

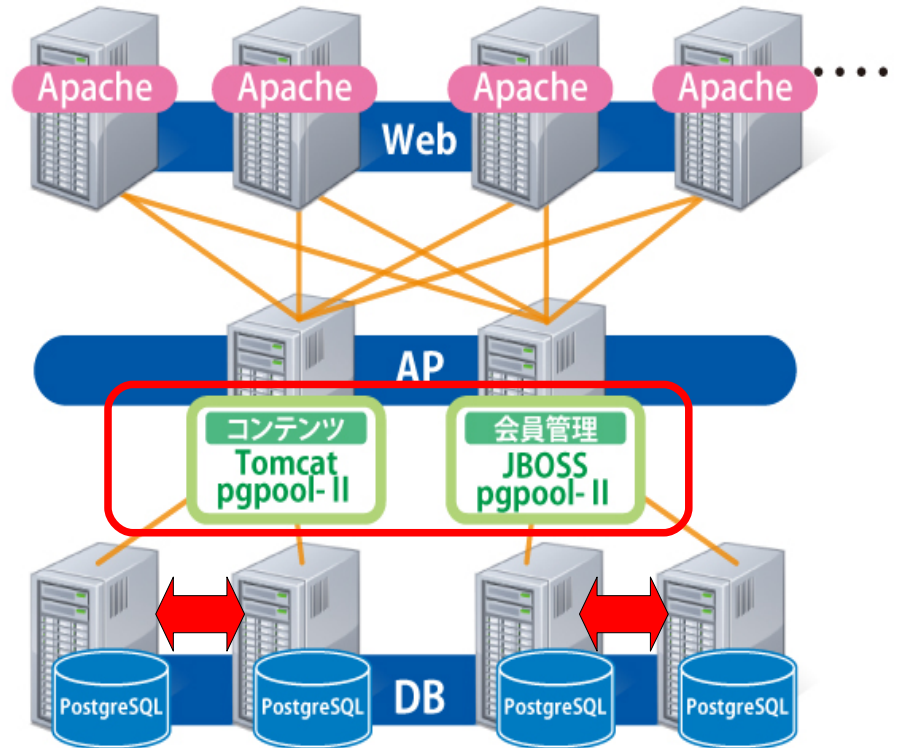
pgpool-IIによるオンメモリ クエリキャッシュの実装

SRA OSS, Inc. 日本支社

pgpool-IIとは

- PostgreSQL専用のミドルウェア
- OSSプロジェクト (BSDライセンス)
- proxyのように、アプリケーションとPostgreSQLの間に入って様々な機能を提供
 - コネクションプーリング
 - 負荷分散
 - 自動フェイルオーバー
 - レプリケーション
 - クエリキャッシュ

導入事例:JTBBが発行する「JTBB旅カード」 のWebサイト

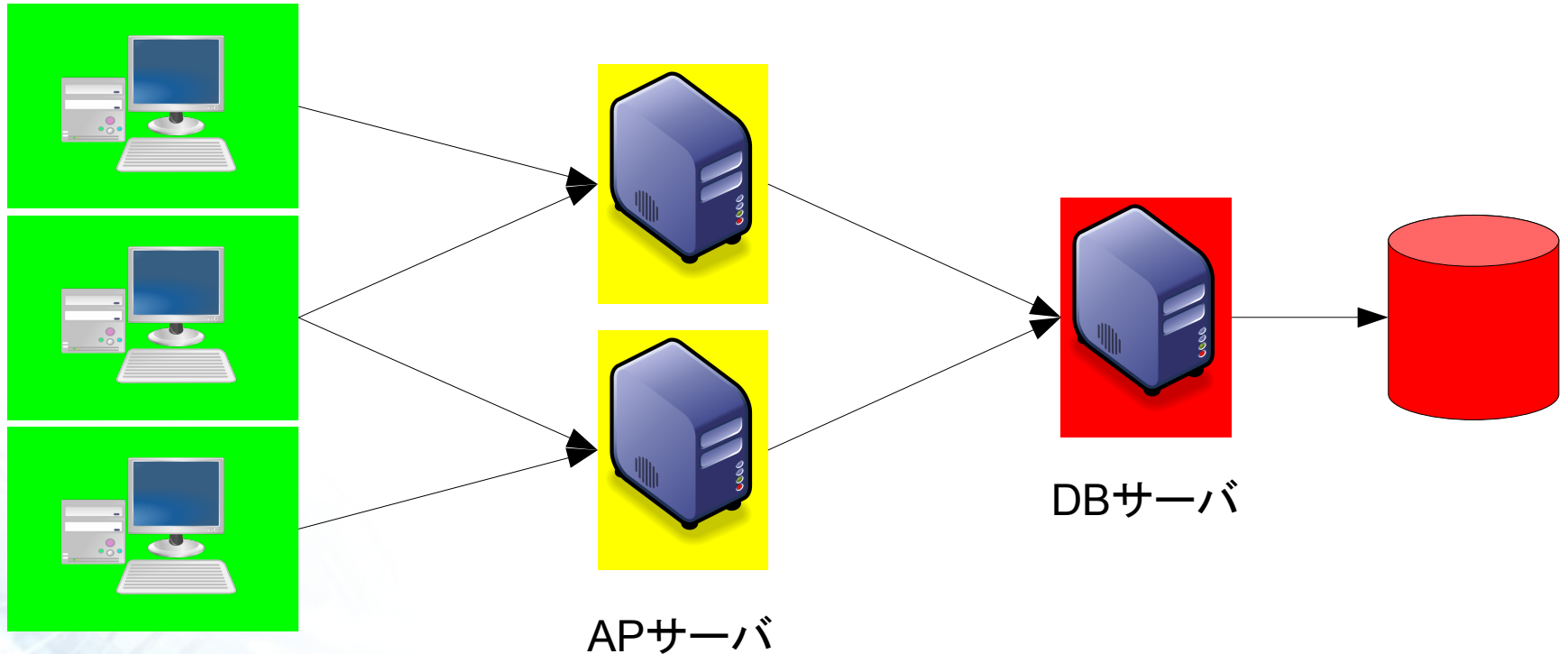


- カード会員が利用するポータルサイトと、ポイント管理を行うバックヤード機能から構成
- 従来クレジットカード会社で行っていたポイント管理をJTBBに移管することで、顧客のニーズを反映した情報提供を行うCRMの役割を果たす

今作っているもの

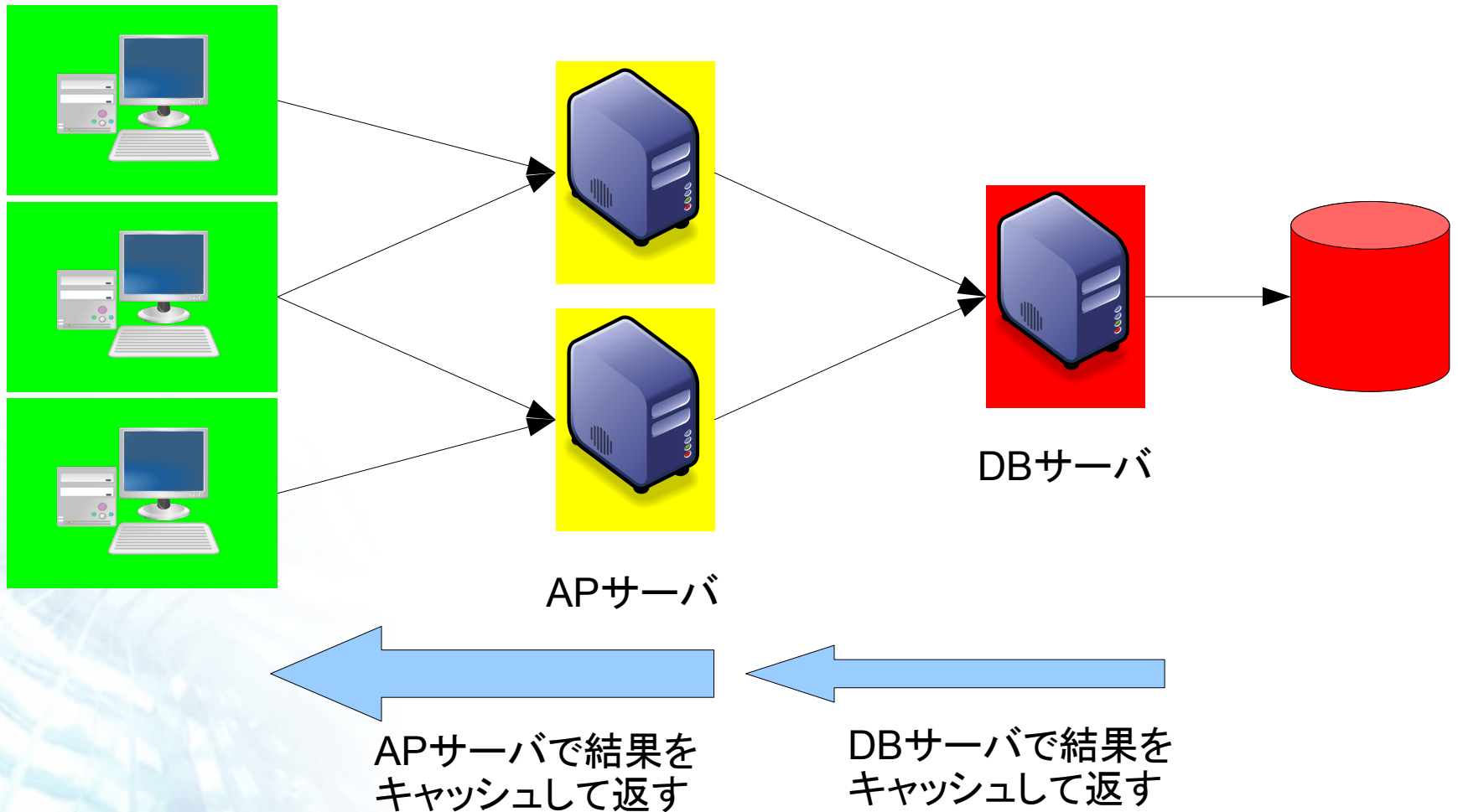
- オンメモリクエリキャッシュ
- 複数pgpool-IIの連携機能
 - 外部プログラムに依存しないpgpool-II組み込みのHA機能
 - オンラインリカバリなどの制御

Web環境における レイヤー別負荷の違い



後ろのレイヤーほど負荷が高く、ボトルネックになりやすい

キャッシュを活用して負荷を軽減



キャッシュの実装例(1)

- アプリケーションサーバ／httpdサーバレベルでのキャッシュ実装
 - APC(Alternative PHP cache)
 - PHPが動いているサーバ上の共有メモリにキャッシュを作る
 - Apache2のコンテンツキャッシュ
 - mod_cache, mod_file_cache
 - RailsやHibernateなどのORMマッパーでのキャッシュ
 - SquidやVarnishなどのリバースプロキシを使う
 - memcachedなどのKVS(Key Value Store)を使ったキャッシュ
- APIやフレームワーク固有の方法でのキャッシュなので、それぞれ使い方や特性が違う
 - 使用するフレームワークが変わると実装し直し

キャッシュの実装例(2)

- DBMSでのキャッシュ
 - MySQLの実装が有名。DB2にもある？
 - DBMSへのアクセスがなくなるわけではないので、DBMSがボトルネックになることもある
- MySQLのクエリキャッシュ
 - オンメモリキャッシュ
 - クエリ文字列が一致したらキャッシュヒット
 - Prepared queryには対応していない
 - テーブルが更新されたら該当キャッシュは全クリアされる
 - MySQLを再起動したらキャッシュの内容はなくなる

APサーバのキャッシュと DBサーバのキャッシュの比較

	APサーバでキャッシュ	DBサーバでキャッシュ
キャッシュの効果	◎	○
DBサーバへの負荷	○	△
アプリケーション透過性	×	○
スケーラビリティ	○	×

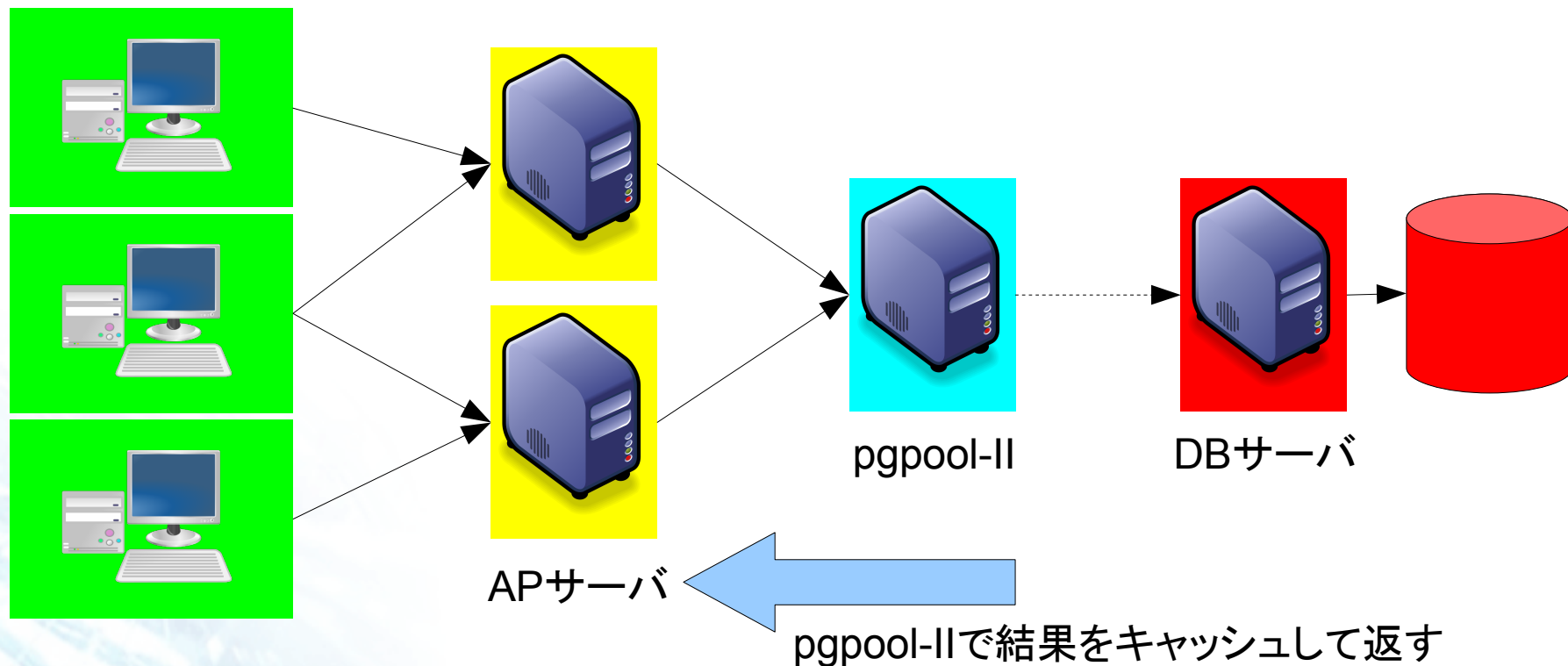
実はpgpool-IIにもすでに クエリキャッシュがあります

- しかしいろいろ問題が...
 - キャッシュストレージがDBなので遅い
 - DBを更新しても自動ではキャッシュが更新されない
 - 拡張問い合わせに対応していない
- そこで、新しく実装しなおすことにしました
 - Google Summer of Codeとしてプロトタイプを実装
 - 現在バグ取り、未実装部分の実装を行っています

pgpool-IIオンメモリ クエリキャッシュの主な機能

- キャッシュストレージとして、共有メモリか memcachedを選択できる
- セッション、ユーザをまたがってキャッシュが再利用できる
 - memcachedを使う場合は、pgpool-II、PostgreSQLを再起動した後にキャッシュを再利用することも可能
- テーブル別にキャッシュする・しないを指定できる
- キャッシュの更新は自動的に行われる
 - 更新問い合わせが来たらキャッシュをクリア
 - 一定時間が過ぎたキャッシュを自動クリアすることも可能
- 拡張問い合わせでもキャッシュが使える(実装中)

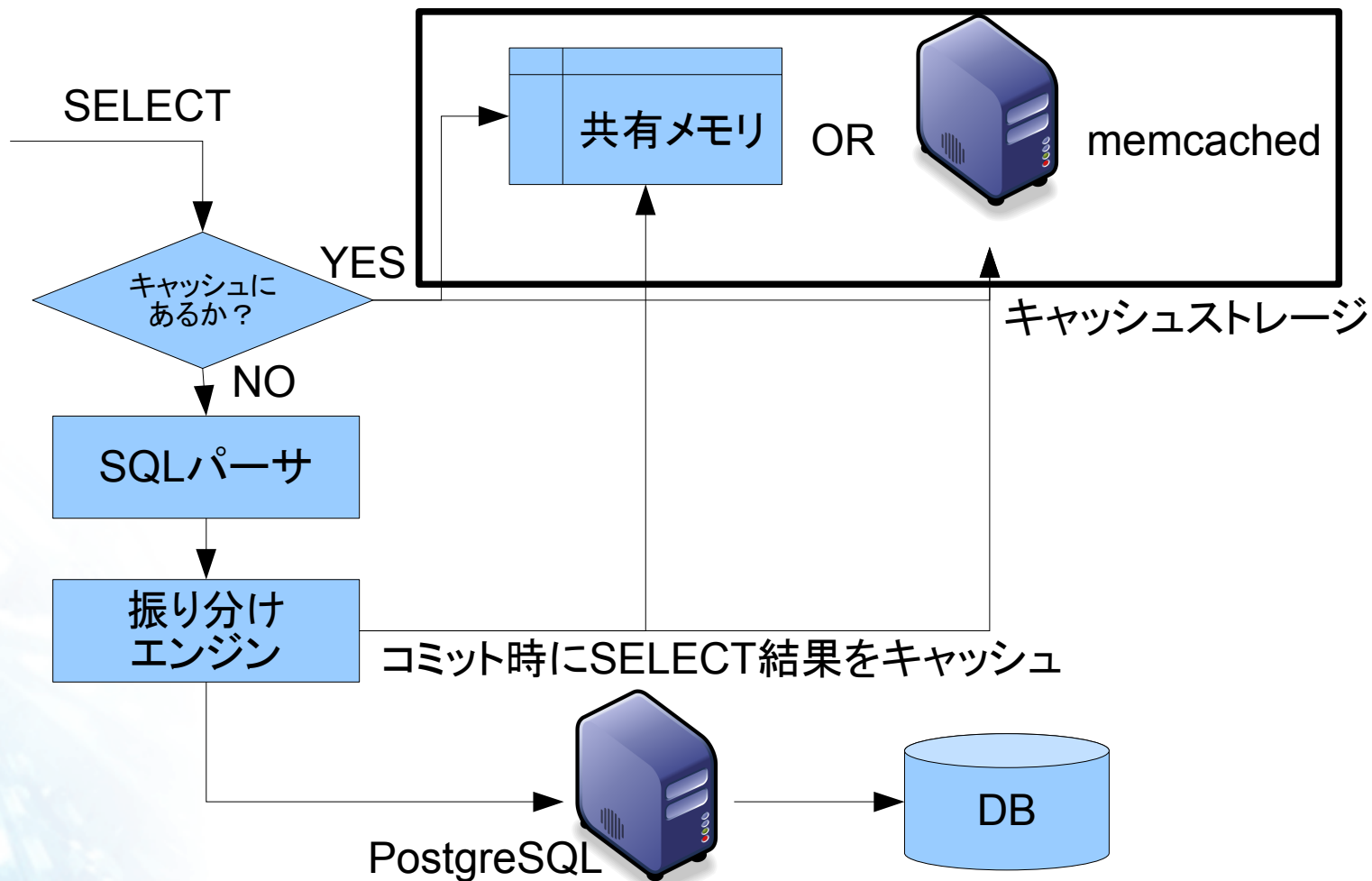
pgpool-IIオンメモリ クエリキャッシュの利用イメージ



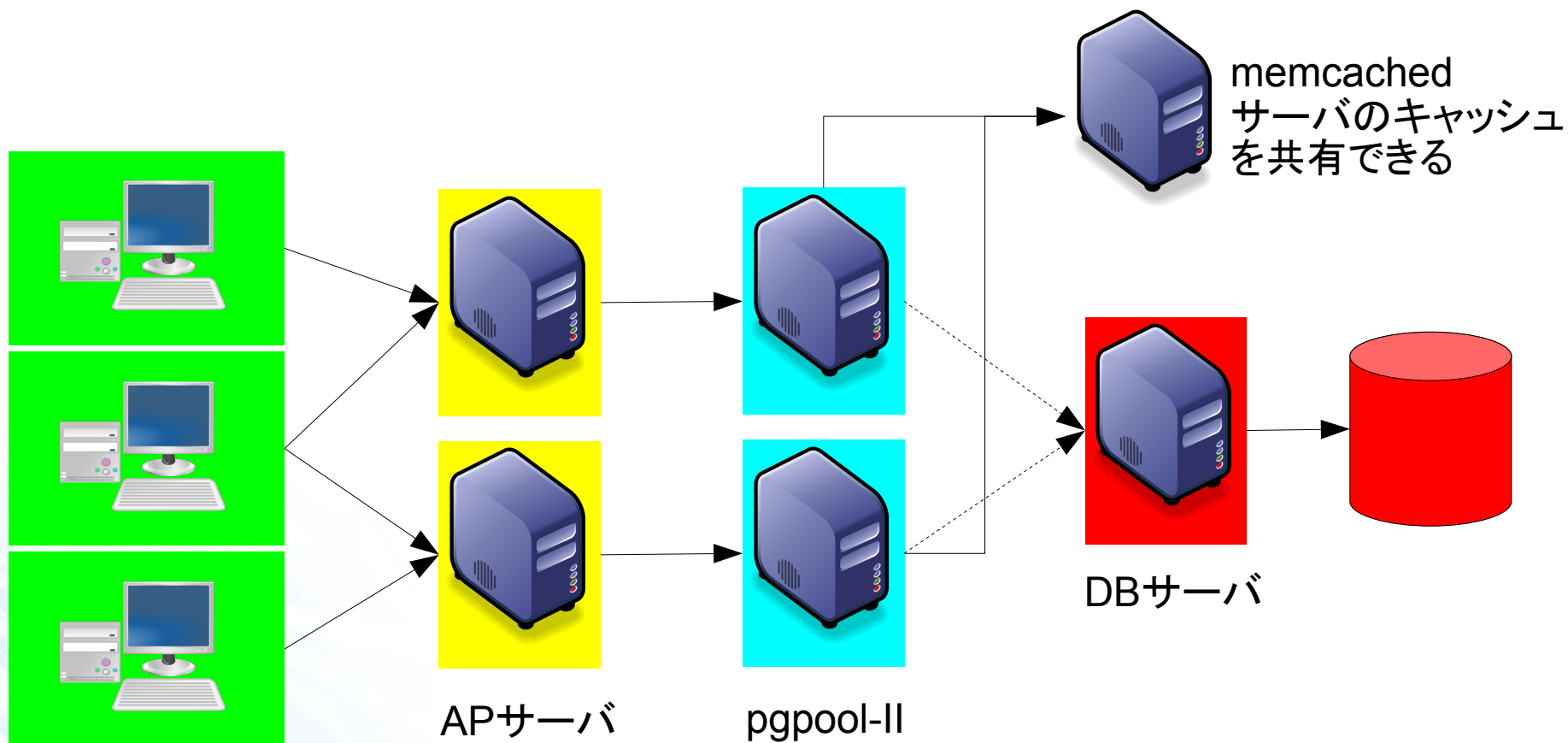
APサーバ、DBサーバ pgpool-IIのキャッシュ機能の比較

	APサーバで キャッシュ	DBサーバで キャッシュ	pgpool-IIで キャッシュ
キャッシュの効果	◎	○	○
DBサーバへの負荷	○	△	○
アプリケーション透過性	×	○	○
スケーラビリティ	○	×	○

pgpool-IIオンメモリ クエリキャッシュの仕組み



キャッシュサーバのスケールアウト



工夫した点 キャッシュの「ヒット」の定義

- クエリ文字列+データベース名+テーブル名+ユーザ名をMD5ハッシュしたものをキーにして一致を判断し、ハッシュ値が一致したらキャッシュがヒットしたものと見なす
- 直接SQL文を比較しないのは、長大なSQL文を保存する必要性を避けるため
- ユーザ名がキーに含まれているのは、参照権限がないテーブルのデータをキャッシュを経由して他のユーザが参照できないようにするため
 - 一方で参照しても良いキャッシュを参照できないデメリットがある
- MD5ハッシュで管理してキーが衝突する心配はないのか？
 - 管理するキーの数を 2^{32} (約42億)とする。1マイクロ秒(1/1,000,000秒)に1回キャッシュを登録しても、衝突するまでの平均時間は太陽の寿命より長い(たぶん)
 - $2^{(128-32)}/(1000*1000*60*60*24*365)$
= 2,512,308,552,583,217年
= 2,512兆年 = 2.5京年

キャッシュ対象とならないSELECT

- Immutableでない関数への呼び出しを含むSELECT
- `SELECT CURRENT_TIMESTAMP;`
- `SELECT INTO`, `SELECT FOR UPDATE`, `SELECT FOR SHARE`
- SELECTの結果データが大きいもの(> memqcache_maxcache)
 - 巨大なデータでキャッシュが使い尽くされてしまうのを防ぐため
- 成功しなかったSELECT
- ロールバックされたトランザクション内のSELECT結果
 - `BEGIN;`
`INSERT INTO t1 VALUES(1);`
`SELECT * FROM t1;`
`ROLLBACK;`
 - もしSELECT * FROM t1の結果をキャッシュしてしまうと、次のSELECT * FROM t1で存在しないはずの「1」が返ってしまう
- 一時テーブルを含んでいるSELECT
 - セッションが終了するとテーブルが消えてしまうため
- Unloggedテーブルを含んでいるSELECT
 - PostgreSQLクラッシュ後の再起動でデータが消えてしまうため

キャッシュ更新／無効化ポリシー (Invalidation)

- キャッシュ対象のテーブルが一部でも更新されたら、そのテーブルを参照しているキャッシュをすべて自動的に削除する
 - 更新クエリ: INSERT/UPDATE/DELETE/TRUNCATE/COPY FROM
 - そのために、キャッシュを登録する際に参照しているテーブルのOIDを調べ、ファイルに登録しておく
 - 更新クエリが実行されたらそのファイルを調べて、関連するキャッシュを削除する
- データベースやテーブルが削除された場合も同様
- テーブル構造が変わるようなコマンド(ALTER TABLE)が実行された場合も同様
- 更新が多いDBには向かない
- 有効時間を過ぎたキャッシュは無効になる(無効にしない設定も可能)
- 手動でキャッシュを削除可能(計画中)
- キャッシュストレージがmemcachedの場合はちょっと悩ましい
 - pgpool-IIやPostgreSQLが再起動しても、memcachedが動いていれば前回のキャッシュを有効に使える？

制限事項

- VIEWもキャッシュされるが、VIEWが参照しているテーブルが更新されてもキャッシュは有効
- スキーマが異なっても、DB名、テーブル名が同じならば同じテーブルと見なされる
- トリガによって暗黙的に更新されるテーブルが認識できない
- 外部キーが指定されていて、ON DELETE CASCADEなどで他のテーブルの行が暗黙的に更新されたことが認識できない
- DROP TABLE CASCADEで暗黙的に削除されたテーブルが認識できない
- TRUNCATE TABLE CASCADEで暗黙的に内容が削除されたテーブルが認識できない

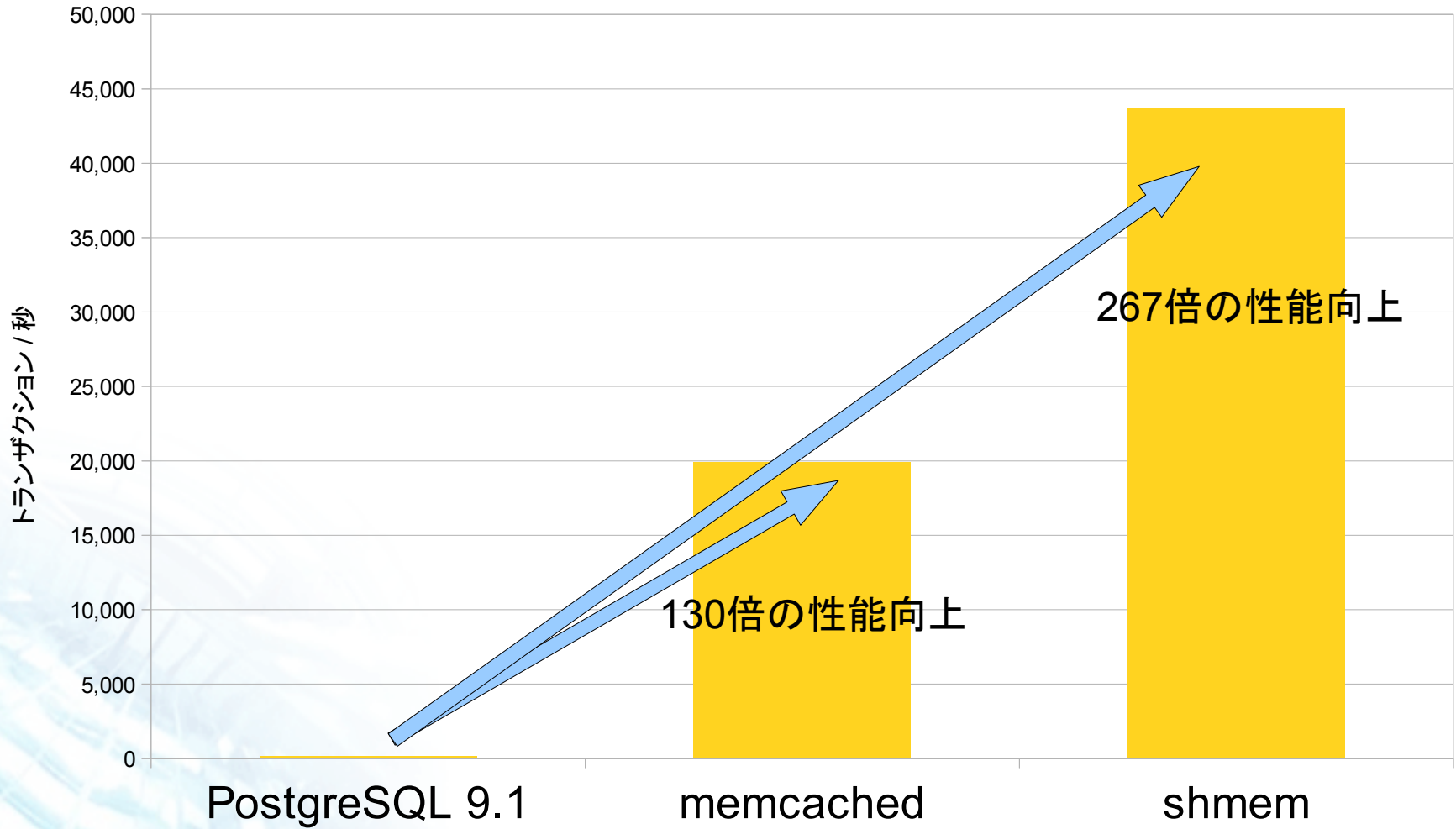
オンメモリクエリキャッシュの設定項目

- `memory_cache_enabled = false`
 - メモリキャッシュの有効／無効
- `memqcache_method = 'shmем'`
 - キャッシュストレージの選択。shmем(共有メモリ) or memcachedが選択可能
- memcachedを選択した場合の設定項目
 - `memqcache_memcached_host = 'localhost'`
 - `memqcache_memcached_port=11211`
- `memqcache_total_size=134217728`
 - トータルキャッシュサイズ
- `memqcache_expire=60`
 - キャッシュの有効時間
- `memqcache_maxcache=1024`
 - 格納できる最大のSELECT結果サイズ
- `memqcache_cache_block_size=1048576`
 - キャッシュブロックのサイズ(共有メモリのときのみ)
- `memqcache_oiddir = '/var/log/pgpool/oiddir'`
 - テーブルOIDを格納する領域

ベンチマーク！

- SELECT count(*)という、PostgreSQLでは全スキャンになる遅いクエリを約100%のキャッシュヒット率でアクセスしたケース(ベストケース)
 - SSDを使い、しかもテーブルがキャッシュに乗った状態なので、実環境ではもっと差が開く可能性がある
- ハードウェア
 - ノートPC(dual core i5-2540M CPU @ 2.60GHz、Hyper threading有効、メモリ8GB、SSD(Intel 320 MLC)
- ソフトウェア
 - pgpool-II, PostgreSQL, memcachedを同じマシンで動かしている
 - PostgreSQL 9.1
 - shared_buffers = 32MB
 - pgbench
 - SQL: SELECT count(*) FROM pgbench_accounts;
 - 1000回実行
 - 10万件、テーブルサイズ13MB
 - I/Oネックにならない

ベンチマーク結果



まとめ

- 各レイヤにおけるキャッシュソリューションを比較
 - アプリケーションサーバでのキャッシュは効果が高いが、アプリケーションの改造が必要になることが多い
 - DBMSでのキャッシュは、アプリケーションの改造が要らないメリットがある。また、DBMSの負担を軽減する。更新の多い用途には向かない。スケールアウトができない
 - pgpool-IIのオンメモリキャッシュは、アプリケーションの改造が不要、DBMSの負担を軽減し、さらにスケールアウトすることができる。ただし、更新の多い用途には向かない
- pgpool-IIのようなミドルウェアにおけるキャッシュは、用途によって大きなメリットがある

今後の開発予定

- 拡張問い合わせへの対応
- テーブル別に自動キャッシュバリデーションをする・しないを指定できる
- 2012年5月位にリリース予定

参考URL

- pgpool-IIのホスティングサイト
 - <http://www.pgpool.net>
 - 旧サイト pgfoundry から引っ越したので注意！
 - tar ballのダウンロードができます
 - ソースリポジトリは git.postgresql.org で
 - <http://www.pgpool.net> からリンクされています
 - 日本語と英語のメーリングリストがあります
- Twitter
 - @pgpool2