RDBMS in Action

詳しくないとプロダクション環境で炎上しがちな RDBMS の勘所

2019/09/12

Seiya Yazaki

この話の目指すところ

RDBMS 理解度の壁:

プロダクションや運用保守で困らないシステムを作れる知識

∨

それっぽく動くものを作れる知識

実際のシステムで遭遇・見聞きした事象をもとに、
上記のスキマにある各種 RDBMS 知識を説明します。

RDBMS 本体の運用よりも、現実のアプリケーションにおける設計・実装上のハマリどころが中心。

例えばこういった話を知ってほしい

- 存在しないレコード、に大いに注意が必要であること
- 可変長/追記型の PostgreSQL と、固定長/in-place型の MySQL, Oracle の長所短所
- Locale, Collation や driver ライブラリによる型マッピングの注意点
- Failover や replication の特性, DB の選定への影響

章立て

今回は以下のトピックについて触れます:

1. レコードロック・トランザクション整合性
2. RDBMS の内部アーキテクチャによる性能上の考慮事項
3. Index
4. SELECT 文に関するその他のトピック
5. プリミティブ型の取扱い
6. 高可用性・高速性のためのシステム構成・DB 選定

RDBMS ある程度使ったことある前提になっているので、
そうでない方は「こんな話があるんだな」程度に聞いて
時が来たらこの話を思い出してやってください。
(本資料は公開予定)

Non-goal

コンテキストをある程度絞っています:

- 一般的な web サービスやそのバッチを前提に書きます
  - 多少の劣化も本気で許されないミッションクリティカル用途もスコープ外
- 事前の負荷試験をガチガチにやればいいんじゃん！という発想では語りません
- 専属 DBA (DataBase Administrator) が手厚くケアする世界観ではないです
- MySQL, Oracle, PostgreSQL のみに触れます
  - 他の DB は筆者が実運用経験ないため

レコードロック・トランザクション整合性

トランザクション分離レベル

SQL標準では、"トランザクション分離レベル" を 4 種類定義している:

↑ 弱い保証

- `READ UNCOMMITTED` : 他トランザクションの書きかけデータが見えてしまう
- `READ COMMITTED` : 他トランザクションが COMMIT したデータが見える
- `REPEATABLE READ` : 同じ SELECT 文を何回実行しても同じ結果を保証
- `SERIALIZABLE` : 他トランザクションの影響が全く見えないことを保証

↓ 強い保証

どのトランザクション分離レベルを使うか

実用的には...

- `READ UNCOMMITTED` : 保証が弱すぎ, まず使わない
- `READ COMMITTED` : (PostgreSQL, Oracle デフォルト) **筆者のおすすめ**
- `REPEATABLE READ` (MySQL デフォルト)
  - 一見便利そうだが、あまりおすすめしない (後述)
- `SERIALIZABLE` : 保証が強いが代償も大きすぎてあまり使わない
  - `トランザクションのリトライ` を回避困難なのでアプリ側の設計が厄介
  - RDBMS 内部の `ロック昇格` による予測困難な dead lock のリスク
  - 性能上のオーバーヘッドやロック周りでの DB 負荷も高い

※ MySQL だけデフォルトが違うので覚えておきましょう
MySQL でも `READ COMMITTED` は利用可能です。

レコードロック

以下のような処理を(非 `SERIALIZABLE` で)素朴に実装するとやばい:

1. SELECT
2. SELECT 結果をもとにアプリケーション側で色々計算
3. 計算結果をもとに INSERT/UPDATE/DELETE

1 と 3 の間に他のトランザクションが同じレコードを更新している場合、
その更新を無視してしまう結果になる (lost update)。

こういうケースではレコードロックをすることでそれを防ぐのが常識。

レコードロックする流れ

1. SELECT `FOR UPDATE`
    - この時点で対象のレコードがロックされる
2. SELECT 結果をもとにアプリケーション側で色々計算
3. 計算結果をもとに INSERT/UPDATE/DELETE

`FOR UPDATE` で行ロックすることで
1 と 3 の間に他のトランザクションが同じレコードを更新できなくなり、
整合性を確保できる。

Dead Lock

計算機における一般的なロックがそうであるように、行ロックも dead lock しうる。

対処法も一般的なロックに準ずる:

- 複数の行を個別にロックせずに、ロックの粒度を大きくする
  - e.g. 子テーブルの行を操作するときは、必ず親テーブルの行をロック
  - 本来は並行でできる処理が並行できなくなりえるデメリットはある
- ロック順序を決めて、常に特定の順序でロック獲得するようにする
  - 性能面では失うものがあまりない
  - 設計・実装がかなり面倒 & ミスりやすいというデメリットがある

粒度の大きいロックはやりすぎ注意

行ロックをまじめに設計するより、大きい単位でロックしてしまったほうが楽だが...

- 大きく設計した粒度を後から小さく改修するのは大変困難
  - トランザクション周りのロジックをほぼ作り直すことになる
  - 後から無理やり治すと大体モレや矛盾が生まれてしまうのでバグりやすい

特に「ユーザー単位で常に排他的ロック」は殆の場合粒度が過大。
バッチ処理が出てきたときにロック過剰で死ぬ。
リアルタイム処理をバッチが妨害してしまうので。

仕様的にアトミックだと思える範囲(e.g. 記事ごと)に留めるほうが吉。

粒度大きめのロックが活きる場面

複数の行を一括更新する UPDATE は「ロック順序を決める」ポリシーと相性が悪い:

- MySQL, PostgreSQL, Oracle いずれも一括 UPDATE 内部のロック順序は保証不能
- かといって、数万行オーダーの更新を 1 行ごとに SQL 発行するのはかなり遅い

こういうケースでは、一括更新対象の行の親レコード(単一のレコード)をロックする設計にしたほうが良い。

他にも、「存在しない」レコード区間をロックする目的でも有用 (後述)。

ロックのタイムアウト

他のトランザクションがロックしている場合、ロック獲得が待たされる。

待ち続けてしまうと DB サーバー・アプリサーバーの各種資源を圧迫するので、
明示的にタイムアウトすべきである。

やり方は DB によって違う, 以下などの方法で設定する:

- PostgreSQL: `lock_timeout` セッション変数や `FOR UPDATE NOWAIT`
- MySQL: `innodb_lock_wait_timeout` セッション変数
- Oracle: `FOR UPDATE WAIT n` や `FOR UPDATE NOWAIT`

存在しないレコードのロック・Gap Lock

以下のようなロジックは、かなり要注意:

1. SELECT `FOR UPDATE`
2. SELECT 結果の **レコードが「存在しない」か「存在する」かに依存** した計算
3. 計算結果をもとに INSERT/UPDATE/DELETE

存在しない行をどうロックするか・ロックできるのか という問題がある。
存在しない行をロックしない限り、他トランザクションで INSERT されて破綻する。

存在しない行に対する `FOR UPDATE`

`READ COMMITTED` の場合は、そもそも存在しない行に対するロックや保証がない。

`REPEATABLE READ` の場合:

- Oracle: そもそも `REPEATABLE READ` 非対応
- PostgreSQL: 存在しない行を `FOR UPDATE` ロックできない (後述)
- MySQL: Gap lock によって実現 (後述)

要するに MySQL 以外の 2 つでは、 `FOR UPDATE` で行が無かったとしても、
他トランザクションからの `INSERT` を防げない。

MySQL の Gap Lock (Next Key Lock)

MySQL の `REPEATABLE READ` での `FOR UPDATE` はテーブル上の**いずれかのindexの区間**を gap X-lock することで他トランザクションが当該区間に `INSERT` することを防ぐ。

しかし gap lock はクエリの `WHERE` 条件を正確に反映するのではなく、
現存するレコードとレコードの間の区間全体をロックする。

それゆえに、意外な広範囲がロックされ、dead lock の温床にもなる。

「MySQL は Next Key Lock なる挙動が怖い」という風説はこれに由来している。

(Next Key Lock は 行ロック + Gap Lock の総称)

`REPEATABLE READ` でなく `READ COMMITTED` を使えばこれは発生しない。

PostgreSQL と存在しない行のロック

PostgreSQL は存在しない行をロックしない (`SERIALIZABLE` 除く):

```
postgres=# BEGIN;
postgres=# SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

このタイミングで別トランザクションから test テーブルに INSERT

postgres=# SELECT FROM test FOR UPDATE; -- 全行ロック (0 rows)
postgres=# INSERT INTO test VALUES ('a', 'value of a');
ERROR:  duplicate key value violates unique constraint "test_pkey"
```

(公式ドキュメントがこの点についてはちゃんと書いていないので PostgreSQL 11.4 で確認した)

SELECT 結果の行の有無に依存するロジックは、 `REPEATABLE READ` でも厳しい。

存在しない行をロックしないのが最善

MySQL でも `READ COMMITTED` がオススメである (Gap lock 問題を回避できる)。
存在しない行はロックできない前提でプログラミングすることになる。

存在しない行のロックをしないで済ませるテクニック:

- 投機的に `INSERT` し、unique key / PK の重複エラーを catch する
  - この方法ならば行の存在有無を確実に保証できる
- 「行がなければ `INSERT`, あれば `UPDATE`」を atomic に行う RDBMS 機能を使う
  - `MERGE` (Oracle), `INSERT ... ON CONFLICT` (PosrgreSQL), `insert ... on duplicate key update` (MySQL)

上記いずれも難しい場合は、親レコードをロックするといったロック粒度の拡大で対処するのも手である。

行の存在・不存在に依存する処理いろいろ

行の存在・不存在に依存する処理は意外にあるので注意:

- `COUNT` や `SUM` に依存する処理
  - 行が増減すれば `COUNT` や `SUM` も変わる
  - 親レコードロックで対策するのが手堅く、意味的にも妥当
- 外部から入力される値に対する一意性担保
  - 例えばメールアドレスを一意の ID として使う場合
  - 投機的 `INSERT` 方式との相性が良い
- 状態遷移に対する制限
  - e.g. "Hoge 状態のレコードは 2 つ存在してはいけない"
  - 対処例: Unique key を貼る
  - 対処例: "Hoge 状態のレコードを作成・削除する際にロックするレコード" で排他制御

REPEATABLE READ, MVCC と更新系の相性

`REPEATABLE READ` と、DML, `FOR UPDATE` は相性悪い:

- `REPEATABLE READ` は過去のある時点のデータを常に返す
- `FOR UPDATE` や DML は最新のデータを取得・操作しないと不整合になってしまう

3 大 RDBMS の挙動:

- MySQL: `FOR UPDATE` は `REPEATABLE READ` であっても最新レコードが取得される
  - REPEATABLE 性が崩れる
- PostgreSQL: `SELECT` と DML, `FOR UPDATE` の間に更新があるとエラー
  - トランザクションを最初からやり直す必要あり
- Oracle: そもそも `REPEATABLE READ` がない

これも、 `REPEATABLE READ` より `READ COMMITTED` をおすすめする要因。

レコードロック・トランザクション整合性

まとめ:

- `REPEATABLE READ` より `READ COMMITTED` がおすすめ
  - MySQL では明示的に変える必要あり
- Dead lock は、ロック順の固定か大粒度ロックで対策するのがメジャー
- レコードの存在・不存在に依存する処理は要注意
  - 投機的 INSERT, Unique Key, 大粒度ロックなどで対処

閑話休題: RDBMS 以外のシステムとのリトライ整合性

DB と外部の API 等の間でデータ整合性を保つ必要がある場合、
外部の API は DB のトランザクションとは無関係に動くため
片方の更新が成功 & 片方が失敗 した場合の対処を考える必要がある。

一つの有用な方法として
`外部 API でのデータ更新などは commit 直前に行う`
というアプローチがある。
commit の失敗が非常に稀にしか起きないという 3 大 DB の性質を使っている。

他には `取り消し API` を呼び出すアプローチ(`代償トランザクション` 方式)もあるが
その場合、取り消し API の失敗ケースなどの考慮事項が発生してしまい、
複雑・テスト困難になりがちなので、あまりおすすめしない。

RDBMS の内部アーキテクチャによる性能上の考慮事項

RDBMS という漏れのある抽象化

RDBMS には内部設計の違いがある。
SQL である程度似たように使えるからといって同じではない。

特に意識すべき点:

- ロック・トランザクションの実挙動 (先述)
- DB サーバーの負荷特性

そして、内部設計の違いはアプリケーション設計時に十分考慮すべきである:

- 内部設計起因の課題・限界は ユニットテスト, 手作業 QA, 負荷試験 いずれの手法でも発覚しにくい
    - 大量かつ多種多様なトラフィックと蓄積されたデータがないと再現しにくい
- 特性の差がアプリケーションの設計に影響を与えてしまう (手戻りが大きい)

可変長文字列(`VARCHAR`)の扱い

- MySQL, Oracle: `VARCHAR(n)` は常に n 文字/byte 確保する
- PostgreSQL: `varchar` は内部的にも可変長

「とりあえず `VARCHAR(255)` ！」みたいなことは MySQL, Oracle では避ける:

- Disk I/O 帯域(iops)やキャッシュメモリを圧迫する
  - 症状としては I/O 詰まりになる
  - AWS RDS では I/O のネットワークの帯域幅もインスタンスサイズ比例
- CPU からのメモリ(ひいては disk)アクセス効率の悪化
  - L1/L2/... cache hit しにくくなり演算性能が下がる
  - 見た目上の症状として CPU 負荷が高く見える (実際は主記憶 I/O 待ち)

※ カラムの意味を曲げてまで無理やり短縮するべき、という話ではない

長大な文字列・バイト列

JSON とかを DB に入れると時に便利という気持ちは大変分かりますが...

- I/O 負荷とレスポンス速度の問題
  - DB はキャッシュヒットしないと超遅いので、メモリを食う
    - リアルタイム処理用途ではヒット率 90% 程度は保ちたい
  - ディスクやネットワークの帯域幅・最大速度にも限りがある
  - DB サーバーのメモリや I/O 速度・帯域はコスト高い
- BLOB/CLOB や 長大な文字列 は特別扱いになりオーバーヘッド増
  - LOB/長大な文字列 は専用領域に格納され、レコード本体に格納されないのでオーバーヘッドあり (3 大 DB 共通)
  - (PostgreSQL) 自動で圧縮するため CPU オーバーヘッドもあり
    - 特に master はスケールアウト困難なので CPU は限りある貴重な資源

Object Storage (S3, GCS) や KVS 等を最大限検討しましょう。

in-place 更新型と追記型

PostgreSQL : レコードの実体は immutable
MySQL, Oracle : レコードの実体は mutable

PostgreSQL の場合、 `UPDATE` であってもレコードの実体(tuple)を「作成」する。

古い tuple は削除・更新後にもゴミとして残るため、以下の課題が発生する:

- ゴミ tuple を消さないと table, index が肥大化し性能劣化する
  - 経時でゴミ tuple が大量に積み重なると論外レベルで遅くなる
- AWS Aurora のように最大時のサイズで課金されるケースで課金額も増えがち
  - 大量 update 時などに、前世代のゴミ + 新世代 のサイズで課金されるため

また、index の B-Tree 等にも無駄が発生する。

PostgreSQL の VACUUM

PostgreSQL には、ゴミ tuple に、再利用フラグを立てる `VACUUM` という処理がある。

フラグを立てることで、その tuple の領域が再利用の対象になる。

また、自動で `VACUUM` が行われる機能がありデフォルト有効(`AUTO VACUUM`)。

しかし VACUUM には限界が色々ある:

- PostgreSQL の tuple は可変長なので、断片化がネックになる
  - レコードの内容次第でサイズがバラバラ
    - 60byte の隙間に 80byte のレコードは入らない
    - 60byte の隙間に 40byte のレコードを入れるとデッドスペースが出来る
- Index の断片化(スカスカ状態)は解消されない

PostgreSQL の VACUUM 系処理の負荷と速度

VACUUM 処理の動き:

1. まず、テーブル全体を走査してゴミ tuple を列挙する
2. ゴミ tuple を片付ける

VACUUM 処理はテーブル全体を走査するため、DB 負荷の要因になる。
特にデフォルトの AUTO VACUUM 頻度では多すぎたり、
ピーク時間帯の性能圧迫要因になりがち (意外と CPU 30%-50% とか食ったりする)。

しかし、頻度を下げすぎると負のスパイラルに陥る:

- AUTO VACUUM にはタイムアウトがある, `1.` が進捗せずにタイムアウトすると...
- 次に AUTO VACUUM するまでの間にテーブルがより肥大化し...
- `1.` に要する時間が伸びてなおさらタイムアウト...

PostgreSQL の VACUUM 系をどうするか

PostgreSQL の根幹の構造に起因しており、頑張るしかない:

- DB 負荷の高いクエリ一覧を監視し、 `AUTO VACUUM` が現れたら対処する
  - 頻度を下げたり、ピーク時間帯には走らないようしたり
  - 先述の負のスパイラルには要注意
- ゴミ tuple, 断片化の程度を監視し、過度に荒れているテーブルは処置する
  - `AUTO VACUUM` の頻度を上げる
  - 手動で `pg_repack` / `VACUUM FULL` する (応急処置, 後述)
- 不要にレコードを update, delete しないようにし、ゴミ tuple の発生量を減らす
- 洗い替え(変化がないレコードすらも delete + insert)しないように設計する
  - しかし 1 record づつの update/merge 処理は遅い....
- PostgreSQL をやめる

PostgreSQL の VACUUM FULL, pg_repack

本来論で言えば、 `VACUUM` によってレコードに削除フラグを立てて、
それが自然に再利用されるのが望ましい。

しかし、それでは足りないことも実際よくある...

- VACUUM 頻度が不足しており肥大化してしまう事故
- 負荷が変動しがちなシステムで、突発的負荷に負ける
- 断片化が積もりに積もって死 (VACUUM は断片化解消しない)

`VACUUM FULL` / `pg_repack` で table, index 全体を強制的に再構築するしかない。
`pg_repack` PostgreSQL 拡張は trigger や一時テーブルを駆使してテーブルロックしない利点がある。権限周りで苦労があるが、基本的にはこちらを使ったほうが良い。

RDBMS の内部アーキテクチャによる性能上の考慮事項

まとめ:

- (MySQL, Oracle) 可変長文字列の定義長さを長くしすぎないように
- JSON とかを DB に入れるのはやめよう
- (PostgreSQL) 内部データ構造が断片化するしゴミも貯まるのが仕様
  - VACUUM で苦しめられることは未だによくあるが、気合で頑張るしかない
  - レコードの更新や洗い替えを減らせるならぜひ減らそう

Index

Index を語るにはこの余白は小さすぎる

Index 設計は、当然ながら性能上重要である。RDBMS は...

- DML 実行時に索引(index)を更新し
- クエリの実行時には最適な index を利用する計画(実行計画)を立案している

しかし、深く語ると際限がないので、今回は筆者的 FAQ トピックのうち
いくつかに触れるに留める。

なお、 USE THE INDEX, LUKE! という online book (日本語化もされている)があり、
非常におすすめである。
RDBMS を触る・テーブル設計する人には広くおすすめしたい。

B-Tree index のおさらい

特に指定しない場合、index は木構造として実現される。
(B-Tree をベースにしたデータ構造が使われる)

カラム (`A` , `B`) の複合 index の場合、以下のような木構造になる:

```
  ↑ Tree の Root 側
- Level 1: `A` カラムの値
- Level 2: `B` カラムの値
- Level 3: レコードへの参照
  ↓ Leaf 側
```

Skip scan / loose indexscan

```
  ↑ Tree の Root 側
- Level 1: `A` カラムの値
- Level 2: `B` カラムの値
- Level 3: レコードへの参照
  ↓ Leaf 側
```

これに対してカラム `B` の値だけで検索するとどうなるか？

- MySQL, Oracle : Level 1 は全走査するが、Level 2 を使って絞り込む
  - Skip scan (Oracle), Loose indexscan (MySQL) と呼ばれる
  - 例えば Level 1 に 10 通りの値しかないなら、かなり高速
- PostgreSQL : 上記 index は活用できない

(`B` , `A`) の順序に入れ替えるか、 `B` カラム単独の index があれば勿論最適。

Index が使われない！

理想的な index があるにも関わらず使われない、というトラブルもありがち。

殆どの場合、以下のどちらかが要因:

- テーブルの統計情報が古い
  - 実際のデータ分布とかけ離れた状態を前提に実行計画を組んでしまう
    - 特に、レコード数が僅かである前提で index scan より full scan が選択されるケースがありがち
- クエリ中の JOIN, サブクエリがあまりに多すぎる
  - 実行計画の探索空間が爆発するため、ヒューリスティック探索になったり一定時間で適当に実行計画作成されたりする (3 大 DB 共通)

なお、Oracle は他の 2 DB よりもオプティマイザが圧倒的に強力だが、
それゆえに上記の 2 問題の影響も顕著に出やすい (問題ない時が速いので)。

統計情報の明示的メンテナンス

DB の統計情報は DML 実行回数や更新行数などを元に自動で再統計される。

しかし、特に洗い替え(一括削除 + 一括投入)をする場合、
再投入完了後の状態で統計しないとズレやすいため、
明示的に `ANALYZE TABLE` といった SQL を発行するのは定石。

また、カラムの MIN, MAX がある程度変わった際も再統計しないと、
範囲外の値が「ない」前提で最適化されてしまう。なので定期的な再統計も大事。

また、あえて自動的な統計情報取得を無効にし、
負荷の低い時間帯に統計情報を一気更新するのも良い工夫。
ただし、データの傾向が時間帯で変わらないことが大前提。バッチに要注意。

Hint

3 大 DB それぞれ、特定の記述をすることで実行計画を指定する機能がある(hint)。

JOIN が多すぎて組合せ爆発し最適化が安定しないクエリなどでは、
hint を明示するのも選択肢ではあるが...

- クエリの内容や各テーブルのデータ傾向が変わった際に、hint が的外れになり逆効果にすらなる
- Hint の正当性を普段の CI テストなどで検証するのが困難
- クエリのバインド変数の値に応じて実行計画を最適化できなくなる
- DB のバージョンアップの恩恵が得られにくくなる

といった性質があるため、最大限オプティマイザに任せるべきである。
それが無理な場合は、まずシンプルなクエリに分解することを検討するのが良い。

保守性の観点で、**Hint は本当に最終手段**と考えるべし。

Index の更新オーバーヘッド

あらゆる `SELECT` に対して最適な index があれば `SELECT` は高速になるが...

- DML で更新するたびに index の更新コストが発生
  - B-Tree のリバランスなどによる CPU 計算コスト + メモリアクセス待ち
  - I/O の待ち時間や帯域の消費
- Index のディスク消費

Index を貼ること = 絶対的な善 ではない。

テーブル全体の数割以上のデータを取得するならば index を使わない full scan でも十分速い。
(オプティマイザも実際にそう判断することが多い)

複数 index の活用

`SELECT ... WHERE x = 1 AND y = 2` のようなクエリの場合、
(`x` , `y`) または (`y` , `x`) の複合 index があれば十分だが
あらゆるクエリに対して複合 index を用意するのは時として過剰。

一方で、 `x` , `y` カラムそれぞれ単独の index がある場合、
MySQL (>= 5), PostgreSQL (>= 8.1), Oracle であれば以下相当の最適化が可能である:

```
SELECT ... WHERE x = 1
    INNER JOIN
SELECT ... WHERE y = 2
```

`x` , `y` index それぞれで絞り込んだ結果の AND を取る。
INNERT JOIN 相当のコストが掛かるため複合 index には劣るが、有用ではある。

(MySQL は何故かこれが出来ないと思い込まれていることが...)

Index

まとめ:

- B-Tree index の構造と活用法は知っておこう
- テーブルの統計情報はちゃんとメンテすべし
    - データが大きく変わるとき + 定期的 に更新しよう
- 殺人的な JOIN 連発は人間だけでなくオプティマイザにも厳しい
    - クエリの簡素化を頑張ろう
- Hint 句は本当に最終手段
- Index を闇雲に増やすと更新系が重くなる
    - 複数の index を結合して活用してくれる機能もある
        - MySQL でそれができないという風説は誤り

SELECT 文に関するその他のトピック

一時ディスクの消費

クエリの処理中にディスク領域を消費することがある。
特に大量データの JOIN はディスク上に一時ファイル・一時テーブルが作られがち。

クラウドサービスで DB に最小限のディスク容量を割当てる場合、
クエリの実行に起因する一時ディスク消費は考慮が必要。

サービスによって細部は異なるが、DB のログ・監査ログや REDO, UNDO ログなどとも容量を食い合うことがあるため、それも含めて余裕があったほうが無難。

また、AWS Aurora は(今の所)一時ディスクの容量を自由に指定できず自動スケールもしないため、特に注意が必要である。

Prepared Statement の用途

Prepared statement を使う動機:

- 元々の目的: prepare した statement を何回も使い回せば、SQL のパースなどが 1 回で済むので DB 負荷が減る
- 実情: バインド変数機能(`:1` とか `?`)が SQL injection 対策になるので使う

筆者のオススメは後者のスタンス。前者の用途はおすすめしない:

- PS 使い回しはリーク(close 忘れ)の温床
  - PS は DB 上のリソースを確保してしまうので、リークすると痛い
- SQL のパースや実行計画の計算速度で困る状況があまりに稀
  - DB 側が高速化や statement cache などを頑張っているので

Prepared Statement, バインド変数 と 実行計画

実行計画・性能上の特性が 3 大 DB それぞれで異なる:

- MySQL : クエリ実行のたびに素直に実行計画を計算する
- PostgreSQL : セッション(DB接続)内で PS ごとに実行計画をキャッシュ
  - 最初の 5 回は素直に実行計画を計算
  - それ以降はバインド変数の値を無視した generic な実行計画を優先的に使う
- Oracle : DB インスタンス全体で PS の中身ごとに実行計画をキャッシュ
  - 初回のクエリのバインド変数の値を前提とした実行計画を計算し保存
  - n 回目にクエリが極端に遅くなった場合、n + 1 回目に実行計画を追加生成
    - つまり n 回目のクエリは犠牲になる

見ての通り PostgreSQL, Oracle は固有の癖があるので要注意。

プリミティブ型の取扱い

文字列の扱い

文字列は注意すべき点がある:

- 長さ
- いわゆる文字コード (appendix 参照)
- `=` や `DISTINCT`, `ORDER BY`, `LIKE`, ... の整合性
- DB の周辺ツールとの互換性

そして、実は DB それぞれで大変込み入っている...

文字列の長さ

ドキュメントなどを見つつ、適切な設定・型を使いましょう。

Unicode の非 LOB 型に絞って、かいつまんでまとめると...

- MySQL: `utf8` (<= 3byte/char), `utf8mb4` (<= 4byte/char)
  - 基本的には絵文字なども無難に扱える `utf8mb4` が便利だが...
  - 行長が最大 65,535 byte であることにも注意
- PostgreSQL: 1GB まで可能 (自動で圧縮された後のサイズ)
  - 長い文字列は行外に配置され、他 DB における CLOB/BLOB に近い性能特性になる点に注意
- Oracle: 1 カラム 4000 byte まで
  - 複数の varchar カラムに分けることで長さを稼ぐテクも一応ある
  - 行長は 65,535 byte まで

いわゆる文字コード

RDBMS において、いわゆる文字コードは以下の 2 箇所で問題になる:

1. DB で文字列を保存する際の文字コード
    - できるだけ Unicode 系にすべし
2. クライアントライブラリ・CLI の動作時の文字コード
    - `1.` と揃えるべし

`1.` と `2.` が揃っていない場合、文字コードの変換が行われる (3 大 DB 共通)。
しかし、その挙動に依存することは望ましくない:

- アプリの実装や運用作業でのデータ読み書きの際の文字コード変換と一致しない
    - 運用上の混乱、突き合わせ作業の実施困難、といった苦難を生む
    - もともと同一だった文字列が不一致になり得るため、バグの温床
- 意図せぬ変換が発生し、制御も難しい
    - PostgreSQL, Oracle ではテーブル・カラム単位の制御もできない

Appendix も参照のこと。

Locale, Collation

3 大 DB いずれも、以下の挙動を色々と制御できる:

- どの文字とどの文字を「同一」とみなすか
  - `=` や `DISTINCT`, `LIKE` などなどに影響
- 文字の順序
  - `ORDER BY`
- 合字(例: `㼿`, `㽘`)や絵文字などの扱い
  - それらの文字が入ってしまう場合に全体的に影響

気をつけないと詰むので、少なくとも **DB の新規作成時には明示的に制御**すること。

引き継いだシステムでここがダメダメなときのガッカリ感はすごい。

特に MySQL や PostgreSQL では、デフォルトのまま使うとヤバいことになる。
Appendix を参照されたし。

DB ごとの collation の各操作への影響

- MySQL
  - だいたい collation 通りに動く...はず
  - `LIKE` は collation を無視して文字単位で比較 (SQL 標準準拠)
- PostgreSQL
  - `C` ロケールならバイナリ一致, 性能面でも問題なし
  - それ以外の場合は collation に従う
    - index 作成時に `text_pattern_ops` などのオプションを要考慮
- Oracle
  - 大抵は `NLS_COMP=BINARY`。そうであればバイナリでのソート・一意性になる
  - そうでない場合は この表 とかを見ながら頑張れ...

バイナリ一致の collation 環境下での LIKE のコツ

DB の collation 機能を活用しようとすると**沼が深い**上に
アプリケーション側との挙動不整合にさいなまれる要因にしかならない。
アプリ側で文字列を正規化し、DB ではバイナリ的に文字列を扱うのが良い。

しかし `LIKE` 検索では「似たような」文字は検索でマッチさせたいであろう。

そのような検索用途では Unicode 正規形 `NFKC` を使うと便利。
(大抵の言語処理系で利用可能)

検索対象のカラムの中身と LIKE クエリ両方をアプリ側で正規化すれば、
合字・異体字を始めとする各種の表記ゆれを確実に吸収する検索が実現できる。

※ 正規化は非可逆操作なので、**正規化前のデータも残しておいたほうがいい。**

日時

文字列に加え、日時型も DB ごとに癖が大いにあるので要注意。

日時・日付型は driver ライブラリが勝手に変換していることが多いため、
使っている言語・ライブラリ・O/R mapper 依存の振る舞いがある:

- 精度の違い
  - 言語側の日時型がミリ秒/マイクロ秒単位なのに DB 側が秒単位、とか
  - DB 由来の日時とアプリ側の日時オブジェクトを `==` するのは危ない
- タイムゾーン
  - タイムゾーンを持たない日時型ではタイムゾーンに要注意
  - 特にクラウド環境ではサーバーの `NOW` は UTC だったりする
- 特殊値・異常値の取り扱い
  - PostgreSQL は OS 依存でうるう秒 (08:59:**60**) を返すことがある
  - MySQL の TIMESTAMP 型は NULL の代わりに `0000-00-00 00:00:00` になる

プリミティブ型の取扱い

まとめ:

- 文字列カラムや行の最大長には要注意
- DB にいわゆる文字コードを自動変換させるのはやめよう
  - 変換はアプリ側で明示的に行う方が、不整合や思わぬ変換を防げる
- DB に文字列の同一視を頑張らせるのはやめよう
  - `utf8_bin` (MySQL), `C` locale (PostgreSQL) などを明示的に設定しよう
  - 必要なら、アプリケーション側で NFKC 正規化などしよう
    - 正規化前のデータを捨てるべきではない点に注意
- 日時は 精度 と タイムゾーン と 特殊値・異常値 に要注意

文字列周りのもう少し踏み込んだ話は Appendix も参照のこと。

高可用性・高速性のためのシステム構成・DB 選定

Connection Pooling で良くなる点

クライアントから見たオーバーヘッドの削減:

- DNS クエリ時間 + TCP のハンドシェイク時間 (普通は微々たるもの)
- DB 側のコネクション作成所要時間
- 初期化処理 (user 認証など) のハンドシェイク時間
- Proxy を経由する場合、その接続オーバーヘッド
    - GCP の cloud SQL proxy や pgpool などなど

DB 側の負荷の軽減:

- コネクションの作成・初期化の処理コスト
    - PostgreSQL, Oracle はプロセスが立つので、作成が重くなりやすい

Connection Pooling の悪い点

クライアント側:

- セッション(接続)が使い回される
  - セッションの設定や変数がキレイにクリアされないと...
  - セッション内のリソース(prepared statement)などがリークしやすい
- Failover や接続断の対応が厄介
  - Connection を再利用する際に死活チェックが必要

DB 側:

- コネクションの維持コスト
  - 使われていないコネクションも維持されてしまうのでリソースの無駄
    - PostgreSQL, Oracle はプロセスが立つので特に

Connection Pooling どうするか

大体、以下のどれかのパターンになる:

1. 接続が重いかもしれない場合 (重い = 遅い or 接続処理の DB 負荷が無視できない)
    - Pooling する
    - MySQL の場合は結構軽いので、このケースにならないことも多い
        - ただし Google Cloud SQL Proxy 経由だとスパイク的に遅くなるので...
2. 接続が十分軽い しかし 対象の DB への接続数を抑制したい 場合
    - Pooling ライブラリを使うことで DB への最大同時接続数を制限する
    - しかし、コネクションは毎回再生成する (pooling はしない)
3. 接続が十分軽い かつ DB に大量接続・過負荷を掛けても許される 場合
    - Pooling しない

※ RDBMS 種別やネットワーク構成の前提なく要・不要を断定している情報に踊らされないこと。

Failover 対応

Failover には大体 2 パターンがある:

- IP アドレスが変わらず L2 ルーティングが変わる (Cloud SQL やオンプレに多い)
  - オンプレの場合は OS の ARP cache に注意を要することが多い
- ホスト名が変わらないが IP アドレスが変わる (AWS RDS)
  - JVM などの勝手に DNS キャッシュする処理系で要注意

Connection pooling する場合は先述の通り、
コネクション再利用時にコネクションが死んでいないかのチェックが必須。
(ライブラリ機能でできることが多いはず)

AWS RDS や Cloud SQL は手動で failover を発生させられるので、
アプリケーション側の振る舞いを一回テストしておくと大変良い。

Failover 方式とその特性

Failover の実現方式パターン:

1. DB インスタンス間でストレージを共有 (AWS RDS や Oracle RAC)
    - ストレージ(≒ EBS)が不調になると詰む
        - その点に対策したのが Aurora
2. master の更新ログを slave がリプレイ (Google Cloud SQL)
    - 更新ログのリプレイが間に合わないと、failover に時間がかかる
        - Cloud SQL が高可用性用途にあまり向かない理由
    - master はログを slave に送るだけ
3. DB インスタンス間で同期的に更新ログをレプリ (オンプレ DB でよくある)
    - master が死んだときに即座に replica が master になる
    - 更新ログのリプレイが間に合わないと master が低速化する

Read replica

DB の master ノードはスケールしにくいため、
SELECT 系負荷を slave (read replica) に逃がす手法もしばしば用いられる。

特に分析系クエリは、それによってアプリケーションの主要機能を劣化させるべきではないことが多いので、read replica に逃がす意義が大きい。

ただし、レプリケーションには遅延を伴うことが多いため、以下の用途では適さない:

- 最新のデータが見えないといけない要件 / ユーザーが混乱するケース
  - データを登録したのに見えないぞ？？等と混乱されることが許容できるか
- SELECT 結果を元に更新系処理を行うケース
  - 古いデータを元に更新処理を行うと、データの先祖返りを招く

Read replica の形態ごとの特性

1. 完全リアルタイムのレプリカ (同期的にレプリ, オンプレで使われることがある)
    - レプリケーションが詰まると master が低速化する
        - Read replica 用途では本末転倒なのでオススメできない
2. ストレージ共有型のレプリカ (Oracle RAC, Aurora)
    - 完全リアルタイム(Oracle), ミリ秒〜数秒程度の遅延 (Aurora)
3. 非同期レプリケーション (Cloud SQL, AWS RDS - not Aurora)
    - master の更新内容を slave でリプレイするため、詰まることがある
        - 時系列を再現する都合上、単一 or 少数のスレッドでリプレイするので、master の全力には追いつけないこと多し
    - PostgreSQL の場合、slave 側に long transaction があると tuple を消すリプレイができない構造上の問題でレプリが大幅遅延することもある

DB 選定の観点

DB 選定の観点は数多いが、ここまでに述べた性質が少なからず参考になるはず。

例えば...

- 低遅延な read replica が必要なら AWS Aurora が有用
- Google Cloud SQL 使うならば、非同期レプリケーションゆえの failover 遅延を意識する必要あり
- VACUUM による負荷(または肥大化・断片化による低速化) + レプリケーションの大幅遅延が望ましくないケースでは PostgreSQL は要注意

高可用性・高速性のためのシステム構成・DB 選定

まとめ:

- Connection Pooling する・しない の判断はシステム構成次第
  - `Connection Pooling どうするか` スライドを参照
- Failover のことを忘れずに
  - Pool されたコネクションの生存チェックが必要
  - AWS, GCP では手動で failover して挙動を確かめるべし
- Failover, Replication の特性・制限事項には RDBMS やクラウド環境での差がある
  - DB 選定をする上でも意識すべきポイントである

今回、深く扱わなかったトピック

- データのモデリング, テーブル設計
  - `第3正規形` とかは知っておこう
    - ※病的にテーブルを細切れにすると作者以外が不幸になるので程々に
  - 名前付けが怪しいときは、たいていモデリング自体が怪しい
  - テーブル設計とクエリパターンは表裏一体
- SELECT の効率化の様々な話: Use the index, Luke! をとりあえず読むべし
- RDB or NoSQL
  - KVS, Object Storage, Document DB はスケーラビリティやコスパが良い
  - 異種 DB を混用する場合、transaction 境界は分断するべからず
  - NoSQL にちゃんと取り組むと RDBMS の長所・短所の理解が深まる

終

> I do not know what I may appear to the world, but to myself I seem to have been only like a boy playing on the seashore, and diverting myself in now and then finding a smoother pebble or a prettier shell than ordinary, whilst the great ocean of truth lay all undiscovered before me.

> 私は、海辺で遊んでいる少年のようである。ときおり、普通のものよりもなめらかな小石やかわいい貝殻を見つけて夢中になっている。真理の大海は、すべてが未発見のまま、目の前に広がっているというのに

—— Isaac Newton

深く果てしない DB の沼で足掻いていくための知見のシェアを歓迎します！

Appendix. DB ごとの文字の取り扱いの沼

いわゆる文字コード

いわゆる `文字コード` は曖昧な概念であり、以下の概念に分離した方が良い:

- 文字集合 (e.g. Unicode, JIS 文字集合)
  - コードポイント(文字・文字列の最小構成要素)の集合
  - ※厳密には "符号化"文字集合 と 文字集合 の違いなどもあるだろうが踏み込まない
- エンコーディング (e.g. UTF-8, EUC-JP)
  - コードポイントの列をビット列でどう表現するかの定義
  - 基本的には文字集合に従属する (UTF-8 は Unicode 文字集合のための規格)
- Unicode における、書記素クラスター
  - 人間の文化的解釈に近い "1文字" の単位
  - 1 つ以上のコードポイントの列で表される (e.g. `か` + `゙` (結合文字の濁点))

上記 3 概念の区別を意識すれば、個別の RDBMS の諸概念・各種設定項目や実際の挙動も理解してゆけるだろう (ここでは各 DB の仕様の説明まではしない)。

実用上の留意点

- 文字集合とそのバージョン
  - Unicode にはバージョンがあり、DB のバージョンや設定しだいで異なる
    - Unicode バージョンで定義されていないものはうまく扱えないことも
  - JIS の文字集合についても同様
- コードポイントと書記素クラスターの関係
  - 何をもって "1 文字" とするかの問題

異体字,合字(例: `㍾`),絵文字 などを使うときは、DB の仕様をよく調べよう。

MySQL で特に気をつける設定

テーブルやカラム毎に設定を変えられてしまうが、
そうするメリットはめったに無いので、DB 全体でエンコーディングと collation を設定したほうが良い。

MySQL の unicode collation の考え方は独特, デフォルトで以下すべてを同一視する:

- 異体字など unicode 的に同じ文字を同一視/区別
- アクセントの有無を同一視/区別
    - **濁点・半濁点も "アクセント" 扱いで同一視される**
- 大文字・小文字を同一視/区別
    - **平仮名の大小も同一視される**

これらを同一視するメリットがない限り (アプリ側での正規化で対処できる限り)、
極力 `utf8_bin` を使ってすべて同一視しないようにするのが無難。

異体字をどうしても同一視したいなら `utf8mb4_0900_as_cs` といったものにする。

PostgreSQL で特に気をつける設定

ロケールを忘れずに `C` にするのが安定:

- PostgreSQL は文字コードを **OS の処理系に丸投げする** (環境依存する)
  - その仕組み上、パフォーマンスもあまり良くない...
- ダンプ・リストア(pg_dump)とかでも変えられない呪いになる
- `C` ロケール(要するにバイナリ的に一意判定・ソート)なら速いし確実

しかし、database はデフォルトでは OS のデフォルトロケールになっている。
AWS RDS もデフォルトが `en_US` (AWS DMS 等のツールで困る)。

`C` ロケールで database を新規作成するおまじないを使いましょう:

```
CREATE DATABASE hogehoge LC_COLLATE 'C' LC_CTYPE 'C' TEMPLATE template0;
```

Oracle

Oracle の文字周りの実装はかなり高機能だが...

- -> DB の設定は `AL32UTF8` + `AL16UTF16` が鉄板 (のはず)
  - Oracle は文字コードを DB 全体で設定する
  - 後者は `NVARCHAR` といった national なデータ型のための設定
- -> 接続時の設定は `NLS_LANG=AMERICAN_AMERICA.AL32UTF8` が無難 (Windows以外)
  - DB の設定と違う文字コードを指定すれば変換してもらえるが、`〜` 問題などの温床になるので変換はアプリ側に倒すほうがよい
  - なお、文字化けをエラーとして検知できない(`？` に変換されてしまう)。

より知るための資料:

- 公式のセミナー資料
- マルチバイト・キャラクタセット - SHIFT the Oracle