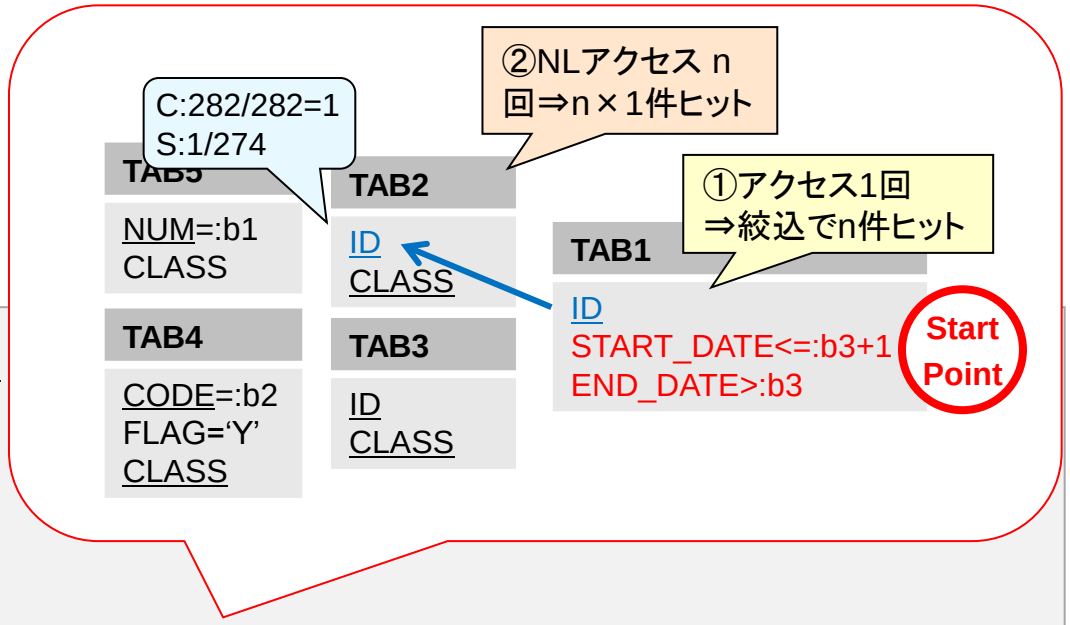
OWNER	TABLE_NAME	COLUMN_NAME	NUM_ROWS	NUM_DISTINC
SCOTT	TAB1	ID	275	275
SCOTT	TAB1	START_DATE	275	5
SCOTT	TAB1	END_DATE	275	1
SCOTT	TAB2	ID	282	282
SCOTT	TAB2	CLASS	282	17
SCOTT	TAB3	ID	17442	274
SCOTT	TAB3	CLASS	17442	8210
SCOTT	TAB4	CODE	834030	834030
SCOTT	TAB4	FLAG	834030	1
SCOTT	TAB4	CLASS	834030	834030
SCOTT	TAB5	NUM	133	132
SCOTT	TAB5	CLASS	133	133

結合順序
表の関係と結合順序を整理する　例1

［表の統計情報］

TAB1	ID	275	275
TAB1	START_DATE	275	5
TAB1	END_DATE	275	1
TAB2	ID	282	282
TAB2	CLASS	282	17

- TAB1とTAB2を結合



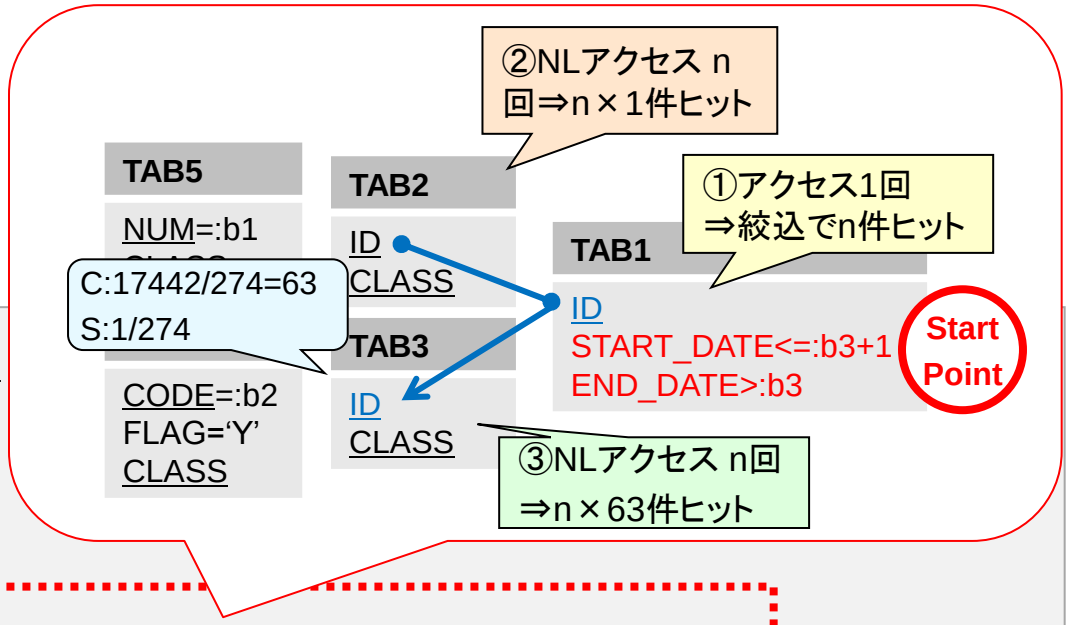

```
Execution Plan
----------------------------------
SELECT STATEMENT   GOAL: CHOOSE
 SORT (AGGREGATE)
  NESTED LOOPS
   ├NESTED LOOPS
   │ ├NESTED LOOPS
   │ │  ├ NESTED LOOPS
   │ │  │ ├TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'TAB1'
   │ │  │ └INDEX   GOAL: ANALYZED (RANGE SCAN) OF 'TAB2_PK' (UNIQUE)
   │ │  └INDEX   GOAL: ANALYZED (FULL SCAN) OF 'TAB3_I1' (NON-UNIQUE)
   │ └ TABLE ACCESS   GOAL: ANALYZED (BY INDEX ROWID) OF 'TAB4'
   │    INDEX   GOAL: ANALYZED (UNIQUE SCAN) OF 'TAB4_PK' (UNIQUE)
   └TABLE ACCESS   GOAL: ANALYZED (BY INDEX ROWID) OF 'TAB5'
     INDEX   GOAL: ANALYZED (UNIQUE SCAN) OF 'TAB5_PK' (UNIQUE)
```

結合順序
表の関係と結合順序を整理する　例2

［表の統計情報］

TAB	COL	NUM_ROWS	NUM_DISTINC
TAB3	ID	17442	274
TAB3	CLASS	17442	8210

- 結果セットをTAB3と結合



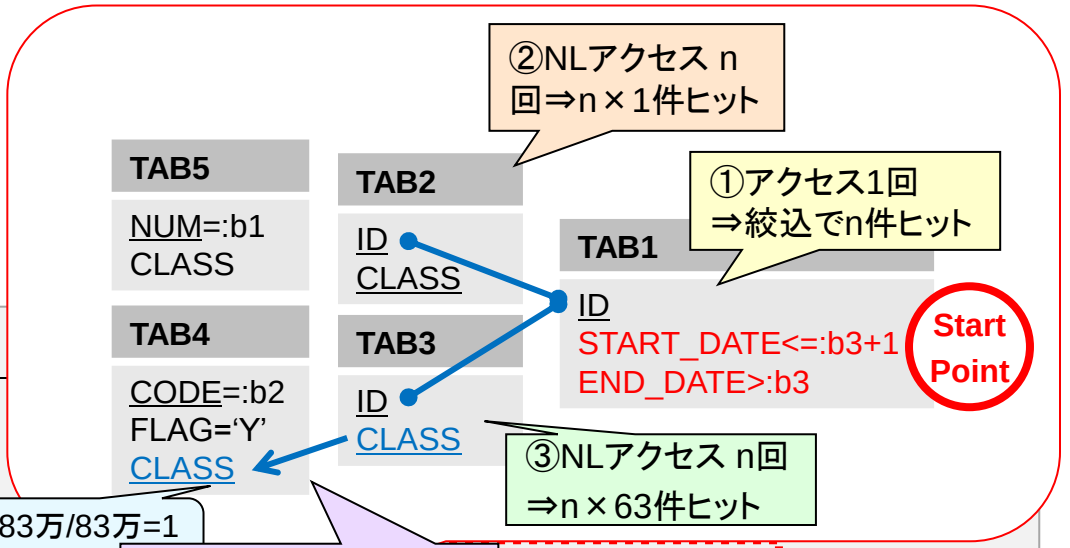

```
Execution Plan
----------------------------------
SELECT STATEMENT   GOAL: CHOOSE
 SORT (AGGREGATE)
  NESTED LOOPS
    ├NESTED LOOPS
    │ ├NESTED LOOPS
    │ │  ├ NESTED LOOPS
    │ │  │ ├TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'TAB1'
    │ │  │ └INDEX   GOAL: ANALYZED (RANGE SCAN) OF 'TAB2_PK' (UNIQUE)
    │ │  └INDEX   GOAL: ANALYZED (FULL SCAN) OF 'TAB3_I1' (NON-UNIQUE)
    │ └ TABLE ACCESS   GOAL: ANALYZED (BY INDEX ROWID) OF 'TAB4'
    │     INDEX   GOAL: ANALYZED (UNIQUE SCAN) OF 'TAB4_PK' (UNIQUE)
    └TABLE ACCESS   GOAL: ANALYZED (BY INDEX ROWID) OF 'TAB5'
       INDEX   GOAL: ANALYZED (UNIQUE SCAN) OF 'TAB5_PK' (UNIQUE)
```

結合順序
表の関係と結合順序を整理する　例3

［表の統計情報］

TAB	COL	NUM_ROWS	NUM_DISTINC
TAB4	CODE	834030	834030
TAB4	FLAG	834030	1
TAB4	CLASS	834030	834030

- 結果セットをTAB4と結合



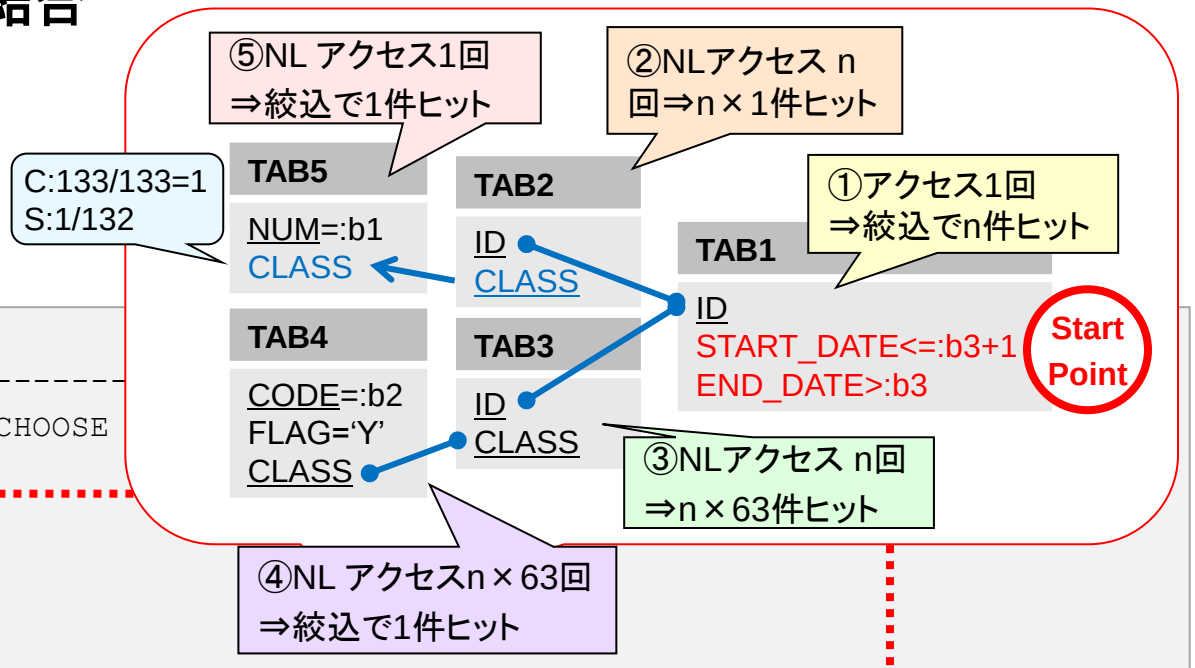

```
Execution Plan
----------------------------------------------------------
SELECT STATEMENT   GOAL: CHOOSE
 SORT (AGGREGATE)
  NESTED LOOPS
   NESTED LOOPS
    NESTED LOOPS
     NESTED LOOPS
      TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'TAB1'
      INDEX   GOAL: ANALYZED (RANGE SCAN) OF 'TAB2_PK' (UNIQUE)
     INDEX   GOAL: ANALYZED (FULL SCAN) OF 'TAB3_I1' (NON-UNIQUE)
    TABLE ACCESS   GOAL: ANALYZED (BY INDEX ROWID) OF 'TAB4'
     INDEX   GOAL: ANALYZED (UNIQUE SCAN) OF 'TAB4_PK' (UNIQUE)
   TABLE ACCESS   GOAL: ANALYZED (BY INDEX ROWID) OF 'TAB5'
    INDEX   GOAL: ANALYZED (UNIQUE SCAN) OF 'TAB5_PK' (UNIQUE)
```

結合順序
表の関係と結合順序を整理する　例4

［表の統計情報］

TAB	COL	NUM_ROWS	NUM_DISTINC
TAB5	NUM	133	133
TAB5	CLASS	133	133

- 結果セットをTAB5と結合



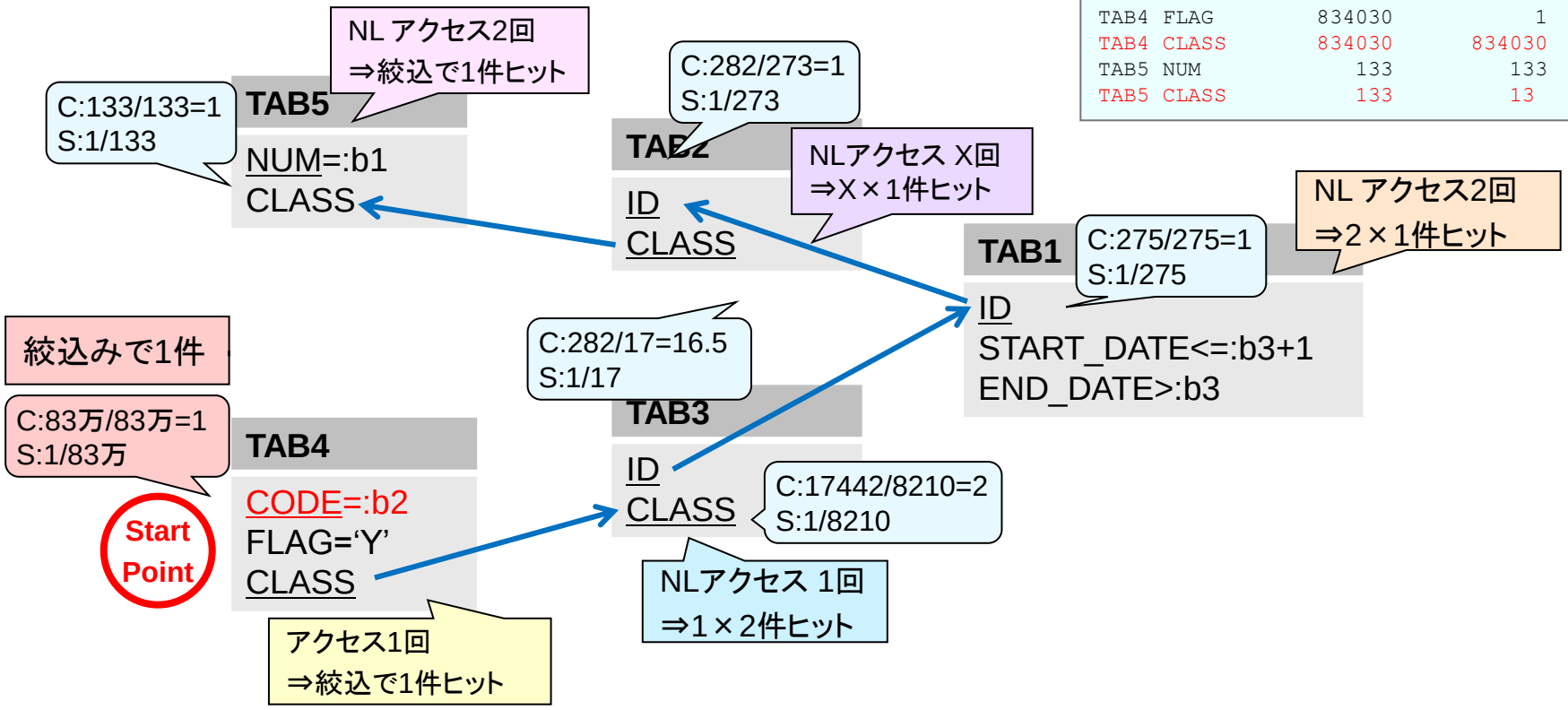

```
Execution Plan
----------------------------------------------------------
SELECT STATEMENT   GOAL: CHOOSE
 SORT (AGGREGATE)
  NESTED LOOPS
    ├NESTED LOOPS
    │ ├NESTED LOOPS
    │ │  ├ NESTED LOOPS
    │ │  │ ├TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'TAB1'
    │ │  │ └INDEX   GOAL: ANALYZED (RANGE SCAN) OF 'TAB2_PK' (UNIQUE)
    │ │  └INDEX   GOAL: ANALYZED (FULL SCAN) OF 'TAB3_I1' (NON-UNIQUE)
    │ └ TABLE ACCESS   GOAL: ANALYZED (BY INDEX ROWID) OF 'TAB4'
    │     INDEX   GOAL: ANALYZED (UNIQUE SCAN) OF 'TAB4_PK' (UNIQUE)
    └TABLE ACCESS   GOAL: ANALYZED (BY INDEX ROWID) OF 'TAB5'
      INDEX   GOAL: ANALYZED (UNIQUE SCAN) OF 'TAB5_PK' (UNIQUE)
```

結合順序
別の結合順序を検討

- 別の結合順序を検討する
  - より絞り込みのできる開始点はないか
  - 索引を作成することによって、アクセス方法を減らすことはできないか　等

［表の統計情報］

TAB	COL	NUM_ROWS	NUM_DISTINC
TAB1	ID	275	275
TAB1	START_DATE	275	5
TAB1	END_DATE	275	1
TAB2	ID	282	282
TAB2	CLASS	282	17
TAB3	ID	17442	274
TAB3	CLASS	17442	8210
TAB4	CODE	834030	834030
TAB4	FLAG	834030	1
TAB4	CLASS	834030	834030
TAB5	NUM	133	133
TAB5	CLASS	133	13



結合順序
結合順序の比較

- 結合の開始点と結合順序、結合方式の関係を整理して、最適な結合方法を検討する

元の実行計画例

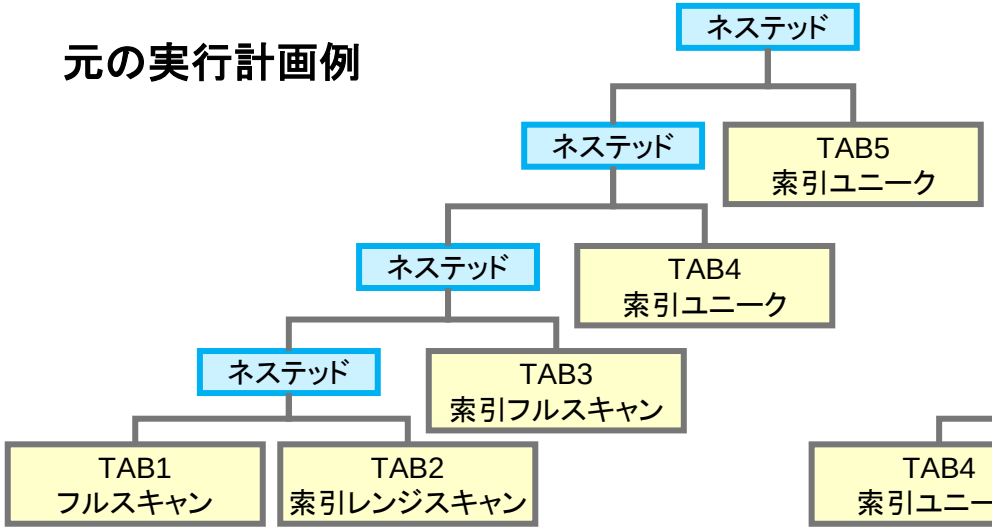


表	アクセス回数	カーディナリティ
TAB1	1回	n行
TAB2	n回	n×1行
TAB3	n回	n×63.5行
TAB4	n×63.5回	1行
TAB5	1回	1行

別の実行計画例

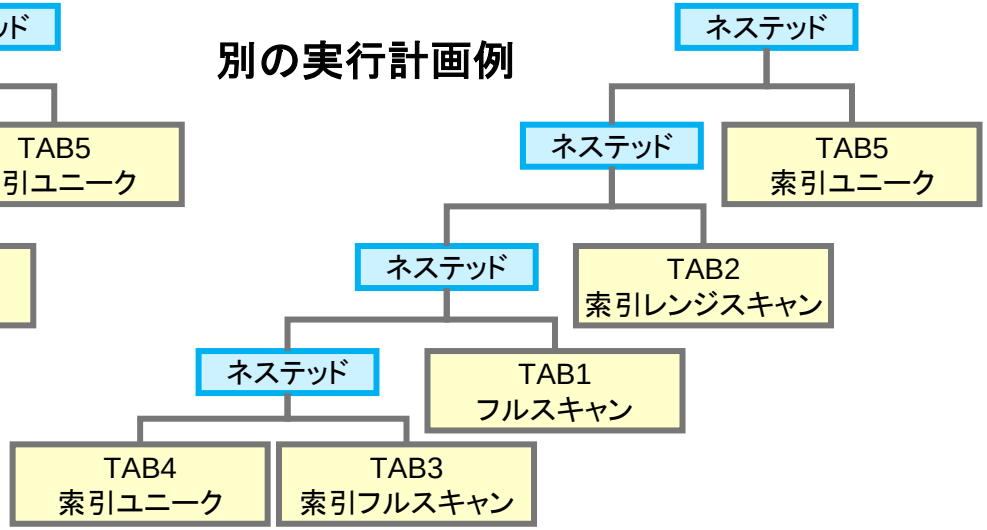
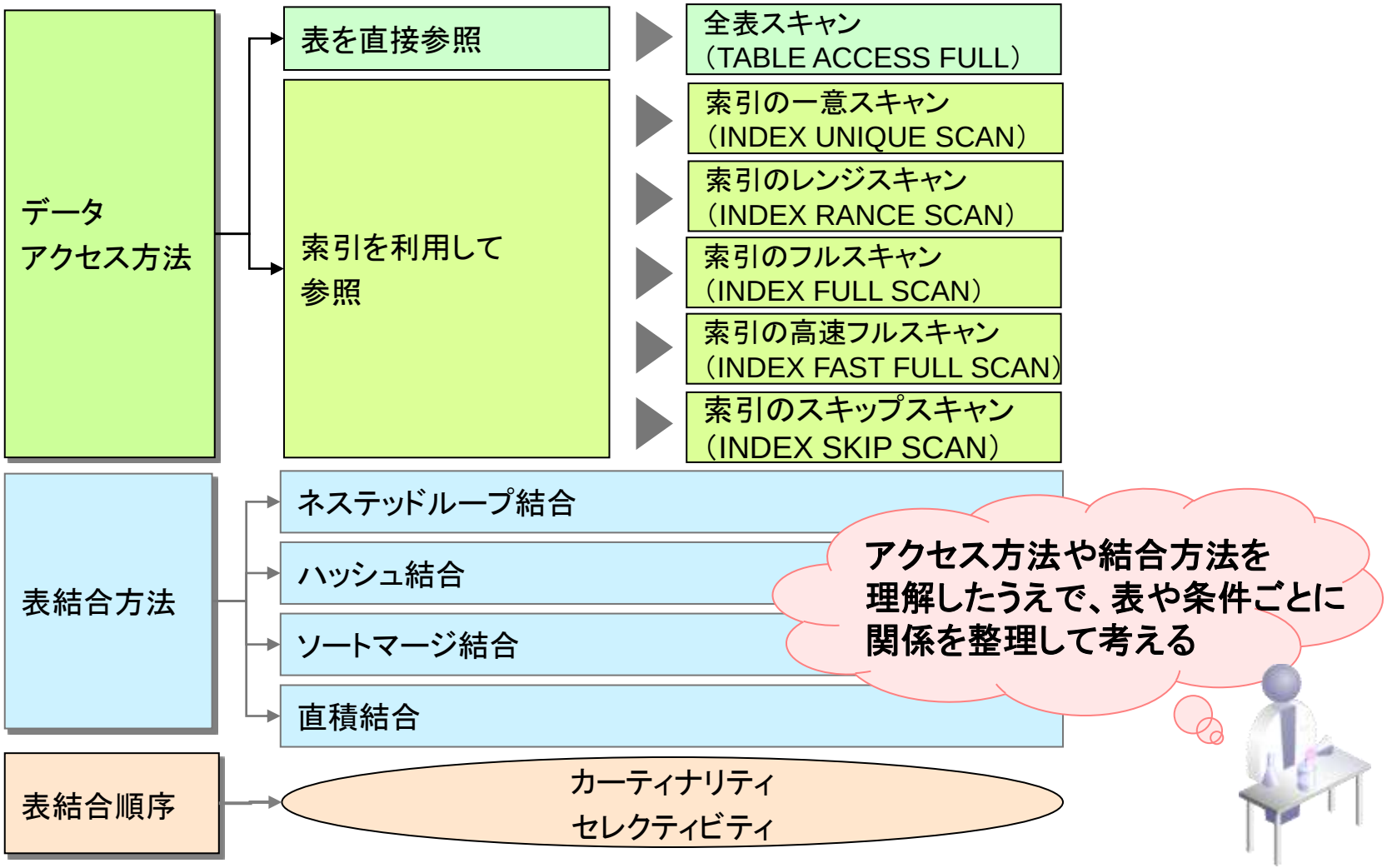


表	アクセス回数	カーディナリティ
TAB4	1回	1行
TAB3	1回	2行
TAB1	2回	2行
TAB2	2回	2行
TAB5	2回	1行

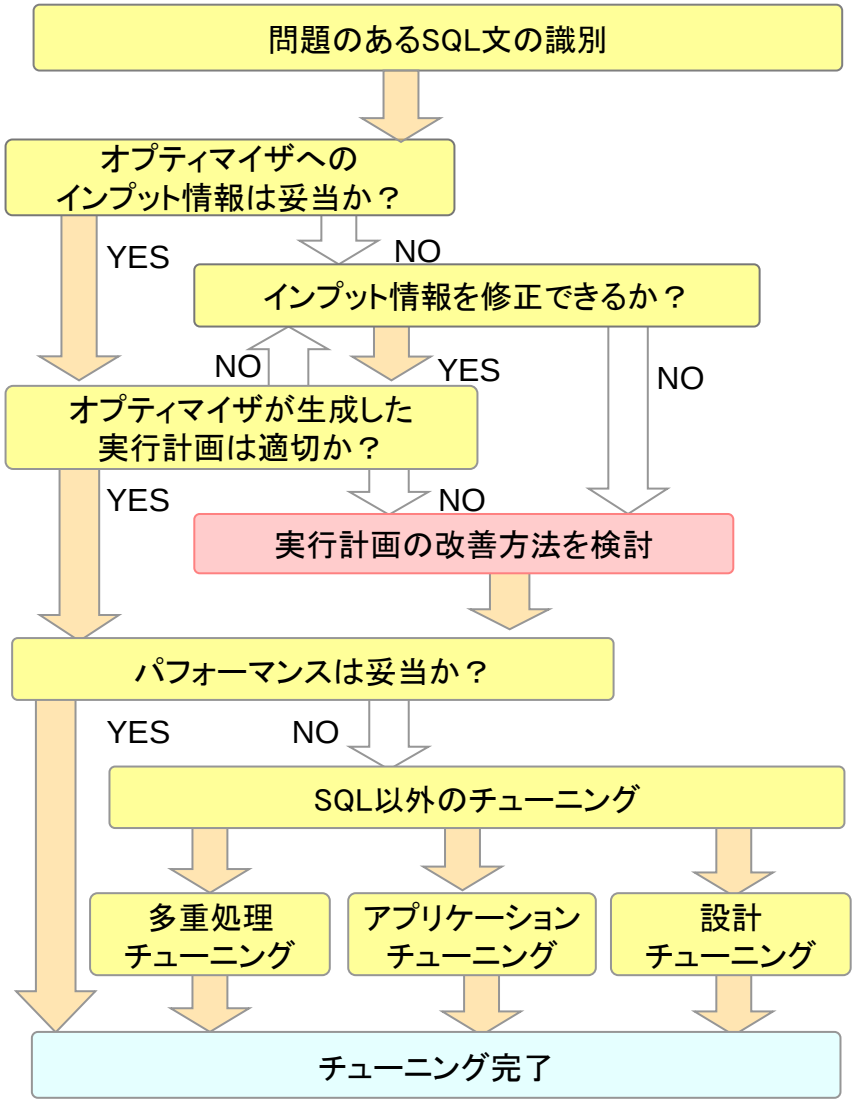
実行計画を判断するポイント
まとめ



SQLチューニングの流れ
コストの高い処理の改善方法を検討

- SQLチューニングの流れ
  - 問題のあるSQL文を識別する
  - 前提条件を確認する
    - オプティマイザ統計は適切か
  - 実行計画を読み解く
  - 読み解いた実行計画から、コストの高い処理の改善方法を検討する
    - 結合順序、方法は変更できるか
    - 効率的な索引を作成できるか
    - SQLの構文を変更できるか
    - ヒント句を利用できるか

- 解決できない場合には、SQLチューニング以外の方法を検討する



コストの高い処理の改善方法の検討

読み解いた実行計画から、コストの高い処理の改善方法を検討する

代表的なチューニング例

- 結合順序、方法を変える
- 効率的な索引を作成する
- SQLの構文を変更する
- ヒント句を利用する

最適なアクセス・パスや結合方法は
データ量やデータの偏りによって異なるため
「必ず早くなる」という正解はない

アクセス・パスや結合方法を理解したうえで
ケースごとに考えることが重要

結合順序、方法を変える①
最適な索引作成による結合順序の改善

- 索引で絞り込めない場合、ハッシュ・ジョインが選択されやすい

※各表の主キー列にのみ索引が作成されている

```
SQL> SELECT cust_last_name  FROM cust c, prod p, sales s
 2 > WHERE  c.cust_id=s.cust_id
 3 > AND    p.prod_id=s.prod_id AND c.cust_last_name='Ruddy' AND s.time_id >= '00-01-01'
```

実行計画

```
-------------------------------------------------------------------------------------
| Id  | Operation           | Name         | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |              |  3984 |   132K|  1646   (2)| 00:00:20 |
|   1 |  NESTED LOOPS       |              |  3984 |   132K|  1646   (2)| 00:00:20 |
|*  2 |   HASH JOIN         |              |  3984 |   116K|  1646   (2)| 00:00:20 |
|*  3 |    TABLE ACCESS FULL| CUST         |    61 |   793 |   405   (1)| 00:00:05 |
|*  4 |    TABLE ACCESS FULL| SALES        |   460K|  7637K|  1239   (2)| 00:00:15 |
|*  5 |    INDEX RANGE SCAN | PROD_ID_IDX  |     1 |     4 |     0   (0)| 00:00:01 |
-------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("C"."CUST_ID"="S"."CUST_ID")
   3 - filter("C"."CUST_LAST_NAME"='Ruddy')
   4 - filter("S"."TIME_ID">='00-01-01')
   5 - access("P"."PROD_ID"="S"."PROD_ID")
```

結合条件や検索条件列に索引が作成されているか？

結合順序、方法を変える①
最適な索引作成による結合順序の改善

- 索引で行を絞り込み、ネステッド・ループにすることにより、処理を効率化

※各表の検索条件列に索引を作成

```
CREATE INDEX sales_cust_id_idx ON sales(cust_id);
CREATE INDEX cust_cust_lname_idx ON customers(cust_last_name);
CREATE INDEX sales_time_id_idx ON sales(time_id);
CREATE INDEX sales_prod_id_idx ON sales(prod_id);

SQL> SELECT cust_last_name  FROM cust c, prod p, sales s
 2 > WHERE  c.cust_id=s.cust_id
 3 > AND    p.prod_id=s.prod_id AND c.cust_last_name='Ruddy' AND s.time_id >= '00-01-01'
```

実行計画

```
-------------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                     |  3984 |   132K|   949   (0)| 00:00:12 |
|   1 |  NESTED LOOPS                 |                     |  3984 |   132K|   949   (0)| 00:00:12 |
|   2 |   NESTED LOOPS                |                     |  3984 |   116K|   949   (0)| 00:00:12 |
|   3 |    TABLE ACCESS BY INDEX ROWID| CUSTOMERS           |    61 |   793 |    12   (0)| 00:00:01 |
|*  4 |     INDEX RANGE SCAN          | CUST_CUST_LNAME_IDX |    61 |       |     1   (0)| 00:00:01 |
|*  5 |    TABLE ACCESS BY INDEX ROWID| SALES               |       |       |             (0)| 00:00:02 |
|*  6 |     INDEX RANGE SCAN          | SALES_CUST_ID_IDX   |       |       |             (0)| 00:00:01 |
|*  7 |   INDEX RANGE SCAN            | PROD_PROD_ID_IDX    |       |       |             (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------------
```

より効率的な結合方法が選択され、コストも1646⇒949に下がった

効率的な索引を作成する①
単一列索引、複合列索引

- 索引を使わない検索が行われているSQL文

※各表の主キー列にのみ索引が作成されている

```
SQL> SELECT c.cust_last_name    FROM customers c
  2    WHERE  cust_gender = 'M'AND cust_year_of_birth > 1970 AND cust_city ='Nagoya';

実行計画
----------------------------------------------------------------------------
| Id  | Operation          | Name      | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |    12 |   288 |   405   (1)| 00:00:05 |
|*  1 |  TABLE ACCESS FULL| CUSTOMERS |    12 |   288 |   405   (1)| 00:00:05 |
----------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("CUST_CITY"='Nagoya' AND "CUST_YEAR_OF_BIRTH">1970 AND
              "CUST_GENDER"='M')
```

全表走査が行われている

効率的な索引を作成する②
単一列索引、複合列索引

- 検索条件の各列に単一列索引を作成することで、処理を効率化

```
SQL> CREATE INDEX cust_cust_city_idx ON customers(cust_city);
SQL> CREATE INDEX cust_cust_year_of_birth ON customers(cust_year_of_birth);
SQL> CREATE INDEX cust_cust_gender ON customers(cust_gender);

SQL> SELECT c.custlast_name    FROM customers c
  2    WHERE  cust_gender = 'M'AND cust_year_of_birth > 1970 AND cust_city ='Nagoya';
```

実行計画

```
-------------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name                | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                     |    12 |   288 |    63   (2)| 00:00:01 |
|*  1 |  TABLE ACCESS BY INDEX ROWID  | CUSTOMERS           |    12 |   288 |    63   (2)| 00:00:01 |
|   2 |   BITMAP CONVERSION TO ROWIDS |                     |       |       |            |          |
|   3 |    BITMAP AND                 |                     |       |       |            |          |
|   4 |     BITMAP CONVERSION FROM ROWIDS|                  |       |       |            |          |
|*  5 |      INDEX RANGE SCAN         | CUST_CUST_CITY_IDX  |    90 |       |     1   (0)| 00:00:01 |
|   6 |     BITMAP CONVERSION FROM ROWIDS|                  |       |       |            |          |
|*  7 |      INDEX RANGE SCAN         | CUST_CUST_GENDER_IDX|    90 |       |    51   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("CUST_YEAR_OF_BIRTH">1970)
   5 - access("CUST_CITY"='Nagoya')
   7 - access("CUST_GENDER"='M')
```

各索引に対してバラバラにアクセスしている

効率的な索引を作成する③
単一列索引、複合列索引

- 検索条件列に複合索引を作成することで、さらに処理を効率化

```
SQL> CREATE INDEX cust_composit_idx
 2 > ON customers(cust_city,cust_year_of_birth,cust_gender);

SQL> SELECT c.custlast_name    FROM customers c
  2    WHERE  cust_gender = 'M'AND cust_year_of_birth > 1970 AND cust_city ='Nagoya';
```

実行計画

```
-----------------------------------------------------------------------------------------------
| Id  | Operation                    | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |                   |    12 |   288 |    13   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| CUSTOMERS         |    12 |   288 |    13   (0)| 00:00:01 |
|*  2 |   INDEX RANGE SCAN           | CUST_COMPOSIT_IDX |    12 |       |     1   (0)| 00:00:01 |
-----------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("CUST_CITY"='Nagoya' AND "CUST_YEAR_OF_BIRTH">1970 AND "CUST_GENDER"='M')
       filter("CUST_GENDER"='M')
```

複合索引を使うことにより、処理がシンプルになり、コストも大幅に下がっている

SQLの構文を変更する①
結合や集計処理をする前に対象行数を減らす

- グループ化の処理をした後に、条件を評価するSQL文

```
SQL> SELECT p.prod_subcategory,sum(s.amount_sold)
  2  FROM   products p ,sales s
  3  WHERE  p.prod_id = s.prod_id
  4  GROUP BY p.prod_subcategory
  5  HAVING prod_subcategory='Camera Media';
```

全ての行をグループ化してから条件で絞り込み

実行計画

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time	TQ	IN-OUT	PQ Distrib
0	SELECT STATEMENT		1	27	704 (3)	00:00:09			
1	PX COORDINATOR								
2	PX SEND QC (RANDOM)	:TQ10002	1	27	704 (3)	00:00:09	Q1,02	P->S	QC (RAND)
* 3	FILTER						Q1,02	PCWC	
4	HASH GROUP BY		1	27	704 (3)	00:00:09	Q1,02	PCWP	
5	PX RECEIVE		1	27	704 (3)	00:00:09	Q1,02	PCWP	
6	PX SEND HASH	:TQ10001	1	27	704 (3)	00:00:09	Q1,01	P->P	HASH
7	HASH GROUP BY		1	27	704 (3)	00:00:09	Q1,01	PCWP	
* 8	HASH JOIN		918K	23M	689 (1)	00:00:09	Q1,01	PCWP	
9	BUFFER SORT						Q1,01	PCWC	
10	PX RECEIVE		72	1296	1 (0)	00:00:01	Q1,01	PCWP	
11	PX SEND BROADCAST	:TQ10000	72	1296	1 (0)	00:00:01		S->P	BROADCAST
12	INDEX FULL SCAN	PROD_ID_CATEGORY_IDX	72	1296	1 (0)	00:00:01			
13	PX BLOCK ITERATOR		918K	8075K	686 (1)	00:00:09	Q1,01	PCWC	
14	TABLE ACCESS FULL	SALES	918K	8075K	686 (1)	00:00:09	Q1,01	PCWP	

SQLの構文を変更する②

結合や集計処理をする前に対象行数を減らす

- 条件を評価して対象行を絞ったあとに、グループ化の処理をすることで効率化

```
SQL> SELECT p.prod_subcategory,sum(s.amount_sold)
  2  FROM   products p ,sales s
  3  WHERE  p.prod_id = s.prod_id
  4  AND    prod_subcategory='Camera Media'
  5  GROUP BY p.prod_subcategory;
```

条件に合う行のみをグループ化

この項文のほうがコストが低い

実行計画

```
----------------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name                       | Rows  | Bytes | Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                            |     1 |    27 |   392   (1)| 00:00:05 |
|   1 |  SORT GROUP BY NOSORT         |                            |     1 |    27 |   392   (1)| 00:00:05 |
|   2 |   NESTED LOOPS                |                            |       |       |            |          |
|   3 |    NESTED LOOPS               |                            | 38285 |  1009K|   392   (1)| 00:00:05 |
|*  4 |     INDEX RANGE SCAN          | PROD_SUB_CATEGORY_ID_IDX   |     3 |    54 |     1   (0)| 00:00:01 |
|*  5 |     INDEX RANGE SCAN          | SALES_PROD_ID_IDX          | 12762 |       |    26   (0)| 00:00:01 |
|   6 |    TABLE ACCESS BY INDEX ROWID| SALES                      | 12762 |   112K|   130   (0)| 00:00:02 |
----------------------------------------------------------------------------------------------------------
```

ヒント句を利用する
索引や結合方法を明示的に指定する

- より良い検索方法、結合方法が分かっている場合には、ヒント句で指定
  ※ただし、実行計画が固定されてしまうので必要最低限にする

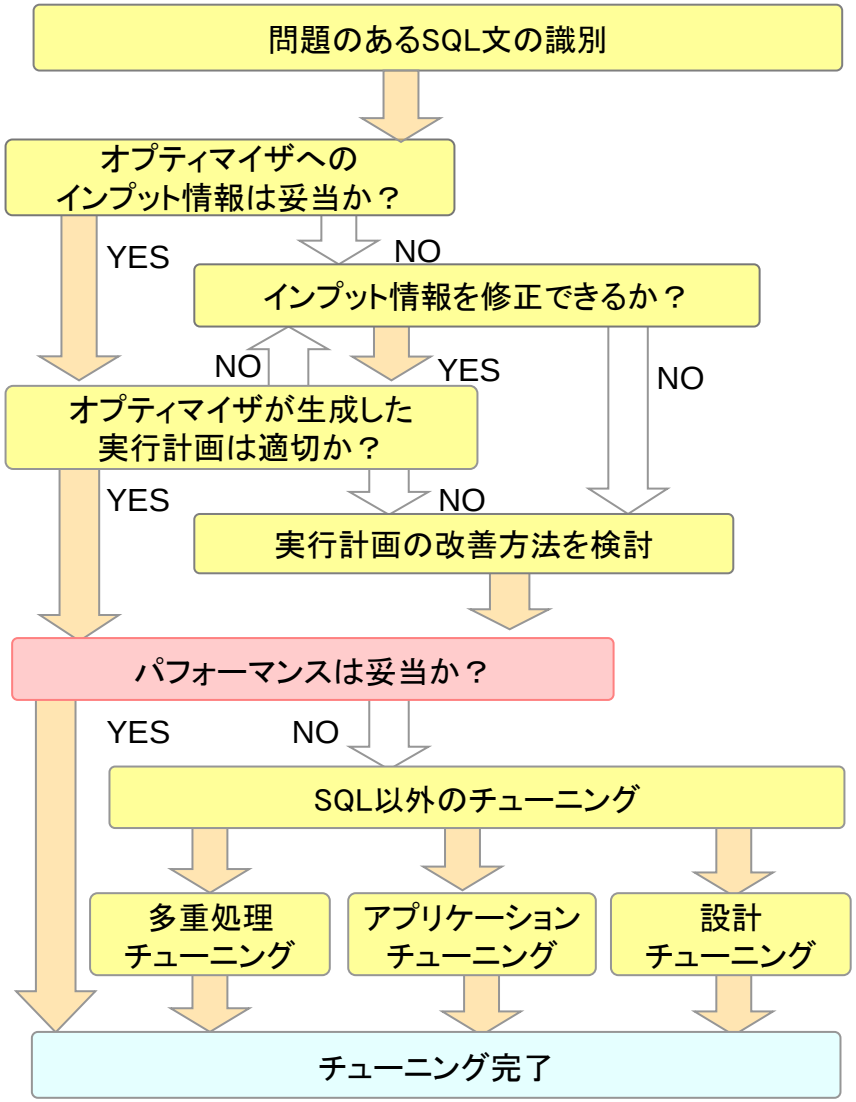
```
SQL> SELECT /*+use_NL(e d)*/ *
 2 > FROM   hr.employees e , hr.departments d
 3 > WHERE  e.department_id = d.department_id
  実行計画
  ----------------------------------------------------------------------------------------------------
  | Id  | Operation                     | Name              | Rows  | Bytes | Cost (%CPU)| Time     |
  ----------------------------------------------------------------------------------------------------
  |   0 | SELECT STATEMENT              |                   |   106 |  9540 |    14   (0)| 00:00:01 |
  |   1 |  NESTED LOOPS                 |                   |       |       |            |          |
  |   2 |   NESTED LOOPS                |                   |   106 |  9540 |    14   (0)| 00:00:01 |
  |   3 |    TABLE ACCESS FULL          | DEPARTMENTS       |    27 |   567 |     3   (0)| 00:00:01 |
  |*  4 |    INDEX RANGE SCAN           | EMP_DEPARTMENT_IX |    10 |       |     0   (0)| 00:00:01 |
  |   5 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES         |     4 |   276 |     1   (0)| 00:00:01 |
  ----------------------------------------------------------------------------------------------------


SQL> SELECT /*+use_merge(e d)*/ *
 2 > FROM   hr.employees e , hr.departments d
 3 > WHERE  e.department_id = d.department_id
  実行計画
  ----------------------------------------------------------------------------------------------------
  | Id  | Operation                     | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
  ----------------------------------------------------------------------------------------------------
  |   0 | SELECT STATEMENT              |             |   106 |  9540 |     6  (17)| 00:00:01 |
  |   1 |  MERGE JOIN                   |             |   106 |  9540 |     6  (17)| 00:00:01 |
  |   2 |   TABLE ACCESS BY INDEX ROWID| DEPARTMENTS |    27 |   567 |     2   (0)| 00:00:01 |
  |   3 |    INDEX FULL SCAN            | DEPT_ID_PK  |    27 |       |     1   (0)| 00:00:01 |
  |*  4 |   SORT JOIN                   |             |   107 |  7383 |     4  (25)| 00:00:01 |
  |   5 |    TABLE ACCESS FULL          | EMPLOYEES   |   107 |  7383 |     3   (0)| 00:00:01 |
  ----------------------------------------------------------------------------------------------------
```

SQLチューニングの流れ
パフォーマンスは改善されたかの確認

- SQLチューニングの流れ
  - 問題のあるSQL文を識別する
  - 前提条件を確認する
    - オプティマイザ統計は適切か
  - 実行計画を読み解く
  - 読み解いた実行計画から、コストの高い処理の改善方法を検討する
    - 結合順序、方法は変更できるか
    - 効率的な索引を作成できるか
    - SQLの構文を変更できるか
    - ヒント句を利用できるか

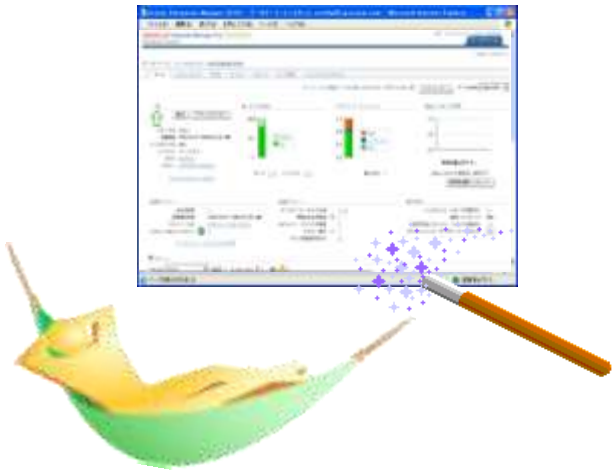
- 解決できない場合には、SQLチューニング以外の方法を検討する



SQLチューニングの流れ
【補足】自動SQLチューニングを活用したチューニング手順

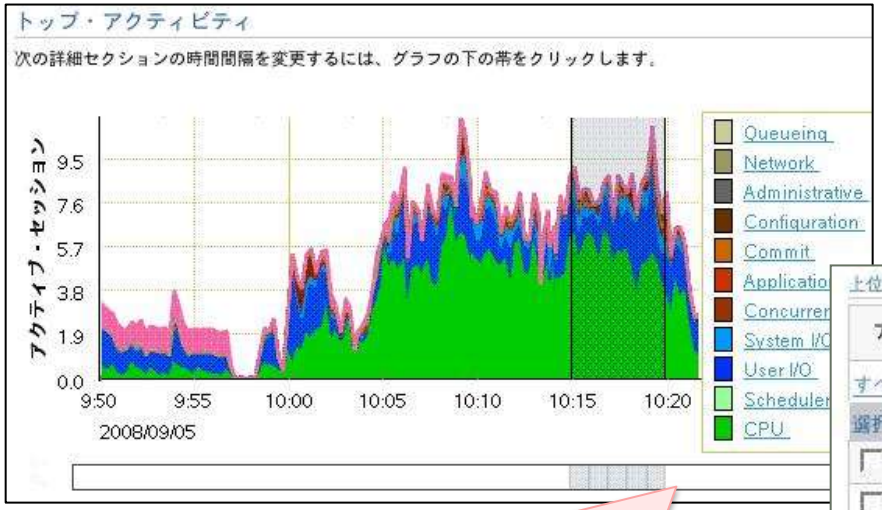
- SQLチューニングの流れ
  - 問題のあるSQL文を識別する
  - 前提条件を確認する
    - オプティマイザ統計は適切か
  - 実行計画を読み解く
  - 読み解いた実行計画から、コストの高い処理の改善方法を検討する
    - 結合順序、方法は変更できるか
    - 効率的な索引を作成できるか
    - SQLの構文を変更できるか
    - ヒント句を利用できるか
- 解決できない場合には、SQLチューニング以外の方法を検討する

Oracle Enterprise Managerの自動チューニング機能を使って、自動化することが可能
Enterprise Edition
- Diagnostics Pack
- Tuning Pack



【補足】Oracle Enterprise Managerを使ったチューニング
問題のあるSQL文の識別

- **Oracle Enterprise Managerの上位SQL**
  - **Enterprise Managerの「パフォーマンス」画面から負荷の高い時間帯をグラフィカルに確認**
  - **負荷の高い時間帯のSQL文を確認**



影の時間帯に問題のあったSQL文が負荷の高順に表示（上位SQL）

【補足】Oracle Enterprise Managerを使ったチューニング
問題のあるSQL文を分析する

- Oracle Enterprise Managerの上位SQL
  - 上位SQLから問題のSQL文と、実行計画を確認

SQLの詳細: 504rs29ywmk85

SQL IDに切替え [ ] 実行 データの表示 [実行時間: 手動リフレッシュ] リフレッシュ SQLワークシート SQLチューニング・アドバイザのスケジュール

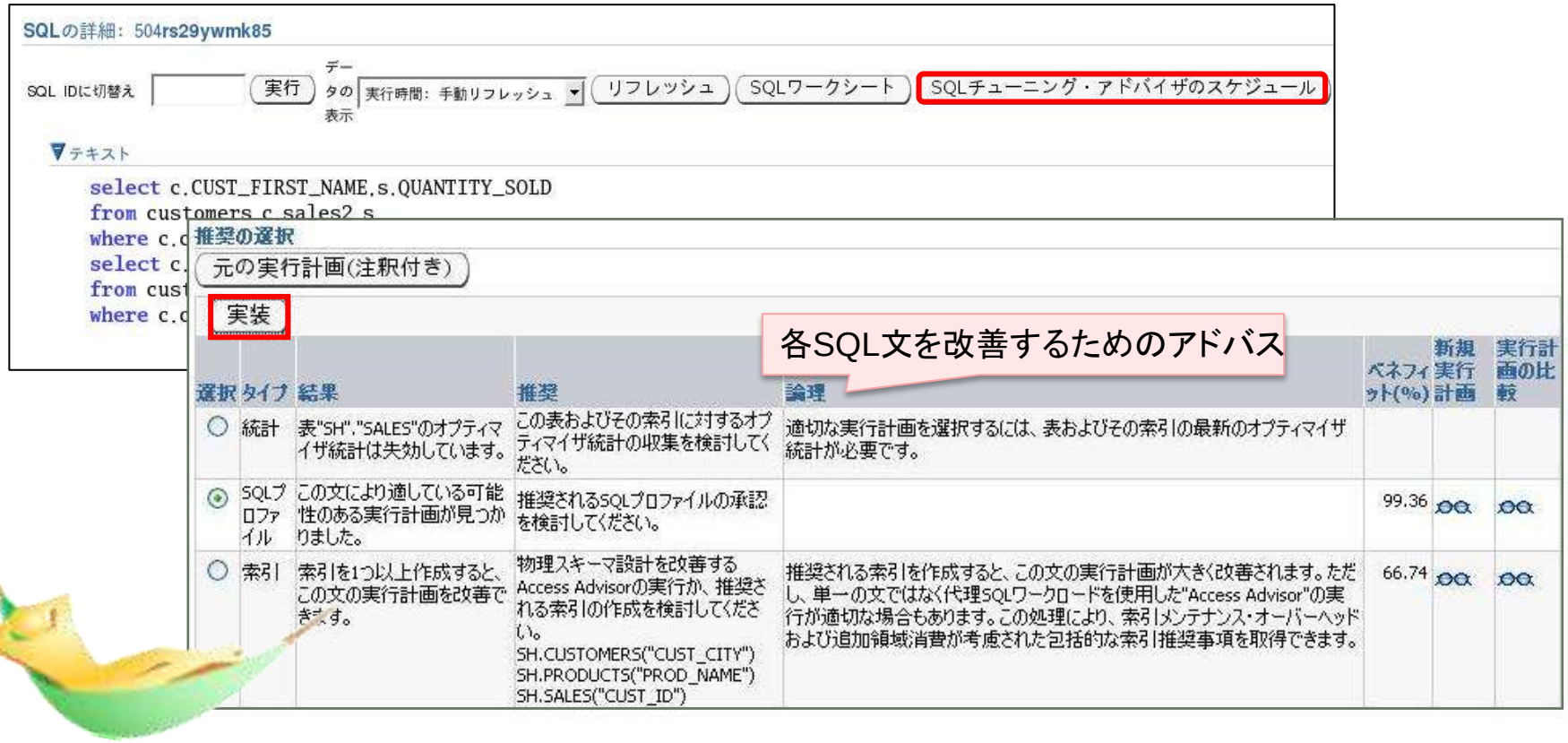
テキスト

```
select c.CUST_FIRST_NAME,s.QUANTITY_SOLD
from customers c,sales2 s
where c.cust_city = 'Tokyo' UN
select c.CUST_FIRST_NAME,s.QU
from customers c,sales2 s
where c.cust_city = 'Paris'
```

操作	オブジェクト	順序	行	バイト	コスト	CPU (%)	時間	ノの開始	ノの停止	問合せブロック名/オブジェクトの別名	述語	フィルタ
SELECT STATEMENT		25			3	100						
HASH GROUP BY		24	1	159	3	33	0:0:1			SEL$62199231		
NESTED LOOPS SEMI		23	1	159	2	0	0:0:1					
NESTED LOOPS SEMI		20	1	144	1	0	0:0:1					
NESTED LOOPS		17	1	114	0							
NESTED LOOPS		15	1	109	0							
NESTED LOOPS		12	1	79	0							
NESTED LOOPS		7	1	14	0							
NESTED LOOPS		5	1	10	0							
NESTED LOOPS		3	1	7	0							
INDEX UNIQUE SCAN	PROMO_PK	1	1	4	0					SEL$62199231 / PR@SEL$1	"PR"."PROMO_ID"=999	
INDEX UNIQUE SCAN	CHANNELS_PK	2	1	3	0					SEL$62199231 / CH@SEL$1	"CH"."CHANNEL_ID"=3	
INDEX UNIQUE SCAN	CHANNELS_PK	4	1	3	0					SEL$62199231 / CH2@SEL$2	"CH2"."CHANNEL_ID"=3	"CH"."CHANNEL_ID"="CH
INDEX UNIQUE SCAN	PROMO_PK	6	1	4	0					SEL$62199231 / PR2@SEL$5	"PR2"."PROMO_ID"=999	"PR"."PROMO_ID"="PR2"
PARTITION RANGE ALL		11	1	65	0			1	28			
TABLE ACCESS BY LOCAL INDEX	SALES	10	1	65	0			1	28	SEL$62199231 / S@SEL$1		"S"."PROMO_ID"=999

【補足】Oracle Enterprise Managerを使ったチューニング
問題のあるSQL文をチューニングをする

- **Oracle Enterprise Managerの上位SQL**
  - 「SQLチューニング・アドバイザ」でチューニングできる項目を洗い出す
  - ほとんどの項目について、画面から実装することができる

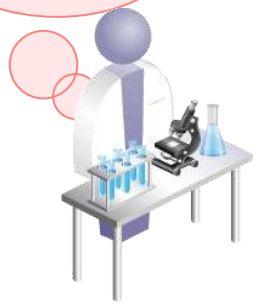


まとめ

- SQLチューニングの流れ
  - 問題のあるSQL文を識別する
  - 前提条件を確認する
    - オプティマイザ統計は適切か
  - 実行計画を読み解く
  - 読み解いた実行計画から、コストの高い処理の改善方法を検討する
    - 結合順序、方法は変更できるか
    - 効率的な索引を作成できるか
    - SQLの構文を変更できるか
    - ヒント句を利用できるか
- 解決できない場合には、SQLチューニング以外の方法を検討する

最適なアクセス・パスや結合方法は
データ量やデータの偏りによって異なるため
「必ず早くなる」という正解はない

アクセス・パスや結合方法を理解したうえで
ケースごとに考えることが重要



OTN×ダイセミ　でスキルアップ!!

・一般的な技術問題解決方法などを知りたい！
・セミナ資料など技術コンテンツがほしい！

一般的技術問題解決にはOTN掲示版の
「データベース一般」をご活用ください

Oracle Technology Network(OTN)
http://otn.oracle.co.jp/forum/index.jspa?categoryID=2

※OTN掲示版は、基本的にOracleユーザー有志からの回答となるため100%回答があるとは限りませんが
過去の履歴を見ると、質問の大多数に関してなんらかの回答が書き込まれております。

過去のセミナ資料、動画コンテンツはOTNの
「OTNセミナー　オンデマンド コンテンツ」へ

OTNセミナー　オンデマンド コンテンツ
http://www.oracle.com/technology/global/jp/ondemand/otn-seminar/index.html

※ダイセミ事務局にダイセミ資料を請求頂いても、お受けできない可能性がございますので予めご了承ください。
ダイセミ資料はOTNコンテンツ　オン　デマンドか、セミナ実施時間内にダウンロード頂くようお願い致します。

OTNセミナー　オンデマンド　コンテンツ

ダイセミで実施された技術コンテンツを動画で配信中!!

ダイセミのライブ感はそのままに、お好きな時間で受講頂けます。

OTN　オンデマンド　検索

※掲載のコンテンツ内容は予告なく変更になる可能性があります。
　期間限定での配信コンテンツも含まれております。お早めにダウンロード頂くことをお勧めいたします。

オラクル クルクルキャンペーン

あのOracle Database Enterprise Editionが超おトク!!

おトクな買い方
オラクル5年分

- ライセンス使用期間　を5年間に設定
- 初期のライセンスコストがなんと67%OFF　！
- テクニカル・サポート価格も53%OFF　！

Oracle Databaseの
ライセンス価格を大幅に抑えて
ご導入いただけます

多くのお客様でサーバー使用期間とされる
5年間にライセンス期間を限定

- 期間途中で永久ライセンスへ差額移行
- 5年後に新規ライセンスを購入し継続利用
- 5年後に新システムへデータを移行

Enterprise Editionはここが違う!!

- 圧倒的なパフォーマンス!
- データベース管理がカンタン!
- データベースを止めなくていい!
- もちろん障害対策も万全!

この機能でこの価格
ライセンスパック

- Oracle Databaseの機能を存分に使える！
- 2ノードRAC構成も可能！
- サーバー構成によって計4種類のパックから選べる！

詳しくはコチラ
http://www.oracle.co.jp/campaign/kurukuru/index.html

Oracle Direct 0120-155-096

お問い合わせフォーム
http://www.oracle.co.jp/inq_pl/INQUIRY/quest?rid=28

ORACLE

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。