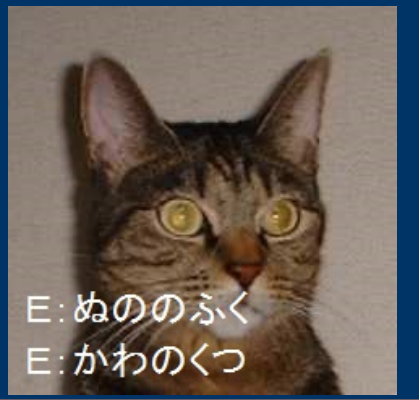
PGCon2014

(2014-12-05)

JSONB データ型を使ってみよう

ぬこ@横浜 (@nuko_yokohama)

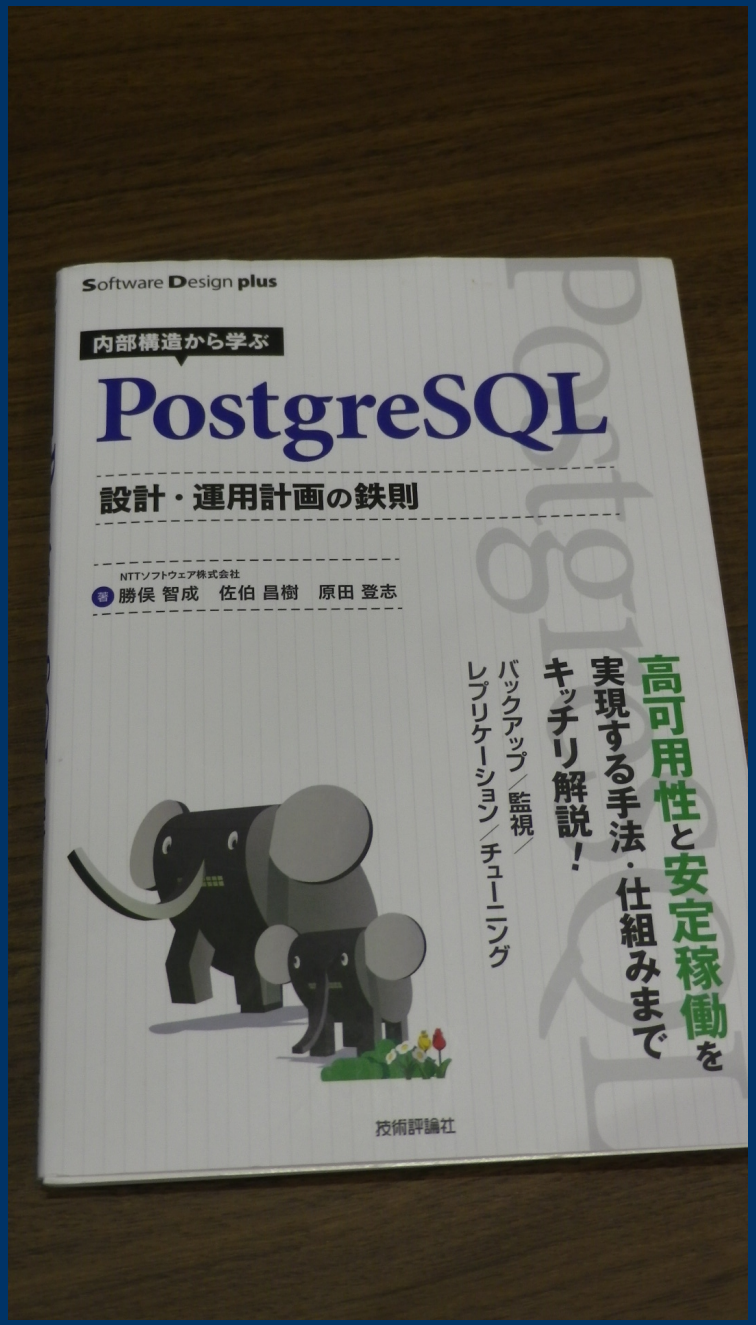


自己紹介

PostgreSQL

SELECT '{
"名前":"ぬこ@横浜",
"Twitter":"@nuko_yokohama",
"所属":{"会社":"某通信系ソフトウェア会社",
"部署":"クラウドな感じ",
"勤務地":"ミナトミライ・スゴイトオイビル"},
"仕事":"PostgreSQL あれこれ",
"趣味":["猫","ラーメン","スパ","バイク旅","PostgreSQL"],
"メモ":["PostgreSQL は 7.4 の頃からの腐れ縁",
"ときどき PostgreSQL イベントで喋ってます","ただのユーザ"],
"作ったもの":["漢数字型 (ksj)","類似文字型 (ntext)",
"new4j_fdw", "hit_and_blow BGW", "pg_heartman"]}'::jsonb;

最近、 PostgreSQL の本を
書きました。
よろしくおねがいします⇒



まずは・・・

祝・ PostgreSQL 9.4.0 Release! （のはず）

今日は、その
postgreSQL 9.4 の
新機能の一つ、
JSONB 型について
話そうと思います。

目次

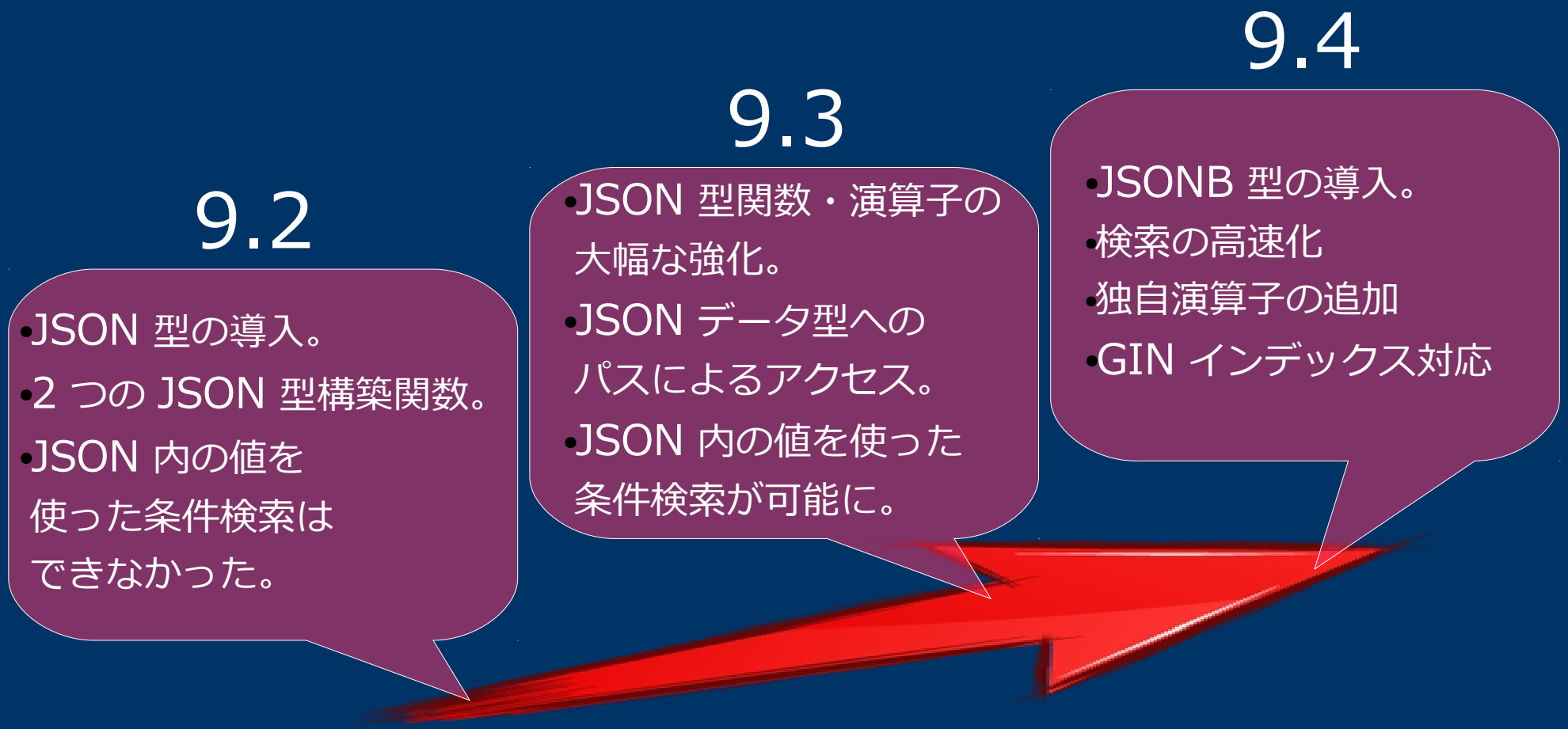
JSON/JSONB 型概説
JSONB 型の実装 / 性能
アプリケーションから使う
JSONB の想定適用例

JSON/JSONB 型概説

JSON とは

JavaScript Object Notation
XML よりも軽量
構造化されたデータを
文字列で表現可能

PostgreSQL における JSON への取り組み



9.2 から順調に進化。 9.5 でも…？

JSON 型でできること

JSON 文字列のパース
キーによる値の取り出し
パスによる値の取り出し
PostgreSQL 配列や行との変換
etc ・・・

JSON 文字列のパース

PostgreSQL へのデータ格納時に格納される JSON が正しい書式かチェックする。
正しくない書式でない場合 PostgreSQL がエラーにする。

JSON 文字列のパース例

正しい JSON 文字列

```
jsonb=# SELECT '{"key1":"value1", "key2":[100, 20, 5]}'::json;
                  json
-----------------------------------------
 {"key1":"value1", "key2":[100, 20, 5]}
(1 row)
```

誤った JSON 文字列

```
jsonb=# SELECT '{"key1":"value1", "key2":[100, 20, ]}'::json;
ERROR:  invalid input syntax for type json
LINE 1: SELECT '{"key1":"value1", "key2":[100, 20, ]}'::json;
               ^
DETAIL:  Expected JSON value, but found "]".
CONTEXT:  JSON data, line 1: {"key1":"value1", "key2":[100, 20, ]...
```

※ パースは JSON/JSONB 型に変換されるときに行われる。

キーによる値の取り出し

JSON 型に対してキーを指定して対応する値を取得する。

”->”, ”->>” 演算子など

キーによる値の取り出し

JSON 型全体を取得

```
jsonb=# ¥pset null (null)Null display (null) is "(null)".
jsonb=# SELECT data FROM test ORDER BY id LIMIT 3;
                                                                    data


---------------------------------------------------------------------------
-----
-------------------------------------------------------------
 {"Id": 0, "Email": "Laverna@junius.io", "Country": "Paraguay", "Full
Name": "Ca
rolyne Kohler", "Created At": "1987-08-21T18:42:02.269Z"}
 {"Id": 1, "Country": "France", "Full Name": "Paul Weber DVM", "Created
At": "19
89-03-16T14:37:36.825Z"}
 {"Id": 2, "Email": "Elbert@norma.co.uk", "Country": "Uzbekistan", "Full
Name":
"Florence Murphy", "Created At": "1980-02-19T04:16:52.113Z"}
(3 rows)
```

キーによる値の取り出し

キーから JSON オブジェクトを取得

```
jsonb=# SELECT data->'Email' as email , data->'Full Name' as fullname FROM
test ORDER BY id LIMIT 3;
        email          |     fullname
-----------------------+------------------
 "Laverna@junius.io"   | "Carolyne Kohler"
 (null)                | "Paul Weber DVM"
 "Elbert@norma.co.uk"  | "Florence Murphy"
(3 rows)
```

キーから文字列を取得

```
jsonb=# SELECT data->>'Email' as email , data->>'Full Name' as fullname
FROM test ORDER BY id LIMIT 3;
       email         |    fullname
---------------------+----------------
 Laverna@junius.io   | Carolyne Kohler
 (null)              | Paul Weber DVM
 Elbert@norma.co.uk  | Florence Murphy
(3 rows)
```

パスによる値の取り出し

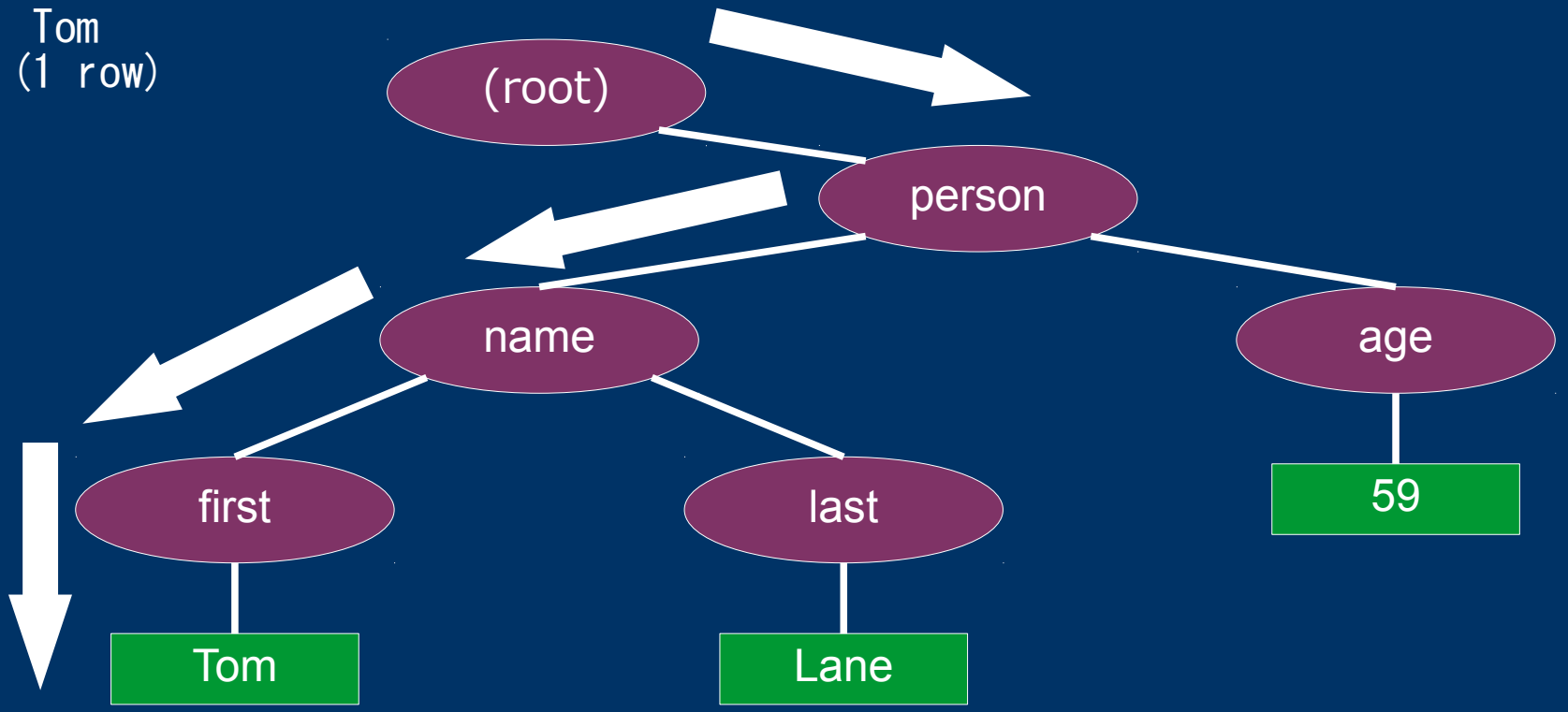
JSON 型に対してパスを指定して対応する値を取得する。

“#>”, “#>>” 演算子

パスによる値の取り出し

person, name, first のパスで取得

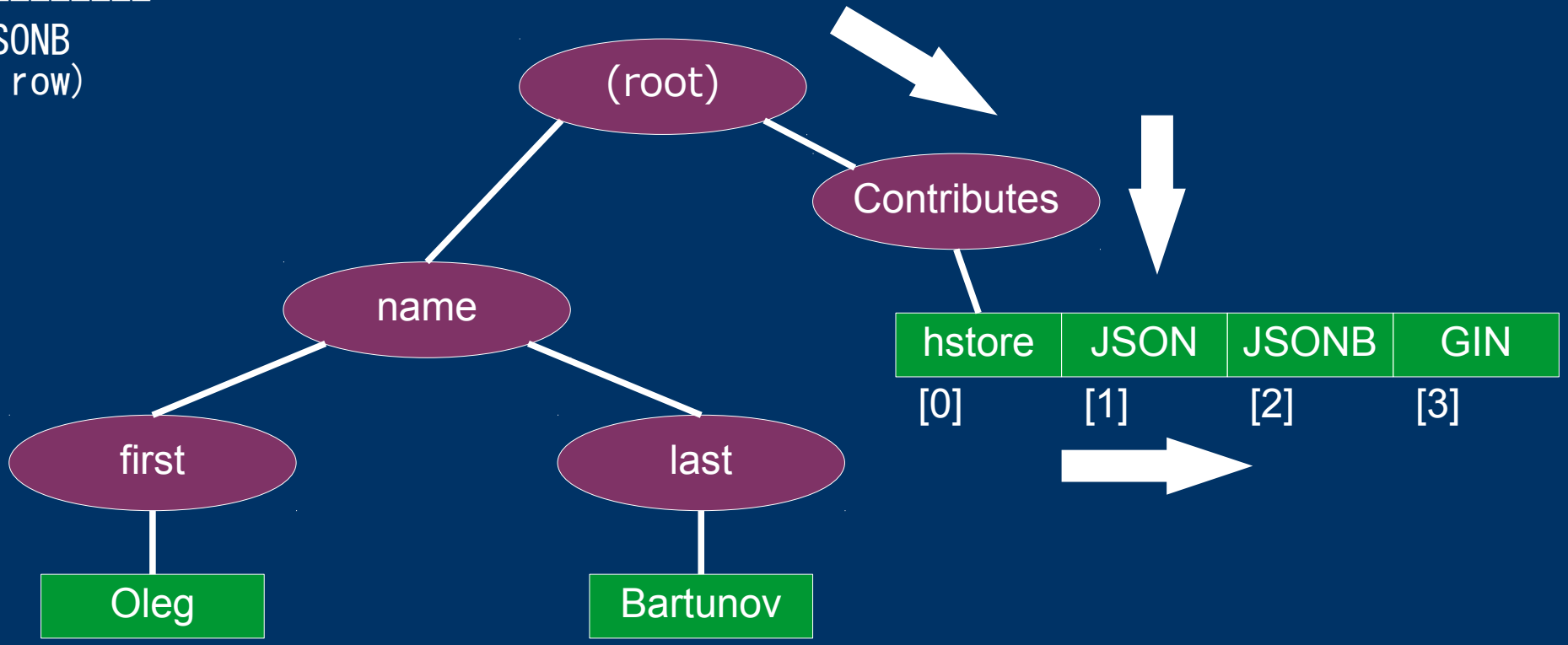
```
jsonb=# SELECT '{"person":{"name":
{"first":"Tom","last":"Lane"},"age":59}}'::json #>> '{person, name,
first}';
 ?column?
----------
 Tom
(1 row)
```



パスによる値の取り出し

Contributes の 2 番目 (0 相対) を取得

```
jsonb=# SELECT '{"Name":
{"First":"Oleg","Last":"Bartunov"},"Country":"Russia","Contributes":
["hstore","JSON","JSONB","GIN"]}'::json #>> '{Contributes,2}';
 ?column?
----------
 JSONB
(1 row)
```



式インデックスとの組合せ

キーやパスによって取り出した
値は式インデックスとしても
使用可能
⇒JSON 型を条件列として
インデックス検索も可能！

式インデックスとの組合せ

式インデックスの設定

```
jsonb=# SELECT data FROM json_t LIMIT 1;

data

-------------------------------------------------------------------------
-----
-------------------------------------------------------------------------
 {    "Email": "Laverna@junius.io",    "Created At": "1987-08-
21T18:42:02.269Z",
    "Country": "Paraguay",    "Id": 0,    "Full Name": "Carolyne Kohler"}
(1 row)

jsonb=# CREATE INDEX json_id_idx ON json_t USING btree ((data->>'Id'));
CREATE INDEX
jsonb=# ¥d json_t
                    Unlogged table "public.json_t"
 Column |  Type   |                       Modifiers
--------+---------+-------------------------------------------------
 id     | integer | not null default nextval('json_t_id_seq'::regclass)
 data   | json    |
Indexes:
    "json_id_idx" btree ((data ->> 'Id'::text))
```

式インデックスとの組合せ

式インデックスを利用した検索

```
jsonb=# EXPLAIN ANALYZE SELECT data FROM json_t WHERE data->>'Id' = '1000';
                                                          QUERY PLAN


_____________________________________________________________________________
_____
_____________________________________
 Bitmap Heap Scan on json_t  (cost=4.68..130.11 rows=50 width=32) (actual
time=0
.078..0.078 rows=1 loops=1)
   Recheck Cond: ((data ->> 'Id'::text) = '1000'::text)
   Heap Blocks: exact=1
   ->  Bitmap Index Scan on json_id_idx  (cost=0.00..4.66 rows=50 width=0)
(actual time=0.072..0.072 rows=1 loops=1)
         Index Cond: ((data ->> 'Id'::text) = '1000'::text)
 Planning time: 0.248 ms
 Execution time: 0.106 ms
(7 rows)
```

その他の JSON 関数

PostgreSQL 配列と JSON 配列との変換
行から JSON への変換
JSON から PostgreSQL 行への展開
キー集合の取得

その他の JSON 関数

PostgreSQL 配列と JSON 配列との変換
行から JSON への変換
JSON から PostgreSQL 行への展開
キー集合の取得

PostgreSQL 配列と JSON 配列との変換

```
jsonb=# SELECT array_to_json(ARRAY[1, 3.14, -1.2e3]);
 array_to_json
---------------
 [1,3.14,-1200]
(1 row)
```

PostgreSQL 配列を
JSON 配列に変換

```
jsonb=# TABLE bar;
 id |     n_datas      |        t_datas
----+------------------+-----------------------
  1 | {1.32,9.76,5.55} | {bdho,fjoal}
  2 | {6.43,0.48}      | {vbwdahoi,3dsai,cfjia}
  3 | {}               | {}
(3 rows)
```

```
jsonb=# SELECT array_to_json(bar.t_datas) FROM bar;
        array_to_json
------------------------------
 ["bdho","fjoal"]
 ["vbwdahoi","3dsai","cfjia"]
 []
(3 rows)
```

テーブル内の特定の列を
JSON 化した例

行から JSON への変換

```
jsonb=# TABLE foo;
 id | n_data |  t_data
----+--------+----------
  1 |   8.93 | 366fd4cf
  2 |   5.23 | 3f0a243b
  3 | (null) | (null)
(3 rows)

jsonb=# SELECT row_to_json(foo.*) FROM foo;
                 row_to_json
-----------------------------------------
 {"id":1,"n_data":8.93,"t_data":"366fd4cf"}
 {"id":2,"n_data":5.23,"t_data":"3f0a243b"}
 {"id":3,"n_data":null,"t_data":null}
(3 rows)
```

テーブル全体を
JSON 化した例

カラム値が null の場合は
JSON の null に変換

JSON から PostgreSQL 行への展開

```
jsonb=# select * from json_each_text('{"id": 2, "age": 59, "name": {"last":
"Lane", "first": "Tom"}}');
 key  |               value
------+-----------------------------------
 id   | 2
 age  | 59
 name | {"last": "Lane", "first": "Tom"}
(3 rows)
```

※ 内側のキー ("first", "last") と値は展開しない。

※ key, value という固定の列名として返却される。

※ key も value も TEXT 型として返却される。

キー集合の取得

以下のような JSONB を含む全レコードからキーを取得する。

```
jsonb=# TABLE test;
 {"id": 1, "name": {"first": "Oleg"}, "distribute": ["GIN", "hstore",
"json", "jsonb"]}
 {"id": 2, "age": 59, "name": {"last": "Lane", "first": "Tom"}}
 {"id": 3, "name": {"nickname": "nuko"}, "distribute": ["ksj", "neo4jfdw"]}
```

jsonb_object_keys 関数の結果

```
jsonb=# SELECT DISTINCT jsonb_object_keys(data) FROM test;
 distribute
 age
 id
 name
```

※ 内側のキー ("first", "last") は取得できない点に注意！

※ 自分で Outer キーを指定して取得した JSONB に対して jsonb_object_keys を発行することは可能。

JSON と JSONB

JSON 型

格納形式は文字列

JSONB 型

格納形式は規定のバイナリ形式

データ型の導入

専用演算子の追加

JSON と JSONB の格納状態例

beta-3 で確認

入力 JSON 文字列

```
{"aaa" : "AAAA", "bbb1" : {"cccc": "CCCC1"}, "bbb2" :{"cccc" : "CCCC2"} }
```

JSON 型の格納状態

入力文字列そのまま

```
00001fb0  95 7b 22 61 61 61 22 20  3a 20 22 41 41 41 41 22  |.{"aaa" : "AAAA"|
00001fc0  2c 20 22 62 62 62 31 22  20 3a 20 7b 22 63 63 63  |, "bbb1" : {"ccc|
00001fd0  63 22 3a 20 22 43 43 43  43 31 22 7d 2c 20 22 62  |c": "CCCC1"}, "b|
00001fe0  62 62 32 22 20 3a 7b 22  63 63 63 63 22 20 3a 20  |bb2" :{"cccc" : |
00001ff0  22 43 43 43 43 32 22 7d  20 7d 00 00 00 00 00 00  |"CCCC2"} }......|
```

JSONB 型の格納状態

バイナリ化された
データが格納される

```
00001fa0  b5 03 00 00 20 03 00 00  80 04 00 00 00 04 00 00  |.... ............|
00001fb0  00 04 00 00 00 16 00 00  50 18 00 00 50 61 61 61  |........P...Paaa|
00001fc0  62 62 62 31 62 62 62 32  41 41 41 41 00 01 00 00  |bbb1bbb2AAAA....|
00001fd0  20 04 00 00 80 05 00 00  00 63 63 63 63 43 43 43  | ........ccccCCC|
00001fe0  43 31 00 00 00 01 00 00  20 04 00 00 80 05 00 00  |C1...... ........|
00001ff0  00 63 63 63 63 43 43 43  43 32 00 00 00 00 00 00  |.ccccCCCC2......|
```

バイナリ形式で格納すると何が嬉しいのか？

JSONB は格納効率を優先はしていない

格納サイズに関しては
むしろ JSON より大きくなる
ケースもある。

(Oleg 氏のレポートでは約 4% 程度増加する
ケースが報告されている)

JSONB の格納性能も
JSON と比較すると遅い。

・格納容量が大きい

・シリアライズ化処理

（性能検証結果は後述）

JSONB は検索効率を優先

JSON 関数を使ってキーによる
検索を行ったときに
JSON と比較して非常に高速
（性能検証結果は後述）

JSONB データ型

JSON 自体にはデータ型の概念がある。
PostgreSQL の JSONB 型もデータ型に一部対応している。

JSON 型	対応する PostgreSQL 型	備考
文字型	text	エンコーディングの問題は注意。 なお、キーは文字型扱い。
数値型	numeric	浮動小数点型の NaN や Infinity の表記は許容されない。
時刻型	（未対応）	PostgreSQL では文字列扱い。
真偽値	boolean	小文字の true/false のみ許容。 (on/off とかもダメ)
null	_	小文字 null のみ許容。 SQL NULL とは別概念。

JSONB は入力した
JSON 文字列は保持されない。

JSON は入力文字列を
そのまま保持する。

JSONB 入力結果が保持されない例（数値型）

```
jsonb=# INSERT INTO jsonb_t VALUES
(1, '{"key":1, "value":100}'),
(2, '{"key":2, "value":100.0}'),
(3, '{"key":3, "value":1.00e2}'),
(4, '{"key":4, "value":1.000e2}')
;
INSERT 0 4
jsonb=# SELECT * FROM jsonb_t ;
 id |              data
----+--------------------------------
  1 | {"key": 1, "value": 100}
  2 | {"key": 2, "value": 100.0}
  3 | {"key": 3, "value": 100}
  4 | {"key": 4, "value": 100.0}
(4 rows)
```

小数なしの numeric として同値

小数点ありの numeric として同値

指数表記でなくなる

JSON だと単に文字列として格納されているだけなので、入力形式そのままで出力される

```
jsonb=# INSERT INTO json_t VALUES
jsonb-# (1, '{"key":1, "value":100}'),
jsonb-# (2, '{"key":2, "value":100.0}'),
jsonb-# (3, '{"key":3, "value":1.00e2}'),
jsonb-# (4, '{"key":4, "value":1.000e2}')
jsonb-# ;
INSERT 0 4
jsonb=# SELECT * FROM json_t;
 id |              data
----+--------------------------------
  1 | {"key":1, "value":100}
  2 | {"key":2, "value":100.0}
  3 | {"key":3, "value":1.00e2}
  4 | {"key":4, "value":1.000e2}
(4 rows)
```

JSONB 入力結果が保持されない例（重複キー、キー名の順序）

同一階層で重複したキーがある場合、後に記述したもののみ有効となる。
また、キー名でソートされている。

```
jsonb=# SELECT
'{"key_z":99, "key_q": 20, "key_q" :25, "key_a" : 3}'::json,
'{"key_z":99, "key_q": 20, "key_q" :25, "key_a" : 3}'::jsonb;
-[ RECORD 1 ]-----------------------------------------
 json  | {"key_z":99, "key_q": 20, "key_q" :25, "key_a" : 3}
 jsonb | {"key_a": 3, "key_q": 25, "key_z": 99}
```

最初の”key_q”:20 の組み合わせはなくなる。

配列の場合、順序は保証される。

```
jsonb=# SELECT
'{"key_x":[ 1, 400, 8888, 2, 99, 25, 3]}'::json,
'{"key_x":[ 1, 400, 8888, 2, 99, 25, 3]}'::jsonb ;
-[ RECORD 1 ]---------------------------------
 json  | {"key_x":[ 1, 400, 8888, 2, 99, 25, 3]}
 jsonb | {"key_x": [1, 400, 8888, 2, 99, 25, 3]}
```

JSONB 入力結果が保持されない例（空白の扱い）

キーまたは値でない JSON 文字列中の空白は、
JSONB では無視される。
また、 JSONB の検索結果として出力される JSON 文字列では
キーと値のセパレータである“ :” の後に 1 つ空白文字を入れる
という規則で文字列を生成している。

```
jsonb=# SELECT
'{"key_z"     :   99, "key q":    20, "key_a" : "aa  aa"}'::json,
'{"key_z"     :   99, "key q":    20, "key_a" : "aa  aa"}'::jsonb;
-[ RECORD 1 ]-------------------------------------------
 json  | {"key_z"     :   99, "key q":    20, "key_a" : "aa  aa"}
 jsonb | {"key q": 20, "key_a": "aa  aa", "key_z": 99}
```

JSONB 専用演算子

演算子	意味
<@	左辺の JSON が右辺のキー＆値の組を含むかを評価する。
@>	右辺の JSON が左辺のキー＆値の組を含むかを評価する。
?	左辺の JSON が右辺の（一つの）キーまたは要素を含むかどうかを評価する。
|?	左辺の JSON が右辺のキーまたは要素を 1 つでも含むかどうかを評価する。（ ANY 評価）
&?	左辺の JSON が右辺のキーまたは要素を全て含むかどうかを評価する。（ ALL 評価）

特に　<@　演算子は
GIN インデックスと
組み合わせると
非常に強力

GIN インデックス

汎用転置インデックス
配列型での利用
全文検索等でも利用

※ これも JSONB と同じ開発チームで開発している。
http://www.sai.msu.su/~megera/wiki/Gin

<@ と GIN インデックス

JSONB カラム自体に GIN インデックスを設定

```
jsonb=# SELECT data FROM jsonb_t LIMIT 1;
                                                                  data



------------------------------------------------------------------------------
--
----------------------------------------------------------
 {"Id": 0, "Email": "Laverna@junius.io", "Country": "Paraguay", "Full Name":
"Ca
rolyne Kohler", "Created At": "1987-08-21T18:42:02.269Z"}
(1 row)

jsonb=# CREATE INDEX jsonb_idx ON jsonb_t USING gin (data jsonb_ops);
CREATE INDEX
jsonb=# ¥d jsonb_t
                    Unlogged table "public.jsonb_t"
 Column |  Type   |                        Modifiers
--------+---------+------------------------------------------------------
 id     | integer | not null default nextval('jsonb_t_id_seq'::regclass)
 data   | jsonb   |
Indexes:
    "jsonb_idx" gin (data)
```

JSONB 型用の
GIN インデックス
メソッドの指定が必要

<@ と GIN インデックス

この状態で　@>　演算子（右辺にはキーと値の組み合わせ）
を使うと GIN インデックスによる効率的な検索が可能になる

JSONB カラムに <@ 演算子で条件を記述

```
jsonb=# EXPLAIN SELECT data->'Full Name' as fullname, data->'Country' as
country FROM jsonb_t WHERE data @> '{"Country" : "Monaco"}';
                                QUERY PLAN
---------------------------------------------------------------------------
 Bitmap Heap Scan on jsonb_t  (cost=20.08..54.18 rows=10 width=160)
   Recheck Cond: (data @> '{"Country": "Monaco"}'::jsonb)
   ->  Bitmap Index Scan on jsonb_idx  (cost=0.00..20.07 rows=10 width=0)
         Index Cond: (data @> '{"Country": "Monaco"}'::jsonb)
 Planning time: 0.074 ms
(5 rows)
```

<@ と GIN インデックス

JSONB カラムに <@ 演算子で条件を記述（検索結果）

```
jsonb=# SELECT data->'Full Name' as fullname, data->'Country' as country FROM
jsonb_t WHERE data @> '{"Country" : "Monaco"}';
         fullname          | country
---------------------------+----------
 "Romaine Hickle"          | "Monaco"
 "Mr. Joana Huel"          | "Monaco"
 "Jeff Wilkinson"          | "Monaco"
 (以下略)
```

"Country" が "Monaco" の行だけ選択されている。

今までの JSON 型でも
"->"，"->>" 演算子と
関数インデクスの組み合わせで
Btree インデクスによる
インデクス検索は出来たが
@> ＋ GIN の組み合わせは
キーが不定でも有効！

しかも、 PostgreSQL 9.4 では GIN インデックスのサイズ縮小 / 性能改善も行われているのでさらに効果的！

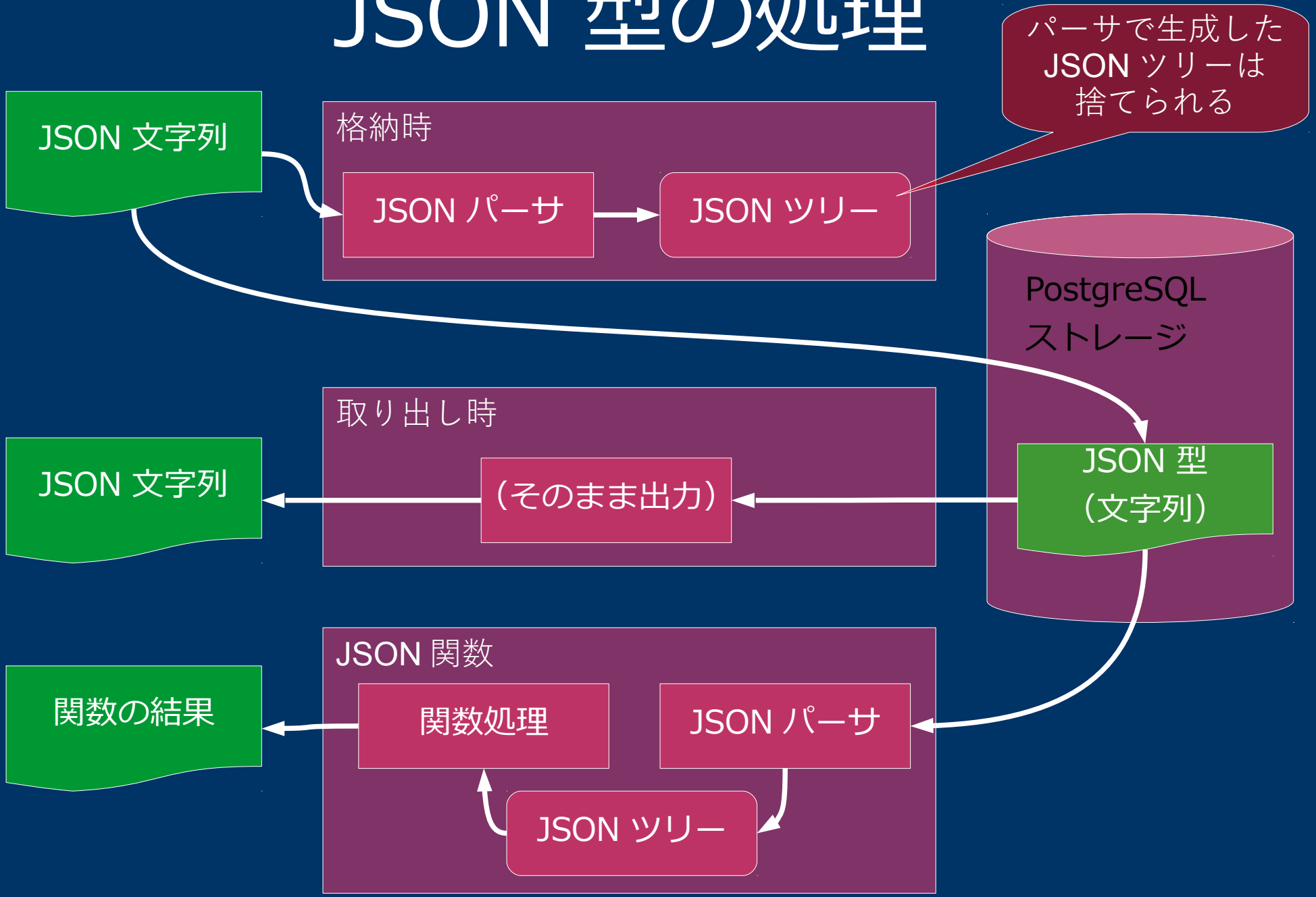
（参考： GIN インデックスの改善も JSONB 型と同じメンバが実施している）

JSONB 型の実装

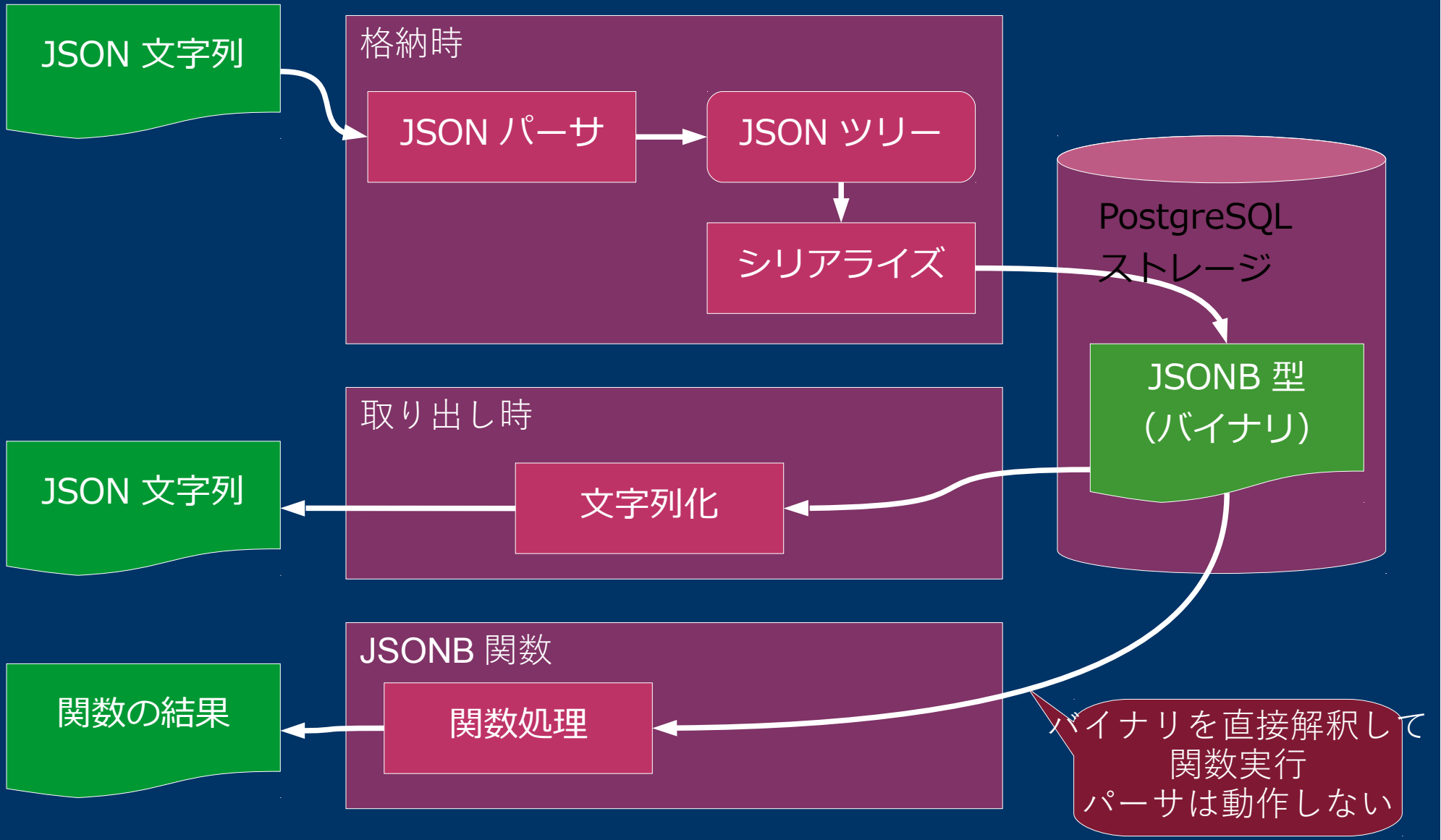
JSON 型と JSONB 型の処理

JSON 型と JSONB 型の
データ格納
データ取得
JSON 関数処理
の処理イメージを次ページ以降に示す。

JSON 型の処理

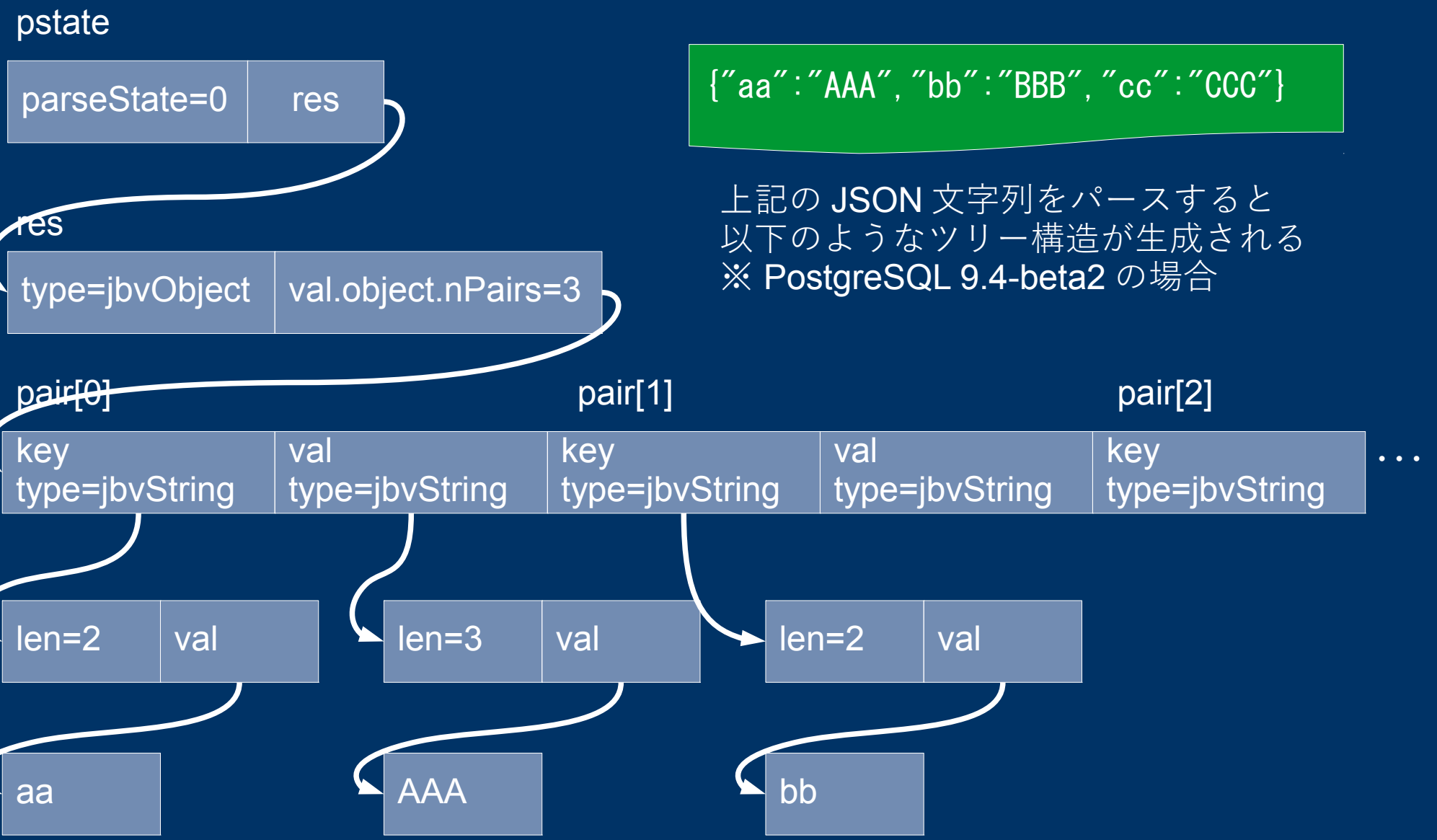

JSON 文字列
格納時
JSON パーサ
JSON ツリー
パーサで生成した
JSON ツリーは
捨てられる
PostgreSQL
ストレージ
取り出し時
JSON 文字列
（そのまま出力）
JSON 型
（文字列）
JSON 関数
関数の結果
関数処理
JSON パーサ
JSON ツリー

JSONB 型の処理


JSON 文字列
格納時
JSON パーサ
JSON ツリー
シリアライズ
PostgreSQL
ストレージ
JSONB 型
(バイナリ)
取り出し時
JSON 文字列
文字列化
JSONB 関数
関数の結果
関数処理
バイナリを直接解釈して
関数実行
パーサは動作しない

beta-2 調査時

JSON ツリー



格納サイズ

さっき説明した JSON と JSONB の
処理内容から格納サイズを予想すると
以下のようになる。

格納サイズは JSON のほうが小さい？
（ JSONB はツリー構造情報も保持）

以下のような JSON データ

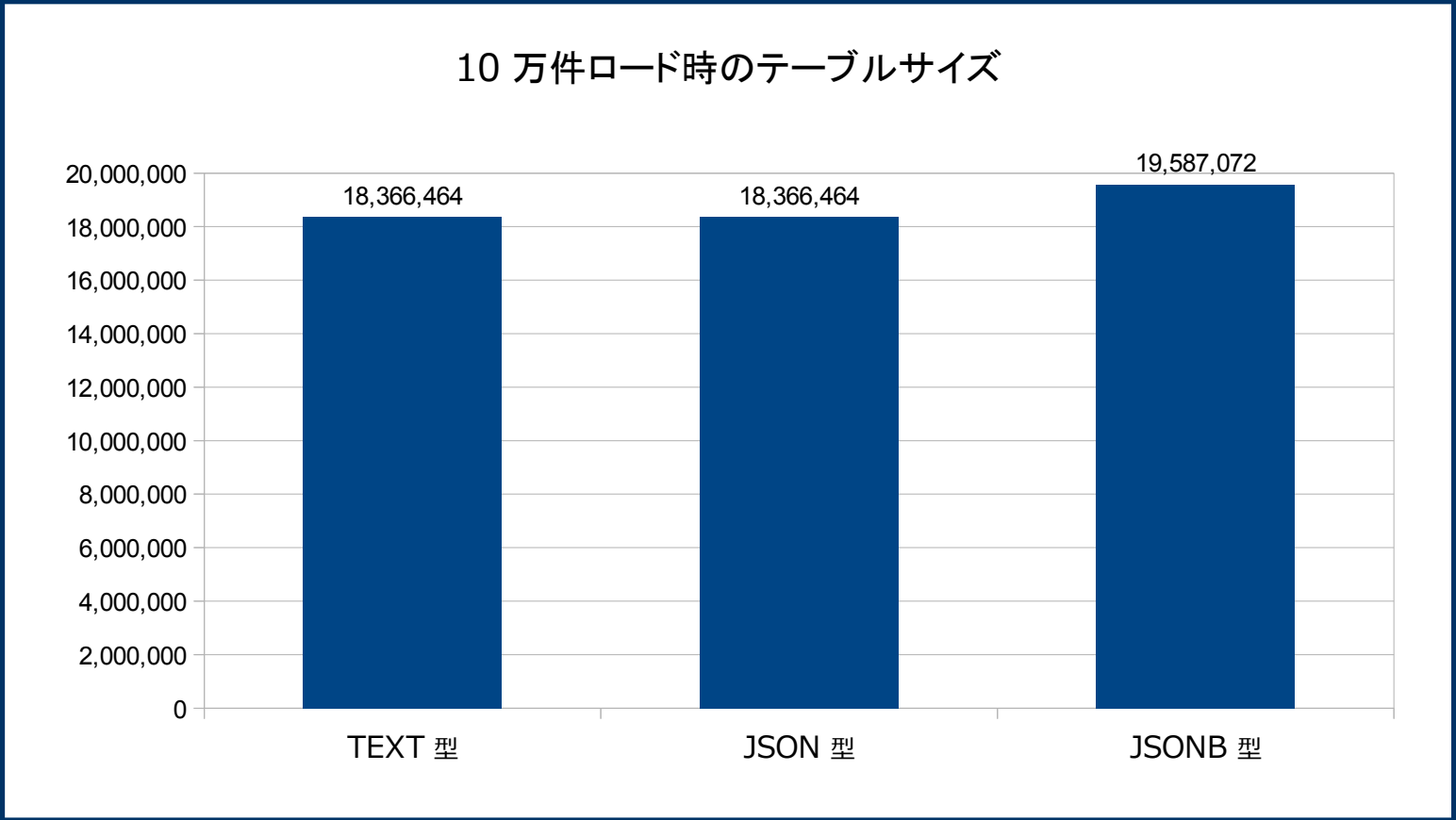
{ "Email": "Laverna@junius.io", "Created At": "1987-08-21T18:42:02.269Z", "Country": "Paraguay","Id":"0", "Full Name": "Carolyne Kohler"}

測定内容

TEXT/JSON/JSONB カラムのみをもつテーブルを作成
上記 1 万件を 10 回 COPY した後の
テーブルサイズを pg_relation_size() で測定

格納サイズ

RC1 で確認



TEXT 型と JSON 型のサイズは同じ

JSONB は JSON 型と比較すると 7 %増加

インデックスサイズ

JSON 型のキーに対する
btree 関数インデックス
Vs
JSONB の GIN インデックス

さっきの JSON 型データの 5 種のキーに
対する btree インデックスの総サイズと
JSONB 型の GIN インデックスサイズを比較してみる。

インデックスサイズ

各キーから値を取得する btree インデックス群を設定

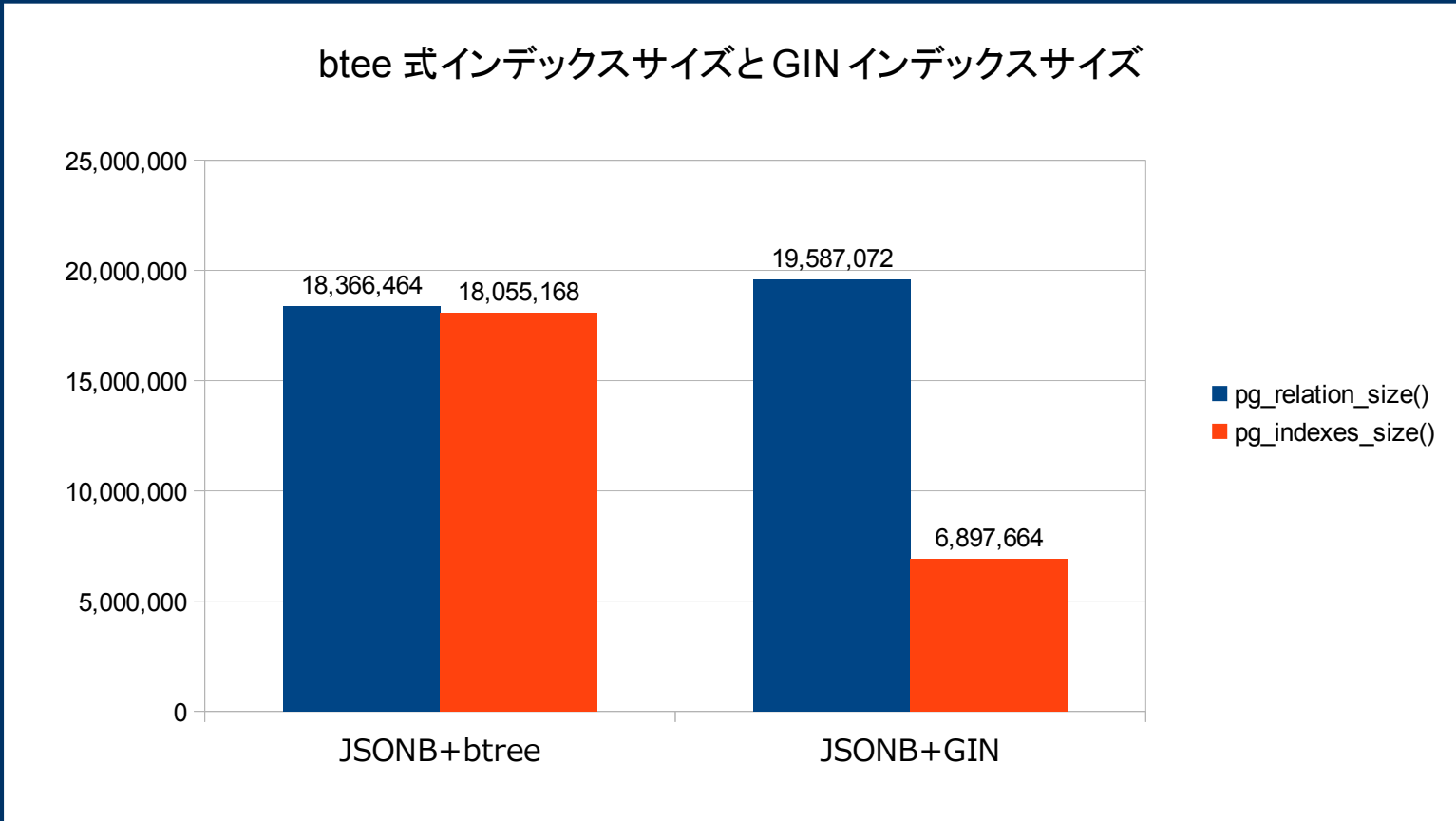
```
CREATE INDEX jsonb_id_idx ON jsonb_t
  USING btree ((data->>'Id'));

CREATE INDEX jsonb_fullname_idx ON jsonb_t
  USING btree ((data->>'Full Name'));

CREATE INDEX jsonb_email_idx ON jsonb_t
  USING btree ((data->>'Email'));

CREATE INDEX jsonb_created_idx ON jsonb_t
  USING btree ((data->>'Created At'));

CREATE INDEX jsonb_country_idx ON jsonb_t
  USING btree ((data->>'Country'));
```

JSONB 列自体に GIN インデックスを設定

```
CREATE INDEX jsonb_data_idx ON jsonb_t USING gin (data jsonb_ops);
```

インデックスサイズ

RC1 で確認



全てのキーに btree インデックスを
設定するより GIN インデックスのほうが
インデックスサイズは大幅に小さくなる。

性能比較

さっき説明した JSON と JSONB の
処理内容から処理性能を予想すると
以下のようになる。

格納処理は JSON のほうが高速
単なる SELECT は JSON のほうが高速？
JSON 関数を使った場合は JSONB が高速

測定環境

PC:Let's note SX2(SSD)
OS:CentOS 6.3(メモリ 4GB)

PostgreSQL パラメータ
shared_buffers=128MB
checkpoint_segments=30

以下のような JSON データ

{ "Email": "Laverna@junius.io", "Created At": "1987-08-21T18:42:02.269Z", "Country": "Paraguay", "Id": 0, "Full Name": "Carolyne Kohler"}

測定内容

JSON/JSONB カラムをもつテーブルを UNLOGGED で作成

上記 1 万件を 10 回 COPY したときの平均性能

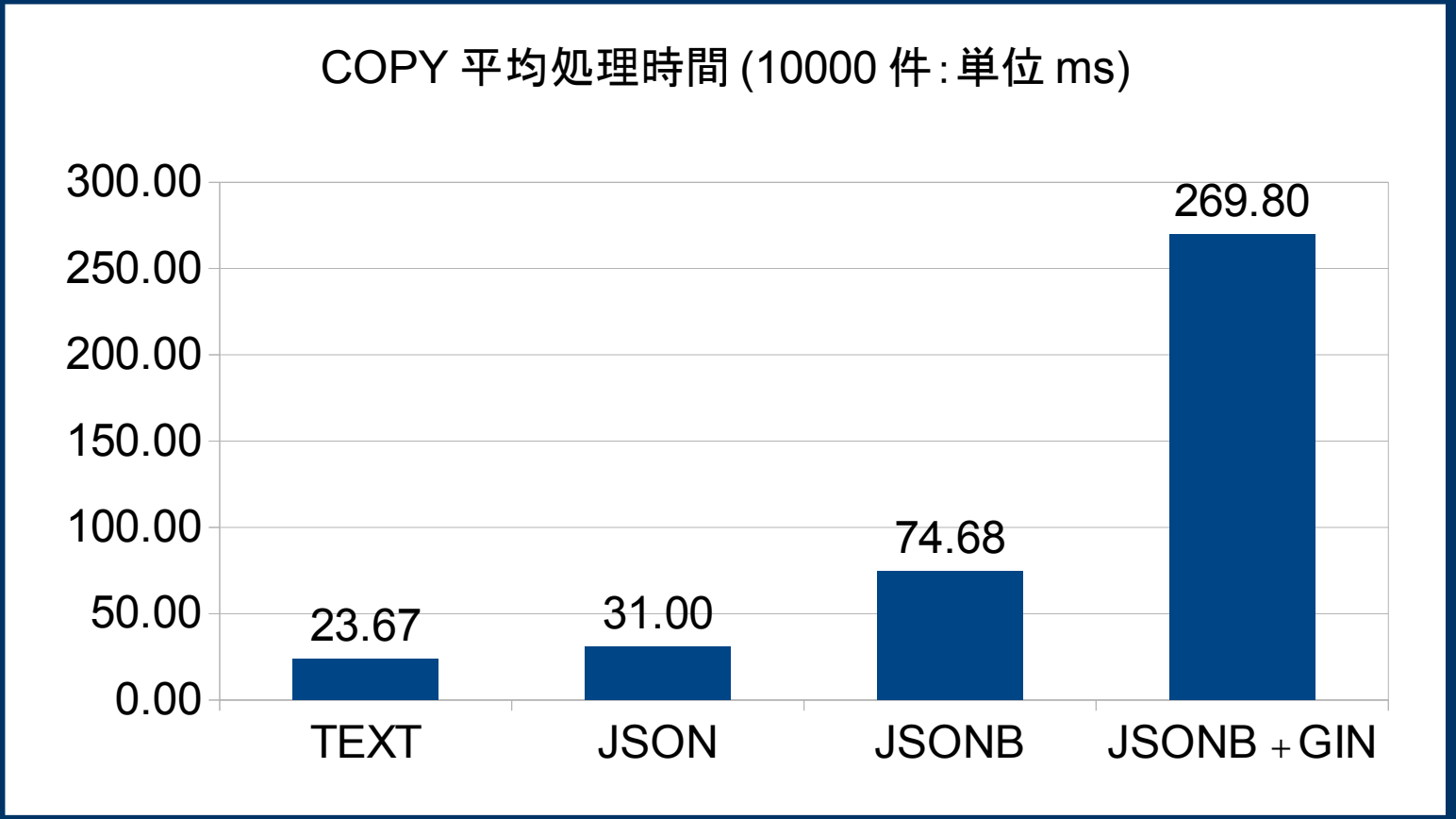
SELECT(10 万件) の平均性能

SELECT(10 万件) + JSON 演算子を使ったときの平均性能

btree vs GIN インデックス検索性能

COPY 処理時間

RC1 で測定



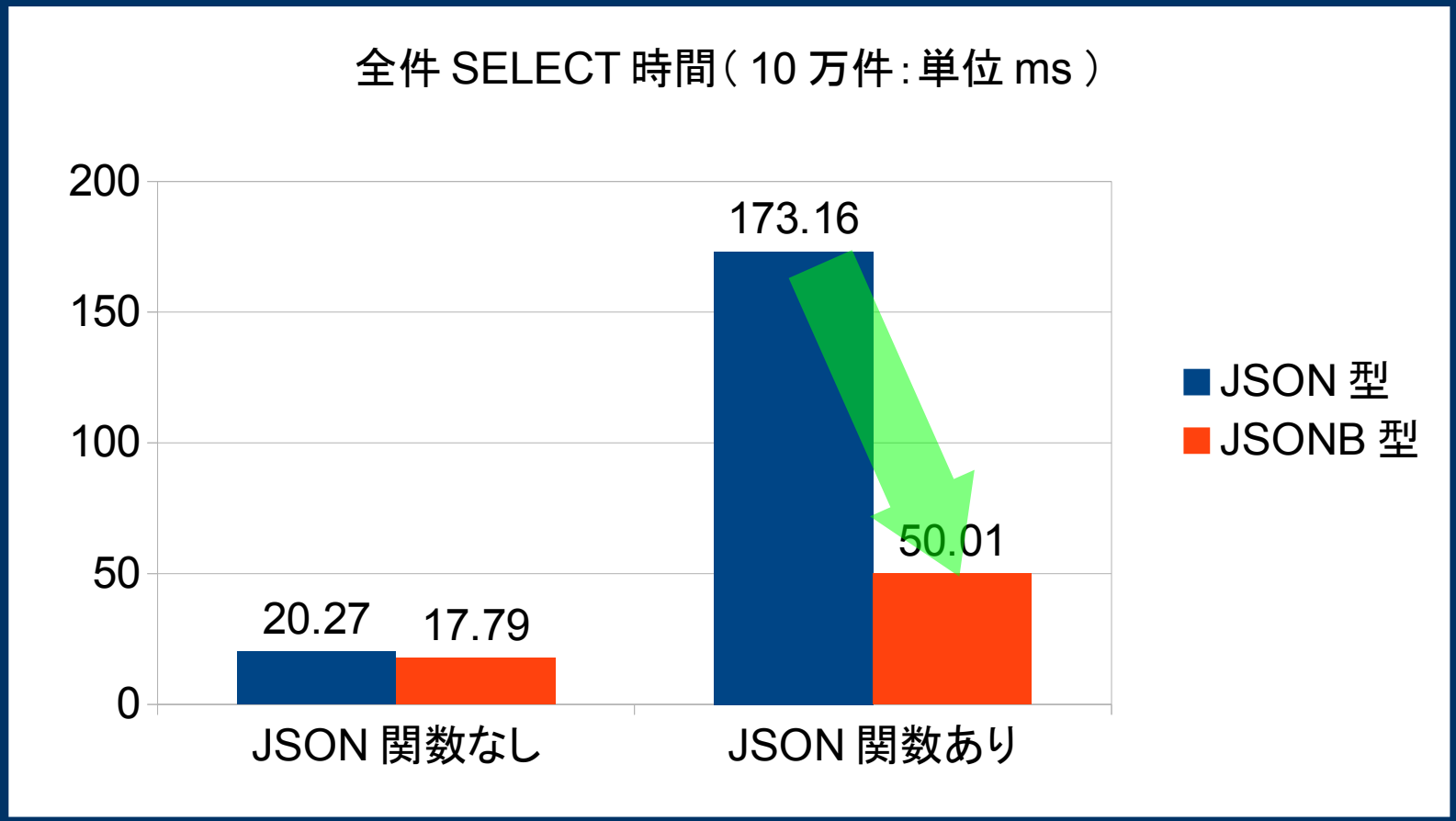
TEXT と JSON の差分が 10000 件分の JSON パース時間？

JSON と JSONB の差分が 10000 件文のシリアライズ時間？

GIN インデックスありの場合はかなり遅くなる。

SELECT 処理時間

RC1 で確認



単なる SELECT はほとんど差はない？

JSON 関数を使用した場合の性能差は明確。

btree vs GIN

btree 式インデックス使用時

```
EXPLAIN ANALYZE SELECT * FROM jsonb_t WHERE data->>'Country' = 'Monaco';
 Bitmap Heap Scan on jsonb_t  (cost=12.29..1240.26 rows=500 width=161) (actual
time=0.180..0.692 rows=310 loops=1)
   Recheck Cond: ((data ->> 'Country'::text) = 'Monaco'::text)
   Heap Blocks: exact=280
   ->  Bitmap Index Scan on jsonb_country_idx  (cost=0.00..12.17 rows=500 width=0)
(actual time=0.112..0.112 rows=310 loops=1)
         Index Cond: ((data ->> 'Country'::text) = 'Monaco'::text)
 Planning time: 0.607 ms
 Execution time: 0.767 ms
(7 rows)
```

GIN インデックス使用時

```
EXPLAIN ANALYZE SELECT * FROM jsonb_t WHERE data @> '{"Country":"Monaco"}';
 Bitmap Heap Scan on jsonb_t  (cost=28.77..362.50 rows=100 width=161) (actual
time=1.709..2.502 rows=310 loops=1)
   Recheck Cond: (data @> '{"Country": "Monaco"}'::jsonb)
   Heap Blocks: exact=280
   ->  Bitmap Index Scan on jsonb_data_idx  (cost=0.00..28.75 rows=100 width=0) (actual
time=1.630..1.630 rows=310 loops=1)
         Index Cond: (data @> '{"Country": "Monaco"}'::jsonb)
 Planning time: 0.086 ms
 Execution time: 2.566 ms
(7 rows)
```

GIN より btree 式インデックスのほうが若干速い

測定結果まとめ

格納： JSON が高速

SELECT ：ほぼ同じ？

JSON 関数： JSONB が高速

GIN より btree 式インデックスが高速

結局、 JSON と JSONB はどう使い分けるのか？

格納効率優先なら JSON

検索効率優先なら JSONB

入力形式を保持するなら JSON

⇒ だいたいの場合、 JSONB で OK?

【余談】 jsquery

PostgreSQL 9.4 には取り込まれてませんが、 JSONB 型に対して更に強力なクエリやインデックスを提供する jsquery という外部プロジェクトもあります。

https://github.com/akorotkov/jsquery

PostgreSQL 9.5 に入るかも？

JSONB を
アプリケーションから
使ってみる

ここまでは
SQL レベルでの
JSON/JSONB の
使い方を説明しました

しかし、実際には
何らかの
アプリケーションから
使うことになるはず

C 言語： libpq

Java ： JDBC

その他の LL/FW…

今回は基本となる

libpq と JDBC を説明します

libpq 経由で JSON を使う。

といっても
libpq で組む場合、
JSONB だからといって
何か特殊な API を使うわけではない。

JSON/JSONB データの挿入 / 更新 / 削除は普通に PQexec や PQexecParams を使う

JSON/JSONB データは
文字列として渡す。
（SQL べた書きでも
パラメータ渡しでも）

パラメータ渡しで INSERT する例

```
char* values[3] = {
  "{¥"id¥":1, ¥"name¥": {¥"first¥": ¥"Oleg¥"}, ¥"distribute¥":
[¥"GIN¥", ¥"hstore¥", ¥"json¥", ¥"jsonb¥"]}",
  "{¥"id¥":2, ¥"age¥": 59, ¥"name¥": {¥"last¥": ¥"Lane¥", ¥"first¥": ¥"Tom¥"}}",
  "{¥"id¥":3, ¥"name¥": {¥"nickname¥": ¥"nuko¥"}, ¥"distribute¥":
[¥"ksj¥", ¥"neo4jfdw¥"]}"
};

int
insert_table(PGconn* conn) {

  int i = 0;
  PGresult* res;
  res = PQexec(conn, "BEGIN");

  for (i=0; i < 3; i++) {
    res = PQexecParams(conn,
      "INSERT INTO test VALUES($1)",
      1,
      NULL,
      (const char* const*) &values[i],
      NULL,
      NULL,
      0);

    if (PQresultStatus(res) != PGRES_COMMAND_OK) {
      fprintf(stderr, "Truncate error, %s¥n", PQerrorMessage(conn));
      exit(-1);
    }
  }
 (省略)
```

検索結果の受け取りで注意すること。

TEXT 型であれ
JSON 型であれ、
JSONB 型であれ、
libpq 経由で取り出した
値は文字列になる。

JSONB 型自体には
数値データ型があるが
libpq 経由で取り出した
値は文字列になる。
数値評価時には型変換要

検索結果の処理コードの例

```
  printf("sql=%s¥n", sql);
  res = PQexec(conn, sql);

(略)

  for (i=0; i < tuples; i++) {
    for (j=0; j < fields; j++) {
      // nullかどうか判断する
      if (PQgetisnull(res, i, j) == 1) {
        // null
        printf("data(%d, %d) is null¥n",  i, j );
      } else {
        // not null
        // 結果の長さを取得
        printf("length(%d, %d) = %d¥n", i, j, (int) PQgetlength(res, i, j));
        // 結果の取得
        printf("data(%d, %d) = %s¥n",  i, j, PQgetvalue(res, i, j));
      }
    }
  }
```

実行例

```
sql=SELECT data #>> '{name,first}' as name, data->>'age' FROM test
ntuples = 3
fields = 2
name = name
type oid = 25
internal size= -1
name = ?column?
type oid = 25
internal size= -1
length(0, 0) = 4
data(0, 0) = Oleg
data(0, 1) is null
length(1, 0) = 3
data(1, 0) = Tom
length(1, 1) = 2
data(1, 1) = 59
data(2, 0) is null
data(2, 1) is null
```

59 という数値は、 59 という数値文字列として取得される。

なお、 libpq には結果の型を
取得する PQftype 関数がある。
取得結果は型の oid 。

oid の値	対応するデータ型
25	TEXT
114	JSON
3802	JSONB

結果の型が TEXT なのか JSON/JSONB なのかで
処理を変えたい場合に使えるかも。

キーが存在しない場合は
NULL となるが、
getValue() では
空文字列が返却される。
NULL と区別できない。

キーが確実に存在していることが不明の場合、Pqgetisnull() 関数で NULL かどうか要確認

検索結果の処理コードの例

```
  printf("sql=%s¥n", sql);
  res = PQexec(conn, sql);

(略)

  for (i=0; i < tuples; i++) {
    for (j=0; j < fields; j++) {
      // nullかどうか判断
      if (PQgetisnull(res, i, j) == 1) {
        // null
        printf("data(%d, %d) is null¥n",  i, j );
      } else {
        // not null
        // 結果の長さを取得
        printf("length(%d, %d) = %d¥n", i, j, (int) PQgetlength(res, i, j));
        // 結果の取得
        printf("data(%d, %d) = %s¥n",  i, j, PQgetvalue(res, i, j));
      }
    }
  }
```

実行例

```
sql=SELECT data #>> '{name,first}' as name, data->>'age' FROM test
ntuples = 3
fields = 2
name = name
type oid = 25
internal size= -1
name = ?column?
type oid = 25
internal size= -1
length(0, 0) = 4
data(0, 0) = Oleg
data(0, 1) is null
length(1, 0) = 3
data(1, 0) = Tom
length(1, 1) = 2
data(1, 1) = 59
data(2, 0) is null
data(2, 1) is null
```

キーにヒットしないものは null になる。

私見では C で処理する場合には、なるべく SQL で処理して、結果は文字で扱うほうが良い？

C 言語にも JSON プロセッサはあるが・・・

JDBC 経由で JSON を使う。

といっても
JDBC 経由で JSONB に
アクセスする場合も、
基本的には普通の型と
変わらない。

キーが確実に存在していることが不明の場合、wasNull() メソッドで NULL かどうか要確認

値の取得は getString() メソッドを使う。

getJson() のような
メソッドは追加されていない。

単純な検索の例

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(query);

(略)

while (rs.next()) {
  for (int i=1; i <= cols; i++) {
     String s = rs.getString(i);
     if (rs.wasNull()) {
        System.out.print("(null)" + ",");
     } else {
        System.out.print(s + ",");
     }
  }
  System.out.println("");
}
```

注意が必要なのは
パラメータとして
JSON/JSONB を渡す時

JSON 文字列を setString で渡すと…

```
Connection conn =
DriverManager.getConnection(url,"postgres","postgres");

PreparedStatement ps =
conn.prepareStatement("INSERT INTO jsonb_t VALUES(?)");
// JSON文字列を設定
ps.setString(1, json);

// 検索実行
int i = ps.executeUpdate();
```

```
Exception in thread "main" org.postgresql.util.PSQLException: ERROR:
column "data" is of type jsonb but expression is of type character varying
  ヒント: You will need to rewrite or cast the expression.
```

※setString() で JSON 文字列をセットすと、
executeUpdate() 背景でエラーになる・・・

JSON 文字列をどうやって渡す？（正道）

```
Connection conn =
DriverManager.getConnection(url,"postgres","postgres");

PreparedStatement ps =
conn.prepareStatement("INSERT INTO jsonb_t VALUES(?)");
// JSON文字列を設定
org.postgresql.util.PGobject pgo =
  new org.postgresql.util.PGobject();
pgo.setValue(json);
pgo.setType("jsonb");
ps.setObject(1, pgo);

// 検索実行
int i = ps.executeUpdate();
```

※PotgreSQL JDBC Driver の PGObject を作成し、そこに
setValue() で JSON テキストを、
また setType() で”jsonb”を指定する必要がある。

JSON 文字列をどうやって渡す？（邪道）

```
Connection conn =
DriverManager.getConnection(url,"postgres","postgres");

PreparedStatement ps =
conn.prepareStatement("INSERT INTO jsonb_t VALUES(?)");
// JSON文字列を設定
ps.setObject(1, json, 1111); // 1111：JSONB型を示す数値

// 検索実行
int i = ps.executeUpdate();
```

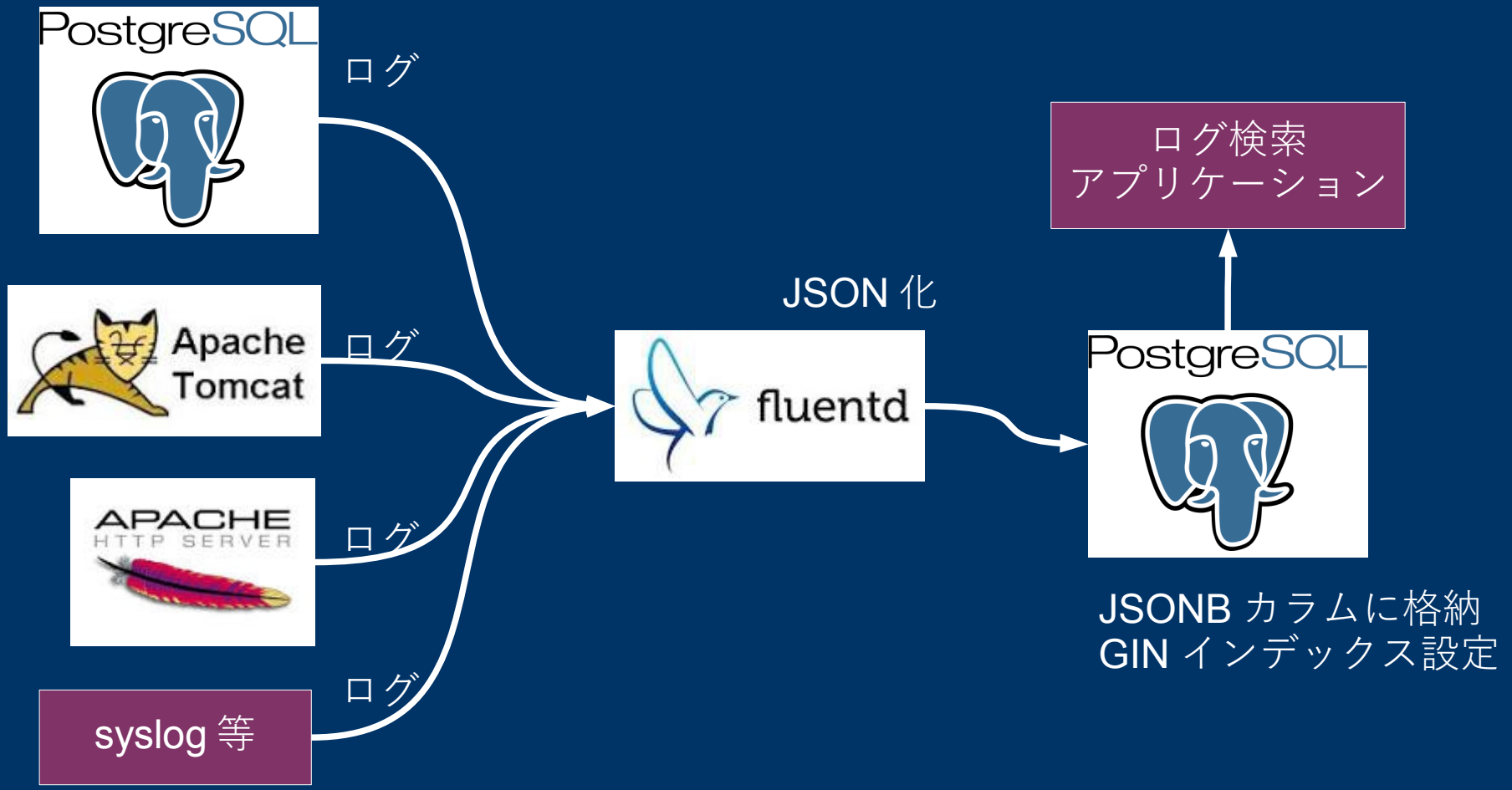
※setString() ではなく、 setObject() に JSON 文字列と、
　JSON/JSONB 型を示す型値を第 3 パラメータに設定する。

※1111 という数値は検索結果のメタデータから
　getColumnType() で取得したもの。
　将来的にこの値のままかどうかは不明・・・。

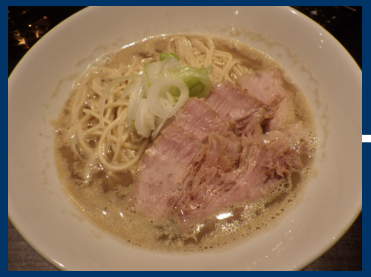
本当は RoR 等の
Or マッパをもつ
フレームワークで
どうやって JSON/JSONB を
使うのか書きたかったが
今回は時間切れ・・・
すいません・・・

JSON 型の想定用途

様々なログの収集
（ fluentd との組み合わせ等）



様々な食べ物のレビューサイト



{"タイプ":"ラーメン",
"麺":"中細麺",
"スープ":"醤油",
"具":["チャーシュー"、"葱"],
"スコア":85}

{"タイプ":"ざる",
"具":["鴨","葱"],
"スコア":75}

{"タイプ":"焼き",
"皮":"厚め"
"餡":["肉","大葉"],
"スコア":70}

{"タイプ":"欧風カレー",
"主食":"ご飯",
"具":["鴨","葱"],
"スコア":75}

レビュー検索
アプリケーション

PostgreSQL

JSONB カラムに格納
GIN インデックス設定

微妙なテーブル設計の救済？

```
CREATE TABLE foo {
  id integer primary key,
  name text,
 ・・・
  memo1 text,
  memo2 text,
  memo3 text,
 ・・・
  memo100 text
}
```

可変の個数
設定されるような
カラム群

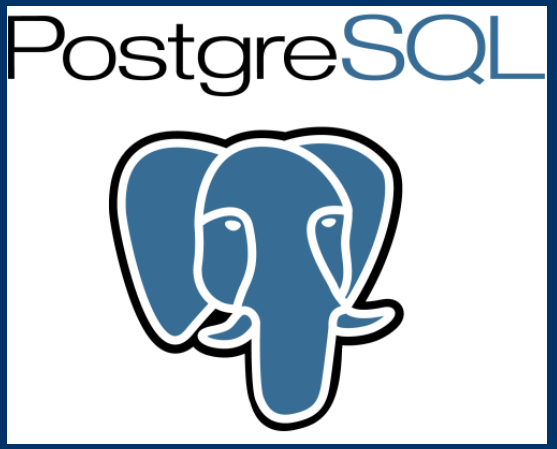
```
CREATE TABLE foo {
  id integer primary key,
  name text,
 ・・・
  memo jsonb
}
```

JSONB 型に
まとめてしまう

※ きちんと設計をしていれば不要だろうけど・・・

MongoDB 等の NoSQL ドキュメント DB の代替？

or



どの程度の量の JSON データを扱うか次第かも？

例えば数十 GB 程度、
スケールアウトしなくてもいい程度なら
PostgreSQL の JSONB は
十分代替候補になりそう？

既存の PostgreSQL 環境に
新たに JSONB 型を
追加することで
SQL レベルで統一して
扱えるのは強みになるかも？

PostgreSQL と NoSQL

適用領域次第では
PostgreSQL JSON でも OK

	得意	不得意
PostgreSQL	トランザクション SQL との連携 (結合など)	スケールアウト 並列処理 集計処理
NoSQL （ MongoDB 等）	スケールアウト 並列処理 集計処理	トランザクション 結合

【参考】
類似データ型比較

PostgreSQL には JSONB 以外
にも半構造化データを
管理するデータ型がある。

⇒XML 型，hstore 型

XML 型

XML を格納するデータ型

属性、階層、順序、名前空間対応

本体機能 (configure で指定)

格納時に XML パースを行なう

型自体に比較演算機能はない

xpath によるアクセスが可能

libxml2 ライブラリに依存

XML 型の使用例

XML 型カラムを持つテーブル

```
other=# ¥d xml_t
                    Table "public.xml_t"
 Column |  Type   |                      Modifiers
--------+---------+-----------------------------------------------
 id     | integer | not null default nextval('xml_t_id_seq'::regclass)
 data   | xml     |

other=# SELECT data FROM xml_t;
 <rdb_t><email>Laverna@junius.io</email><created_at>1987-08-
21T18:42:02.269Z</created_at><country>
Paraguay</country><id>0</id><full_name>Carolyne Kohler</full_name></rdb_t>
 <rdb_t><email>Nakia_Rolfson@cecelia.ca</email><created_at>1989-03-
16T14:37:36.825Z</created_at><c
ountry>France</country><id>1</id><full_name>Paul Weber DVM</full_name></rdb_t>
 <rdb_t><email>Elbert@norma.co.uk</email><created_at>1980-02-
19T04:16:52.113Z</created_at><country
>Uzbekistan</country><id>2</id><full_name>Florence Murphy</full_name></rdb_t>
(3 rows)
```

XML 型の使用例

xpath 関数による XML 文書からの抽出

```
other=# SELECT
(xpath('/rdb_t/email/text()', data)::text[])[1] as email,
(xpath('/rdb_t/full_name/text()', data)::text[])[1] as fullname
FROM xml_t;
          email             |    fullname
----------------------------+-----------------
 Laverna@junius.io          | Carolyne Kohler
 Nakia_Rolfson@cecelia.ca   | Paul Weber DVM
 Elbert@norma.co.uk         | Florence Murphy
(3 rows)
```

XML や xpath 関数は強力だけど、書くのは少々面倒・・・

（例） xpath 関数は XML 配列を返却するので、テキストを取り出す場合に、 TEXT 配列にキャストして最初の要素を取り出すなどの操作が必要。

（例）名前空間 url と prefix の組を配列化して xpath 関数に渡す必要がある。

hstore

Key-Value store データ型
Contrib モジュール
ネストはできない。
部分更新インタフェースあり
PostgreSQL 8.3 ～対応
※ 作者は JSONB と同じ Oleg 氏

hstore の使用例

hstore 型カラムを持つテーブル

```
other=# ¥dx
                                 List of installed extensions
  Name   | Version |   Schema   |                      Description
---------+---------+------------+------------------------------------------------------
 hstore  | 1.3     | public     | data type for storing sets of (key, value) pairs
 plpgsql | 1.0     | pg_catalog | PL/pgSQL procedural language
(2 rows)

other=# ¥d hstore_t
                        Table "public.hstore_t"
 Column |  Type   |                        Modifiers
--------+---------+---------------------------------------------------------
 id     | integer | not null default nextval('hstore_t_id_seq'::regclass)
 data   | hstore  |
```

hstore の使用例

hstore 型カラムの検索

```
other=# SELECT data FROM hstore_t;
                                                                         data


--------------------------------------------------------------------------------
--------------------
--------------------------------------------
 "id"=>"0", "email"=>"Laverna@junius.io", "country"=>"Paraguay",
"full_name"=>"Carolyne Kohler", "
created_at"=>"1987-08-21T18:42:02.269Z"
 "id"=>"1", "email"=>"Nakia_Rolfson@cecelia.ca", "country"=>"France",
"full_name"=>"Paul Weber DVM
", "created_at"=>"1989-03-16T14:37:36.825Z"
 "id"=>"2", "email"=>"Elbert@norma.co.uk", "country"=>"Uzbekistan",
"full_name"=>"Florence Murphy"
, "created_at"=>"1980-02-19T04:16:52.113Z"
(3 rows)
```

hstore の使用例

hstore 型カラムから特定のキーのみ抽出

```
other=# SELECT data->'email' as email, data->'full_name' as full_name FROM
hstore_t;
          email            |    full_name
---------------------------+-----------------
 Laverna@junius.io         | Carolyne Kohler
 Nakia_Rolfson@cecelia.ca  | Paul Weber DVM
 Elbert@norma.co.uk        | Florence Murphy
(3 rows)
```

条件式を使う例

```
other=# SELECT data->'id' as id, data->'email' as email FROM hstore_t WHERE
data->'id' = '1';
 id |          email
----+-------------------------
 1  | Nakia_Rolfson@cecelia.ca
(1 row)
```

XML 型 /xpath 関数を使うよりシンプルに書ける

hstore に関する情報

hstore については、 JPUG の「第 20 回しくみ + アプリケーション勉強会 (2011 年 6 月 4 日) 」でも解説しているので、そちらも参考にしてください。

http://www.postgresql.jp/wg/shikumi/sikumi_20/
http://www.postgresql.jp/wg/shikumi/study20_materials/hstore_introduction/view

XML, hstore, JSON 比較

データ型	表現能力	格納領域	処理性能
XML	◎	△	△
hstore	△	○	◎
JSON	○	○	○
JSONB	○	○	◎

XML は表現能力は高いが扱いにくい

hstore はシンプルだが速い

JSONB は良いとこ取り？

⇒ 用途で使い分け可能

FAQ

Q. JSONB って日本語扱える？

A. キーにも値にも日本語は使えます。
ただし、サーバエンコーディングが
UTF-8 環境で使うのが無難そう。

日本語キーのパスを使って日本語を取得する例

```
jsonb=# SELECT '{"種別":"ラーメン", "値段" : 650, "スープ":"豚骨醤油", "
トッピング":["チャーシュー","ほうれん草","海苔","葱"]}'::jsonb #> '{トッピ
ング, 2}';
 ?column?
----------
 "海苔"
(1 row)
```

Q. JSONB ってトランザクションに対応しているの？

A. してます。 JSONB も PostgreSQL 上では他のデータ型と一緒の扱いです。

Q. PostgreSQL の
クライアントライブラリでは JSONB を
オブジェクトとして扱えるの？

A. すいません、きちんと調べてません。
少なくとも libpq としては特別な扱いは
してないみたいです。

Q. フレームワーク環境で使える？

A. JSON/JSONB 演算子は特殊なので
フレームワークで自動生成された
SQL はそのまま使えないかも。

※ フレームワークとの相性が
JSON/JSONB 型普及の妨げ？

Q. 結局、どういうときに
JSONB 型を使えばいいの？

A. 開発時にスキーマが決定できない情報を管理する場合。
（ XML や hstore も検討してみる）
あるいは、外部データとして JSON を使っていて、それを PostgreSQL で管理したい場合とか・・・？

Q. JSON データ管理なら
MongoDB とかじゃダメなの？

A. それも一つの選択肢です。
スケールアウト重視なら MongoDB 。
スタートアップ開発にも向いている。

トランザクションなど堅牢性の重視や
SQL アクセスしたいなら PostgreSQL 。

参考情報

- “PostgreSQL 9.4beta2 Documentation”
  http://www.postgresql.org/docs/9.4/static/index.html
- “Schema-less PostgreSQL”
  http://www.sraoss.co.jp/event_seminar/2014/20140911_pg94_schemaless.pdf
- SRA OSS. “PostgreSQL 9.4　評価検証報告”
  http://www.sraoss.co.jp/event_seminar/2014/20140911_pg94report.pdf
- 第 20 回しくみ+アプリケーション勉強会
  "PostgreSQL の KVS hstore の紹介”
  http://www.postgresql.jp/wg/shikumi/study20_materials/hstore_introduction/view
- EDB 社ブログ “ Open Enterprise: The PostgreSQL Open Source Database Blog from EnterpriseDB”
  http://blogs.enterprisedb.com/2014/09/24/postgres-outperforms-mongodb-and-ushers-in-new-developer-reality/
- ”7 つのデータベース　7 つの世界”
  ISBN 978-274-06907-6

ご清聴

ありがとう

ございました