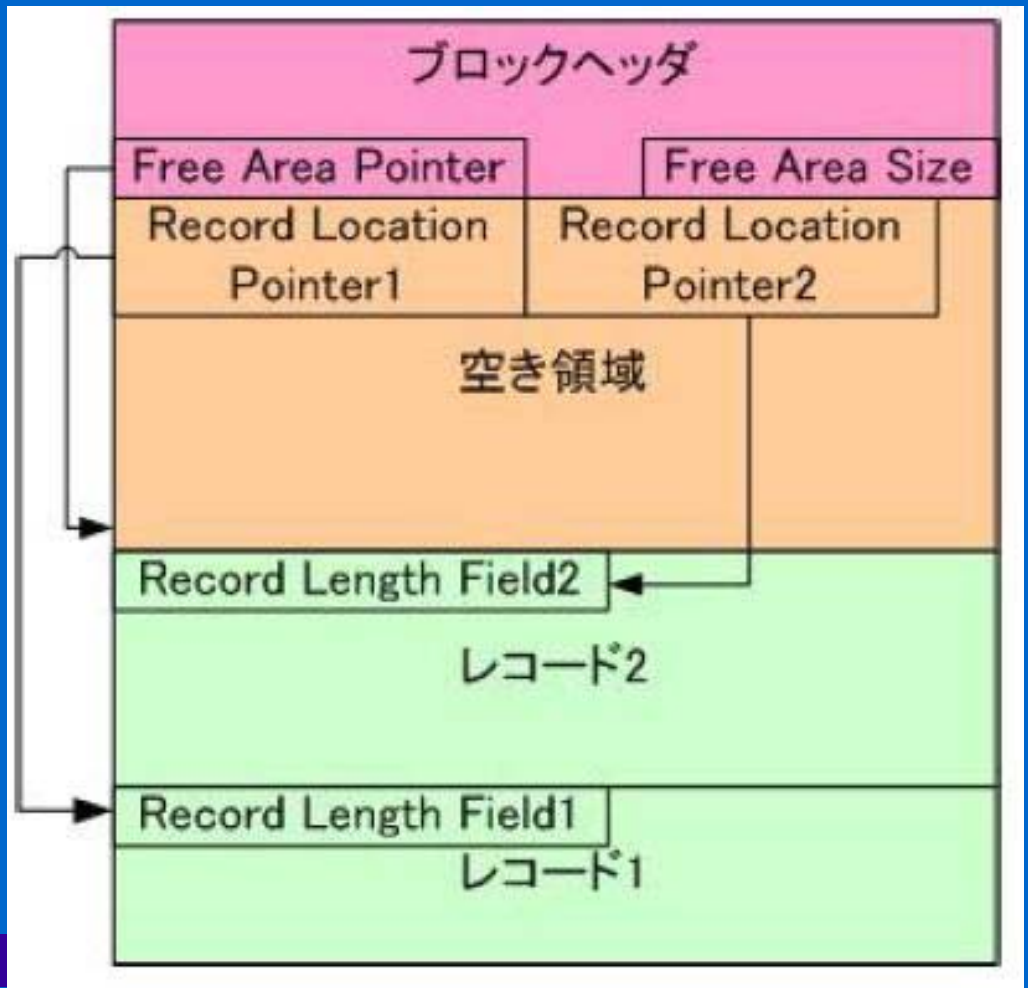
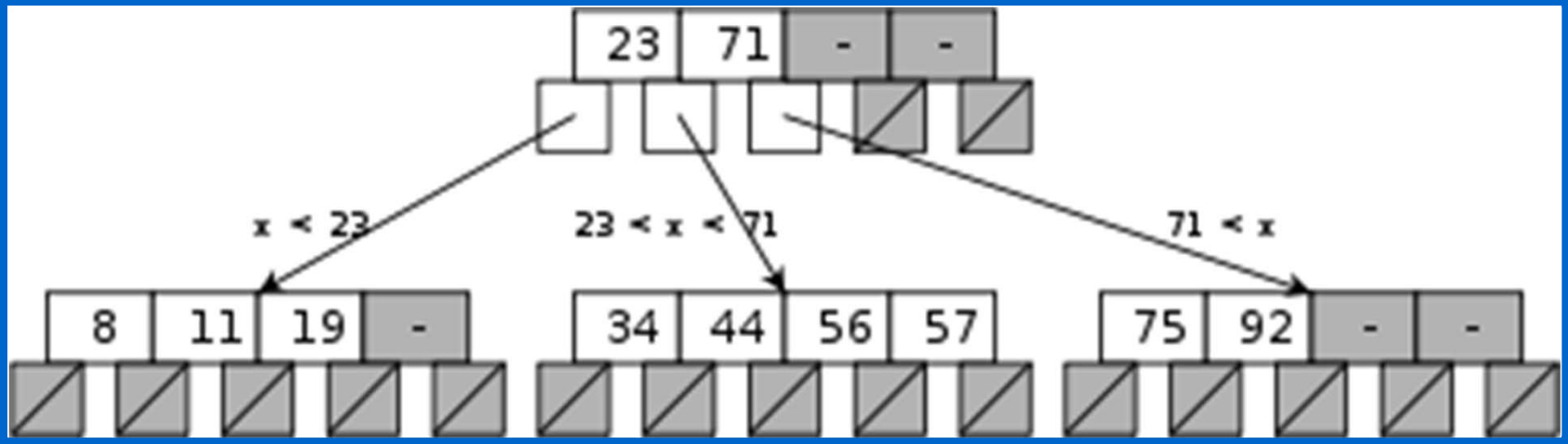
リレーショナルデータモデルの実装

- レコード（タップルの値を入れるデータ領域）をポインタでつないで管理（ユーザにはポインタは見えない）
- 問合せでタップルを検索するときにレコードを探索する方法は？
  → インデックスを作成することで検索を効率化



リレーショナルデータモデルでのインデックスの構造

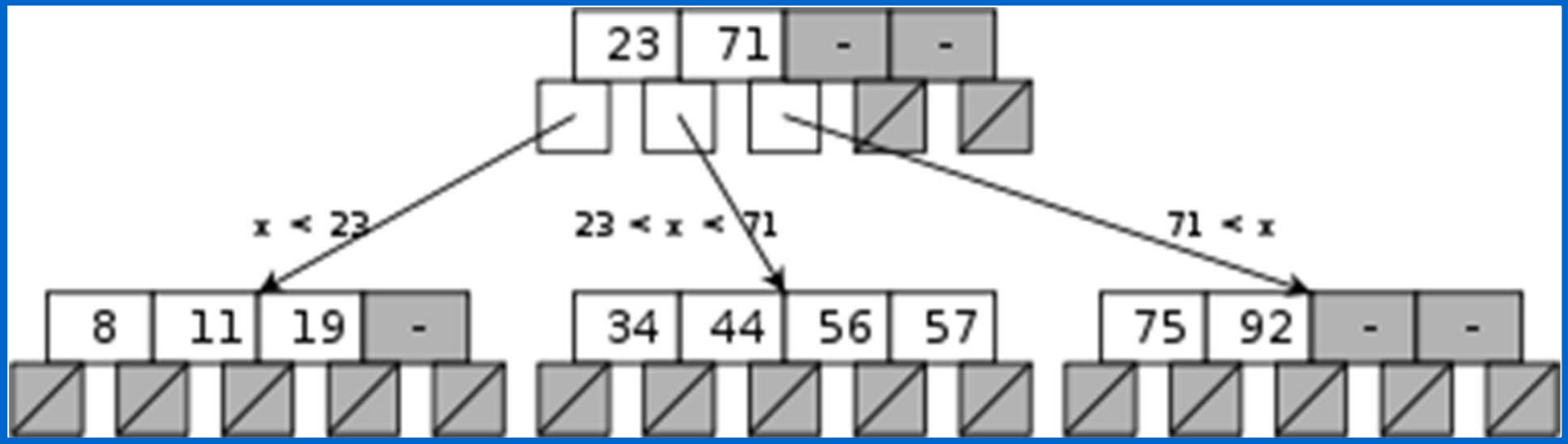
- DBMSの多くは**B木によるインデックス**（正確にはB木から派生したデータ構造）を実装している
- B木の性質
  1. 各ノードは最大でm個のキーを保持する
  2. 根以外のノードは、最小でも$m/2$個のキーを持つ
  3. i個のキーを持つノードは最大$i+1$個の枝を持つ
  4. 葉はすべて同じ深さになる（**平衡木**）



m=4のB木（出典 Wikipedia）

B木の探索

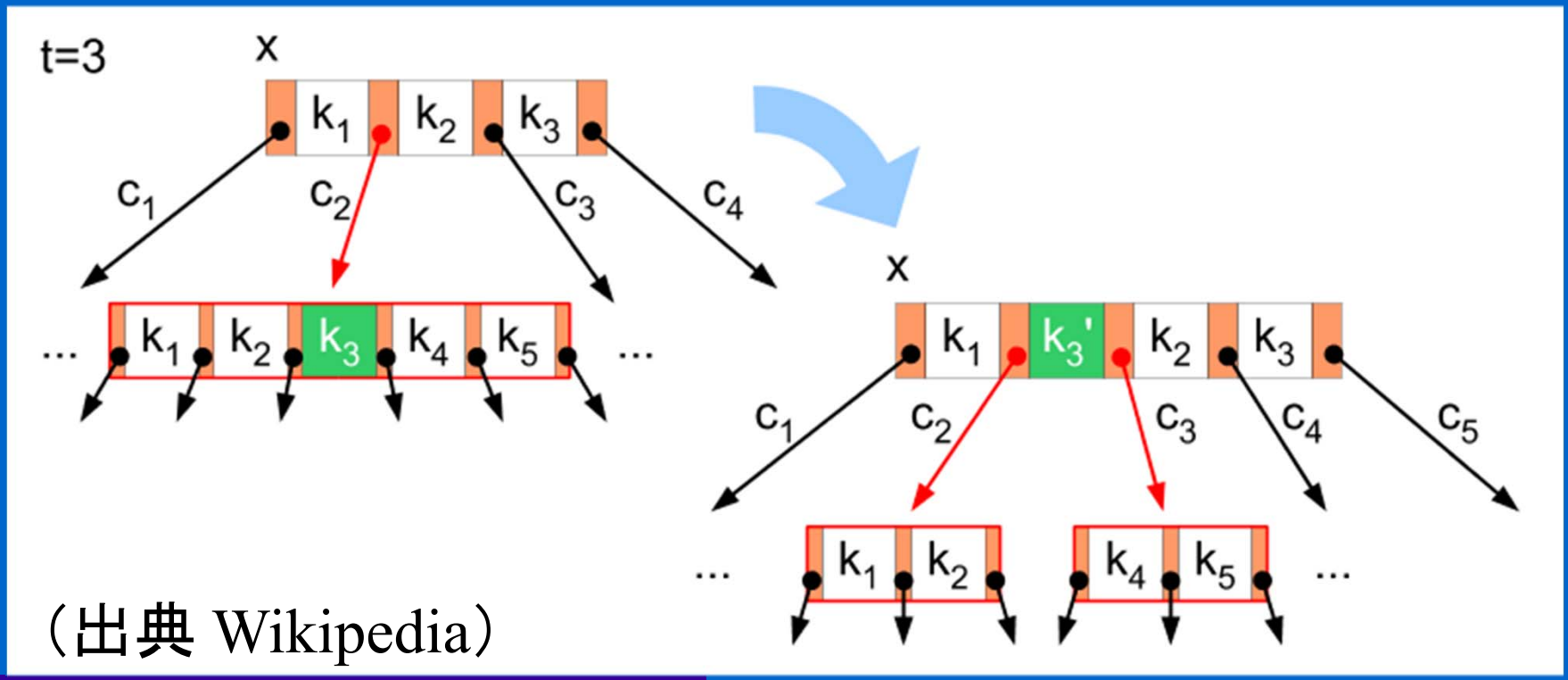
1. 根から探索を開始する
2. 対象となり得るノードが存在しない場合は探索値が木に登録されていないものとして終了
3. $i = 1$
4. ノードにi番目のキーが存在しないか、探索値 $<$ i番目のキー の場合、枝i が指すノードを対象として2へ
5. 探索値 $=$ i番目のキー の場合、探索の成功として終了
6. $i = i + 1$ として4へ



m=4のB木（出典 Wikipedia）

B木へのキーの挿入

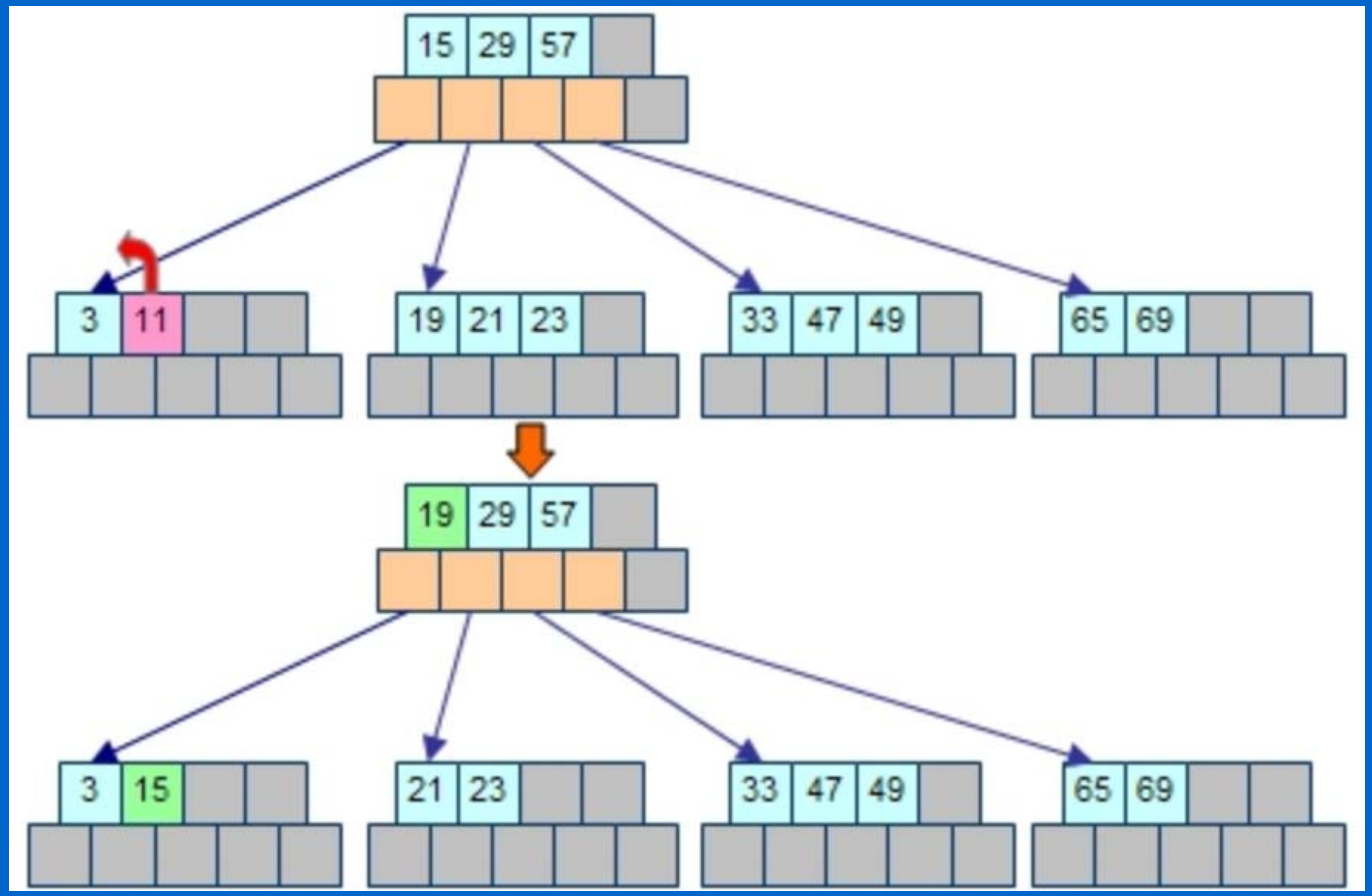
- 挿入すべきキーの位置を、「B木の探索」で求める
- まだ登録されていない値を探索した場合、処理は葉ノードまで達する
- 葉ノードにキーを登録する場所があるときは登録して終了する
- 場所がないときはノードの分割をする
  - 分割が必要なノードから**中央値**のキーを選択し、このキーより小さいキー**m/2**個を含むノードと、より大きいキー**m/2**個を含むノードに分割する
  - 中央値のキーを、親ノードに移動する



（出典 Wikipedia）

B木からのキーの削除

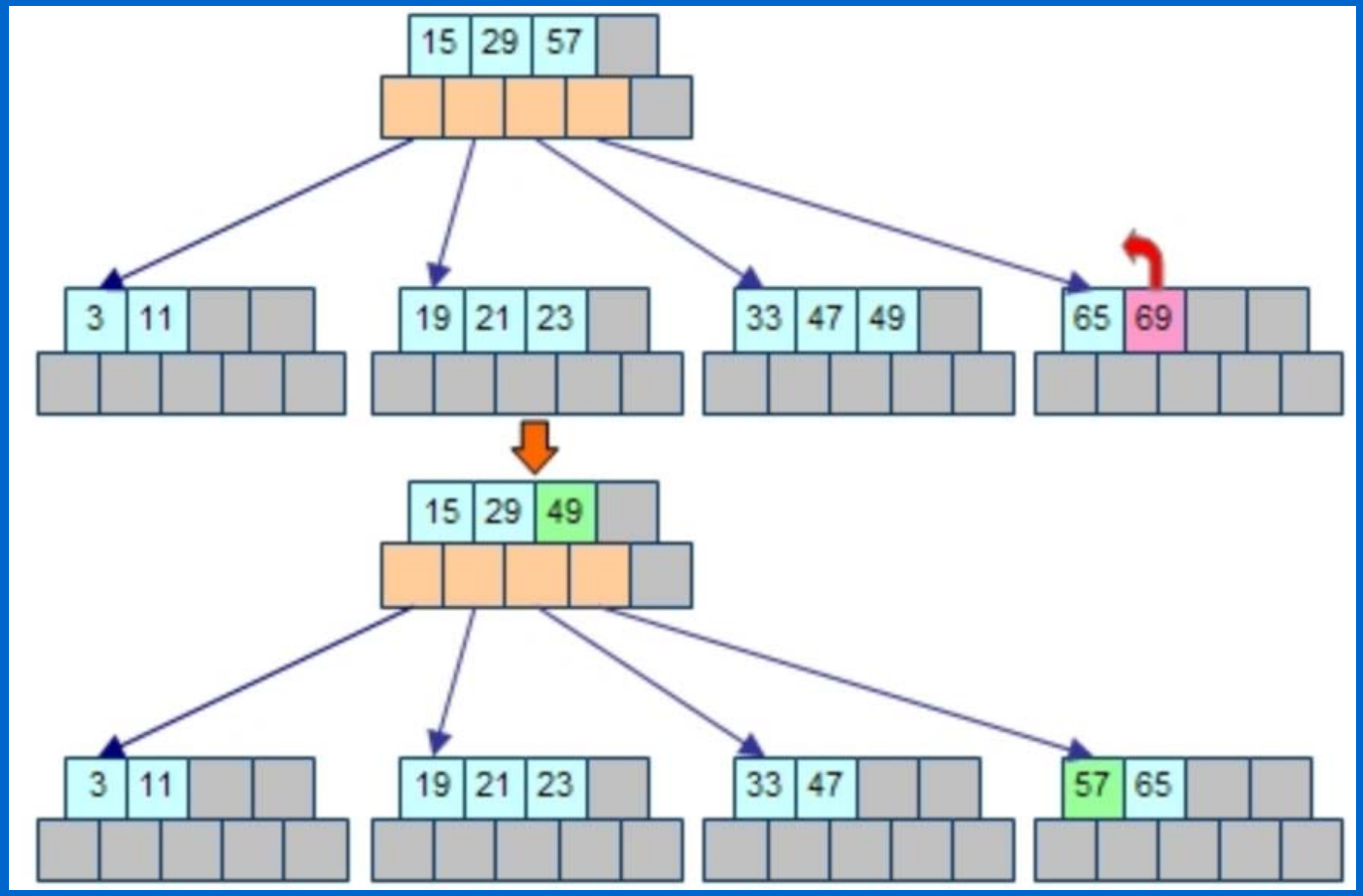
- 該当するキーを持つノードから、そのキーを削除する
- 削除することで、そのノードのキーの個数が**m/2**個より小さくなるときは、**右隣のノード**とキーの調整を行う



（出典 @IT Coding Edge）

B木からのキーの削除(2)

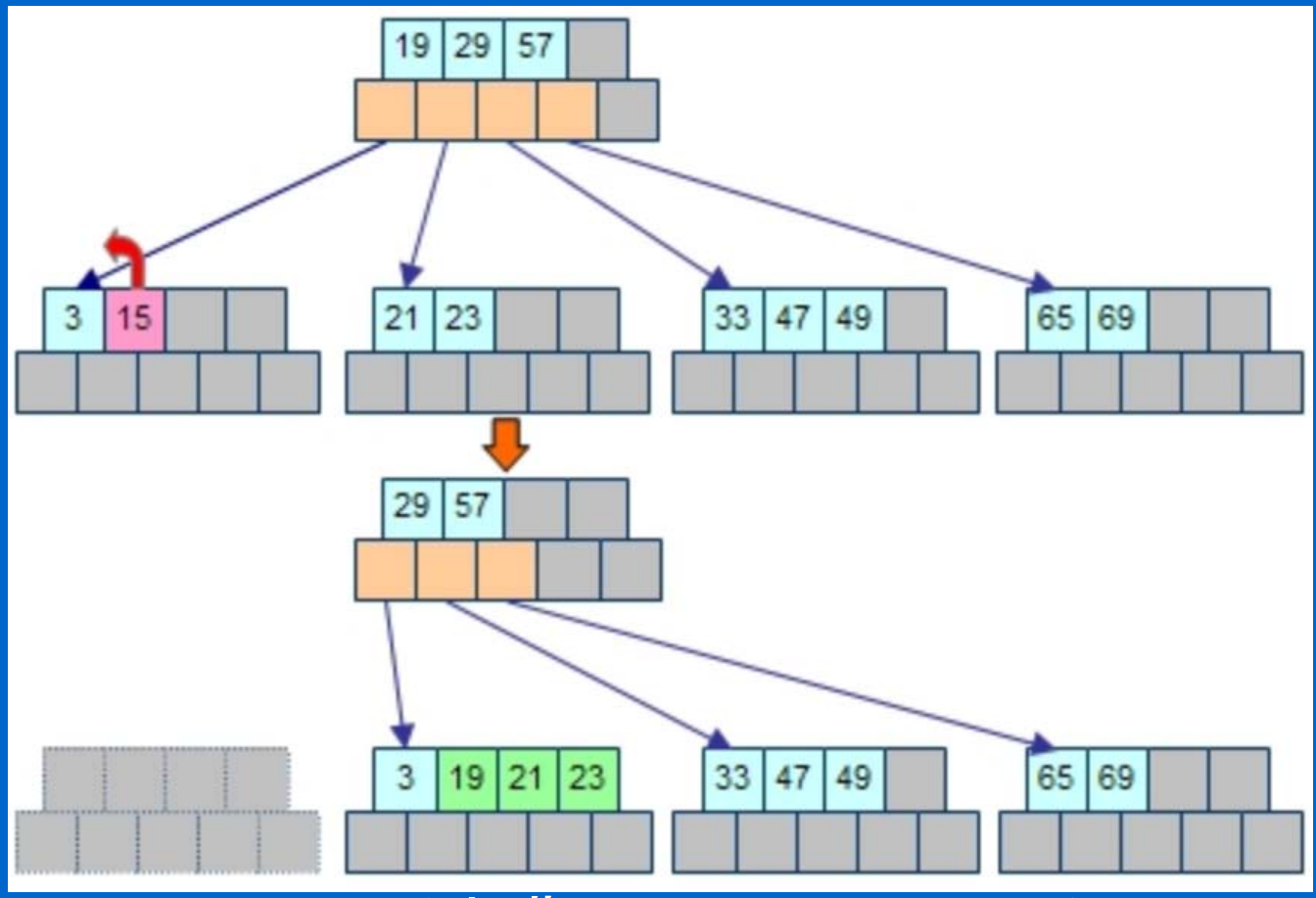
- 該当するキーを持つノードから、そのキーを削除する
- 削除することで、**一番右のノード**のキーの個数が**m/2**個より小さくなるときは、**左隣のノード**とキーの調整を行う



（出典 @IT Coding Edge）

B木からのキーの削除(3)

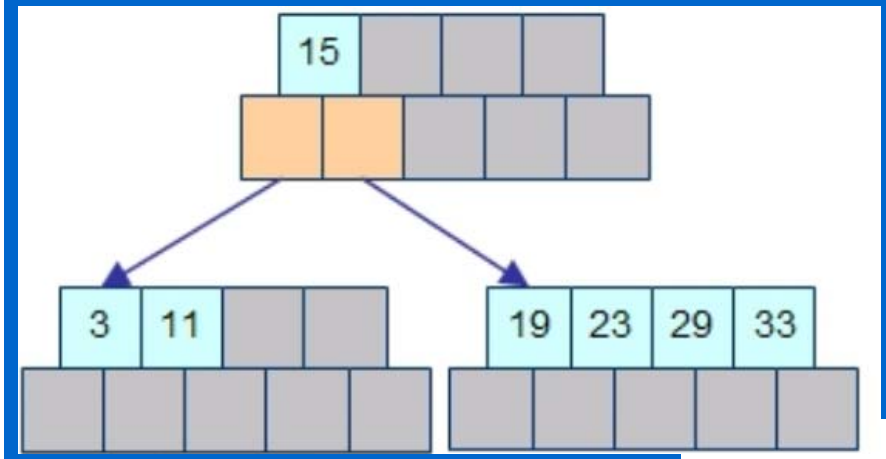
- 該当するキーを持つノードから、そのキーを削除する
- 削除することで、ノードのキーの個数が**m/2**個より小さくて**右隣のノード**と調整すると再度**m/2**個より小さいときは**併合する**



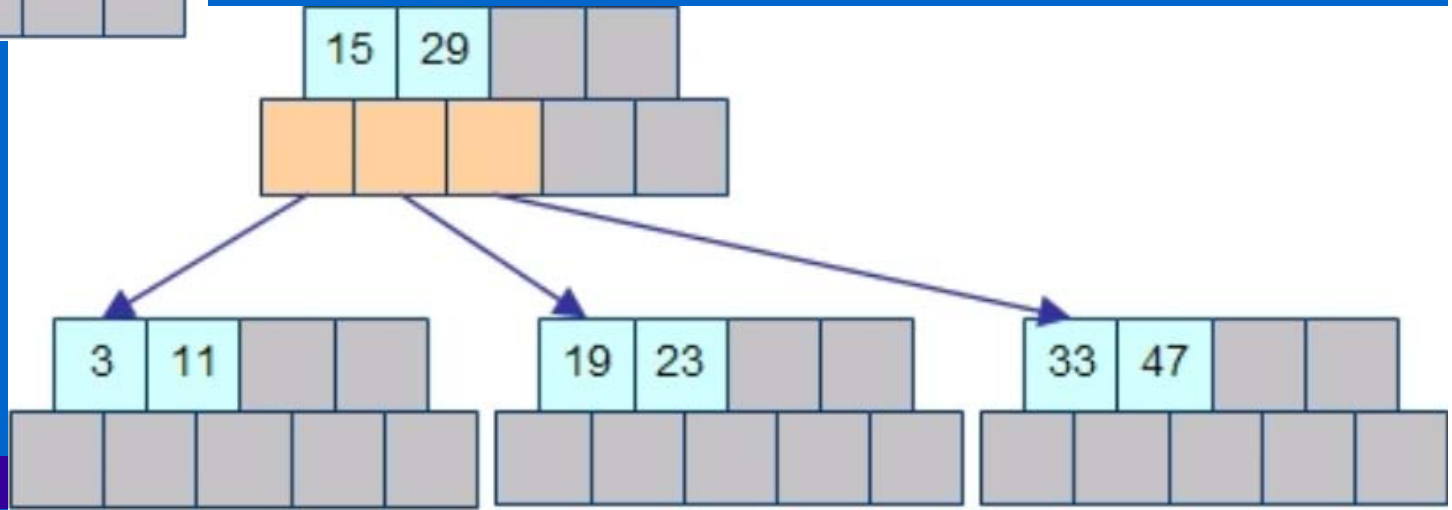
（出典 @IT Coding Edge）

挿入後に深さが平衡に保たれる理由(1)

- ノードが既にキーで埋まっている状態（*m*個ある状態）で新たなキーが挿入されると、ノードの分割が起こる
- ノードの分割では、中央値のキーを親ノードに上げて、親ノードの枝を増やす（単に分割するだけでは**木全体の高さは変わらない**）

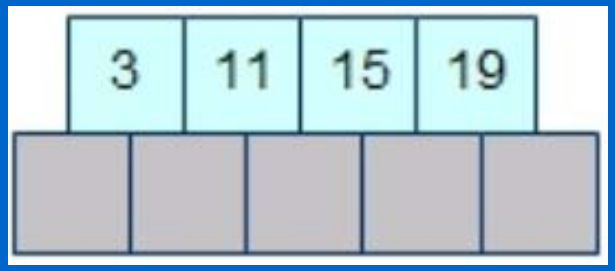


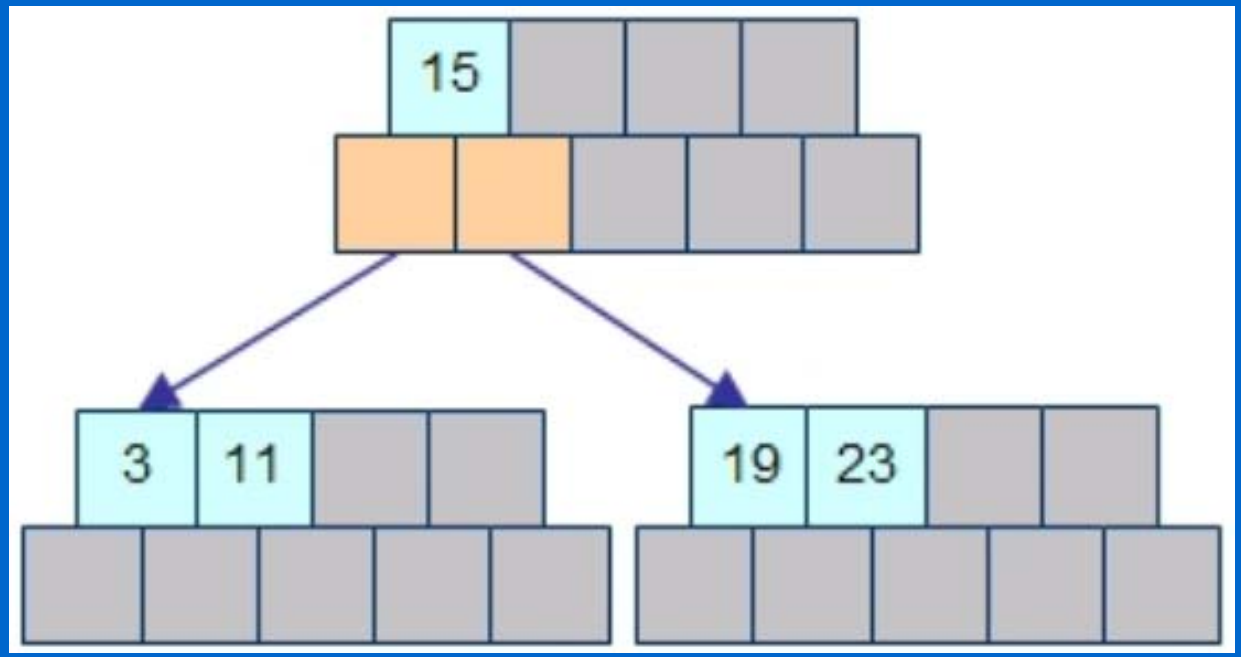
（出典 @IT Coding Edge）



挿入後に深さが平衡に保たれる理由(2)

- **親ノードがキーで埋まっている状態**で、子ノードから中央値が上がってくると、親ノードのそのまた親ノードの枝を増やす（**木の高さは変わらない**）
- 一番上の根が埋まっている状態で子ノードから中央値が上がってくると、根を分割し、その上に新たに根を作成する（**ここで初めて木の高さが一段増えるが、平衡は保たれる**）





（出典 @IT Coding Edge）

削除後に深さが平衡に保たれる理由

- キーの削除のときも、ノードの「分割」を「併合」、キーの数や木の高さの「増やす」を「減らす」に置き換えるだけで同様に平衡に保たれる
  1. 削除することで、そのノードのキーの個数がm/2個より小さくなるときは、隣接するノードとキーを調整する
  2. 調整してもキーの個数が再度m/2個より小さくなるときは、隣接するノードと併合する（親ノードのキーの数が1個減る）
  3. 親ノードのキーの個数がm/2個より小さくなるときは、再度、隣接ノードと調整し必要に応じてノードを併合する
  4. 根ノードのキーの個数がm/2個より小さくなるときは、必要に応じて木の高さを減らす（平衡は保たれる）

B*木

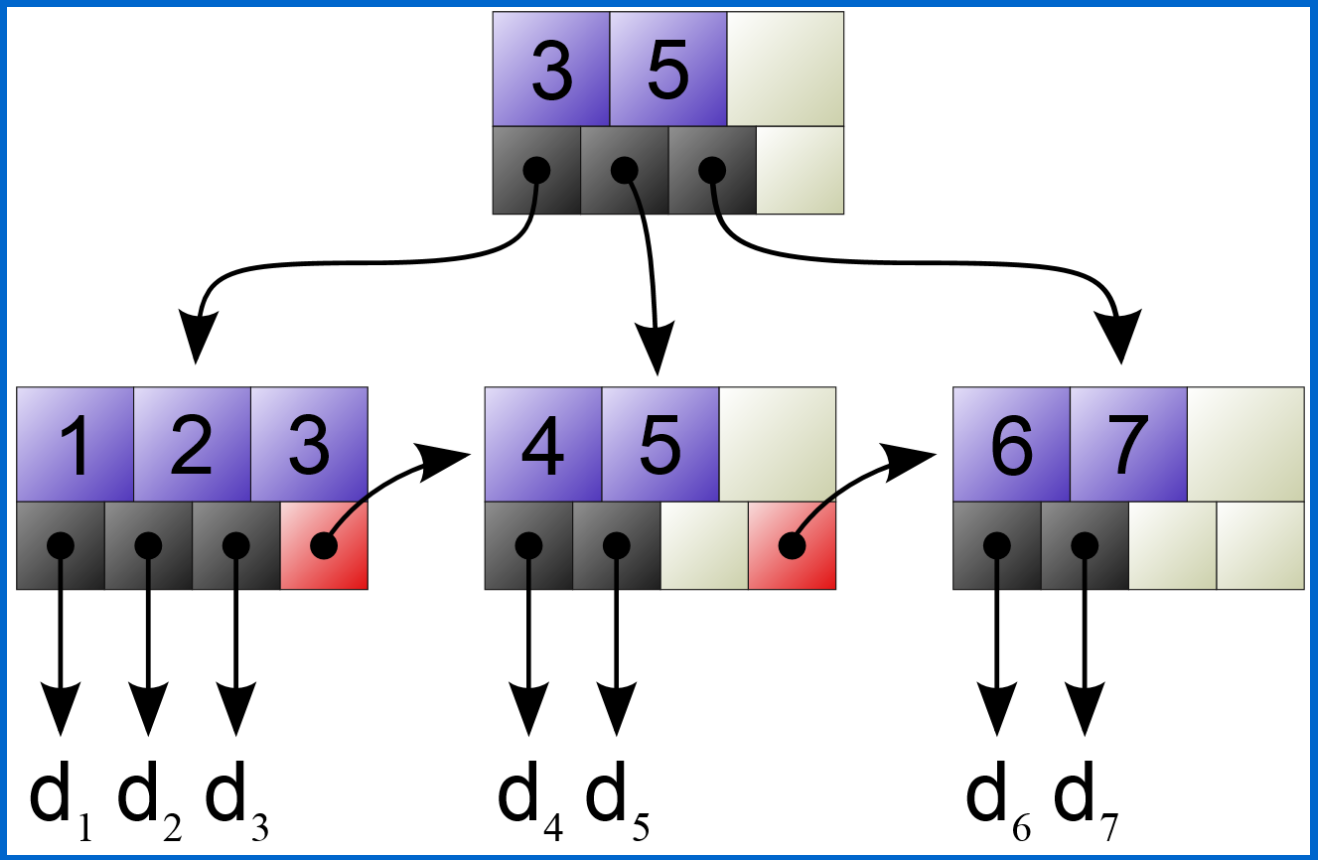
- B木では、ノード内のキーの個数は**m/2**個以上となる

→最悪の場合はノード内の半分しかキーが埋まらない（空きが半分できてしまい**使用効率がよくない**）

- 解決策：B*木
  - ノード内のキーの個数を**2m/3**個以上に保つ（**B木より使用効率がよい**）
  - ノードがいっぱいになったとき即座に分割するのではなくキーを次のノードと共有する
  - 連続する2つのノードがいっぱいになると、それを3つのノードに分割する
  - 常に左端のキーは使わずに残しておく

B+木

- **レコードを指すキー**は木の最下層（葉ノード）に格納する
- 葉ノード間をキーの値の昇順にポインタでつなぐ
- 葉でないノード（内部ノード）には**探索のためのキー**を格納する
- B+木は、特にブロック型記憶装置でのデータ検索の効率化に効果を発揮する(**多くのRDBMSではB+木でインデックスを構成する**)

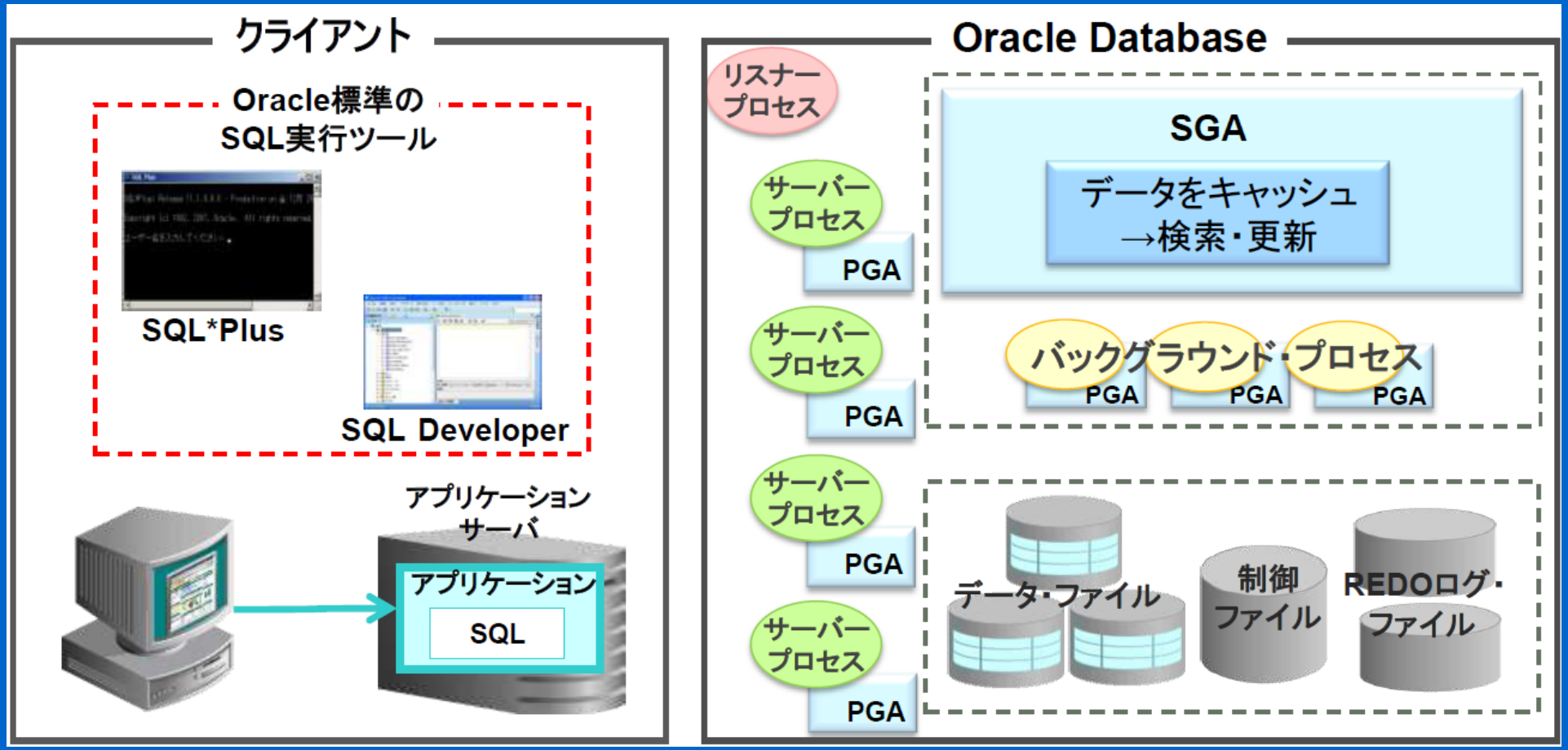


出典 Wikipedia

データベース管理システムの構成

- データベース管理システムは次の構成要素を基にして実装される（Oracleでの実装例）
  - ➢データベースファイル
    - データファイル、ログファイル、制御ファイル
  - ➢共有メモリ
    - データベースバッファキャッシュ、ログバッファ
  - ➢データベースプロセス
    - サーバプロセス、データベースライタープロセス、ログライタープロセス

データベース管理システムの構成(2)



出典　今さら聞けないOracle入門!? アーキテクチャ編
http://www.oracle.com/technetwork/jp/ondemand/db-basic/0420-1330-oracle-architecture-366291-ja.pdf

データファイル

- データファイルは、表の内容を保持しているファイルである
- 実運用されているデータベースシステムのデータ量は増加の一途をたどっており、データ量が何テラバイトにも及ぶデータベースも珍しくなくなっている
- このため、1個のデータベースの内容を複数のデータファイルに分けて管理する機能が必要となる
- データファイル中のデータには、表のデータ以外にインデックスのデータも含まれる

ログファイル

- ログファイルは、主にデータベースの変更履歴を格納するファイルである
- ログファイルは単に変更履歴の格納だけでなく、障害が発生してしまった場合の復旧に必要となる
- 障害でデータが失われてしまったとしても、ログファイルにある履歴を基に処理を再実行することでデータを復旧させることが可能となる

制御ファイル

- 制御ファイルには、データベースを構成するファイルの名前や格納されている場所、および各ファイルの現在の状態など、データベースに関する管理情報が格納されている
- データベース管理システムは、データベース起動時にこの制御ファイルを読み込むことで、データファイルやログファイルの場所や状態を把握する

データベースバッファキャッシュ

- データベースの表操作で使われる
- データベースへの問合せ時は、このキャッシュに対象の表データがあるかどうかを調べ、あればそれを検索し、なければデータファイルから表データを読み込んでキャッシュに格納した後、問合せ操作を実行する

ログバッファ

- ログファイルの読み書きで使われるメモリ領域である
- データベースの変更履歴などログファイルに書き込むデータは、いったんログバッファにためておき、必要なタイミングでまとめてログファイルに書き込むことでディスクへのアクセス回数を軽減している

サーバプロセス

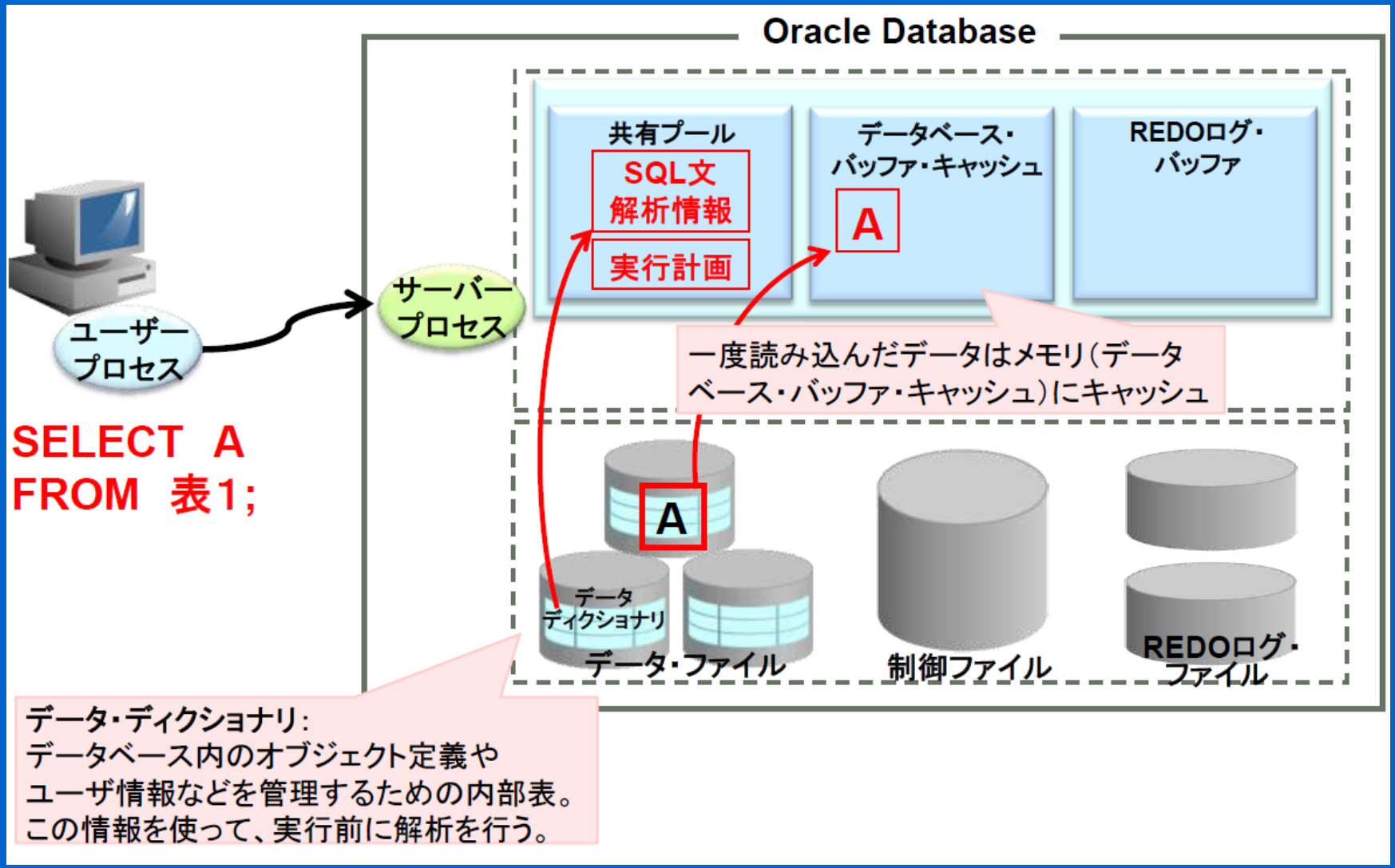
- データベースに対する問合せや更新の操作を担当する
- 一つのサーバプロセスが全部の操作を行うのではなく、複数のプロセスで処理を分担するのが通常である
- 操作の処理では、まずデータベースバッファキャッシュにアクセスし、キャッシュになければデータファイルから読み出す

データベースライタープロセス

ログライタープロセス

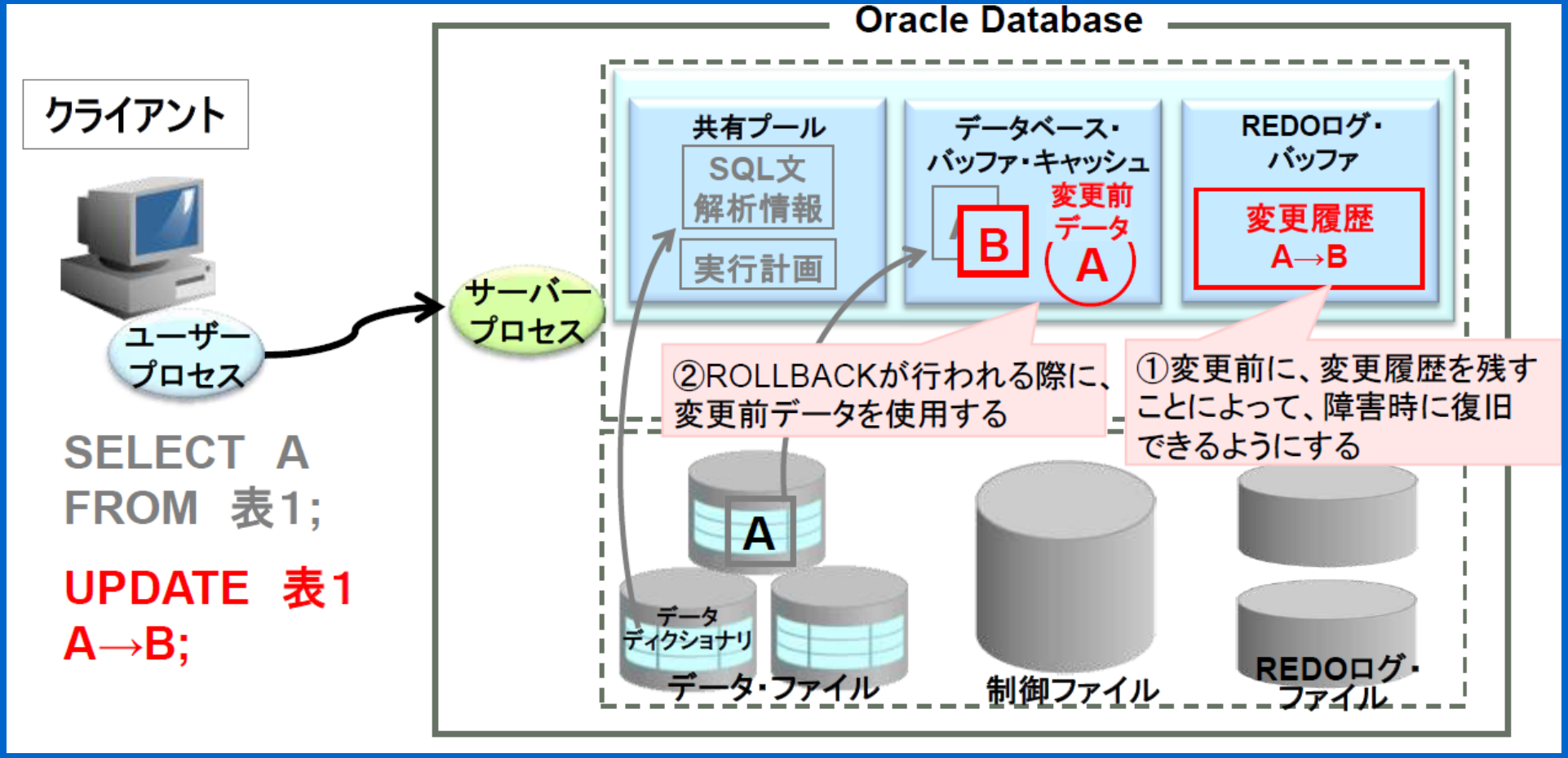
- データベースライタープロセスは、データファイルに書き込まれたデータベースの更新データをデータファイルに書き込む処理を担当する
- ログライタープロセスは、ログバッファに書き込まれたデータベースの変更履歴等の情報をログファイルに書き込む処理を担当する

問合せの処理



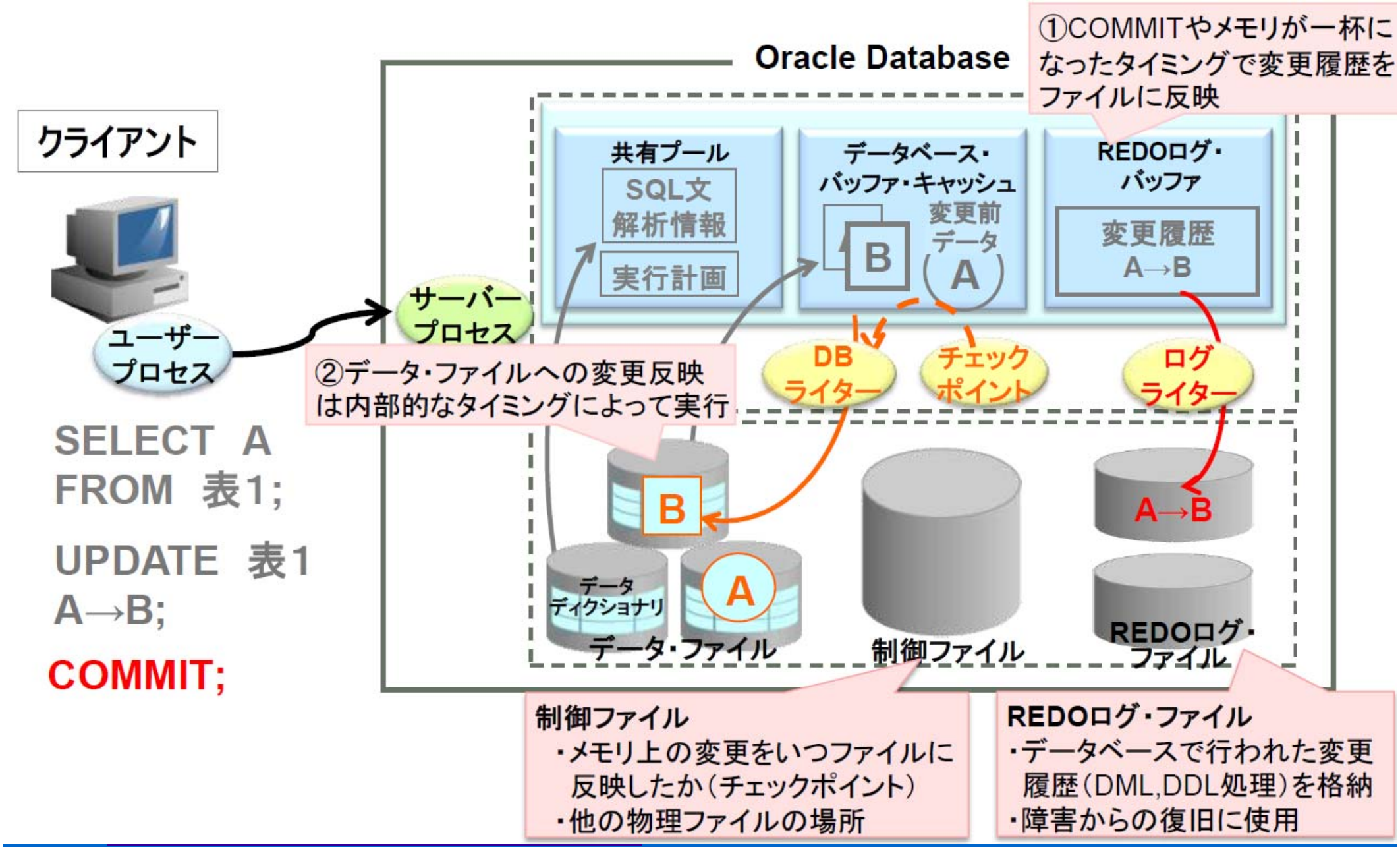
出典　今さら聞けないOracle入門!? アーキテクチャ編

更新の処理



出典　今さら聞けないOracle入門!? アーキテクチャ編

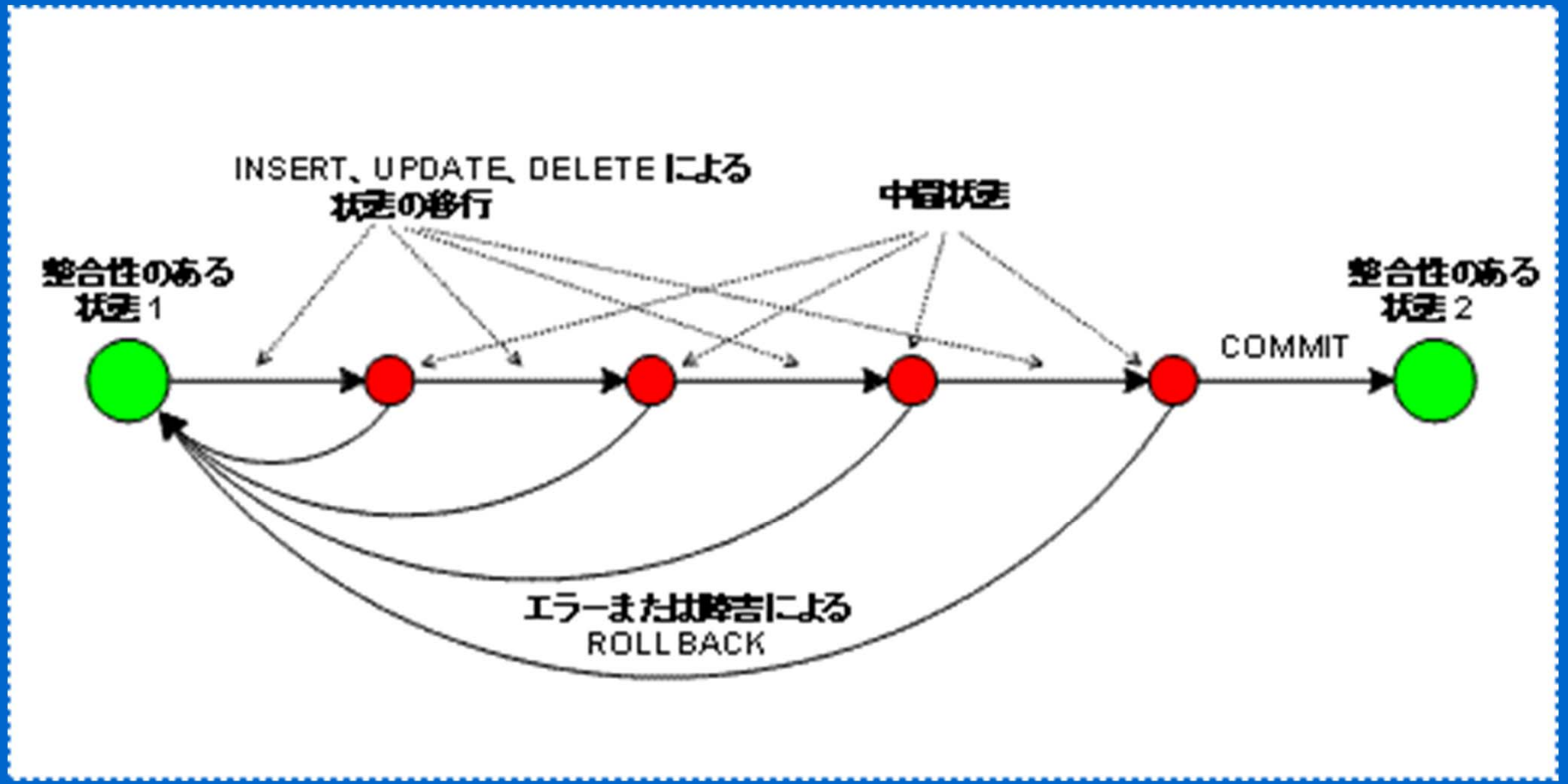
コミットの処理


クライアント
ユーザープロセス
SELECT A
FROM 表1;
UPDATE 表1
A→B;
COMMIT;
サーバープロセス
②データ・ファイルへの変更反映は内部的なタイミングによって実行
Oracle Database
①COMMITやメモリが一杯になったタイミングで変更履歴をファイルに反映
共有プール
SQL文
解析情報
実行計画
データベース・バッファ・キャッシュ
B
変更前データ
A
REDOログ・バッファ
変更履歴
A→B
DBライター
チェックポイント
ログライター
B
データディクショナリ
A
データ・ファイル
制御ファイル
A→B
REDOログ・ファイル
制御ファイル
・メモリ上の変更をいつファイルに反映したか（チェックポイント）
・他の物理ファイルの場所
REDOログ・ファイル
・データベースで行われた変更履歴（DML,DDL処理）を格納
・障害からの復旧に使用

トランザクション

- データベース内のデータに対する一連の「論理的」操作群のことを**トランザクション**と呼ぶ
- このトランザクションという概念は障害に対する復旧操作とも関係しており、どの操作からどの操作までが1個のトランザクションかは、データベース管理システムが自動的に知ることはできない
- このため、「ここからここまでが1個のトランザクションである」ということは、**ユーザが明示的に指定する**必要がある

トランザクションとは？



出典　https://help.sap.com/

トランザクションが満たすべき性質

以下の4つの性質があり、頭文字を取って**ACID特性**と呼ばれることがある

- 原子性（atomicity, 不可分性とも呼ばれる）
- 整合性（consistency, 一貫性とも呼ばれるがintegrityと区別するためここでは整合性と呼ぶ）
- 独立性（isolation）
- 永続性（durability）

原子性

- オペレーティングシステムにおける臨界領域と類似の概念
- トランザクションに含まれる個々の操作は不可分であり、各操作がすべて実行されるか、あるいは全く実行されないことを保証する性質をいう
- データベースの**障害からの回復**で、この原子性を満たすためには**永続性が必要**となる

整合性

- 整合性(consistency)は、次回で述べる一貫性(integrity)と基本的には同じ性質である
  - 「一貫性」という場合は、データの持つ性質（実世界で成り立っている性質）を基に、**データベースの設計時**に常に成り立つ性質として定義される
  - 「整合性」は、**トランザクションの開始時と終了時に満たされていることが要求される**性質である
  - 逆に言えば、トランザクションの処理中は処理の効率化のために満たさない場合が想定されている
  - **同時実行制御**（複数のトランザクションを並行に実行する制御）では整合性を保つ必要ある

独立性

- トランザクション中に行われる操作の過程がその**トランザクションの外から隠蔽される**ことを指す
- トランザクションの外部からは、トランザクションの開始時または終了時のデータベースの内容しか参照できない（つまり、トランザクション実行中の途中のデータベースの内容は参照できない）ことを保証する
- **同時実行制御**で整合性を保つための方針として、**直列化可能性（serializability**、逐次に実行したときと同じ結果となること）があるが、これには**独立性が必要**となる。

永続性

- トランザクション操作の完了通知をユーザが受けた時点で、その操作は永続的となり、結果が失われないことを指す
- 前述のログファイルの機能により、トランザクションが完了できない場合はトランザクションの開始時の内容、完了されたときには終了時の内容となり、障害等によりトランザクション実行過程の途中の状態のまま保持されることがないことを保証する