

いまどきの索引技術

石川 佳治

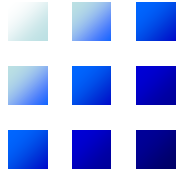
名古屋大学 情報学研究科

ishikawa@i.nagoya-u.ac.jp



あらまし

- 索引（インデックス）とは
- 記憶階層と記憶媒体
- B木
- ログ構造化マージ木
- 索引設計の指針
- さらにB木について



索引（インデックス）とは



索引（インデックス）とは

- データベースへのアクセスを高速化するためのデータ構造
 - 本における「索引」に対応
- 与えられたキー値に対し、該当するレコードやオブジェクトを検索
- データベースの問合せ処理では**圧倒的な効果**を発揮
- **アクセスメソッド**
 - データベース中のデータに対するアクセスを提供する仕組み
 - 索引を含む概念

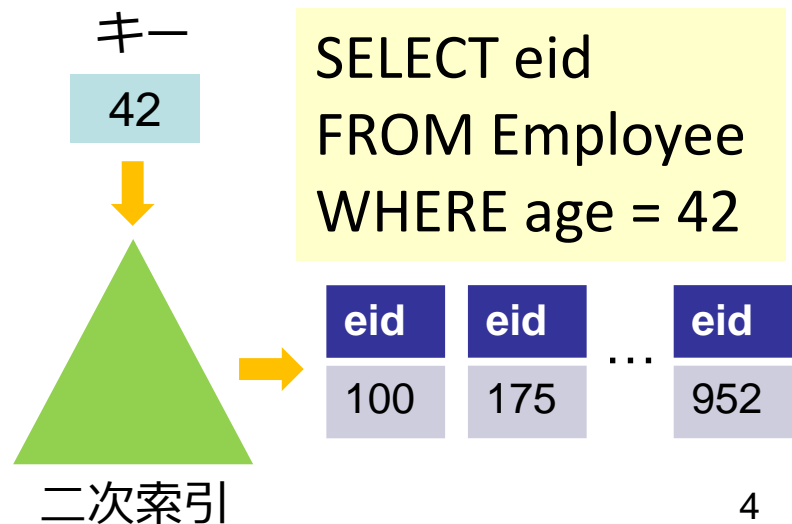
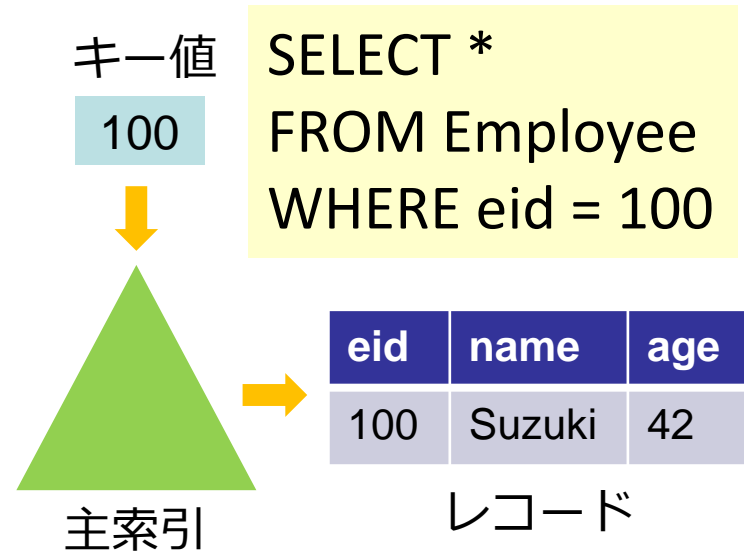
主索引と二次索引

- **主索引** (primary index)

- 主キーによる索引付け
 - レコードを一意に特定
- 与えられた主キー値に対応するレコードを返す

- **二次索引** (secondary index)

- 主索引でない索引
- 検索キーは主キーでない
- 一般にはレコードIDのリストなどを返す



索引手法の分類：伝統的手法

- ハッシュファイル*
 - ハッシュ法に基づく
 - 索引付きファイル*
 - 順次ファイル+索引部
 - 範囲検索を支援，静的データに対応
 - B木 / B+木*
 - 木構造のデータ構造
 - 範囲検索を支援，動的データに対応
 - ログ構造化マージ木 (LSM木)
 - 追記型の更新に特化した索引技術
- ・ " *"がついた手法については教科書 [1] を参照
・ 今回は赤字の手法について主に解説



索引手法の分類：空間索引

- 2次元以上の点, 矩形などを扱う索引
 - 地図データなどの検索に活用
 - マルチメディアデータへの応用
- R木
 - 矩形をキーとする木構造の索引
 - 広く利用 (例: PostgreSQL)
- 四分木 (quadtree)
 - 空間の4分割を繰り返し木構造を構築
 - 3次元の場合は八分木 (octree)
- [2] が包括的な教科書



索引手法の分類：その他の手法（1）

- ビットマップ索引*

- ビットマップに基づく
- カテゴリ属性（例：性別，部署）に適する
- 性別 = '女' AND 部署 = '人事' といった問合せに対応

- 文字列索引

- 文字列を対象とした索引
- 例：与えられた文字列が，対象のテキストデータのどこに出現するかを返す
- 接尾辞配列，ウェーブレット木，FM-indexなどの提案^[3]



索引手法の分類：その他の手法 (2)

● 高次元データの索引

- 高次元データにはR木などは不向き：**次元の呪い** (curse of dimensionality) の現象
- 近年では近似的なアプローチが主流
- **LSH (Locality Sensitive Hashing)**^[4]：近いほど衝突しやすいハッシュ関数族を利用

● フィルタ

- キー値が登録されているか否かのみを答える
- **ブルームフィルタ**^[5]
 - ハッシングに基づく確率的なデータ構造
 - **偽陽性** (false positive) の存在：登録されていないのに登録されていると判断してしまうことがある



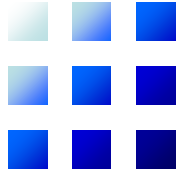
参考文献 (1)

1. 北川博之: データベースシステム (改訂2版) , オーム社, 2020.
2. H. Samet: *Foundations of Multidimensional and Metric Data Structures*, Morgan Kaufmann, 2006.
3. 定兼邦彦: 簡潔データ構造, 共立出版, 2018.
4. 入江豪: 効率的な類似画像検索のためのハッシング, 映像情報メディア学会誌, Vol. 69, No. 2, pp. 124-130, 2015. <https://doi.org/10.3169/itej.69.124>
5. B.H. Bloom: Space/Time Tradeoffs in Hash Coding with Allowable Errors, *Communications of ACM*, Vol. 13, No. 7, pp. 422-426, 1970. <https://doi.org/10.1145/362686.362692>



あらまし

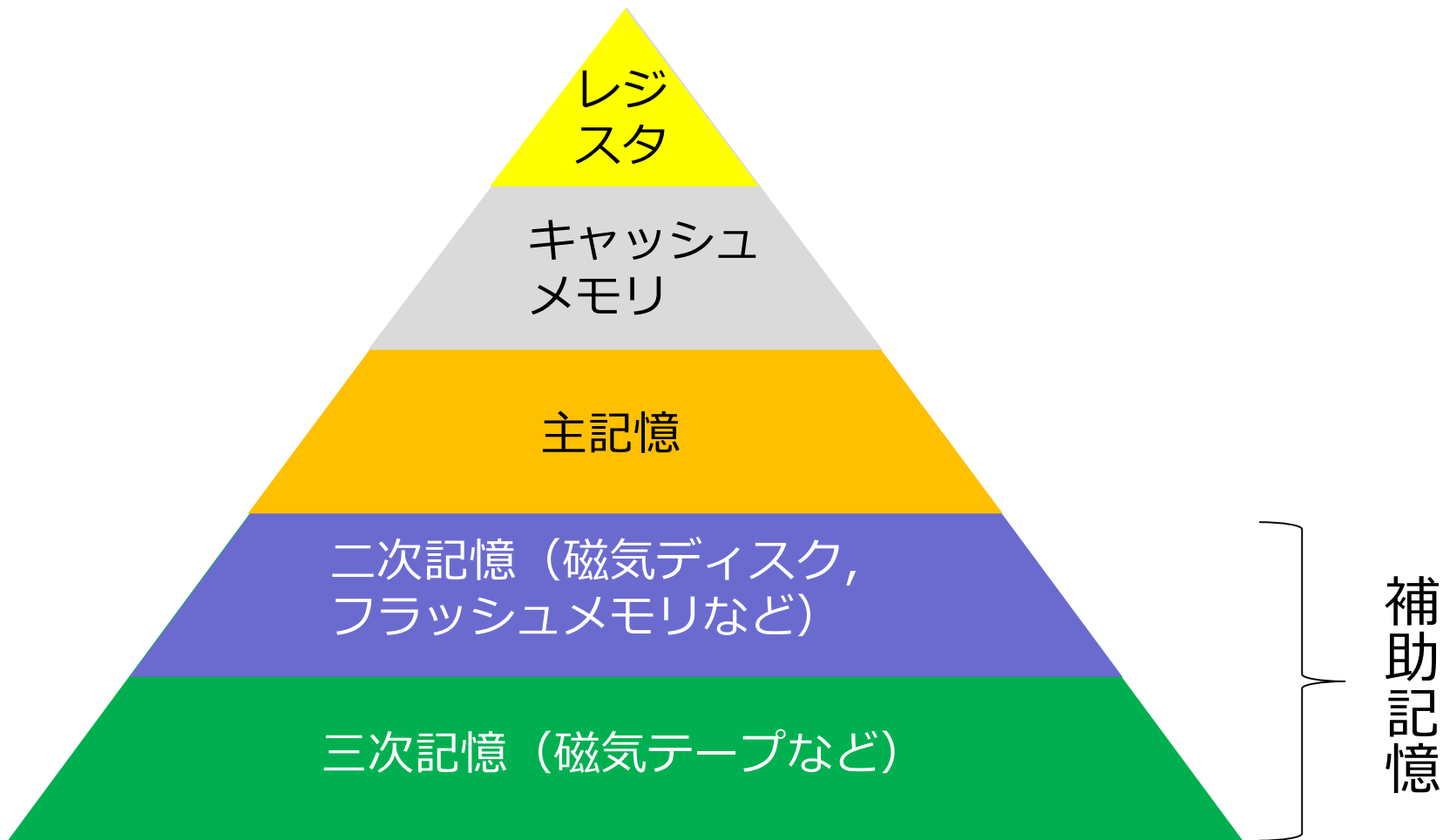
- 索引（インデックス）とは
- 記憶階層と記憶媒体
- B木
- ログ構造化マージ木
- 索引設計の指針
- さらにB木について



記憶階層と記憶媒体



記憶階層 (1)





記憶階層 (2)

- 主記憶
 - DRAM (random access memory) で実現
 - 揮発性 (volatile) : システム停止でデータ消失
- 二次記憶
 - 不揮発性 (non-volatile)
 - 磁気ディスク (hard disk)
 - SSD (solid state drive) : フラッシュメモリに基づく
- 三次記憶
 - 磁気テープ : 大容量だが, シーケンシャルアクセスのため低速

- Jim Grayの「5分間ルール」^[7]を今日のハードウェアで評価
 - コストとアクセス時間を天秤にかけた分析
 - 「5分間ルール」については教科書^[1]を参照（演習問題8.9）
- アクティブなデータは主記憶とSSDで管理
- コールドなデータはテープドライブに
- 不揮発性メモリ（NVM）は今後DRAMに近づいていく
 - NVMベースのデータベースが出てくる
- HDDの適用場面がなくなる



索引技術への影響

- 伝統的には、**主記憶 + HDD**の組合せを活かすことが重要
- 今後は**主記憶 + SSD**の組合せに移行
 - これまでの技術は活用可能
 - SSDの特性（例：小さい消去が苦手）を考慮
 - さらに**NVM上の索引**へ
- **インメモリの索引**：もう一つの選択肢
 - 二次記憶 (HDD/SSD) にアクセス不要
 - 主記憶 ⇔ 二次記憶でデータ構造の変換が不要
 - 障害対策が必要なことも
 - キャッシュヒット率向上のためには局所性必要



参考文献 (2)

6. R. Appuswamy, G. Graefe, R. Borovica-Gajic, and A. Ailamaki: The Five-Minute Rule 30 Years Later and Its Impact on the Storage Hierarchy, *Communications of ACM*, Vol. 62, No. 11, pp. 114-120, 2019. <https://doi.org/10.1145/3318163>
7. J. Gray and F. Putzolu: The 5 Minute Rule for Trading Memory for Disk Accesses and the 10 Byte Rule for Trading memory for CPU Time, *Proc. ACM SIGMOD Conference*, pp.395-398, 1987. <https://doi.org/10.1145/38713.38755>



あらまし

- 索引（インデックス）とは
- 記憶階層と記憶媒体
- **B木**
- ログ構造化マージ木
- 索引設計の指針
- さらにB木について



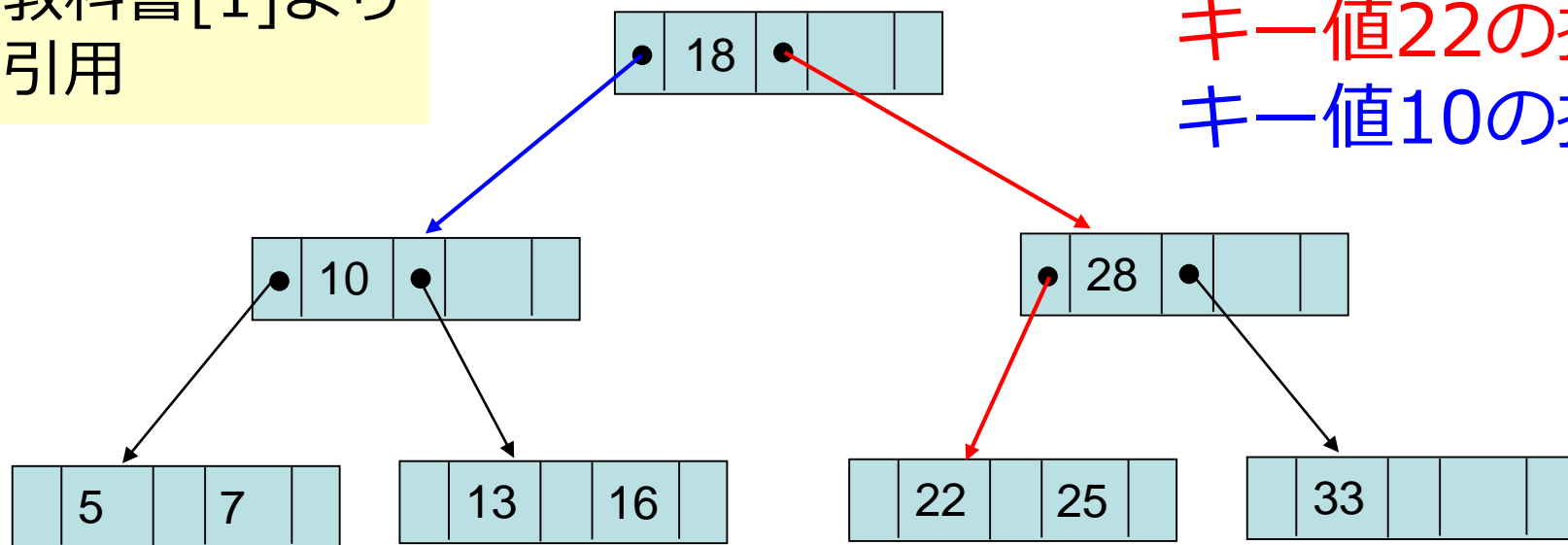
B木の概要

- BayerとMcCreightが提案^[8]
- サーベイは [9,10]に
- **DBMSで広く利用**：実際には後述の**B⁺木**
- 主記憶上のデータ構造
- バランスした多分木：ルートからリーフまでの長さ（木の高さ）は木全体で一定
- **ファンアウト**（fanout：1つの内部ノードから子ノードへの参照数）**が大きい**
 - 大量のデータでも**木の高さは小さい**： $O(\log n)$ のオーダー



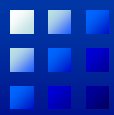
B木の構造と検索

教科書[1]より引用



各ノードは一定の充足率を満たす必要あり

- ルートから探索を行う
- 必ずしもリーフまで辿る必要はない
- 挿入・削除処理は、木のバランス構造を保つよう工夫されている



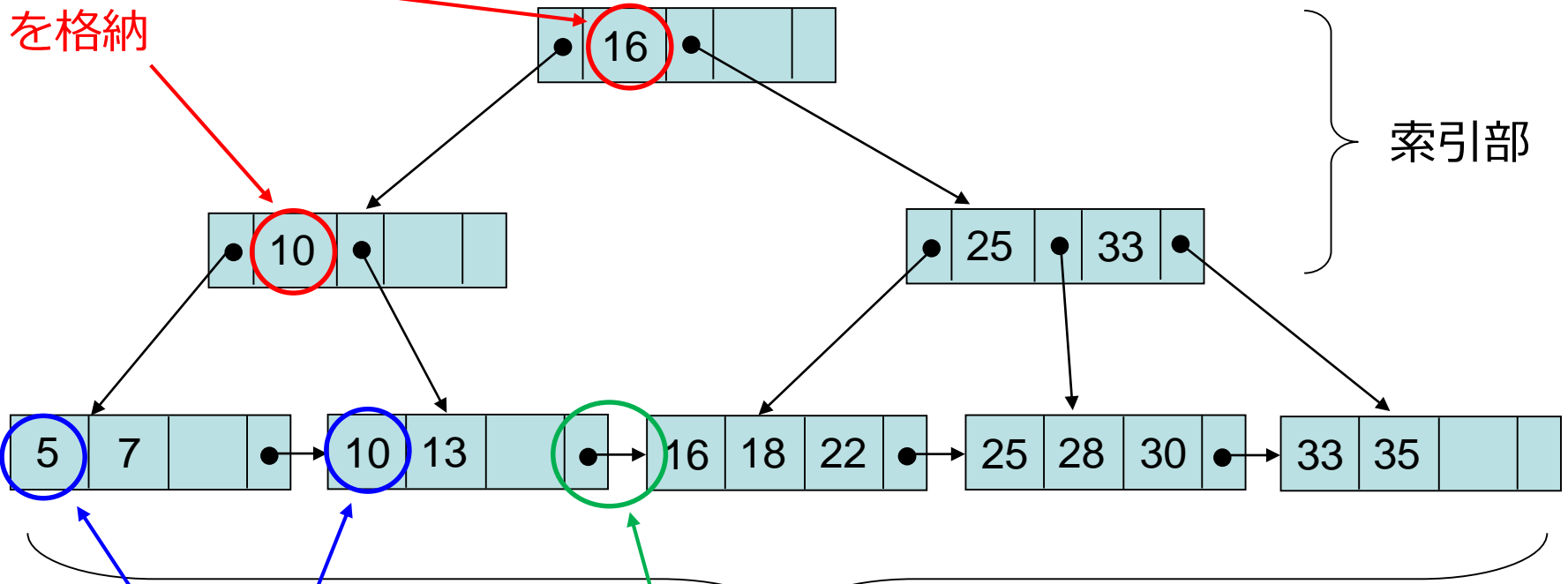
B+木の概要

- DBMSで最も用いられる
- B+木を「**B木**」と呼ぶことが多い
- B木との相違点
 - レコードは**リーフノードにのみ**格納：
データ部に相当
 - 非リーフノードには**キー値のみ**を格納：
索引部に相当
- **二次記憶に対応**した構造
- 索引部のファンアウトが大きい



B+木の構造

索引部には
キー値のみ
を格納



索引部

データ部にはレコードを
格納 (主索引の場合)

データ部

リーフノードを横に
つなぐリンクが存在
(双方向の場合も)



B+木における検索

- キー値による検索

- ルートを最初のノードとし, キー値に応じて子ノードポインタをリーフノードまでたどる (B木との相違点)

- キー値順での読み出し (順次スキャン)

- データ部のリーフノードを順次読み出す

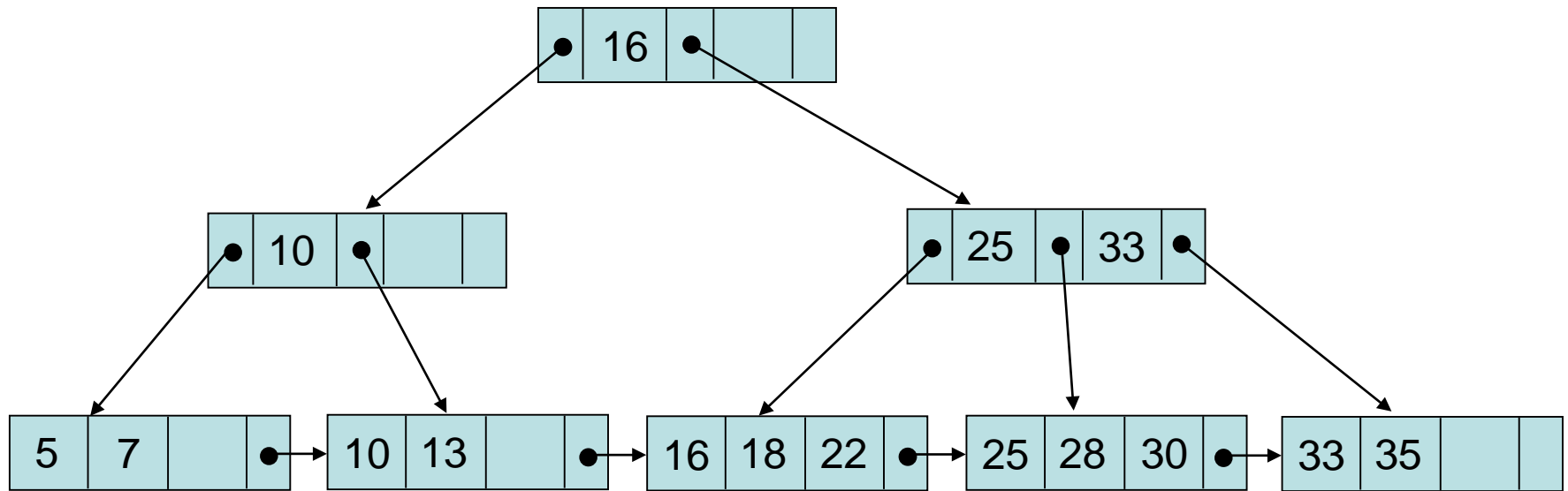
- 範囲検索

- 索引部を用いて先頭リーフノードを特定
- そこから横方向のポインタをたどって範囲を超えるまでリーフノードを読む

- 順次スキャン, 範囲検索が効率的

B+木におけるレコード挿入 (1)

- キー値31の挿入の場合

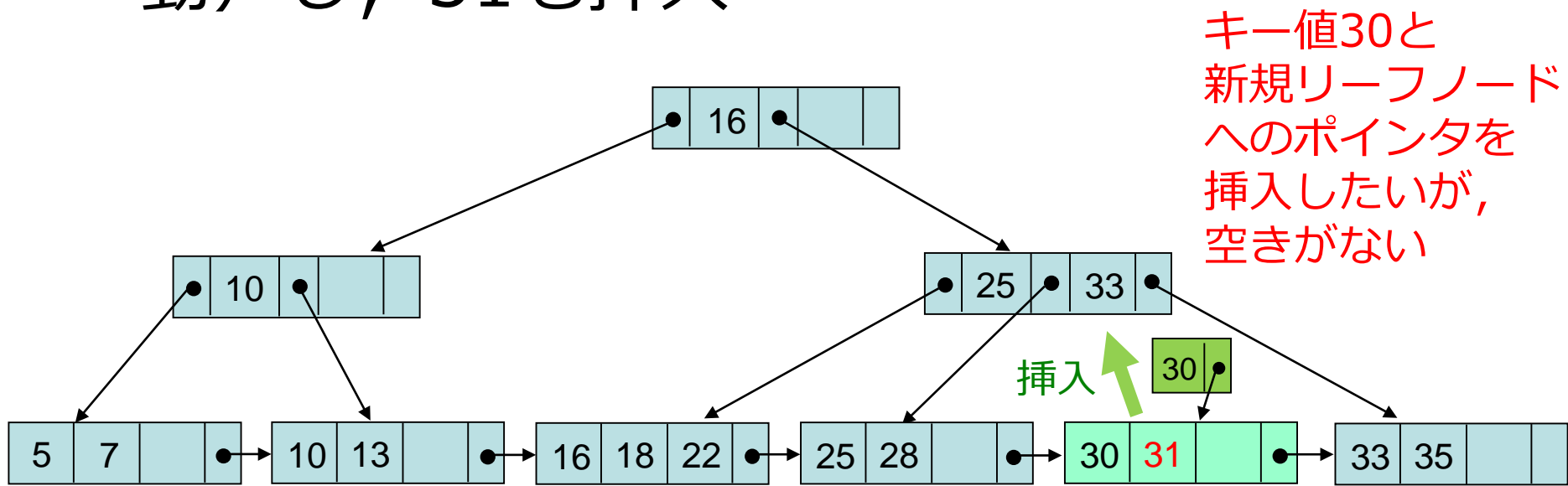


このリーフノードに
挿入したいが
オーバーフロー発生

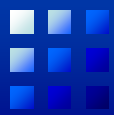


B+木におけるレコード挿入 (2)

- 新たにリーフノードを割り当てし, オーバーフローノードのキーを分配 (30を移動) し, 31も挿入

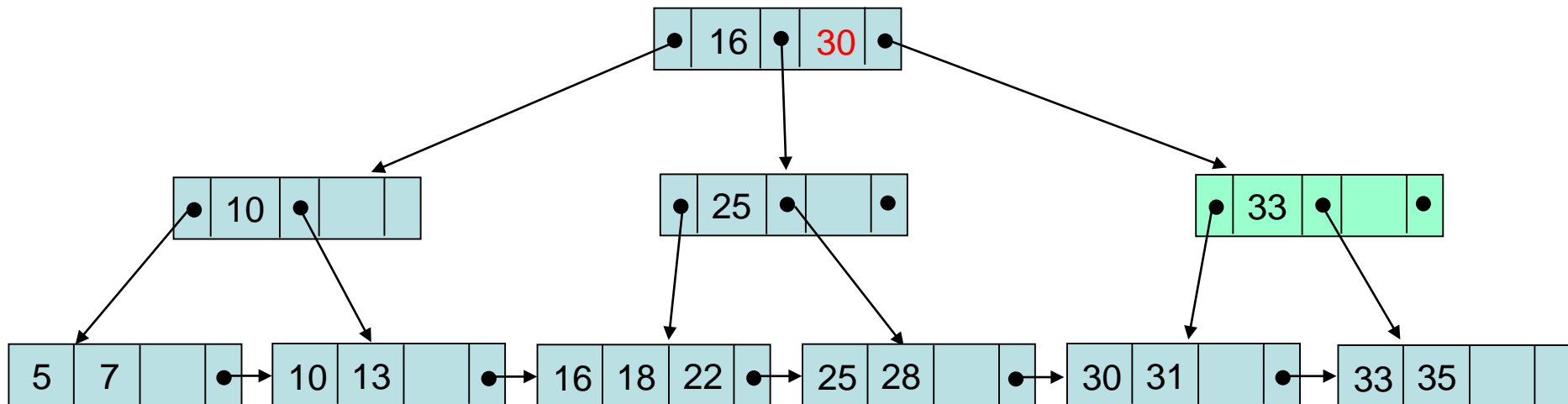


- キー値30と新リーフノードへのポインタを親ノードに挿入: 再びオーバーフロー



B+木におけるレコード挿入 (3)

- 新たに内部ノードを割り当て、既存のキー値を振り分ける
- キー値30は25と33の間なので、さらに親ノードに新内部ノードへのポインタとともに挿入





B+木アルゴリズムの工夫

● リバランシング

- 利用率の高いノードから低いノードへ要素を移動する処理
- 挿入・削除の手順に含まれることが多い

● 最右ノードへの追加

- DBMS利用では、主キーがインクリメンタルに増加することが多い：右端への挿入が続く
- 最右ノードへの追加を工夫した手法が存在

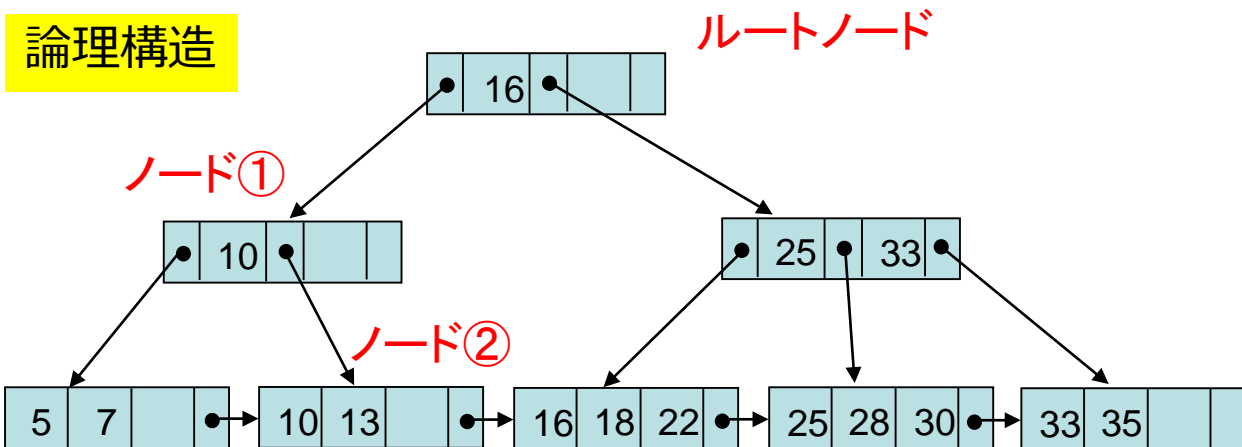
● バルクロード

- 一気にB+木を構築：リーフノードをまず作って、その後で索引部を構築



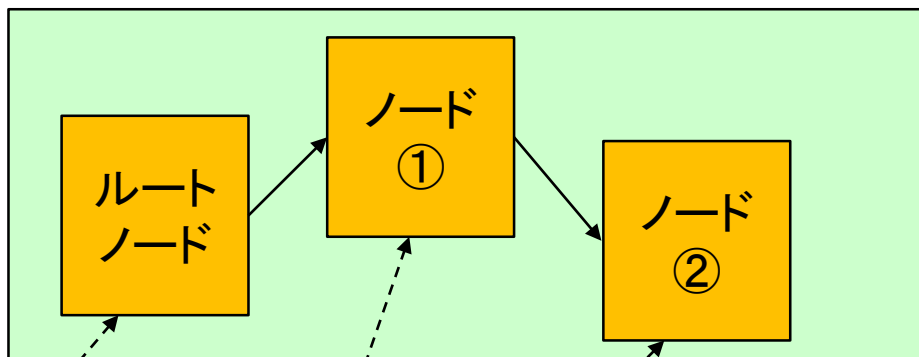
B+木の実装例

論理構造

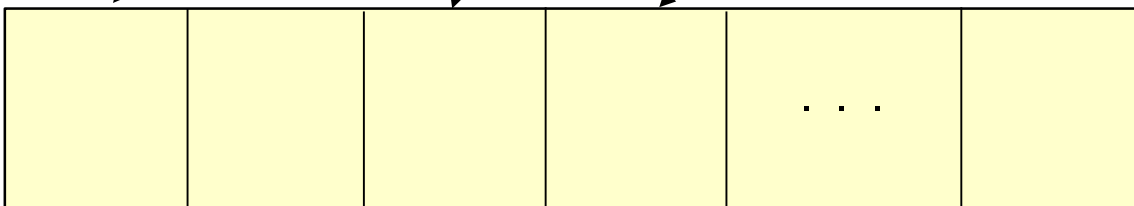


物理構造

主記憶上のバッファ



二次記憶上のファイル



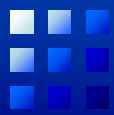
- 二次記憶上のファイルはページ（例：4KB）で構成
- 各ページがノードに対応
- ノードへのアクセス時には、該当するページを読み込み、主記憶上の表現に変換
- B+木に更新が発生したらページを書き込み
- ルートノードおよび上位ノードはしばしば主記憶上に常駐



B+木の高さの計算例

- パラメータ
 - キーのサイズ：4バイト
 - レコード全体のサイズ：256バイト
 - レコード数：10万件
 - ページサイズ：4KB
 - ノード間のポインタ：4バイト
- 内部ノードのファンアウト：256
- B+木の高さは2となる
- 大規模なデータでも木の高さ（検索コスト）は小さい

教科書[1]の
問題8.9より



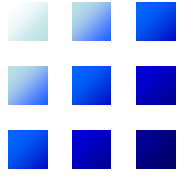
参考文献 (3)

8. R. Bayer and E.M. McCreight: Organization and Maintenance of Large Ordered Indices, *Acta Informatica*, Vol. 1, pp. 173-189, 1972.
<https://doi.org/10.1007/BF00288683>
9. D. Comer: The Ubiquitous B-Tree, *ACM Computing Surveys*, Vol.11, No.2, pp.121-137, 1979. <https://doi.org/10.1145/356770.356776>
邦訳：広くゆきわたったB木, bit別冊 acm computing surveys '79 コンピュータ・サイエンス, 共立出版, 1981.
10. G. Graefe: Modern B-Tree Techniques, *Foundations and Trends in Databases*, Vol. 3, No. 4, pp. 203-402, 2011.
[B木の実装技術について詳しい]
<https://doi.org/10.1561/19000000028>
11. A. Petrov: *Database Internals*, O'Reilly, 2019.
邦訳：詳説 データベース, オライリー・ジャパン, 2021.
[B木, LSM木, Bw木の詳しい説明あり]



あらまし

- 索引（インデックス）とは
- 記憶階層と記憶媒体
- B木
- ログ構造化マージ木
- 索引設計の指針
- さらにB木について



ログ構造化マージ木 (LSM木)



索引更新のアプローチ

- 索引に対する2種類の更新のアプローチ
 - 直接的な更新：B+木などでとられる，データ値の直接の更新
 - 追記型の更新：ログのような方式で更新情報を追記
- プログラミング言語の概念との関連
 - ミュータブル (mutable)：変更可能の変数・データ型
 - イミュータブル (immutable)：変更不可の変数・データ型
 - 追記型の更新では，過去のデータは変更不可



追記型の索引

- ログのような形で，索引に更新データを追記
- 書き込みのための最適化
- 検索時のオーバヘッドは大：冗長な情報の調停 (reconciliation) が行われる
- ログ構造化マージ木 (LSM木)
 - Log-Structured Merge Tree
 - [12] で提案. その後, さまざまな発展
 - 索引木をログ構造で蓄積管理し, 適宜マージしていくことから命名
 - 単一の木構造をとるわけではない



追記型の更新

- さまざまな状況で発生
- 例：履歴データ（例：売上データ）
 - ログの形で追加されるのみ
 - 過去のデータを更新することは（ほぼ）ない
 - 検索の頻度は書き込みに対し少ない
- 例：ビッグデータ処理
 - Hadoopなどにおけるバッチ処理など
 - 一気にデータの追加・更新が発生
- 追記型の更新に特化した索引が有効



参考：ログ構造化ファイルシステム

- Log-Structured File System
- 例：LFS, NILFS, F2FS
- ファイルシステムへの更新を**追記**の形で処理. 上書きしない
- **書き込み処理に特化**
- ファイルシステムの一貫性の管理が容易
- 古くなったログを適宜回収し, 空き容量を作る作業が必要
- **SSD (フラッシュメモリ) の特性に合致**
 - ランダムな小さい書き込みはコスト大



基本的なアイデア : memtable (1)

- データの追加が発生すると、主記憶上のバッファ (memtable) に追記
- **memtable**
 - 主記憶上のソートされたデータ構造
 - (key, value) 形式のデータを蓄積
 - 検索支援のための索引 (B木, 赤黒木, AVL木など) が伴う
 - 値を削除する際は削除エントリ (tombstone) を明示的に記録
- 次ページで実現法の一例を説明



基本的なアイデア : memtable (2)

主記憶

主記憶上の
索引 (例: B木)

memtable

Key	Value
5	Apple
30	Peach
70	Banana
90	Orange

更新

memtable

Key	Value
5	Apple
20	<DELETE>
30	Peach
40	Grape
70	Mango

削除
エントリ
を挿入

挿入

上書き

追記
データ

Key	Value
40	Grape
20	<DELETE>
90	<DELETE>
70	Mango

キー値40の追加
キー値20, 90の削除
キー値70の追加
(実際は上書き)

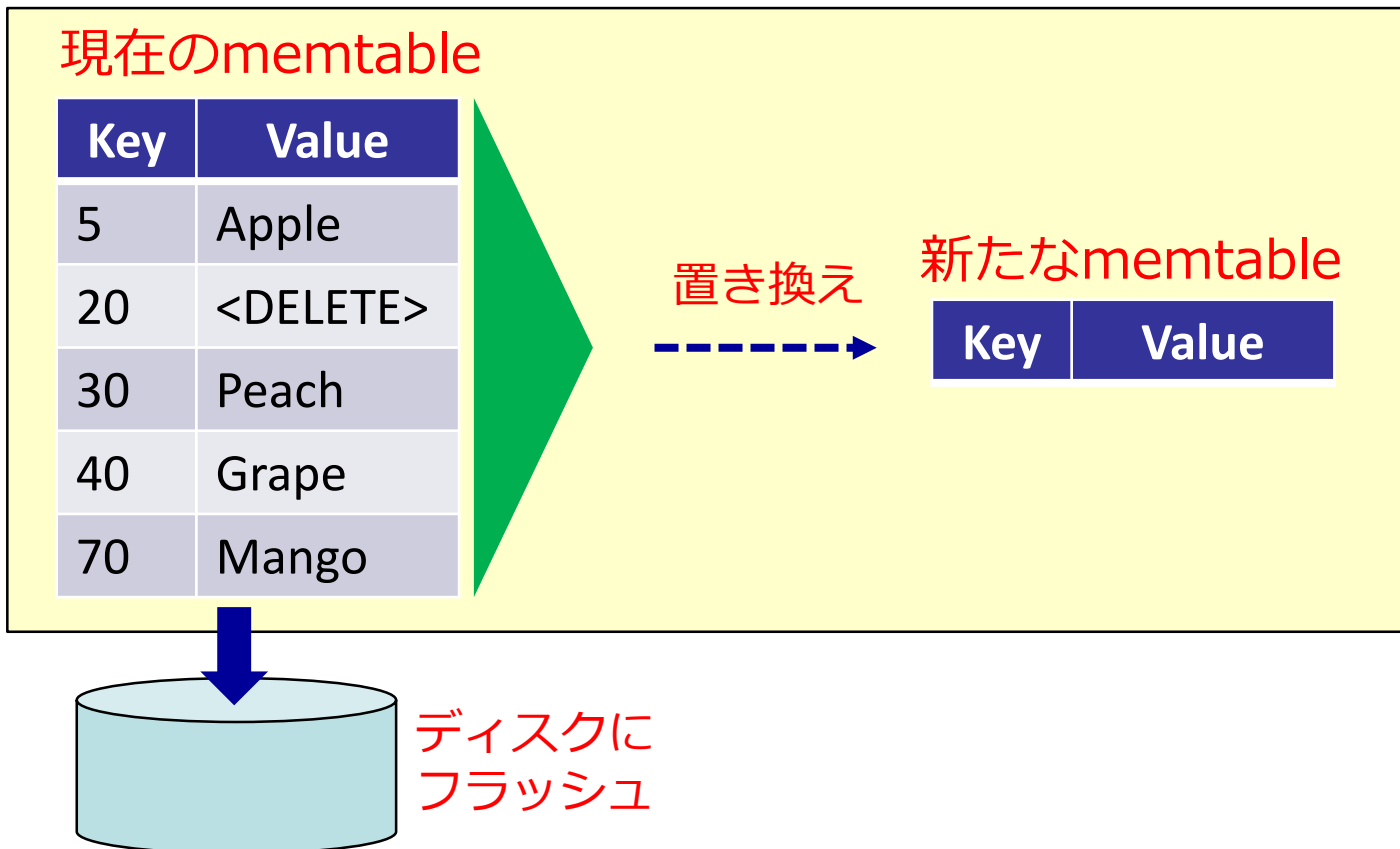
主記憶上の修正は可能:

- キー値70については
値を上書き
- キー値90については
その場で削除



基本的なアイデア : memtable (3)

- memtableのサイズが閾値を超えると, その内容をディスクにフラッシュ
- 新たな memtable に置き換え




基本的なアイデア : SSTable (1)

- **SSTable** (Sorted String Table)
 - ディスク上のテーブル : 読み取り専用
 - B木などで管理
 - バルクロードされ100%の充足率
 - 索引部は主記憶に置かれることも

ディスク


SSTable at $t = 25$

Key	Value
5	Apple
20	<DELETE>
30	Peach
40	Grape
70	Mango




SSTable at $t = 15$

Key	Value
10	Plum
30	Chestnut
40	<DELETE>
50	Cherry
60	Mango



SSTable at $t = 10$


Key	Value
0	Lemon
20	Pear
40	Melon
60	Kiwi
80	Pineapple



基本的なアイデア：検索処理 (1)

- 検索処理では**調停** (reconciliation) が必要


Key	Value
10	Apricot
60	<DELETE>
90	Muscat



例：キー値10, 30, 50, 60, 80での検索では、○を付けたエントリにマッチ


SSTable at $t = 25$

Key	Value
5	Apple
20	<DELETE>
30	Peach
40	Grape
70	Mango




SSTable at $t = 15$

Key	Value
10	Plum
30	Chestnut
40	<DELETE>
50	Cherry
60	Mango



SSTable at $t = 10$


Key	Value
0	Lemon
20	Pear
40	Melon
60	Kiwi
80	Pineapple



基本的なアイデア：検索処理 (2)

- 範囲検索では複数のイテレータを使用

Key	Value
10	Apricot
60	<DELETE>
90	Muscat




例： $10 \leq \text{key} \leq 50$ の範囲検索

- ・各SSTableがソート済という性質を利用
- ・SSTableごとにイテレータを設定し、同期して読み出し（マージソートの的）
- ・ディスクのシーク数 = SSTableの数


SSTable at $t = 25$

Key	Value
5	Apple
20	<DELETE>
30	Peach
40	Grape
70	Mango




SSTable at $t = 15$

Key	Value
10	Plum
30	Chestnut
40	<DELETE>
50	Cherry
60	Mango



SSTable at $t = 10$

Key	Value
0	Lemon
20	Pear
40	Melon
60	Kiwi
80	Pineapple





基本的なアイデア：コンパクション

● 複数のテーブルをマージ

SSTable at $t = 15$

Key	Value
10	Plum
30	Chestnut
40	<DELETE>
50	Cherry
60	Mango

SSTable at $t = 10$

Key	Value
0	Lemon
20	Pear
40	Melon
60	Kiwi
80	Pineapple

新たなSSTable

Key	Value
0	Lemon
20	Pear
50	Cherry
60	Mango
80	Pineapple

- ・ 何らかの基準をもとに起動
- ・ ファイル数・全体のサイズを減らす
- ・ マージソートに似た処理で実現
- ・ 古いSSTableは削除
- ・ さまざまなコンパクションのアプローチが存在 [11]



LSM木の利用

- Google Bigtable
- Apache HBase^[13]
- Apache Cassandra^[14]
- LevelDB
- RocksDB
- InfluxDB
- SQLite4
- Apache AsterixDB

ビッグデータのための
NoSQL/key-valueストア
での実装例多し



参考文献 (4)

12. P. O’Neil and E. Chang, D. Gawlick, E. O’Neil: The Log-structured Merge-tree (LSM-tree), *Acta Informatica*, Vol. 33, No. 4, pp. 351-385, 1996.
<https://doi.org/10.1007/s002360050048>
13. L. George: *HBase: The Definitive Guide*, O’Reilly, 2011.
邦訳 : HBase, オライリー・ジャパン, 2012.
14. J. Carpenter and E. Hewitt: *Cassandra: The Definitive Guide*, 3rd ed., O’Reilly, 2020. 邦訳 (第1版) E. Hewitt: *Cassandra*, オライリー・ジャパン, 2011.
15. M. Kleppmann: *Designing Data-Intensive Applications*, O’Reilly, 2017.
邦訳 : データ指向アプリケーションデザイン, オライリー・ジャパン, 2019.
[LSM木についての言及あり]



あらまし

- 索引（インデックス）とは
- 記憶階層と記憶媒体
- B木
- ログ構造化マージ木
- **索引設計の指針**
- さらにB木について



RUM予想 (The RUM Conjecture)

- [16]における提案 (証明があるわけではない)
- アクセスメソッドの設計では, **読み出し (Read)**, **更新 (Update)**, **メモリ (ストレージも含む) (Memory)** オーバヘッドを最小化したい
- **RUM予想**: これらのうち2つを最適化すると残り1つが犠牲になる
 - 3つ全部を最適化することはできない
 - 状況に合わせた選択が必要



3つの観点からの分類

Read Optimized

ハッシュ

点 & 木構造
索引

B木 トライ
スキップリスト

クラッキング

適応的な構造

適応的マージ

PDT

LSM

PBT

差分による
構造

MaSM

疎な索引

ブルームフィルタ

ビットマップ 近似的な
索引

Write Optimized

Space Optimized



補足：手法の説明

- **トライ (trie)**：文字列検索などにつかわれる木構造のデータ構造
[Wikipedia: トライ\(データ構造\)](#)
- **スキップリスト (skip list)**：連結リストによる乱択アルゴリズムのデータ構造
[Wikipedia: スキップリスト](#)
- **疎な索引 (sparse index)**：伝統的には、すべてのキー値が出現しない索引を指す^[1]が、ここでは、最近の軽量な二次索引手法を意図。
- **クラッキング (および適応的マージ)**：検索の発生のために徐々にデータ構造を構築する手法
- 他はマイナーなので省略



B+木とLSM木の比較

- B+木：読み取り効率を重視
 - キー値に対応するレコードの場所は一意
 - データ構造の「その場での」更新
- LSM木：書き込み効率を重視
 - 追記型の更新
 - 検索では調停が必要

N : タプル数
 m : 問合せ結果数
 B : ブロックサイズ
 T : LSMのlevel ratio
 M : メモリサイズ

計算量の比較^[16]

	作成コスト (バルク挿入)	索引 サイズ	点問合せ	範囲問合せ	更新
B+木	$O\left(\frac{N}{B} \cdot \log_{\frac{MEM}{B}}\left(\frac{N}{B}\right)\right)$	$O\left(\frac{N}{B}\right)$	$O(\log_B N)$	$O(\log_B N + m)$	$O(\log_B N)$
Levelled LSM ^[17]	N/A	$O\left(\frac{N \cdot T}{T-1}\right)$	$O\left(\log_T \frac{N}{B} \cdot \log_B N\right)$	$O\left(\log_T \frac{N}{B} \cdot \log_B N + \frac{m \cdot T}{T-1}\right)$	$O\left(\frac{T}{B} \cdot \log_T \frac{N}{B}\right)$



参考文献 (5)

16. M. Athanassoulis, M.S. Kester, L.M. Maas, R. Stoica, S. Idreos, A. Ailamaki, M. Callaghan: Designing Access Methods: The RUM Conjecture, *Proc. EDBT Conference*, pp. 461-466, 2016.
<http://openproceedings.org/2016/conf/edbt/paper-12.pdf>
17. B.C. Kuzmaul: A Comparison of Fractal Trees to Log-Structured Merge (LSM) Trees. *Tokutek White Paper*, 2014.



あらまし

- 索引（インデックス）とは
- 記憶階層と記憶媒体
- B木
- ログ構造化マージ木
- 索引設計の指針
- さらにB木について

- **書き込みの増幅** (write amplification)
 - 更新頻度が高い場合、二次記憶上のページの更新が毎回必要なことも
- **利用領域の増幅** (space amplification)
 - 使用領域に無駄がある：ただし、最悪50%、通常70%程度の充足率なので重大な問題ではない
- **並行した更新・アクセスへの対応**
 - 複数の同時更新・アクセスを効率的に処理
 - 「その場更新」のB+木では制御が複雑になる
 - マルチコア、マルチスレッド化がさらに進むことによりより深刻に

書き込みの増幅への対応^[11]

- 多くのアプローチが存在
 - 基本的なアイデア：複数の書き込み処理を1回にまとめることで効率化
- 遅延B⁺木 (Lazy B⁺-Tree)
 - 一つの手法というよりもアプローチの総称
 - 更新発生時に、その都度B⁺木の構造を更新するのではなく、更新内容を追記でバッファに蓄積
 - B⁺木の検索時には、元のB⁺木の内容とバッファの内容を調停して結果を返す
 - 更新バッファがいっぱいになるなどのタイミングで木の構造を変更



B+木の並行処理制御

- 2つの要因で生じる
- データベースのトランザクション
 - 例：トランザクションAがキー値200のレコードを挿入と同時に、トランザクションBが [100, 300]の範囲で検索
- 複数スレッドによる更新・アクセス
 - 例：あるアプリでキー値100とキー値200の2レコードを挿入、それぞれが別スレッドで動く
 - 前者との違い：あるスレッドが別スレッドの内容を見てよい。ただし、データ構造の変更などのクリティカルセクションには排他制御が必要



余談：ロック vs ラッチ

- 一般に，並行処理制御に**ロック** (lock) および **ラッチ** (latch) が用いられる
- 使い分けは曖昧：以下の指針あり^[10]

	ロック	ラッチ
何を隔離？	トランザクション	スレッド
何を守る？	データベースの内容	インメモリのデータ構造
期間	トランザクション全体	クリティカルセクション
モード	共有，排他など	read, writeなど
デッドロック	検出 & 解消	回避
デッドロック検出	待ちグラフ，タイムアウトなど	コーディングでの工夫など
情報の保持	ロックマネージャのハッシュ表	保護されたデータ構造



B+木のロック処理 (1)

- 単純なプロトコルの例
 - 以下は木ロックングプロトコル^[1]に基づく
 - 実際には, B+木のさまざまな操作に対し多くのプロトコルが存在
- 検索処理
 - ルートから順に進める: まずはルートノードへのS (共有) ロックを取得することを試みる
 - 子ノードにSロックが取れたら, 親ノードのSロックをアンロック
 - リーフノードのSロックが取れたら, その内容をreadし, その後アンロック



B+木のロック処理 (2)

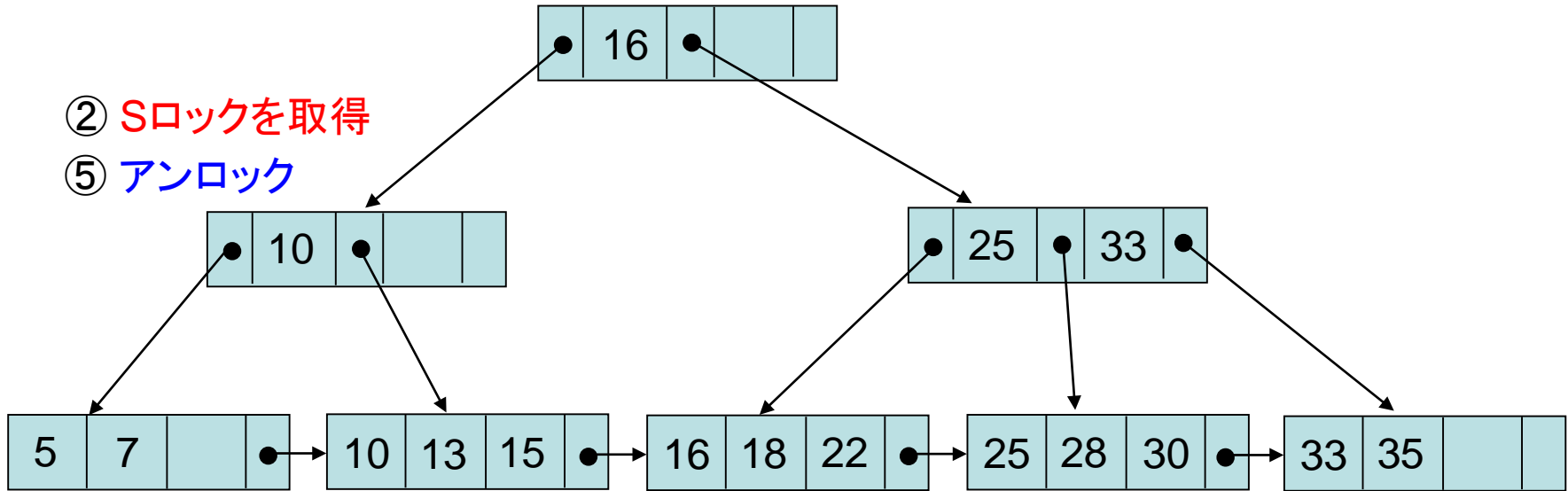
例：キー値13の検索：数字の順で処理

① Sロックを取得

③ アンロック

② Sロックを取得

⑤ アンロック



④ Sロックを取得

⑥ ノードを read

⑦ アンロック



B+木のロック処理 (3)

- 挿入処理

- ルートから順に進める：まずはルートノードへのX (占有) ロックを取得することを試みる
- 子ノードにXロックが取れたら、まず、子ノードが「安全」であるかをチェック
- 安全であれば親ノードのXロックをアンロック

- 「安全」とは：オーバフロー（削除の場合はアンダーフロー）が生じない

- 検索の場合と異なり、最終的にリーフノードのみがロックされるとは限らない

- ロックについて、詳しくは[18-21]を参照

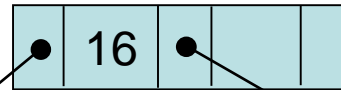


B+木のロック処理 (4)

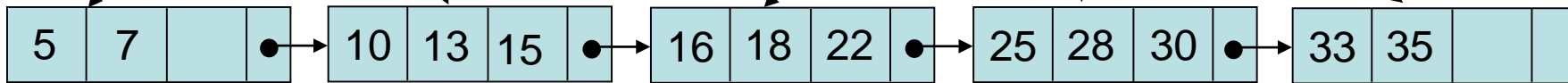
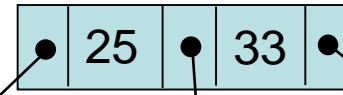
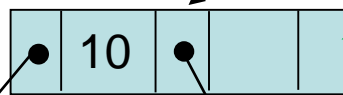
例：キー値14の挿入

- ② **Xロックを取得**：このノードは安全でないの
ですぐにアンロック不可
- ⑦ 子ノードからキー値と
子ノードポインタを
受け取り挿入
- ⑧ ノードをwriteし、
アンロック

- ① **Xロックを取得**
- ③ **アンロック**



このノードは、子ノードに
オーバーフローが発生すると
更新が発生するので、
すぐにはアンロックできない



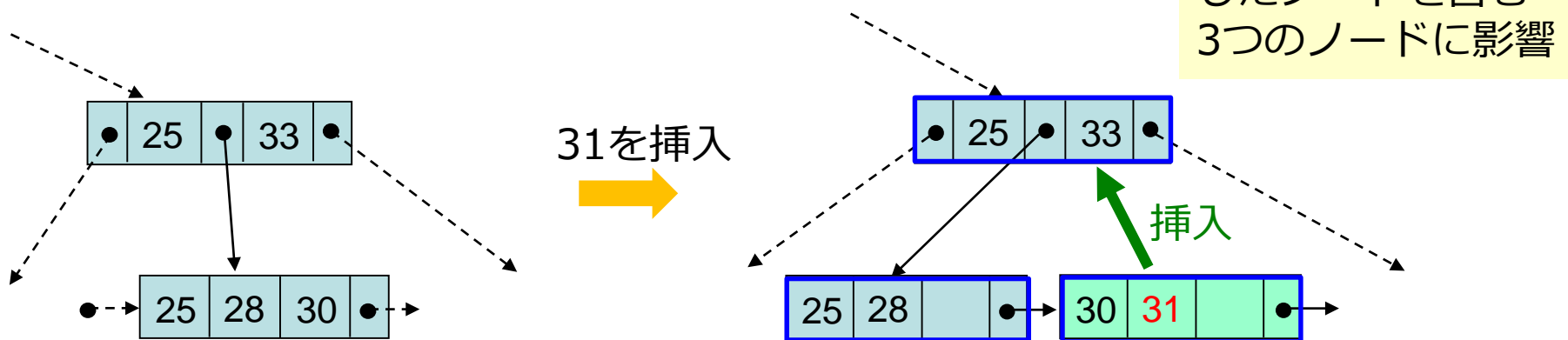
- ④ **Xロックを取得**
- ⑤ ノードをreadし、新たなノードを
割り当て、レコードを分配：(10, 13) と (14, 15)
- ⑥ 両ノードを write し**アンロック**. (14, 新リーフノードへの
ポインタ) を親ノードに挿入



B^link木 (1)

● B⁺木の問題点

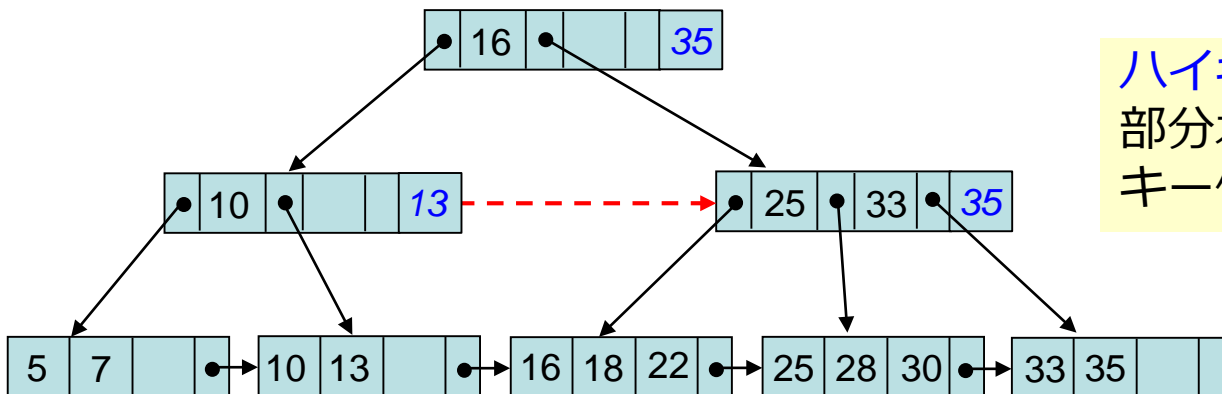
- あるノードへの挿入は, 少なくとも3つのノードに影響
 - オーバフローしたノード
 - 新たに割り当てたノード
 - 親ノード
- さらに祖先にも影響しうる





Blink木 (2)

- [22]で提案：ノード分割を**独立した2ステップ**に分ける ⇒ **並行性の向上**
 - ① 新ノードを割り当てし, レコードを振り分け
 - ② 親ノードに (キー値, 新ノードへのポインタ) を挿入
- 各内部ノードに**右兄弟へのリンク**を設ける
- 内部ノードに**ハイキー** (high key) を格納



ハイキー：現在のノードの部分木における最大のキー値

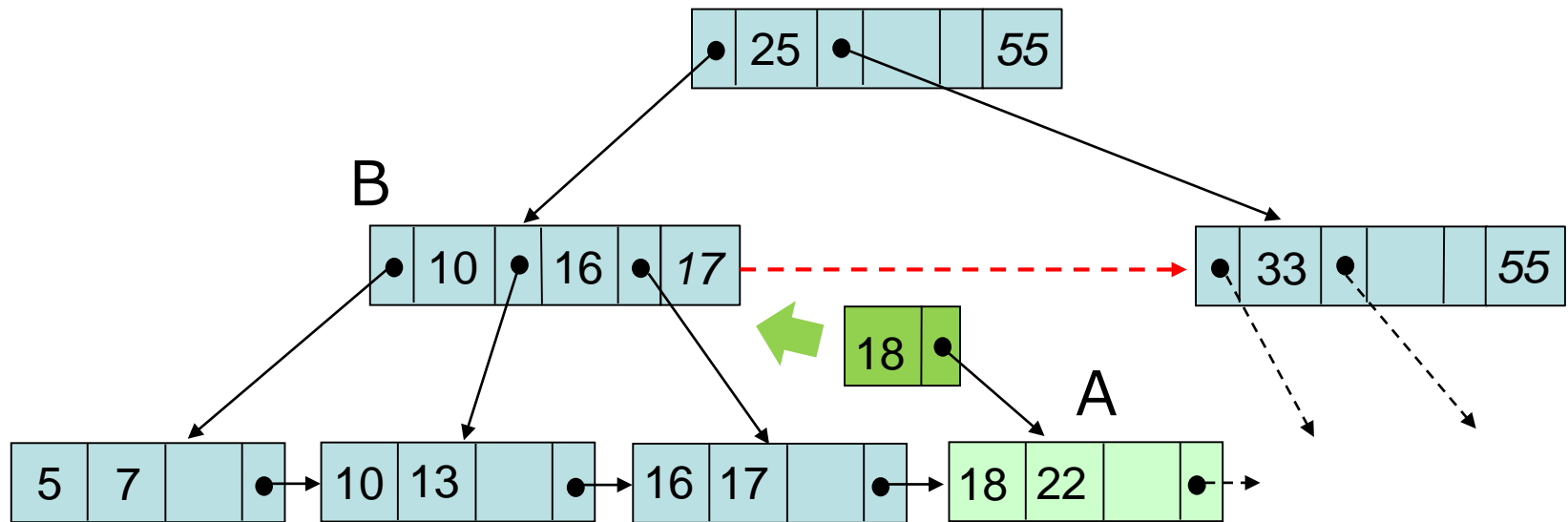
Blink木は
PostgreSQLで使用



Blink木のアイデア (1)

- 想定する状況

- リーフノードへの挿入が発生：ノードAを作成
- 親ノードBに (18, ノードAへのポインタ) 挿入

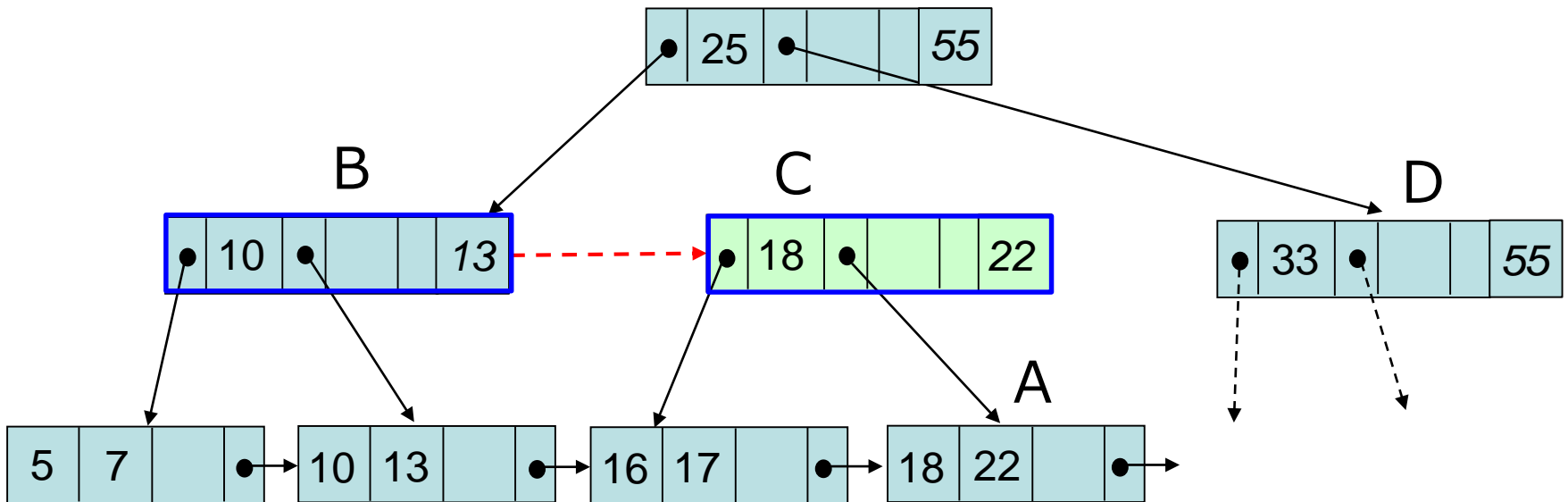


- ここまではB⁺木と同じ



Blink木のアイデア (2)

- ステップ1：ノードCを割り当てて分配を行う

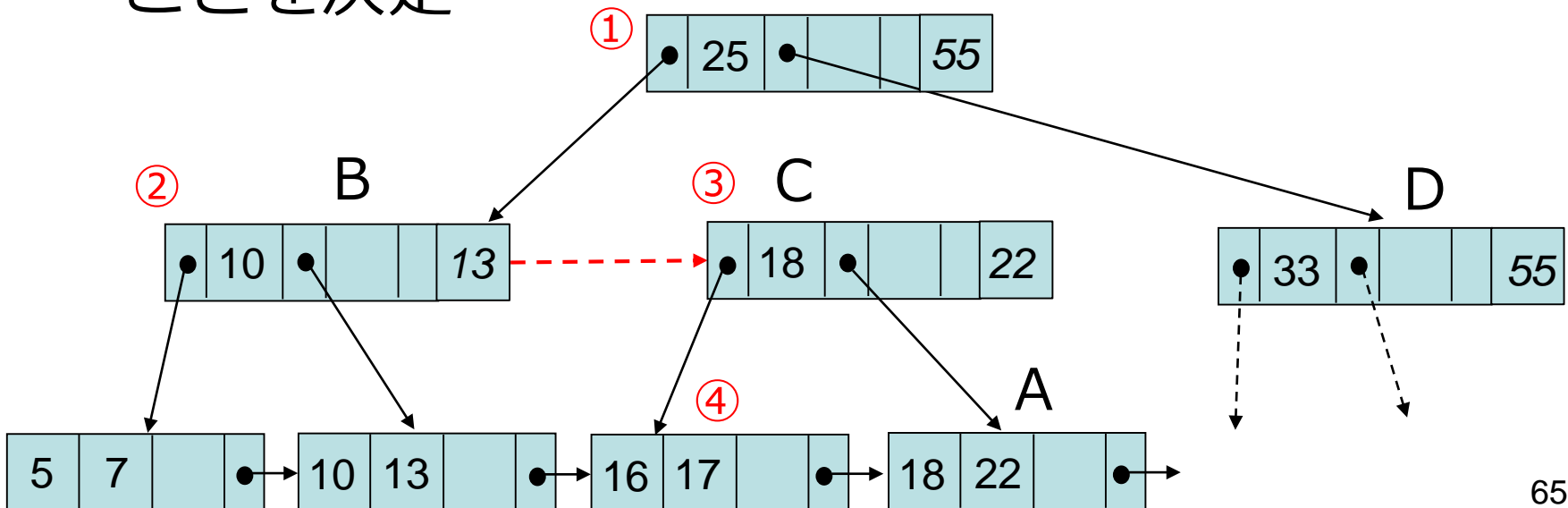


- ノードB, Cにラッチをかけて処理
- この間, Bをルートとする部分木にはアクセスできないが, ルートノードやノードDにはアクセス可能



Blink木のアイデア (3)

- ステップ2 (ルートに16とノードCへのポインタを挿入) は**適当なタイミング**で実施
- **検索処理ではリンクを考慮**
 - キー値17の検索では①~④の順でアクセス
 - ノードBでハイキー13を見て, ノードCに進むことを決定



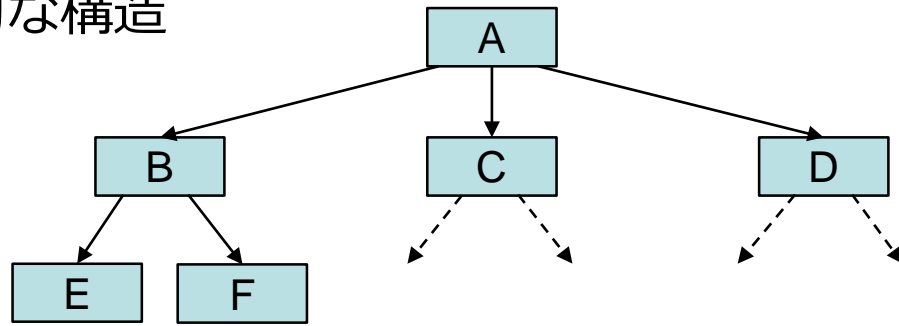
Bw木^[23] : ラッチフリーのB木

- 「バズワード (buzz word) 木」 とのこと
- 想定する対象 : インメモリOLTP
- マイクロソフトが開発 : SQL Server Hekatonで使用
- 並行処理・更新処理の性能向上の方針
 - ポインタで直接ページをつながない
 - 間接的レイヤ : ページ論理IDを物理ポインタにマッピング
 - ノードへの更新時には, そのノードの修正ログにデルタレコードを追加
- CMUによるOpen Bw木の実装あり^[24,25]



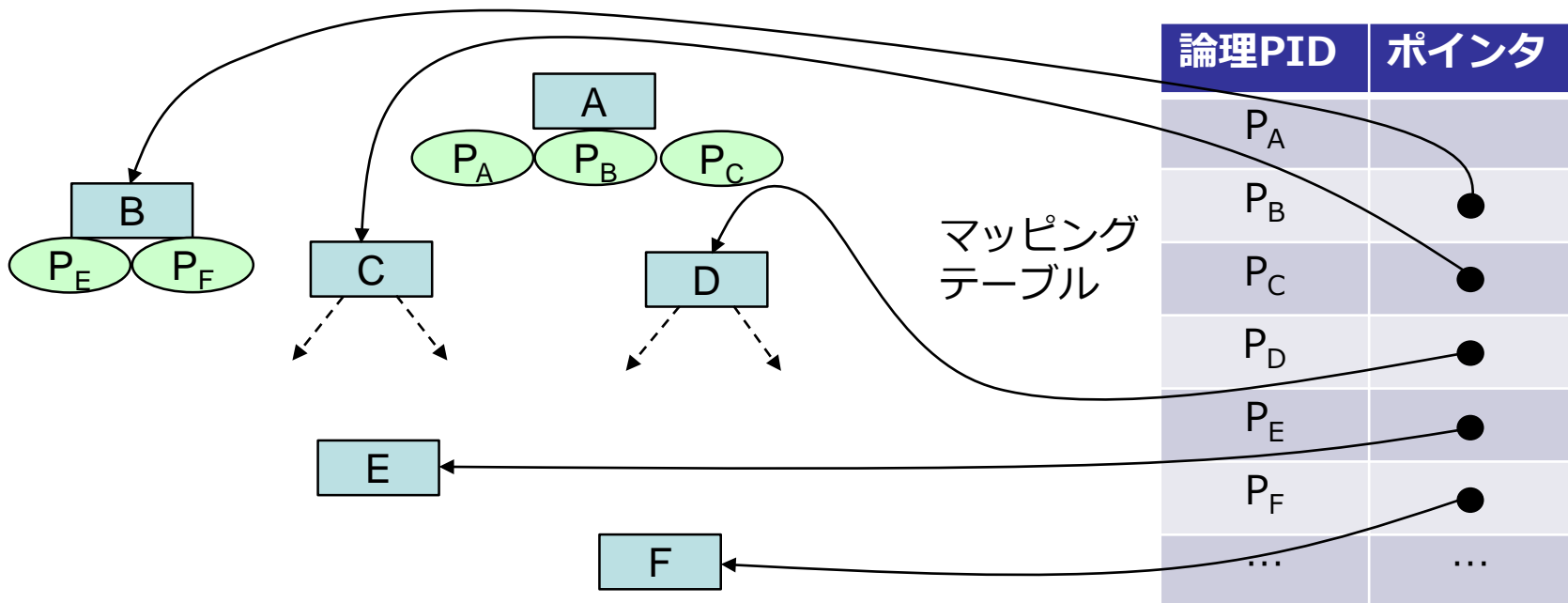
Bw木のアイデア (1)

論理的な構造



- 直接的なポインタを用いず、論理ページIDを介して間接的に参照
- ポインタは、主記憶上のアドレス、またはフラッシュストレージのオフセット
- 物理ページの位置を自由に移動できる利点

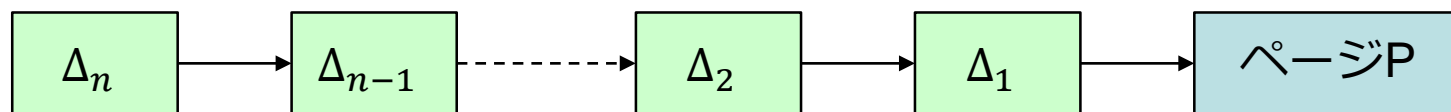
物理的な構造



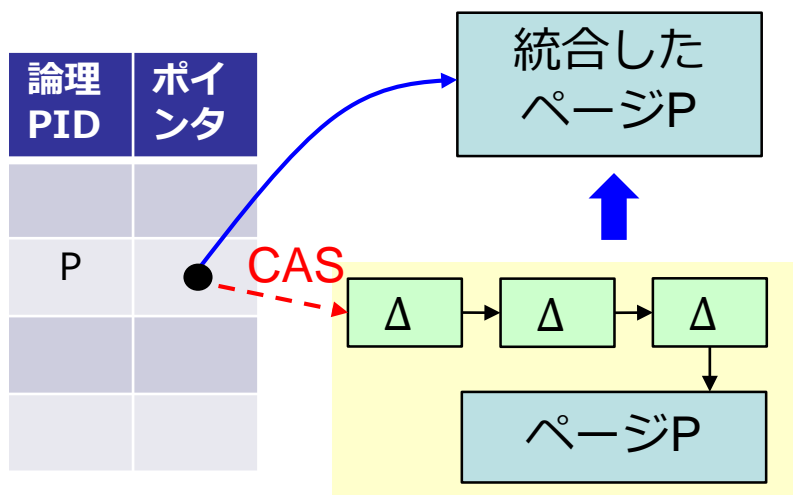


Bw木のアイデア (2)

- 更新時には、ページ（ノード）を直接書き換えるのではなく、**追記の連鎖**（**デルタチェーン**）で表現
⇒ 一括更新処理による効率化



- CAS (Compare and Swap) 命令**で、一気に参照を切り替え



- ロック（ラッチ）フリー処理のためにしばしばとられるアプローチ
- アトミックにページ参照先を切り替え
- 追記の連鎖を元のページと統合した新たなページに一気に切り替える



Bw木のまとめ

- 間接的ページ参照

- 大域的な状態の変更をアトミックな処理に分解し、ロックを避ける：CAS命令の使用

- デルタチェイン

- 索引の更新をデルタレコードの追記処理で処理することは、キャッシュが無効になることが減る
- 更新処理は一括して反映

- ノード分割・統合の処理は複雑

- アトミックな処理に分割
- ロックに近い処理が一部含まれる

- インメモリ主体のデータ構造：各ページは可変長

- ログ構造化フラッシュストレージを下位に想定



参考文献 (6)

18. J. Gray and A. Reuter: *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993.
邦訳：トランザクション処理：概念と技法（上・下），日経BP社, 2001.
[B木のロック処理を解説]
19. P.A. Bernstein and E. Newcomer: *Principles of Transaction Processing*, 2nd Ed., Morgan Kaufmann, 2009. [B木およびB^{link}木のロック処理を解説]
20. G. Weikum and G. Vossen: *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*, Morgan Kaufmann, 2001. [B木のロック処理を解説]
21. S. Sippu and E. Soisalon-Soininen: *Transaction Processing*, Springer, 2015.
[B木およびB^{link}木のロック処理を解説]
22. P.L. Lehman and S.B. Yao: Efficient Locking for Concurrent Operations on B-Trees, *ACM Transactions on Database Systems*, Vol. 6, No. 4, pp. 650-670, 1981. <https://doi.org/10.1145/319628.319663>
23. J.J. Levandoski, D.B. Lomet, S. Sengupta: The Bw-Tree: A B-tree for New Hardware Platforms, *Proc. ICDE Conference*, pp. 302-313, 2013.
<https://doi.org/10.1109/ICDE.2013.6544834>



参考文献 (7)

24. Z. Wang, A. Pavlo, H. Lim, V. Leis, H. Zhang, M. Kaminsky, D.G. Andersen: Building a Bw-Tree Takes More Than Just Buzz Words, *Proc. ACM SIGMOD Conference*, pp. 473-488, 2018. <https://doi.org/10.1145/3183713.3196895>
25. wangziqu2013/BwTree: An open sourced implementation of Bw-Tree in SQL Server Hekaton <https://github.com/wangziqu2013/BwTree>