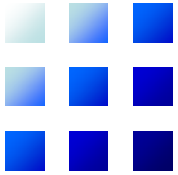
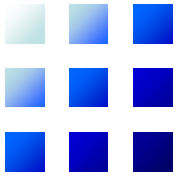


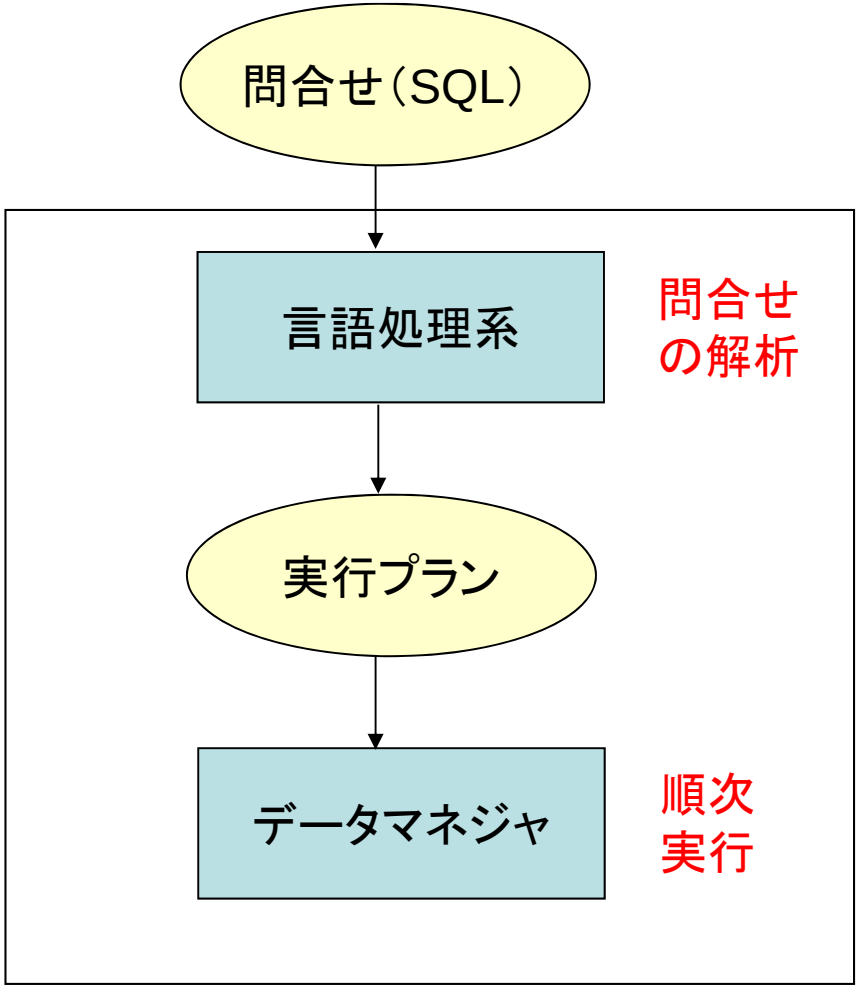
データベース
【11:問合せ処理】

石川　佳治



問合せ処理と最適化

リレーショナルDBMSにおける問合せ処理(1)



- 問合せ処理のステップ
  - 問合せ(通常SQLで記述)を言語処理系が解析
  - 実行プラン(アクセスプラン)を生成
    - アクセスステップ(基本データ操作)の集まり
  - データマネジャにより実行

リレーショナルDBMSにおける問合せ処理(2)

- 論理的かつ非手続き問合せ記述
  ⇒ 物理的かつ手続き的問合せ記述
  という変換を実現
- 主要なアプローチの一つ:2フェーズに分ける
  1. SQL問合せ ⇒ リレーショナル代数式への変換
     - 等価なリレーショナル代数式を複数生成
  2. リレーショナル代数による記述
     ⇒ 具体的なアクセスステップの列への変換
     - ファイル編成，索引，ファイル中のレコードの並び順など，さまざまな項目を考慮

問合せ処理の例(1)

- SQL問合せ

```
SELECT 科目.科目番号, 科目名, 成績
FROM   科目, 履修
WHERE 科目.科目番号 = 履修.科目番号 AND 学籍番号 = '00100'
```

- 第一フェーズの実行プランの例

① $\pi_{\text{科目.科目番号, 科目名, 成績}}(\sigma_{\text{科目.科目番号 = 履修.科目番号}\wedge\text{学籍番号='00100'}}(\text{科目}\times\text{履修}))$

② $\pi_{\text{科目番号, 科目名, 成績}}(\sigma_{\text{学籍番号='00100'}}(\text{科目}\bowtie_{\text{科目.科目番号=履修.科目番号}}\text{履修}))$

③ $\pi_{\text{科目番号, 科目名, 成績}}(\text{科目}\bowtie_{\text{科目.科目番号=履修.科目番号}}(\sigma_{\text{学籍番号='00100'}}\text{履修}))$

問合せ処理の例(2)

- 実行プラン①～③は等価であるが，実行コスト(実行時間，必要作業領域)は大きく異なる
- 具体例
  - リレーション「科目」:1万タプル
  - リレーション「履修」:100万タプル
  - 両者の結合結果:100万タプル
  - 学籍番号00100の学生の履修登録数:50科目

問合せ処理の例(3)

- 実行プラン①:処理の見積もり

科目:
1万タプル

科目番号	科目名	単位数

科目番号	学籍番号	成績

履修:
100万タプル

大量の中間結果:
・多くの記憶領域が必要
・選択演算にも多くの実行時間が必要

× 直積

直積結果
100億タプル

科目.科目番号	科目名	単位数	履修.科目番号	学籍番号	成績

選択処理:
100億タプル
を見る

科目.科目番号	科目名	単位数	履修.科目番号	学籍番号	成績

50タプルに減少

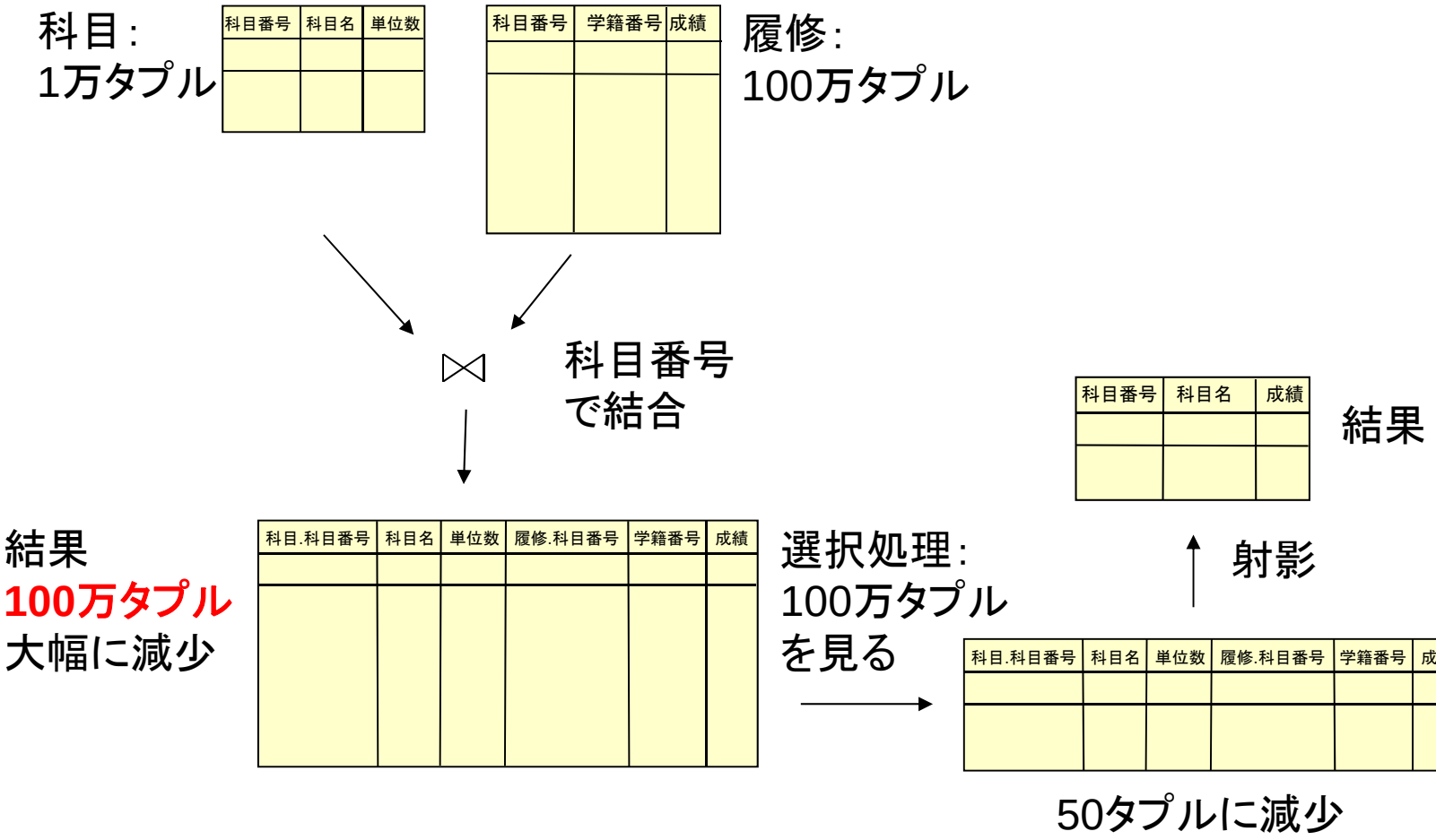
射影

科目番号	科目名	成績

結果

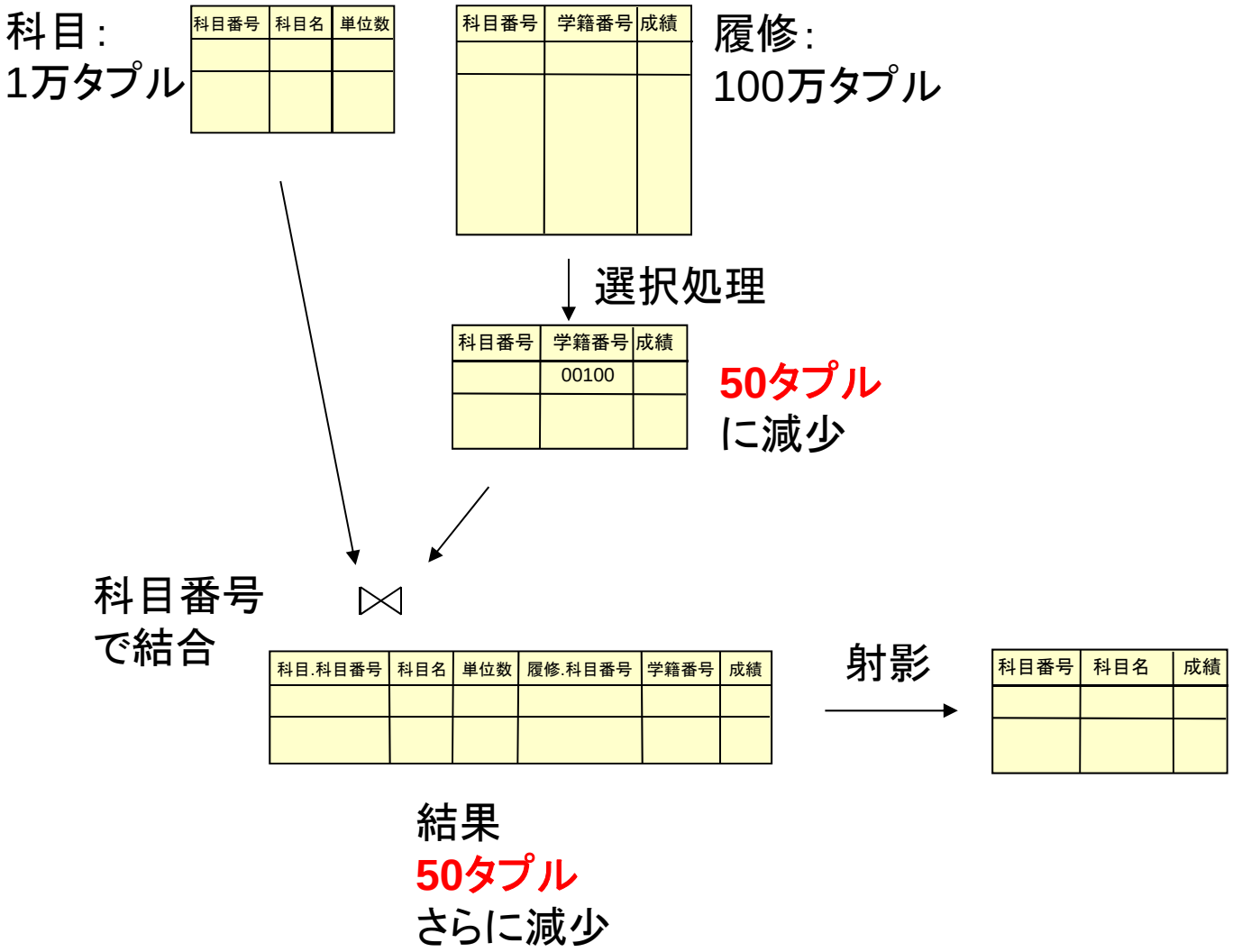
問合せ処理の例(4)

- 実行プラン②



問合せ処理の例(5)

- 実行プラン③

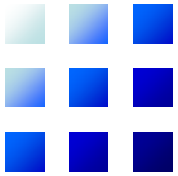


問合せ処理の例(6)

- 実行プラン③が最も効率的
  - 早い時点でタプルを絞り込む
  - 実際には第2フェーズでどのように処理するかも関係
- 基本的な考え方
  - 問合せ結果に含まれないタプルをできるだけ早く除去

問合せ最適化（query optimization）

- 実行コストの小さい実行プランを選択する処理
- 主要なコスト要因：ディスクページアクセス回数
  - データベース問合せ処理では支配的な影響
  - データの多くが二次記憶に存在，二次記憶は低速
- 最小コストの実行プランの選択は困難
  - 多数の候補
  - データベースの内容，システム構成などに依存
  - 一定範囲の合理的な実行コストのプラン選択を目指す
  - 必ずしも最良の実行プランとは限らない
- 2つのアプローチ
  - 経験的選択基準による手法
  - 定量的なコスト見積もりによる手法



リレーショナル代数式を対象とした問合せ最適化

アプローチの概略

- 与えられたリレーショナル代数式のうち，より効率的と予想されるものを経験的選択基準により選択
- リレーショナル代数式を段階的に変換
  - リレーショナル代数式の等価な変換を行うための変換規則を利用

変換規則(1)

① 選択の分解・融合

$$\sigma_{F_1 \wedge F_2}(R) = \sigma_{F_1}(\sigma_{F_2}(R))$$

② 選択の交換

$$\sigma_{F_1}(\sigma_{F_2}(R)) = \sigma_{F_2}(\sigma_{F_1}(R))$$

③ 射影の分解・融合 : $\{A_1,\ldots,A_n\} \subseteq \{B_1, \ldots, B_m\}$ のとき

$$\pi_{A_1,\ldots,A_n}(\pi_{B_1,\ldots,B_m}(R)) = \pi_{A_1,\ldots,A_n}(R)$$

変換規則(2)

④ 選択と射影の交換:選択条件 F が参照する属性がすべて $A_1, \ldots, A_n$ に含まれるとき

$$\pi_{A_1,\ldots,A_n}(\sigma_F(R)) = \sigma_F(\pi_{A_1,\ldots,A_n}(R))$$

⑤ 選択と直積，結合の交換:選択条件 F が R_1 の属性のみを参照するとき

$$\sigma_F(R_1 \times R_2) = \sigma_F(R_1) \times R_2$$

$$\sigma_F(R_1 \bowtie_{F'} R_2) = \sigma_F(R_1) \bowtie_{F'} R_2$$

変換規則(3)

⑥ 選択と和，差，共通部分の交換

$$\sigma_F(R_1 \cup R_2) = \sigma_F(R_1) \cup \sigma_F(R_2)$$
$$\sigma_F(R_1 - R_2) = \sigma_F(R_1) - \sigma_F(R_2)$$
$$\sigma_F(R_1 \cap R_2) = \sigma_F(R_1) \cap \sigma_F(R_2)$$

⑧ 射影と和の交換

$$\pi_{A_1,\ldots,A_n}(R_1 \cup R_2) = \pi_{A_1,\ldots,A_n}(R_1) \cup \pi_{A_1,\ldots,A_n}(R_2)$$

変換規則(4)

⑦ 射影と直積，結合の交換：属性 $A_1, \ldots, A_n$ のうち $B_1, \ldots, B_m$ が R_1 の属性，$C_1, \ldots, C_k$ が R_2 の属性であり，また，結合条件 F が参照する属性はすべて $A_1, \ldots, A_n$ に含まれるとき

$$\pi_{A_1,\ldots,A_n}(R_1 \times R_2) = \pi_{B_1,\ldots,B_m}(R_1) \times \pi_{C_1,\ldots,C_k}(R_2)$$

$$\pi_{A_1,\ldots,A_n}(R_1 \bowtie_F R_2) = \pi_{B_1,\ldots,B_m}(R_1) \bowtie_F \pi_{C_1,\ldots,C_k}(R_2)$$

変換処理の概略(1):一般的指針

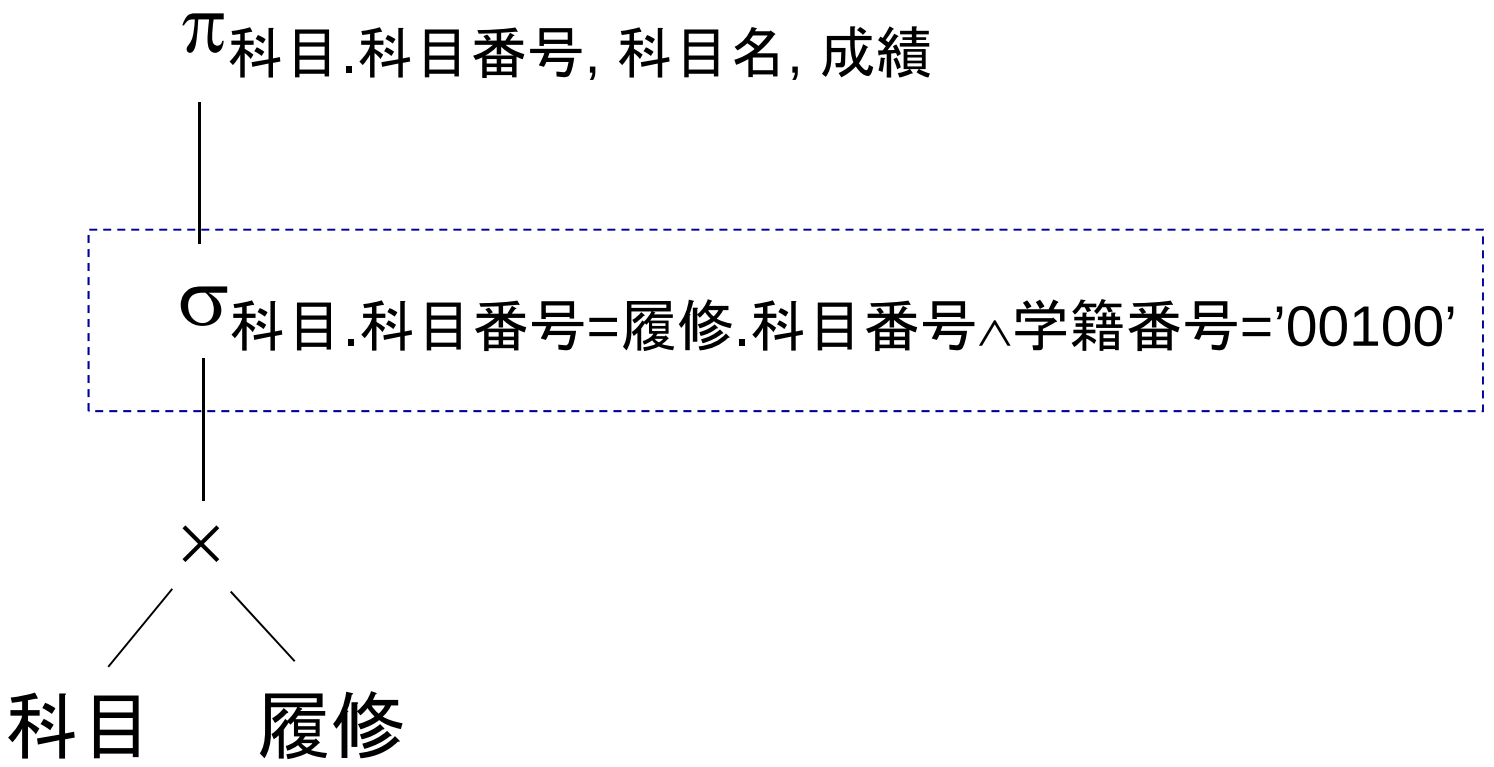
- 中間結果のデータ量を削減
  - ① 選択をできるだけ早く適用:問合せ結果に関与しないタプルを除去
  - ② 射影による不要属性の削除をできるだけ早く実行
  - ③ 直積とその直後の選択を(可能ならば)結合にまとめる

変換処理の概略(2):変換規則の適用手順

1. 規則①を適用:選択条件に論理積を含む選択を複数の選択に分解
2. 規則②④⑤⑥を適用:選択を可能な限り先に実行
3. 直積とそれに続く選択を結合にまとめる
4. 規則③④⑦⑧を適用:射影を可能な限り先に実行
5. 規則①③④を適用:連続した選択・射影を単一の選択・単一の射影に変換

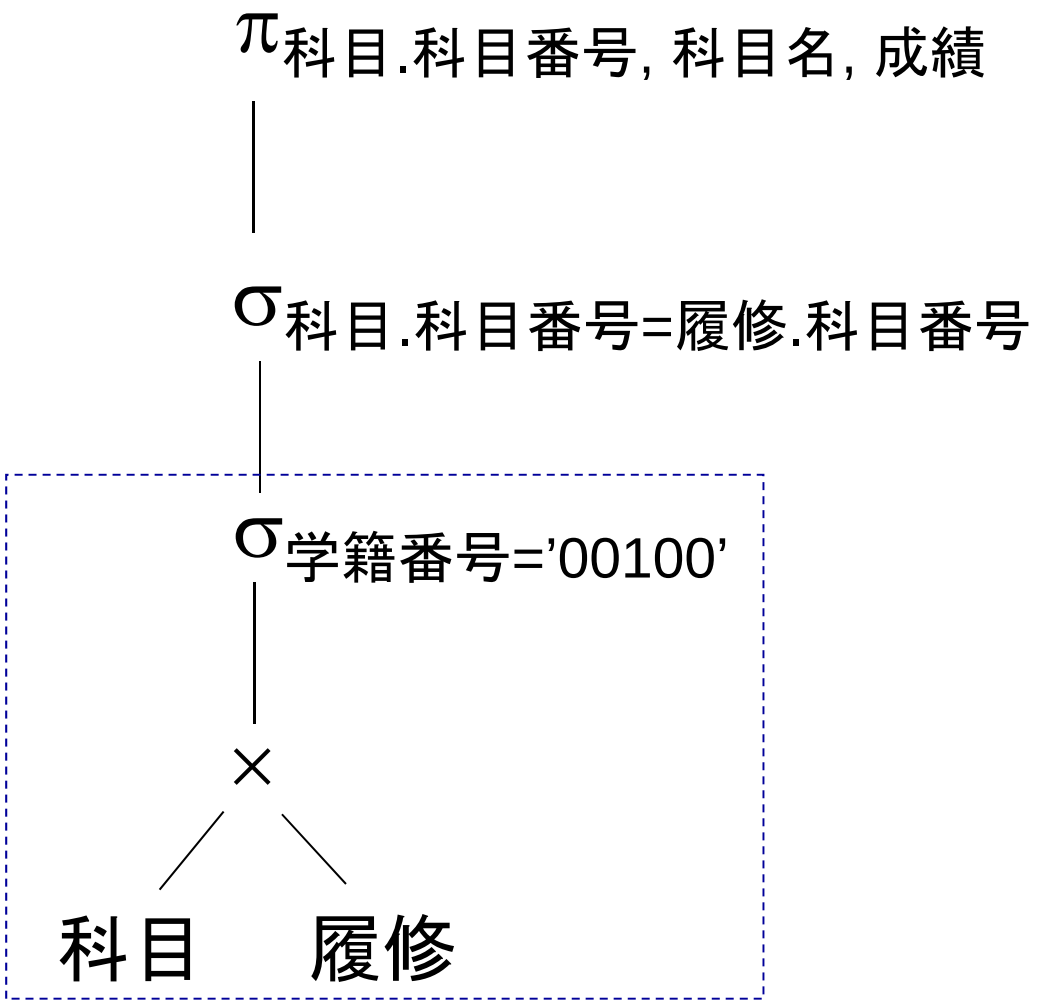
変換処理の例(1)

- 処理木(processing tree)の例:初期状態



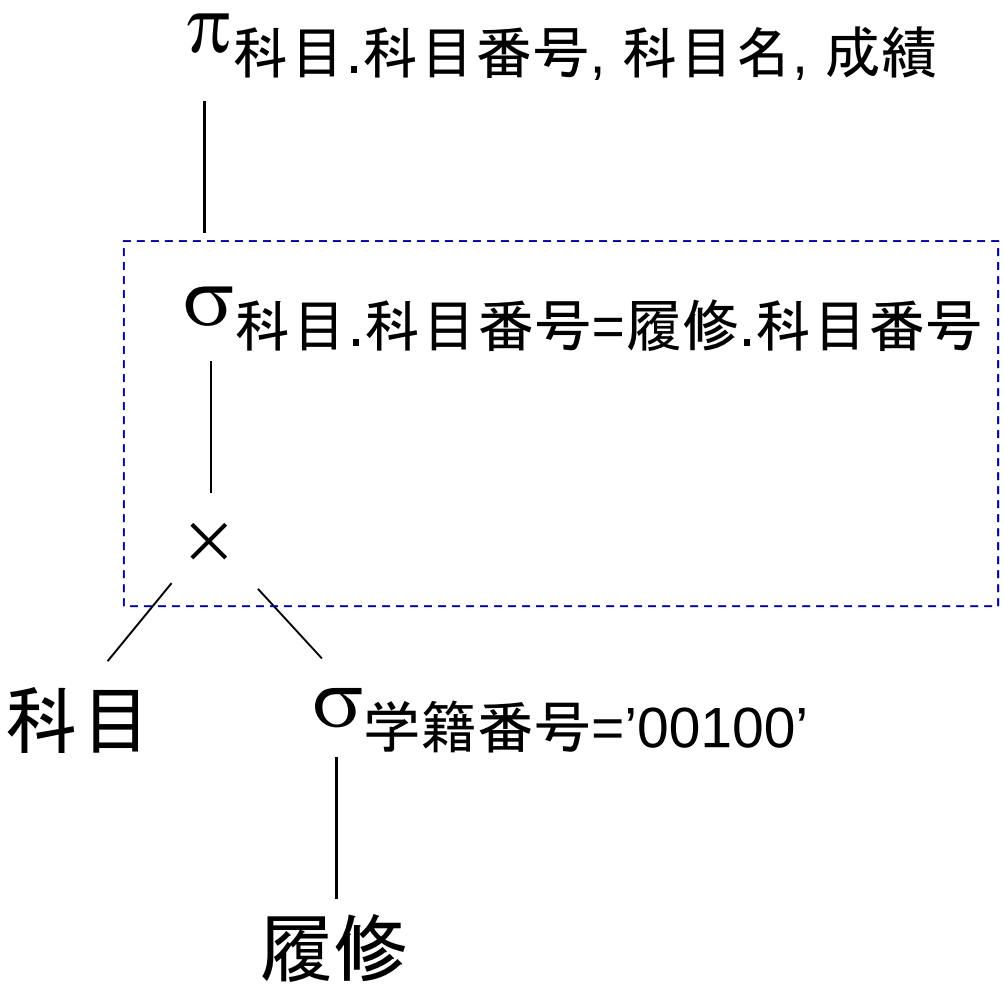
変換処理の例(2)

- ステップ1:規則①を適用し，選択を分解



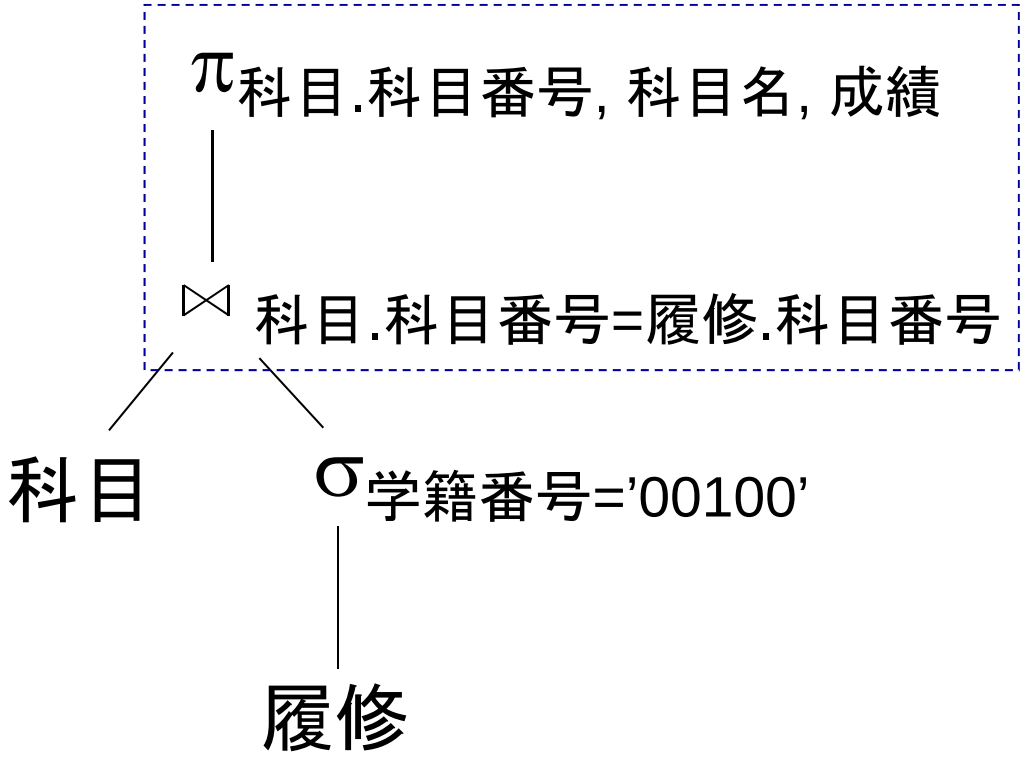
変換処理の例(3)

- ステップ2:規則⑤を適用し，選択を下へ



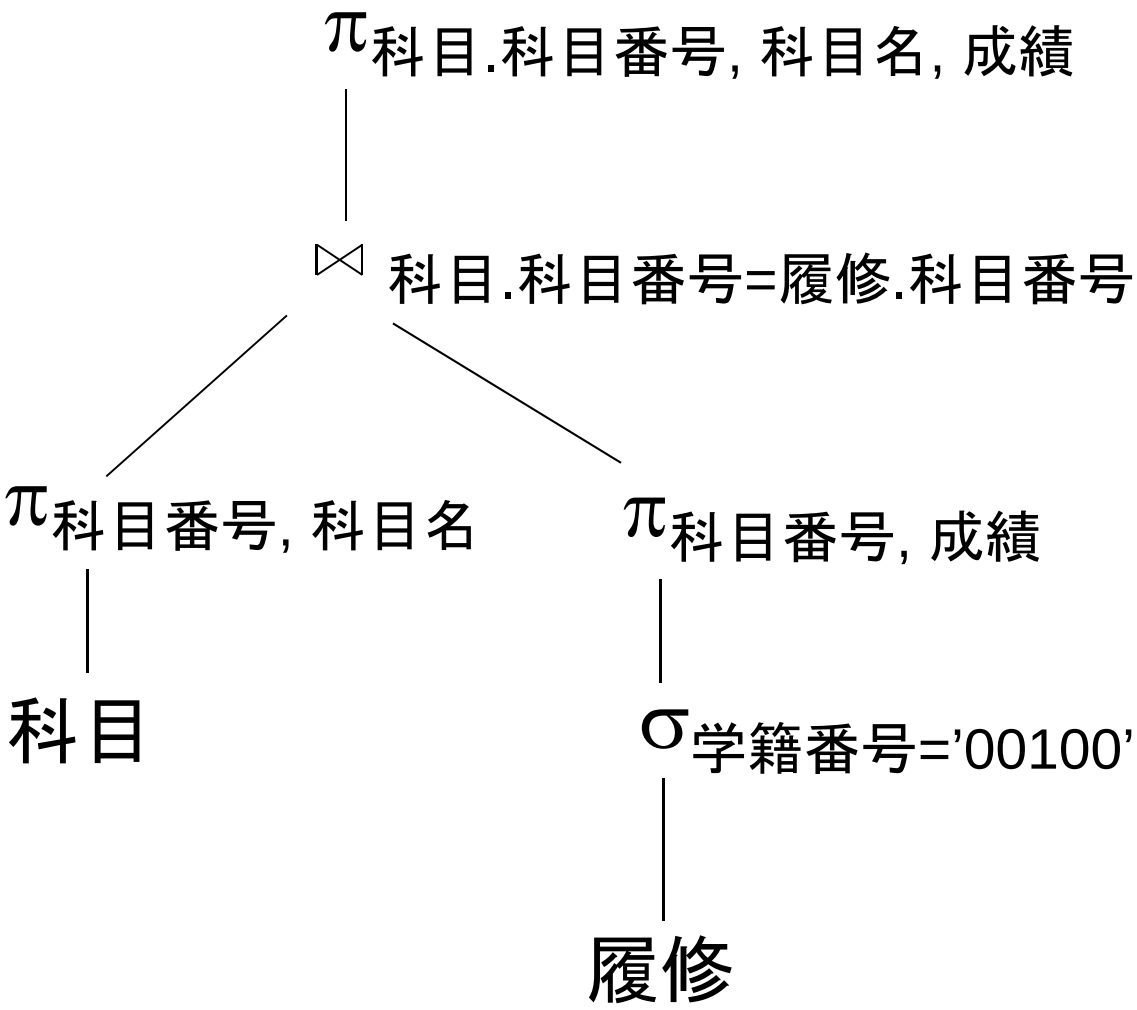
変換処理の例(4)

- ステップ3: 直積とそれに続く選択を結合に



変換処理の例(5)

- ステップ4:規則⑦により射影を下へ



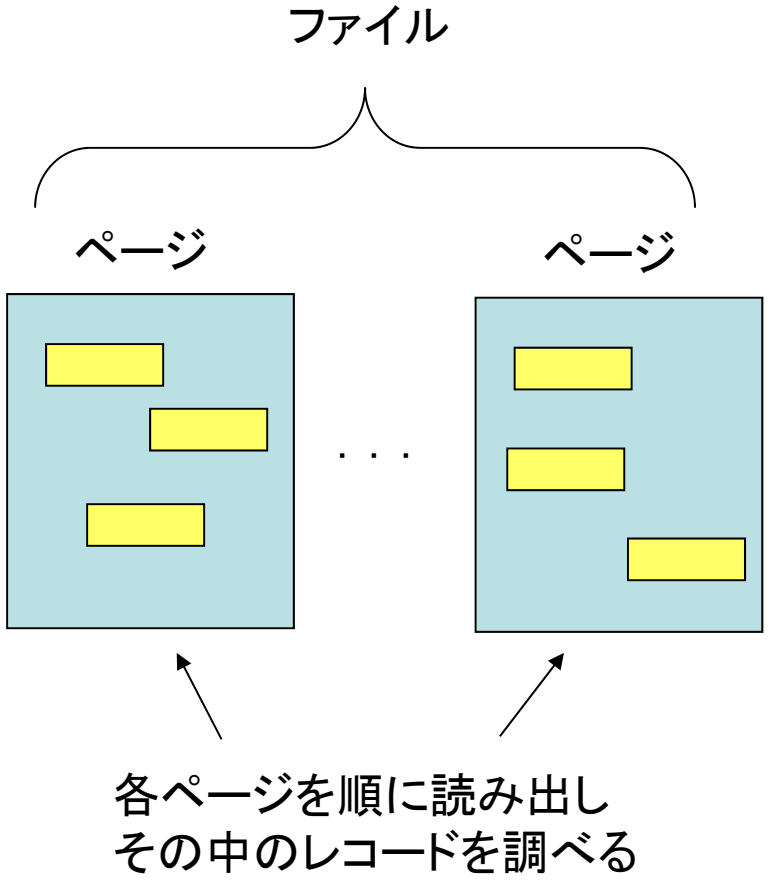
基本データ操作の実行法

第2フェーズでの処理

- リレーショナル代数演算子のデータ操作を具体的に実行
  - 物理的なデータファイルや索引ファイルが対象
  - 同じ演算子（例：結合）でも複数の処理方法（アクセスステップ）が存在
  - リレーショナル演算子とアクセスステップは必ずしも一対一に対応しない
- 以下では以下の処理について説明
  - 選択
  - 結合：処理時間を要するため，効率化は非常に有益
  - ソート

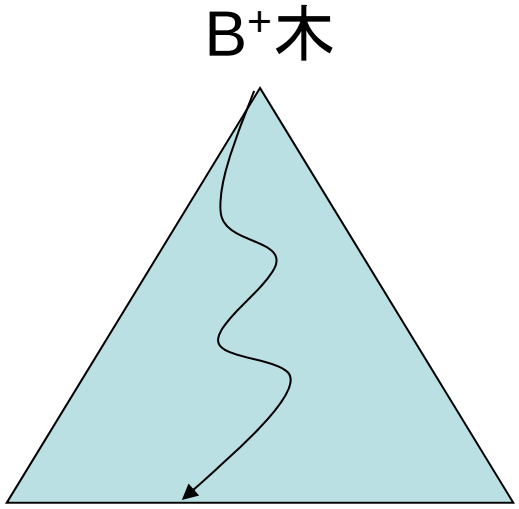
選択の実行法(1)

- 選択が単一の比較条件 $A\ \theta c$ の場合
  - A は属性，θ は比較演算，c は定数
  - 例:「成績 = 80」,「年齢 < 30」
- 方法A) 線形探索
  - データファイルの全レコードを順次読み出し，選択条件を満たすものを抽出
  - 適用範囲が広い
  - 対象レコード数が多い場合非効率



選択の実行法(2)

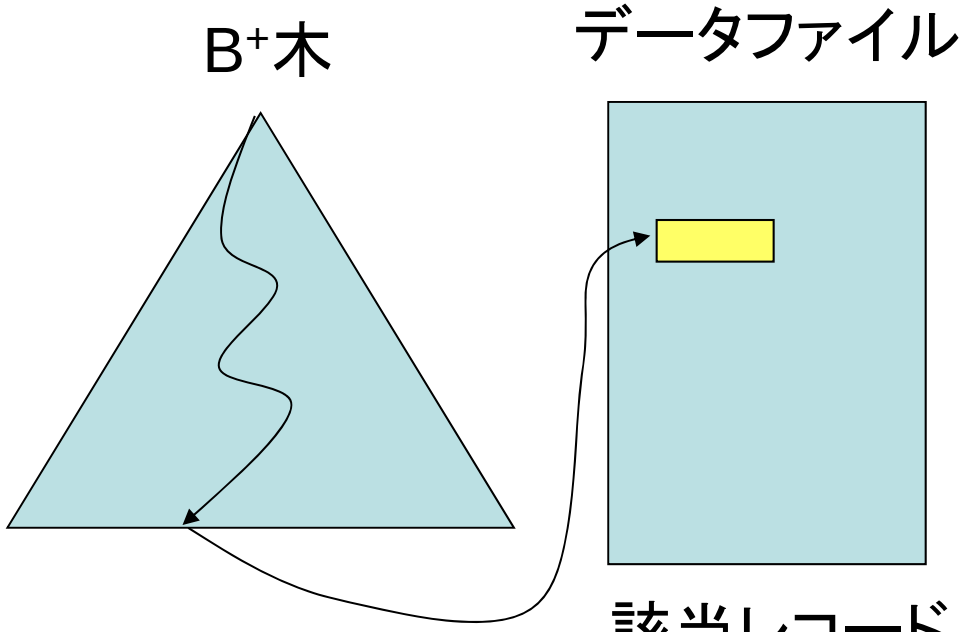
- 方法B) 主索引を用いた探索
  - データファイルが主索引を持つファイル編成(ハッシュファイル，索引付ファイル，B^+木など)の場合適用可能
  - 効率的
  - 適用できない場合あり



・キー値でB^+木を探索
・必要なレコードのみ読み出す

選択の実行法(3)

- 方法C) 二次索引を用いた探索
  - 属性 A に二次索引がある場合
  - 選択条件を満たすデータレコードへのポインタを獲得し，データファイル中の該当レコードを読み出す
  - 効率的
  - 適用できない場合あり



選択の実行法(4)

- 選択条件が複数の比較条件の論理積の場合(例:「年齢 = 21 AND 成績 > 80」
- 方法E) レコードポインタ集合の共通集合演算
  - すべての比較条件について索引が利用できる場合
  - 例
    - 「年齢 = 21」で索引を探索し，レコードポインタ(レコード識別子の集合)$\{R_2, R_3, R_5\}$ を取得
    - 「成績 > 80」で別の索引を探索し，$\{R_1, R_3, R_5, R_8\}$ を取得
    - 共通集合 $\{R_3, R_5\}$ を求める
    - R_3, R_5で指されたレコードを読み出す

選択の実行法(5)

- 方法F）索引を用いた候補レコードの絞込み
  - 一部の比較条件について索引が利用可能な場合，そのレコードを探索
  - 次にそれらレコードを順に読み出し，残りの選択条件を満たすものを抽出
  - 例：「年齢 = 21 AND 成績 > 80」で，成績のみに索引がある場合
    - 「成績 > 80」で索引を探索し，$\{R_1, R_3, R_5, R_8\}$ を取得
    - 各レコードを取得し，「年齢 = 21」を満たすものだけを選択

結合の実行法(1)

- 想定
  - 2つのリレーションのデータを格納したファイル S_1 と S_2 の結合操作を実行
  - 等結合(自然結合)を対象
  - 等結合の条件は $A_1 = A_2$
  - S_1, S_2 のレコードを $R_1(1), \ldots, R_1(N_1)$ および $R_2(1), \ldots, R_2(N_2)$ とする

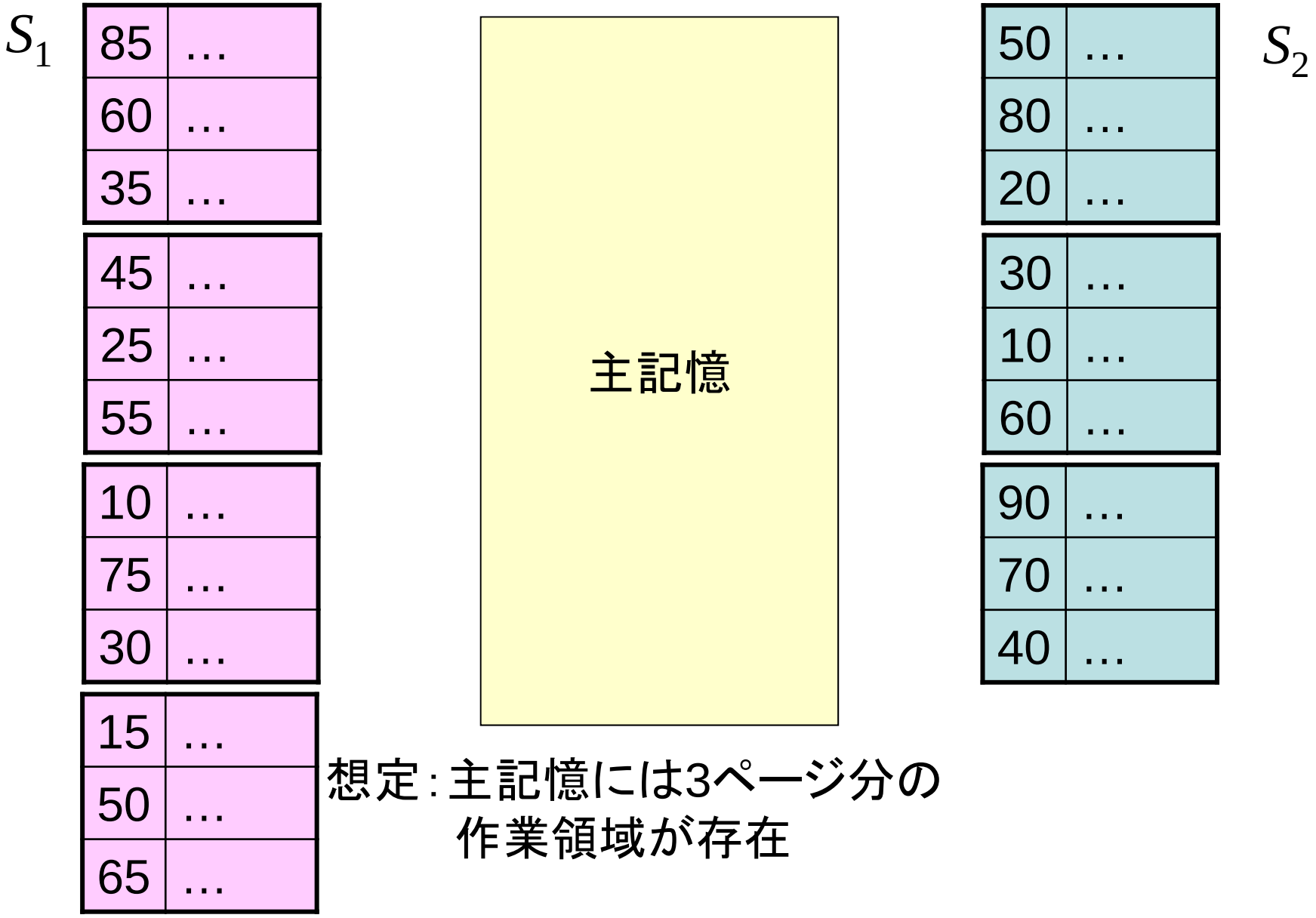
結合の実行法(2)

- 入れ子ループ結合(nested loop join)
  - 最も基本的なアルゴリズム
  - ファイル S_1 のレコード $R_1(i)$ を一つ読み出しては，ファイル S_2 の N_2 個のレコードとつきあわせ，結合条件を判定

```
for i := 1 to N1 do   ← 外部ループ:S1 のレコードを順次読み出し
  for j := 1 to N2 do   ← 内部ループ:S2 のレコードを順次読み出し
    if R1(i)[A1] = R2(j)[A2] then
      R1(i) と R2(j) を結合したレコードを出力
```

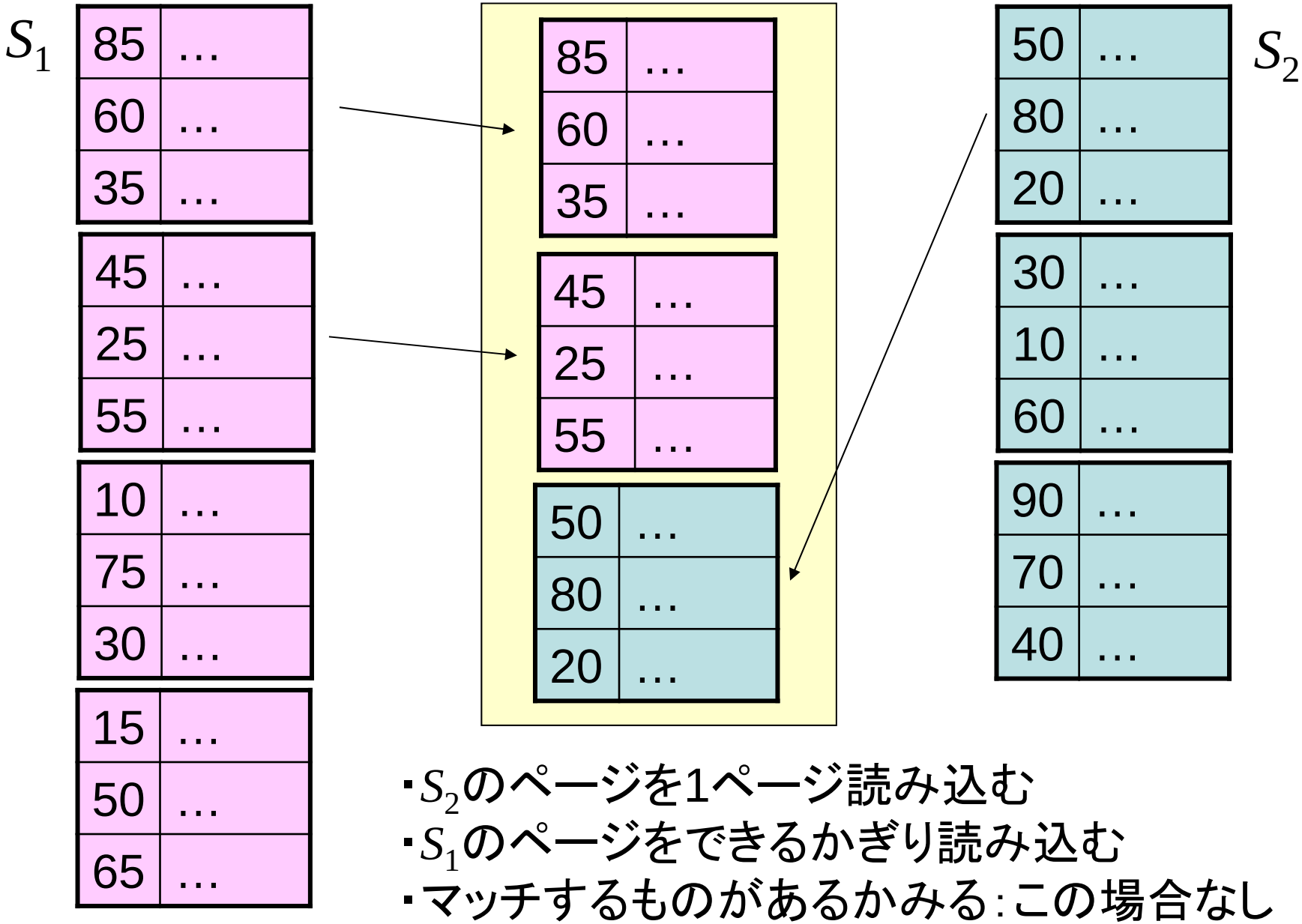
  - 実際の処理:ページ単位の読み出しで効率化

入れ子ループ結合の処理(1)



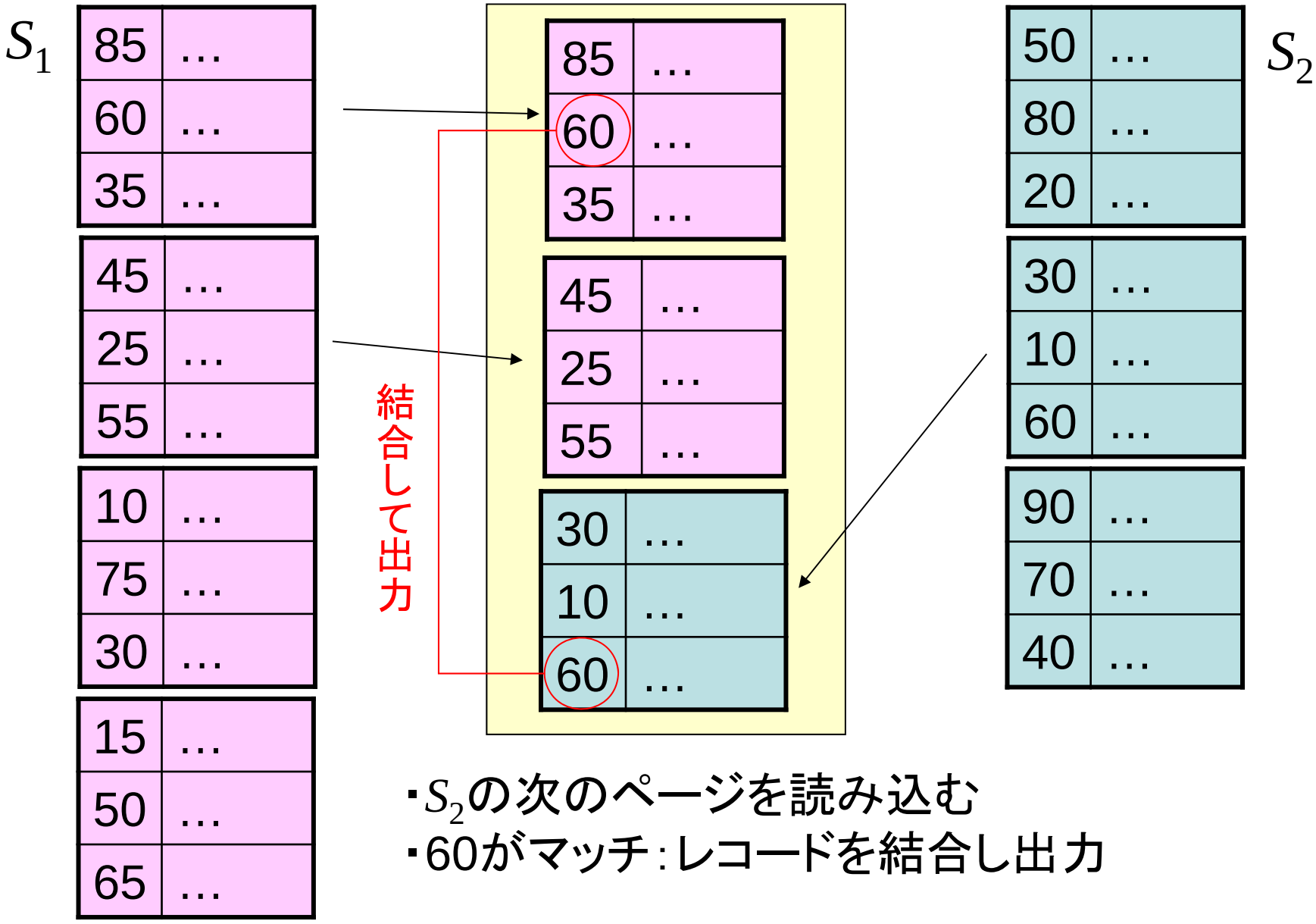
想定:主記憶には3ページ分の
作業領域が存在

入れ子ループ結合の処理(2)



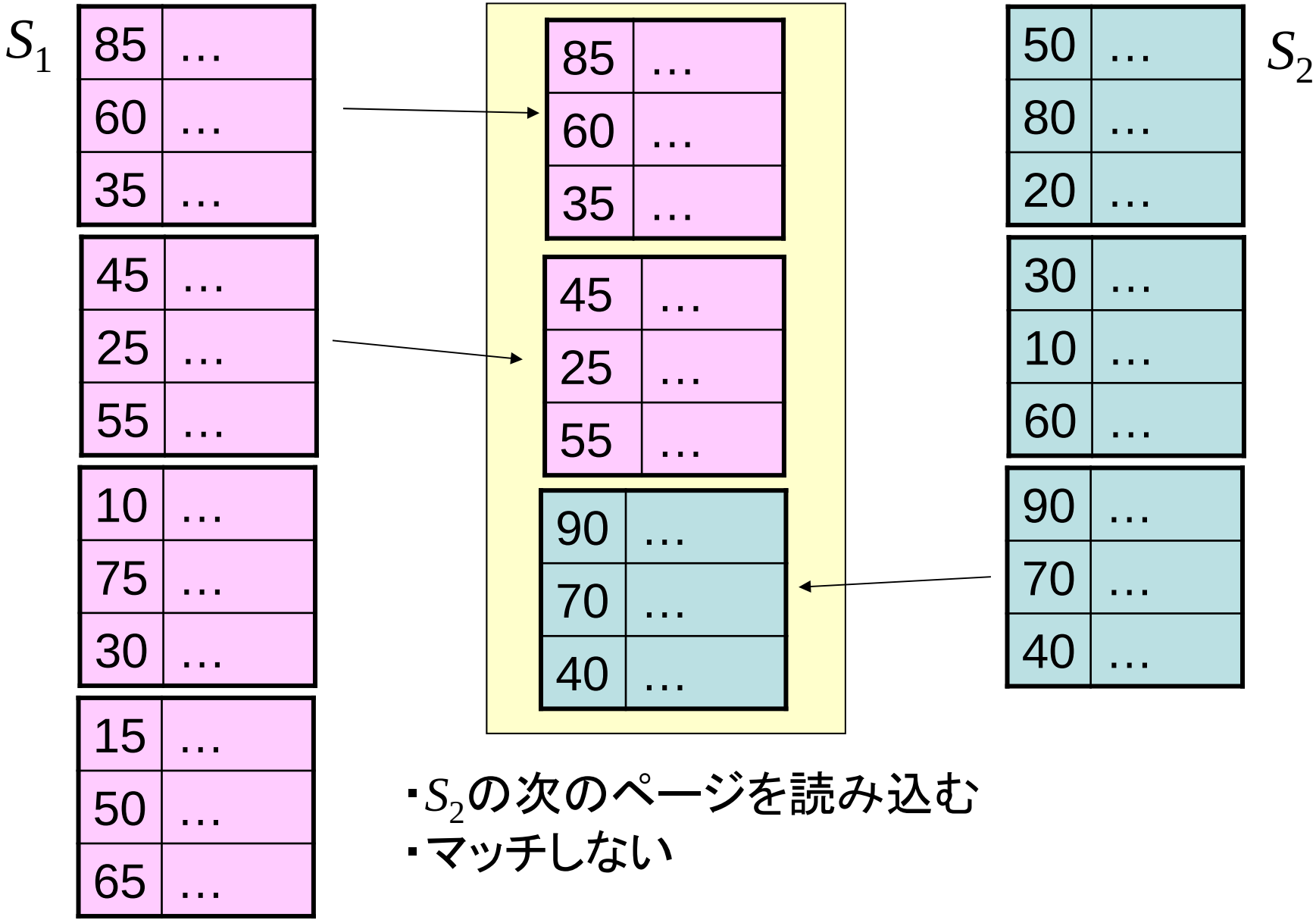
- S_2のページを1ページ読み込む
- S_1のページをできるかぎり読み込む
- マッチするものがあるかみる:この場合なし

入れ子ループ結合の処理(3)



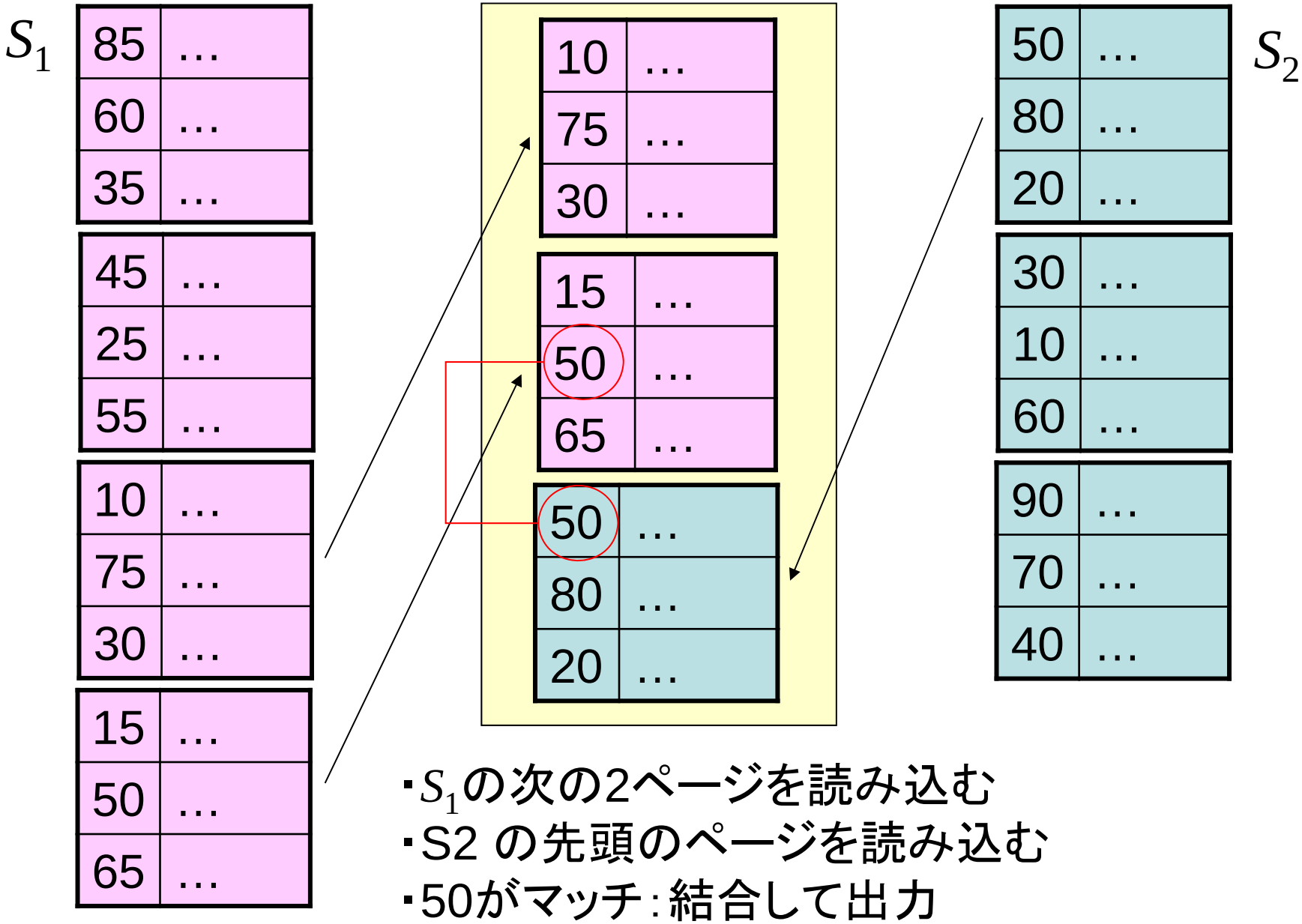
- S_2の次のページを読み込む
- 60がマッチ:レコードを結合し出力

入れ子ループ結合の処理(4)



- S_2の次のページを読み込む
- マッチしない

入れ子ループ結合の処理(5)



- S_1の次の2ページを読み込む
- S2 の先頭のページを読み込む
- 50がマッチ：結合して出力

入れ子ループ結合の処理(6)

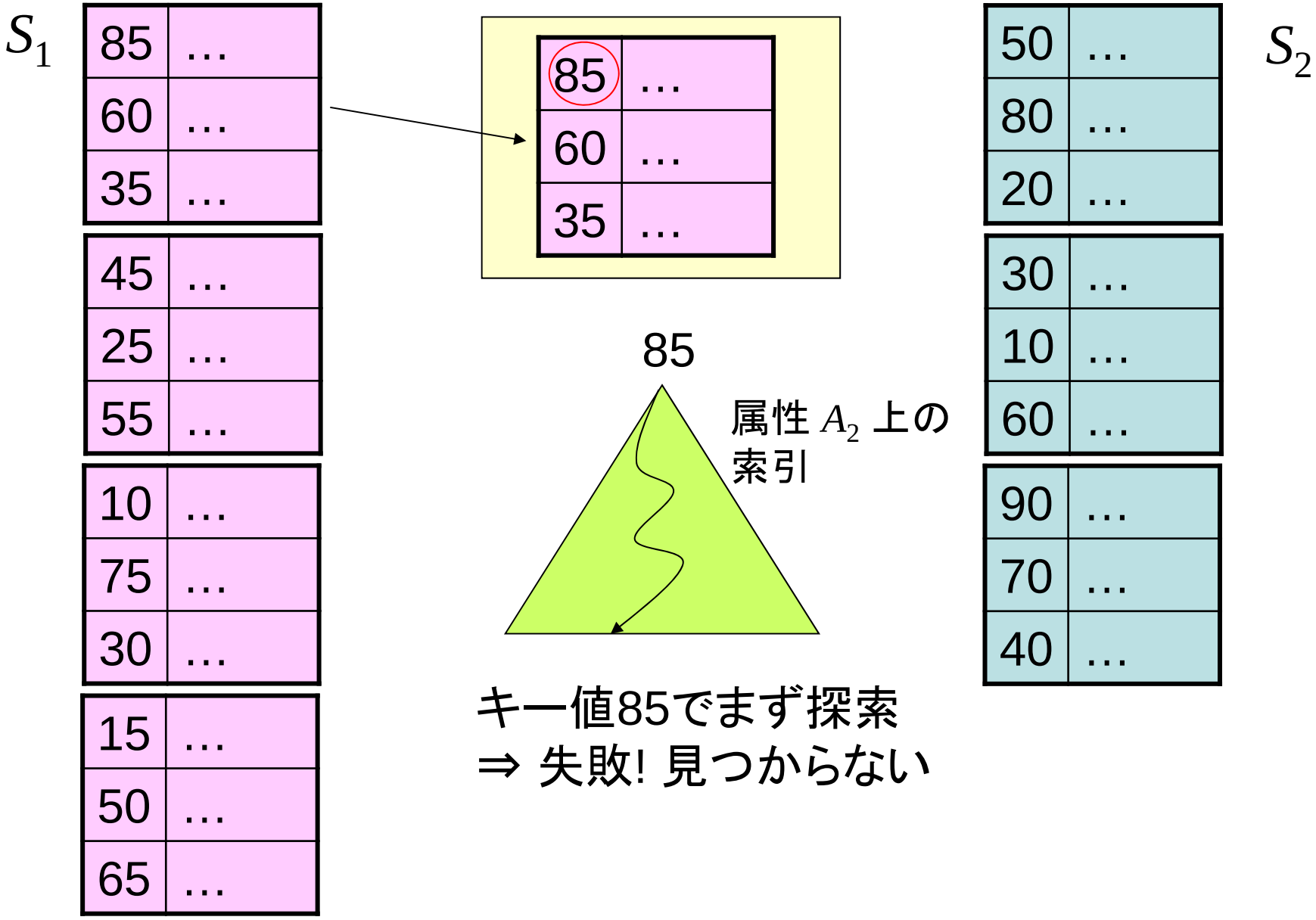
- 具体的なアルゴリズムの処理コストの解析
  - S_1, S_2 のページ数を P_1, P_2 とする
    - 例では $P_1 = 4, P_2 = 3$
  - S_1, S_2 のページをそれぞれ M_1, M_2 ページずつ主記憶に読み出すとする
    - 例では $M_1 = 2, M_2 = 1$
  - S_1 の各ページは1回ずつ読み出される
    - 合計で4ページ
  - S_2 の各ページは $\lceil P_1 / M_1 \rceil$ 回ずつ読み出される
    - 合計で $\lceil P_1 / M_1 \rceil \times 3 = \lceil 4/2 \rceil \times 3 = 6$ ページ
  - つまり，総計では10ページの読み出し
  - M_1 が大きいほうが有利

索引を用いた結合（1）

- A_2 に関する S_2 の索引がある場合に，それを利用して効率化

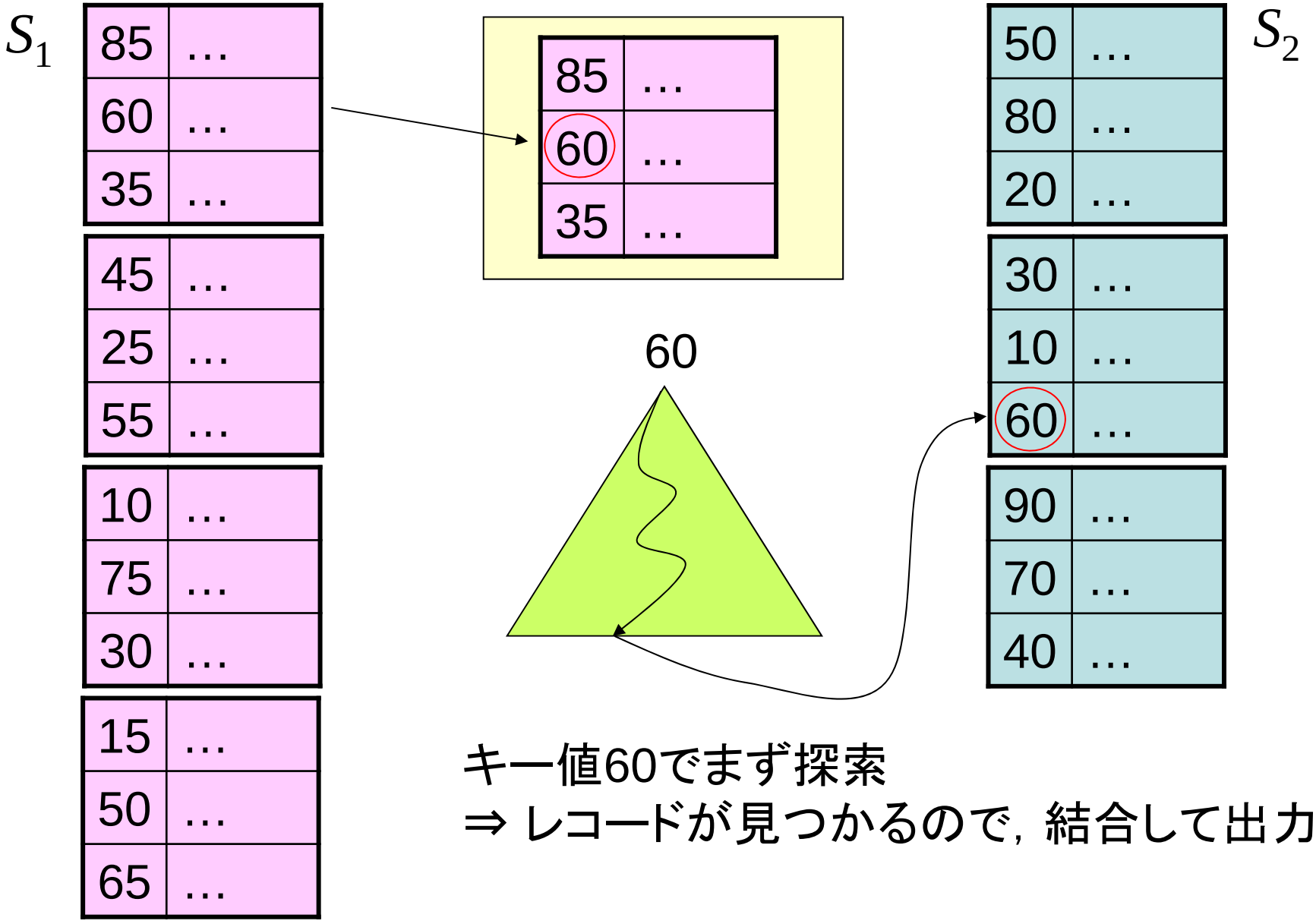
for i := 1 **to** N_1 **do**
  begin
    A_2 に関する索引を用いて $R_1(i)[A_1] = R_2(j)[A_2]$ を満たす S_2 のレコード $R_2(j)$ を探索
    $R_1(i)$ と $R_2(j)$ を結合したレコードを出力
  end

索引を用いた結合（2）



キー値85でまず探索
⇒ 失敗! 見つからない

索引を用いた結合（3）

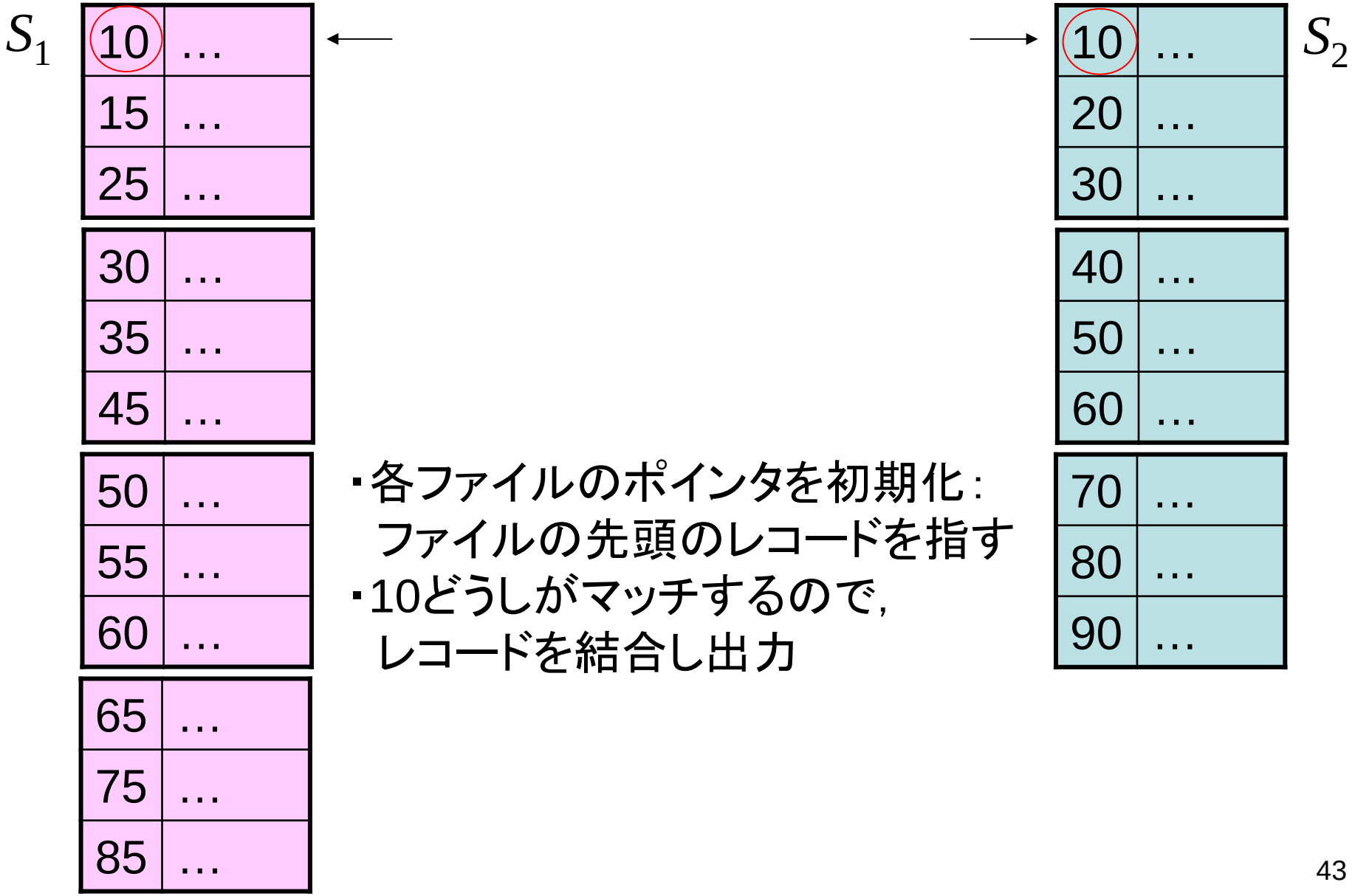


キー値60でまず探索
⇒ レコードが見つかるので，結合して出力

マージ結合（1）

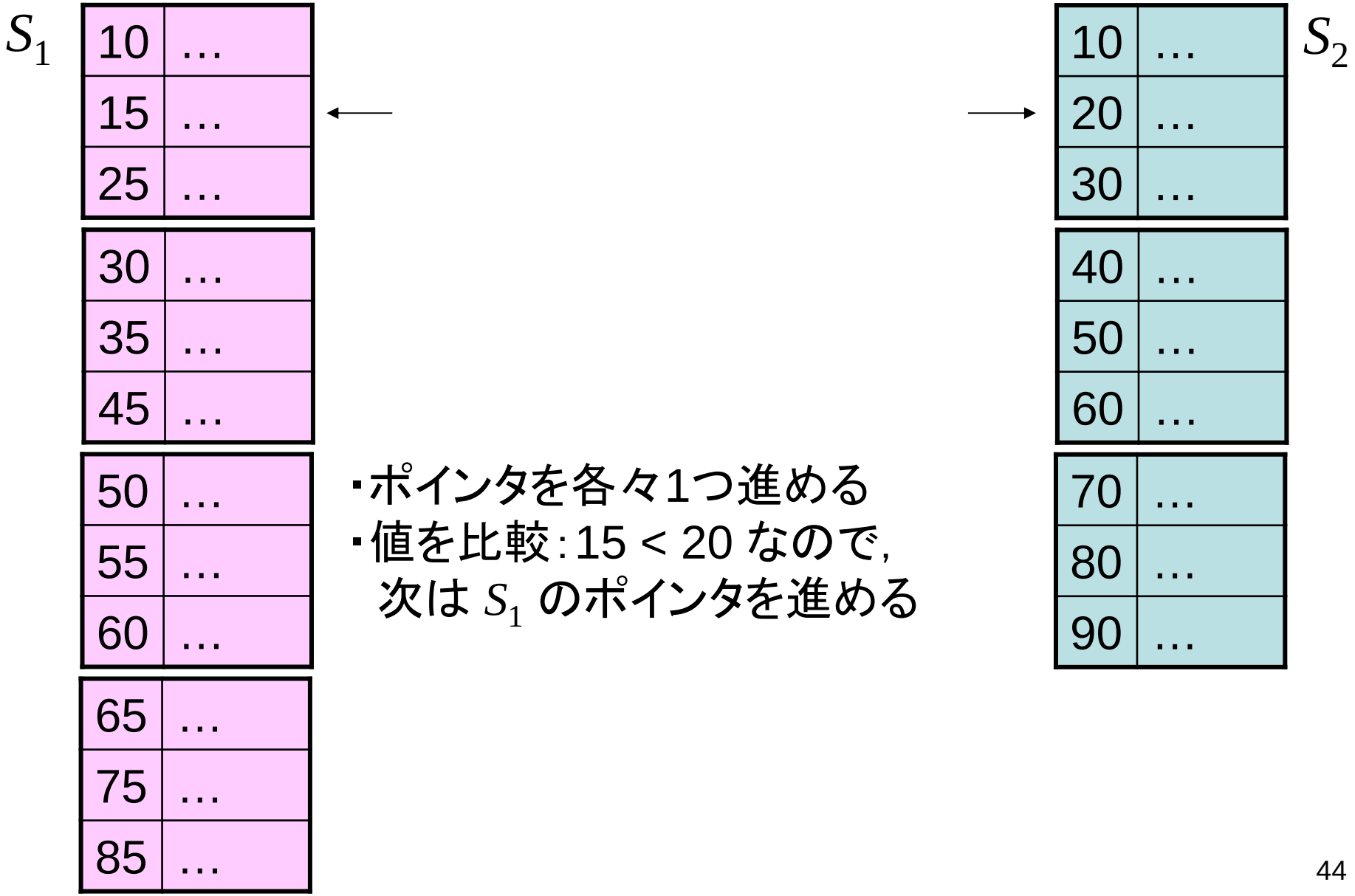
- マージ結合（merge join）
- S_1 と S_2 がそれぞれ A_1, A_2 でソートされている場合に適用可能
  - ソートされていない場合は，事前にソート処理を行う
- 両者のファイルのレコードを先頭から比べていく
- アルゴリズムは教科書を参照

マージ結合（2）



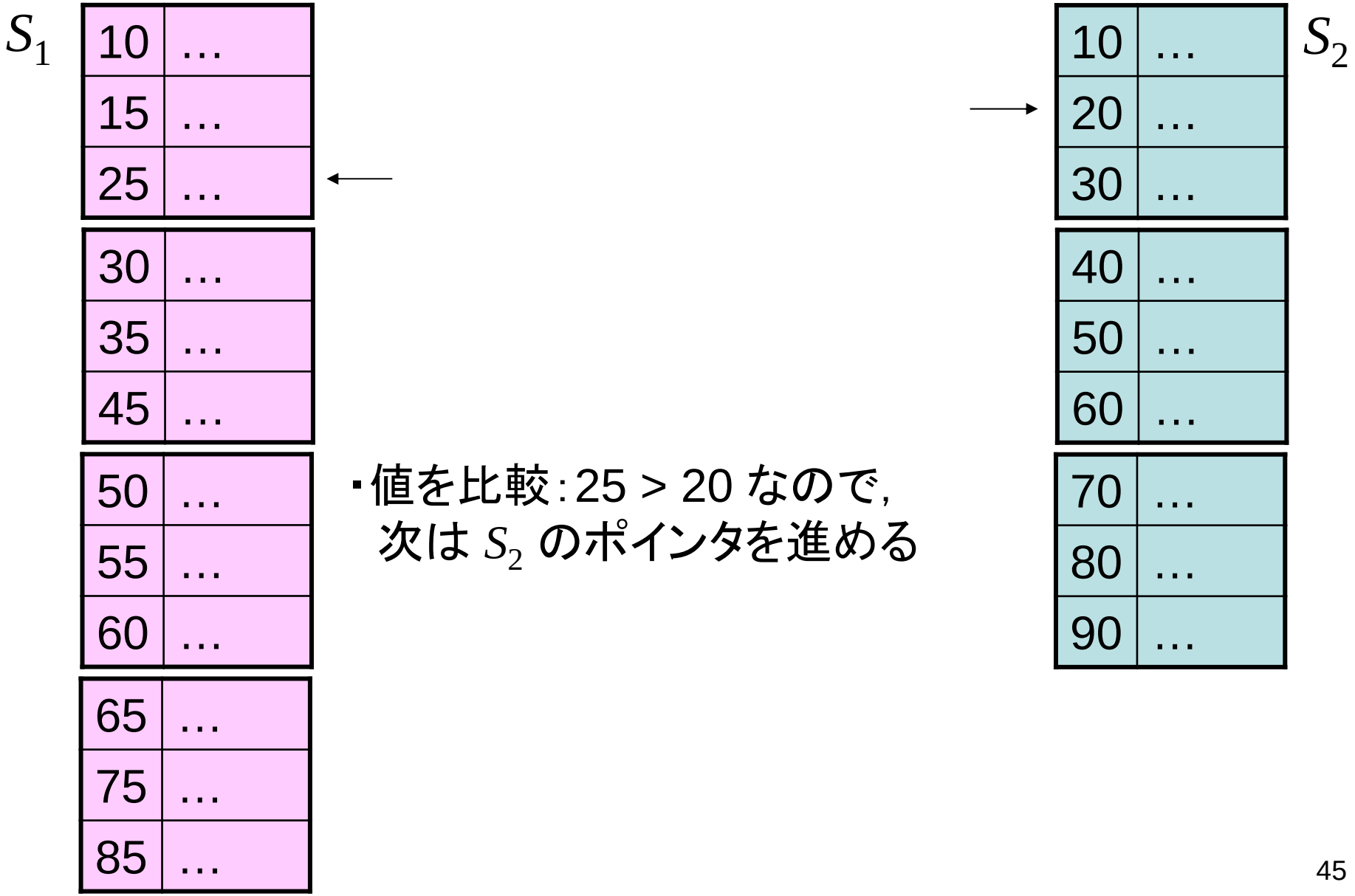
- 各ファイルのポインタを初期化：
  ファイルの先頭のレコードを指す
- 10どうしがマッチするので，
  レコードを結合し出力

マージ結合（3）



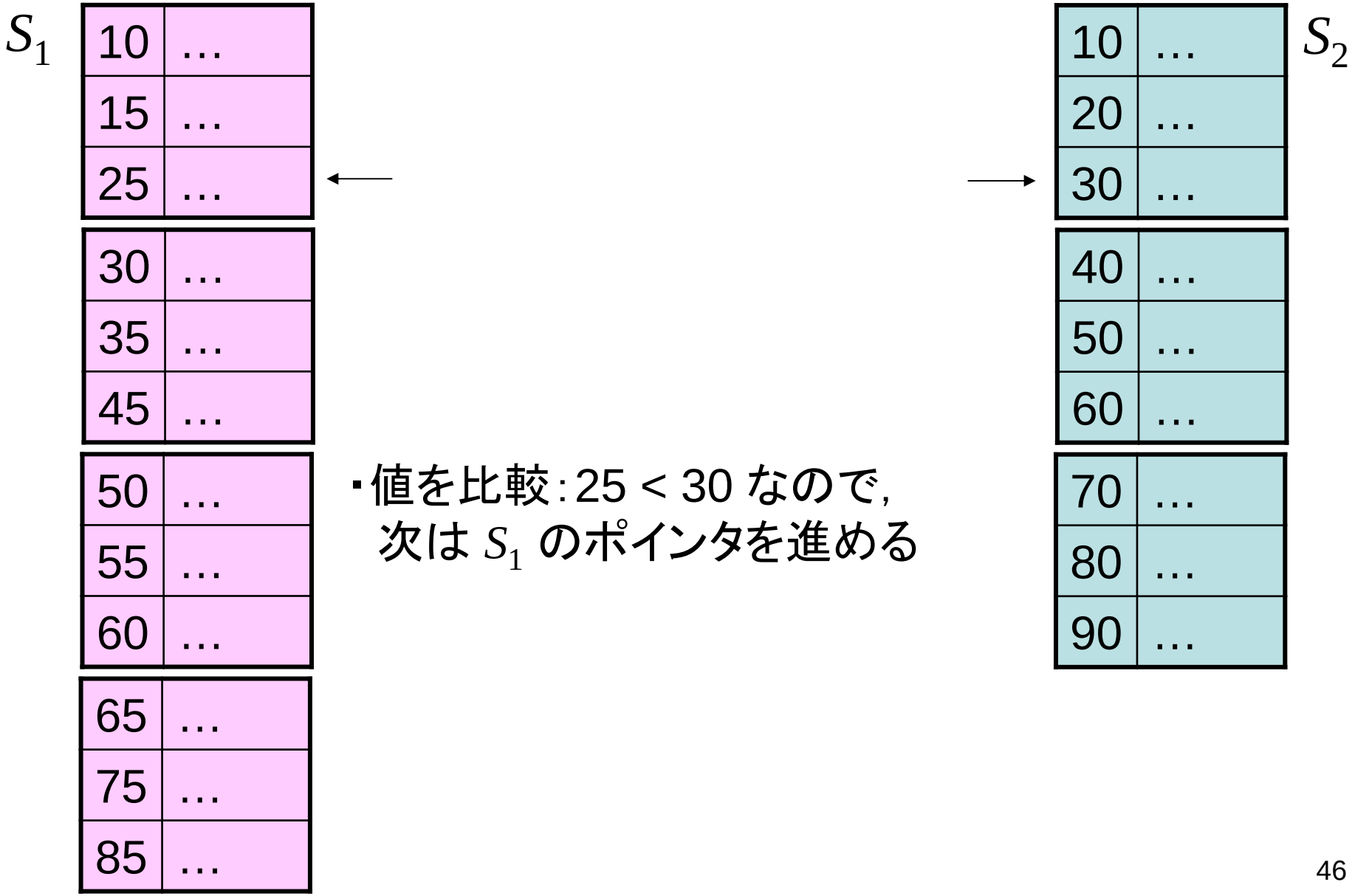
- ポインタを各々1つ進める
- 値を比較：15 < 20 なので，
  次は S_1 のポインタを進める

マージ結合（4）



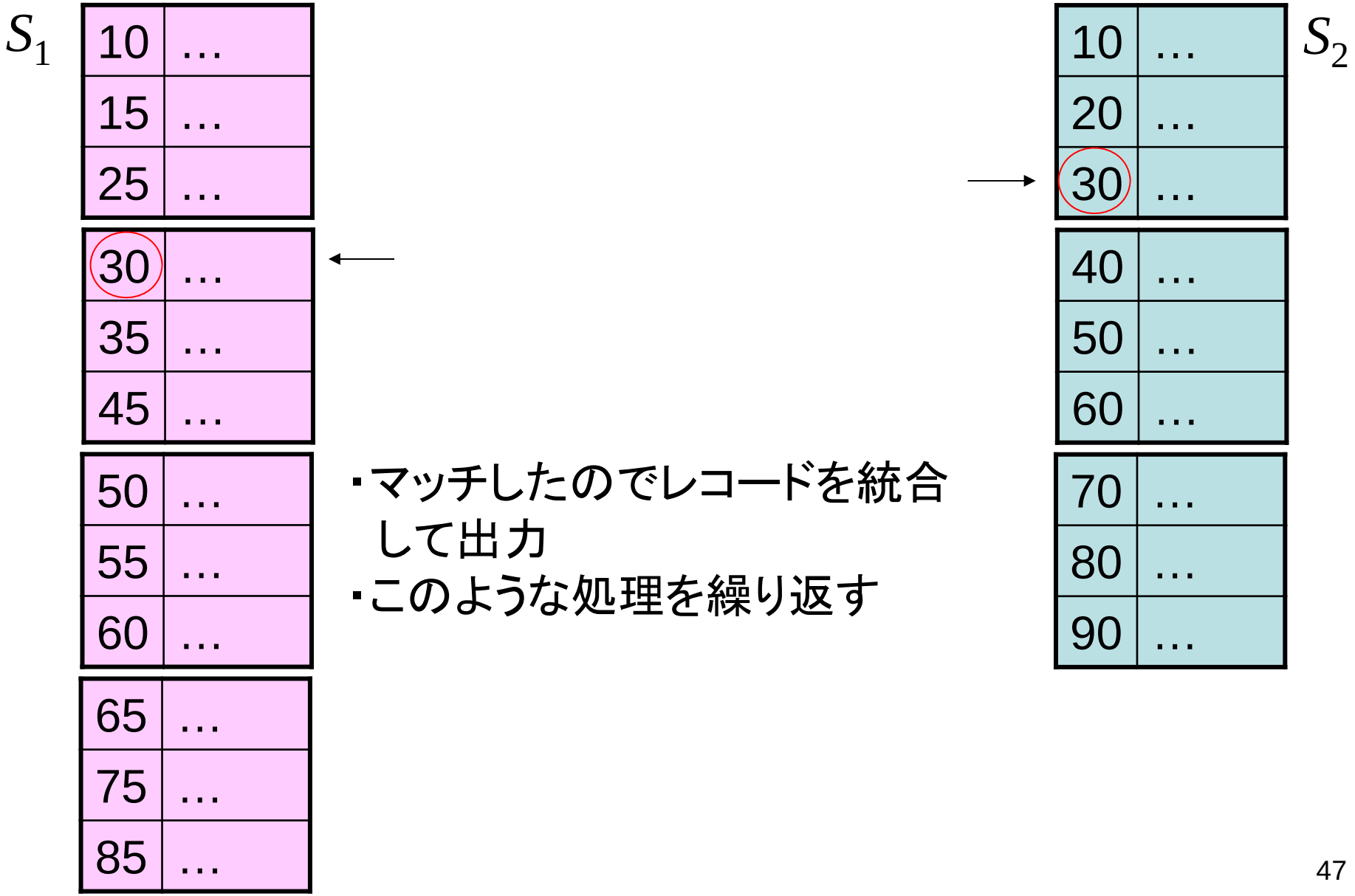
- 値を比較：25 > 20 なので，
  次は S_2 のポインタを進める

マージ結合（5）



- 値を比較：25 < 30 なので，
  次は S_1 のポインタを進める

マージ結合（5）



- マッチしたのでレコードを統合して出力
- このような処理を繰り返す

ハッシュ結合（1）

- ハッシュ結合（hash join）
- 一方のファイルのレコードをハッシュ表に展開
- hash：ハッシュ関数，$B(1), \ldots, B(K)$：バケット

```
// S1 に対しハッシュ表を構築
for i := 1 to N1 do
  レコード R1(i) を B(hash(R1(i)[A1])) に格納
// S2の各レコードについてハッシュを引く
for j := 1 to N2 do
  for each レコード R1(i) in B(hash(R2(j)[A2]))
   if R1(i)[A1] = R2(j)[A2] do
      R1(i) と R2(j) を結合したレコードを出力
```

ハッシュ結合（2）

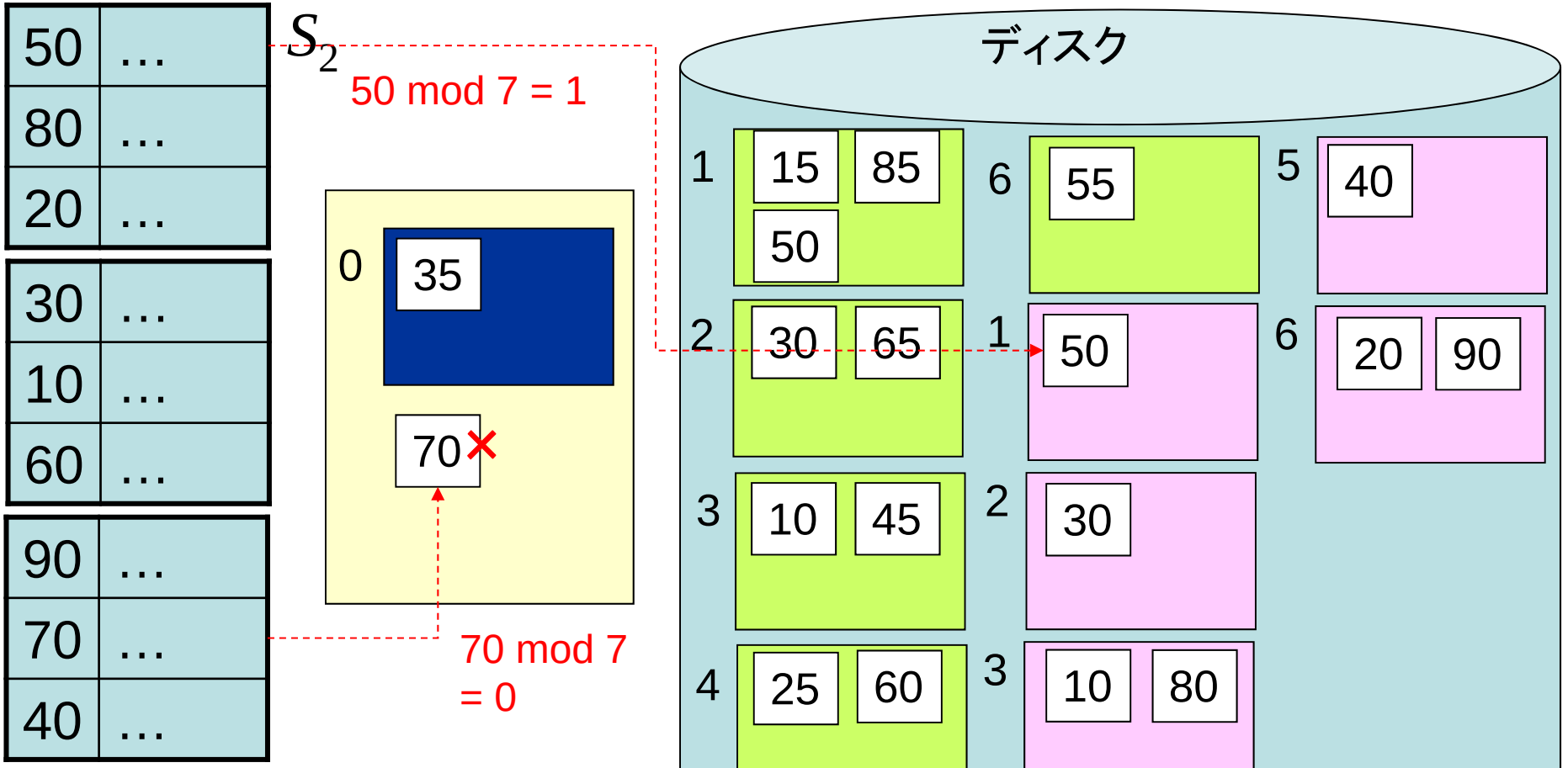
- 特徴
  - ハッシュ関数により，S_1 のレコード集合をバケットに分配
  - S_2 のあるレコード $R_2(j)$ に対し，それが結合する可能性がある S_1 のレコードは（あるとすれば）バケット $B(\mathrm{hash}(R_2(j)[A_2]))$ の中にしかない
    - 候補の大幅な絞込みが可能
  - 問題点：等結合にしか利用できない
- ハイブリッドハッシュ結合（hybrid hash join）
  - ハッシュ表が主記憶上に保持できない場合
  - 並列化による効率化に適する

ハイブリッドハッシュ結合（1）

ハッシュ表の構築

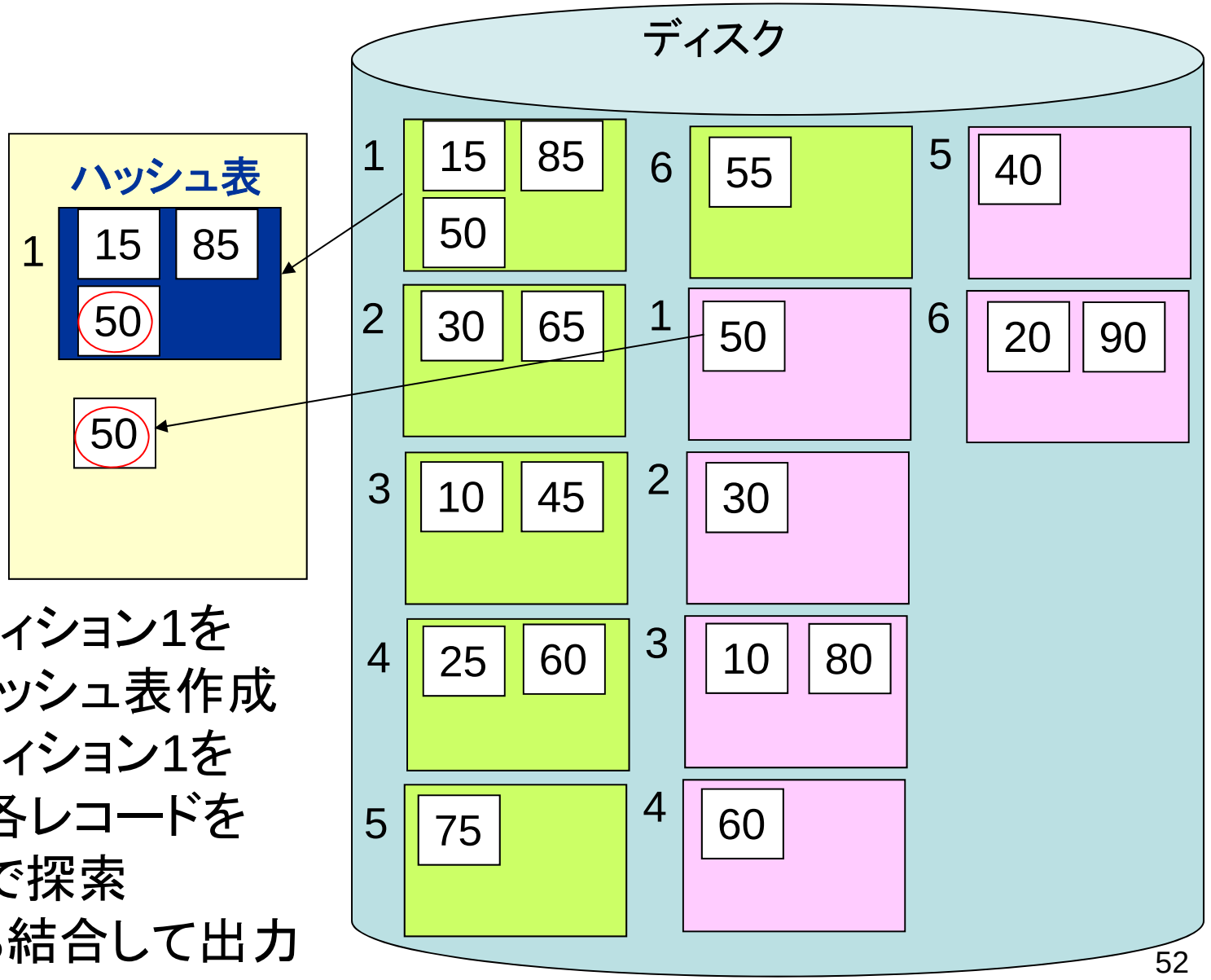
- ハッシュ値の値で L + 1 個のパーティションに分割
  - この例では $L = 6$ 個
- ハッシュ関数の例：$\mathrm{hash}(x) = x \bmod 7$
- ハッシュ値 0 のレコードはハッシュ表に登録
- 残りはハッシュ値ごとにディスク上のパーティションへ

ハイブリッドハッシュ結合（2）



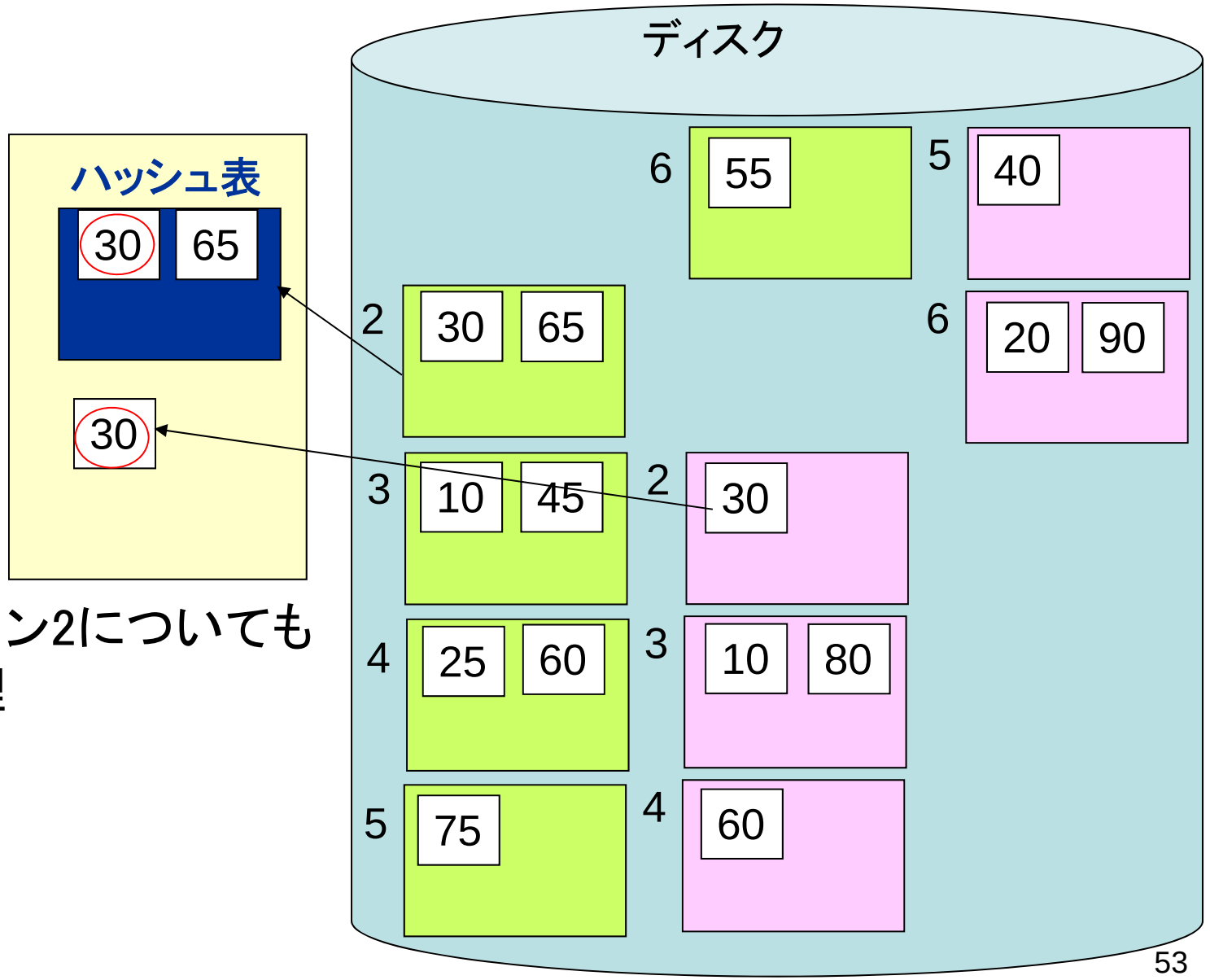
- S_2 のレコードをハッシュ
- ハッシュ値 0 ならハッシュ表を探索：マッチしたら出力
- 0以外のハッシュ値：ディスクへ

ハイブリッドハッシュ結合（3）



- S_1 のパーティション1を読み出しハッシュ表作成
- S_2 のパーティション1を読み出し，各レコードをハッシュ表で探索
- マッチしたら結合して出力

ハイブリッドハッシュ結合（4）



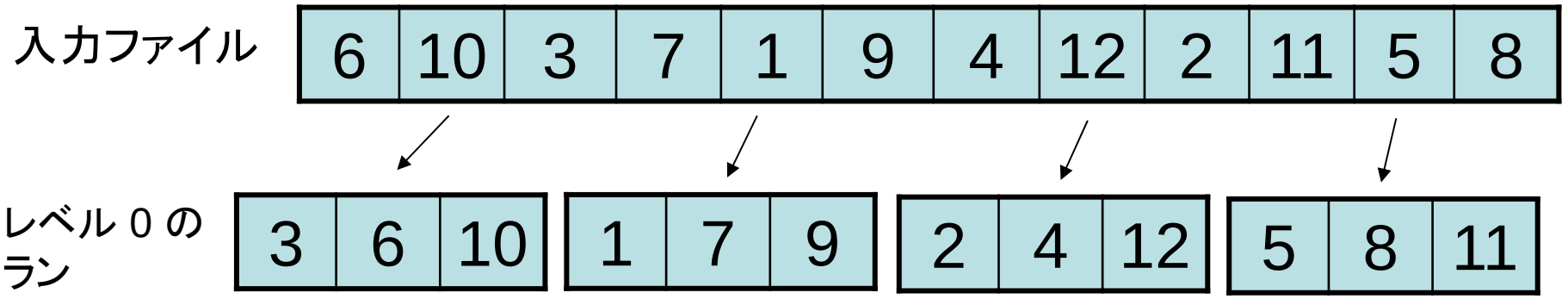
・パーティション2についても
同様に処理

外部ソート

- 問合せ処理ではしばしばソートが必要
  - SQLで「ORDER BY」が指定されたとき
  - マージ結合を行う下準備
  - 重複した行の除去，グルーピング
- 外部ソート（external sort）
  - 主記憶に入りきれない大量のデータをソートする処理
  - 主記憶上で有効なクイックソートは不適切
  - マージソート（merge sort）が一般に利用される

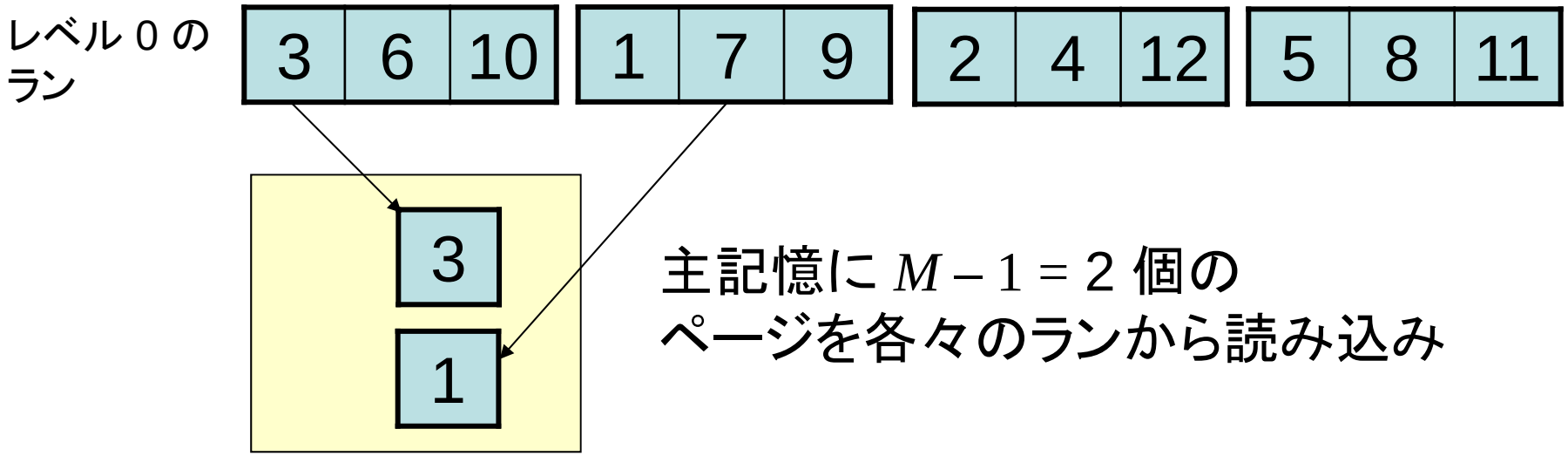
マージソート(1)

- 処理手順①
  - ファイルを先頭から M ページずつ読み出して，主記憶上でそのレコードをソート
  - ソートの済んだ M ページ分のレコードの集まりを，レベル 0 のラン(run)と呼ぶ
  - ランはファイルに出力
- 例：$M = 3$，各ページに1レコードのみ入る

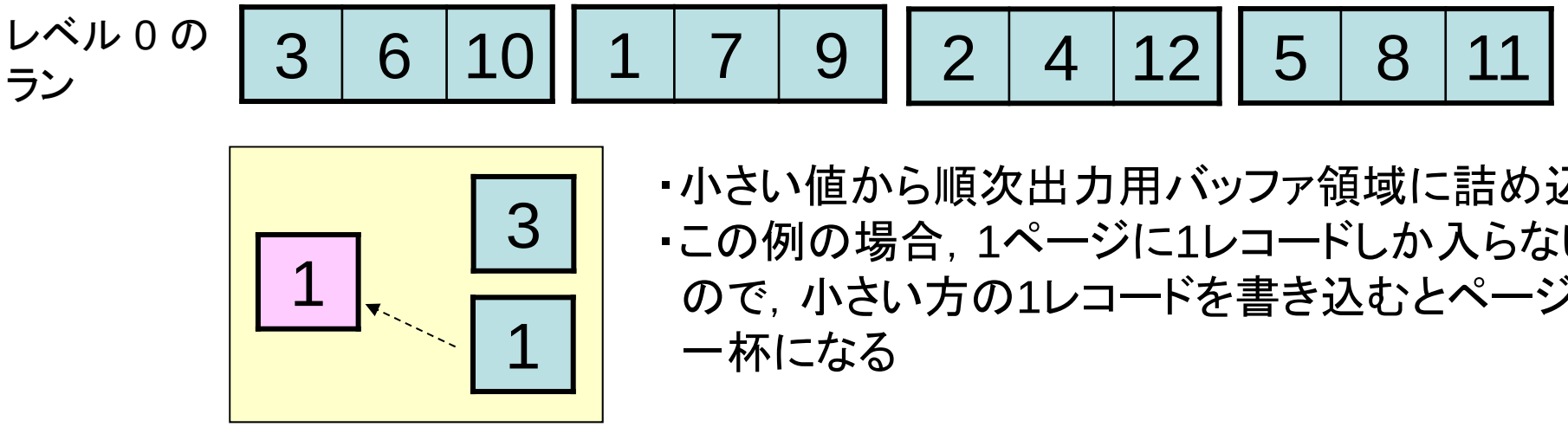


マージソート(2)

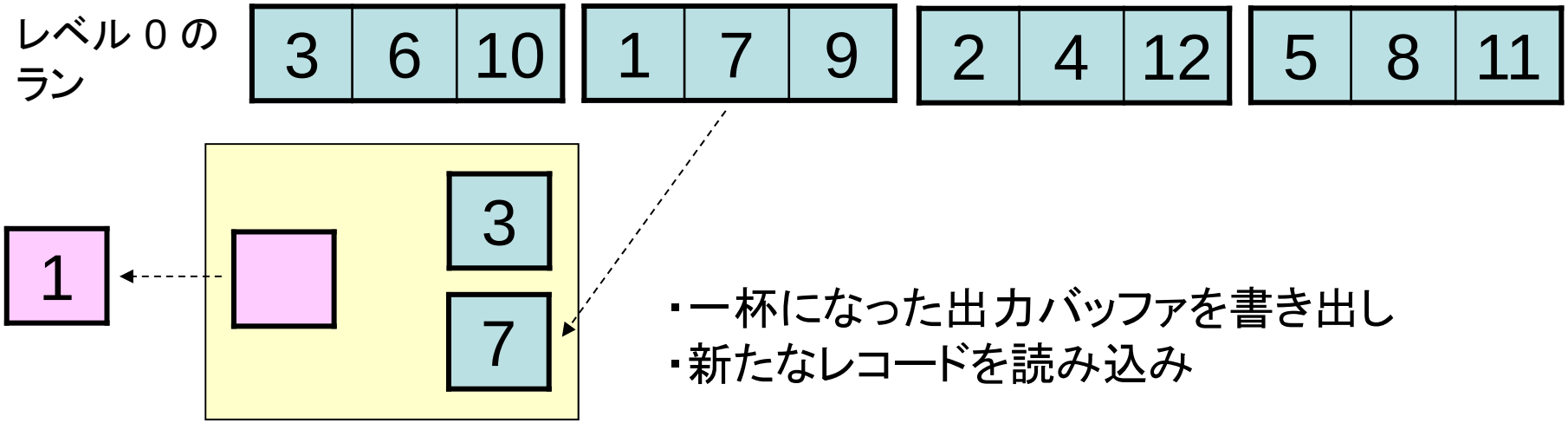
- 処理手順②
  - $K = M - 1$ 本ずつのレベル 0 のランをマージして，1本のレベル1のランとする
  - ランの本数は元の $1 / K$ となる



マージソート(3)

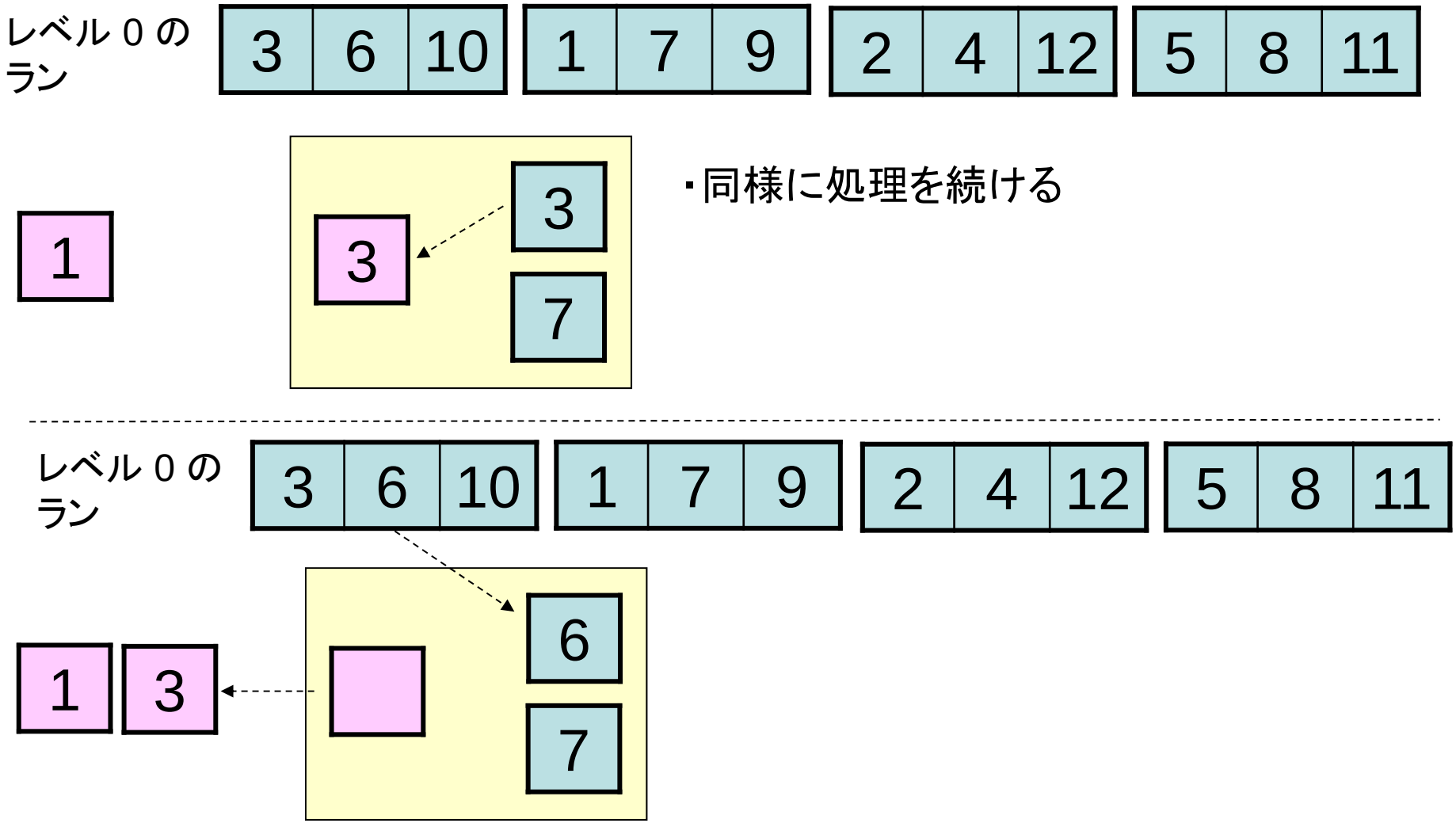


・小さい値から順次出力用バッファ領域に詰め込み
・この例の場合，1ページに1レコードしか入らないので，小さい方の1レコードを書き込むとページが一杯になる

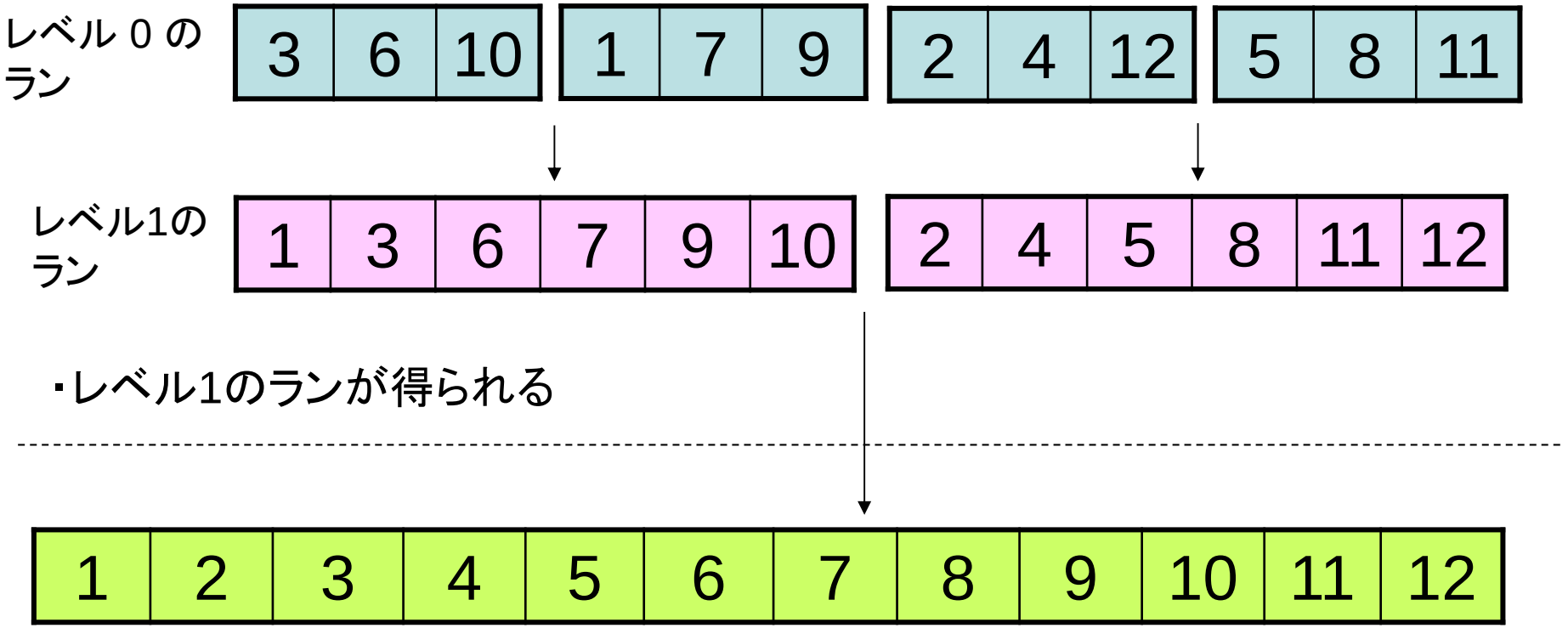


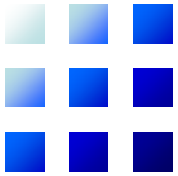
・一杯になった出力バッファを書き出し
・新たなレコードを読み込み

マージソート(4)


レベル 0 のラン
3 6 10
1 7 9
2 4 12
5 8 11
1
3
3
7
・同様に処理を続ける
レベル 0 のラン
3 6 10
1 7 9
2 4 12
5 8 11
1 3
6
7

マージソート(5)


レベル 0 のラン
3 6 10
1 7 9
2 4 12
5 8 11
レベル1のラン
1 3 6 7 9 10
2 4 5 8 11 12
・レベル1のランが得られる
1 2 3 4 5 6 7 8 9 10 11 12
・再度ランをマージして，レベル2のランを得る
・この例の場合はここでソート完了



コスト見積りに基づく問合せ最適化

コスト見積りに基づく問合せ最適化

- いくつかの候補実行プランのコスト（主にページアクセス回数）を定量的に見積り
- コストが最小のものを選択
- 用いられる統計情報
  - ① ファイル中のレコード数
  - ② ファイル中のレコードサイズ
  - ③ ファイルを構成するページ数
  - ④ ファイル中に現れる各フィールドの値の種類
  - ⑤ B^+木ファイルなどではその木の高さ

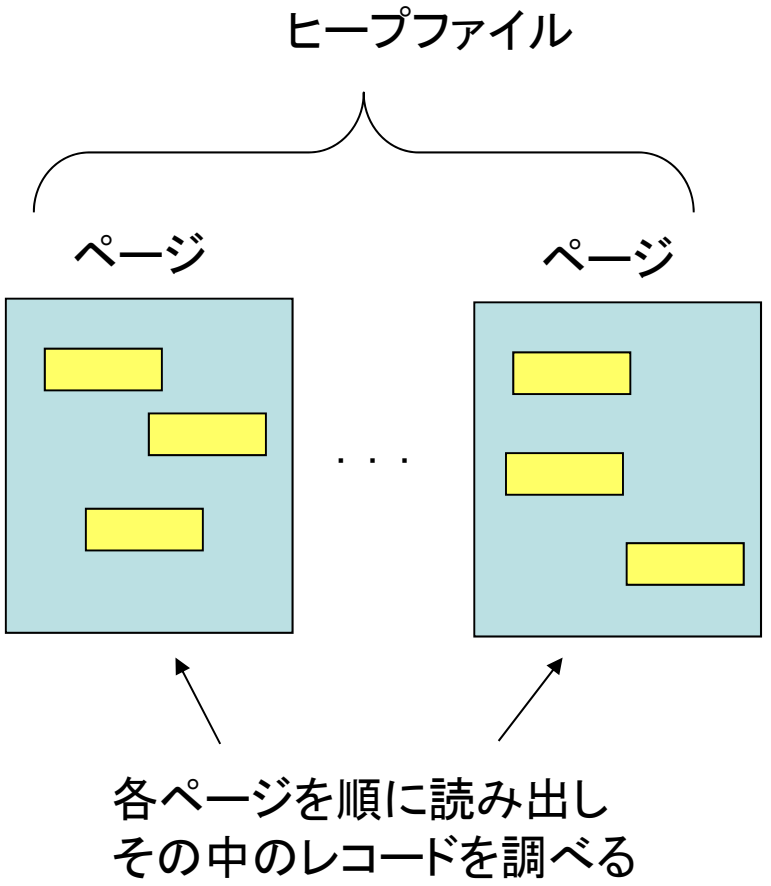
選択(1)

- 想定
  - 選択条件:$A\ \theta c$
    - 例:「年齢 = 20」,「成績 > 80」
  - データファイル S 中のデータレコード数:N
  - 選択条件を満たすデータレコードの割合:λ
    - 選択率(selectivity)と呼ばれる
    - 選択率の見積りのため,さまざまな手法が工夫されている
- 以下では典型的なコスト見積り方式を紹介
  - 教科書を一部省略

選択(2):線形探索の場合

① S がヒープファイルの場合

- S のページ数を P とする
- 基本的に全ページを探索: $COST = P$
- ただし，選択条件が $A = c$ で，$A = c$ を満たすレコードが S 中で唯一のとき，$COST = P / 2$
  - 例:「学生番号 = '00100'」
  - 半分のページを探した時点で見つかる(期待値)

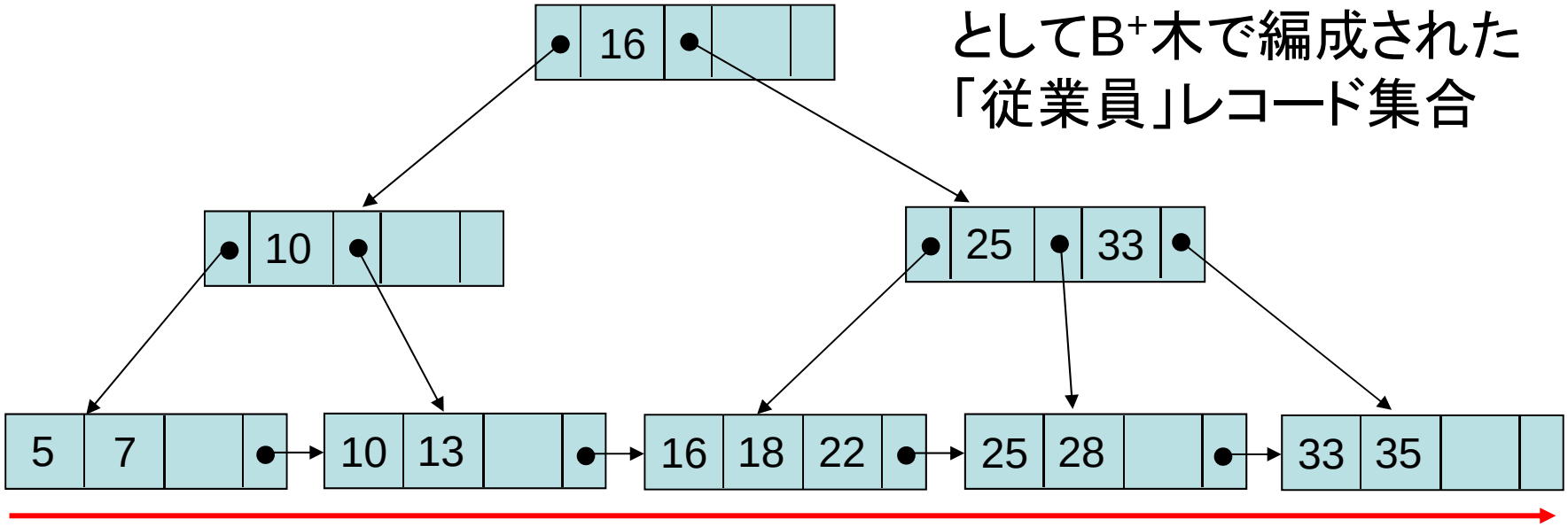


選択(3):線形探索の場合

② S が A 以外を索引フィールド付きファイルや B^+木の場合

- ①と同様の見積り

例:「従業員番号」をキーとしてB^+木で編成された「従業員」レコード集合

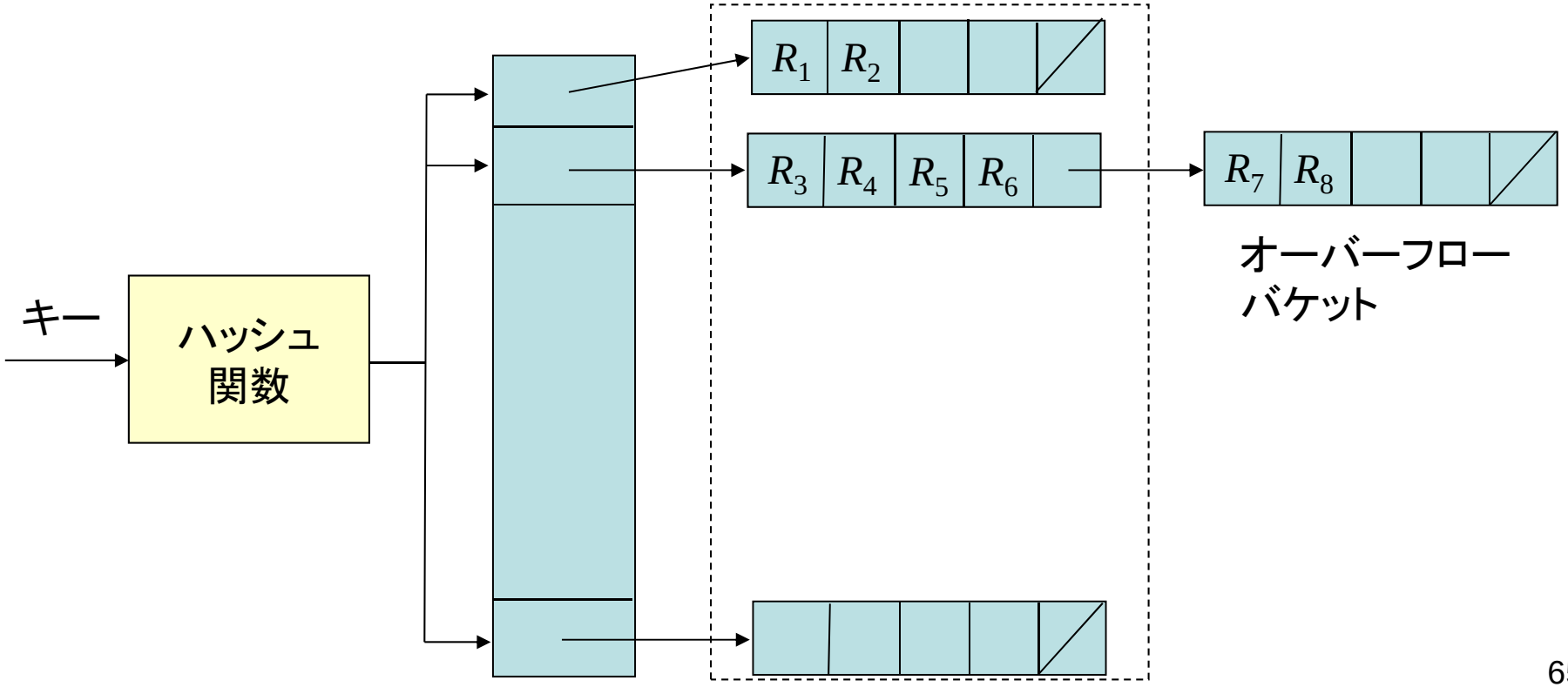


「氏名 = '山田一郎' 」という条件で検索する場合は,
データ部のページを逐次読み出し条件をチェックする

選択(4):主索引を用いた探索

① S がハッシュファイルの場合

- 選択条件 $A = c$ にのみ対応可能
- オーバーフローがないとすれば $COST = 1$



選択(5):主索引を用いた探索

② S が索引付きファイルやB$^+$木の場合

- 選択条件が $A = c$ のとき:$COST = H + 1$
  - H は主索引をたどるページアクセス回数
  - B$^+$木では木の高さ
- 選択条件が $A = c$ 以外のとき:
  $COST = H + \lambda \times P$
  - P はデータ部のページ数

結合(1)

- 結合選択率(join selectivity)を利用
  - ファイル S_1, S_2 のレコード数を N_1, N_2
  - 結合結果のレコード数を N_{12}
  - 結合選択率 $\lambda_J = N_{12} / (N_1 \times N_2)$
  - さまざまな見積り手法が工夫

結合(2):入れ子ループ結合

- 外部ループで多くの S_1 のページをまとめて読み出す方が効果的
- 設定
  - 主記憶上のバッファ領域:M ページ
  - 外部ループでの S_1 の入力用バッファ:$M-1$ ページ
  - 内部ループでの S_2 の入力用バッファ:1ページ
- コスト

$$COST = P_1 + \lceil P_1/(M-1) \rceil \cdot P_2$$

  - サイズの小さいファイルを S1 とする方が有利

問合せ処理に関するまとめ（1）

- 基本的にはディスク上のページアクセス数が処理時間に最も影響
- 中間結果を減らすことも重要
  - 選択，射影をできるだけ早く実施
  - 直積演算はできる限り避ける
- DBMSは高度な最適化を行う
  - ユーザはSQLでの高レベルの記述に集中できる
  - 基本的にはDBMSに任せる

問合せ処理に関するまとめ（2）

- パフォーマンスのチューニング
  - 索引の設定
    - 問合せで頻繁に利用される属性には適切な種類の索引を設定：大幅な効率化が可能
    - 問合せで用いられない属性に索引を設定することはオーバヘッドの増加につながる
  - 問合せの修正
    - 基本的には不要
    - 結果が重複していてもよいなら「SELECT DISTINCT～」を「SELECT ～」に修正する
  - 分析コマンドの利用
    - 多くのDBMSが分析ツールを提供：実行プランを解析できる（OracleやPostgreSQLではEXPLAINコマンド）
    - 適切な実行プランでない場合，理由を考え対応