EISUKE KOZU
Twitter: @CSTYLES_JP

Gangsta

PostgreSQLのしくみ分科会+アプリケーション分科会

ECサイト構築でPostgreSQLを採用するメリット

Gangsta

■本日お話すること。

Webアプリケーション実装プログラマとして、ＥＣサイトのカスタマイズや運用をしている立場から、初級〜中級Webプログラマーの方に役立つTipsや失敗事例の共有など。

9.0のストリーミングレプリケーションも試したので、そこらへんについても話します！

■自己紹介など。

元々はファーム系Cプログラマー。
その後、プログラムを書くことから逃げ出し、レストランでピザを焼いてみたりする。

さらに、日本から逃げ出して、ニュージーランドのレンタカー屋でバイトをしてグダグダと過すが、なぜか結婚相手(日本人)をみつけて帰国。

帰国後、某小売業のＥＣシステムなどを触るはめに
（PHP+Postgresql。Webサービス開発たのしい！）

最近はWebマーケティングチーム"Gangsta"として
Webサービス開発なども請け負ってます。

Gangsta

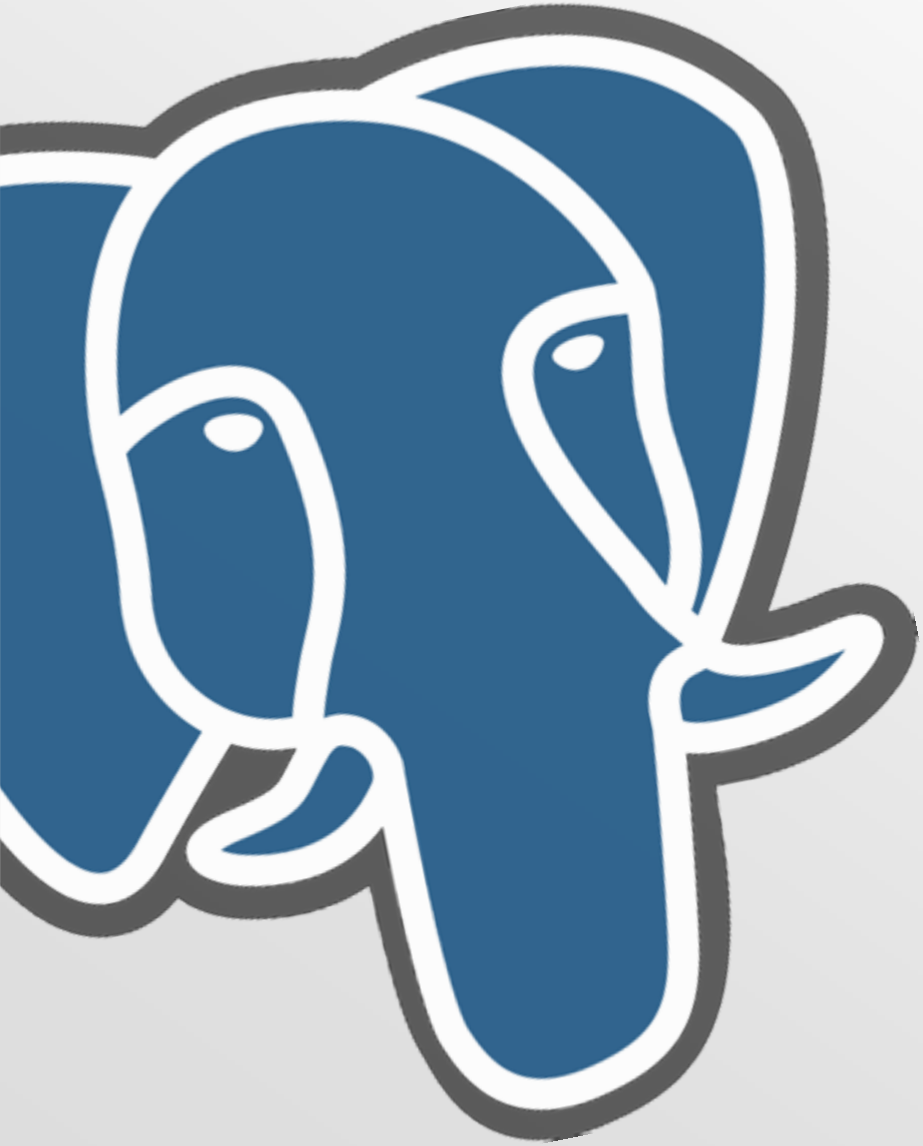
■免責事項。

いつも上司に
「日本語が不自由すぎる」
と言われてるので、

なに言ってるかわからない
かもしれません。

先に謝ります。
ごめんなさい（＾＾；

Gangsta



本題に
はいります。

Gangsta

ＥＣサイトの規模と
適したシステムについて。

Gangsta

ＥＣサイト

今日、話すのは
ここらへんの領域

モール型

楽天 ICHIBA

カラメル

YAHOO! JAPAN ショッピング

amazon.co.jp

ASP型

カラーミーショップ

e-shops

SHOP-Maker

パッケージ型

Commerce 21

ecbeing7

フルスクラッチ

OSS型

EC CUBE 日本発のECオープンソース

Soy CMS

osCommerce Open Source E-Commerce Solutions

zencart the art of e-commerce

■ＥＣやってると起きること。

テレビやメジャー誌などに取り上げられるとアクセスが当然多くなるのですが、インフラ冗長化されていないと検索などがまったく動かなくなります。。。

テレビなどと違い、ヤフートピックスへの掲載されると、事前告知もないので、呆然と立ち尽くすことになります ε =(～Д～;)

ツイッターとかフラッシュマーケティングとかで予想を超えたアクセスが生まれたときも同様です。。。

■ＥＣやってると起きること。

サーバー屋さんに相談しても、大抵はサーバー(ハード)をグレードアップするのが良いという、誰でも思いつく答えと高額の見積書しか返ってきませんが、DBはチューニングにより同じサーバーを使っていてもパフォーマンスが上がる可能性があるので、予算がない時はまずはそこから（予算があれば当然ハードから）

実際にチューニング を施したところスピードが数倍になったこともあります。

Gangsta

■チューニングの参考にしたもの。

「Let‘s postgres」目的別ガイド：チューニング編

http://lets.postgresql.jp/map/tuning/

「PostgreSQL のチューニング技法

しくみを知って賢く使う」

http://www.postgresql.jp/events/pgcon09j/j/

※リクエストしたらまたしくみ会で笠原さんが喋ってくれるかもしれませんね。期待(*´д`*)

■ＥＣやってると起きること。

ある程度の売上になってくると、人間が経理処理とか在庫管理とかやってらんなくなってくるので、在庫管理システムやＰＯＳシステムとの連携、会計システムとの連携などが求められます。

言語やＤＢの違いを乗り越えてのデータ連携は当然ですが、デイリーで数十万件のデータ交換とかさせられるので、バッチ処理が２０時間を越えたりしてDBもアップアップですわ(; ´д⊂)

流れ的にはHadoopとかなんですかね？

ＥＣサイトに求められる
機能とＤＢの関係。

■何が求められているか。

売れること

・・・当たり前ですが、深い話ですよね・・・。

Gangsta

■売れるシステムとは？

もちろんその答えは会社や状況によって違うでしょうが、データベースに関するエンジニアの解としては・・・

高速かつ、的確にヒットする検索

ＥＣサイトは、検索に対してのユーザーニーズが高い分、複雑化して遅くなってしまう傾向があり「高速かつ、的確にヒット」を両立させることは難しいのですが、Postgresqlで何が解決できるのかを説明します。

商品カテゴリ検索の高速化

■商品に複数カテゴリを持たせる。

商品データ

	名前ボックス	B	C
1	商品ID	商品名	カテゴリID
2	0000029323	豚まん	030401,050103
3	0000029324	あんまん	030401,050103
4	0000029325	肉まん	030401,050103
5	0000017554	コロッケ	030401,050205
6	0000017555	フライドチキン	030401,050205
7	0000017556	フランクフルト	030401,050205
8	0000017566	おでん	030401

カテゴリデータ

	A	B
1	カテゴリID	カテゴリ名
2	030401	店内で調理
3	050103	中華まん
4	050205	ホットスナック

データベースへ格納したときはこんなで。

出力ビュー

データの出力 | 解釈 | メッセージ | ヒストリー

	商品ID character varying(10)	商品名 character varying(200)	カテゴリID character varying(50)
1	0000017554	コロッケ	030401,050205
2	0000017555	フライドチキン	030401,050205
3	0000017556	フランクフルト	030401,050205
4	0000029323	豚まん	050103,030401
5	0000029324	あんまん	050103,030401
6	0000029325	肉まん	050103,030401
7	0000177266	おでん	030401

検索時にプログラムはどんな問合せ(SQL)を発行するかというと、、、

select 商品ID from 商品テーブル where カテゴリID='030401'

このデータならカテゴリIDに B-Treeインデックス を張れば
高速に検索できる・・・？

出力ビュー

データの出力 | 解釈 | メッセージ | ヒストリー

	QUERY PLAN text
1	Index Scan using _index on goods (cost=0.00..266.16 rows=67 width=19) (actual time=0.039..0.291 rows=198 loops=1)
2	Index Cond: (( )::text = '030401'::text)
3	Total runtime: 0.321 ms

しかし、’030401’をヒットさせたいのに、この検索だと「完全一致」のため複数カテゴリが設定された商品にヒットしません・・・。

	名前ボックス	B	C
1	商品ID	商品名	カテゴリID
2	0000029323	豚まん	030401,050103
3	0000029324	あんまん	030401,050103
4	0000029325	肉まん	030401,050103
5	0000017554	コロッケ	030401,050205
6	0000017555	フライドチキン	030401,050205
7	0000017556	フランクフルト	030401,050205
8	0000017566	おでん	030401

カテゴリが複数設定されているとヒットしない・・・

これを漏れなくヒットさせようとして、

select 商品ID from 商品テーブル where カテゴリID like '%030401%'

中間一致で検索してしまうと・・・

出力ビュー

データの出力 | 解釈 | メッセージ | ヒストリー

	QUERY PLAN text
1	Seq Scan on goods "商品データテーブル" (cost=0.00..31472.79 rows=1356 width=45) (actual time=0.866..375.295 rows=1470 loops=1)
2	Filter: ((ct3)::text ~~ '%030401%'::text)
3	Total runtime: 375.424 ms

参考程度の数字ですが900倍近く遅いorz…(0.3ms → 372ms)

Seq Scanの文字通りインデックスが使われずシーケンシャルスキャンが動いているのでそりゃ遅いですね。。。

インデックスが効かないと、とにかく遅い。

商品データが１０００件程度なら体感することもないでしょうが、数十万点の商品データからの検索となると、Web上で検索した体感検索時間は、すごく遅く感じてしまいます。

さらに、実行時間が長いため、実行中に次の検索がドンドン流れてきたりして渋滞するため、サーバーにもユーザーにも優しくありません。

Gangsta

じゃあテーブル分割してみるか？

■テーブルをわけてみる・・・

商品データ

出力ビュー

データの出力 | 解釈 | メッセージ | ヒストリー

	商品ID character varying(10)	商品名 character varying(200)
1	0000029323	豚まん
2	0000029325	肉まん
3	0000029324	あんまん
4	0000017554	コロッケ
5	0000017555	フライドチキン
6	0000017556	フランクフルト
7	0000177266	おでん

カテゴリIDを別テーブルに。

出力ビュー

データの出力 | 解釈 | メッセージ | ヒストリー

	商品ID character varying(10)	カテゴリID character varying(6)
1	0000017554	030401
2	0000017554	050205
3	0000017555	030401
4	0000017555	050205
5	0000017556	030401
6	0000017556	050205
7	0000029323	030401
8	0000029323	050103
9	0000029324	030401
10	0000029324	050103
11	0000029325	030401
12	0000029325	050103
13	0000177266	030401

Gangsta

テーブル結合して検索！

```
select
    商品データテーブル.商品ID,
    商品カテゴリIDテーブル.カテゴリID
from
    商品データテーブル
inner join 商品カテゴリIDテーブル
on 商品データテーブル.商品ID=商品カテゴリIDテーブル.商品ID
where
    商品カテゴリIDテーブル.カテゴリID='030401';
```

インデックスも効いていますし、ヒットさせたいデータもとれています。

テーブル結合しているせいか、少しだけ遅いですが、仕様を満たすためには仕方ないところです。。。

出力ビュー

データの出力 | 解釈 | メッセージ | ヒストリー

	QUERY PLAN text
1	Nested Loop (cost=26.78..11222.54 rows=1354 width=19) (actual time=0.201..16.304 rows=1470 loops=1)
2	-> Bitmap Heap Scan on _ _ (cost=26.78..2061.14 rows=1354 width=11) (actual time=0.172..0.410 rows=1470 loops=1)
3	Recheck Cond: (( )::text = '030401'::text)
4	-> Bitmap Index Scan on (cost=0.00..26.44 rows=1354 width=0) (actual time=0.166..0.166 rows=1470 loops=1)
5	Index Cond: (( )::text = '030401'::text)
6	-> Index Scan using okey on goods (cost=0.00..6.75 rows=1 width=19) (actual time=0.010..0.010 rows=1 loops=1470)
7	Index Cond: ((.id)::text = (_ .id)::text)
8	Total runtime: 16.487 ms

Gangsta

しかし、落とし穴が・・・

こんどは、逆に
「複数のカテゴリに
ヒットする検索」
を作ってよ。

上司

たしかにＥＣでは、複数カテゴリ検索の要望は多いです。
（「長袖または七部袖のＴシャツ」のように）

■複数カテゴリ検索してみると・・・

```
select
        商品データテーブル.商品ID,
        商品カテゴリIDテーブル.カテゴリID
from
        商品データテーブル
inner join 商品カテゴリIDテーブル
on 商品データテーブル.商品ID=商品カテゴリIDテーブル.商品ID
where
        商品カテゴリIDテーブル.カテゴリID  in
('030401','050103')
```

出力ビュー

データの出力 | 解釈 | メッセージ | ヒストリー

	QUERY PLAN text
1	Nested Loop (cost=54.90..20665.10 rows=2886 width=19) (actual time=0.332..31.236 rows=2803 loops=1)
2	-> Bitmap Heap Scan on _ ---_ _ (cost=54.90..2420.26 rows=2886 width=11) (actual time=0.317..0.746 rows=2803 loops=1)
3	Recheck Cond: ((_)::text = ANY ('{050103,030401}'::text[]))
4	-> Bitmap Index Scan on _ _ _ (cost=0.00..54.18 rows=2886 width=0) (actual time=0.310..0.310 rows=2803 loops=1)
5	Index Cond: (( )::text = ANY ('{050103,030401}'::text[]))
6	-> Index Scan using _ . _pkey on goods (cost=0.00..6.31 rows=1 width=19) (actual time=0.010..0.010 rows=1 loops=2803)
7	Index Cond: ((.id)::text = (_ .id)::text)
8	Total runtime: 31.557 ms

結果は・・・

出力ビュー

データの出力 | 解釈 | メッセージ | ヒストリー

	商品ID character varying(10)	カテゴリID character varying(50)
1	0000017554	030401,050205
2	0000017555	030401,050205
3	0000017556	030401,050205
4	0000029323	050103,030401
5	0000029323	050103,030401
6	0000029324	050103,030401
7	0000029324	050103,030401
8	0000029325	050103,030401
9	0000029325	050103,030401
10	0000177266	030401

なんかダブっとる(;´д⊂)

Gangsta

テーブル結合のしくみ。

テーブル結合は、数の多い列数に合わせて結合された列が生成されます。

この例のように、１商品がカテゴリテーブル上では複数のレコード（複数のカテゴリ）をもっていると、結合したときに重複したレコードができてしまいます。

PHPやPerlなどのアプリケーション側で重複データを間引くのは処理速度的に非効率的ですので、データベース側でgroup byすることになります・・・

Gangsta

じゃあグループ化 (group by) してみる？

group byしてみると・・・

出力ビュー

データの出力 | 解釈 | メッセージ | ヒストリー

	QUERY PLAN text
1	HashAggregate (cost=20679.53..20708.39 rows=2886 width=19) (actual time=32.889..33.173 rows=1572 loops=1)
2	-> Nested Loop (cost=54.90..20665.10 rows=2886 width=19) (actual time=0.352..31.363 rows=2803 loops=1)
3	-> Bitmap Heap Scan on _ (cost=54.90..2420.26 rows=2886 width=11) (actual time=0.335..0.779 rows=2803 lo
4	Recheck Cond: (( )::text = ANY ('{050103,030401}'::text[]))
5	-> Bitmap Index Scan on _ _ (cost=0.00..54.18 rows=2886 width=0) (actual time=0.326..0.326 rows=2
6	Index Cond: ((_)::text = ANY ('{050103,030401}'::text[]))
7	-> Index Scan using _pkey on _ (cost=0.00..6.31 rows=1 width=19) (actual time=0.010..0.010 rows=1 loops=2803)
8	Index Cond: ((.id)::text = (.id)::text)
9	Total runtime: 33.303 ms

遅い(;´д⊂)

Gangsta

グループ化は重い。。。

"HashAggregate (cost=20679.53..20708.39 rows=2886 width=19)
(actual time=32.889..33.173 rows=1572 loops=1)"

group by を行うと上記処理が行われます。
これがまた意外に早くはないのです。

Gangsta

結果セットが少なければメモリ使用量も少ないので体感できないと思いますが、SQLが複雑になってきたり結果セットが数万になるような検索が投げられると処理に数秒かかったりします。

(今回のサンプルは30万件から1200件程度を検索していますので2ミリ秒程度の違いしか出ていませんが
全体の実行速度の１割増しです)

僕には work_mem などconfをチューニングするくらいしか対策が思いつきません。

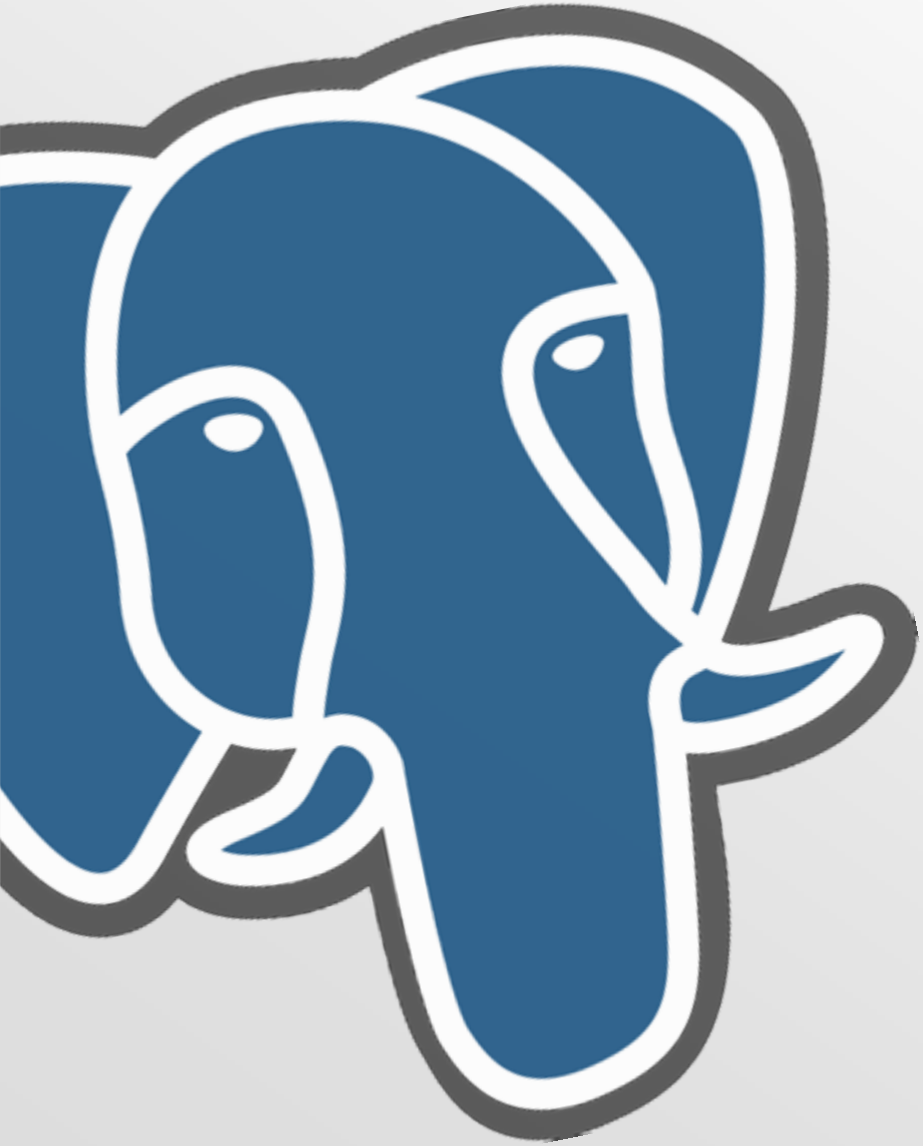
それだと限界がありますし、同時負荷に弱い作りになる可能性もあります。

Gangsta

と、いうわけで

ここまで全部、失敗談でした（＾＾；

Gangsta

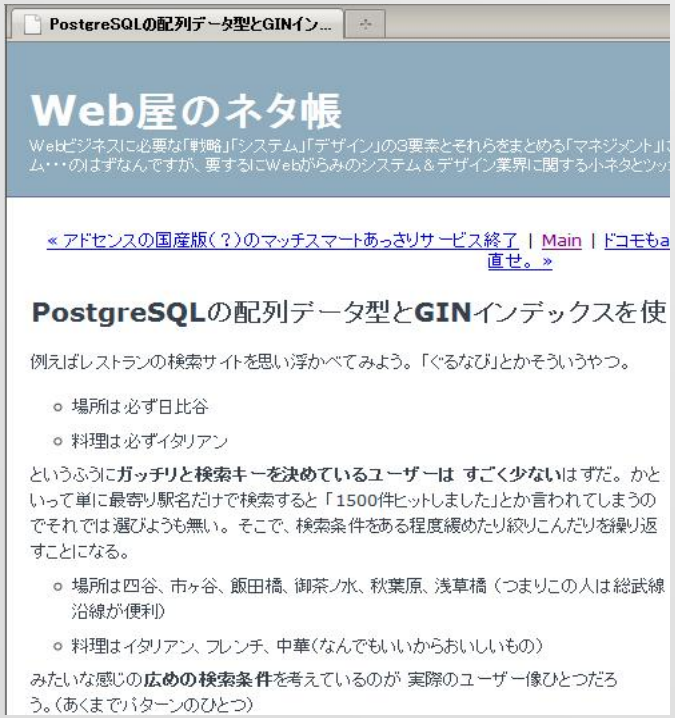


Postgresql
での
解決策は！

■配列型とGINインデックス

そんな失敗を繰り返した３年くらい前、参考になるブログ記事をみつけました。

PostgreSQLの配列データ型とGINインデックスを使って高速検索する
（Web屋のネタ帳）



有名ブログの「Web屋のネタ帳」さん
（http://neta.ywcafe.net/000800.html）

配列型？(´・ω・`)

GIN？(´・ω・`)

■配列型で再設計。

配列型にGINインデックスを張る。

出力ビュー

データの出力 | 解釈 | メッセージ | ヒストリー

	商品ID character varying(10)	商品名 character varying(200)	カテゴリID character varying(6)[]
1	0000017556	フランクフルト	{030401,050205}
2	0000029323	豚まん	{050103,030401}
3	0000029324	あんまん	{050103,030401}
4	0000029325	肉まん	{050103,030401}
5	0000017554	コロッケ	{030401,050205}
6	0000017555	フライドチキン	{030401,050205}
7	0000177266	おでん	{030401}

■配列型を検索。

```
select
        商品ID,
        カテゴリID
from
        商品データテーブル
where
カテゴリID && array ['050103'::character varying,'030401'::character varying]
```

配列型同士で検索されるように出来ているので、上記のように検索要素を配列型に変換して検索します。

配列を検索する演算子はとても特殊ですので詳しくはドキュメントで（PostgreSQL 文書　第 9章関数と演算子　9.17. 配列関数と演算子）
http://www.postgresql.jp/document/current/html/functions-array.html

■配列型で再設計。

出力ビュー

データの出力 | 解釈 | メッセージ | ヒストリー

	QUERY PLAN text
1	Bitmap Heap Scan on goods_search (cost=23.81..4716.09 rows=1490 width=45) (actual time=0.466..2.160 rows=1572 loops=1)
2	Recheck Cond: (ct3 && '{050103,030401}'::character varying[])
3	-> Bitmap Index Scan on goods_search_ct3_index (cost=0.00..23.44 rows=1490 width=0) (actual time=0.413..0.413 rows=15
4	Index Cond: (ct3 && '{050103,030401}'::character varying[])
5	Total runtime: 2.271 ms

Group by したときより約10倍近くのスピードアップ。
もちろん結果セットも思った通りのものが返ってきました。

Gangsta

配列型イイネ！

Postgresqlステキ！

■配列型はＥＣでかなり使える。

今回はカラムが一つだけ配列型のサンプルをしめしましたが、ECで商品を絞り込む要素ってとても複雑です。

例えば、車の部品のように「ジャンル」と「対応車種」という検索要素が、それぞれ１商品に複数設定される(n対n)ような業種の場合は特に有効です。

前述したように、ここでテーブル結合＋グループ処理を行うと、パフォーマンスを悪くします。

１つのレコードしか持たないようにして配列型のデータを持つ設計にすると検索のスピードが非常に速くなります。この設計はテーブルサイズの圧縮やメモリ削減にもつながります。

Gangsta

ＥＣで有効といっても、GINインデックスは商品レコードが数十万あって、複雑な検索が求められるＥＣではないと、威力は発揮しませんが。。。

まあ、様々な検索方法や絞込みが求められるものなので、この手法はとても有効だと思います。

Gangsta

注意事項

■注意事項。

・PostgreSQL独自機能として実装することになります。

他のデータベースでも使用可能な条件のときはお勧めできません。

・データベースオブジェクトを利用しようとする場合、where()などの関数で配列で引数に渡しても、配列型専用演算子に対応していません。

この場合where()の部分はsql書式にする必要があります。
または、抽象クラスもしくは既存PostgreSQLのクラスを継承・拡張して配列型に対応できるようなメソッドで対応する方法があります。

メソッドを実装する際は文字列エスケープに注意してください。
　{'A','B'}　という配列型文字列をエスケープしてしまうと
　{''A'',''B''}　と余計にエスケープしてしまうからです。
必ず配列の中身だけをエスケープしましょう

Gangsta

これでやっと、
カテゴリ検索が速くなりました。

次は・・・

n-gram組み込み型全文検索
で
商品検索を高速化

■全文検索って何？

もともとはサーバー上にあるファイルから文字列を検索するための技術です。

ＥＣサイトではデータをデータベースから取り出すことが多いためファイルとしてデータをもっていません。
（持たせる手法もありますがここでは割愛）

組み込み型全文検索は、データベースやアプリケーションの処理に全文検索機能を組み込んで、全文検索が使えるよう機能拡張してくれるステキなやつです。

n-gramは、それぞれの文字にインデックスを張るような概念なので、ちょっとした長さの文字でも、かなりの（メガ単位当たり前）データ領域を使用します。
数十万件のレコードとなるとギガ単位も当たり前みたいになってきます。

形態素解析は、単語単位にデータをもっているのでインデックスサイズがそこまで大きくなることはありませんが、形態素解析を利用する場合はどうしても辞書データが必要になります。

ECでよくあるフリーワード検索を形態素解析で作ろうとすると、数十万点を超える商品データに対してヒットさせる語を考えてメンテナンスし続ける必要があるため、n-gram方式で実装しています。

■ＥＣのフリーワード検索

商品名や型番、メーカー名（CDや書籍などは作家名）などなどが対象。サイトによっては商品説明文にヒットさせる場合もありますね。

大文字　←→　小文字
全角　←→　半角

これくらいの入力揺らぎは吸収して検索結果を返すのが「買い物しやすいサイト」への第一歩かと。

Gangsta

ただ、これ検索対象となるカラムを指定して like で繋げるなんてことでは対応できませんよね。

入力揺らぎへの対応まで考えていくと大変です。

中間一致でとっていくのでインデックスが効きませんし、そもそも全て OR検索 なんて、遅くて使い物になりません。。。

■実装例

僕が実装した例を紹介します
（もちろん全てのサイトに当てはまるものではないです）

商品テーブルとは別に検索用のテーブルをもって、フリーワード検索用のフィールドに検索されるデータをすべて詰め込む方式にしました。

ＰＨＰで例を示しますが、検索フィールドにデータを入れるときに下記の処理をします。

1. mb_convert_kanaですべて半角英数化
2. strtoupperですべての英語を大文字化
3. mb_convert_kanaですべて全角英数化
(ここはやる必要はないですがセキュリティのフェイルセーフとして)
4. 仕様次第ではハイフンや波ダッシュなど削除。

しかし、この方法ではまだ実用に耐える速度にはなりません。そこで全文検索の登場です。

PostgreSQL　textsearch_senna　をオススメします。

理由は簡単でPostgreSQLで今もっとも情報が多いからです。

サーバーへのインストールはある程度OSの知識がないと厳しいので、そっち方面は苦手というアプリ屋さんは、サーバー屋さんにお願いしたほうがいいと思います。Windowであればバイナリが提供されているのでローカル環境で試してみるのも良いかも。

Gangsta

このモジュールを利用する最大のポイントは

中間一致検索でLike構文を乗っ取って動作すること

通常のSENNA検索構文は特殊構文にしなければいけませんが　textsearch_senna　では、LIKE インデックス(like_ops)というのを張れば LIKE文 を書き換えなくても驚異的なパフォーマンスを発揮します。

■pgAdmin IIIについて注意

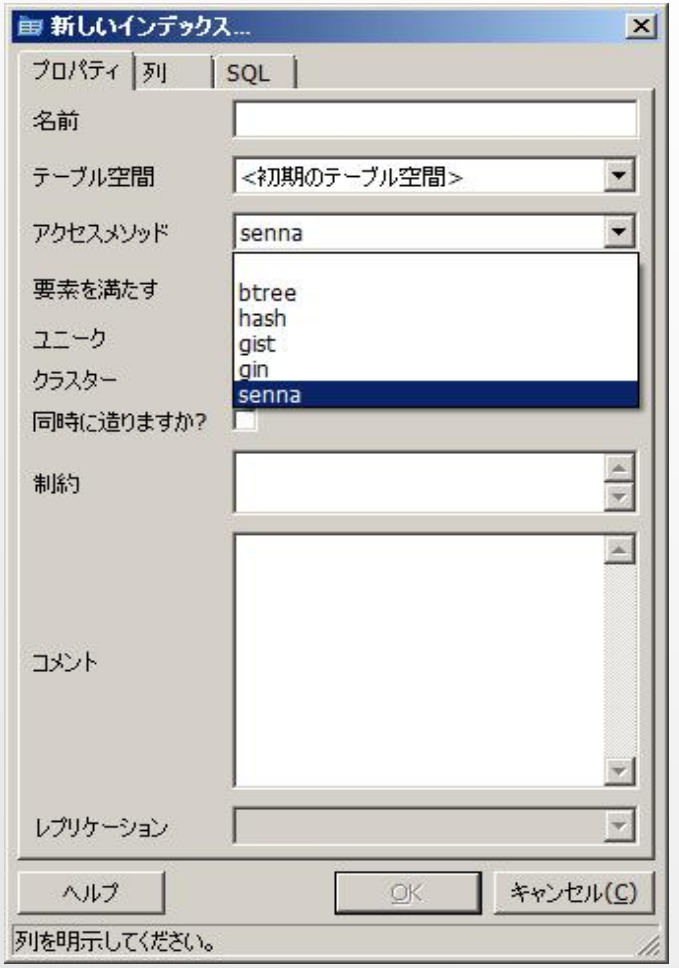
psqlとかコマンドラインは苦手って方は、pgAdmin III を利用しているかもしれません。便利ですよね。

しかし、pgAdmin III は like_ops でインデックスを張らず、通常の sennaインデックスを使用します。

インデックスを張る場合は

CREATE INDEX ユニークなインデックス名
ON テーブル名 USING senna（カラム名 like_ops）;

のようなSQLをクエリーツールから流す必要があります。



textsearch_senna速い！

普通の中間一致

出力ビュー

データの出力 | 解釈 | メッセージ | ヒストリー

	QUERY PLAN text
1	Seq Scan on (cost=0.00..77627.07 rows=6108 width=11) (actual time=0.292..677.508 rows=2348 loops=1)
2	Filter: (~~ '%ステッカー%'::text)
3	Total runtime: 677.798 ms

sennaインデックス利用時

出力ビュー

データの出力 | 解釈 | メッセージ | ヒストリー

	QUERY PLAN text
1	Bitmap Heap Scan on (cost=133.33..39266.54 rows=16656 width=11) (actual time=9.974..31.441 rows=4932 loops=1)
2	Recheck Cond: (%% 'ステッカー'::text)
3	-> Bitmap Index Scan on _freeword_index (cost=0.00..129.17 rows=16656 width=0) (actual time=7.252..7.252 rows=15548 loops=1)
4	Index Cond: (%% 'ステッカー'::text)
5	Total runtime: 31.840 ms

すごく速いです！！！！
しかし、普通の中間一致がsennaインデックスよりも速く動作することもありますので、ちゃんとパフォーマンス確認してから投入を。。

Gangsta

Tips

sennaのインデックスファイルは別途PostgreSQL本体管理外に保存されます。

VACUUM FULLなどをかける場合に別途そのインデックスファイルを消す必要あり。

ファイルを消す関数自体はsennaモジュールに定義されていますが、バッチ処理でDB丸ごとVACUUM FULLをかける場合など、自動処理できません。

この場合は、pg_indexesテーブルの中にあるindexdefを検索してsennaという文字列が含まれているものを検索してファイルを消すバッチ処理をVACUUM FULLのあとに呼び出すことで対策しています。

Tips-2

いま話題の（笑）PostgreSQL9.0レプリケーション機能を利用すると、当たり前ですが再インデックスしてもsennaのインデックスは同期されません。

いまのところ、力技ですが *.SEN* ファイルをlsyncdとrsyncでリアルタイム同期かけて対応しています。

実際のところ同期のタイミングはズレるので確実な動作は理論上保証されませんが・・・

text_search_senna注意事項

・インデックスファイルの肥大化に注意。

・すでにSENNAは開発終了して（安定しているとも言えますけど）、次期プロジェクトgroongaが始まっています。

【余談】
text_search_sennaつくった板垣さんがgroongaに手を出しているようなので、もっと便利なの作ってくれると期待しておりまーす！！！！！！！！！！（人任せ＆期待）

これでフリーワード検索も
速くなりましたね。

次は検索から離れた話題です。

Gangsta

個人情報や決済での
データ保存はトランザクションを
上手に使おう

■トランザクション

複数テーブルを同時更新する際、途中でエラーしたら
更新中のデータを元に戻せる機能。

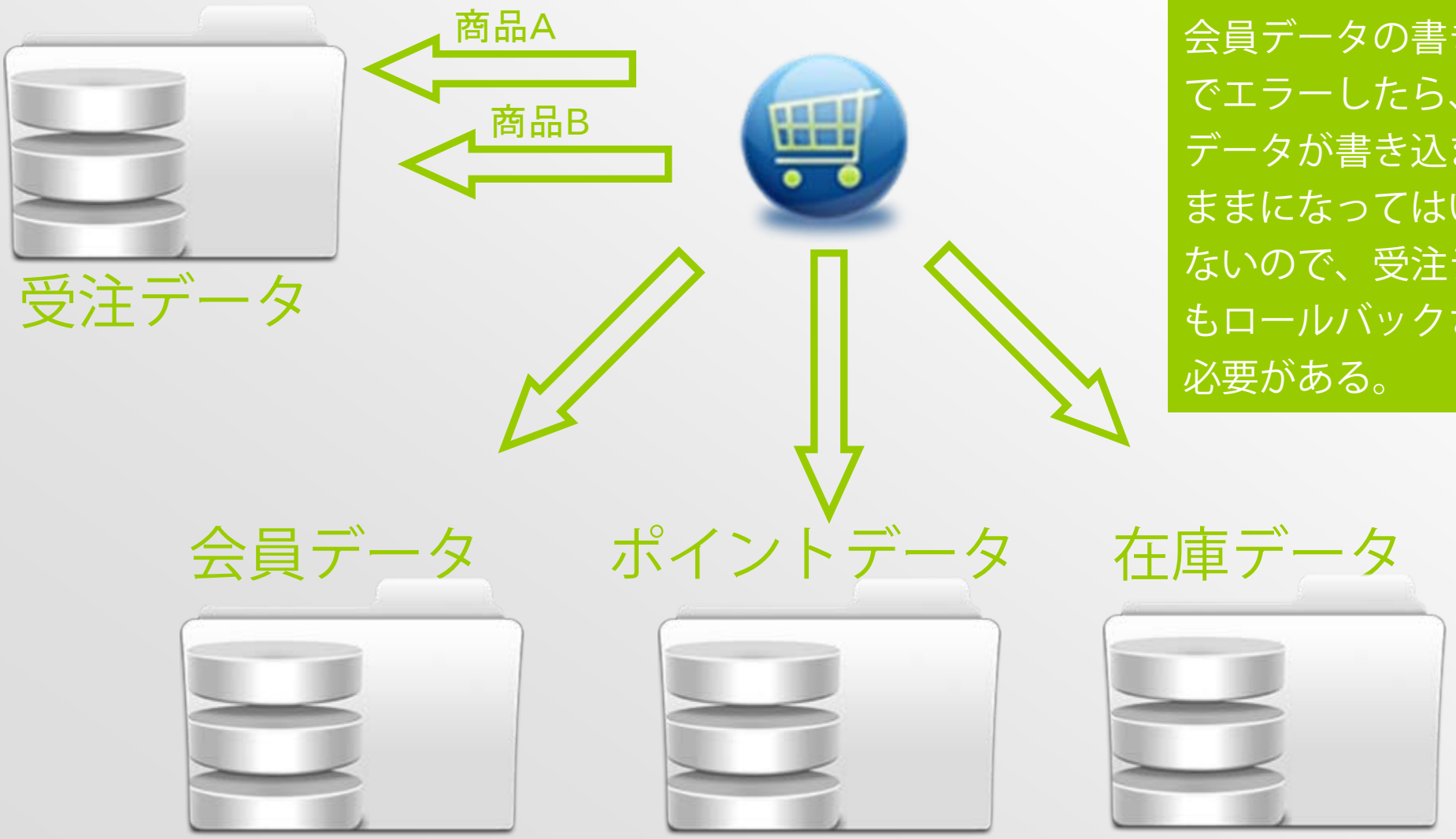
※上記はトランザクションの一部を噛み砕いたものでこれが全てではありません
（今回は割愛します）

詳しく知りたい方は、２部で
スピーカーをされる鈴木 啓修
さんの書籍がオススメ！

ちょっと内容が古くなってい
ますが良書です。

Postgres9.0版を書いてくださ
ることを期待してますー♪

■なぜトランザクションなのか



受注データ更新のあと会員データの書き込みでエラーしたら、受注データが書き込まれたままになってはいけないので、受注データもロールバックされる必要がある。

Gangsta

トランザクションという
「ここから、ここまでは一連の更新作業です」
という明示的な宣言。

これなら途中でエラーした時に、
全ての更新を止めることができます。

もしトランザクションを使わなかったら、ポイントや在庫のテーブルが古いまま受注だけが更新されるという恐ろしいことが発生します。

適切にトランザクション処理を使って、更新が途中でコケたら「システムエラーです(´・ω・`;)」などの表示を返すような設計が一般的です。

■環境についてちょっと

と、いうわけでトランザクションがサポートされていないＤＢはＥＣには不向きだと思います。

トランザクションがサポートされているデータベース管理システムでもストレージエンジンの選択によりトランザクションがつかえないときがあります。

有名なパターンがMySQL+MyISAMです。

MyISAMは、MySQL+SENNA=TRITTONで使用されていたのと、高速だったため、以前はよく見かけました。ＥＣなどトランザクションが活躍する環境ではInnoDBに変更することをオススメします。

なお参考までに・・・

MySQL5.5では、InnoDBのパフォーマンスが200%で、デフォルト採用だそうです。
（まだリリースされてませんけど）

トランザクションの重要性が世の中に浸透してきたんだなあと個人的には感じました。

トランザクションはPostgreSQL以外のデータベースでも使い方を間違えなければ問題ありませんが、デフォルトでトランザクションが利用できるPostgreSQLは、使いやすいと思ってます（と、ポスグレの話に戻す）

次の話題いきます。

Gangsta

負荷対策
と
レプリケーション

■9.0＋クラウドもどき

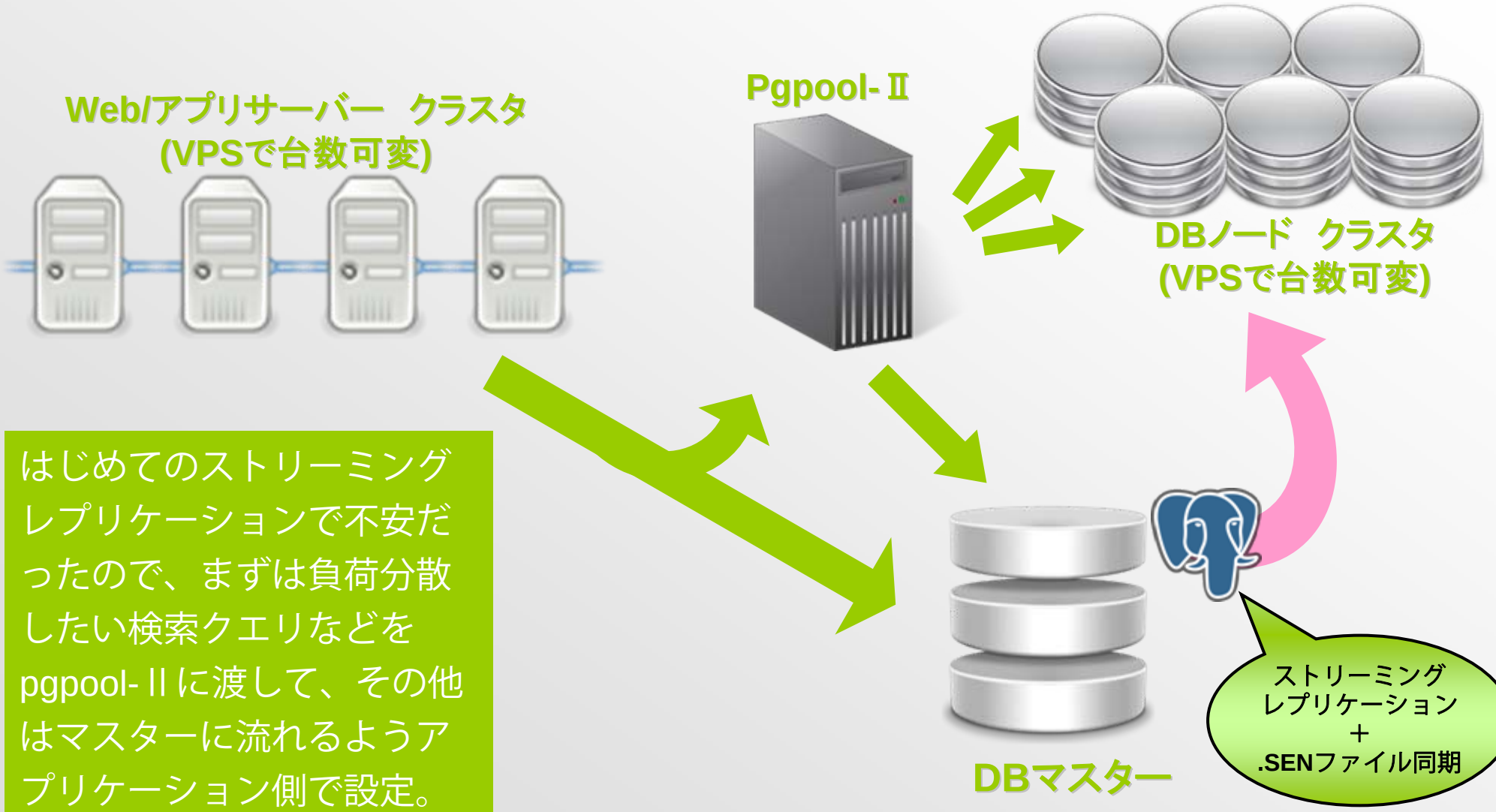
普段はアクセス少ないのに、セールをすると津波のようになるアクセスの波に耐えうるの環境を作るべく、
一週間ほど前に、僕が関っている某ＥＣサイトを

Postgresのバージョンを9.0にし、レプリケーション機能でＤＢサーバーを増やしました！

Amazon Web Servicesとか怖いので、国内のサーバー屋さんのVPSでDBノード軍団を編成ヽ(｀Д´)ノ

当然、pgpool II はver3.0にしました。

■クラウドもどき


Web/アプリサーバー　クラスタ
(VPSで台数可変)
Pgpool-Ⅱ
DBノード　クラスタ
(VPSで台数可変)
はじめてのストリーミングレプリケーションで不安だったので、まずは負荷分散したい検索クエリなどをpgpool-Ⅱに渡して、その他はマスターに流れるようアプリケーション側で設定。
DBマスター
ストリーミング
レプリケーション
＋
.SENファイル同期

■未解決問題。

正直レプリケーションによるトラブルは少なめ。
多分更新系のSQLをマスター側に直接接続していたのがよかった。

ただし、ストリーミングレプリケーションの非同期ラグは正直秒単位では発生している時がある。

■まだまだマスターDBサーバーの負荷が高くてレプリケーションが間に合わないのか？

■pg-pool-IIのことがよくわかっていない（笑）
SRA OSS 北川様の講演で今日わかれば嬉しいｗｗｗ

Gangsta

Tips

僕もよくわかってませんがpg-pool II の設定が重要。

例えば、アプリ側からsennaのreindex構文を投げる場合はpg-pool II にreindex構文はマスター側に投げるという登録しておくなどが必要。

PostgreSQLの設定だけでなくpg-pool II の設定にも気を配りましょう。

■クラウド化は簡単ではない。

Amazon EC2などクラウドサーバーは、アクセスの波の幅が大きいWebサービスで、負荷が高まりそうな期間だけサーバーを増やして対策できるのが魅力です。

アクセスの台風は、ずっと続くものではないため、こういった負荷に耐えられるだけのサーバーを用意していたらお金の無駄で、中小企業に無理な話です。

いつでも増やせて、稼働時間分だけ費用発生するのは魅力的ですが、普通のWebサービスがクラウド対応するのは簡単ではありません・・・頑張りましょう。

次の話題いきます。

Gangsta

PostgreSQL(8系/9系)で Webプログラマーが 知っておくべき機能

Gangsta

■テーブルスペースサポート(8.0)

物理HDDをわけることができるのでハードウェア構成
変更だけでスピードアップが可能。
アプリ屋にはとても便利。
なおテーブルパーティショニングはアプリ屋には色々
不便なことが多いので使わないほうがいいかも。

http://lets.postgresql.jp/documents/technical/partitioning/

Gangsta

■VACUUMの改善(8.0)・HOT(8.3)

Postgresqlの黒歴史。
今でもそのイメージは根付いてしまってますね。

auto_vacuumやHOTの実装によりほぼメンテナンスフリーとなってます。

ただし、text_search_sennaを使うときは定期的に専用のバキュームを行ったほうがいいとはおもいます。

■バッファ管理（8.2）

おかげでマルチプロセッサ環境での高速化が顕著になりました。劇的に速くなった。ばんざーい。

■ビットマップスキャン（8.1）

ＥＣで多用されるのでＯＲ検索の高速化は嬉しい。
アプリはこの機能で対応することはなくなった。

■ＧＩＮインデックス（8.2）

すでにご説明しましたとおり。
構文が特殊なのでPostgresql独自の実装が必要。
すごく速いので価値あります。

★7系や8系前半を使っているシステムであれば8.4や9.0をインストールするだけで速くなる可能性があります！

Gangsta

■レプリケーションホットスタンバイ(9.0)

すでに説明済み＆二部が楽しみ！

Gangsta

■ オプティマイザ: 利用されないJOINの削除 (9.0)

JOIN していても、データを取得しなければ JOIN 対象から外す最適化が行われるようになりました。

特に OUTER/INNER JOIN を利用するシステムでは、PDOなどで実装しているアプリ屋さんのSQLチューニング手法としてかなり有効です。

JOIN を切り替えるための不毛なＩＦ文を書かないでガリガリとテーブルJOINしちゃえます。

正しく指定しないとオプティマイザが理解できるかどうかは不明なので注意（誰か解説してー！）

■ pgAdmin-III

pgAdmin とても便利です！
最新版では早くもレプリケーション機能にも対応していたり、クエリ解釈がグラフィカル表示になっていたり・・・

おしいところは日本語のわかりやすい解説がないんですよね。誰か解説つくるか、セミナーやってくれないかなー

psqlとかアプリ屋さんにはちょっと縁遠いツールなんですよね・・・とぼやいてみます(´・ω・`;)

Gangsta

最後にオマケ。

Gangsta

最近は、本業の傍ら

Webマーケティングチーム「Gangsta」

としてWebサービスの立ち上げや改善の

お手伝いをしております。

kozu@gangsta.jp

何かあればご相談ください。

Gangsta

ご清聴ありがとうございました。

ご質問・ツッコミはツイッターかメールで。

↓

Mail: kozu@gangsta.jp

Twitter: @CSTYLES_JP