OSS-DB

OSS-DB Exam Silver
技術解説無料セミナー
2021/7/25 開催

主題	DBの環境設計・運用設計の基礎
副題	物理設計（テーブル設計、インデックス設計） 運用設計（DBメンテナンス、セキュリティ）

本日の講師

NTT テクノクロス株式会社
上原 一樹

LPI-JAPAN

© LPI-Japan / EDUCO all rights reserved.

講師紹介

■上原　一樹

- NTTテクノクロス株式会社　DBチーム所属
- PostgreSQL関連業務に従事
  - テクニカルサポート、コンサルティング
  - データベースの移行支援
  - 研修講師

■NTTテクノクロス株式会社（https://www.ntt-tx.co.jp/）

- DBチームでは、PostgreSQLを中心に各種サポートを提供。
  - オンプレのPostgreSQL、パブリッククラウド上のマネージドサービスを対象。
  - PostgreSQLに対応する周辺ツールについても対応（pg_statsinfo, pg_hint_plan等々）
- 技術ブログ（情報畑でつかまえて）では、DBをはじめ、様々な技術情報を投稿しています。

© LPI-Japan all rights reserved.

■OSSクラウド基盤トータルサービス

- https://www.ntt-tx.co.jp/products/osscloud/
- PostgreSQL移行・運用支援サービス、チケットサービスを提供しています。

© LPI-Japan all rights reserved.

■OSS-DBとは

オープンソースのデータベースソフトウェア「PostgreSQL」を扱うことができる技術力の認定です。様々な分野でPostgreSQLの利用拡大が進む中でOSS-DBの認定を持つことは、自分のキャリアのアピールにもつながります。

✓OSS-DB Goldは設計やコンサルティングができる技術力の証明

PostgreSQLについての深い知識を持ち、データベースの設計や開発のほか、パフォーマンスチューニングやトラブルシューティングまで行えることが証明できます

✓OSS-DB Silverは導入や運用ができる技術力の証明

PostgreSQLについての基本的な知識を持ち、データベースの運用管理が行えるエンジニアとしての証明ができます

✓対象のバージョンはPostgreSQL 11

© LPI-Japan all rights reserved.

認定試験の傾向と対策

■公式ドキュメントは使用するバージョンのドキュメントを読むべき！
- バージョンによって、機能の違いがあるため、採用するバージョンとあったものを読む。
- 本セミナー資料でも参考として、バージョン11のPostgreSQL文書のリンクを記載する。

■GUCパラメータやシステムテーブル・ビューは、単純に意味を覚えるのではなく、影響まで理解しなければならない

例 deadlock_timeout

意味:　ロック状態になった時にデッドロック検出処理を開始するまでの待機時間

影響:　値を小さくすると、デッドロックの検出は早くなるが、 実際にはデッドロックが発生していないのに検出処理が 動くことが多くなるため、CPUに無駄な負荷がかかる可能性も高くなる

■実機での動作確認は極めて重要
- 自分が予想した通りに動作しなければ、何かしら理解不足があるということ。

© LPI-Japan all rights reserved.

セミナーのテーマについて

■本日のテーマ

- 設計の基礎として、「PostgreSQLの設計要素には何があるのか」、「設計のポイントはどういったところにあるか」について解説を行います。
  - 過去のセミナで解説済み且つ、補足説明する必要がないものは、対象から外しています。
- 本セミナーはSilverの範囲を中心に解説を行います。ただし、普段の業務で活かせるようにPostgreSQLを利用する上でぜひ理解してもらいたいというものは、Gold範囲であっても紹介だけはさせていただきます。

■過去のオンラインセミナーのテーマ（どれも重要なテーマですので、ご活用ください。）

- 「PostgreSQLのバックアップ方法」
  - https://oss-db.jp/__/download/5f0fb75b37471d78510554e4/20200719-silver-01.pdf
- 「VACUUM、ANALYZEの目的と使い方」と「自動バキュームの概念と動作」
  - https://oss-db.jp/__/download/5f3de5a37ea72b5fa4113218/20200905-silver-01.pdf
- 「トランザクションの概念、SQLコマンド、Ver.2.0で追加された項目」
  - https://oss-db.jp/__/download/5f90e6a2c1fa8478bd3cd9a0/20201017-silver-01.pdf
- 「標準ツールの使い方、SQLコマンド（データ操作、データ型、インデックス）」
  - https://oss-db.jp/__/download/6073d1843701c51de41e20ee/20210410-silver-02.pdf
- 「運用管理、設定ファイル」
  - https://ferret-one.akamaized.net/files/60dae99cd38ec80f4ec003f8/20210629-silver.pdf

© LPI-Japan all rights reserved.

アジェンダ

黄色の項目：OSS-DB／Gold

■物理設計

- テーブル設計、インデックス設計
  - 容量設計（サイジング）
  - テーブル定義
    - データ型
    - HOT、FILLFACTOR
  - インデックス設計
  - テーブルスペース
- パラメータ設計
  - メモリ設計
    - shared_buffers, work_mem

■運用設計

- DBのメンテナンス
  - VACUUM、ANALYZE、FREEZE
  - autovacuum
- バックアップ・リカバリ
  - バックアップの種類
- セキュリティ
  - ROLE
  - GRANT, REVOKE

設計の詳細は、基本的にGoldの範囲となります。
本日はSilverの範囲で各項目から
設計の基礎となる要素について解説を行います。

© LPI-Japan all rights reserved.

事前準備1

■VMを用意する

- CentOS 7

```
[root@ossdb ~]# cat /etc/redhat-release
CentOS Linux release 7.9.2009 (Core)
```

■PostgreSQLのインストール

- リポジトリ登録

```
[root@ossdb ~]# rpm -ivh https://download.postgresql.org/pub/repos/yum/reporpms/EL-7-
x86_64/pgdg-redhat-repo-latest.noarch.rpm
```

- PostgreSQL 11のインストール

```
[root@ossdb ~]# yum install -y postgresql11-server
```

© LPI-Japan all rights reserved.

事前準備2

■OSユーザを切り替える

- postgresユーザはRPMを用いたインストールで自動で作成される。

```
[root@ossdb ~]# su - postgres
```

■環境変数の設定

- postgresユーザの .bash_profile は、~/.pgsql_profile を読み込む仕様になっている。

```
[postgres@ossdb ~]$ cat << EOF > ~/.pgsql_profile
PATH=/usr/pgsql-11/bin:$PATH
PGDATA=/var/lib/pgsql/local/11/data
export PATH PGDATA
EOF
```

- 再ログインまたは、sourceコマンドで環境変数を反映する。

```
[postgres@ossdb ~]$ source ~/.pgsql_profile
```

© LPI-Japan all rights reserved.

事前準備3

■DBクラスタ作成

- initdbコマンドでDBクラスタを作成する。

```
[postgres@ossdb ~] $ initdb -D ~/local/pg11/data --encoding=utf8 --no-locale
```

■PostgreSQLの起動

- pg_ctlコマンドで起動を行う。

```
[postgres@ossdb ~] $ pg_ctl -D ~/local/pg11/data start
```

- psqlでPostgreSQLに接続することができる。

```
[postgres@ossdb ~]$ psql postgres
psql (11.12)
Type "help" for help.

postgres=#
```

© LPI-Japan all rights reserved.

黄色の項目： OSS-DB/Gold

■物理設計

- テーブル設計、インデックス設計
  - 容量設計（サイジング）
  - テーブル定義
    - データ型
    - HOT、FILLFACTOR
  - インデックス設計
  - テーブルスペース
- パラメータ設計
  - メモリ設計
    - shared_buffers, work_mem

■運用設計

- DBのメンテナンス
  - VACUUM、ANALYZE、FREEZE
  - autovacuum
- バックアップ・リカバリ
  - バックアップの種類
- セキュリティ
  - ROLE
  - GRANT, REVOKE

まずは物理設計として、テーブル設計、インデックス設計、
また、それらに関連する項目について解説します。

パラメータ設計はGoldの範囲であるため、紹介のみとなります。

© LPI-Japan all rights reserved.

【物理設計】容量設計（サイジング） OSS-DB/Gold

（Silver範囲外であるため、本日は設計要素としての紹介のみ行う）

PostgreSQLの容量設計では、以下の領域について見積もりを行う必要がある。

■データ領域（テーブルやインデックスが使用する領域）

- データ型、レコード数、後述のFILLFACTORの設定から見積もることが可能。
- インデックスの見積もりは格納されている想定レコード数から算出が可能。
  - ただし、インデックスの種類によって構造は異なるため、同じデータでも種類によってサイズは異なる。

■WAL領域（Write-Ahead Logファイルを格納する領域）

- postgresql.confのmax_wal_sizeでおおよその値が決まる。
- min_wal_sizeの設定次第ではあるが、WALファイルの再利用、削除によって使用量は増減する。

■アーカイブ領域（アーカイブされたWALファイルを格納する領域）

- アーカイブWALの保存期間、その期間で処理する業務で発生するWAL量から見積もることが可能。

■一時領域（ソートやインデックス作成時に一時的に使用する領域）

- テーブルに格納されるデータと実行するSQLによって決まる。

■その他

- ログ出力：設定によっては大量のメッセージが出力されることもあるため、注意が必要。

© LPI-Japan all rights reserved.

【物理設計】データ型

■データ型のサイズ

- PostgreSQL文書でデータ型の格納サイズを確認することができる。

表8.2 数値データ型

型名	格納サイズ	説明	範囲
smallint	2バイト	狭範囲の整数	-32768から+32767
integer	4バイト	典型的に使用する整数	-2147483648から+2147483647
bigint	8バイト	広範囲整数	-9223372036854775808から+9223372036854775807
decimal	可変長	ユーザ指定精度、正確	小数点より上は131072桁まで、小数点より下は16383桁まで
numeric	可変長	ユーザ指定精度、正確	小数点より上は131072桁まで、小数点より下は16383桁まで
real	4バイト	可変精度、不正確	6桁精度
double precision	8バイト	可変精度、不正確	15桁精度
smallserial	2バイト	狭範囲自動整数	1から32767
serial	4バイト	自動増分整数	1から2147483647
bigserial	8バイト	広範囲自動増分整数	1から9223372036854775807

PostgreSQL文書：https://www.postgresql.jp/document/11/html/datatype-numeric.html

■データ型の選択について

- 適切なデータ型を選択することで様々なメリットがあり、逆に不適切なものを選択した場合にはデメリットが存在することを念頭に設計を行う必要がある。
- 次ページではデータ型選択で考慮すべき点をいくつか紹介する。

© LPI-Japan all rights reserved.

【物理設計】データ型

■文字列型

- PostgreSQLでは、固定長文字列を選択することによるメリットはない。
- 可変長文字列型を選択することでデータサイズが節約され、処理速度もわずかではあるが可変長のほうが長さチェックがない分有利となる。

■日付型

- タイムスタンプ情報は基本的に日付型に格納することを推奨する。
- 文字列型で格納した場合、提供されている演算子、日付関数を使用できない。

7/25の40日前を算出するには：

```
postgres=# CREATE TABLE bar (id int, create_time timestamp, drop_time text);
CREATE TABLE
postgres=# INSERT INTO bar VALUES (1, '2021-07-25 00:00:00', '2021-07-25 00:00:00');
INSERT 0 1
postgres=# SELECT create_time - interval '40 day' as timestamp FROM bar WHERE id = 1;
      timestamp
---------------------
 2021-06-15 00:00:00
(1 row)
postgres=# SELECT drop_time - interval '40 day' as timestamp FROM bar WHERE id = 1;
ERROR:  operator does not exist: text - interval
LINE 1: SELECT drop_time - interval '40 day' as timestamp FROM bar W...
                         ^
HINT:  No operator matches the given name and argument types. You might need to add explicit type casts.
```

文字列型では演算子を使用できない。

© LPI-Japan all rights reserved.

■数値型

- 必要以上にnumeric型を使用しない。特にOracleからの移行において、選択されるケースが多いがnumeric型はデータサイズが大きく、処理速度が他の数値型に劣る点には注意が必要。
- 処理速度の例として、それぞれの列でsum(),avg(),max(),min()に掛かる時間を比較すると、int型に比べて、15%程度低速となった。

```
postgres=# CREATE TABLE suchi (i int,n numeric);  -- 検証用にint型、numeric型の列を定義する
CREATE TABLE
postgres=# INSERT INTO suchi SELECT *,* FROM generate_series(1,100);  -- 各列には同じ値のデータを100件挿入する
INSERT 0 100
postgres=# SELECT * FROM suchi LIMIT 3 ;  -- テストデータを3件分確認する
 i | n
---+---
 1 | 1
 2 | 2
 3 | 3
(3 rows)
```

同じ値が格納してもINT型の列での処理に比べてnumeric型は低速となる。

```
-bash-4.2$ pgbench -n -T 10 -f agg_int.sql
...
number of transactions actually processed: 75718
latency average = 0.132 ms
tps = 7571.398165 (including connections establishing)
tps = 7573.133113 (excluding connections establishing)
```

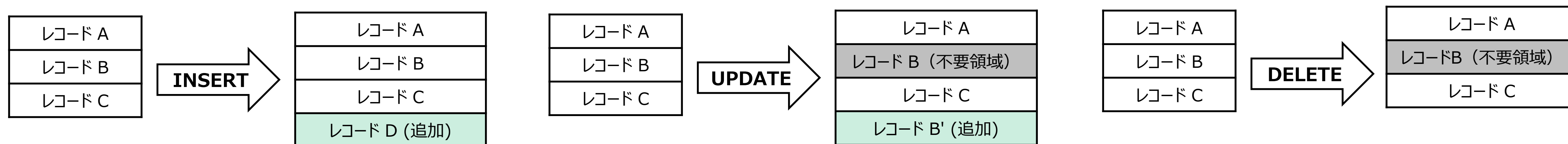
```
-bash-4.2$ pgbench -n -T 10 -f agg_num.sql
...
number of transactions actually processed: 64291
latency average = 0.156 ms
tps = 6426.681201 (including connections establishing)
tps = 6430.121681 (excluding connections establishing)
```

© LPI-Japan all rights reserved.

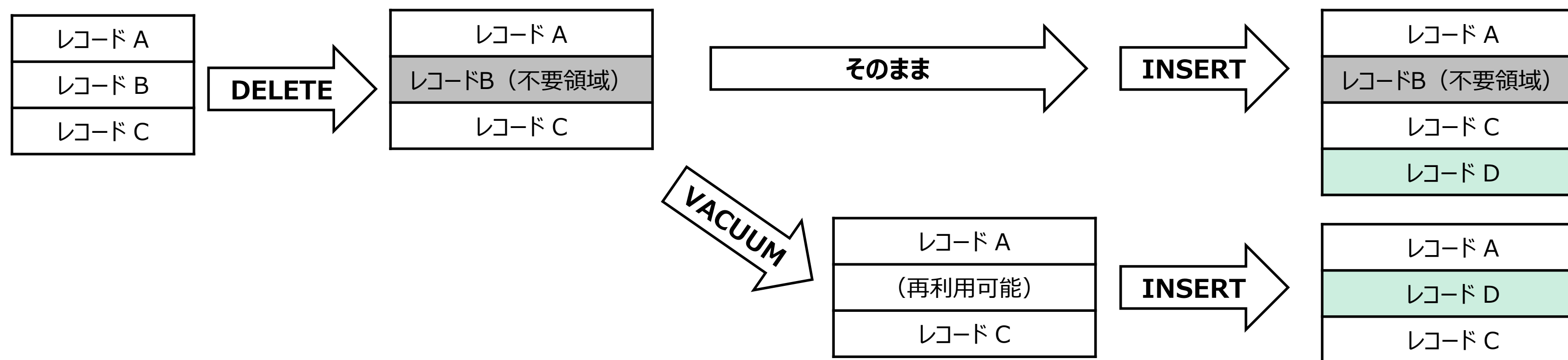
■追記型アーキテクチャについて

（Silver範囲外であるため、本日は設計要素としての紹介のみ）

- PostgreSQLは追記型のアーキテクチャであり、UPDATEやDELETEで削除された古いレコードは不要領域（dead tuple）として残る。
- 追記型アーキテクチャのイメージは以下のとおり。



- 不要領域（dead tuple）は、そのままでは再利用できない。
  - DBのメンテナンス処理（VACUUM）によって、不要領域の回収を行う必要がある。
  - VACUUMを適切に実施しないとデータ量が拡大し、テーブル肥大化等の原因となる。

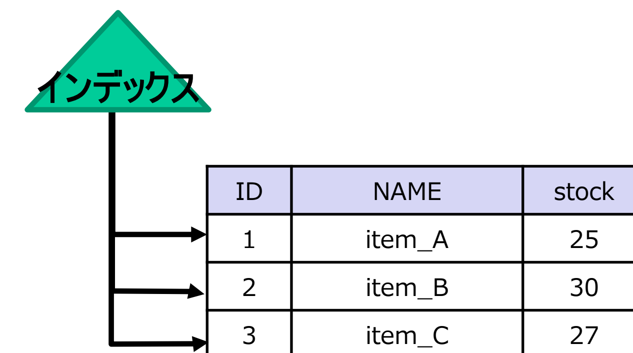


© LPI-Japan all rights reserved.

■Heap only tuple（HOT） とは

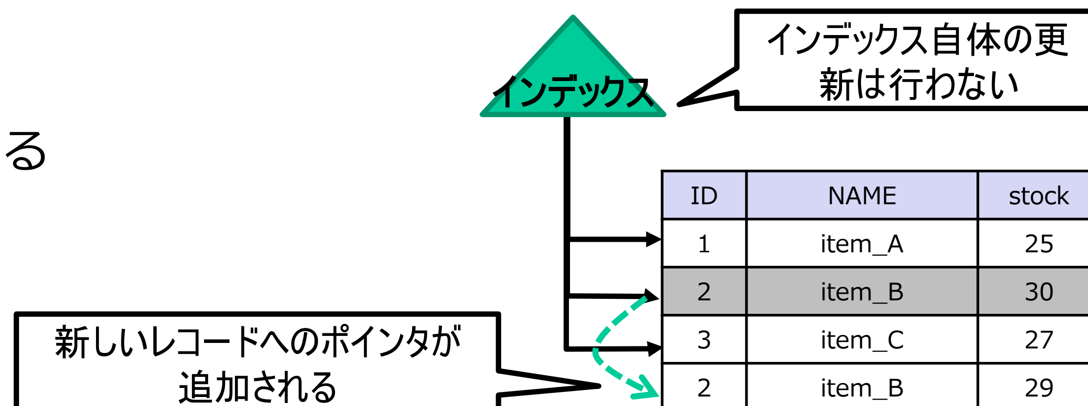
（Silver範囲外であるため、本日は設計要素としての紹介のみ）

- PostgreSQLでは、HOTという機能が提供されており、主な効果は次のとおり。
  - この機能を利用した更新処理ではインデックスエントリの追加処理が発生せず、更新性能が向上する。
  - 更新処理によって発生した不要領域はVACUUMによる回収を待たずに再利用可能となる。
- 本機能は以下の両方の条件を満たす場合に自動で有効となる。（無効にはできない）
  - 更新対象がインデックスを持たない列である
  - 更新対象と同じページ(*1)内に空き領域がある



ID	NAME	stock
1	item_A	25
2	item_B	30
3	item_C	27

ITEMテーブルを更新して、
item_Bのstockを1減らす



ID	NAME	stock
1	item_A	25
2	item_B	30
3	item_C	27
2	item_B	29

*1 PostgreSQLアクセスする最小単位（8KB）。ブロックとも呼ばれる。

© LPI-Japan all rights reserved.

■FILLFACTOR

（Silver範囲外であるため、本日は設計要素としての紹介のみ）

- HOTの条件の一つである空き領域を確保するための設定。（10～100で指定する）
- CREATE TABLEを実行する際に指定することができる。
  - デフォルトは100となっており、空き領域を確保しない。
  - https://www.postgresql.jp/document/11/html/sql-createtable.html
- FILLFACTORを指定することでHOTを利用することができるが、ページの充填率が下がるため注意が必要である。

■コマンド例

```
postgres=# CREATE TABLE foo (id int primary key, name text, stock int) WITH (FILLFACTOR=70);
CREATE TABLE
postgres=# ¥d+ foo
                                        Table "public.foo"
 Column |  Type   | Collation | Nullable | Default | Storage  | Stats target | Description
--------+---------+-----------+----------+---------+----------+--------------+-------------
 id     | integer |           | not null |         | plain    |              |
 name   | text    |           |          |         | extended |              |
 stock  | integer |           |          |         | plain    |              |
Indexes:
    "foo_pkey" PRIMARY KEY, btree (id)
Options: fillfactor=70
```

© LPI-Japan all rights reserved.

■HOTとFILLFACTORについて

（Silver範囲外であるため、本日は設計要素としての紹介のみ）

- FILLFACTORを使用する際、容量設計にどのように影響を与えるか確認する。
  - pgbenchのテーブルがFILLFACTORの設定有無でどのように変化するか確認する。
  - デフォルト（FILLFACTOR=100）の場合、pgbench_accountsのサイズは128MBとなる。

```
$ pgbench -i -s 10
$ psql postgres -c "¥d+"
                          List of relations
 Schema |        Name         |    Type    |  Owner   |    Size    | Description
--------+---------------------+------------+----------+------------+-------------
 public | pgbench_accounts    | table      | postgres | 128 MB     |
 public | pgbench_branches    | table      | postgres | 40 kB      |
 public | pgbench_history     | table      | postgres | 0 bytes    |
 public | pgbench_tellers     | table      | postgres | 40 kB      |
(4 rows)
```

  - FILLFACTOR=70に設定した場合、同じデータ量でテーブルサイズはどのように変化するか？

```
$ pgbench -i -s 10 --fillfactor=70
$ psql postgres -c "¥d+"
```

© LPI-Japan all rights reserved.

■HOTとFILLFACTORについて （Silver範囲外であるため、本日は設計要素としての紹介のみ）

- FILLFACTORによるテーブルサイズの変化
  - デフォルト（FILLFACTOR=100）の場合、pgbench_accountsのサイズは128MBだったが、FILLFACTORを設定したことでデータの充填率が下がり、テーブルサイズは182MBに増えている。

```
$ pgbench -i -s 10 --fillfactor=70 postgres
$ psql postgres -c "¥d+"
                          List of relations
 Schema |        Name        | Type  |  Owner    |  Size    | Description
--------+--------------------+-------+-----------+----------+-------------
 public | pgbench_accounts | table | postgres | 182 MB  |
 public | pgbench_branches | table | postgres | 40 kB   |
 public | pgbench_history  | table | postgres | 0 bytes |
 public | pgbench_tellers  | table | postgres | 40 kB   |
(4 rows)
```

© LPI-Japan all rights reserved.

【物理設計】インデックス設計

■PostgreSQLがサポートする主なインデックスの種類

名称	バージョン	説明
B-tree	ALL	一般的なインデックスで利用され、CREATE INDEXでのデフォルト。B-treeのみが一意性インデックスをサポートする。
hash	ALL	9.6までは非推奨。PostgreSQL10以降はWAL対応。一致検索のみサポート。
GiST	ALL	主に全文検索で使用する。GINとの違いはマニュアルを参照。
GIN	8.2～	主に全文検索で使用する。GiSTとの違いはマニュアルを参照。
BRIN	9.5～	データウェアハウス向けのデータに対して利用する。

PostgreSQL文書：
https://www.postgresql.jp/document/11/html/sql-createindex.html
https://www.postgresql.jp/document/11/html/textsearch-indexes.html

■インデックス設計のポイントについて

- 必要な分だけインデックスを定義する。
  - インデックスは作成した分、当該テーブルの更新性能に影響を与える。また、インデックスファイル自体がディスクサイズの圧迫の原因となるため、必要最低限だけ定義する。
- インデックスの種類によって、用途（強み、弱み）は異なる。
  - 種類によって、インデックスファイルのサイズ自体も異なる。

© LPI-Japan all rights reserved.

■不必要にインデックスを作成している場合の影響の例

OSS-DB /Gold

- 以下の例では、pgbench_accountsに対してインデックスを追加した場合の性能の変化を確認する。
- 確認としてpgbenchを3回実行したところ、対象テーブルの更新性能が20%ほど低下していることを確認した。

```
Table "public.pgbench_accounts"
  Column  |     Type      | Collation | Nullable | Default
----------+---------------+-----------+----------+---------
 aid      | integer       |           | not null |
 bid      | integer       |           |          |
 abalance | integer       |           |          |
 filler   | character(84) |           |          |
Indexes:
    "pgbench_accounts_pkey" PRIMARY KEY, btree (aid)
```

この状態で処理を流す

```
$ for i in 1 2 3 ; do pgbench -T 30 -r postgres | grep
  "UPDATE pgbench_accounts"; done
         0.115  UPDATE pgbench_accounts SET abalance =
  abalance + :delta WHERE aid = :aid;
         0.122  UPDATE pgbench_accounts SET abalance =
  abalance + :delta WHERE aid = :aid;
         0.122  UPDATE pgbench_accounts SET abalance =
  abalance + :delta WHERE aid = :aid;
```

```
Table "public.pgbench_accounts"
  Column  |     Type      | Collation | Nullable | Default
----------+---------------+-----------+----------+---------
 aid      | integer       |           | not null |
 bid      | integer       |           |          |
 abalance | integer       |           |          |
 filler   | character(84) |           |          |
Indexes:
    "pgbench_accounts_pkey" PRIMARY KEY, btree (aid)
    "pgbench_accounts_abalance_aid_idx" btree (abalance, aid)
    "pgbench_accounts_abalance_idx" btree (abalance)
    "pgbench_accounts_aid_abalance_idx" btree (aid, abalance)
```

3つのインデックスを追加

インデックスを追加した結果

```
$ for i in 1 2 3 ; do pgbench -T 30 -r postgres | grep
  "UPDATE pgbench_accounts"; done
starting vacuum...end.
         0.148  UPDATE pgbench_accounts SET abalance =
  abalance + :delta WHERE aid = :aid;
starting vacuum...end.
         0.150  UPDATE pgbench_accounts SET abalance =
  abalance + :delta WHERE aid = :aid;
starting vacuum...end.
         0.163  UPDATE pgbench_accounts SET abalance =
  abalance + :delta WHERE aid = :aid;
```

【物理設計】インデックス設計

■インデックスの種類によって、用途（強み、弱み）は異なっている。

- インデックスはそれぞれ異なった構造をしており、投入されるデータや利用用途によって向き不向きが存在する。
- 用途に応じて使い分けることで、より効果的な設計となる。
  - btree : 多くのケースで利用できるので基本的にはbtreeを選択することになる。
  - hash : 一致検索しかサポートしていないが、安定した性能を発揮する。
  - brin : データウェアハウスでの利用に適しているが、更新系には適していない。

■インデックスの種類によって、インデックスサイズは異なる。

- pgbench_accounts.aidに対して、btree, hash, brinを作成した結果は以下のとおり。

```
postgres=# ¥di+ pgbench_accounts_aid_*
                                        List of relations
 Schema |             Name             | Type  |  Owner   |      Table       | Size  | Description
--------+------------------------------+-------+----------+------------------+-------+-------------
 public | pgbench_accounts_aid_brin    | index | postgres | pgbench_accounts | 48 kB |
 public | pgbench_accounts_aid_btree   | index | postgres | pgbench_accounts | 21 MB |
 public | pgbench_accounts_aid_hash    | index | postgres | pgbench_accounts | 32 MB |
(3 rows)
```

© LPI-Japan all rights reserved.

■INDEX作成について

- インデックス作成時に取得されるロックについて
  - インデックス作成時は対象のテーブルに強いロック（SHARE）が取得され、INSERT、UPDATE、SELECTと競合するため、基本的にサービス中に実施することはできない。

- CONCURRENTLYオプション
  - このオプションを使用した場合、INSERT、UPDATE、SELECTと競合しない。
    - ロックレベルが下がり、 SHARE UPDATE EXCLUSIVEとなる。
  - 利用にあたって、以下の点に注意が必要となる。（詳細はPostgreSQL文書を参照。）
    - 実行中のすべてのトランザクションが終わるまで待機する。
      - » ロングトランザクションが存在する場合、いつまで処理が完了しない。
    - 通常の方式よりも総処理時間がかかり、また、完了するまでの時間が長くなる。
    - インデックス作成によりCPUや入出力に余分に負荷が掛かるため、他の操作や処理に性能影響を与える可能性がある。
    - 何らかの問題によって、CREATE INDEXが失敗すると「無効な」インデックスが残る。

PostgreSQL文書：
https://www.postgresql.jp/document/11/html/explicit-locking.html
https://www.postgresql.jp/document/11/html/sql-createindex.html#SQL-CREATEINDEX-CONCURRENTLY

© LPI-Japan all rights reserved.

■マルチカラムインデックス（複数列インデックス）

- インデックスは、テーブルの2つ以上の列に定義することができ、最大で32列までをサポートしている。
  - B-tree、GiST、GIN、BRINのみがマルチカラムインデックスに対応している。
- コマンド例：

```
postgres=# CREATE INDEX test2_mm_idx ON test2 (major, minor);  -- majorとminorカラムに対するインデックスを作成
```

- 設計のポイント
  - アプリケーションのワークロードを考慮し、どのようなインデックスを組み合わせて使用するかベストな形を探すこと。
    - 絞り込みでx列とy列に対する絞り込みが必ず行われるなら複数列インデックスが最善になることがある。
    - ただし、絞り込みでx列のみ、y列のみのケースが存在する場合、複数列インデックスはy列のみのケースでは複数列インデックスが有効に機能しない可能性がある。x列のみであれば複数列インデックスは有効であるが、x列の単一列インデックスに比べると低速になる。

PostgreSQL文書：
https://www.postgresql.jp/document/11/html/indexes-multicolumn.html
https://www.postgresql.jp/document/11/html/indexes-bitmap-scans.html

© LPI-Japan all rights reserved.

■部分インデックス

- インデックスを作成する際に対象の行を絞り込むことで、インデックスサイズを抑えることができ、性能を向上させることができる。

- 設計のポイント
  - 絞り込み条件において、頻出値がわかっている（決まっている）場合に有効。

- コマンド例：

```
postgres=# CREATE TABLE bar (id int, val double precision); -- 確認用のテーブル作成
CREATE TABLE
postgres=# INSERT INTO bar SELECT *,random() FROM generate_series(1, 3000); -- テストデータを挿入
INSERT 0 3000
postgres=# CREATE INDEX ON bar USING btree (id) WHERE NOT (id > 400 AND id < 599); -- 部分インデックス作成
CREATE INDEX
```

- 以下のケースでインデックスが有効なのはどちらか？

```
postgres=# EXPLAIN ANALYZE SELECT * FROM bar WHERE id = 700;
postgres=# EXPLAIN ANALYZE SELECT * FROM bar WHERE id = 500;
```

PostgreSQL文書：
https://www.postgresql.jp/document/11/html/indexes-partial.html

© LPI-Japan all rights reserved.

■部分インデックス

- コマンド例（続き）：

```
postgres=# EXPLAIN ANALYZE SELECT * FROM bar WHERE id = 700;  -- OKパターン
                                                   QUERY PLAN
------------------------------------------------------------------------------------------------------------------
 Index Scan using bar_id_idx on bar  (cost=0.28..8.30 rows=1 width=12) (actual time=0.020..0.021 rows=1 loops=1)
   Index Cond: (id = 700)
 Planning Time: 0.340 ms
 Execution Time: 0.038 ms
(4 rows)

postgres=# EXPLAIN ANALYZE SELECT * FROM bar WHERE id = 500;  -- NGパターン
                                          QUERY PLAN
--------------------------------------------------------------------------------------------------
 Seq Scan on bar  (cost=0.00..54.50 rows=1 width=12) (actual time=0.167..0.985 rows=1 loops=1)
   Filter: (id = 500)
   Rows Removed by Filter: 2999
 Planning Time: 0.077 ms
 Execution Time: 0.999 ms
(5 rows)
```

PostgreSQL文書：
https://www.postgresql.jp/document/11/html/indexes-partial.html

© LPI-Japan all rights reserved.

サンプル問題（インデックス設計）

■**OSS-DB Silver サンプル問題/例題解説 から本テーマに関するものを紹介します。**

- 2分後に解説を行うので、ぜひ回答を考えてみてください。

※この例題は実際のOSS-DB技術者認定試験とは異なります。

© LPI-Japan all rights reserved.

サンプル問題 3.46

■以下のSQL文でテーブルを作成した。
CREATE TABLE sample (id INTEGER, val TEXT);
このテーブルのid列に一意のインデックスを作成したい。
以下のSQL文で適切なものを2つ選びなさい。

A) ALTER TABLE sample ADD UNIQUE INDEX ON id;
B) ALTER TABLE sample ALTER COLUMN id UNIQUE;
C) ALTER TABLE sample ADD UNIQUE(id);
D) CREATE UNIQUE INDEX ON sample(id);
E) CREATE INDEX sample_id_unique ON sample(id);

セミナーでは解説していない部分になりますが、実行できる環境を持っている方は、ぜひ実際に試して、その結果をご確認ください。

実行する際は、以下のようにテーブルを作成しておく必要があります。
CREATE TABLE sample(id int);

■引用元

- https://oss-db.jp/sample/silver_development_03/46_141001

© LPI-Japan all rights reserved.

【物理設計】テーブルスペース（テーブル空間）

■PostgreSQLのファイル構成のおさらい

- PostgreSQLでは、テーブルやインデックス等のオブジェクトは実ファイルとして管理されている。
  - デフォルト設定でテーブルを作成すると $PGDATA/base配下に作成される。
  - ファイルパスを調べるための関数も提供されている。

```
postgres=# SELECT pg_relation_filepath(oid) FROM pg_class WHERE relname = 'foo';
 pg_relation_filepath
----------------------
 base/13881/34005
(1 row)
```

環境によってファイルパスは異なる

■テーブルスペースの設計について

- テーブルスペースを使用することで、データベースオブジェクトのレイアウトを制御することができる。
  - IOが集中するテーブルやインデックスを高速・高可用なディスク配置することや、ディスクを分離してIO負荷を分散させるかどうかがポイントなる。

PostgreSQL文書：
https://www.postgresql.jp/document/11/html/manage-ag-tablespaces.html

© LPI-Japan all rights reserved.

【物理設計】テーブルスペース（テーブル空間）

■コマンド例：

```
postgres=# CREATE TABLESPACE fastspace LOCATION '/ssd1/postgresql/data';  -- テーブルスペースを作成する
CREATE TABLESPACE
postgres=# CREATE TABLE foo(i int) TABLESPACE fastspace;  -- 作成したテーブルスペース上にオブジェクトを作成する
CREATE TABLE
postgres=# SELECT pg_relation_filepath(oid) FROM pg_class WHERE relname = 'foo';  -- ファイルパスを参照する
            pg_relation_filepath
---------------------------------------------
 pg_tblspc/34119/PG_11_201809051/13881/34120
(1 row)
postgres=# ¥q
-bash-4.2$ ll local/11/data/pg_tblspc/  # シンボリックリンクで指定された領域が使用されていることを確認できる
total 0
lrwxrwxrwx 1 postgres postgres 21 Jun 28 16:35 34119 -> /ssd1/postgresql/data
```

```
postgres=# CREATE TABLESPACE fastspace2 LOCATION '/ssd2/postgresql/data';  -- テーブルスペースを作成する
CREATE TABLESPACE
postgres=# SET default_tablespace = fastspace2;  -- デフォルトで使用されるテーブルスペースを変更する
SET
postgres=# CREATE TABLE foo(i int);  -- 設定変更後はTABLESPACEの指定は不要となる
CREATE TABLE
```

PostgreSQL文書：
https://www.postgresql.jp/document/11/html/manage-ag-tablespaces.html

© LPI-Japan all rights reserved.

サンプル問題（テーブルスペース（テーブル空間））

■OSS-DB Silver サンプル問題/例題解説 から本テーマに関するものを紹介します。

- 2分後に解説を行うので、ぜひ回答を考えてみてください。

※この例題は実際のOSS-DB技術者認定試験とは異なります。

© LPI-Japan all rights reserved.

サンプル問題 3.113

■**テーブルスペース（テーブル空間、tablespace）の使い方として、間違っているものを1つ選びなさい。**

A) CREATE TABLESPACE コマンドでテーブルスペースとして使用するディレクトリを指定するが、このディレクトリは既存で、かつ空でなければならない。

B) テーブルを作成する際、 CREATE TABLE (…テーブル定義…) TABLESPACE テーブルスペース名; のようにすれば、指定したテーブルスペースが使用される。

C) パラメータ default_tablespace でテーブルスペース名を指定すると、CREATE TABLE などで作成されるオブジェクトは、設定されたテーブルスペースを使用する。

D) テーブルスペース内にオブジェクトを作成するには、そのテーブルスペースについての CREATE 権限が必要である。

E) DROP TABLESPACEコマンドでテーブルスペースを削除するとき、その中にあるオブジェクトも自動的に削除される。

■**引用元**

- https://oss-db.jp/sample/silver_development_06/113_200326

© LPI-Japan all rights reserved.

■メモリ設計

（Silver範囲外であるため、本日は設計要素としての紹介のみ）

- メモリは大きく分けて、共有メモリとプロセスメモリが存在する。それぞれの役割については以下のとおり。
  - 共有メモリ
    - 各プロセスから共通で利用できる共有のメモリ領域。PostgreSQL起動時に確保される。
    - 共有バッファ、WALバッファ、Visibility Map(*1)、Free Space Map(*2)に分けて使用される。
  - プロセスメモリ
    - バックエンドプロセス（セッション）ごとに確保される領域。確保したプロセスのみが利用できる。
- これらは postgresql.conf で設定することができる。
  - shared_buffers, wal_buffers, work_mem 等々

■設計のポイント

- 性能に直結する設計要素であるため、特に注意が必要となる。
- PostgreSQLはディスクから読み込んだデータは共有バッファ上で操作を行うため、共有バッファが小さすぎるとディスクアクセスが発生し、処理性能が低くなる。
- 扱うデータサイズ、実行するSQLに合わせて値を決定する必要がある。

*1 Visibility Map：データの可視性を管理する領域。VACUUM処理等で利用される。
*2 Free Space Map：対象のリレーション無いで利用可能な領域を追跡するための情報をもつ領域。

© LPI-Japan all rights reserved.

運用設計

黄色の項目： OSS-DB /Gold

■物理設計

- テーブル設計、インデックス設計
  - 容量設計（サイジング）
  - テーブル定義
    - データ型
    - HOT、FILLFACTOR
  - インデックス設計
  - テーブルスペース
- パラメータ設計
  - メモリ設計
    - shared_buffers, work_mem

■運用設計

- DBのメンテナンス
  - VACUUM、ANALYZE、FREEZE
  - autovacuum
- バックアップ・リカバリ
  - バックアップの種類
- セキュリティ
  - ROLE
  - GRANT, REVOKE

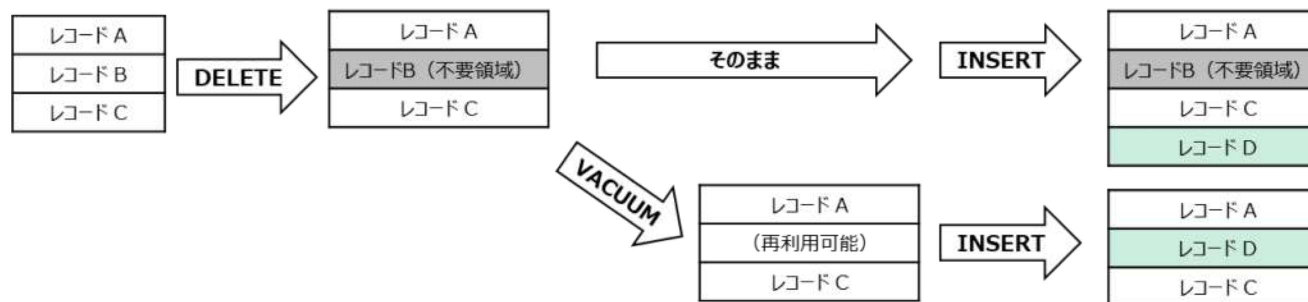
続いて運用設計について解説します。

© LPI-Japan all rights reserved.

【運用設計】DBのメンテナンス

■VACUUM：データベースの不要領域を回収する

- 追記型アーキテクチャを採用しており、削除された領域はVACUUM処理で回収するまで再利用することができない。



- 設計のポイント
  - 不要領域を適正なタイミングで回収し、テーブルの肥大化を発生させない。
    - VACUUMを過剰に実施しても、無駄なリソースの消費につながる。

- 現在存在する不要領域を確認する方法
  - n_dead_tupとしてシステムカタログから確認することができる。

```
postgres=# postgres=# select n_dead_tup from pg_stat_user_tables where relname = 'pgbench_accounts';
 n_dead_tup
------------
       3850
(1 row)
```

3850レコードが不要領域として存在

© LPI-Japan all rights reserved.

■FREEZE：タプルを凍結しXIDを再利用する

- PostgreSQLはトランザクション処理をトランザクションID（XID）で管理しており、32bitの数値（約40億）を再利用しながら使用している。
  - XIDの回収ができずに、使い切ってしまうとDBは最悪停止してしまう。
  - XIDはDBクラスタ単位で管理されている。

```
postgres=# SELECT datname,age(datfrozenxid) FROM pg_database ;
  datname  |  age
-----------+-------
 postgres  | 18713
 template1 | 18713
 template0 | 18713
(3 rows)
postgres=# VACUUM FREEZE;
VACUUM
postgres=# SELECT datname,age(datfrozenxid) FROM pg_database ;
  datname  |  age
-----------+-------
 postgres  |     0
 template1 | 18713
 template0 | 18713
(3 rows)
```

age()で、XIDがどれだけ消費されているか確認できる。

VACUUM FREEZEによって、XIDの回収をおこなったことでカウントが「0」に戻った。

© LPI-Japan all rights reserved.

■FREEZE（続き）

- 設計のポイント
  - FREEZE処理は、VACUUM FREEZEで明示的に実行することもできるが、VACUUM処理を実行した際、裏でFREEZE処理は行われている。
    - VACUUM処理の中でFREEZEされる範囲はチューニング要素。
  - 注意が必要な要素ではあるが、VACUUM処理が十分に実行できていれば、FREEZEをシビアに考える必要はない。

```
postgres=# SELECT datname,age(datfrozenxid) FROM pg_database WHERE datname = 'postgres';
 datname  |    age
----------+------------
 postgres | 1610593462
(1 row)
postgres=# VACUUM;
VACUUM
postgres=# SELECT datname,age(datfrozenxid) FROM pg_database WHERE datname = 'postgres';
 datname  |   age
----------+----------
 postgres | 50000000
(1 row)
```

XIDが消費された状態

テーブルを指定しない場合、DB全体にVACUUMが行われる。

5000万XID分を残して、FREEZEされた。

© LPI-Japan all rights reserved.

【運用設計】DBのメンテナンス

■ANALYZE：データベース内のテーブルの内容に関する統計情報を収集する

- PostgreSQLは与えられたSQL問い合わせに対して、統計情報やデータ分布等の情報を利用して、その処理を実行するためのベストな手順（実行計画）を作成している。
  - 実行計画は以下のようにEXPLAIN文で確認することができる。

```
postgres=# EXPLAIN SELECT pc.relname, pl.mode FROM pg_locks pl ,pg_class pc
            WHERE pl.relation = pc.oid AND locktype = 'relation';
                                   QUERY PLAN
---------------------------------------------------------------------------
 Hash Join  (cost=12.56..34.74 rows=5 width=96)
   Hash Cond: (pc.oid = l.relation)
   ->  Seq Scan on pg_class pc  (cost=0.00..20.18 rows=518 width=68)
   ->  Hash  (cost=12.50..12.50 rows=5 width=36)
         ->  Function Scan on pg_lock_status l  (cost=0.00..12.50 rows=5 width=36)
               Filter: (locktype = 'relation'::text)
(6 rows)
```

JOINの方法、スキャンの方法等が書かれている。

- 設計のポイント
  - 統計情報がうまく利用できないケースがあり、そういった状況においては不適切な実行計画が採用され、性能問題等の原因になることがある。
  - バックアップからリストアした直後は統計情報がクリアされるため、統計情報の取得が必要。
  - データ分布が変わるような大量のデータ操作を行った後には、統計情報の更新が必要。

© LPI-Japan all rights reserved.

【運用設計】DBのメンテナンス

■これらのDBのメンテナンス処理は、必要なタイミングで処理を行う必要がある。

- PostgreSQLでは、これらのメンテナンス処理（VACUUM、ANALYZE、FREEZE）を自動で行うautovacuum機能（自動バキューム機能）が備わっている。

■設計のポイント OSS-DB /Gold

- autovacuumによるメンテナンス処理の閾値
  - autovacuumが起動する際の閾値を制御できる。INSERT、UPDATE、DELETE等にデータが操作された量を監視しており、閾値を超えた場合にautovacuumのworkerプロセスが起動される。
- autovacuumによるVACUUMの処理負荷の制御
  - 全力で不要領域の回収を行わせるか
  - IO負荷を軽減するために加減しながら回収を行うか
- DBのメンテナンス処理を阻害する要因を理解する
  - 阻害の主な要因としてあげられるのが、ロングトランザクション（トランザクションを開始した状態が長時間続いている処理）が存在するケース。PostgreSQLでは、アクセスされる可能性があるデータは回収して再利用することができない。不要領域もXIDも同様。

© LPI-Japan all rights reserved.

サンプル問題（DBのメンテナンス）

#OSS-DB

■**OSS-DB Silver サンプル問題/例題解説 から本テーマに関するものを紹介します。**

- 2分後に解説を行うので、ぜひ回答を考えてみてください。

※この例題は実際のOSS-DB技術者認定試験とは異なります。

© LPI-Japan all rights reserved.

サンプル問題 1.109

■**バキューム（VACUUM）と自動バキュームの違いの説明として適切なものを3つ選びなさい。**

A) VACUUMはオプションで指定しなければ不要領域の回収のみを実行するのに対し、自動バキュームは不要領域の回収と解析（ANALYZE）の両方を実行する。

B) VACUUMは対象とするテーブルのリストを指定する必要があるが、自動バキュームは挿入・更新・削除されたデータが一定量以上のテーブルが自動的に対象となる。

C) VACUUMは一時テーブル（temporary table）を対象とできるが、自動バキュームでは一時テーブルは対象外である。

D) VACUUMはオプションを指定することでファイルサイズを縮小することができるが、自動バキュームには同等の機能がない。

E) VACUUMはOSの機能などを使って定期的に自動実行するようにできる。自動バキュームはPostgreSQLサーバの稼働中で、トランザクションの実行量が少ないときに自動的に実行されるので、OSの機能の設定は不要である。

■**引用元**

- https://oss-db.jp/sample/silver_management_06/109_201029

© LPI-Japan all rights reserved.

【運用設計】バックアップ・リカバリ

■PostgreSQLが提供するバックアップ

- 論理バックアップ
  - pg_dumpコマンド、pg_dumpallコマンドが提供されている。
  - 論理バックアップではPostgreSQLの設定ファイルのバックアップはできない。

- 物理バックアップ
  - オンラインで実施する場合は、pg_basebackupコマンド, pg_start_backup(), pg_stop_backup()が提供されている。
  - オフラインであれば、DBを停止後にcpコマンド等で取得することができる。
    - オフラインで取得するバックアップはコールドバックアップと呼びます。

■PostgreSQLが提供するバックアップ方式と各方式の主な特徴

項目	pg_dump, pg_dumpall	pg_basebackup	pg_start_backup, pg_stop_backup
概要	DBを論理的な形に変換する方式	レプリケーション機能を利用した方式	停止点を作って行う方式
取得条件	オンライン	オンライン	オンライン
バックアップ対象	オブジェクト単位で指定可能	DBクラスタ全体	DBクラスタ全体
WALによるリカバリ	不要	必要	必要

© LPI-Japan all rights reserved.

【運用設計】バックアップ・リカバリ

■バックアップ・リカバリの手順について

- 詳細な手順については過去のセミナー資料を参照。
  - 「PostgreSQLのバックアップ方法」
  - https://oss-db.jp/__/download/5f0fb75b37471d78510554e4/20200719-silver-01.pdf

■設計のポイント

- バックアップ方式の選定
  - 何をバックアップする必要があるか？
    - どういった障害に備える必要があるかによって取得対象は変わる。
  - バックアップ、リカバリに掛けてもよい時間がどれだけあるか？
    - 方式によってバックアップの取得に掛かる時間、バックアップからの復旧に掛かる時間は異なる。

  ※システム要件に合わせた方式を選択する必要がある。

- バックアップファイルの管理
  - 不要になったもののバックアップファイルを定期的に削除する（アーカイブWAL等）
  - バックアップファイルの管理機能はPostgreSQLでは提供されていないため、サードパーティ製品の利用が必要か検討が必要。

© LPI-Japan all rights reserved.

サンプル問題（バックアップ・リカバリ）

■OSS-DB Silver サンプル問題/例題解説 から本テーマに関するものを紹介します。

- 2分後に解説を行うので、ぜひ回答を考えてみてください。

※この例題は実際のOSS-DB技術者認定試験とは異なります。

© LPI-Japan all rights reserved.

サンプル問題 1.101

■コールドバックアップ（データベースを停止した状態で取得するバックアップ）について適切なものを３つ選びなさい。

A) cp などのコマンドを利用して、データベースクラスタのディレクトリのコピーを作成する。
B) tar などのコマンドを利用して、データベースクラスタのディレクトリのアーカイブを作成する。
C) rsyncなどのコマンドを利用して、データベースクラスタのディレクトリの複製をネットワーク上の他のホストに作成する。
D) pg_basebackup コマンドを利用して、データベースクラスタのベースバックアップを作成する。
E) pg_dumpall コマンドを利用して、データベースクラスタ全体のバックアップを作成する。

■引用元

- https://oss-db.jp/sample/silver_management_06/101_200212

© LPI-Japan all rights reserved.

【運用設計】セキュリティ

■ROLE

- データベースオブジェクトを所有することができ、データベース権限を持つことができる実体。
- ロールは、使用状況に応じて「ユーザ」、「グループ」、もしくは、その両方として利用することができる。
  - 例えば、DBに接続する際に使用するユーザは、ログイン権限を持つロールであり、実体は同じものである。

PostgreSQL文書：https://www.postgresql.jp/document/11/html/sql-createrole.html

■自動で作成されるロールについて

- RPMでインストールした場合、デフォルトで postgres ロールが作成される。
  - postgresロールは強力な superuser権限 が付与されているため、取り扱いには注意が必要。
- デフォルトロール
  - 監視機能や権限をもつユーザにしか参照できない情報へのアクセスを容易に設定するための権限のセット。これを利用することで権限の管理が少しやりやすくなる。
  - https://www.postgresql.jp/document/11/html/default-roles.html

■ロールの作成について

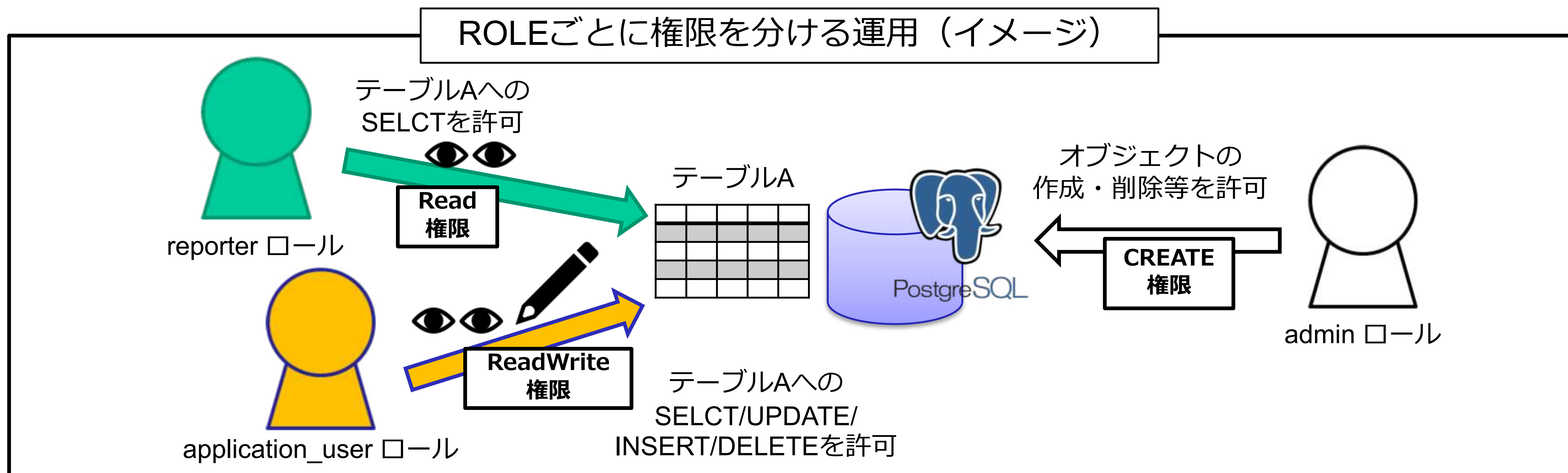
- CREATE ROLE文で作成可能。似たSQLとして、CREATE USER文があるが作成されたロールがデフォルトでログイン権限を持つか持たないかの違いしかない。
- PostgreSQLにはスキーマが存在するが、ロールとは異なる権限の管理構造であり、ロール作成時に自動作成されることはない。（特にOracleとは扱いが異なるので注意が必要。）

© LPI-Japan all rights reserved.

【運用設計】セキュリティ

■設計のポイント

- superuserは権限として強力すぎるため、通常の業務では使用しない。
  - superuserは制限なく全てのオブジェクトにアクセス可能で、OSのroot権限相当の扱い。
- データベースオブジェクトにアクセスするための最小権限を付与する。
  - 例えば、DBの管理タスクを行うためのロール、Read Onlyのロール、ReadWriteのロール等、複数のロールを作成し、それらのロールをアプリケーションが使用するユーザ（ログイン権限をもつロール）に付与する。



© LPI-Japan all rights reserved.

■GRANT（権限付与）/REVOKE（権限取り消し）

- ロールに対して、データベースオブジェクトに対する権限を付与する。
- 指定したロール内のメンバ資格を他のロールに付与します。

PostgreSQL文書：https://www.postgresql.jp/document/11/html/sql-grant.html

■コマンド例：

```
postgres=# CREATE USER application_user;  -- LOGIN権限を持つ application_user ロールを作成する
CREATE ROLE
```

```
-bash-4.2$ psql postgres -U application_user   # application_userでPostgreSQLに接続
psql (11.6)
Type "help" for help.

postgres=> SELECT count(*) FROM pgbench_accounts;  -- 権限の無い参照はエラー
ERROR:  permission denied for table pgbench_accounts
postgres=> ¥dp pgbench_accounts
                                Access privileges
 Schema |       Name       | Type  |     Access privileges      | Column privileges | Policies
--------+------------------+-------+----------------------------+-------------------+----------
 public | pgbench_accounts | table | postgres=arwdDxt/postgres |                   |
(1 row)
```

Accessprivilegesの読み方：
role名=与えられた権限/権限付与したrole名

権限：
a：INSERT
r：SELECT
w：UPDATE
d：DELETE … 詳細はマニュアル参照

■コマンド例（続き）：

```
postgres=# CREATE ROLE readwrite;
CREATE ROLE
postgres=# GRANT SELECT, UPDATE, INSERT, DELETE ON ALL TABLES IN SCHEMA public TO readwrite;
GRANT
postgres=# GRANT readwrite TO application_user ;
GRANT ROLE
```

```
-bash-4.2$ psql postgres -U application_user
psql (11.6)
Type "help" for help.

postgres=> SELECT count(*) FROM pgbench_accounts;
  count
---------
 1000000
(1 row)
postgres=> ¥dp pgbench_accounts
                                    Access privileges
 Schema |       Name       | Type  |     Access privileges      | Column privileges | Policies
--------+------------------+-------+----------------------------+-------------------+----------
 public | pgbench_accounts | table | postgres=arwdDxt/postgres+|                   |
        |                  |       | readwrite=arwd/postgres    |                   |
(1 row)
```

pgbench_accountsに対して、readwireteロールはINSERT, SELECT, UPDATE, DELETE権限をpostgres ロールから付与された　という情報が記録されている。

© LPI-Japan all rights reserved.

サンプル問題（ROLE）

■OSS-DB Silver サンプル問題/例題解説 から本テーマに関するものを紹介します。

- 2分後に解説を行うので、ぜひ回答を考えてみてください。

※この例題は実際のOSS-DB技術者認定試験とは異なります。

© LPI-Japan all rights reserved.

サンプル問題 1.106

■**PostgreSQLにおけるCREATE USERとCREATE ROLEの違いの説明として、適切なものを2つ選びなさい。**

A) CREATE ROLEは一般ユーザでも実行できるが、CREATE USER の実行はスーパーユーザ権限が必要である。

B) CREATE ROLEで作成したロールを GRANT コマンドによりユーザに付与することができるが、CREATE USERで作成したユーザをGRANTで他のユーザに付与することはできない。

C) CREATE ROLEで作成したロールのLOGIN属性はNOLOGIN、CREATE USERで作成した場合はLOGINがデフォルトになっている。

D) CREATE USERでユーザを作成したら、同じ名前のスキーマが自動的に作成されるが、CREATE ROLEの場合はスキーマが作成されない。

E) OSのコマンドラインからCREATE USERと同等の機能を実行するために createuser というコマンドが提供されているが、createrole というコマンドは提供されていない。

■**引用元**

- https://oss-db.jp/sample/silver_management_06/106_200708

© LPI-Japan all rights reserved.

■お問い合わせ■

NTTテクノクロス株式会社　ソフト道場

https://www.ntt-tx.co.jp/products/soft_dojyo/

© LPI-Japan all rights reserved.