SRA OSS, INC.

PostgreSQL 11 で登場したJIT コンパイルって結局何者？
(What is JIT Compilation Introduced in PostgreSQL 11?)

長田 悠吾（Yugo Nagata）/ SRA OSS, Inc. 日本支社

PGConf.ASIA 2018
2018.12.12

自己紹介

- 長田 悠吾（Yugo Nagata）
  - チーフエンジニア@ SRA OSS, Inc. 日本支社
  - PostgreSQL
    - 技術支援
    - コンサルティング
    - PostgreSQL インターナル講座講師
    - 研究開発

PostgreSQL 11

- PostgreSQL 11 の新機能として JIT コンパイル基盤が導入

> • Add **Just-In-Time** (JIT) compilation of some parts of query plans to improve execution speed (Andres Freund)
>
> This feature requires LLVM to be available. It is not currently enabled by default, even in builds that support it.

- 実行時間を向上させるため、クエリプランの一部分に対し**実行時（JIT）コンパイル**を追加した
- この機能を使うには **LLVM** が必要である

JIT コンパイル?

LLVM？

Outline

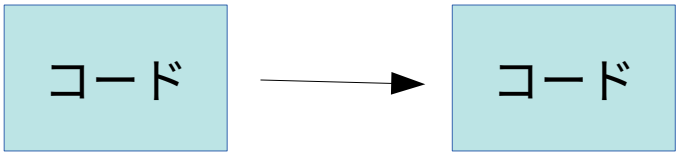
- Just-in-Time (JIT) コンパイルとは
- LLVM とは
- PostgreSQL のクエリ処理の中で、どのように使用されているか

コンパイラ（compiler）

- 高水準言語によるソースコードから、機械語に（あるいは、元のプログラムよりも低い水準のコードに）変換するプログラム

(Wikipedia)

  - 「コード」から「コード」への変換

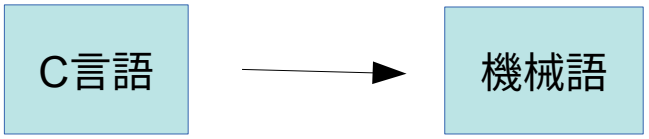


コンパイル

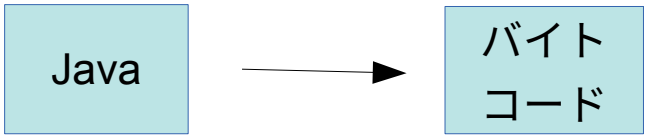
  - 変換後のコード
    - コンピュータにより直接的に実行可能な機械語（ネイティブコード）
    - 元のプログラムよりも低水準な中間言語

事前（AOT）コンパイル

- 事前（Ahead-of-Time: AOT）コンパイラ
  - アプリケーション実行前に事前にコンパイルするコンパイラ
    - C言語
      - ソースコード（.c）を 機械語（ネイティブコード）に変換
      - PostgreSQL バイナリのビルド



    - Java
      - ソースコード（.java）→ バイトコード（.class）
      - バイトコードは、Java 仮想マシン（JVM）で実行可能

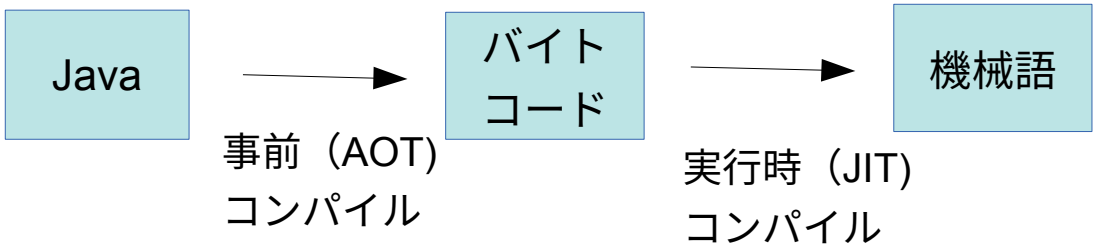


インタプリタ（interpreter）

- プログラミング言語で書かれたソースコードないし中間表現を逐次解釈しながら実行するプログラム (Wikipedia)
  - コードを「頭から順番に解釈しながら実行」
  - 長所
    - 事前にコンパイルを必要としない
    - 特定のアーキテクチャに依存しないプログラム
  - 短所
    - 実行時の性能の低さ
  - Java
    - JVM によるバイトコードの解釈実行

実行時（JIT）コンパイル

- 実行時（Just-in-Time：JIT）コンパイラ
  - ソフトウェアの実行時にコードのコンパイルを行い実行速度の向上を図るコンパイラ
    - Java
      - 実行頻度の高いメソッドを実行時にネイティブコードにコンパイル



    - Python + Numba
      - 指定した関数を実行時にコンパイルして実行

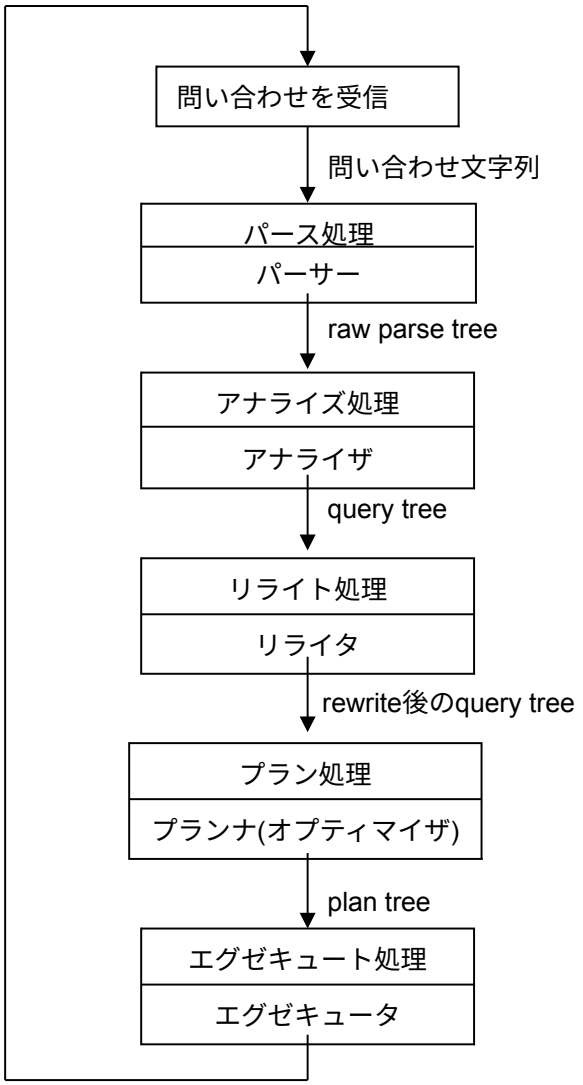
PostgreSQL 11 における JIT コンパイル

- SQL 中に含まれる「式」の評価
  - WHERE 句
  - ターゲットリスト
  - 集約関数

- タプルの deforming
  - ディスク上のタプルを、インメモリの形式に変換する処理

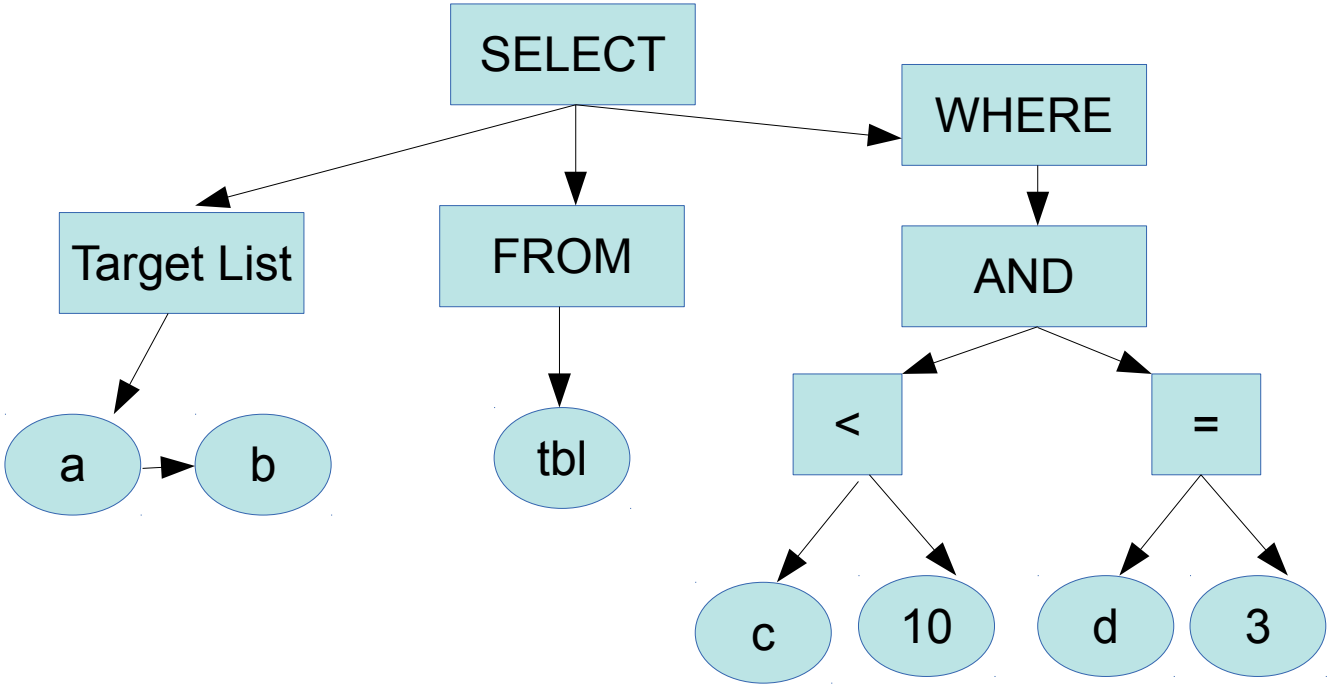
PostgreSQL のクエリ処理

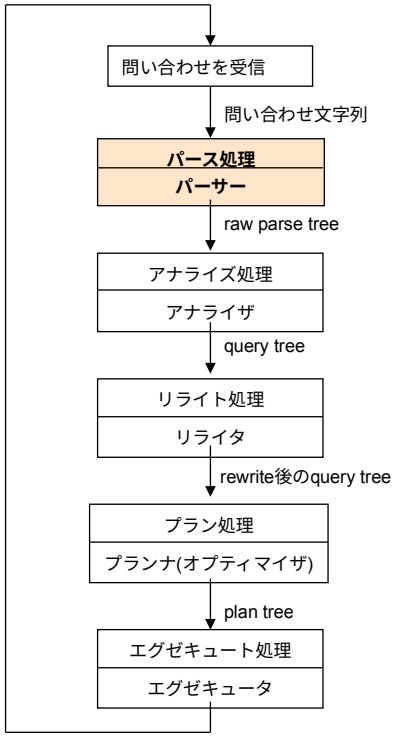
- パーサー
  - 構文解析
- アナライザ
  - 意味解析
  - システムカタログを参照し、テーブルや演算子、型などの情報を追加
- リライタ
  - クエリツリーの書き換え
- プランナ
  - クエリを処理するための最適な実行計画を生成
- エクゼキュータ
  - クエリの実行



パーサ（Parser）

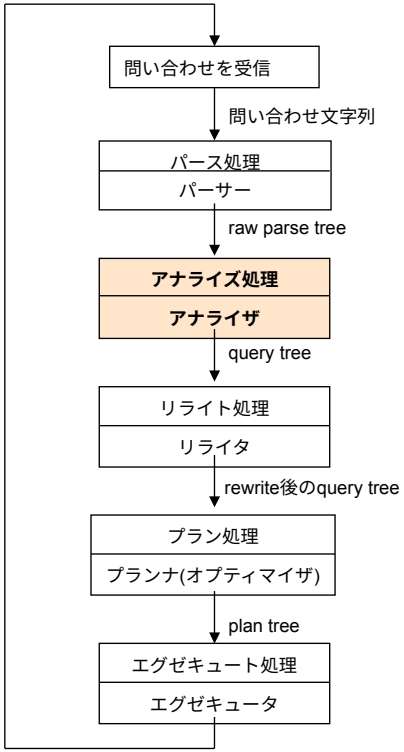
- パース処理
  - SQLの文字列を解析し、パースツリーという木構造に変換する
    - 例）SELECT a, b FROM tbl WHERE c < 10 AND d = 3;





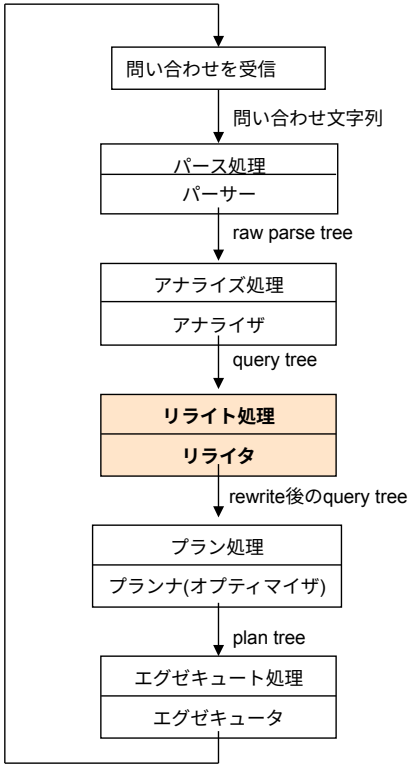
アナライザ（Analyzer）

- アナライズ処理
  - システムカタログを参照しながら、パースツリーに必要な情報を追加して、クエリツリーに変換
  - 追加される情報
    - テーブルのOID
      - スキーマサーチパスを考慮して参照されるテーブルを確定
      - テーブルに含まれる列名も調べられる
    - 型のOID
      - 定数などの型を確定する
    - 演算子のOID
      - 列や定数の型から、使用される演算子を確定する



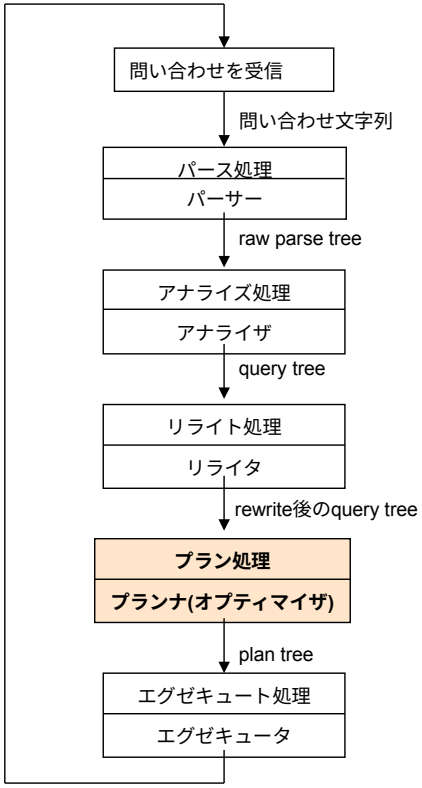
リライタ（Rewriter）

- リライト処理
  - パースツリーを書き換えを行う
  - VIEW や RULE の機能を実現
    - 「ビュー」に対する SELECTは、定義された SQL クエリの実行に書き換えられる
    - 「ビュー」に対する更新クエリは、実テーブルの更新に書き換えられる
      - 自動更新可能ビュー



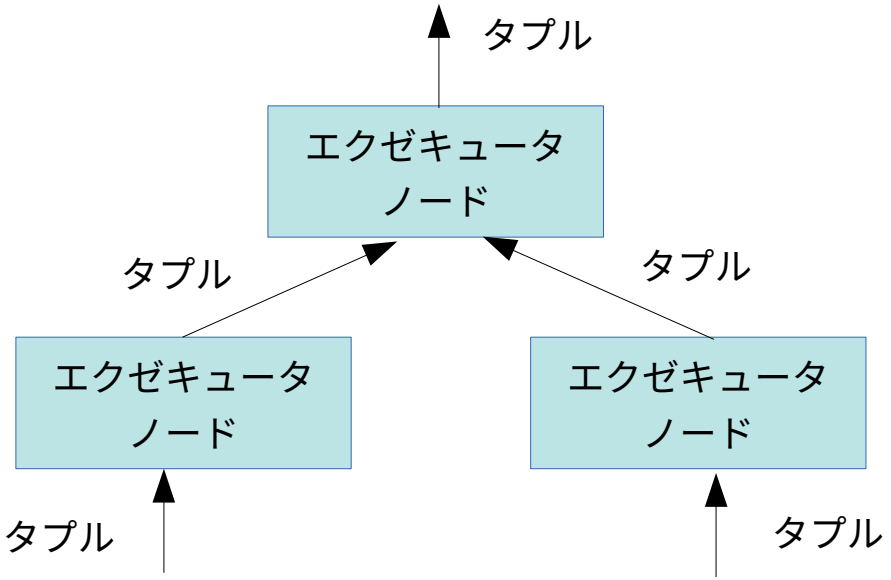
プランナ（Planner）

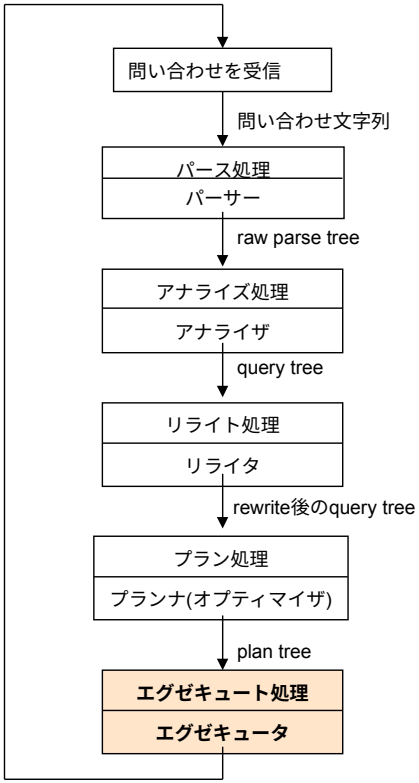
- プラン処理
  - クエリツリーから、クエリの「実行計画」を生成
    - クエリ実行の推定コストを見積もる
    - 最もコストが少なく、実行時間が短いと思われるプランを選択する
  - 実行計画はプランツリーという木構造で表される



エクゼキュータ（Executor）

- エクゼキュート処理
  - プランツリーを実行して、フロントエンドに結果を返す
- 各ノードを再帰的に実行
  - 各ノードはタプルを１行ずつ返す
  - 上位のノードは下位ノードを呼び出し、その結果返ってきたタプルを受け取って処理する

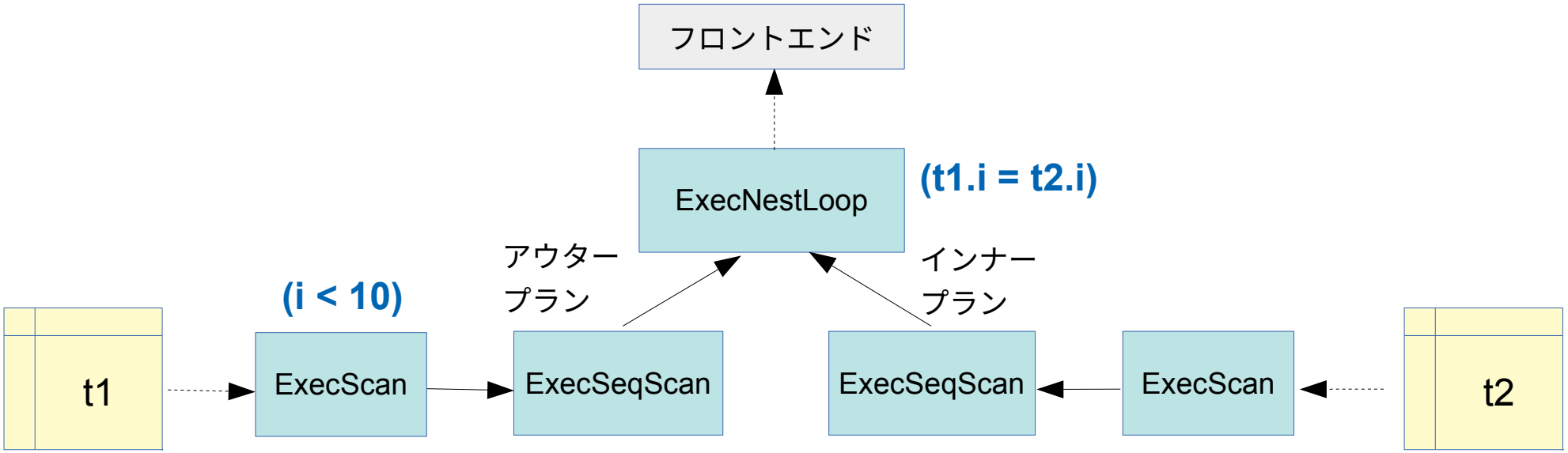




プランツリー実行の具体例

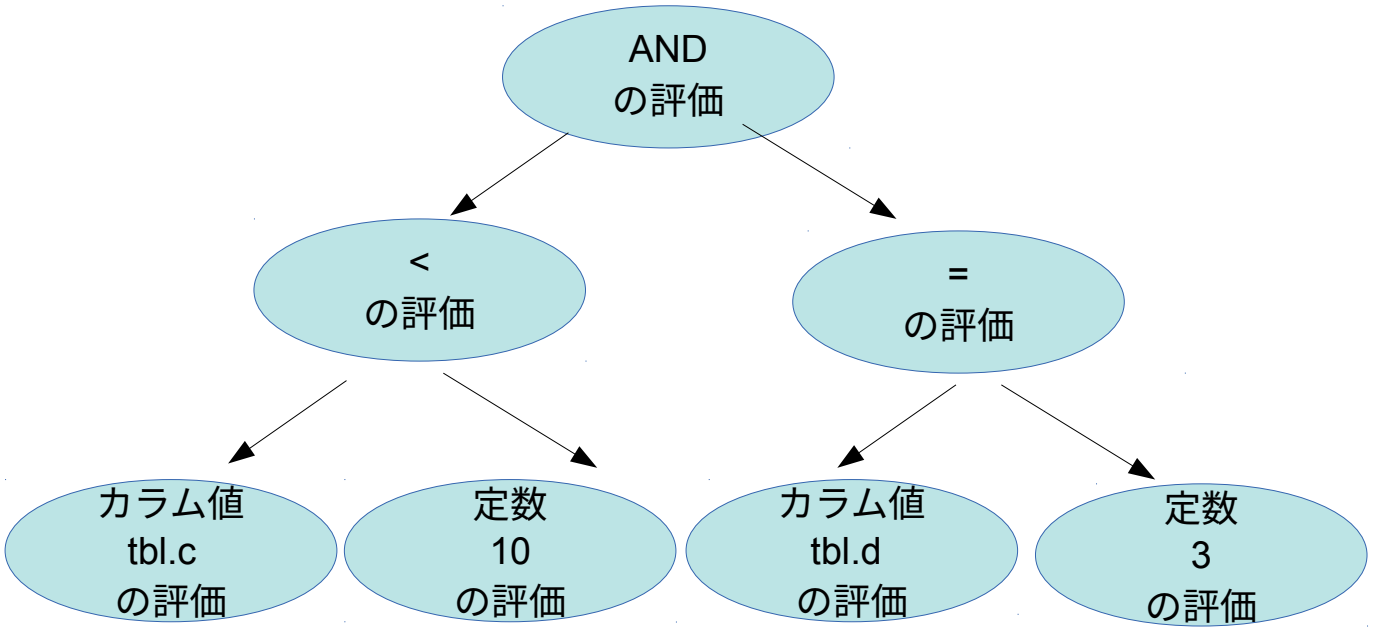
SELECT * FROM t1 INNER JOIN t2 USING(i) WHERE t1.i < 10;

```
                          QUERY PLAN
-----------------------------------------------------------
 Nested Loop  (cost=0.00..2.03 rows=1 width=12)
   Join Filter: (t1.i = t2.i)
   ->  Seq Scan on t1  (cost=0.00..1.01 rows=1 width=8)
         Filter: (i < 10)
   ->  Seq Scan on t2  (cost=0.00..1.01 rows=1 width=8)
(4 rows)
```



式の表現（expression）

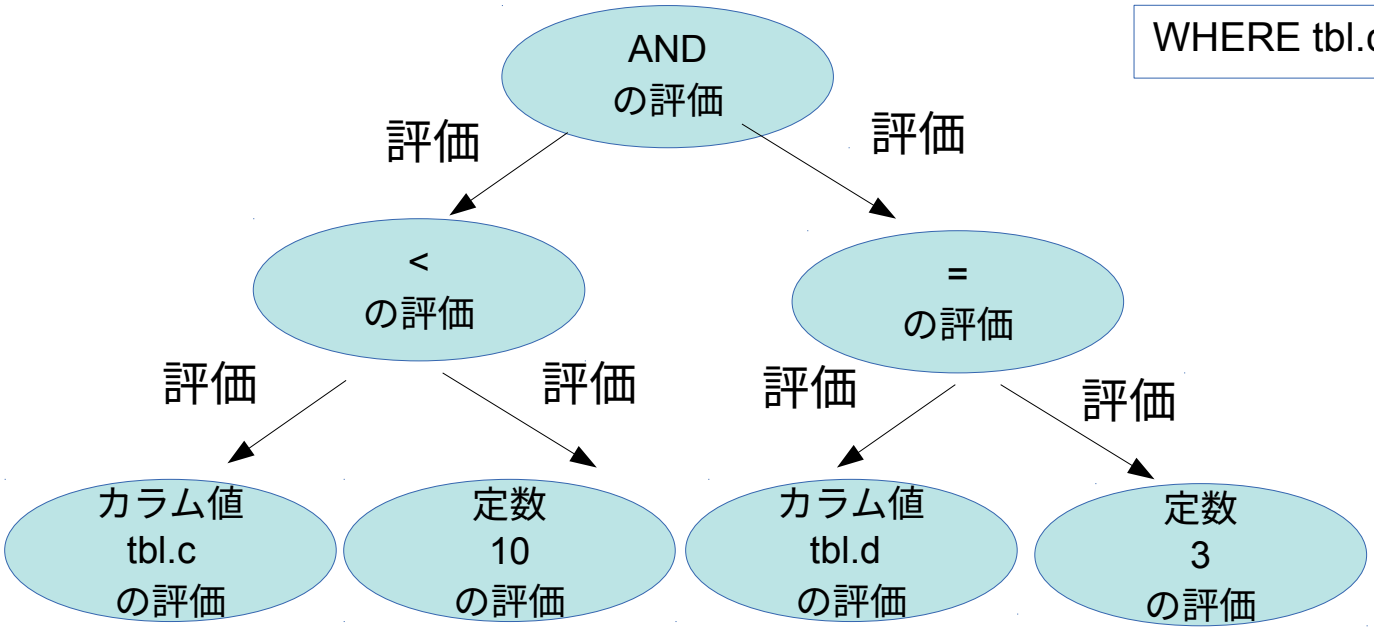
- WHERE 句の条件などに含まれる式も、木構造で表現される
  - 例）WHERE tbl.c < 10 AND tbl.d = 3



式の評価（～9.6）

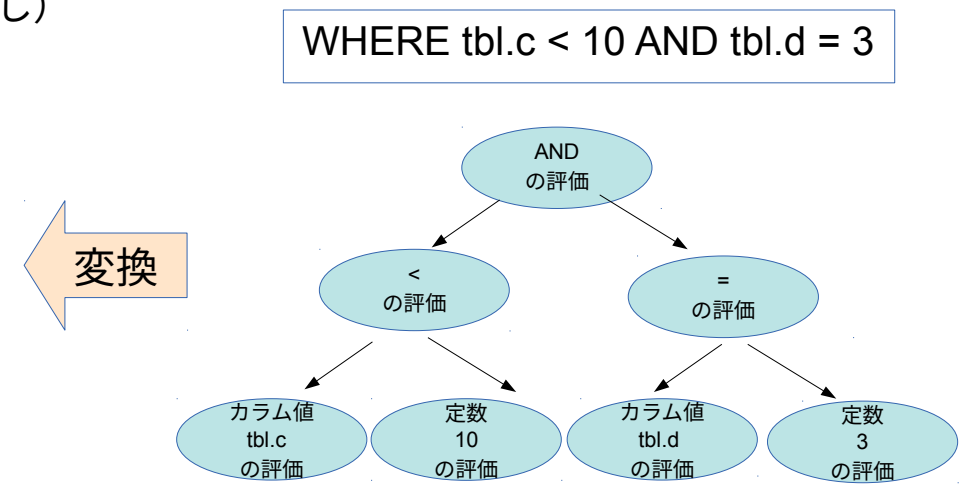
- PostgreSQL 9.6 までは、式の評価も再帰的に行われていた
  - 各ノードを評価する毎に、関数を実行
  - 上位ノードは下位ノードを評価した結果を利用

    → 関数呼び出しのオーバヘッドが高かった



式の評価（10〜）

- 木構造を以下のような中間表現（ExprState）に変換（= コンパイル）してから、実行（＝ インタプリタ）
  - EEOP_SCAN_FETCHSOME　（タプルの取り出し）
  - EEOP_SCAN_VAR　（tbl.c）
  - EEOP_CONST　（10）
  - EEOP_FUNEXPR_STRICT　（< の評価）
  - EEOP_BOOL_AND_STEP_FIRST　（AND）
  - EEOP_SCN_VAR　（tbl.d）
  - EEOP_CONST　（3）
  - EEOP_FUNCEXPR_STRICT　（= の評価）
  - EEOP_BOOL_AND_STEP_LAST　（AND）



- 関数呼び出しコストが削減され、エクゼキュータの実行が高速化
  - それでもなお、条件分岐が多いのがネック

JIT による式評価の高速化

- 一般的な式の評価を行うためには、あらゆる状況に対応しなければならない
  - 任意のテーブルに対する、任意の SQL
  - 次の命令を事前に知ることはできない
  - 予測のできない条件分岐（処理のジャンプ）が多くなる

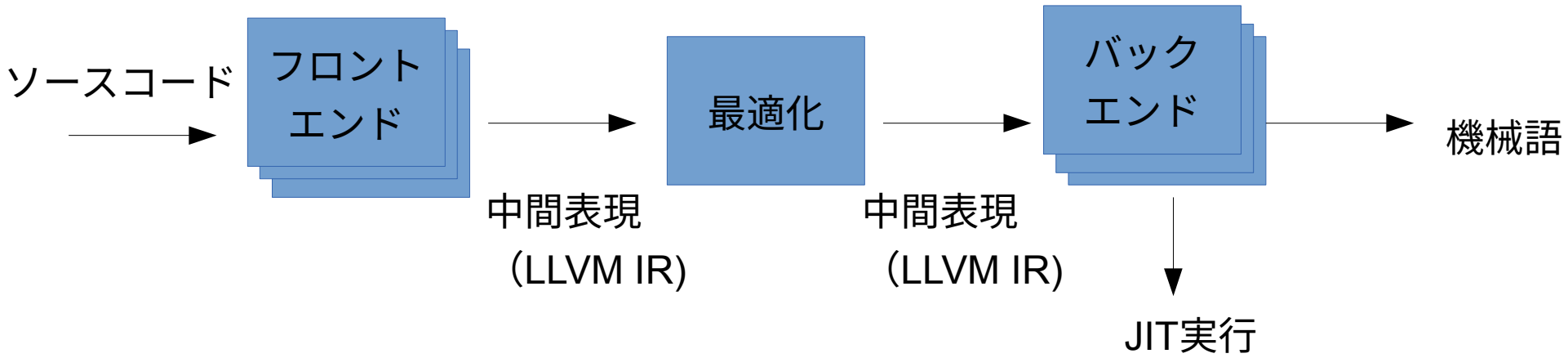
- クエリが決まった段階で、対象のテーブル、および評価すべき式も確定している
  - 「式の評価」を予めネイティブコードにコンパイルしておき、それを呼び出すようにすれば、高速化できる可能性がある

→ JIT コンパイルの出番

- LLVM を使用

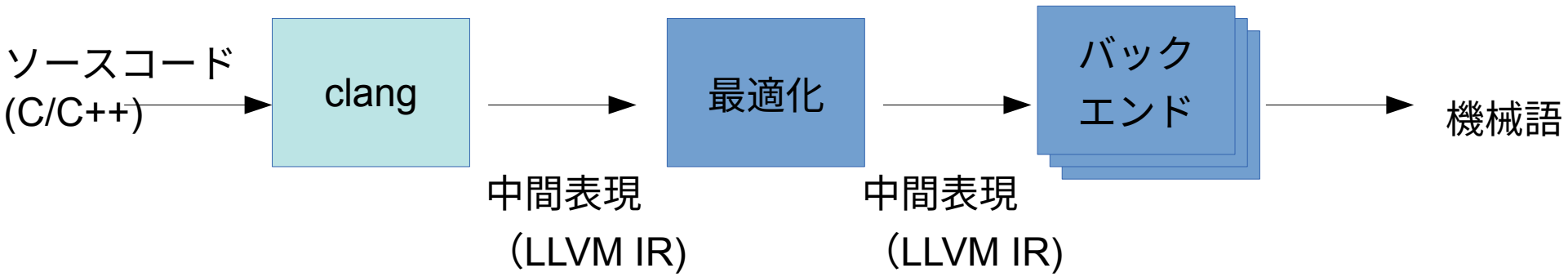
LLVM

- LLVM
  - 任意のプログラミング言語に対応可能なコンパイラ基盤
    - コンパイラ作成のための、再利用可能なコンポーネントを提供
    - LLVM IR のレベルの最適化
    - 新しいコンパイラ作成の時間とコストを削減
  - LLVMを使ったコンパイラの例
    - Clang, Swift, Rust
  - 機械語のコードを出力するだけではなく、JIT実行にも対応



Clang の場合

- Clang
  - バックエンドに LLVM を使った、C/C++ 向けコンパイラ
  - C/C++ のコードを LLVM の中間表現に変換
  - LLVM IR のレベルで最適化

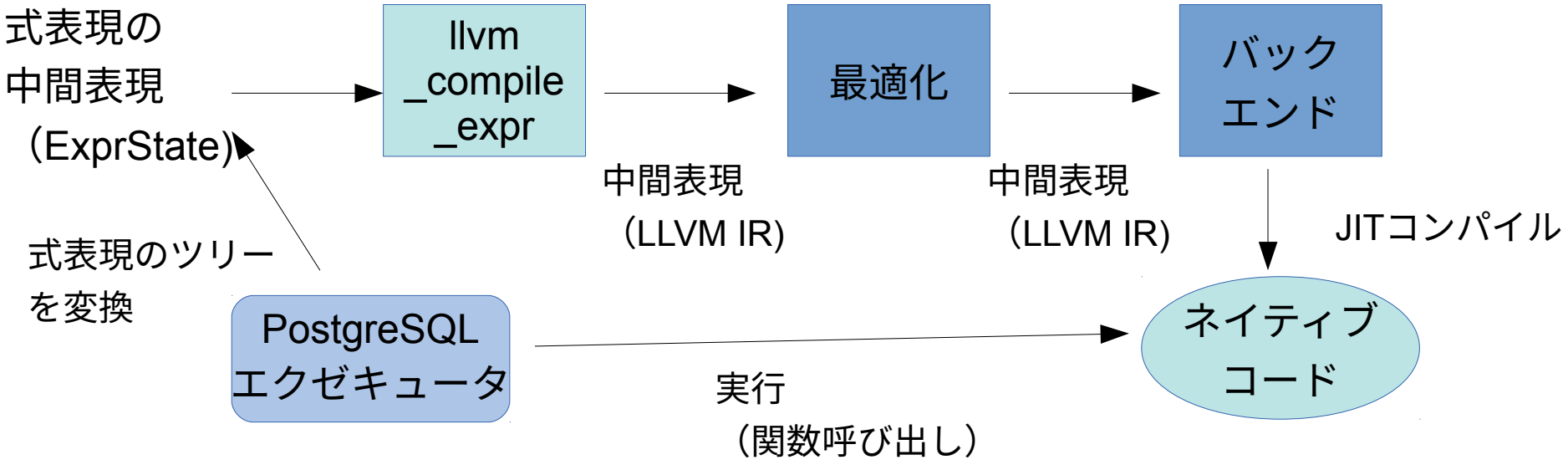


LLVM IR

- LLVM で使われる内部表現言語
- ３つの表現形式
  - メモリ内のバイナリ表現
    - これを生成するための API が用意されている
    - PostgreSQL はこれを使っている
  - ディスク上でのバイナリ表現
    - ビットコードと呼ばれる
    - ファイル拡張子は .bc
  - 人間に読めるテキスト表現
    - ファイル拡張子は .ll

LLVM による JIT

- PostgreSQL の JIT
  - 式表現の中間表現（ExprState）を、LLVMの中間表現（LLVM IR）に変換
  - それを LLVM で JIT コンパイルして、ネイティブコードに変換
  - PostgreSQL 内部から関数として実行



JIT コンパイルのタイミング

- クエリ実行直前（= エクゼキュータの初期化段階）
  - プランツリーの各ノードの初期化が行われる
  - 式表現（WHERE 条件など）を含むノードの初期化時：
    - ツリー構造の式表現が中間表現（ExprState）に変換される
    - 続いて、ExorState は LLVM IR の関数に変換される
      - この時点ではまだコンパイルはされない
  - クエリに含まれる全ての式表現が対象

- クエリ実行時（= エクゼキュータの実行段階）
  - 実際に式を評価する時に、その式評価に対応する関数が呼ばれる
  - 初回の関数呼び出し時に、全ての関数のコンパイルがまとめて実行される

タプルの deforming

- ディスク上のタプルを、インメモリの形式に変換する処理
  - 式を評価するときに、タプルを取り出す命令の中で行われる
    - 例えば、EEOP_SCAN_FETCHSOME
  - タプルからカラムの値を必要な分だけ取り出す
    - 例）カラムを１０個持つテーブルの場合

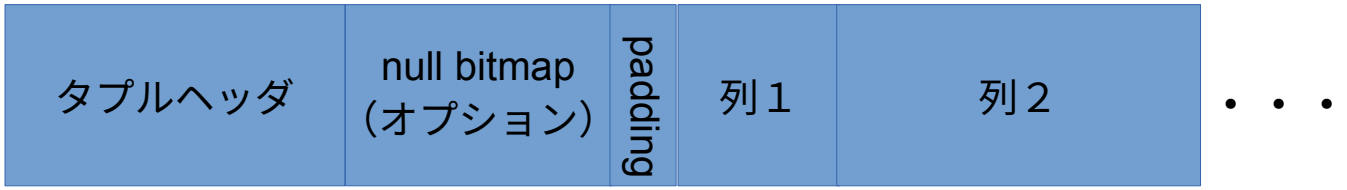
```
CREATE TABLE tbl (a1 int, a2 text, ..., a10 timestamp);
SELECT a3, a7 from tbl;
```

      - この場合は、７番目のカラム（a7） までの値を取り出す
        （８番目以降のカラムは必要ない）

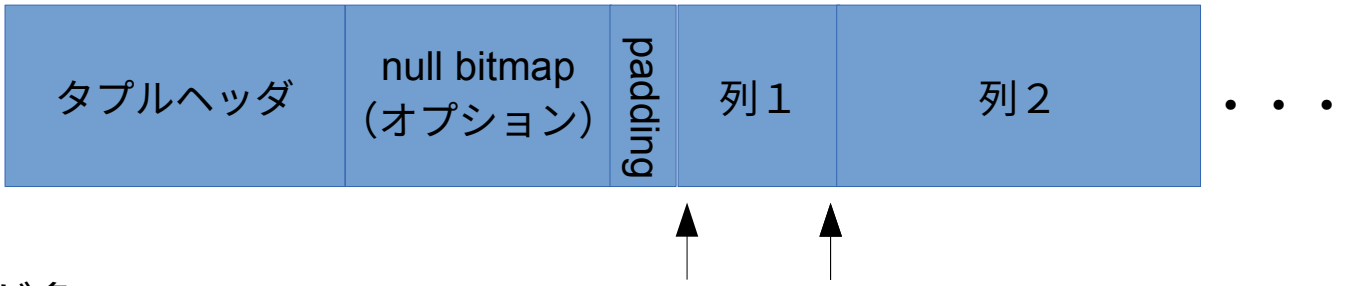
  - この処理もボトルネックとなるため、 JIT コンパイルによる高速化の対象

タプルの構造

- 固定長のタプルヘッダ
- null bitmap
  - 列の数分だけのビット列
  - もしNULLの列がタプル中に存在しなければnull bitmapは省略される
- padding
  - バイト位置調整のための「詰めもの」
- 列データ
  - 型によってサイズが異なる（可変長の型もある）
  - 列と列の間には、データ型に応じたpaddingが入ることがある



タプルの deforming の JIT 化



カラム値を取り出すには、これらの
オフセット位置を計算する必要がある

- 条件分岐が多い
  - 値がNULLかどうかの確認が必要
  - カラムの型によっても処理が異なる

  → CPUボトルネック
- テーブルのカラム型がわかっていれば、事前に取り除ける条件分岐がある
  - NOT NULL 制約があるカラムは NULL かどうかの確認は不要
  - カラムの型は事前に知ることができる

→ JIT コンパイルによって分岐を事前に取り除くことで高速化

関数のインライン展開（１）

- 式の評価には関数の実行が含まれている
  - 組み込み関数
    - sqrt, md5, random
  - ユーザ定義関数
  - 演算子の評価にも関数が使用される
    - 例）int8 同士の等号: int8eq()
- インライン展開
  - LLVM IR のコードからこれらの関数を呼び出すのではなく、関数の方を LLVM IR に変換してしまう
  - 関数呼び出しのオーバヘッドを削減

関数のインライン展開（2）

- 関数のインライン展開を可能にするため、PostgreSQL のソースコードは LLVM IR にコンパイルされる
  - 拡張モジュールをインストールした場合も同様
  - バイナリは $pkglibdir/bitcode 以下にインストールされている

```
$ tree postgresql/lib/bitcode/

lib/bitcode/
├── pg_trgm
│   ├── trgm_gin.bc
│   ├── trgm_gist.bc
│   ├── trgm_op.bc
│   └── trgm_regexp.bc
├── pg_trgm.index.bc
├── postgres
│   ├── access
│   │   ├── brin
│   │   │   ├── brin.bc
│   │   │   ├── brin_inclusion.bc
│   │   │   ├── brin_minmax.bc
│   │   │   ├── brin_pageops.bc
│   │   │   ├── brin_revmap.bc
│   │   │   ├── brin_tuple.bc
│   │   │   ├── brin_validate.bc
│   │   │   └── brin_xlog.bc
│   │   ├── common
│   │   │   ├── bufmask.bc

（以下略）
```

JIT が有効に働く状況

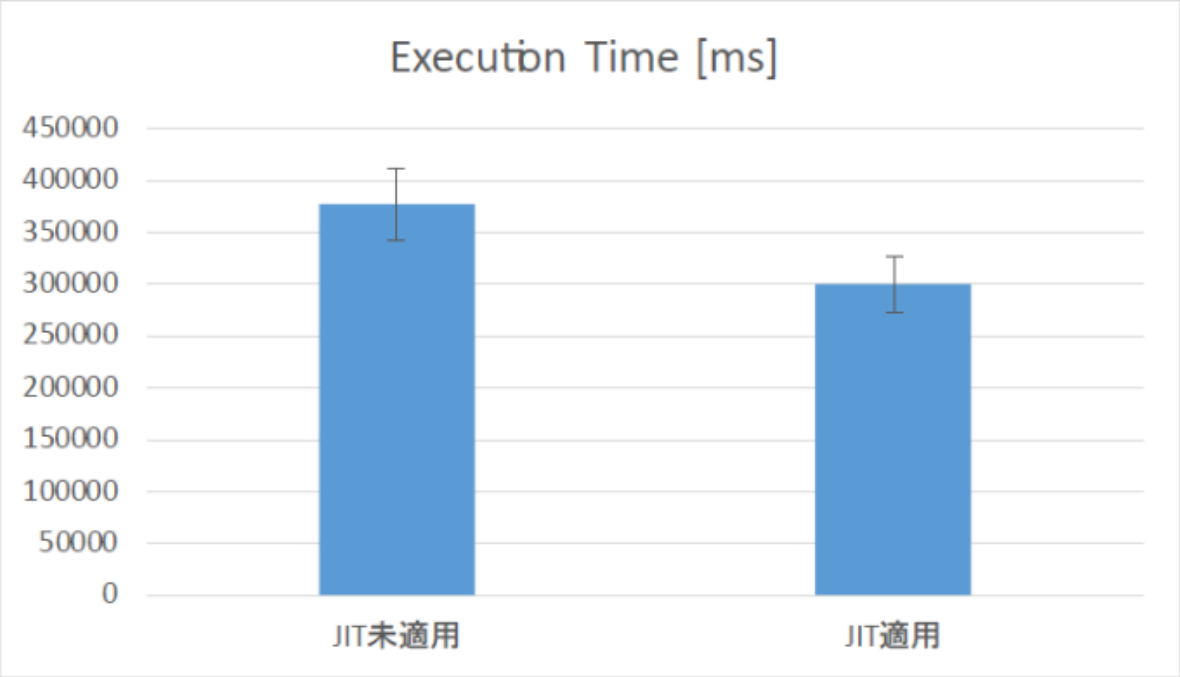
- CPU バウンドな状況が長時間になるようなクエリ
  - 複雑な式表現をもつ
  - 行数が多い
  - 主に解析系のクエリで有効
  - 短時間のクエリでは JIT のオーバーヘッドの方が大きくなってしまう

- プランナの推定コストに基づく判断
  - jit_above_costより大きくなる場合のみ JIT を使う
  - jit_inline_above_cost より大きい場合には、関数のインライン化を行う
  - jit_optimize_above_cost より大きい場合には、積極的な最適化を行う

設定パラメータ

- jit
  - JIT コンパイルを使用するかどうか
  - デフォルト：off
- jit_above_cost
  - JIT コンパイルを使用するコストの閾値
  - デフォルト：100000
- jit_inline_above_cost
  - 関数のインライン化を使用するコストの閾値
  - デフォルト：500000
- jit_optimize_above_cos
  - 積極的な最適化を使用するコストの閾値
  - デフォルト：500000
- jit_provider
  - JIT 機能を提供するライブラリの名前
  - デフォルト：llvmjit
- jit_dump_bitcode
  - 生成された LLVM IR のファイル（ビットコード：.bc）に出力
  - デフォルト：off

動作例（１）



簡易な性能テスト例：

11の関数・演算子と
3箇所のキャスト
×
1億件のループ
↓
このくらいから
効果がでる

```
CREATE FUNCTION f100000000() RETURNS SETOF bigint
  ROWS 100000000 LANGUAGE sql AS $$
  SELECT g FROM generate_series(1::bigint, 100000000::bigint) AS g;
$$;

SELECT g, 'X is "' || random() * pi() *
    substr((g * ln(g::float8 + g / 2))::text, 1, 5)::float8 || '"'
  FROM f100000000() AS g;
```

動作例（２）

- EXPLAIN ANALYZE の結果

```
 Function Scan on f100000000 g  (cost=0.25..5750000.25 rows=100000000 width=40)
(actual time=22073.673..233385.687 rows=100000000 loops=1)
 Planning Time: 0.255 ms
 JIT:
   Functions: 2
   Options: Inlining true, Optimization true, Expressions true, Deforming true
   Timing: Generation 1.926 ms, Inlining 5.988 ms, Optimization 36.134 ms,
           Emission 20.168 ms, Total 64.215 ms
 Execution Time: 236383.943 ms
(7 rows)
```

作成された関数の数

実施された JIT 処理

JIT 処理の所要時間

まとめ

- PostgreSQL 11 への JIT コンパイラ導入
  - JIT コンパイルとは何か
  - LLVM とは何か
  - PostgreSQL のどこで、どのように、なぜ使われるのか
- JIT コンパイルはまだ登場したばかりの機能
  - 今後も改善されていくはず

Thank you