PostgreSQL

A Tour of PostgreSQL Internals

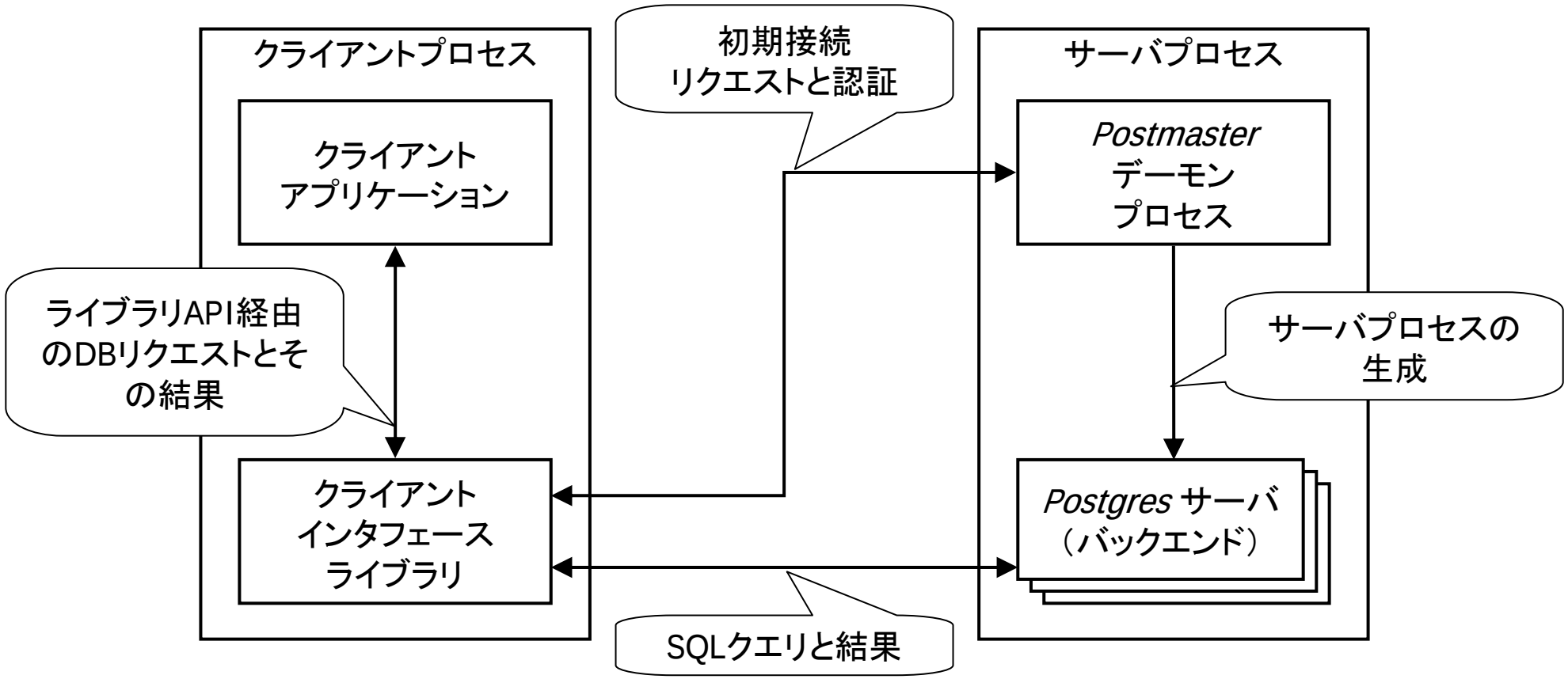
Tom Lane 著

寺本 純司 訳

Outline

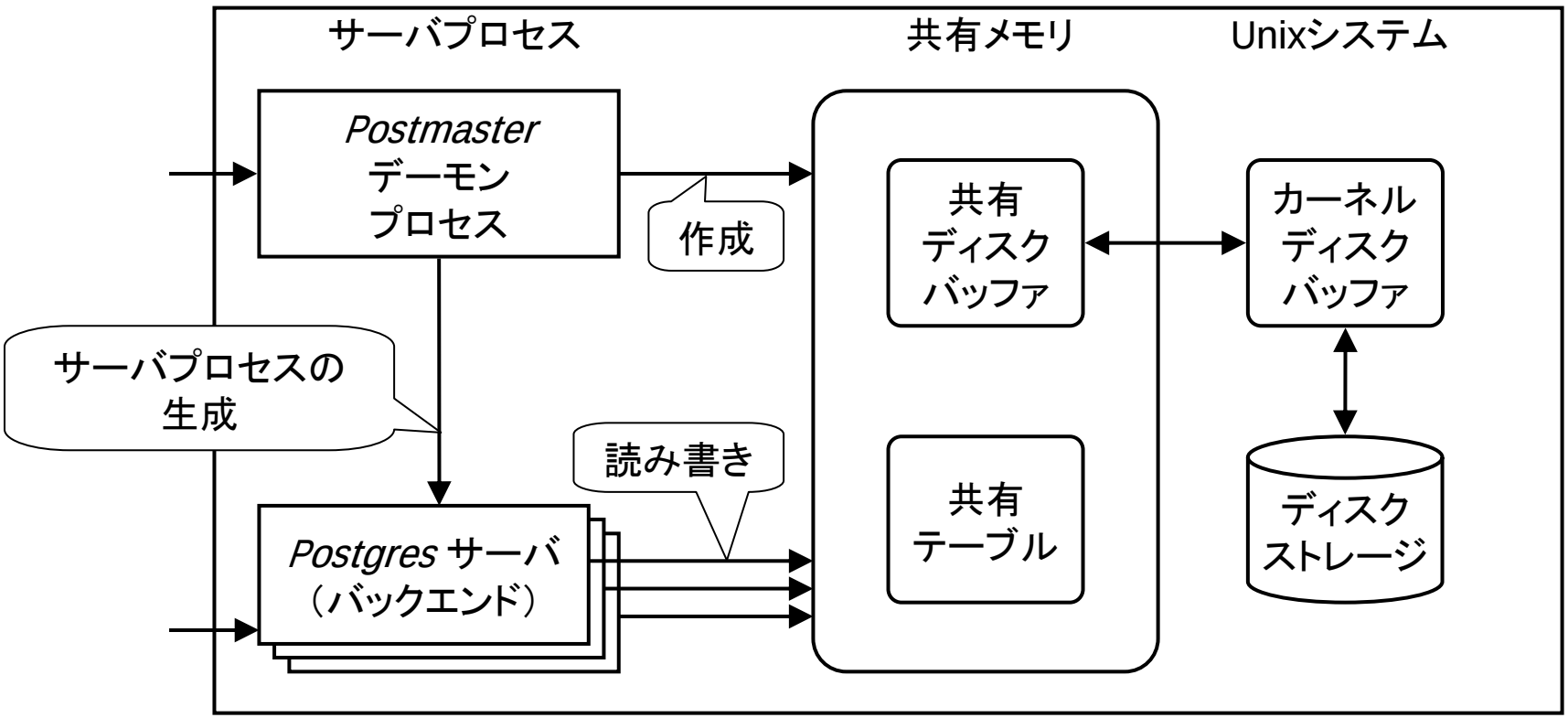
- PostgreSQLの3つの概観を提示
  - 各概観は等しく有効だが、野獣（実際のPostgreSQL）とは少し違っている
- 概観1:プロセスとプロセス間通信の構造
  - 重要なアイデア：クライアントサーバ分離、サーバ間通信
- 概観2:システムカタログとデータタイプ
  - 重要なアイデア：拡張性、区画化
- 概観3:クエリ処理のステップ
  - 重要なアイデア：パース木とプラン木、探索と結合、コストの概算

PostgreSQL processes: client/server communication



- 複数のクライアントライブラリが各種APIを提供する(libpq, ODBC, JDBC, PerlDBD)
- クライアントライブラリは、回線上のプロトコルの変更からクライアントアプリケーションを分離する

PostgreSQL processes: inter-server communication



PostgreSQL processes: summary

- 良い点
  - クライアントとサーバをきっちり分離することは、セキュリティや信頼性の面で良い
  - ネットワーク環境でうまく動作する
  - 大概のUnixで移植が可能
- 悪い点
  - サーバ間通信のために共有メモリへ依存しており、拡張性を制限している
    - 単一サーバサイトを複数マシンに広げることができない
  - 接続開始のオーバーヘッドが短期間のクライアントタスクに良くない
    - 通例の次善策は、クライアント側で接続をポーリングすること（AOLServerの例）

System catalogs and data types: introduction

- Postgresは、他のほとんどのDBMSよりも、かなりカタログ主導型である
  - 通例の、表やカラムやインデックスを記述するシステムカタログのようなものを持つ
  - さらに、データ型式、関数、演算子、インデックスアクセス法などに関する情報を格納するのにシステムカタログを使用する
  - 新しいカタログエントリを加える（関数を加える場合は、バックグラウンドで動くコードを書く）ことによって、システムを拡張することも可能

System catalogs and data types: basic catalogs

- 表を記述する基本システムカタログ
  - **pg_class**
    - DBの各表あたり1行に、名前(relname)・所有者(relowner)・アクセス許可(relacl)・列の数(relnatts)などを含む
  - **pg_attribute**
    - DBの各表あたり1行に、名前(attname)・データタイプ(atttypid)・列の数(attnum)などを含む
  - **pg_index**
    - 表とそのインデックスを関係づける（インデックスは表であり、したがって**pg_class**と**pg_attribute**中に独自の属性も持っている）
    - インデックスごとに1行、以下の情報を含む
      - インデックスの**pg_class**エントリへの参照(indexrelid)
      - このインデックスで構成するテーブル用の**pg_class**エントリへの参照(indrelid)
      - インデックスがテーブルのどの列に計算されているかの情報(indkey)
      - インデックス演算子が何であるかの情報(indclass)
  - それほど重要ではない表（制約定義を格納した**pg_relcheck**(7.3.3では**pg_constraint**?)のような表）が他にもたくさんあるが、ここでは大事なところだけ述べる

System catalogs and data types: functions

- **pg_proc**
  - 関数（またはプロシージャ）を定義する
  - 各関数ごとに、名前(proname)・引数のデータ型(proargtypes)・戻り値のデータ型(prorettype)・関数の実装言語(prolang)・そして関数定義（prosrc:もし関数が手続き型言語であればテキスト、コンパイルされていれば実行コードへのリファレンス、そのどちらか）が1行に格納される
  - コンパイル済関数はサーバに静的にリンクされることもできるし、共有ライブラリとして動的にロードさせることもできる
  - 理論上他の言語を選択することもできるが、コンパイル済関数は一般的にCで書かれる
- **pg_language**
  - 関数を実装する言語を定義する
  - 組み込みでサポートしている言語はinternal（静的リンクされたコンパイル済コード）・C（動的リンクされるコンパイル済コード）・SQL（本体は1つ以上のSQLクエリ）
  - オプションとしてサポートしている言語は現在pl/pgsql（PL/SQL風の手続き型言語）・tcl・perl・などなど。
  - これらの言語は動的にリンクされた関数のハンドラによってサポートされる（コアサーバはそれらに関して関知しない）

System catalogs and data types: example functions

- C

```
int4
square_int4 (int4 x)
{
      return x * x;
}
```

  - 上記ソースを共有ライブラリファイルにコンパイルし、以下を宣言する

```
CREATE FUNCTION square(int4) RETURNS int4
AS '/path/to/square.so', 'square_int4'
LANGUAGE 'C';
```

- PL/PGSQL

```
CREATE FUNCTION square(int4) RETURNS int4
AS 'begin
return $1 * $1;
end; ' LANGUAGE 'plpgsql';
```

System catalogs and data types: aggregate functions

- **pg_aggregate**
  - min()・max()・count()のような集約関数を定義する
  - 集約は、集約関数 の内部遷移（状態）のデータ型(aggtranstype)・遷移関数 (aggtransfn)・最終関数(aggfinalfn) を含む
- pltclでのAVG(int4)実装例

```
-- The working state is a 2-element integer array, sum and count.
-- We use split(n) as a quick-and-dirty way of parsing the input array
-- value, which comes in as a string like '{1,2}'. There are better ways...
create function tcl_int4_accum(int4[], int4) returns int4[] as '
    set state [split $1 "{,}"]
    set newsum [expr {[lindex $state 1] + $2}]
    set newcnt [expr {[lindex $state 2] + 1}]
    return "{$newsum,$newcnt}"
' language 'pltcl';
create function tcl_int4_avg(int4[]) returns int4 as '
    set state [split $1 "{,}"]
    if {[lindex $state 2] == 0} { return_null }
    return [expr {[lindex $state 1] / [lindex $state 2]}]
' language 'pltcl';
create aggregate tcl_avg (
        basetype = int4,          -- input datatype
        sfunc = tcl_int4_accum,   -- update function name
        stype = int4[],           -- working-state datatype
        finalfunc = tcl_int4_avg, -- final-output function name
        initcond = '{0,0}'        -- initial value of working state
    );
```

System catalogs and data types: operators

- **pg_operator**
  - 式中で使うことができる演算子を定義する
  - 演算子は主として、1～2個の引数を扱う関数の単なるsyntactic sugar(注)である
- 自作累乗演算子

```
CREATE FUNCTION mypower(float8, float8) RETURNS float8 AS 'begin
return exp(ln($1) * $2);
end;' LANGUAGE 'plpgsql';

CREATE OPERATOR ** (
    procedure = mypower,
    leftarg = float8,
    rightarg = float8 );

SELECT 44 ** 2, 81 ** 0.5;
 ?column? | ?column?
----------+----------
     1936 |        9
(1 row)
```

（注） syntactic sugar（構文糖）:
人間が楽に書けるように追加される文法。略記や、短縮構文など。

System catalogs and data types: data types(1)

- **pg_type**
  - 表が持つことができ、演算子や関数が入出力することができる、基本的なデータ型を定義する
  - **pg_type**に新しいエントリを作ることで、新しいデータ型を作り出すことが可能
  - 例
    - アプリケーションにおいて、三次元空間の点の座標を格納したいとする。標準で用意された型であるPointは二次元のため格納できない。そこで三次元座標のデータ型を作ることにする。
    - 最低限、データ型は2つの関連した関数を有する。1つは内部表現を外部テキスト形式に変換する関数(typoutput)、もう1つはその逆(typinput)である。
    - これらの関数を書いたら、Postgresに次のようにして型を定義できる。

```
CREATE TYPE point3 (
    input = point3in,
    output = point3out,
    internallength = 24,  -- space for three float8's
    alignment = double ); -- ensure storage will be aligned properly
```

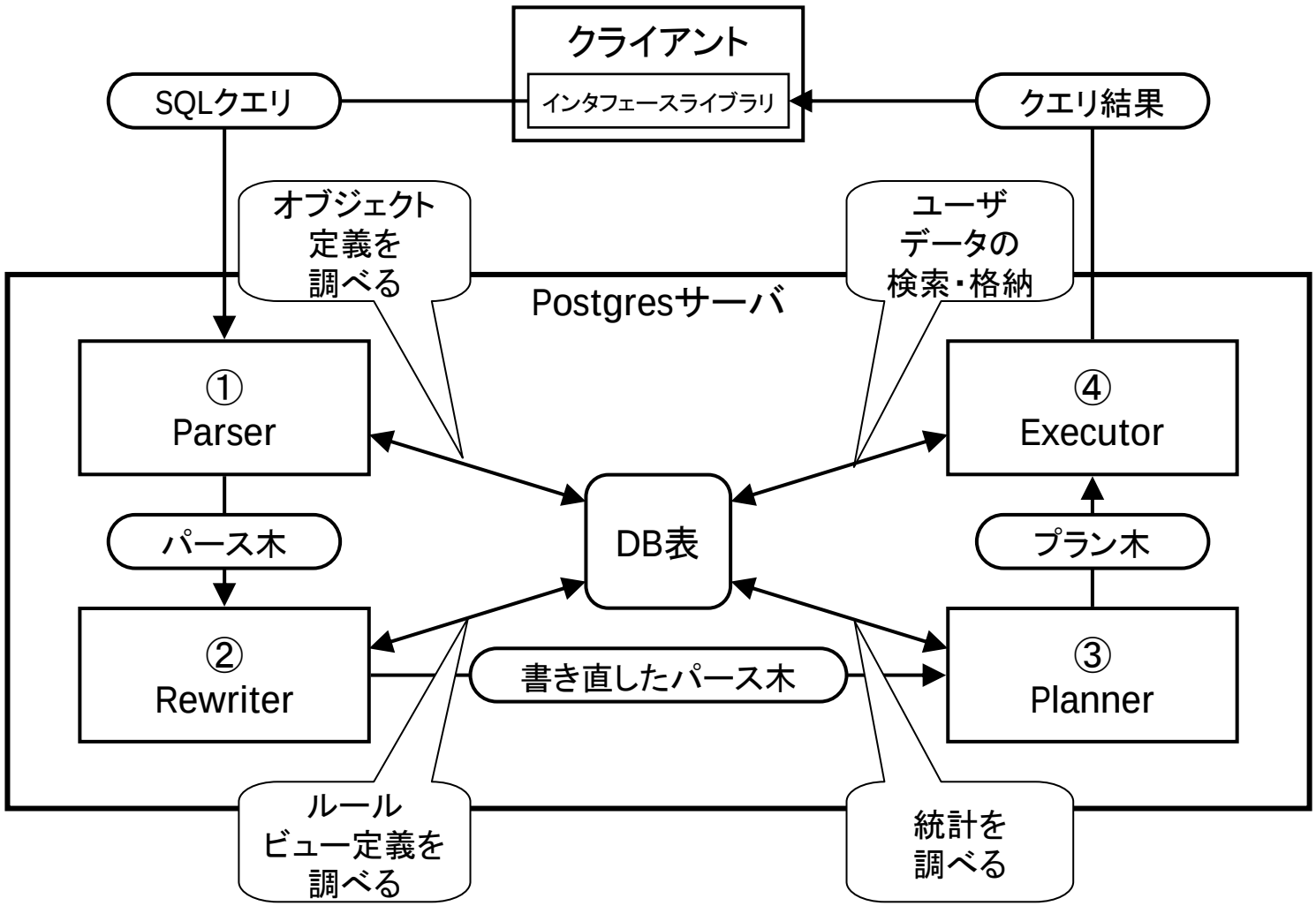
System catalogs and data types: data types(2)

- 一般的には、単に値を格納し検索するだけでなく、もっと他のこともデータ型には期待されているので、次のステップは、データ型を扱うのに役に立つ関数や演算子の付加ということになる。例を挙げると、三次元座標型を用意したなら、2点間の距離を出す関数がおそらく欲しくなるのではないだろうか。
- 新しいデータ型は**pg_amop**と**pg_amproc**にエントリを付加することで、インデックス可能にできる。これらのエントリは、インデックスアクセスメソッドが期待する方法で振る舞う比較演算子と関数のセットについて、インデクシング機械に伝える。例を挙げると、B木インデックス化可能にするために、エントリにはデータ型の **< <= = > >=** 比較演算子および3-wayソート比較関数が提供される必要がある。
- 標準のインデックスのいずれも、あまり三次元座標立体ポイントには適していない（R木は幾何学的だが、二次元である）。 理論上は、ユーザが三次元R木のための新しいインデックスアクセスメソッドを書くことができ、新しい**pg_am**エントリを作ってシステムにリンクすることが可能である。しかし、最近の記憶では、誰もそのようなことを実際にやっていない。

System catalogs and data types: summary

- 良い点
  - 関数やデータ型の拡張性は専用アプリケーションにとって大きな利益になる
- 悪い点
  - 使い勝手の良い充実した関数群を持った新しいデータ型を作るには多くの作業がいる
  - システムデザインは「できるだけ遠ざけた」「ブラックボックス化した」オブジェクトの扱いを要求する
    - 例を挙げると、集約関数MAX()とMIN()は、他のすべての集約関数と同じように処理される。つまり、MAX()とMIN()は全てのデータをスキャンしなくてはならない。整列したインデックスがあるとき、これら関数は外部インデックスの値を見ることによって効果的に実行されることができる（たとえば、Treeの端と端を見れば自明）。しかし、これを完全にkluge（場当たり的プログラム）でない方法で実現するには、集約関数とインデックス間の接続の一般的な表現を設計する必要がある（それはかなり困難な問題である）。システムデザインでは、このような「問題点への部分的解決」はそれほど好まれない。

Steps of query processing: overview



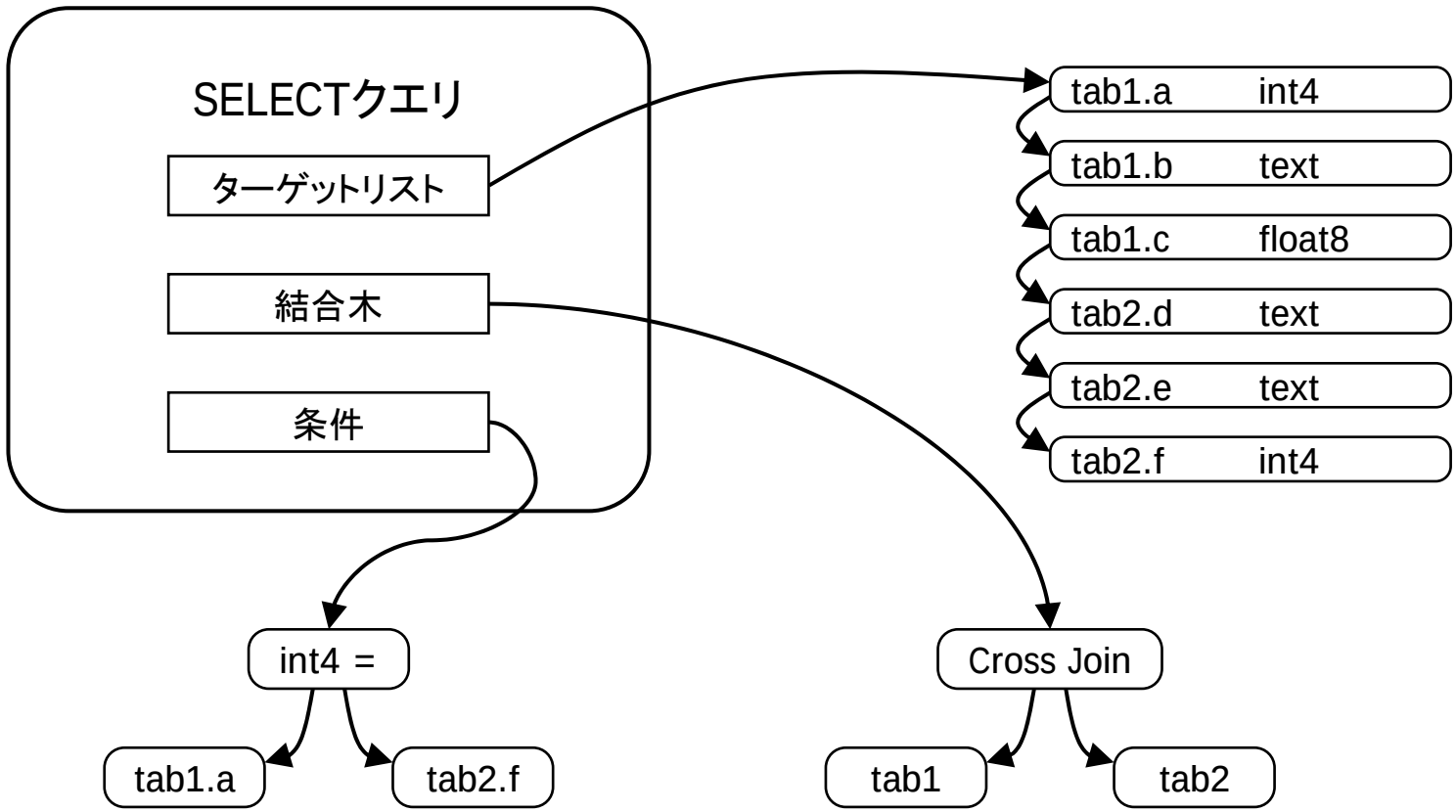
重要なデータ構造：パース木・プラン木

Steps of query processing: parser

- 入力

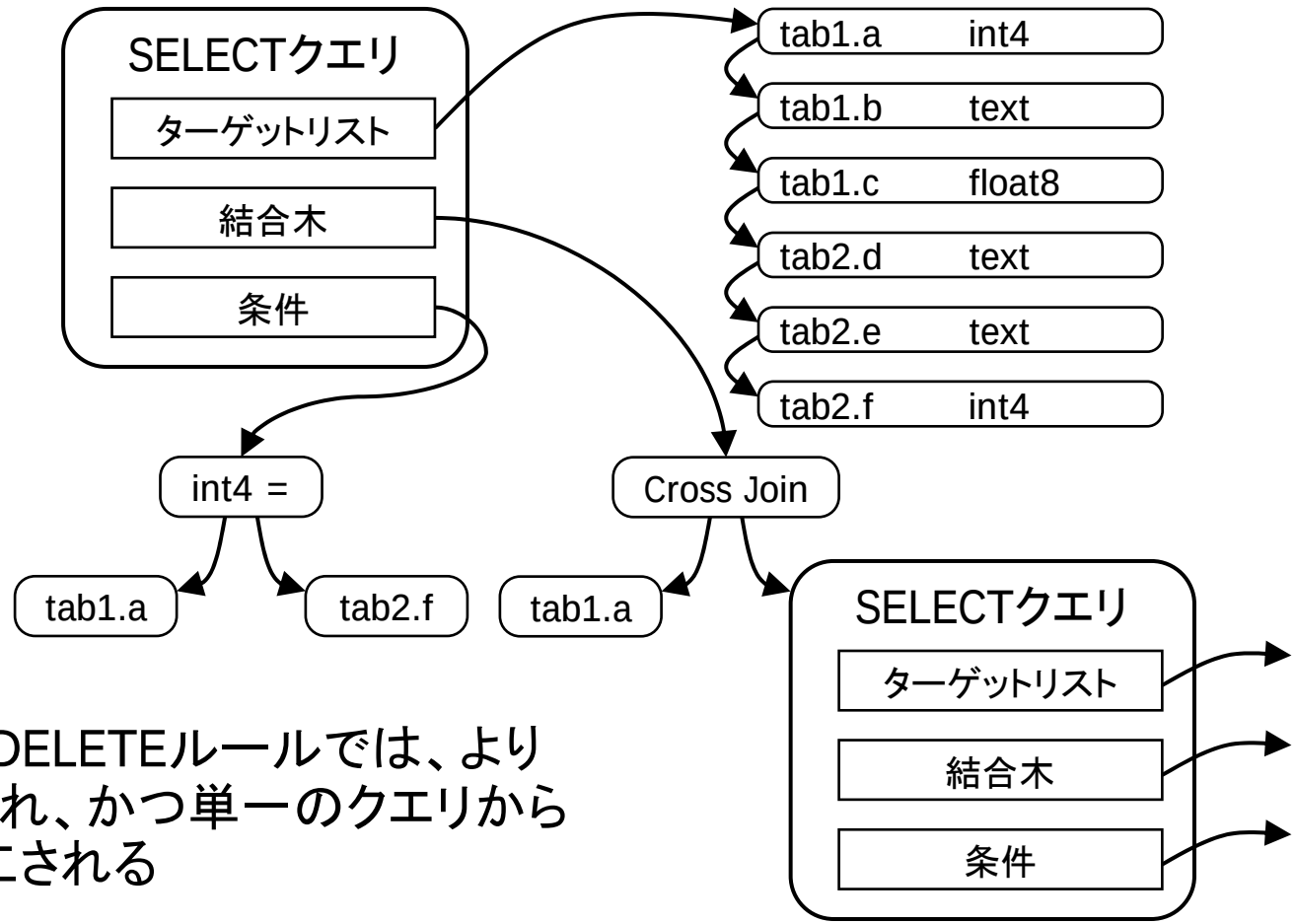
```
SELECT * FROM tab1, tab2 WHERE tab1.a = tab2.f
```

- 出力



Steps of query processing: rewriter

- ビューは、パース木への副クエリの置換によって処理される
  - 例を挙げると、もし tab2 がビューならば、
    rewrite はこんな感じ
    のものを発行する



- ON INSERT/UPDATE/DELETEルールでは、より
  大規模な変換が要求され、かつ単一のクエリから
  複数のクエリが引き起こされる

Steps of query processing: executor

- executorの基本的な目的は、プラン木（処理ノードのパイプライン化されたデマンドプル型ネットワーク）を実行することである。executorが呼ばれるたびに、各ノードはその出力順序における次のタプルを生産する。上位レベルのノードは、自分の子ノードを、子ノード自身が計算した出力タプルから入力タプルを得るために呼ぶ。

- 一番下のレベルのノードは物理的なテーブルをスキャンする（シーケンシャルスキャンか、インデックススキャンのどちらか）。

- 上位レベルのノードは通常結合ノードである（nested-loop・merge・hash joinが利用できる）。各結合ノードは、2つの入力タプルストリームを1つに結合する。

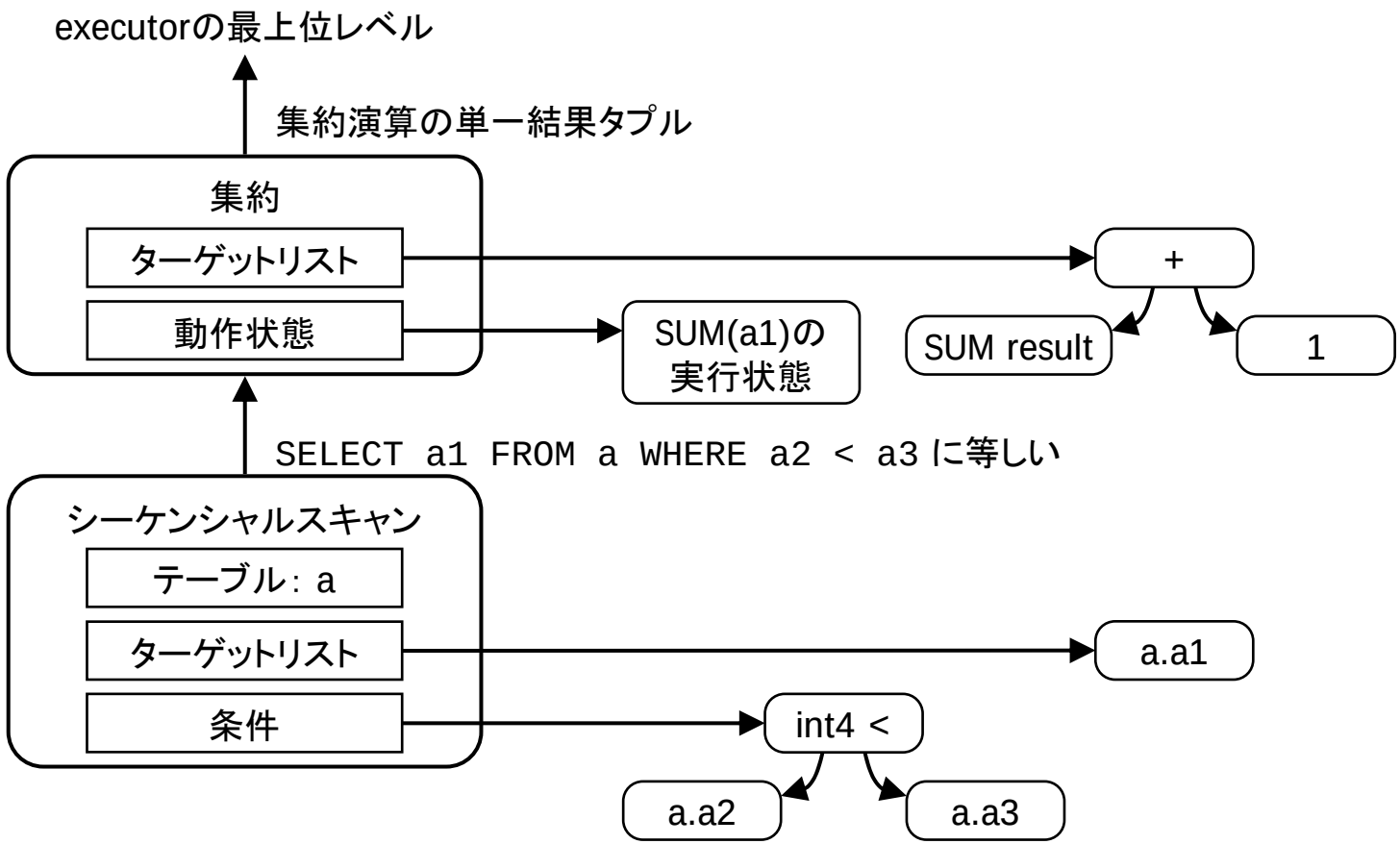
- 他にも、SORTやAGGREGATEのように特別の目的で用いられるノード型が存在する。

Steps of query processing: executor example

- クエリ

```
SELECT SUM(a1)+1 FROM a WHERE a2 < a3
```

- プラン木

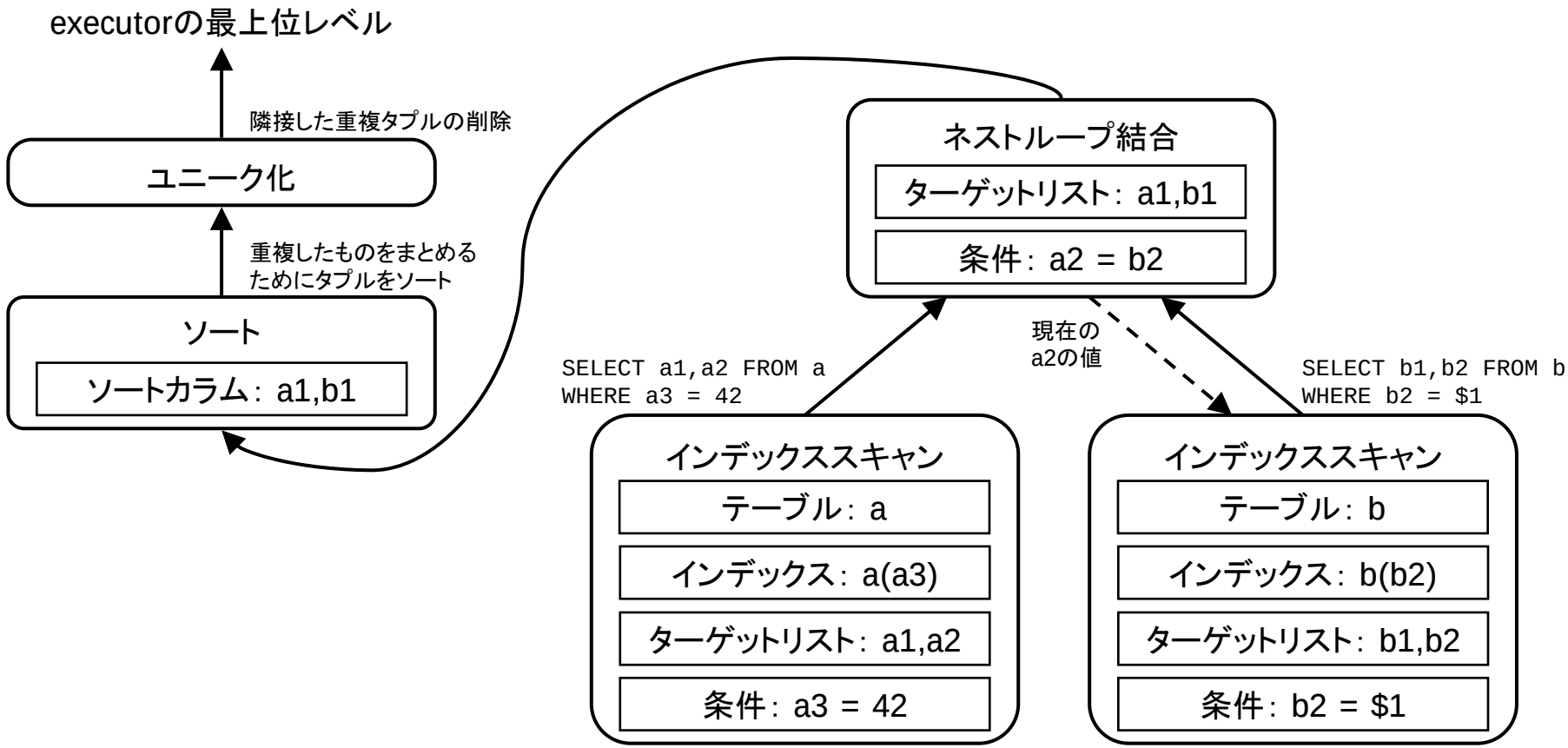


Steps of query processing: executor example

- クエリ

```
SELECT DISTINCT a1, b1 FROM a, b WHERE a2 = b2 AND a3 = 42
```

- プラン木



Steps of query processing: execution of non-SELECT queries

- もしクエリがSELECTでなかったらどうするか？
  実は、プラン木処理機はそれほど気にしない。
  - INSERTは、結果列の出力先を除いてSELECTと同じ。
  - UPDATEとDELETEに対して、Plannerは、Executorの最上位レベルがどの列をアップデートするか、または削除するかを決定するのに使用する選択された行のTIDを返すために、隠しtargetlist itemを加える。
- したがって、プランニングやExecutorのほとんどにとっては、全てのクエリはSELECTに見える。
  Executorの最上位レベルだけが、クエリの種類に応じて違った振る舞いをする必要がある。

Steps of query processing: planner

- プラン木について見てきたので、Plannerについて論じることができる。
  Plannerの基本的アイデアは、与えられたクエリに対するコスト見積ベースの最適プラン木選択である。
- 簡単な例

```
SELECT * FROM t WHERE f1 < 100
```

- t(f1)にインデックスがあると仮定して、2種類の可能性のある実行プランが考えられる。
  - t全体のシーケンシャルスキャン
  - f1 < 100というインデックス制限を設けたインデックススキャン
- 両プランのコスト(ディスクページの取得とCPU時間の面)が見積もられ、低く見積もられたコストのプランが選択される。
- 注意したいのは、インデックススキャンはいつでも勝者ではなく、またそうであるべきではないということ。
  選択は、検索されるために見積もられたtの列の割合に依存する。

Steps of query processing: join planning

- 複数表のクエリでは、最初に各コンポーネント表のシーケンシャルスキャンとインデックススキャン（適用できる場合）のコストを見積もり、次に全ての考えられるペアを成す結合パスが考慮されたうえで結合木を構築する。木の k 番目のレベルが、ベース表のどんな k も結合する最も低コストの方法を提示し、最上位レベル(n 個の表のクエリの場合レベルn)で総合的な低コストの結合パスがわかる。

- もし大量の表が含まれている場合（約10個以上）、可能性のある結合パスの数が指数関数的に上昇するために、この網羅的な結合空間の探索は非現実的である。このような場合、遺伝的最適化アルゴリズムを用いた限定的な数の代替手段による確率的な探索に戻る。

Steps of query processing: summary

- 良い点
  - 通常、システムはユーザの補助なしで良いクエリプランを見つけてくれる
- 悪い点
  - プランは、ユーザデータに関するシステムのコストモデルと統計量の面で良いに過ぎない
    - もし統計が全く的外れだった場合、あっと驚くような悪いプランを選択する可能性もある。
    - モデルと統計の改善に取り組んでいる。
  - プランを生成する時間が、特に繰り返しの多いクエリにおいてじれったくなる可能性がある
    - 複雑なクエリをplpgsql関数に押し付けることが助けになる。なぜなら、plpgsqlはプランをキャッシュするからである。
    - また、より一般的なプランキャッシュメカニズムについて論じているところである。

summary

- PostgreSQLは、精通するためにはたくさんの学習が要る複雑なシステムである。
- もしあなたが学習したなら、多くのデザインの緻密さに気づくだろう。
  - デザインの簡潔さへの名誉の大部分はStonebrakerとバークレー校の彼の学生たちによるもので、現在の開発者によるものではない。
- PostgreSQLはデザイン上含んでいる制限がいくつかあるが、競合のDBMSができないようなこともやってのける能力もある。