バッチ処理について

2020/10

NAUTILUS

背景

- 現状のRDBではconcurrentなwrite処理はそもそも、うまくない
  - 一貫性の担保はコストがかかる
  - その一貫性もserializableの制約が強い
- さらにこのwriteが重いバッチ処理になると致命的遅くなる
  - Multi-Version(MV)といいつつ2Vだったりする→競合が発生するとabort/retryになる
  - 結局、write-lockをとっていたりする→大量の書き込みが大量のlockになって大量のwaitが
- その結果、現状ではRDBとwrite-heavy batch処理は相性としては最悪に近い
  - とはいえ「相性が悪い」で片づけられる話ではない

Tsurugiでやること

- やること
  - Write-heavyなバッチ処理を前提にして、実用に足りうるベンチマーク・アプリケーションを作成する
  - このベンチマークでパフォーマンスを出す
  - ↑目標ですよ。目標ね。絶対できるものは目標とはいいませんよ。

- 事前に想定できるアーキテクチャ要件
  - MV
    - MVは絶対的に必要。ないと話にならない
    - さらに従来のMVよりも拡張されたものが必要
      - 従来：serialization orderの決定過程でのみMVであればよい
      - バッチ処理：永続的なMVである必要がある→やりすぎるとリソース枯渇
  - Lock free
    - lockを抱いたままでバッチが死んだり、オンラインがコケると後始末が大変
    - dead lock detection とかタイムアウト作戦が取れないので厳しい
    - abort roll-backのコストが強烈に高い

- この辺で大体、負け試合の雰囲気が漂う

現状の進捗状況

- ベンチマーク作成
  - 実際の業務バッチの解析→終了
  - シナリオの作成→ほぼ終了
  - データ作成/プログラム実装→今ここ
  - SQL化→今後
  - 実食→今後

- DB要件の整理
  - 論理モデルの作成→ほぼ終了
  - 実装に対する手当の検討→案はできた
  - 実装→今後
  - ベンチマーク実食→今後

- つまり、どーなるかはまだなんとも言えない

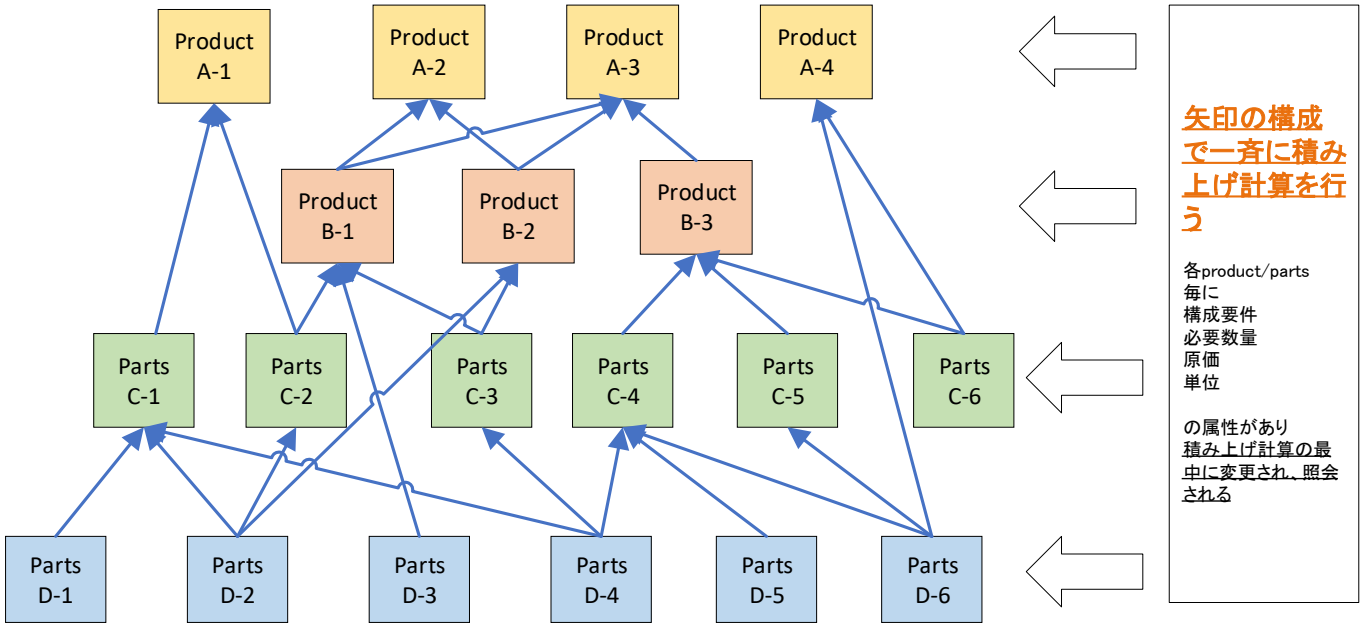
想定シナリオ

- ベンチマーク概要
- 外観
  - 1. 実行処理の長いバッチ処理（既存RDBで数時間/分散環境で数十分）
  - 2. 1のバッチ処理に混在させる断続的なオンライン処理
    - 実行時間の長いTx処理と短いTx処理を混在させる
    - バッチ処理については実行時間/オンライン処理については処理数

- 想定シナリオ
  - 「BOM(bill of material)の原価計算」
    - 食品工場のMRP（manufacturing resource planning)
    - バッチ処理：全製品の原価集計・所要量計算
    - オンライン処理：生成マスター/原材料マスター等の各マスターの個々の更新処理

想定シナリオ

- バッチ処理中もマスター更新が可能なシナリオで、かつ一貫性を担保し、パフォーマンスを担保する
- バッチ処理はtree構造になっている部品表の積み上げ計算をして、原価・所要量の更新処理を行う。
- その間にオンライン処理が、各マスターテーブルの更新も「同時」に行う。
  - 通常はオンラインを閉塞して、バッチ処理のレプリカを作成してバッチ処理を行い、書き戻すことが多い→それをやらない
  - それでもなおRDBでのtree traverse処理はjoinが非常に重いので処理に時間がかかってしまう。
- 正常処理のみだけではなく異常終了もベンチマーク対象とする



閑話休題

- 中身の話を少し
  - まず誰でも考えるPlanA
  - 「MVでSnopshot（SS）とって、バッチ処理して、他TxはRead only のみ許可する」
    - 一切writeできないね・・・
  - ということでPlanB
  - 「必要なデータセットでSSとって、バッチ処理して、他TxはRead onlyのみ許可する。バッチ処理とそれ以外をisolateする」
    - いやそれ、バッチのpredicateを事前全部評価しないと・・・
    - 結局全部か？

もともと無理筋では？疑惑

- もちろん「バッチ処理で全部のレコードを一斉にRMWする」ようなケース（例：全データの洗い替え）は、そもそも手も足も出ない
  - これはDBに限らず、IT全般に言える

- **ただし、すべてはそういうわけではない**
  - **1. Instant write：w(x)　→無条件で可能**
  - **2. Read other write：r(x)w(y)　→条件つきで可能**
  - **3. Read modify write：r(x)w(x)　→無条件で不可能**
- **すなわち、各Txが持つ属性（semantics)により対応が異なる**
  - **普通にsyntaxだけでは総当たり（NP完全）になってアウト**

- 各Txの属性に応じてどう処理するか？
  - 1.特に問題ではない
  - 2.ロジックの問題→Scienceの問題
  - 3.ロジックは関係ない→Engineeringの問題

- バッチ処理は、モデル的な話と技術的な話の総動員が必要
  - 非常に困難なので、アカデミアでもあまり議論されていない
    - ここ30年で数本の論文（例：利他ロック）ぐらいしかない

少しだけ理論の話

- MVとECCによるversion orderの管理
  - 例)
    - w0(x0)w0(y0)c0:バッチ開始時点
    - w1(x1)c1r2(x1)w2(y2)c2:バッチ処理の内容　t1/t2
    - r3(y0)w3(x3):t1が終了時点でオンライン処理が走る/バッチもt2開始
    - できあがったhistory c0< w1(x1)c1r2(x1)r3(y0)w3(x3)c3w2(y2)c2
  - 単純なwrite order
    - x:0<1<(2r)<3
    - y:0<(3r)<2
    - 整合性が取れない。しかしながら、これはserializable
    - x:0<3<1<2rであれば問題ない t0<t3<t1<t2
    - 「バッチが始まった時にはまだ開始されていないオンライン処理を、そのバッチ処理が始める前に開始したことにする」ことができれば可能
    - 物理的なxのwrite orderをひっくり返して、論理的に逆順にする
      - 物理orderは変える必要はない。論理順のみを変える
  - というようなことができるような変態的なSchedulerが必要
    - いままでにはない

現時点

- 対象とするバッチ処理のシナリオは完成
  - 必要な要件はそれなりにわかってはいたが、ここでさらに明確化
- 理論的な枠組みとエンジニア的な枠組みの両者が必要
  - 理論的な枠組みは一から構築
  - エンジニア的な枠組みは、できつつある新しい枠組みに追加
- 今後は試行錯誤しながら、構築→修正の繰り返して行く
- 感想：
  - 薄々わかってはいたが、やはり小手先でどうにかなる問題ではない。
    - 1.そもそも問題を正確に分解することが必要
    - 2.DBのコアから手を入れないととても無理
    - おそらく今後のTsurugiの「オリジナル」になると思われる