aws DEV DAY ONLINE JAPAN

2020.10.21

F-5

マルチテナントデータの再構築 (Amazon RDS for PostgreSQL)

株式会社アンチパターン
塚本 岳史

In Partnership with intel

自己紹介

塚本 岳史

株式会社アンチパターン

取締役兼ソフトウェアエンジニア

イヌとネコならイヌ派

本セッションのゴール

SaaSシステムの更改時に、DBの構成を変更してパフォーマンスを改善した方法についての共有

アジェンダ

- 更改にあたっての課題
- データベース統合をするに至るまで
- データベース構成変更の検討
- データベース移行

aws DEV DAY ONLINE JAPAN

更改にあたっての課題

In Partnership with intel

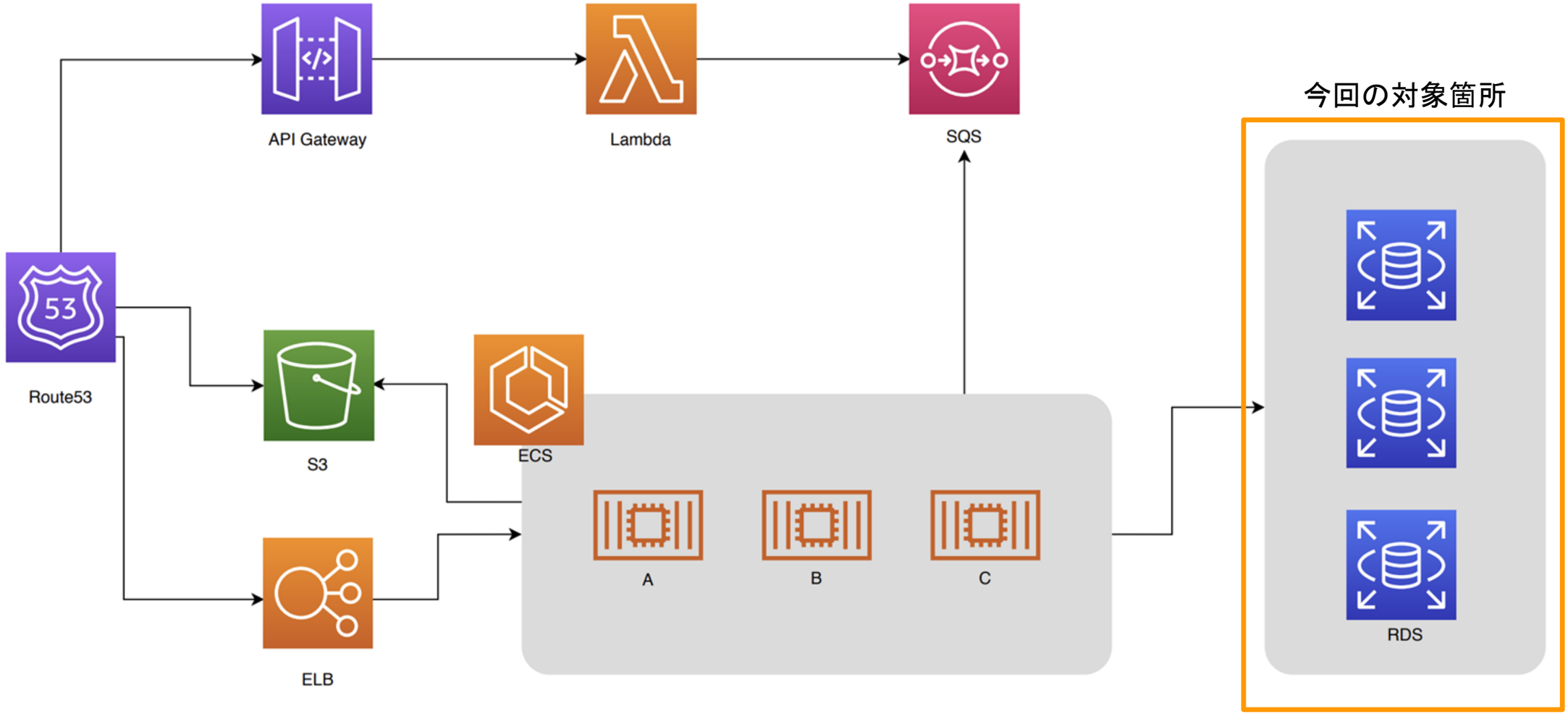
課題概要

・PHP5.6を利用しているため、EOLが迫っている。そのためPHP7.xへマイグレーションをすることが必要

※今回は詳細割愛します

・事業の成長に伴い顧客数が増加した事でデータ量も多くなった。それに従いデータベースのパフォーマンスに問題が出てきたため、データベース処理の性能改善が必要

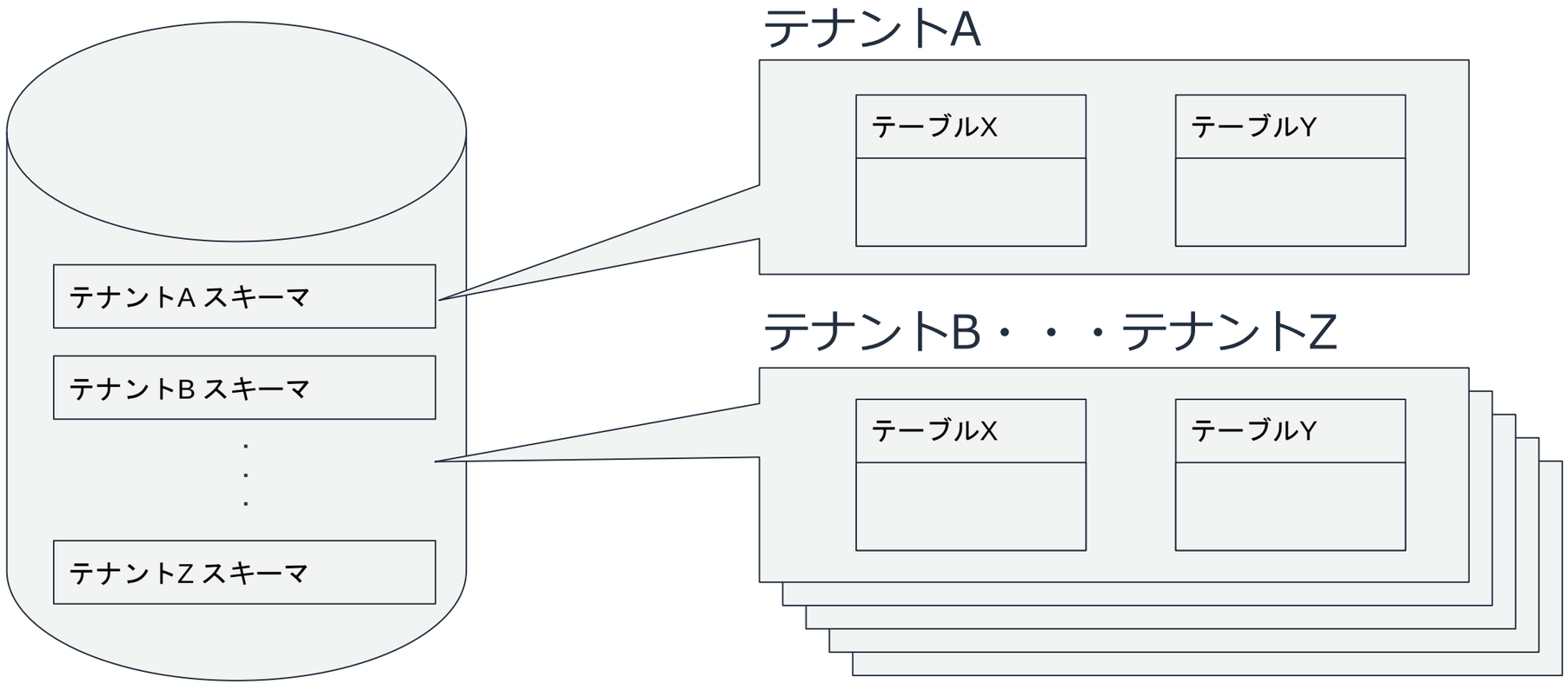
変更前の構成



In Partnership with intel

変更前のデータベース構造

RDS(PostgreSQL)を利用し、テナント単位でスキーマを作成
各テナントに同じ構成のテーブルが存在



In Partnership with intel

aws DEV DAY
ONLINE JAPAN

データベース統合をするに至るまで

In Partnership with intel

困っていたポイント

- 常にDBが高負荷
- 経年によりDBインスタンスが増加
- 運用コストが増加

常にDBが高負荷

PostgreSQLの統計情報コレクタを使用して定点観測を実施
読出しタプル数、I/O量、書込みI/O量、更新量を確認

- pg_stat_user_tables
  各テーブルへ実施された表スキャンやインデックススキャン回数などの情報

- pg_statio_user_tables
  各テーブルへ実施されたブロックアクセス状況

- pg_stat_bgwriter
  バックグラウンドライタプロセスの活動状況に関する統計情報が確認
  クライアントから送られたデータは一旦共有バッファに書き出され、
  バックグラウンドライタを通してテーブルファイルやインデックスファイルに反映されるため

常にDBが高負荷

利用した統計情報コレクタとその値
https://www.postgresql.jp/document/11/html/monitoring-stats.html

名前	説明
seq_scan	このテーブルに対する表スキャンの実行回数
seq_tup_read	表スキャンによって読みとられた行数
idx_scan	このテーブルに対するインデックススキャンの実行回数
idx_tup_fetch	インデックススキャンで読み取られた行数
n_tup_ins	INSERTされた行数
n_tup_upd	UPDATEされた行数
n_tup_del	DELETEされた行数

pg_stat_user_tables

名前	説明
heap_blks_read	テーブルブロックの読み込み数
heap_blks_hit	キャッシュヒットしたテーブルブロック数
idx_blks_read	インデックスブロックの読み込み数
idx_blks_hit	キャッシュヒットしたインデックスブロック数

pg_statio_user_tables

名前	説明
buffers_clean	バックグラウンドライタにより書き出されたバッファ数
buffers_backend	バックエンドにより直接書き出されたバッファ数
buffers_alloc	当られたバッファ数

pg_stat_bgwriter

常にDBが高負荷

スキーマ／テーブルごとに各値を検証

schemaname	relname	seq_scan	seq_tup_read	idx_scan	idx_tup_fetch
tenantA	tableA	3770	602553573	10219.0	3072251.0
tenantB	tableA	525	285799857	0.0	0.0
tenantC	tableA	466	219892121	0.0	0.0
tenantA	tableB	679	181412975	80804.0	218823.0
tenantC	tableD	77	135814923	319001.0	6301541.0
tenantD	tableA	344	122049303	0.0	0.0
tenantE	tableE	50	120382508	1772.0	4062.0

常にDBが高負荷

負荷がかかっていそうなクエリの調査

クエリの特徴としてテーブル結合や副問合せも多く使用

テーブルスキャンが発生しているクエリの特定は実行計画が必要となるため、log_min_duration_statementの閾値を下げて、全スロークエリを取得

EXPLAIN (ANALYZE, BUFFERS) select ・・・・発行するクエリ

常にDBが高負荷

スキーマ／テーブルごとに各値を検証した結果

・全体のテーブル数に対して、I/Oが発生するテーブルは少ない
・単位時間当たりのI/O量が大きい

❏ 読み込みI/O （ブロックI/O）
テーブル数: 89,297
I/Oが発生したテーブル数: 2,532
I/O量合計: 1611.06 GB （**537.02GB/h、152.75MB/s**）

❏ レコード更新（論理I/O）
テーブル数: 89,297
I/Oが発生したテーブル数: 3,063

❏ 書き込みI/O （ブロックI/O）
チェックポイント（容量）: 26.0 回/時間
チェックポイント書き出し: **5.4 GB/時間**
バックグラウンドライタ書き出し: **2.9 GB/時間**
バックエンド書き出し: 0.8 GB/時間

In Partnership with intel

常にDBが高負荷

チェックポイントチューニング結果

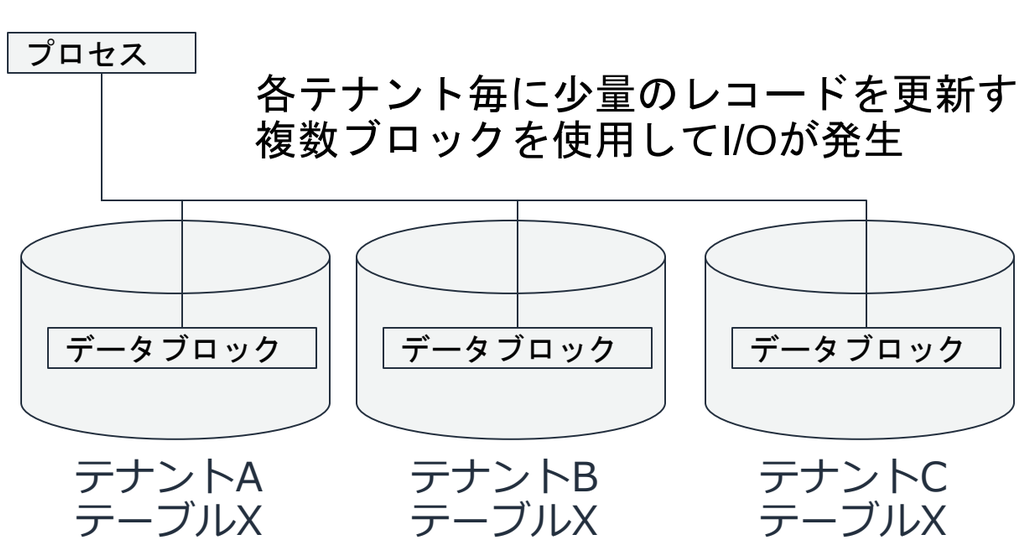
	チューニング無し	チューニング有り
チェックポイント（タイムアウト）	0.0 回/時間	4.0 回/時間
チェックポイント（容量）	26.0 回/時間	3.0 回/時間
チェックポイント書き出し	5.4 GB/時間 (0.2GB/1回)	0.9 GB/時間 （0.3GB/1回)
バックグラウンドライタ書き出し	2.9 GB/時間	9.0 GB/時間
バックエンド書き出し	0.8 GB/時間	0.8 GB/時間

- 単位時間当たりの書き出し容量はほぼ変化無し
- 書き込み容量が減らないのは、一部しか更新を行わないブロックが大量に存在している可能性がある
- この原因は、テーブルがテナントごとに分割されていることにあると想定

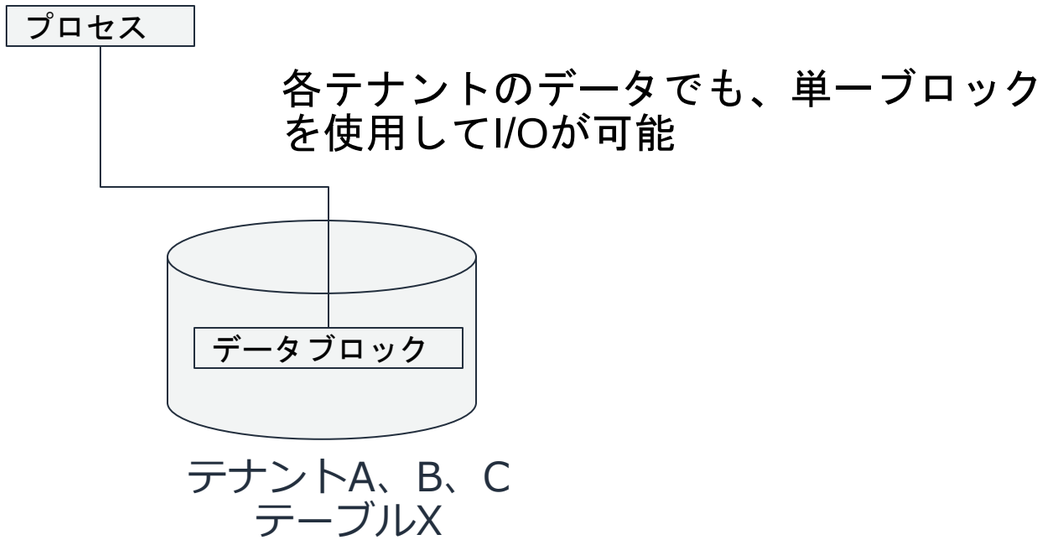
常にDBが高負荷

テナントごとにテーブルを分割している現在の場合

- 各テナントのデータアクセスのたびに、複数のブロックを読み書きすることになる
- PostgreSQLは8KB単位でI/Oするため、少量のデータアクセスでもI/Oのオーバーヘッドが大きい
- 読み書きするブロックの数が増加すると、キャッシュに乗り切らなくなる可能性が高まる
  （結果として、ディスクI/Oが増加する）



テーブル分割している場合

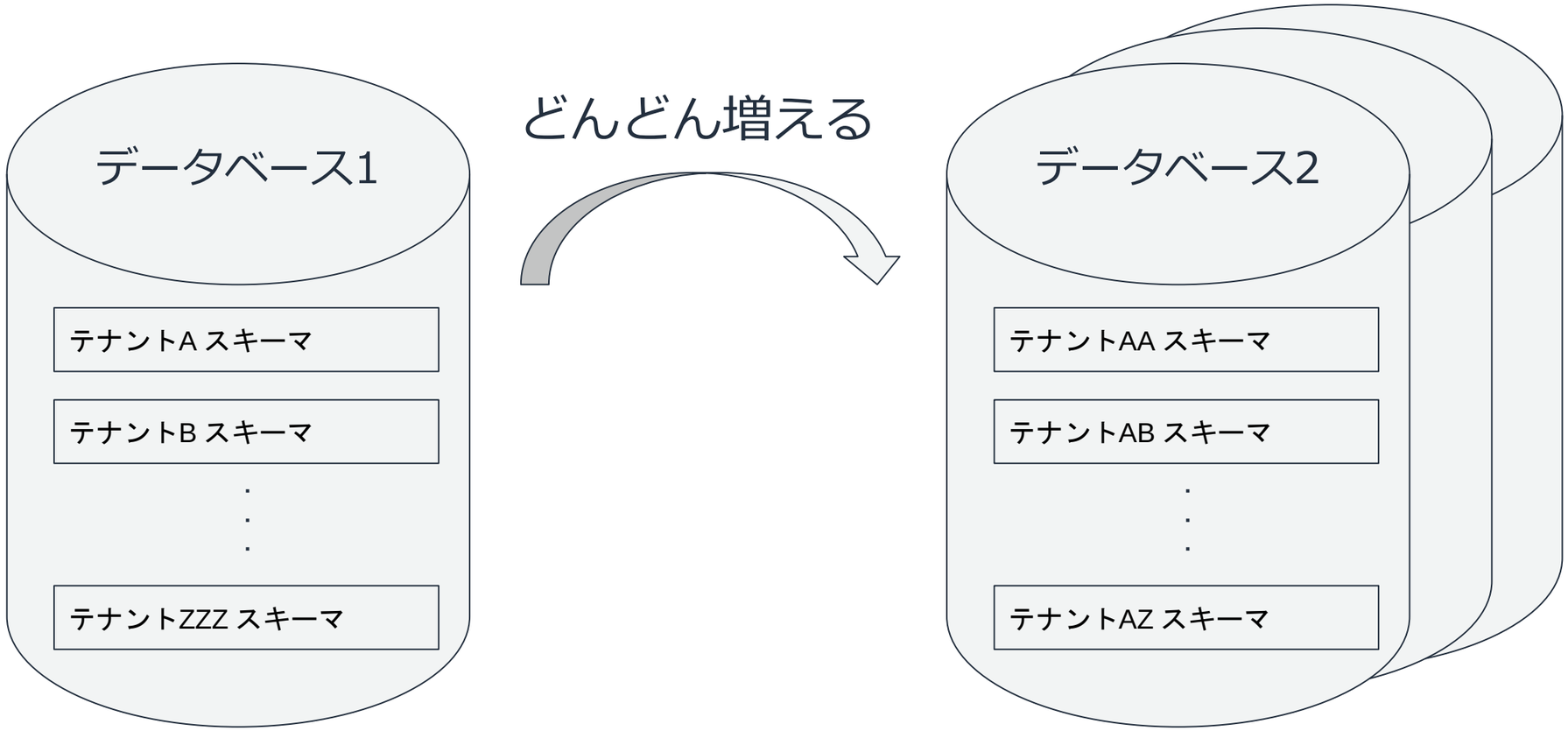


単一テーブルの場合

In Partnership with intel

経年によりDBインスタンスが増加

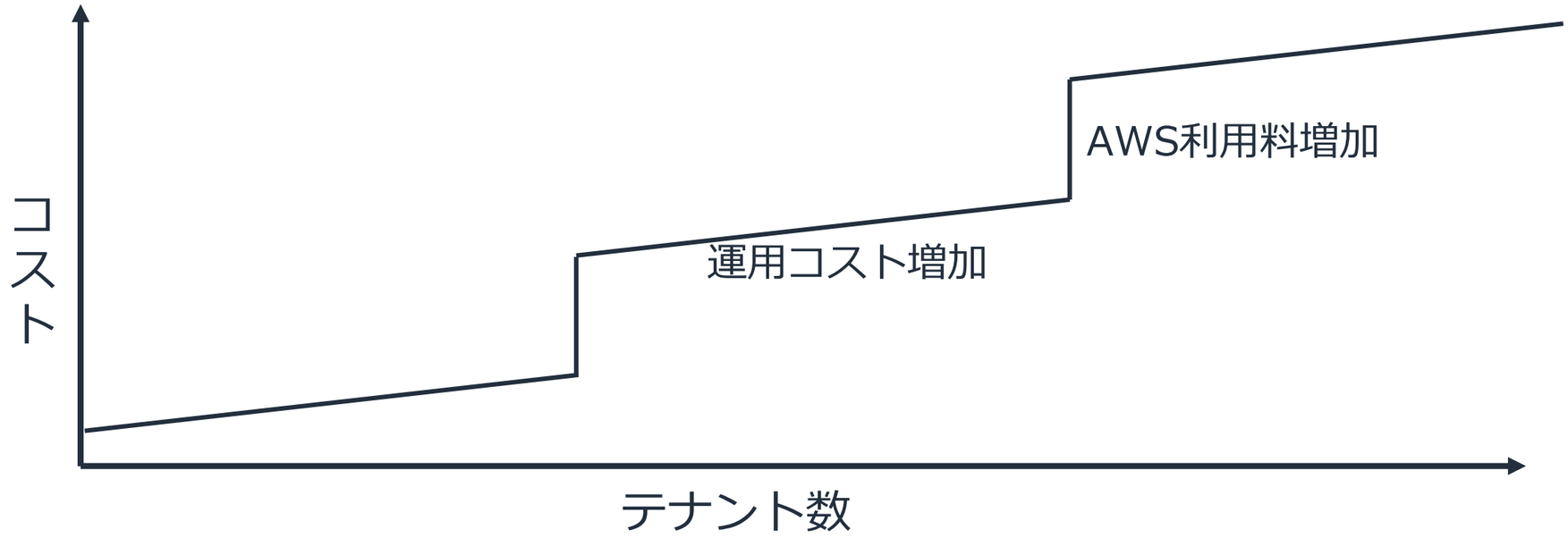
テナント単位でスキーマを作る設計のため、スキーマが増加
スキーマ増加をDBインスタンスを増やして対応
統計情報の取得やバキューム処理コストが増大



In Partnership with intel

運用コストがかかる

テナントの増加がスキーマ数が増加につながる
インデックスのメンテナンスや、スキーマ増によりデータベース全体の処理コストが上がり
バッチ処理の遅延などが発生。これらを対応するための**運用コスト**
また上記対応のためインスタンス数を増やすための**AWS利用料**に直接影響する



In Partnership with intel

aws DEV DAY
ONLINE JAPAN

データベース構成変更の検討

In Partnership with intel

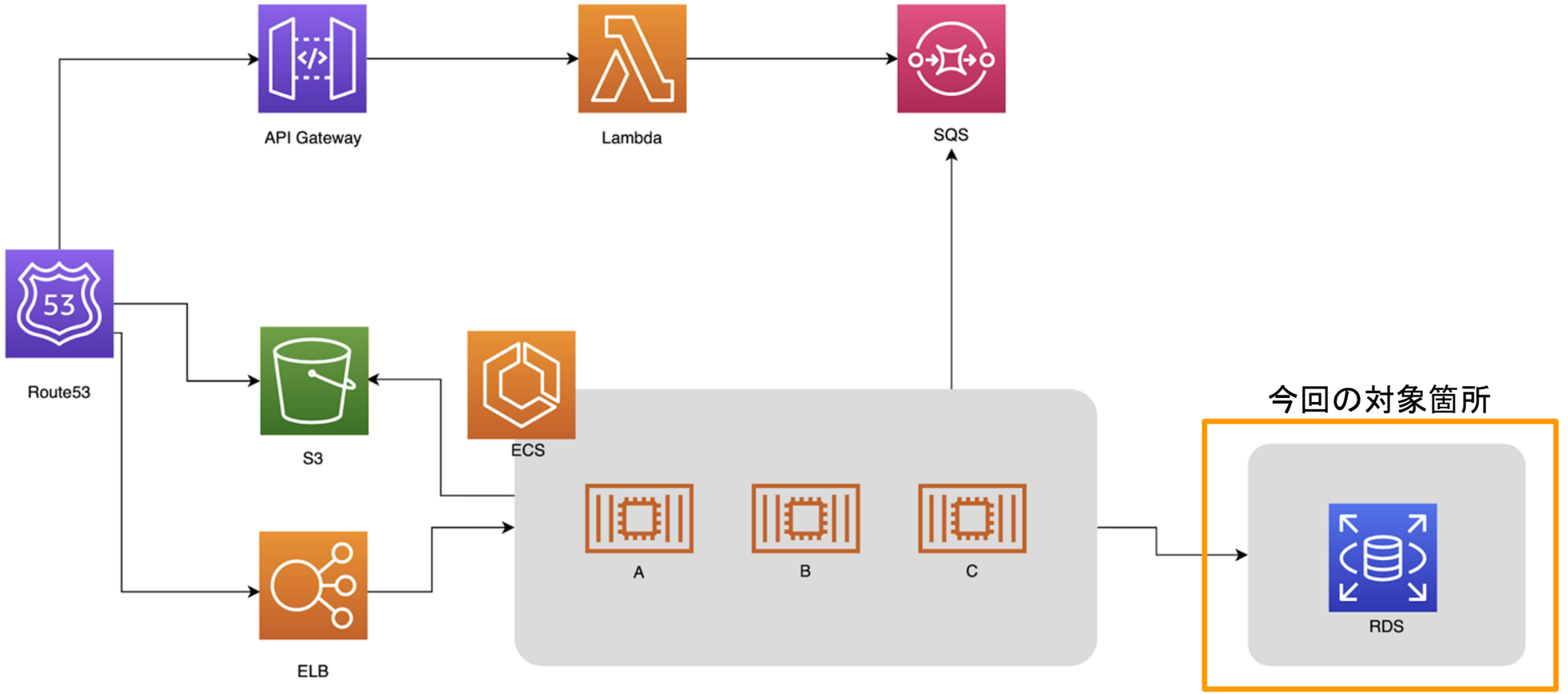
構成変更の検討

設計方針

- テナント毎のスキーマに分かれていたテーブルを、単一のテーブルに集約（以下、スキーマ集約）「テナントID」カラムを追加して、各テナントのレコードを識別する。
- スキーマ集約によって、リソースの利用効率（ディスク空間、I/O量）を向上させる。
- また、データベースオブジェクトの数を減らすことによって、管理コストを低減させる。

構成変更の検討

変更後の構成図



In Partnership with intel

構成変更の検討

スキーマ集約によるマルチテナント実装の考慮点

- セキュリティ担保（分離性・アクセス制御）
- 性能劣化の防止（性能の担保）
- アプリケーション互換性（テーブル、クエリ）

構成変更の検討

セキュリティ担保（分離性・アクセス制御）

変更前はデータオブジェクト単位（スキーマ）でセキュリティ担保していたが、変更後はレコード単位となる。単一テーブルに集約される事となるが、PostgreSQLの「行レベルセキュリティ（Row Level Security）」の機能を利用することで分離性、アクセス制御を実現する。

```
CREATE POLICY sample_table_policy_view ON sample_table
  FOR ALL
  USING (tenant_id = current_tenant_id())
  WITH CHECK (tenant_id = current_tenant_id());
ALTER TABLE sample_table ENABLE ROW LEVEL SECURITY;
```

USINGでデータ参照のできる範囲、WITH CHECKでデータ参照・データ更新の範囲を抑止

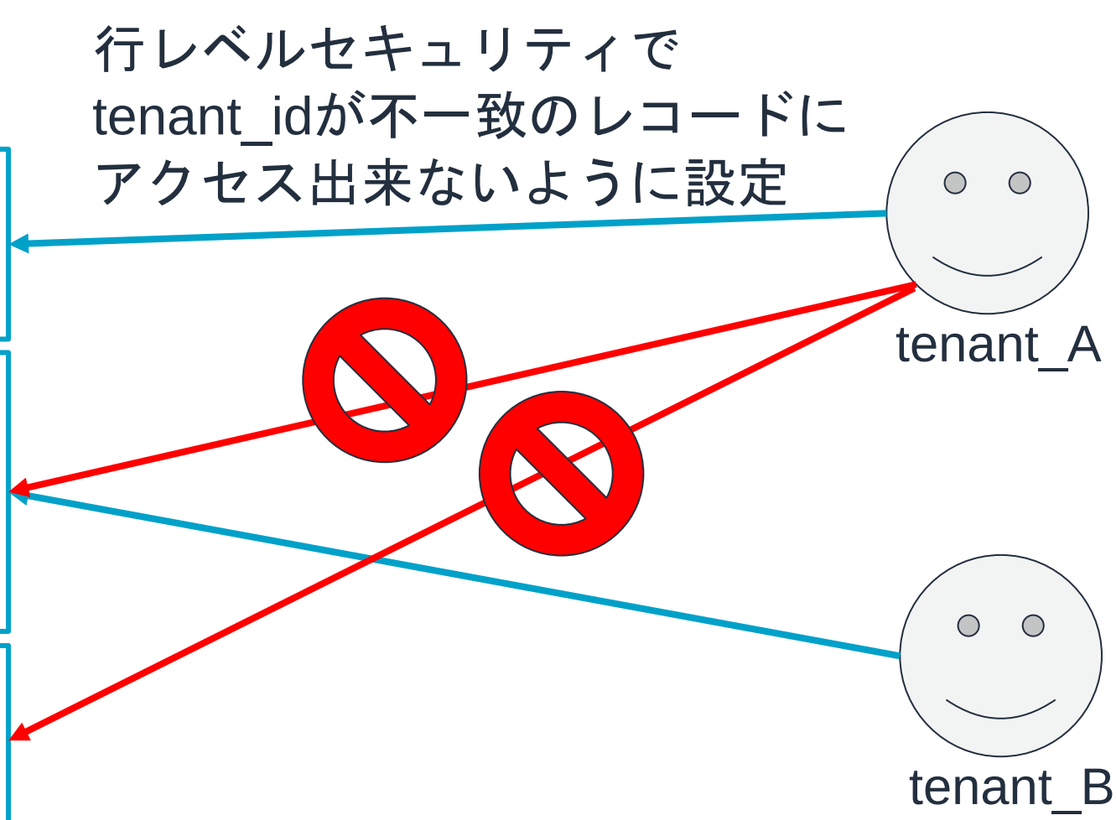
※sample_tableには、tenant_id(テナントID)を設定（tenant_idはDBのroleと一致）
※current_tenant_id()は、コンテキストのユーザ名（CURRENT_USER）からtenant_idを取得

構成変更の検討

セキュリティ担保（分離性・アクセス制御）

テーブルA

record_id	tenant_id	XXXX
1	**tenant_A**	xxxx
2	**tenant_A**	xxxx
3	**tenant_A**	xxxx
4	**tenant_B**	xxxx
5	**tenant_B**	xxxx
6	**tenant_B**	xxxx
7	**tenant_B**	xxxx
8	**tenant_C**	xxxx
9	**tenant_C**	xxxx
10	**tenant_C**	xxxx



In Partnership with intel

構成変更の検討

性能劣化の防止（性能の担保）

スキーマ集約により変更前よりもテーブルサイズが大きくなる。インデックスが利用できないクエリに対しては、処理時間の増大の可能性がある。スキーマ集約により極端にデータ量が増えるテーブル、且、シーケンシャルスキャンが発生する可能性のあるテーブルはパーティション化する。

パーティション化によりスキャン対象となるレコードを絞ることで性能劣化を防止する

構成変更の検討

```
CREATE TABLE parent_table (
 id bigserial,
 tenant_id integer DEFAULT current_tenant_id() NOT NULL,
 partition_id SMALLINT NOT NULL DEFAULT partition_id(current_tenant_id())
) PARTITION BY LIST (partition_id); ---①親テーブル。PARTITION BY LISTで分割キーを指定

CREATE POLICY partition_parent_table_view ON parent_table
  FOR ALL
  USING (tenant_id = current_tenant_id() AND
        partition_id = partition_id(current_tenant_id()))    ---②tenantIDから割り出したpartition_idを設定
  WITH CHECK (tenant_id = current_tenant_id() AND
        partition_id = partition_id(current_tenant_id()));    ---②tenantIDから割り出したpartition_idを設定
ALTER TABLE parent_table ENABLE ROW LEVEL SECURITY;    ---③親テーブル（継承元）に対してRLSを設定

CREATE TABLE partition_0 PARTITION OF parent_table FOR VALUES IN (0);
---④指定したpartition_idの値が格納される子テーブルを定義。親テーブルと同様のPOLICYを設定
```

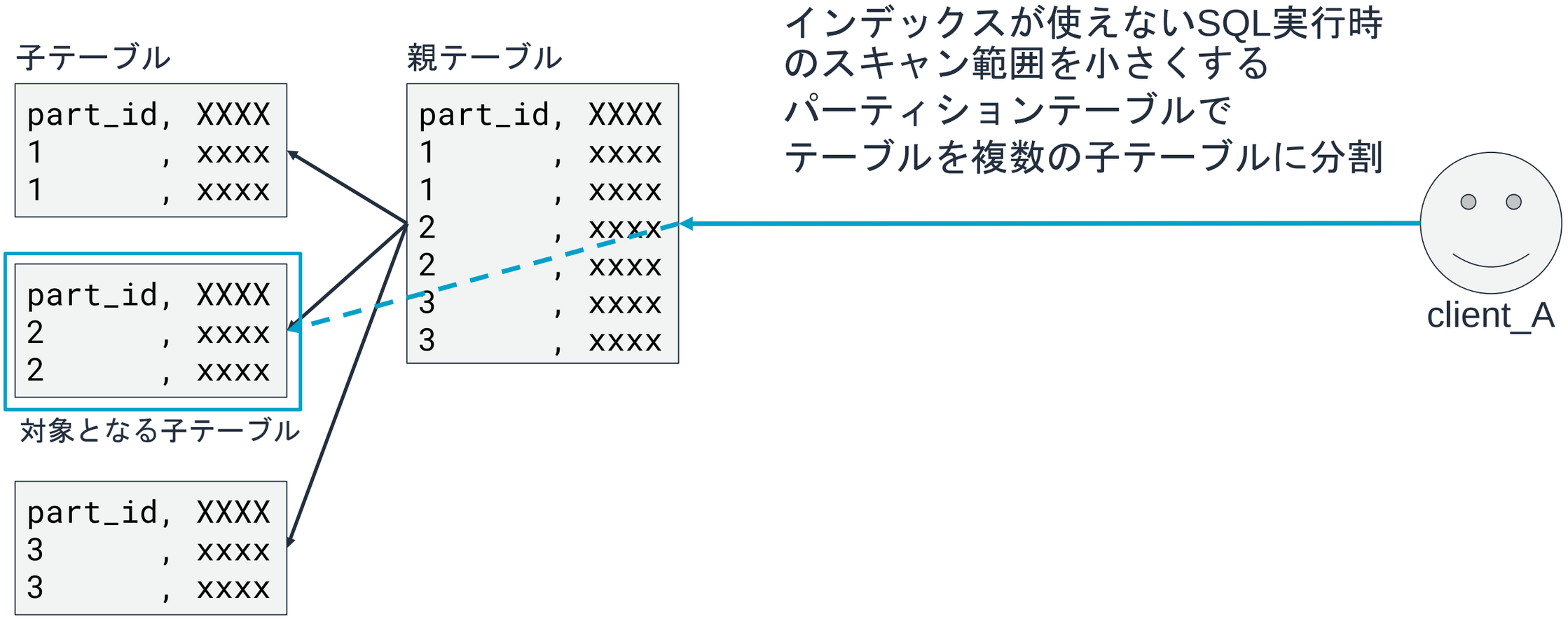
構成変更の検討

パーティション化されたテーブルへのINSERTトリガーの実装サンプル

```
CREATE OR REPLACE FUNCTION utility.partition_insert_trigger() RETURNS TRIGGER AS
$$
DECLARE
  partition_name text;
  partition_id SMALLINT;
  schema_name NAME;
  table_name NAME;
BEGIN
  schema_name = TG_ARGV[0];    ---①
  table_name = TG_ARGV[1];        ---①
  IF NEW.partition IS NULL THEN
    NEW.partition = utility.get_partition_id(utility.current_tenant_id());
  END IF;
  partition_name := schema_name || '.' || table_name || '_' || NEW.partition_id::text;    ・・・②
  EXECUTE 'INSERT INTO ' || partition_name || ' VALUES(($1).*)' USING new;    ・・・③
  RETURN NULL;
END
$$
LANGUAGE plpgsql;
```

構成変更の検討

性能劣化の防止（性能の担保）



In Partnership with intel

構成変更の検討

アプリケーション互換性（テーブル、クエリ）

変更後の設計ではスキーマ集約のためにテナントID、パーティション化のためにパーティションIDが追加される
何もDBの機能によるところなので、アプリケーションには機能的な互換性の問題は発生しない。

但し、インデックス設計次第では、期待する性能を得られない可能性がある。

構成変更の検討

アプリケーション互換性（テーブル、クエリ）

current_tenant_id()で、クエリ実行しているtenantIDを取得

行レベルセキュリティのポリシー
USING (tenant_id = current_tenant_id())で可視化範囲を特定

In Partnership with intel

構成変更の検討

アプリケーション互換性（テーブル、クエリ）

current_tenant_id()で、クエリ実行しているtenantIDを取得

パーティションIDは、tenantIDから関数で設定

トリガーによってデータを格納する子テーブルを特定し登録

In Partnership with intel

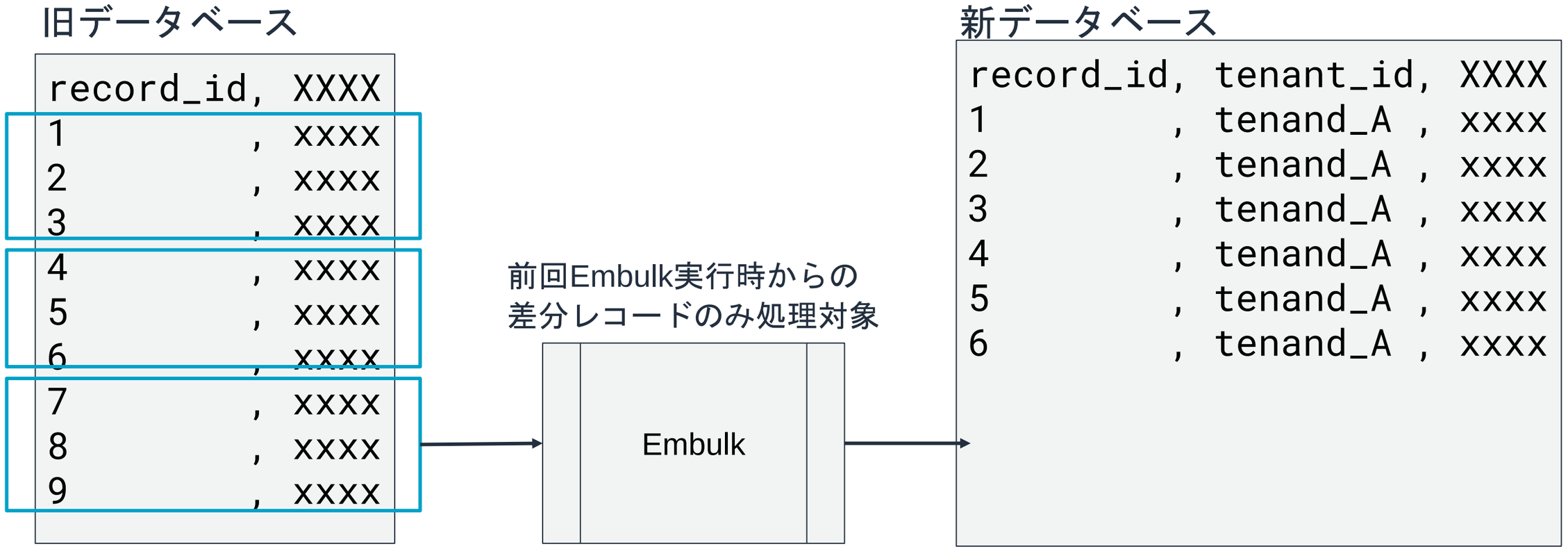
aws DEV DAY
ONLINE JAPAN

データベース移行

In Partnership with intel

移行作業

移行時の影響を小さくするためデータベースの差分移行を選択
Embulkを使って差分移行を行った
Embulkでの移行時にテーブル構成の差異を吸収



In Partnership with intel

移行作業

移行にはサービス停止を伴うため、短時間で実施する必要があった

直前にEmbulkでデータを同期し、
移行時はEmbulkのmergeモードを利用して、前回同期後に登録・更新されたデータのみを対象とした。これにより移行時間を短縮

既存のデータベースはテナント単位でスキーマが作成されているため
全スキーマからデータを集め、1つのテーブルにデータを移行
併せてEmbulkでテーブルの差異を取り込みつつ移行

移行後の問題点

パーティション化したテーブルで一部SQLが遅くなったが、パーティションIDを含めたインデックスで対応。インデックスの効率を上げるため、部分インデックスに変更

SELECT tenantID, type FROM sample WHERE type = 0 GROUP BY tenantID, type

ID	tenantID	type	insert_date	delete_flag
1	0000001	0	2020/05/01 20:10:00	0
2	0000002	0	2020/05/01 20:10:00	0
3	0000003	1	2020/05/01 20:10:00	0

今までのインデックス
create index on sample (tenantID、type);

ID	tenantID	type	insert_date	delete_flag
1	0000001	0	2020/05/01 20:10:00	0
2	0000002	0	2020/05/01 20:10:00	0
3	0000003	1	2020/05/01 20:10:00	0

変更した部分インデックス
create index on sample (tenantID、type) where **type = 0**

移行を終えて

・常にDBが高負荷

=> I/O改善により達成

・経年によりDBインスタンスが増加

=> スキーマ集約しインスタンス１台化により達成

もともとスキーマ単位で分離されていたので、
今後ユーザが増えてもデータベースインスタンスを追加する、
コンピュート層を追加することで対応可能

・運用コストが増加

=> インスタンス１台化により達成

まとめ

・PHP5.6を利用しているため、EOLが迫っている。そのためPHP7.xへマイグレーションをすることが必要

=> PHP7.x＋Golang(バッチ系)にした

・事業の成長に伴い顧客数が増加した事でデータ量も多くなった。それに従いデータベースのパフォーマンスに問題が出てきたため、データベース処理の性能改善が必要

=> PostgreSQLのRLSを利用し、マルチテナントデータの分離方法をスキーマ単位から行単位に変更し、性能担保とコストの最適化が行えた

aws DEV DAY
ONLINE JAPAN

Thank you!

株式会社アンチパターン
塚本 岳史

In Partnership with intel