バットノウハウ・グレーノウハウへの対策および

論理設計における重要な考え方

作成者：蛭田 昌伍

作成年月日：2024年6月18日

所属：システム開発部

私のレポートのテーマは、「バットノウハウ・グレーノウハウへの対策」です。

このテーマに設定した理由は3つあります。

(1)現場での対処を考えておくため

(2)間違った設計を掘り下げることで、正しい設計への認識を深めるため

(3)間違える原因を理解し、他の業務への教訓とするため

レポート全体の構成としては、バットノウハウ・グレーノウハウへの要約と考察を列挙した後、

全体を通じて感じた論理設計における重要な点を記述し、まとめとします。

以下、「達人に学ぶDB設計徹底指南書」にて紹介された、バットノウハウ・グレーノウハウに対し、

(ア)概要

(イ)原因

(ウ)設計段階での対策

(エ)設計以下の工程での対処

の順に、教本の内容と私の考えをまとめ、記述していきます。

ただ4(設計以下の工程での対処)に関しては、想像が及ばない場合や、対処がない場合には記載しません。

1.バットノウハウ

(1)非スカラ値(第1正規化未満)

(ア)概要

SQLの「配列型」という機能を用いて、非スカラ値を含むテーブルを作ること。

リレーショナルデータベースの固定的なデータの扱いに対し、「配列型」は柔軟なデータ型といえるが、

正規化によるデータ整合性を保てなくなる恐れがある。

(イ)原因

一般的なプログラミング言語は「配列型」を持っているため、アプリケーションとリレーショナルデータベース間のデータの受け渡し(インタフェース)を円滑に行えるようにするため、SQLが1999年に標準規格の改定を行ったため。

(ウ)設計段階での対策

大前提として、安易に配列型を採用するのではなく、第1正規形を確保することを優先すべき。

設計の際、データベースだけでなく、接続するアプリケーションやミドルウェアとの整合性も考慮する。

(エ)設計以下の工程での対処

配列型に格納されているデータの意味を見失わないようにする。配列はそれ一つでデータのまとまりなので、

根本的な列の意味を見失いやすいと考えられる。列の意味を第一に考える必要がある。

(2)ダブルミーニング

(ア)概要

列の表す意味が途中から変化すること。

例えば健康診断のテーブルにおいて、体重を表していた列に、

ある年度から年齢の値が入力されている、などの場合。

列の意味が不明確なので、プログラマが混乱し、システムバグの温床となる。

(イ)原因

プログラミングにおける一時変数の考えがリレーショナルデータベースに持ち込まれている。

テーブルの列を変数のように扱っている。

(ウ)設計段階での対策

リレーショナルデータベースにおけるテーブルは現実世界の「写像」であり、静的で固定的な存在であると再認識する。

必要な情報が変わる場合は、新しい列を追加することで対処する。

(エ)設計以下の工程での対処

データの数値の傾向から、どの段階で列の意味が変化しているのか見極める。

必要に応じて新しい列の追加を提案する。

(3)単一参照テーブル

(ア)概要

「ダブルミーニング」の派生形。

同じ構造を持つテーブルを一つにまとめたことにより、複数の意味を持つ列が存在してしまうテーブルのこと。

・データ型は固定であるため、仕様変更に対応できない場合がある。

・レコード数の増大による検索パフォーマンスの低下を招く。

・SQLにてコードタイプやコード値を間違えてもエラーにならないため、バグの発見が困難。

・正確さに欠けるERモデルしか作れなくなってしまう。

など、様々な弊害がある。

(イ)原因

同じ構造を持つテーブルをまとめることで、処理の構成をシンプルにしようとするため。

また、SQLも少し簡潔にすることが可能なため。

(ウ)設計段階での対策

テーブルでは、意味の多様性を排除することを第一とする。

多少テーブルが増えることになったとしても、列の意味の一意性を重視する。

(エ)設計以下の工程での対処

単一参照テーブルでバグが発生した際には、原因の箇所にエラーが出ない場合も考えられるため、

列のデータ型などを確認して、慎重にデバックなどを行う。

(4)テーブル分割

(ア)概要

テーブル分割には、大きく2種類があります。

「水平分割」は、レコード単位でテーブルを分割すること。

分割する意味的な理由に乏しく、拡張性に乏しいといった欠点がある。

「垂直分割」は、列単位でテーブルを分割すること。利用される列のみのテーブルにすることで

「水平分割」と同じく、分割する意味的な理由に乏しいという欠点がある。

(イ)原因

「水平分割」・「垂直分割」のいずれも、テーブルサイズを必要最小限にすることで、

SQLのパフォーマンスを改善することが目的で行われる。

しかし、分割する意味的な理由がない場合は、むやみに分割を行うべきでない。

(ウ)設計段階での対策

「水平分割」は、「パーテーション」という機能で代替する。「パーテーション」はパーテーションキーを軸として物理的に格納領域を分離することが可能。

「垂直分割」は、「集約」で代替する。「集約」は

・列の絞り込み…保持する列を絞った「データマート」を作成する。

・サマリテーブル…集約関数によってレコードを集約した状態で保持する。

という2つの機能がある。これによって元のテーブルをく崩さない形での列の省略が可能。

(5)不適切なキー

(ア)概要

主キー、外部キー、結合キーなどに可変長文字列を使用すること。

キーの満たすべき条件である不変性を備えていないことに加えて、固定長文字列との混合の恐れがある。

値が変わる列をキーにしてしまうと、データ更新処理が多く発生し、

システムの安定的な運用とパフォーマンスの両面でマイナス。

(イ)原因

可変長文字列を使用する「名前」のような列をキーにしてしまっているため。

データの値が頻繁に変わるキーでは、データの更新が多く発生し、システムの安定的な運用とパフォーマンスの両面でマイナス。

(ウ)設計段階での対策

同じデータを意味するキーは同じデータ型にする。データ型は固定長文字列で揃える。

それによって固定長文字列と可変長文字列の保持する文字列の違いを、SQLコーディングの段階で細かく対処する必要がなくなる。

(エ)設計以下の工程での対処

固定長文字列は、空白を穴埋めする「パディング」を行うことを考慮する。

同じ意味を持つ他のテーブルの列とのデータ型がマッチしない場合は、それぞれの型が固定長文字列と可変長文字列となっており、

「パディング」による文字列のミスマッチが起こっている可能性を考える。

(6)ダブルマスタ

(ア)概要

同じ役割を果たすはずのマスタテーブルが、二つ存在すること。

マスタの情報を過不足ないものにするために二つを完全外部結合とUNIONのどちらかによって統合する必要がある。

どちらもコストの高い処理であり、マスタが一つの場合よりもパフォーマンスは落ちる。

(イ)原因

システムの統廃合時のデータクレンジングの怠り。

もともと別のシステムで利用されていたマスタ同士が、システム統合で同じドメインに存在するようになった場合。

(ウ)設計段階での対策

システム統廃合の際に、データの精査およびエンティティの統廃合をきちんと行う。

既存のエンティティを並べたのみでは、システム統合とは言えない。

(エ)設計以下の工程での対処

2.グレーノウハウ

(1)代理キー～主キーが役に立たないとき

(ア)概要

主キーの代理として、後付けでキーを設定すること。

連番などの論理的には意味をなさないものである場合が多く、論理モデルが分かりにくくなってしまう。

(イ)原因

・入力データに主キーにできるような一意キーが存在しない

・一意キーがサイクリック(1,2,3…100,1,2,3…のような感じ)に使いまわされている

・一意キーはあるが、途中で指す対象が変化する

上記のような場合に、場当たり的に設定されてしまう。

(ウ)設計段階での対策

入力データに主キーにできるような一意キーが存在しない場合、設計にできることはない。

業務使用の調整か、アプリケーションによる入力データの一意化を計る。

・一意キーがサイクリックに使いまわされている

・一意キーはあるが、途中で指す対象が変化する

上記の場合は、

・業務的意味のある「時間」を記録したタイムスタンプを表す列を設定する

・「開始」と「終了」の時間設定の列を追加し、必要・不必要に応じて2列の値を調整する

などの対処法がある。

(エ)設計以下の工程での対処

まず、入力データに主キーにできるような一意キーが存在しない場合には、設計段階においてもどうにもできないため、

必ず報告する。

次に、一意キーがサイクリックに使いまわされている場合への対処は、

代理キーの指し示す内容の名称(市区町村名など)を確認し、変化を見逃さないようにすること。

また、一意キーはあるが、途中で指す対象が変化する場合への対処は、

データ内容の大幅な変化などから、キーの指す内容の変化を疑う意識を持つこと。

(2)列待ちテーブル

(ア)概要

配列型を使わないで配列を表現しようとするため。

テーブルの列を増減が見込まれるため、動的な設計が必要になり、大変。

配列の要素数がレコードごとに異なる場合、NULLの仕様が避けられず、SQL文の結果の混乱の原因になる。

(イ)原因

配列型を使用するとデメリットが大きい。

第一正規形を保ちつつ強引に配列を表現するから。

(ウ)設計段階での対策

基本的に「行待ちテーブル」の構成を採用する。

「行待ちテーブル」では、配列における枝番を表す列を追加して配列を表すことで、列数の変化やNULLの存在を許さない。

利用する際には、列待ちテーブルの利点と欠点のバランスを見極めること。

すでに「原因」にて記述した欠点は踏まえたうえで、

・配列を表していることは伝わりやすい

・入出力のフォーマットと合わせやすい

といった利点を活用するためであれば利用は許される。

(エ)設計以下の工程での対処

可能であれば、列待ちテーブルを行待ちテーブルへ変換する。

列待ちテーブルと行待ちテーブルの変換は、SQL上でも容易に行える。

(3)アドホックな集計キー

(ア)概要

レコードを分類するために、場当たり的に集計キーを設定すること。

コード体系が短いスパンで変わったり、別のコード体系が必要になるため、サイズの大きいテーブルへのキーの追加によって

その規模をさらに大きくしてしまい、パフォーマンスの低下を招く。

(イ)原因

短絡的な思考で解決策を導こうとするため。

アドホックなキーを設定することは、一番早くシンプルな解決法ではあるが、論理構成において根本的な解決とは言えない。

(ウ)設計段階での対策

集計キーを追加したビューを利用すること。オリジナルのテーブルへのアクセスとほぼ変わらないコストであるため、パフォーマンスは落ちない。

ただし、多段ビューには注意。

(エ)設計以下の工程での対処

SQL文にて、CASE式を用いて分類を行う。CASE式は様々な句に対応して使えるため、柔軟分類が可能。

(4)多段ビュー

(ア)概要

「ビュー」機能を多層的に活用すること。

ビューの対象である基底テーブルにビューを持ってきてしまうと、処理が複雑化(増加)してパフォーマンスが落ちる。

仕様が複雑になり、管理が困難になる

(イ)原因

「アドホックな集計キー」の場合と同じく、短絡的な思考で解決策を導こうとするため。

すべての機能はトレードオフであり、利点が大きければ、同等の欠点が存在することを頭に入れておく。

(ウ)設計段階での対策

シンプルな設計を心がけること。

ビューを用いた処理はテーブルに左右されずに設計できてしまうため、多用してしまうことがある。

システムの保守性を常に念頭に置き、処理の構成をシンプルに保つこと。

バットノウハウ・グレーノウハウに関してまとめてみて、私が論理設計において最も重要で根本的であると感じたのはDOA(データ中心アプローチ)の考え方です。

考慮すべき要素(整合性、保守性、パフォーマンス,etc)が数多くあり、資金と期限が限られるシステム開発においては、絶対的な正解はないように感じます。

しかし、教本のおけるバットノウハウ・グレーノウハウ(間違い)にはそのほとんどに「プログラムの問題をデータで解決しようとする」という共通点が見受けられました。

データの整合性が保たれている限りは、不要な処理によるパフォーマンスの低下はなく、論理もシンプルに保たれます。

もちろんDOAの考え方だけで十分とは言えませんが、DOAをきちんと押さえることで、システムの設計として最低ラインはクリアできるのではないかと考えました。

「達人に学ぶDB設計徹底指南書」にて学んだ、データベースの重要性を念頭に置き、以降の学習に取り組んで行きます。

以上