データ工学特論（8）

情報基盤センター
天野　浩文

この講義の webサイトを作りました.
http://isabelle.cc.kyushu-u.ac.jp/~amano/data-engineering/index.html
レポート課題も掲示しましたので，参照してください.

1

さて，今回からは…（1）

- **データベース処理の高速化**
  - 単にディスクアクセスの速度や，演算処理の速度を向上させるだけでなく，以下のような機能も重要
    - 大規模データベースの中から**目的のレコードだけを高速に**読み書きする機能
    - **メモリには納まりきれないほどのデータ**（＝メモリにすべて読み込んでしまうことはできないようなデータ）を扱う機能
    - **頻繁な挿入・削除**を効率よく処理する機能
    - **複数のリクエストを並行処理**する場合の一貫性制御
  - これらを解決するために
    - **データベースに特有のデータ構造**
    - **データベースに特有のアルゴリズム**

2

さて，今回からは…（2）

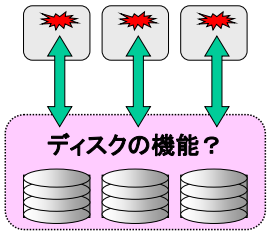
- 大規模データベースにおけるディスクアクセス
  - 複数のファイルを同時に扱う
  - すべてをメモリに読み込むことはできないので，それぞれのファイルから少しずつ読んでは結果（中間データ）を生成する
  - ディスクアクセス回数を減らしたり，中間データのサイズを小さくしたりするために，さまざまなアルゴリズムを用いる
- この場合のディスクアクセスのパターンを詳細に分析すると．．．
  - 一見ランダムに見えるが，**実はある一定のパターンで**データの全体にアクセスしている場合がある
  - **特定のデータに繰り返しアクセス**するような場合もある

通常のファイルアクセスの場合とは異なる特殊な先読みやキャッシングが有効になることもある

3

さて，今回からは…（3）

- ホスト側（OS）に行わせてもよいのだが，ディスクの**機能を高めて，処理を分担させる**ことも効果がある
- より「賢い」ディスクを目指して
  - **高機能ディスク**
  - **自律ディスク**



4

さて，今回からは…（4）

- 並列処理による高速化
  - **（並列）データベースマシン**
    - 専用ハードウェアで実現されるもの
    - 残念ながら，あまり価格が下がらなかった
    - 商業的には失敗…1990年代半ばまでにすたれてしまった
  - **並列データベースシステム**
    - 汎用的な並列計算機やPCクラスタ上にソフトウェアで実現されるもの
    - 「売れ筋」のハードウェアに載れば，価格も下がる
    - 現在の主流…さまざまな商用システムが登場

⇒ そこで用いられる高速化手法について紹介

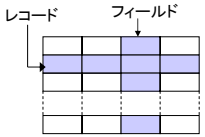
5

データベースの基礎

6

データベース処理の基礎（1）

- **リレーショナルデータベース**
  - **レコード（タプル）**からなる**テーブル**の集まり
  - データベースに対する問い合わせ
    基本的なデータベース演算で表せる
  - テーブルを入力とする演算の出力もまたテーブル



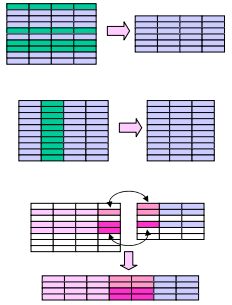
製品番号	品名	価格
000001	ボルト	10
000002	ナット	10
：	：	：
：	：	：

伝票番号	製品番号	数量
000000001	000001	100
000000002	000001	1000
：	：	：
：	：	：

7

データベース処理の基礎（2）

- リレーショナルデータベースから，求める情報を引き出すための基本的な処理
  - **選択**(selection)
    - 条件を満たすレコードだけを抜き出す
  - **射影**(projection)
    - テーブルから必要なフィールドだけを切り出す
  - **結合**(join)
    - 共通のフィールドを仲立ちにして複数のテーブルを横につなぐ
    - 基本リレーショナルデータベース演算の中では**最もコストが高い**



8

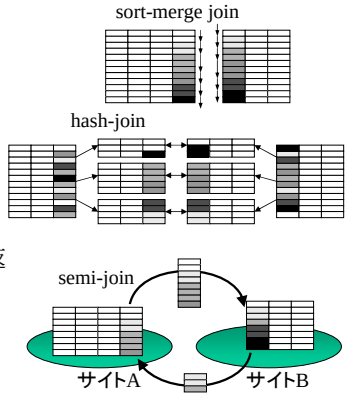
データベース処理の基礎（3）

- データベースにおける高性能化のテクニック
  - **結合処理の効率化**
    - 最もコストのかかる処理をより速くする
  - **問い合わせ最適化**
    - よりコストのかからない形式に問合わせを等価変換する
  - **並行処理・並列処理**
    - 複数の問合わせを並行に処理することによって，スループットを上げる

9

データベース処理の基礎（4）

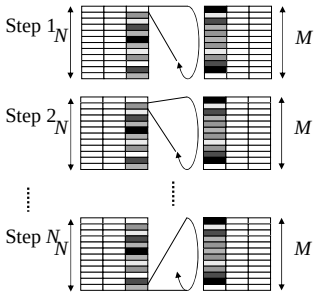
- **結合処理の高速化**
  - 各テーブルをソートしてから結合する(sort-merge join)
  - ハッシュを用いてテーブルを分割し探索のコストを下げる(hash join)
  - 結合に用いられるフィールドだけ射影して送り，結合する相手のあるレコードだけ送り返す(semi-join)
    ⇒通信量の削減



10

結合処理のいろいろ（1）

- **入れ子ループ(nested-loop)結合**
  - 前処理や補助的なデータ構造を必要としない処理法
  - $O\,(M \times N\,)$




```
for r_i in R do
  for r_j in S do[A]
    if r_i[A] = r_j[A]
    then
      r_i と r_j を結合して
      出力に入れる
```

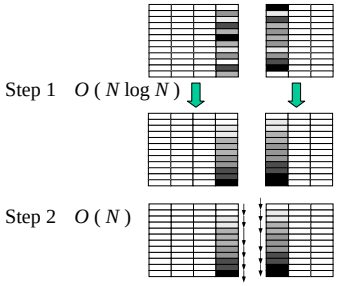
上のRをouter table，Sをinner table などという

現実には，1レコードごとにではなく，ブロック単位でスキャンが繰り返される

11

結合処理のいろいろ（2）

- **ソート・マージ(sort-merge)結合**
  - 2つのテーブルを結合フィールドでソートする
  - それぞれのテーブルを先頭からスキャンしながら結果を生成
  - 各テーブルのソートに$O\,(N \log N\,)$．ソート後は$O\,(N\,)$

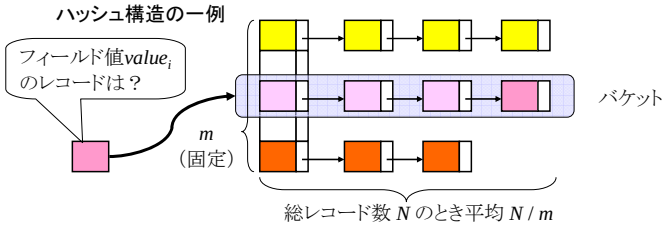


元のテーブルがすでにソートされている場合には，非常に効率がよい

12

ハッシュ（1）

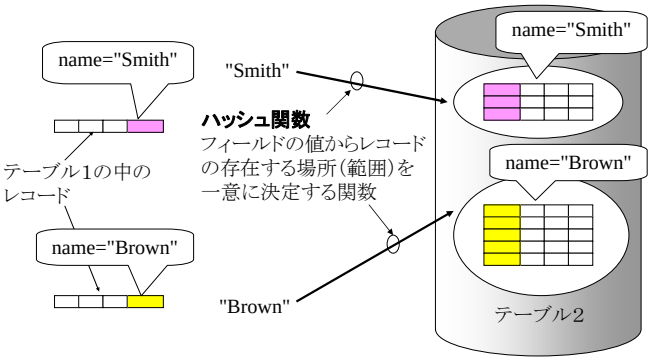
●もともとは，レコードの特定のフィールドの値が与えられたとき，そのレコードのありかを高速に求めるデータ構成法
- ◆フィールドの値に応じて，レコードをいくつかのバケット(bucket)に分割して格納しておく
- ◆逆に，値を決めると，そのフィールド値を持つレコードはすべてそのバケットの中にある



13

ハッシュ（2）

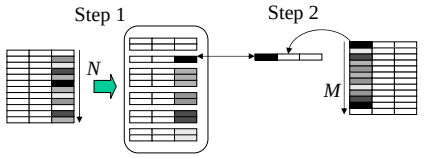
●ハッシュ結合の利点
- ◆結合のために比較すべきレコードを探す手間が少ない



14

結合処理のいろいろ（3）

●**単純ハッシュ結合(simple hash join) [DeWitt 1984]**
- ◆テーブル1の全レコードの結合フィールドにハッシュ関数を適用し，ハッシュ値に基づいてハッシュ表に登録
- ◆テーブル2の各レコードの結合フィールドにハッシュ関数を適用し，そのハッシュ値がハッシュ表に登録されていたら，そこのタプルと結合
- ◆*O(M＋N)*



15

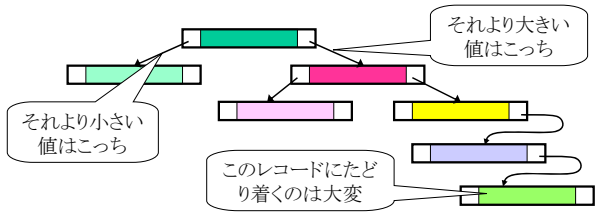
データベース用のデータ構造（1）

●データベースには，データの追加・削除が発生
- ◆平坦なファイルにレコードをぴっちり詰めて格納すると，削除により「すき間」が発生
  - 記憶効率の低下は困る
    →**動的なデータ構造**が必要
- ◆大量のレコードの中から，特定のレコードを高速に見つける必要がある
  - 毎回先頭から順にスキャンするのはばかばかしい
    →**何らかの索引構造**が必要
  - どのレコードにもだいたい同じくらいの時間で到達できることが望ましい
    →たとえば，単純な二分木ではアクセス時間に大きなばらつきが出る

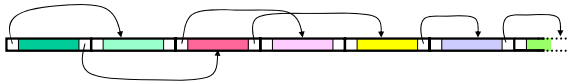
16

データベース用のデータ構造（2）

●単純な二分木にレコードを格納すると…
- ◆レコードの挿入順序によっては，木の高さ（深さ）が平衡しない



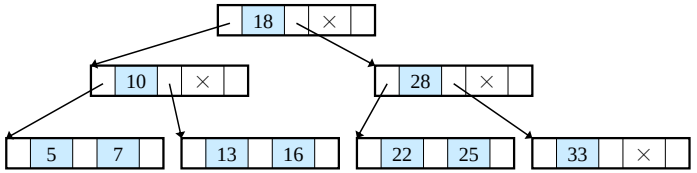
ファイルの中では以下のように配置されている．
先頭のレコードからリンクをたどって目的のレコードまで行く．



17

データベース用のデータ構造（3）

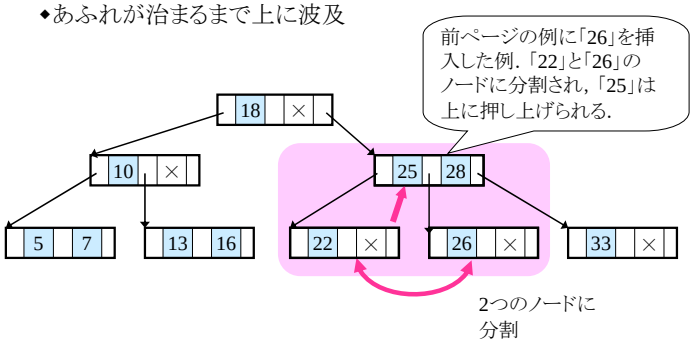
●B木 (B-Tree)
- ◆*m* 分木の一種
- ◆ノードはすべて同じ形（ただし，空の部分も残る）
- ◆ノード内では，レコードは小さい順に整列している
- ◆木の高さが平衡する（どの葉までの距離も等しい）ように挿入・削除を行う



18

データベース用のデータ構造（4）

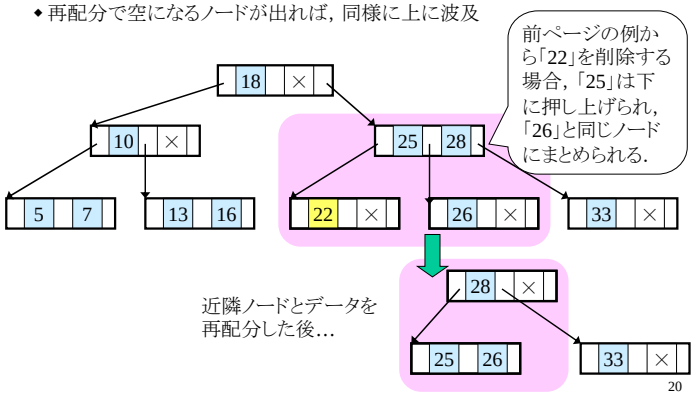
●B木 (B-Tree)への挿入
- ◆挿入は葉の部分から始める
- ◆あふれるときは，ノードを分割し，木を再構築
- ◆あふれが治まるまで上に波及



19

データベース用のデータ構造（5）

●B木 (B-Tree)からの削除
- ◆削除によりノードが空になるときは，近隣のノードのデータと合わせて再配分し，木を再構築
- ◆再配分で空になるノードが出れば，同様に上に波及



20

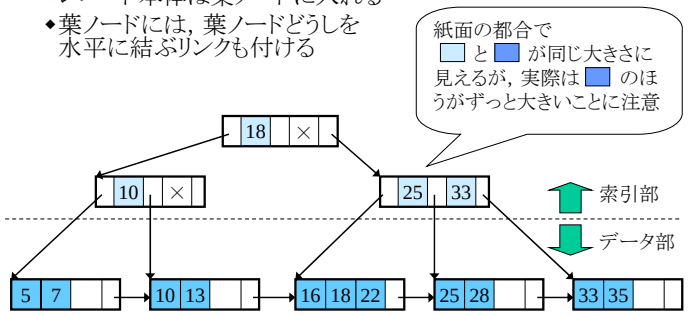
データベース用のデータ構造（6）

- B木の問題点
  - 全レコードをキー値の小さい順にスキャンするのが面倒
  - 実際には，レコードは単一の値からなるわけではないので，「キー」となるフィールドの値で索引付けされ，残りのフィールドもノードの中に納めなければならない
  - 中間ノードにレコード全体が入るので，木の上のほうもメモリの納まりきれない
  - 木の中間ノードにレコードそのものを入れる必要はないのではないか？

21

データベース用のデータ構造（7）

- B^+木 (B^+-Tree)
  - 中間ノードにはキー値のみ入れる
  - レコード本体は葉ノードに入れる
  - 葉ノードには，葉ノードどうしを水平に結ぶリンクも付ける



22

データベース用のデータ構造（8）

- B^+木の利点
  - 中間ノードが小さくなるので，その分，同じノードサイズで m（枝分かれの数）を大きくできる
    - 枝分かれの数が大きくなると，木は横幅が広がり，高さが小さくなる
    - 高さが小さくなると，**任意のレコードに到達できるまでの最大ディスクアクセス回数が小さくなる**
  - 中間ノードが小さくなると，**索引部がメモリに収まる**可能性も出てくる
  - データ部（末端の葉ノード群）は，**索引部と無関係に先頭からスキャン可能**

23

データベース用のデータ構造（9）

- B木，B^+木
  - 単一のフィールド（キー）による索引構造
- 複数のフィールドに索引が必要な場合
  - 別のファイルとして作成
  - データ構造は別のものでも構わない
  - ただし，挿入・削除の際の処理コストは増える

24

データベース処理の特性

- **リレーショナルデータベース**
  - 論理構造（テーブル，レコード）
  - 物理データ構成法（B木，B^+木）
  - 基本データベース演算（選択，射影，結合）
  - 結合処理（入れ子ループ，ソート・マージ，ハッシュ）
- **データベース処理特有のディスクアクセスパターンの例**
  - 一見ランダムに見えて，実は巨大なデータの全体にアクセス
    - メモリ容量の制約で，全データをメモリに読み込んで処理できない
    - 論理的には全レコードのスキャンでよいのに，物理的にはディスクの中を行ったり来たり
  - 特定のデータ（索引ファイル，B木の根に近いレコードなど）には，繰り返しアクセス

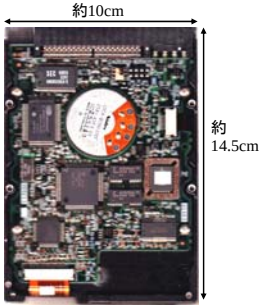
25

ディスクの高機能化

26

今のドライブは何でできている？

- 典型的なハードディスクドライブ
  - プラッタとモータ
  - ヘッドとサーボ機構
  - ディスクコントローラチップ
  - キャッシュメモリ
- ディスクコントローラチップの役割
  - サーボ機構の制御
  - エラー訂正
  - SCSIコマンドの処理
- 専用チップから汎用のチップへ
- キャッシュの大容量化



富士通製 IDE HDDの裏側
（1996年製，1GB，厚さ約2.5cm）
古いものなので，まだたくさんのチップが載っているが…

27

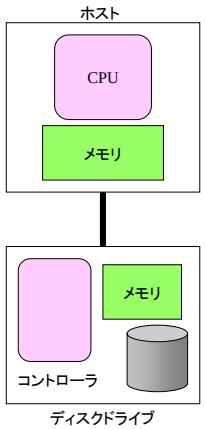
プロセッサの比較

- デスクトップ用プロセッサ
  - 最先端の性能
  - 消費電力は多少大きくても構わない
  - 複数レベルキャッシュ，分岐予測，複数命令の同時実行やout-of-order実行といった高度な機能を駆使
- ディスクコントローラ用プロセッサ
  - 数世代前のデスクトップ機なみの性能
  - 消費電力は低い（最新のデスクトップ用の1割以下）
- 将来，現在のデスクトップ機並みのCPUとメモリがディスクドライブ内に搭載できるようになると…
  - OSやアプリケーションまでもディスク側にダウンロード可能？

28

高機能ディスクの基本的なアイディア

- 現在ホスト側CPUで行われている処理を，**ディスク側CPUに肩代わりさせる**
  - **キャッシュ管理**
    - 繰り返し読まれる索引ファイルとそれ以外のファイルの差異を考慮
  - データベースレコードに対する**スキャン演算**
    - 不要なデータはホストに送らない
  - **ノード間の通信**（！）



29

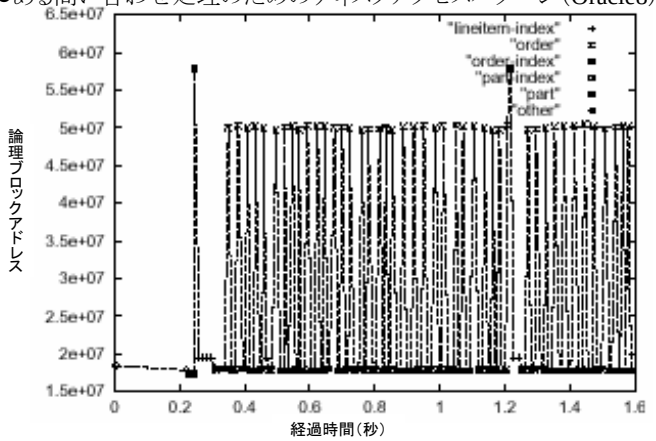
高機能ディスク：いくつかの研究

- **アプリケーションの性質に応じたプリフェッチ**を行う高機能ディスク（東大・喜連川）
- ディスク側の相互結合網と特殊なデータ構造を利用して負荷分散を行う**自律ディスク**（東工大・横田）
- 主にスキャン演算を肩代わりする Active Disk（CMU・Riedel他）
- ソーティングやデータキューブ演算まで行う Active Disk（UCSB・Acharya）
- 何でもかんでもやるIDISK（UCB・Patterson）

30

アプリケーションに応じたプリフェッチ（1）

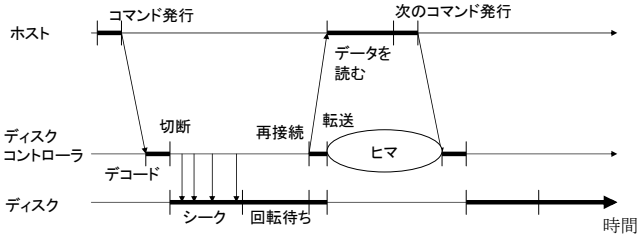
- ある問い合わせ処理のためのディスクアクセスパターン（Oracle8）



31

アプリケーションに応じたプリフェッチ（2）

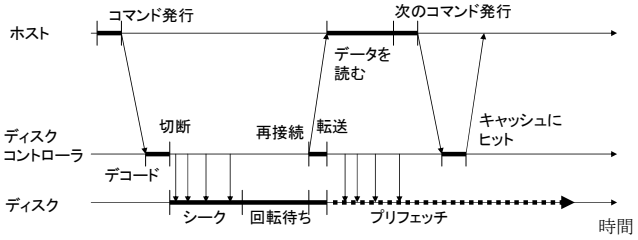
- 何が起きているか？
  - あるテーブルの索引とデータに交互にアクセスしているため，しじゅうシークを繰り返している
  - しかも，ホストがデータを受け取ってから次のコマンドを出すまで，ディスク（コントローラ）はヒマになる



32

アプリケーションに応じたプリフェッチ（3）

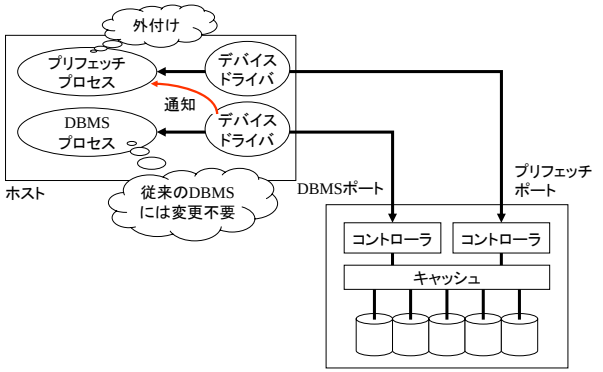
- そういうことなら，ホスト側からディスクに対して
  - このあとどのファイルにアクセスするのか（アクセスプラン）を教えてやる
  - コントローラはヒマな時間にそれをプリフェッチしておく



33

アプリケーションに応じたプリフェッチ（4）

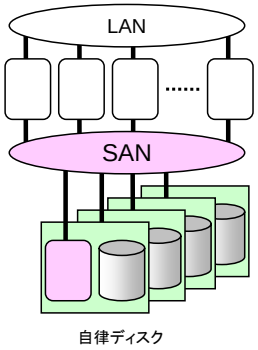
- プリフェッチ機構を有する高機能ディスク
  - ディスク側にアクセスポート，コントローラを2つ用意する



34

自律ディスク（1）

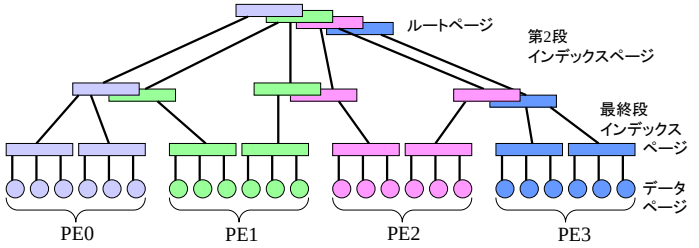
- SAN環境下での運用を想定
  - 各ディスクは，SAN内で相互に（ホストを介さずに）通信できる
  - ホストはどれかのディスク（**どれでもよい**）に処理を依頼
  - 自律ディスクは，自分が持つレコードは自分で処理し，他の自律ディスクが持つレコードは相手に処理を依頼
  - レコードを持つ自律ディスクは，処理の結果を**ホストに直接返信**



35

自律ディスク（2）

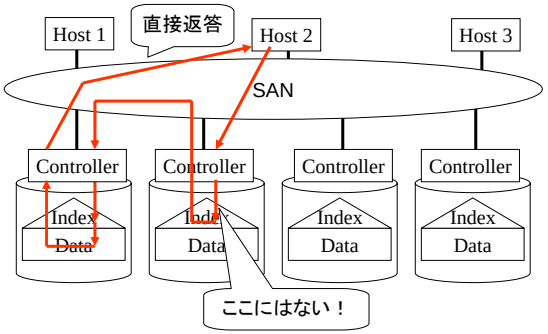
- Fat B-Tree
  - 自律ディスクのための物理データ構造
  - B木をそのままPEに分散させると，根に近いページを抱えたPEがボトルネックになる
  - 複製を配置
    - 根に近いノードは多くのPEに複製を配置
    - 葉に近いページは一部のPEにのみ複製を配置



36

自律ディスク(3)

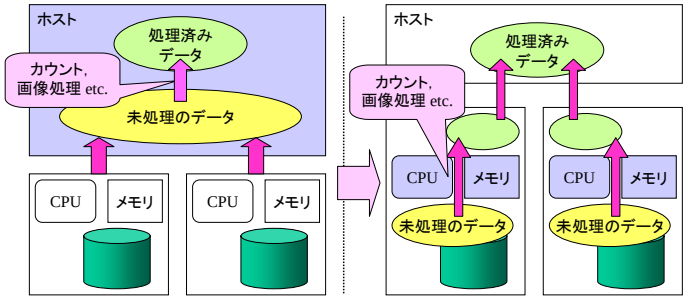
- ホストは，**Insert**，**Delete**，**Search**といった「高級な」コマンドを発行
- どの自律ディスクもルートページを持っているので，自分のところで検索を始めることができる



37

CMUの Active Disk

- 基本的な考え方(Riedel el al.)
  - 特定の応用(データマイニング，画像処理)に必要な，ごく単純な演算をディスク側で実行する
  - ディスクどうしは直接通信しない



38

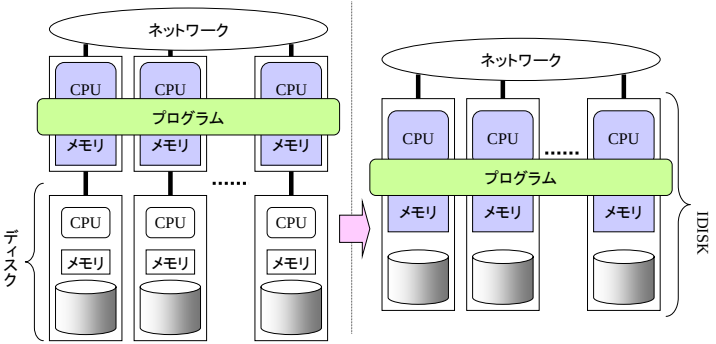
UCSBの Active Disk

- 基本的な考え方(Acharya et al.)
  - ホスト側(host-resident)コードとディスク側(disk-resident)コードが協調しながら処理を進める
  - ホスト経由の限定されたものではあるが，ディスク対ディスクの通信も許す
  - このため，CMU active disk に比べるとさらに複雑な演算も可能になる
    - 例:ソーティング，グルーピング

39

UCBの IDISK

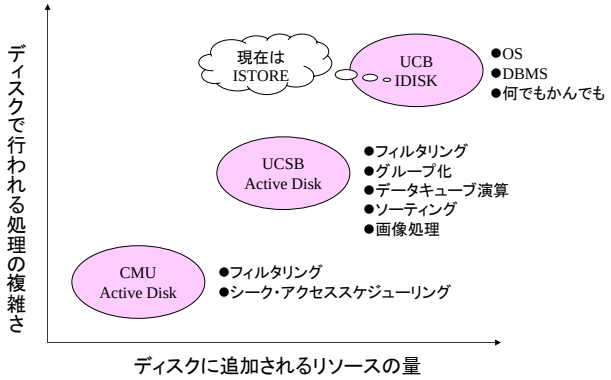
- 基本的な考え方(Patterson et al.)
  - 分散メモリ型並列計算機上で実行されるプログラムコードを一切合切ディスク側に押し込む



40

高機能ディスクの比較

●CMU，UCSB，UCBの高機能ディスクの比較[Patterson 98]



41

並列データベースシステム

42

データベースシステムの高性能化の背景（1）

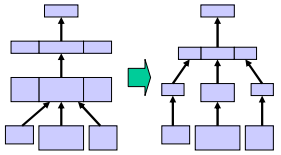
●データベースにおける高性能化のテクニック
- ◆**結合処理の効率化**
  - 最もコストのかかる処理をより速くする
    - –ソートマージ結合
    - –ハッシュ結合
- ◆**問い合わせ最適化**
  - よりコストのかからない形式に問合わせを等価変換する
- ◆**並行処理・並列処理**
  - 複数の問合わせを並行・並列に処理することによって，スループットを上げる

43

問い合わせ最適化(query optimization)

●演算の適用順序を変更することにより最適な問い合わせ処理プランを選択
- ◆処理時間や中間結果の大きさで処理コストを見積もる
- ◆問い合わせを等価な他の形式（処理コストの低い形式）に変換

●並列化以前から行われてきた
- ◆結合演算を遅らせて，中間結果のサイズを小さくする例：



44

データベースシステムの高性能化の背景(2)

- ●データベースの大規模化・要求の高度化
  - ◆より大規模でより高性能のデータベースシステムが必要
- ●ハードウェア技術の進歩
  - ◆CPU・メモリ・ディスクの高速化・大容量化
  - ◆急速に進んではいるが，無限に速く無限に大きなものは実現不可能
- ●有限の容量と速度を持つ(安価な)構成要素を無数に集めて，「**無限に速く無限大の容量を持つシステム**」を構築することはできないか？
- ●**並列性の追求**

45

データベースにおける並列化の目標(1)

- ●スピードアップとスケールアップ
  - ◆**スピードアップ**
    同じサイズの処理をより短時間で実行できる
    理想的には，*N* 倍のハードウェアを投入して，1 / *N* の時間で処理したい (linear speedup)
  - ◆**スケールアップ**
    同じ時間でより大きなサイズの処理が実行できる
    理想的には，*N* 倍のハードウェアを投入して，*N* 倍の処理を行いたい (linear scaleup)

46

データベースにおける並列化の目標(2)

- ●スケールアップの分類
  - ◆**トランザクションスケールアップ**
    同じ時間でより大きなデータベースに対するより多くの処理要求に対処できる
    ⇒ **オンライントランザクション処理**(on-line transaction processing, OLTP)
  - ◆**バッチスケールアップ**
    同じ時間でより大きなデータベースに対するより大きな問題に対処できる
    ⇒ **オンライン分析処理** (on-line analytical processing, OLAP), **意思決定支援システム** (decision-support system)
- ●今回は，スピードアップやバッチスケールアップのためのアプローチを取り上げる

47

最近のデータベースの使われ方

- ●**意思決定支援**
  - ◆企業の経営者が業務内容を分析して，経営戦略の立案に生かす
    例:POSデータから消費の動向を探り，商品の企画に生かす
- ●特徴
  - ◆大量のデータ(ときにはデータベースの全体)をスキャンしなくてはならない
  - ◆コストの高い演算(テーブルの結合)が多数含まれる
  - ◆蓄積されるデータが年々巨大化していく
  - ◆かと言って，毎年システムをリプレースできるほど予算があるわけではない
- ●必要条件
  - ◆予算を追加すればするほど，その分だけ規模・性能を向上させることができる(**スケーラビリティ**が高い)こと

48

最も安価でスケーラブルなシステムとは

- PCクラスタ（PC cluster）
  - 通常のPC（もしくはハイエンドPC）をノードとして，高速ネットワークにより多数結合する
  - ハードウェア的には，デスクトップやラックマウント型のPCとディスクを買ってきて，既設のシステムに追加するだけ
  - このようなシステム（分散メモリ・分散ディスク型，あるいは 完全無共有型 shared-nothing）に適した各種データベースアルゴリズムも開発・実装されてきた
- SMPクラスタ（SMP cluster）
  - シングルCPUのPCの代わりに，SMPをノードに用いる
  - やや高価

49

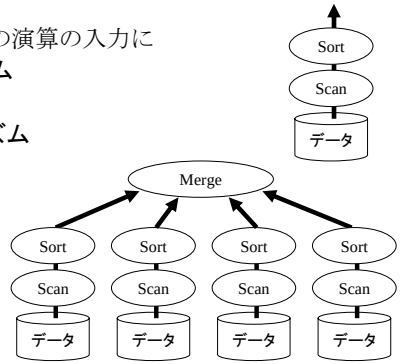
クラスタの弱点

- I/Oバスのボトルネック
  - SバスやPCIバスの規格がディスクドライブの転送速度の高速化に今後も追いついていけるのか？（技術的な問題というよりは，標準化のスピードの問題）
- システム管理の手間
- 台数が増えたときのパッケージング
  - ラックマウント型で少し解決
  - 消費電力の問題

50

データベース処理に内在する並列性

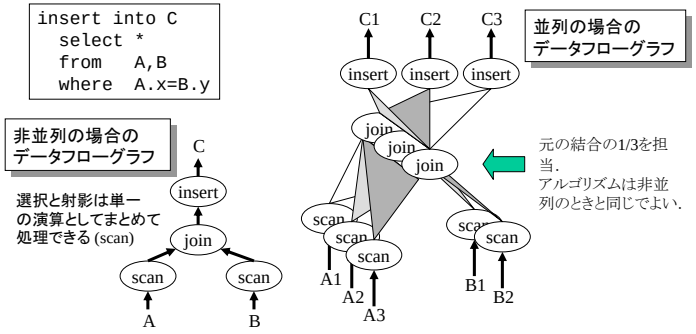
- データフローアーキテクチャとの親和性
  - テーブルを入力とするデータベース演算の出力もまたテーブル
  - 1つの演算の出力を次の演算の入力に
- **パイプライン型パラレリズム**
  - 演算を「縦に」つなぐ
- **パーティション型パラレリズム**
  - 独立に実行できる演算を「横に」並べる
- パーティション型のほうが並列性を抽出しやすい



51

パーティション型パラレリズムによる並列処理

- データ分割→データを複数のディスクに分割して格納
- 分割されたデータの上で，従来の手法を用いて 結合処理



52

並列処理に適した結合処理方式とは

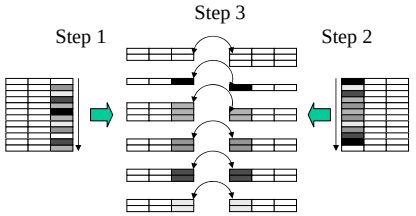
●データ分割によって並列性を抽出しようとすると…
- ◆入れ子ループ結合
  - • outer tableを分割・分配することはできるが，inner tableは分割できない
- ◆ソート・マージ結合
  - • ソート前に分割しても，結合処理の手間は減らない（結局総あたりになってしまう）
  - • ソート後の分割には，マージそのものと同じくらいコストがかかってしまう

●もっとうまくデータ分割を行うには…
- ◆**ハッシュ分割結合**

53

ハッシュ結合の並列化

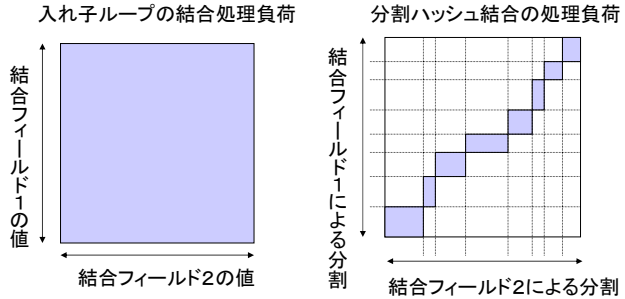
●**ハッシュ分割結合(hash-partitioned join)** の基本的な考え方
- ◆それぞれのテーブルを同一のハッシュ関数で分割（初期データが分割されていれば，ここは並列に処理できる）
- ◆対応するバケット毎に結合（ここは並列に処理できる）
- ◆データの偏りに強いと言われている



54

ハッシュ分割結合の処理負荷

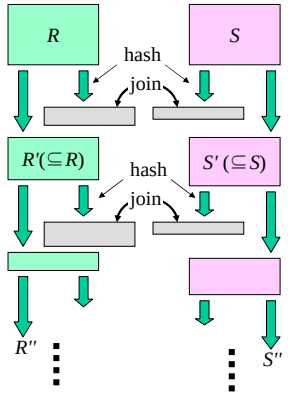
●入れ子ループが $O(M \times N)$ の直積空間をくまなく探索するのに対して，ハッシュ分割結合では，対応するバケット同士の部分のみを探索している



55

ハッシュ分割結合のいろいろ（1）

●**単純ハッシュ分割結合 (simple hash-partitioned join) [Gerber 1986]**
- ◆すべてのハッシュ分割を一度には作らず，一対ごとに作っては結合する
- ◆ハッシュ表に登録できなかったレコードは，一時ファイルに登録する
- ◆次のハッシュ分割を作るときは一時ファイルから読み出す



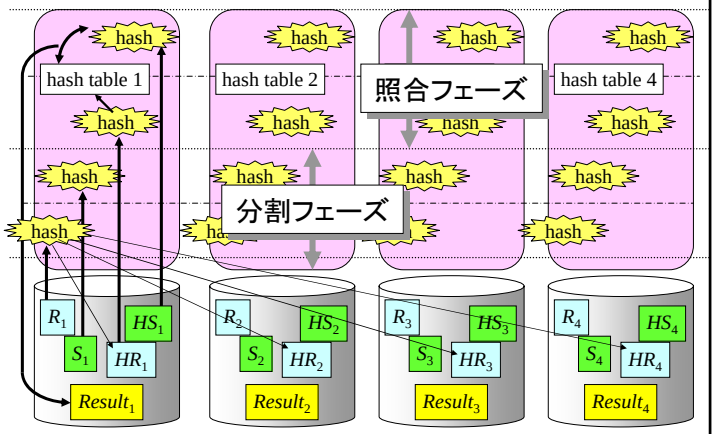
56

ハッシュ分割結合のいろいろ（2）

- **GRACEハッシュ結合(GRACE hash join)**[喜連川1983]
  - **分割フェーズ**
    各テーブルを同一の関数でハッシュしバッファに格納し，それぞれをディスクに書き込む
  - **照合フェーズ**
    - 一方のテーブルのハッシュ分割をディスクから読み，ハッシュ表を構築
    - 他方のテーブルの対応するハッシュ分割を1レコード毎にハッシュし，同一のハッシュ値が先のハッシュ表に見つかれば照合を行い，結果を生成する

57

GRACEハッシュ結合の並列処理の例

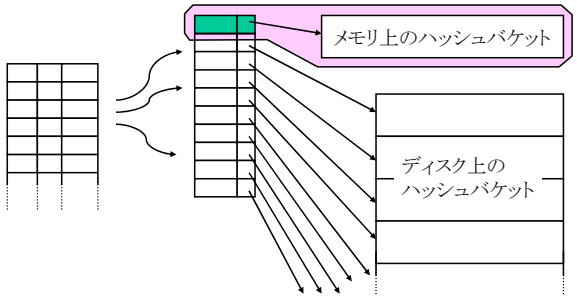


プロセッサ1でのデータの流れのみ示している．また，プロセッサ間通信もR1の分割のみ表示．

58

ハッシュ分割結合のいろいろ（3）

- **ハイブリッドハッシュ結合(hybrid hash join)**
  [DeWitt&Gerber1984]
  - 分割フェーズで全分割をディスクに書くのではなく，1つはハッシュ表とともにメモリ上に残しておく



59

ハッシュ分割結合のいろいろ（4）

- **ハッシュ分割ループ結合(hashed loops join)**[Gerber1986]
  - 入れ子ループ結合の変種
    - outer table を適当な大きさに分割する
    - 各分割をひとつずつメモリに読み込み，そのハッシュ表を構築する
      - inner table のレコードを1つ読み込んではハッシュし，現在メモリ上にあるハッシュ分割と一致するものは結合する
    - すべての分割について上記の処理を繰り返す
  - outer tableの各分割について，inner table の全レコードがスキャンされる

60