ソースコードを読み解くには

How to solve source codes?

NTT サイバースペース研究所
OSS コンピューティングプロジェクト
坂田 哲夫

何をお話するのか？

- ソースを読むとは、どういうことか？
  - 何がわかればよいのか
  - 何を目指すべきなのか
- 実践の仕方はどのようにすべきか？
  - 取り組み方
  - 道具立て
  - 成果にまとめる

理論編

実践編

ソースを読むこととは？

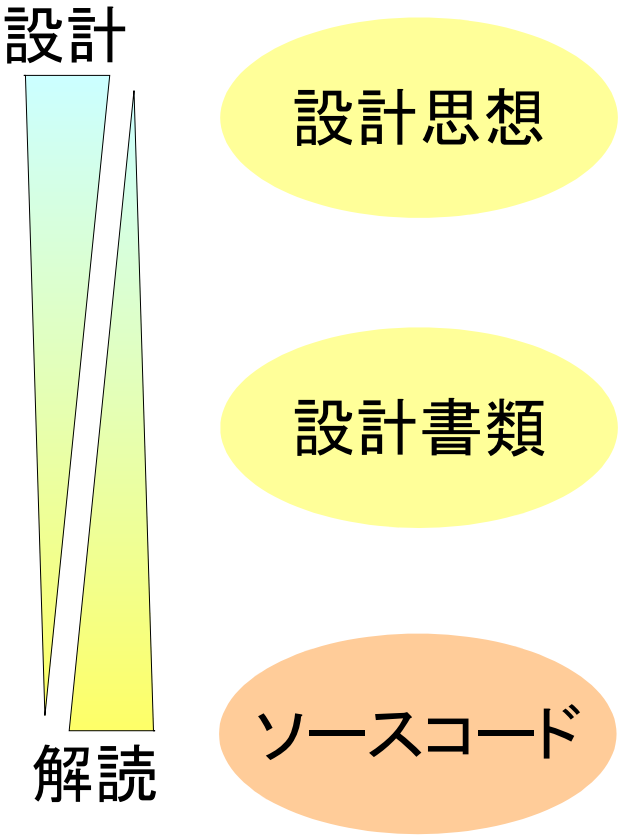
- 以下の点を明らかにしたい
  - ソースを読むとは、いったい何をすることなのか？
  - ソースを理解するとは、いったい何がわかることなのか？
- では、何がわかればいいのだろうか？
  - 設計資料相当の内容がわかればよいはずだ
    - PostgreSQL には設計資料がない
    - たとえ設計資料があったとしても、設計思想は分からない
  - 機能上・実装上の問題点について個々に理解したいはずだ
    - レベルに応じて資料化すべき

再設計としてのソース読み

- ソースを読むとは、いったい何をすることなのか？
  - 目的によって読みは異なる
    - デバッグ：異常事象の特定と解析⇒一部でよい
    - リバースエンジニアリング（再設計）：仕様と振る舞いの再確認⇒ある程度のまとまり
- ソースからシステム全体を理解するという観点は、リバースエンジニアリングに近い

設計作業を振り返る

- 設計の階層
  - 要件定義 : システムの満たすべき外部からみた条件を確定する
  - 機能設計 : 要求を満たすための機能上の構造を確定する (データ構造 (概要) 、モジュール構造 etc.)
  - 詳細設計 : 機能を満たすべき各関数の仕様を確定する (API 、振る舞い、詳細なデータ構造、アルゴリズム)



詳細設計を理解する

- 関数単位での理解
  - API : 入出力の意味、制限 (範囲)
  - 振る舞い : 機能 (入力⇒出力) 、副作用
  - データ構造 : 変数、構造体定義のレベルで
  - アルゴリズム : 無名のアルゴリズムにも注意

機能設計を理解する

- モジュール単位での理解
  - 機能モジュール: DBMS の機能を充足するための関数群を特定する
    (*e.g.* ロックマネージャ、ログマネージャ)
    - 入出力・振る舞い・データ構造・アルゴリズムを特定
  - モジュール構成: 個別関数への機能分解
  - 外部仕様との関係を理解
    - モジュール内・モジュール間

どのように取り組むか

- 事前準備
  - 読む対象を決める
    - 機能モジュール単位くらいが適当
      (1関数では小さすぎて却って意味が掴めない)
  - 機能の本質を知る
    - 教科書等でその機能の位置付け・内容を整理しておく
      ⇔ソースコードから機能の本質を知るのは難しい
  - 時間を確保する
    - まとまったソースを読むには相当の時間を要する
    - 中断されない時間 (1時間以上) が必要
  - ノートを準備する
    - 長期の作業を記録し、サポートする

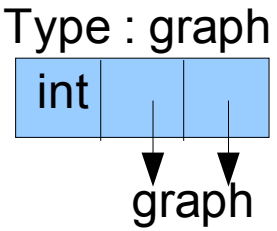
どこから読むか

- アルゴリズム＋データ構造＝プログラム (N.Wirth)
  - データ構造
    - 静的構造＊：宣言された枠組み (構造体)
    - 動的構造＊：複数の構造体のネットワーク
      （生成手順が手続きに内包されている）
  - アルゴリズムがデータ構造を生成し、データ構造がアルゴリズムを駆動する。
    - アルゴリズムの解読とデータ構造の解読が並行して進む
    - 一度に両方解読できないので、ノートを取りつつ進める
      - (＊：ここでの仮の名前）

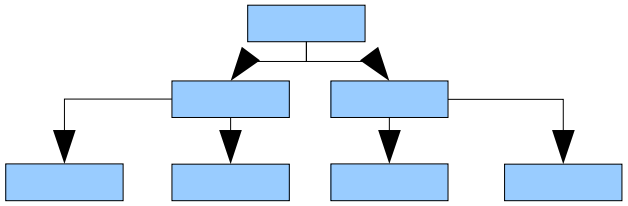
データ構造を読む

- 静的構造を調査する
  - 型 (構造体など) 宣言
    - 各種のツールがあれば効率的に進められる
- 動的構造を調査する
  - 実際に使われる際のデータ構造
    - 生成手順の調査が必要
      - コードを調査してノートを取る

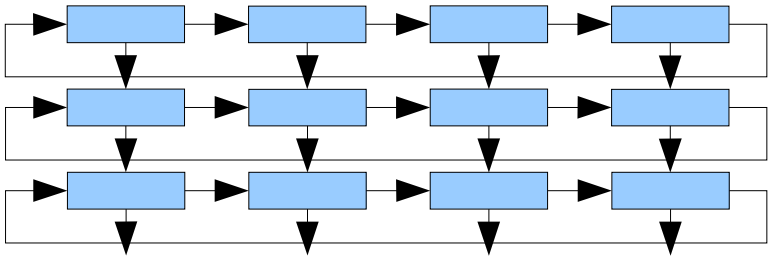
静的構造の例



動的構造の例



動的構造の例



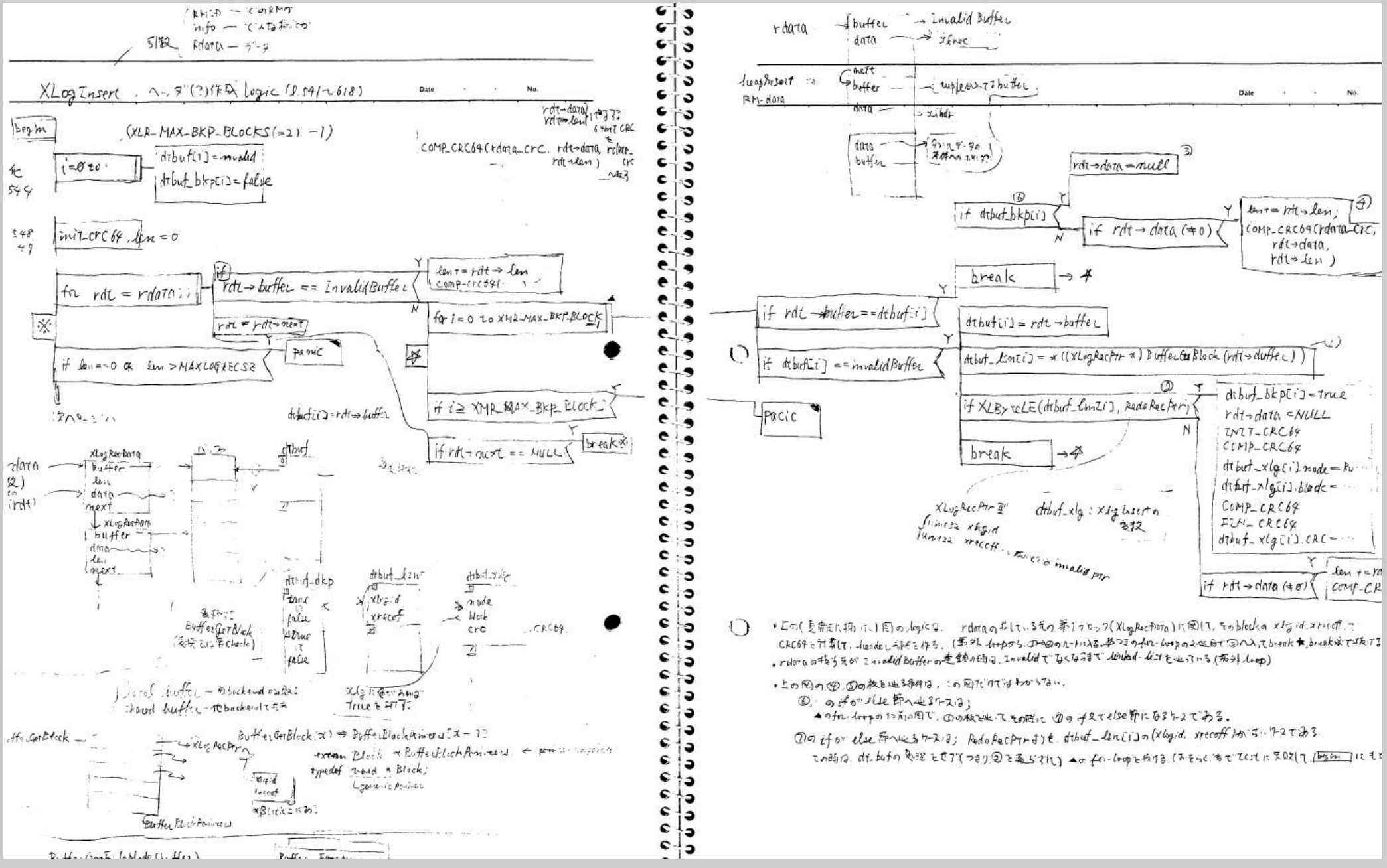
道具立て:ノートの三原則

- 第１条:紙切れに書かない
  - 紙切れは散逸する。もったいないのは紙ではない
- 第２条:詰めて書かない
  - 分かったことは追記する。転記は無駄、ミスの元
- 第３条:専用の大判ノートに書く
  - 多くの頁を参照する際のロスになる
  - １頁に書ける量が多いほうが良い⇒見開きを使う

ノートの例

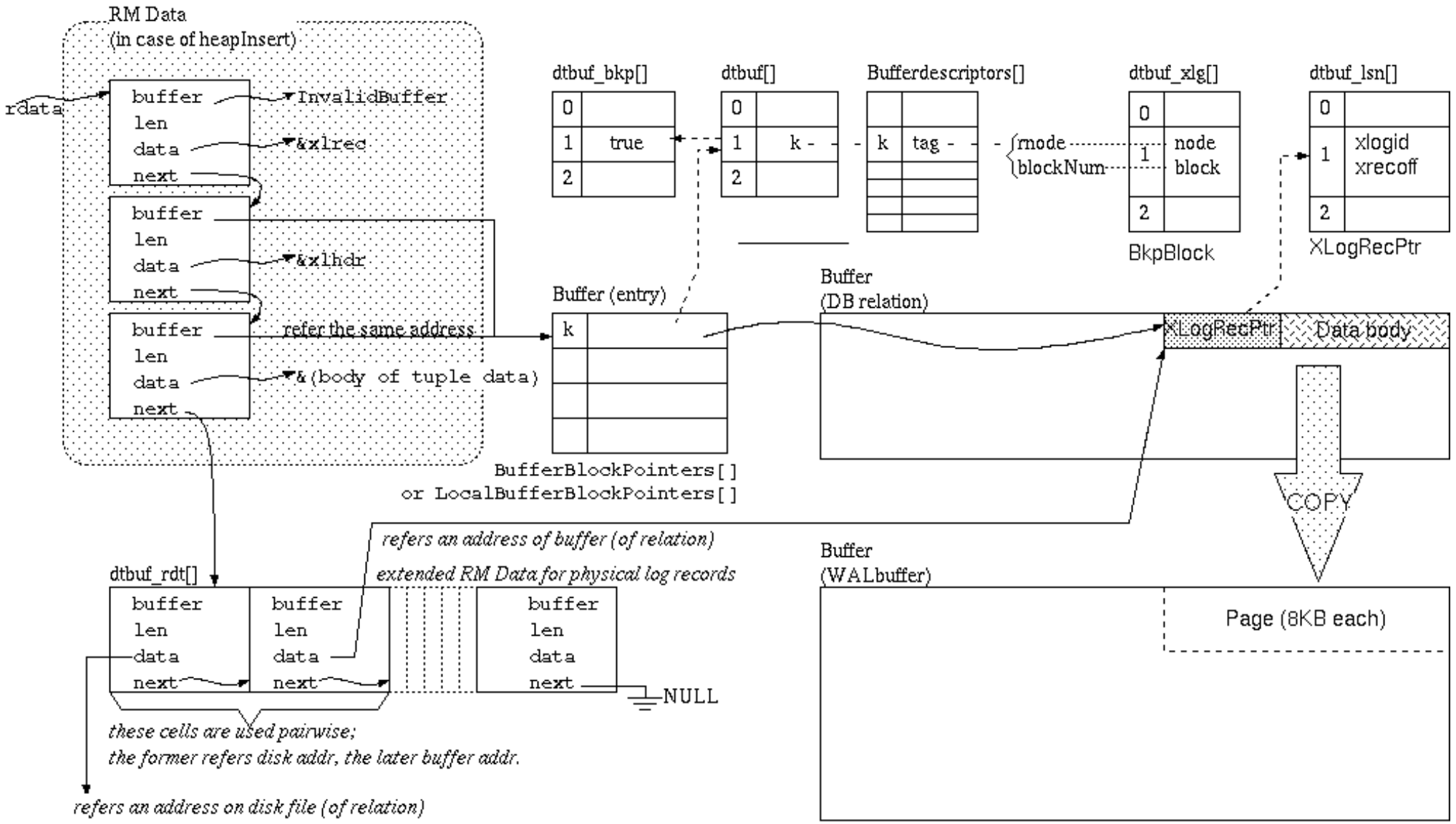
XLogInsert の例
- A4 見開き、0.3 のシャーペン
- 約 70 行分

むだ話



整理したデータ構造の図

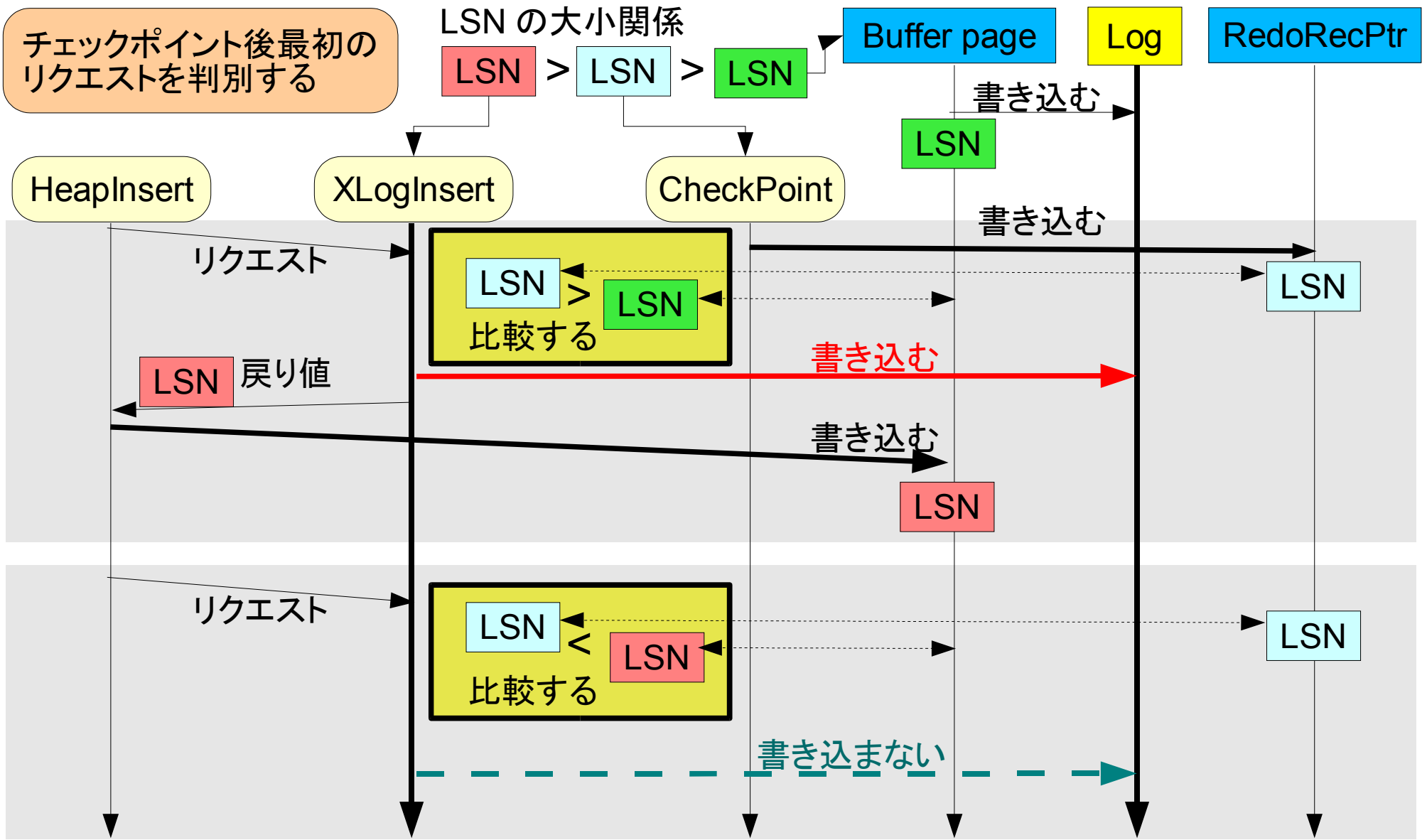
XlogInsert の主要データ構造



アルゴリズム

- 無名のアルゴリズムに注意せよ
  - 大事なのは、PostgreSQLに固有の処理、
    教科書が書かないアルゴリズム
- 隠れたアルゴリズムもある
  - アルゴリズムの断片が各所に埋め込まれている
    ⇒積み上げてみて初めて姿を表す

隠れたアルゴリズムの例



LSN: log serial number

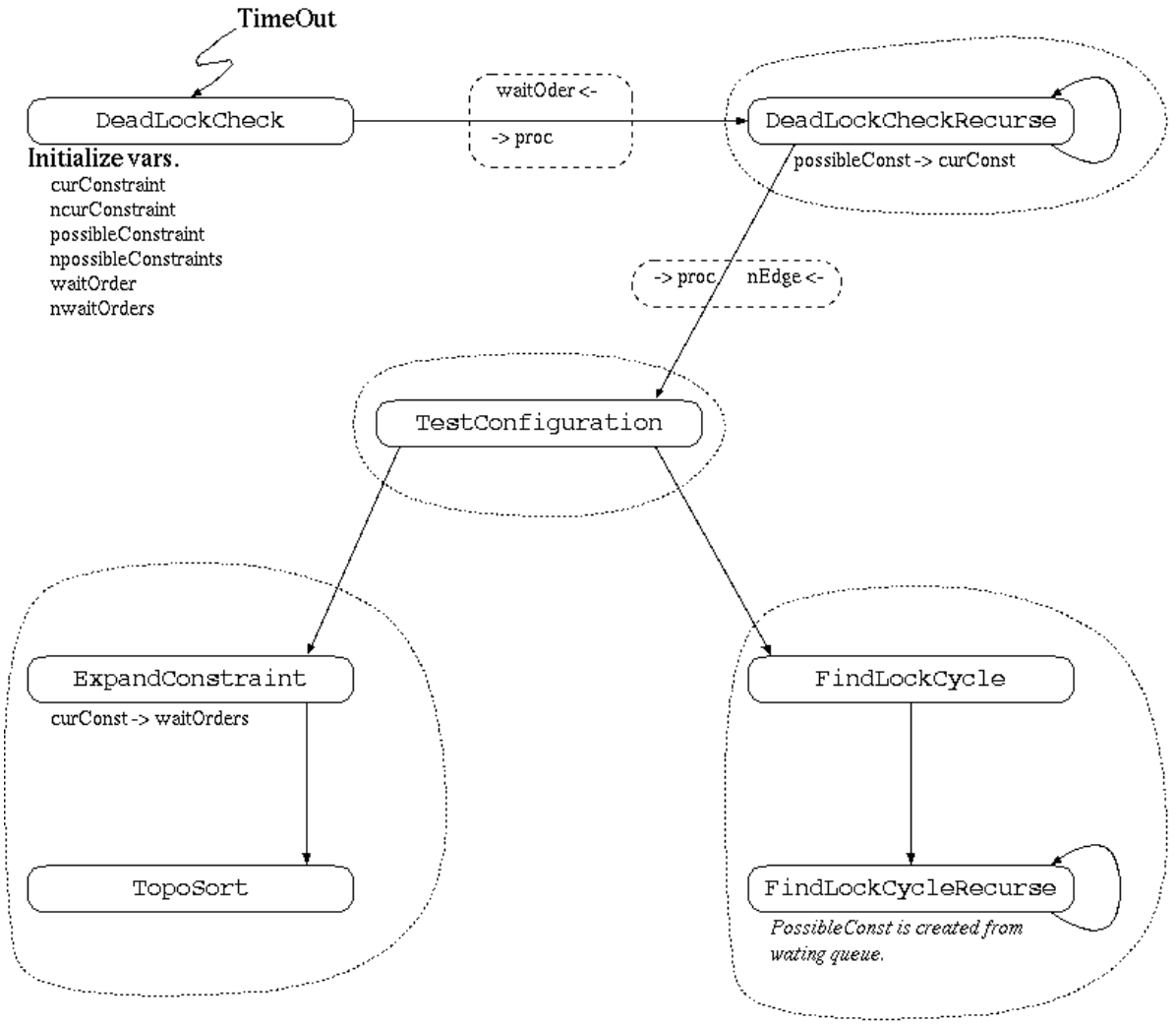
成果をまとめる

- 書かないと分からない・共有できない
  - 書かれない小説は存在しないも同じ
- 本質と偶有
  - 教科書に載っていない情報が重要
- **めそ**な視点を持つ
  - ミクロ = ソースレベル、マクロ = ブロック図
    (両方とも既にあるだろう)⇒ メソ（meso）は中間
  - 各種のマネージャ (log, lock, storage, buffer...) 単位にまとめる
  - マネージャ間 (inter)/ マネージャ内 (intra) の情報

メソレベルの説明の例

- 大機能は複数の小機能で実現される
- モジュール単位での機能の解説
- 小機能への分割が鍵
  - 良い設計ならばモジュール間は I/F 少ない筈
  - 大域変数にも注意

デッドロック検出器のモジュール構造



構成の例：設計書風に

- 要件
  - 分析対象となる機能への要件、分析範囲など
- 機能 / 外部仕様
  - 分析対象の持っている機能
  - インタフェース (概要)
- 内部構造
  - モジュール分割とインタフェース (詳細 ; API, FAP)
  - データ構造 + アルゴリズム
    - アルゴリズムはソースコードの概要を説明

運用・サポートなどを念頭に

運用・サポート / 改造作業両方に

改造作業にも使えるレベルで

各種まとめ作業

- 継続する必要性⇒次の作業への展開
  - PostgreSQL は大規模。継続は力
  - 一人での読了はムリ。共同が鍵 (他の人が継続)
- 測定と計画
  - 実績値に基づく作業計画の必要性
    - 生産性情報の取得;ソフトウェア開発の計画と同じ
- 展開とは共有なり
  - 資料化することが大事 (人間は忘れる葦である)
  - 資料の共有は**もっと**大事

Fin

ソースを読んで資料化したら、是非、
仕組み分科会に寄書してください