PostgreSQL 解析資料

～ ソースツリーと PostgreSQL のアーキテクチャ ～

(株) NTT データ
ビジネス開発事業本部 システム方式 BU
井久保 寛明

1. はじめに

本資料では、PostgreSQL のソースツリーについて紹介する。実際にソースコードの調査を行う際に、どのあたりから見ていけばいいかという指針になることを目的としている。
ソースコードを見ていくためには、DBMS にどのような機能があるかを知っておく必要があるので、はじめに、DBMS エンジンのアーキテクチャの概要と基本的な機能を説明する。その後、ソースコード中のモジュールとの対応付けを行っていくことにする。

本資料の現状

2003 年 7 月現在[1]、PostgreSQLのソースコードの解析は、まだ、半分も終わっていないため、推測の部分が多く含まれる。そのため、誤りも少なからず含まれていることと思う。この点については、あらかじめ御了承頂きたい。
本資料は、*PostgreSQL 7.3.3* をベースに書いてあるので、他のバージョンでは、異なる場合があるので注意して頂きたい。

2. PostgreSQL のアーキテクチャ概論と機能の概要

PostgreSQL のソースツリー及びソースコードを見ていくにあたって、PostgreSQL にどのような機能があるのかを知っておく必要がある。この章では、PostgreSQL のプロセスのアーキテクチャと DBMS サーバ(backend) のアーキテクチャについて紹介する。DBMS のアーキテクチャや要素技術を押さえておくことで、ソースツリーのディレクトリ名やファイル名などから、その内容が想像できるようになる。

2.1. PostgreSQL のアーキテクチャ

まずは、PostgreSQL のプロセスに関するアーキテクチャを紹介する。PostgreSQL のクライアント／サーバ間のアーキテクチャは図 1 のようになっている。

[1] 2003 年 10 月に改訂。9 月のJPUG合宿などの情報をフィードバックしました。

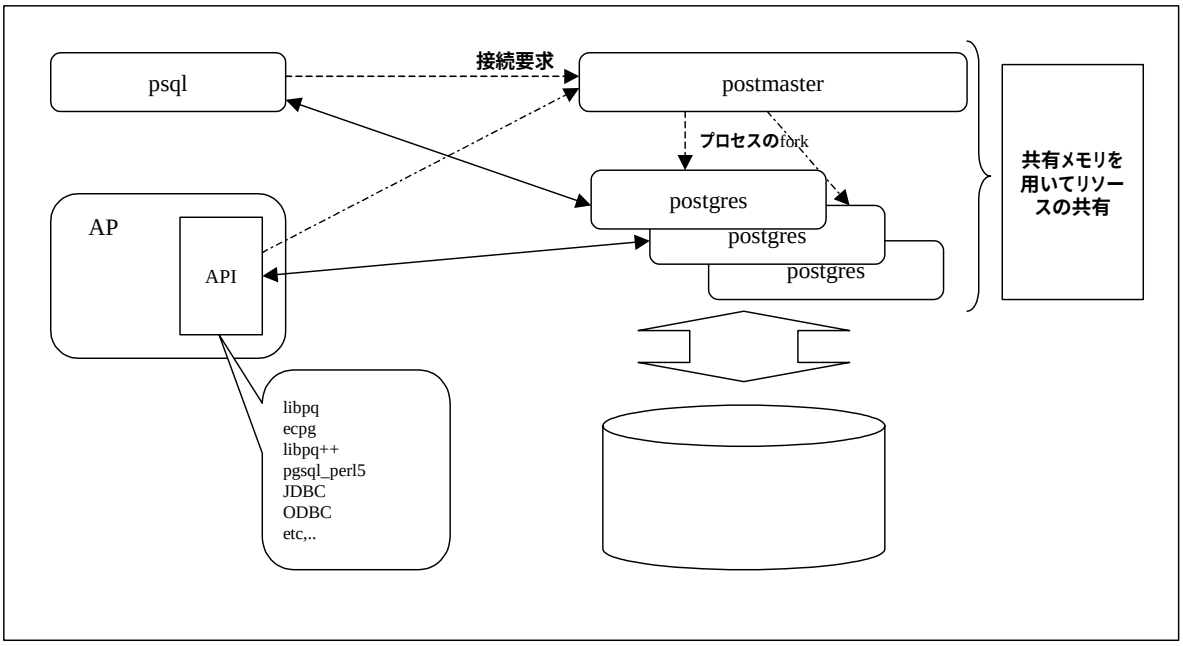

図.1　PostgreSQL のアーキテクチャ

左側２つがクライアントである。”psql”は、PostgreSQL 付属の SQL コマンドラインインタプリタである。DB のメンテナンスなどのときに psql から SQL 文を実行したり、シェルスクリプト上から SQL を実行したりする場合に利用できる。”AP”は、ユーザの作成したアプリケーションである。さまざまな API(Application Programming Interface)が用意されており、これらを通じて PostgreSQL のサーバへアクセスすることになる。
図の右側が PostgreSQL のサーバとして実行されるプロセスになる。postmaster とは、PostgreSQL のサーバの接続要求をまとめて受け付けるプロセスである。まずクライアントは、postmaster に接続要求を出す。すると、postmaster は１つの接続につき１つの postgres プロセスを fork する。以降、そのクライアントの処理は直接 postgres プロセスが受け持つことになる。各 postgres プロセスは、接続が切れる際に終了する。postgres プロセス同士は、共有メモリを利用して、バッファの共有やロック情報の共有を行う。

DB のデータの読み書きを行うのは、必ず postgres プロセスである。図は、クライアントサーバの例であるが、スタンドアロンで DB への処理を実行する場合、コマンドラインから postgres 自体を直接起動することができる。例えば DB の初期化スクリプト(initdb)の中で、このような使い方をしている。

2.2. DBMS エンジンのアーキテクチャ

次に、DBMS のエンジンに関する一般的なアーキテクチャを紹介する。本来なら、PostgreSQL のアーキテクチャをそのまま書きたいところだが、調査が十分に進んでいないため、ここでは一般的な DBMS のアーキテクチャを中心に紹介する。その後、3 章の中で、PostgreSQL のソースコード中のどのモジュールが、DBMS のアーキテクチャのどのあたりの処理に相当するのかを照らし合わせていく。

DBMS の機能を大きく分けると次の３つに分けることができる。もちろんいろいろな分け方が可能だが、本資料ではこれで説明していく。

- ストレージ管理
- キュエリの実行(キュエリコンパイラ、キュエリエグゼキュータ)
- トランザクション管理

これらの関係を表したものが、図 2 である。

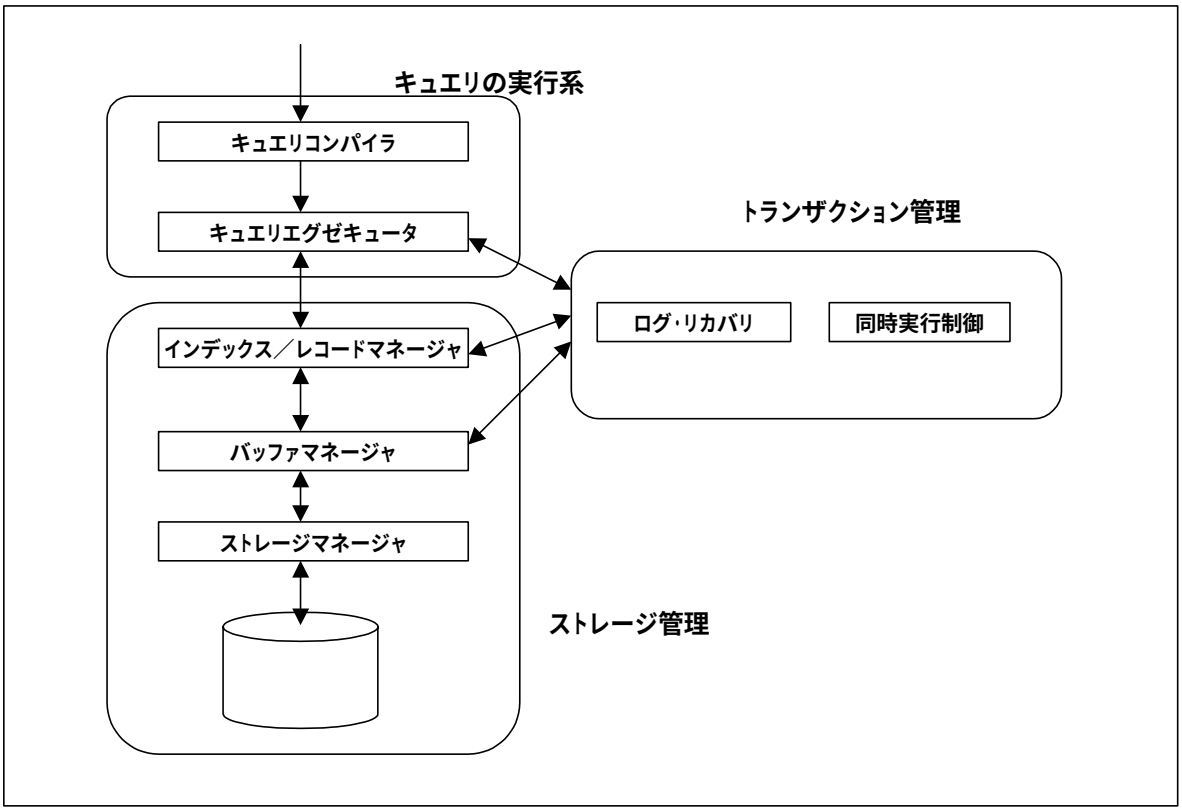

図.2　DBMS エンジンのアーキテクチャ

ストレージ管理とは、データベースに使用する OS ファイルの管理やその I/O の管理、インデックスなどの高速なファイル I/O の手段の提供、上位層へのレコード単位でのデータの供給などを担当する。基本的には OS のファイルシステムと非常によく似た技術である。

キュエリの実行系は大きく 2 つの部分からなる。1 つは、キュエリ処理の様々なアルゴリズムを実装してあるキュエリエグゼキュータで、もう１つが、与えられた SQL をどのように実行するかを決定するキュエリコンパイラである。OS など他の技術に例えると、キュエリエグゼキュータはエミュレータのようなものであり、またキュエリコンパイラはその名のとおりコンパイラである。キュエリコンパイラに至っては、C コンパイラの数倍の規模になる。

トランザクションの管理では、大きく 2 つの仕事がある[2]。1 つは、トランザクションの原子性(Atomicity)を管理することで、PostgreSQLではログを使った管理を行う。ログを使った方式は、多くの商用DBMSで用いられる主流の方法である。もう１つは、同時実行の管理である。PostgreSQLでは、

[2] 正確には、ACID特性を満たすようにする役割を担っている。Durabilityなどは、バックアップの保持なども影響するので、ここではあえて２つとする。

ロックとタイムスタンプ[3]を使った方式を採用している。

2.2.1. ストレージ管理

最近の DBMS は、データ領域として OS のファイルを使用するものも多い。OS のファイルの中を固定長のブロック単位に分けて管理する。そして、ディクス(OS のファイル)に対しては、ブロック単位で I/O を行う。多くの場合、DBMS ではメモリより多くのデータを扱う必要があるので、ディスク上の必要なデータをメモリ中にバッファリングする。各ブロックは、メモリバッファの中ではページと呼ばれる。各ページ内には、レコード(タプル) 単位のデータが格納される。固定長のレコードの中に可変長のレコードを格納する仕組みが提供されている。

テーブルに入っているデータは次第に大きくなってくるので、固定長のブロックを集めてヒープとして管理する。

大規模なデータを扱う場合、必要なレコードをディスク上から検索するのに非常に時間がかかる。そのため、検索時間を節約するために、必要に応じてインデックスと呼ばれるデータ構造を構築する。インデックスを作成するかどうかは、ユーザに委ねられる。インデックス自体も、レコード単位のアクセスモジュールを利用して作成される。

図３が、これらの動作のイメージ図である。

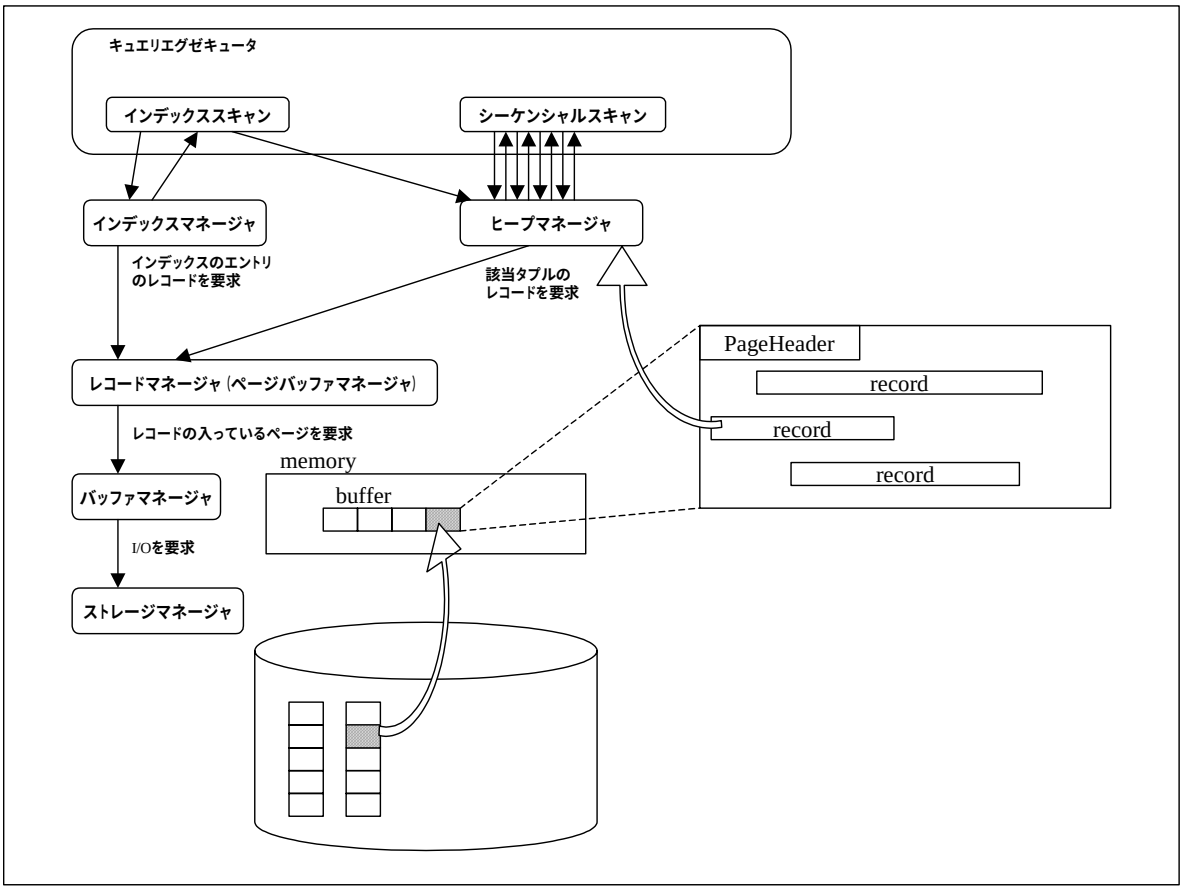

図 3. ストレージ管理

[3] MVCCと呼ばれる、多版式の同時実行性制御を行う。タイムスタンプと言っても実際は、トランザクションIDを使用する。DBMSの教科書的な話では、トランザクションIDもトランザクションの順序性を管理できるので、タイムスタンプという呼び方をする。

2.2.2. キュエリの実行系

キュエリを実行するときの流れは、図 4 のようになる。
テキストで書かれた SQL 文は、スキャナ（字句解析器)、パーザ（構文解析器）によって**字句解析**、**構文解析**が行われて、パーズツリー（図 5 の右上のツリー）に分解される。字句解析とは、テキストを読み込んで、トークンと呼ばれる「予約語や ID、データ部分などの塊」を切り出すことである。構文解析とは、字句解析で取りだしたトークンを構文規則にあてはめ、パーズツリーを生成することである。ここで、木構造にするのは、そのほうがプログラム中で扱いやすいからである。ここまでの段階で、構文のチェックなどが行われる。
続いて、**意味解析**を行う。意味解析では、テーブル名、属性名が実際に DB 内に存在するかということや、演算処理に使われる引数のデータ型のチェックなどが行われる。その他に、SQL 文で省略されているところなどを解析し、論理キュエリプランへの書き換えに必要な情報を埋めておく。教科書的は意味での意味解析の処理はここまでであり、論理キュエリプランの作成は後のフェーズで行われるのだが、PostgreSQL の場合は、この意味解析のフェーズでパーズツリーを論理キュエリプラン（キュエリツリー、図 5 の左下のツリー）に書き換える。
VIEW が存在する場合、VIEW に定義されているキュエリ定義をキュエリツリーに展開する。この書き換え処理を PostgreSQL では、**リライト**と呼んでいる。リライト処理では、単に VIEW の書換えを行うだけでなく、ルールの適用も行っている。
DDL[4]、メンテナンス用のコマンドなどは、この段階で実行が可能な状態にできるので、これらのコマンド類の場合、キュエリコンパイルの処理は終了して、ユーティリティ系処理の実行系に渡される。これ以降は、SQLのうちDML[5]に対して行われる。

教科書的な DBMS の話になるが、論理キュエリプランは、代数学に基づいたさまざまな書き換え処理が可能な木構造である。この書き換え可能な性質を利用して、キュエリの最適化を行っていく。キュエリの最適化は、経験則に基づいて変換する方法や統計情報を用いて変換していく。どのような方針で最適化を行うかは、実装に依存する。
論理キュエリプランの最適化が終わったら、物理キュエリプランへの変換を行う。ここでは、演算のアルゴリズムの決定や中間結果の渡し方、テープルのスキャン方法などを決定していく。物理プランでも統計情報などに基づいて最適化が必要である。これらの処理を経て、キュエリプランが決定される。

PostgreSQL の場合、論理キュエリプランでは、それほど最適化をかけていないように見受けられる。プランナーでは、物理キュエリプラン(パスツリー、図 5 右下のツリー)を生成する。プランナーで実行可能な全ての実行パスを生成して、それぞれのパスツリーのコスト計算を行い、最適なキュエリプランを選択する。
最適なキュエリプランが選択できたら、そのパスツリーをキュエリエグゼキュータに渡してキュエリエグゼキュータを実行する。PostgreSQL の場合、パスツリーだけでなく、キュエリツリーもエグゼキュータに渡している。これは、エグゼキュータの初期化に必要な情報がキュエリツリーにだけ入っているからである。

---

[4] DDL(Data Define Language) : スキーマやテーブルを定義するのに使用されるSQL文
[5] DML(Data Manipulation Language) : Select, Insert, Delete, Update など、データ操作のSQL文

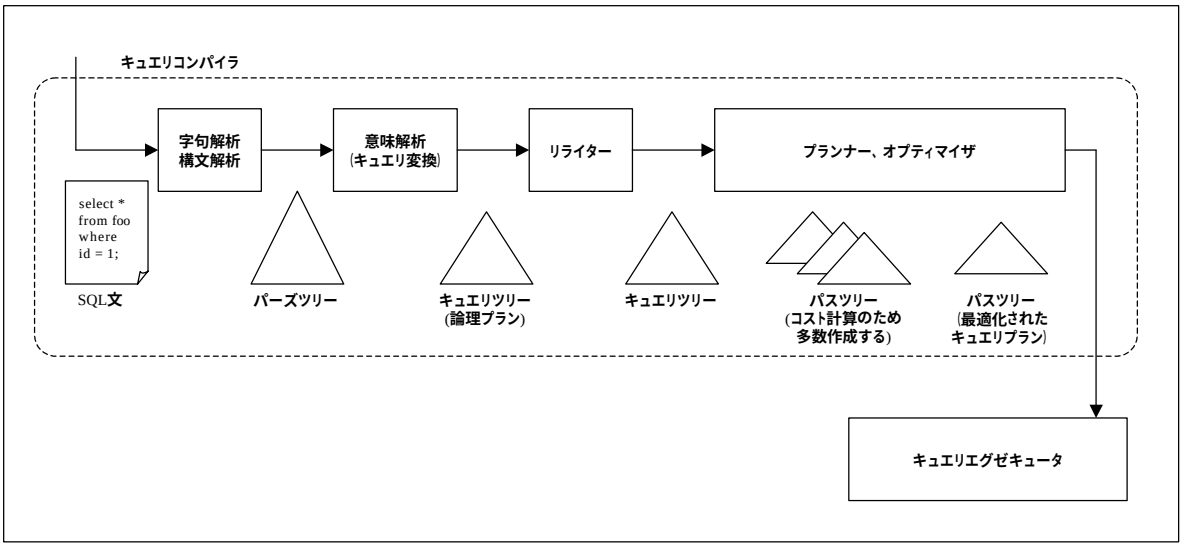

図 4. キュエリの実行系

キュエリエグゼキュータには、さまざまな処理を行うためのアルゴリズムが実装されている。キュエリコンパイラの生成したキュエリプランには、どのアルゴリズムにどの引数を渡して処理するということが細かく指示されているので、エグゼキュータは、その指示どおり動くだけである。
エグゼキュータの実装では、ストレージ管理モジュールに対してレコード単位での処理要求をする。

キュエリを処理していく際のデータ構造の変化は、次の図のようなイメージになる。

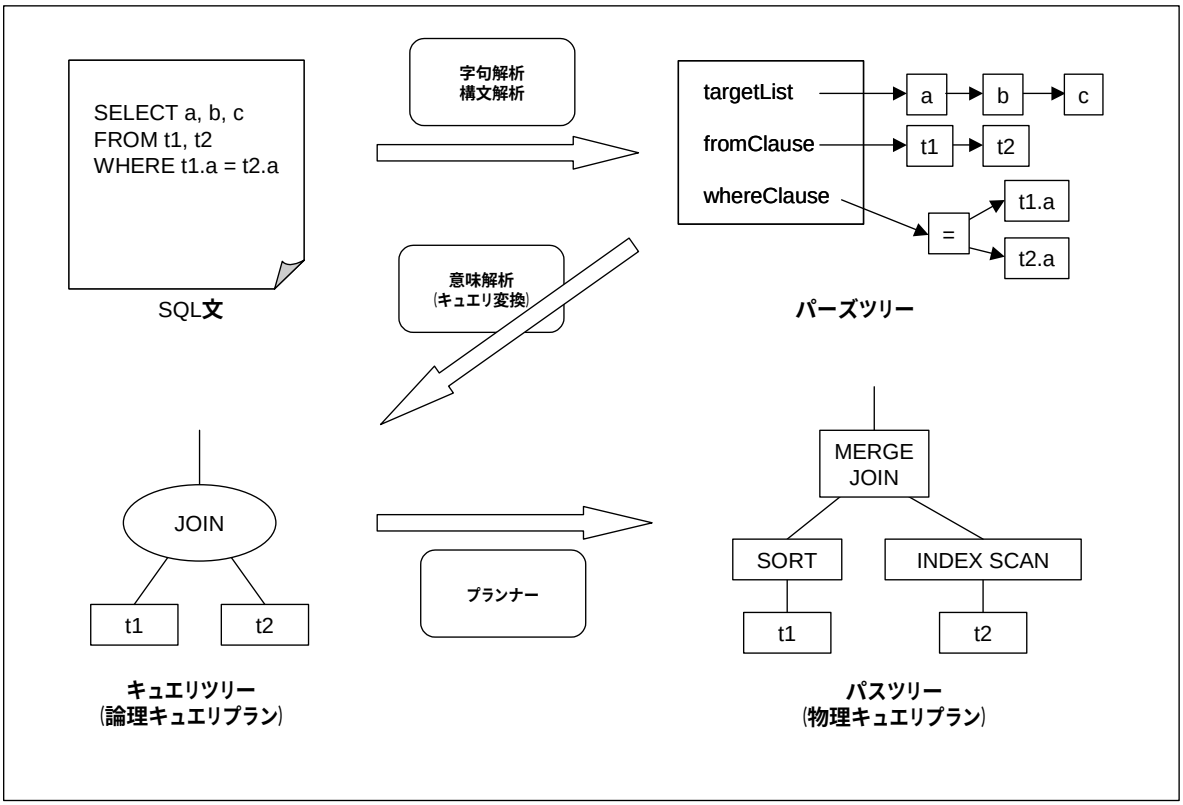

図 5 キュエリのデータの変化

2.2.3. トランザクションの管理

トランザクションマネージャのログ・リカバリの機能は、データベースが壊れた場合にデータベースを復旧する機能である。ログについては、この他にもトランザクションのロールバックにも使われる。ログの内容については、データ自体の変更やデータベースの構造の変更などが書き込まれている。ログに使われるログデータバッファは、通常のデータ用のバッファとは別に管理されている。
リカバリ処理とは、ログを読みながら、個々のログエントリの処理を再実行するものである。
有名な WAL(Write Ahead Logging) とは、ログの書き込み方式で、データバッファをディスクに書き込む前に、必ずその部分のログをディスクに書く方式である。
ちなみに、PostgreSQLのログ・リカバリのモジュールは、追記型の特性（データに対するUNDOログ[6]がいらない）により非常にシンプルに書かれている。

PostgreSQL の同時実行制御は、ロックとタイムスタンプを併用している。タイムスタンプと言っても、実際はトランザクション ID を使用する。DBMS の教科書的な話では、トランザクション ID もトランザクションの順序性を管理できるので、タイムスタンプという呼び方をする。PostgreSQL では、タイムスタンプと追記型のストレージの特性を利用して MVCC(多版型同時実行制御)を実現している。これは、あるレコードが更新中の場合、別のトランザクションは更新前の値を読み込むことで同時実行性を向上させる機能である。
ロック管理のモジュールはロックテーブルを管理している。ロックテーブルを用いるロックの他に、スピンロックも実装されている。

[6] データを更新前の状態に戻すためのログのこと。データを更新後の状態にするためのログをREDO ログと呼ぶ。

3. PostgreSQL のソースツリー

PostgreSQL 7.3.3 のソースツリー直下には、次のようなファイルが含まれる。

```
poseidon(2)% cd postgresql-7.3.3
poseidon(3)% ls -F
COPYRIGHT       INSTALL         aclocal.m4      configure.in    register.txt
GNUmakefile.in  Makefile        config/         contrib/        src/
HISTORY         README          configure*      doc/
poseidon(4)%
```

コンパイルする際は、ここから、configure[7] を実行し、make することになる。詳しくは INSTALL ファイルを参照して頂きたい。
ソースコードについては、src の下にある。本資料では、この下の特に backend に絞って説明していく。

3.1. src 配下

以下のファイル、およびディレクトリが src 配下に含まれるものである。

DEVELOPERS		FAQ が pgsql/doc/FAQ_DEV にあり、開発支援ツールが pgsql/src/tools にあるというメモ
Makefile		各サブディレクトリに対して、make の実行を引き継がせる定義が書いてあるだけ。
Makefile.global		make の suffix などのルールを書いてあるファイル。Makefile などから読み込まれる。configure 実行時に **Makefile.global.in から**作成される
Makefile.global.in		**Makefile.global の元ファイル。**
Makefile.port		プラットフォームごとの定義の書かれた makefiles 以下のファイルへのシンボリックリンク。Makefile.global から include される。configure 実行時に作成される。
Makefile.shlib		Shared ライブラリ作成のルールが書かれた Makefile。Shared ライブラリが使えるプラットフォームの Makefile で include される。
nls-global.mk		NLS のルールが定義されているファイル。Makefile.global から include される。
win32.mak		Windows 用の makefile。ココには、ODBC などでクライアントとしてアクセスするのに必要な部分しか含まれていない。
backend/		PostgreSQL のバックエンドの本体が入っているディレクトリ

7 ソースコードをコンパイルする際に、一番はじめに実行してコンパイル環境に合わせたMakefileなどを作成するコマンド。

bin/		インストール後の bin ディレクトリ以下に納められるツール類のスクリプトやソースを格納したディレクトリ。
corba/		CORBA I/F
data/		charset の変換テーブルのサンプル (日本語では使えない？)
include/		C でクライアントを書く際に必要な include ファイルを集めたディレクトリ
interfaces/		各種 API の実装。ecpg, jdbc, libpgtcl, libpq, libpq++, odbc, perl5, python などが実装されている。
makefiles/		Makefile 中の各プラットフォームの固有の定義。Makefile.port としてシンボリックリンクを張って、Makefile.global に読み込まれる。
pl/		Stored Procedure 用の言語の実装。
template/		各種プラットフォーム用のコンパイルの引数のテンプレート
test/		コンパイル後のテストのためのコード類
tools/		PostgreSQL 開発者向けのツール
tutorial/		チュートリアルとしてのサンプルコード
utils/		ユーティリティ。現在、DLL 初期化ユーティリティのみ。

src 配下の主なディレクトリは、backend, include, interfaces である。
backend が PostgreSQL のサーバにあたる部分のコードである。これについては、後述する。
include には、ソースコードのうちアプリケーションプログラムでもインクルードする必要があるヘッダファイルが入っている。当然、backend, interfaces 配下のソースでも使用しているので、ソースコードを解析する際にはここも見る必要がある。
interfaces には、図 1 にも書いたような各種 API のソースが入っている。
ecpg, jdbc, libpgtcl, libpq, libpq++, odbc, perl5, python などである。
libpq については、backend/libpq というものもあるが、ライブラリとしてのlibpqはinterfaces配下にある方である。backend配下のlibpq は、libpqプロトコル[8]のbackend 側の実装になる。

3.2. backend

以下の表が、backend ディレクトリ配下に含まれるモジュールである。

Makefile		
1 access/		
	1.1 common/	タプル操作、タプルのヘッダ操作の補助関数などが書いてある。
	1.2 gist/	GiST index(多次元インデックス)の実装。
	1.3 hash/	ハッシュインデックスの実装。
	1.4 heap/	heap の実装。catalog 配下の heap.c から呼ばれる。こちらが実ファイルアクセスに近い側。
	1.5 index/	index のインタフェース部分の実装。
	1.6 nbtree/	Btree インデックスの実装。

[8] libpq ライブラリを用いて通信する方法をPostgreSQL では、特に libpq プロトコルと呼んでいる。

	1.7 rtree/	２次元インデックス[9]R-treeの実装。
	1.8 transam/	トランザクションマネージャ(ログ管理部分)
2 bootstrap/		初期データベース構築時のモジュール
3 catalog/		カタログの実装
4 commands/		エグゼキュータのうち、コマンド(DML 以外の処理)部分の実装
5 executor/		エグゼキュータのうち、キュエリエグゼキュータの実装
6 lib/		内部で使用するユーティリティ的なライブラリ。
7 libpq/		サーバ側の libpq プロトコルの実装。主にユーザ認証など
8 main/		postmaster, postgres などのプログラムの main() 関数の実装
9 nodes/		SQL コンパイラで使用するノードオブジェクトの実装
10 optimizer/		SQL コンパイラのうち、オプティマイザの実装
	10.1 geqo/	Genetic Query Optimization (遺伝的キュエリ最適化)部分の実装
	10.2 path/	JOIN するときに、可能な全ての順序のパターンを生成する
	10.3 plan/	キュエリプランの生成を行う
	10.4 prep/	特殊なケースに対するプリプロセスフェーズの実装
	10.5 util/	オプティマイザ用の補助関数などの実装
11 parser/		キュエリコンパイラのうちスキャナ、パーザ、意味解析部分の実装
12 po/		メッセージカタログのようなもの。全体としてあまり使われていない？
13 port/		各プラットフォームの移植用コード
14 postmaster/		Postmaster の実装
15 regex/		正規表現関連の実装
16 rewrite/		SQL コンパイラのうち、リライターの実装
17 storage/		ストレージ全般を管理するモジュールを集めたディレクトリ
	17.1 buffer/	バッファ管理。ここで管理するのは、データ部分及びインデックスに使用するバッファだけであり、ログ用のバッファは、別に用意される。
	17.2 file/	仮想ファイルデスクリプタの実装。
	17.3 freespace/	ページ内 (テーブル内) の空スペースを管理する。
	17.4 ipc/	共有メモリの管理
	17.5 large_object/	blob など lo の実装。
	17.6 lmgr/	ロックマネージャ。
	17.7 page/	ページ中のデータ管理モジュール。レコード単位のデータの取り出しを実装。
	17.8 smgr/	実ストレージへの I/O を実装したモジュール。ディスクへのシステムコールなどを管理。
18 tcop/		PostgresMain などの実装が入っているところ。
19 utils/		
	19.1 adt/	データ型に関する実装。

[9] 地図データのようなものを扱わない限り、普通のトランザクション処理で使用されることはまずない。

	19.2 cache/	カタログデータ、リレーション、ファンクションのキャッシュを担当するモジュール。
	19.3 error/	エラーメッセージ出力に関する実装
	19.4 fmgr/	ファンクションマネージャ: ファンクションの実装。
	19.5 hash/	ハッシュ関数の実装
	19.6 init/	postgres または postmaster 起動時の各種初期化モジュール
	19.7 mb/	マルチバイト処理に関するモジュール
	19.8 misc/	実行時のパラメータを扱う GUC[10] やその他汎用モジュールがある場所。
	19.9 mmgr/	メモリマネージャ。階層的にメモリを管理できるモジュールを提供。
	19.10 sort/	ソートの実装。
	19.11 time/	MVCC の実装（タイムスタンプ型の同時実行制御）

図 2 のキュエリコンパイラにあたる部分のモジュールが、9 nodes, 10 optimizer, 11 parser, 16 rewriter である。11 parserは、SQL(その他のPostgreSQLのコマンドも含む) の字句解析[11]を行って構文解析[12]を行うモジュールと、意味解析[13]を行うモジュールからなる。
図 2 のキュエリエグゼキュータにあたるのが、4 commands と 5 executor である。

図 2 のトランザクションマネージャのうち、ログ・リカバリにあたるのが、1.8 transam 配下のモジュールである。また、ログ・リカバリのためのコードの一部は、ロギングの必要な各モジュールにも埋め込まれている。図 2 のトランザクションマネージャのうち、同時実行制御の部分が、17.6 lmgr と 19.11 time である。

図 2 のストレージ管理のインデックス／レコードマネージャにあたるのが、1 access 以下の transam 以外のモジュールと 17.7 page である。バッファマネージャは 17.1 buffer あたりで、ストレージマネージャは、17.2 file や 17.8 smgr あたりになる。

その他、図 2 の構造に出てこないもので、主要なモジュールには、次のようなものがある。
2 bootstrap:
データベースを初期化する際に使用するモジュールが含まれている。主にシステムカタログのデータを作成するために使用される。データベースの初期化時に読み込むファイルは、C のヘッダファイルからソースコードコンパイル時に自動生成される。ヘッダファイルから自動生成したファイルを読み込むことによって、ソースコード中に埋め込まれている OID と、データとしてシステムカタログに取り込まれる OID を一致されるための機構である。ファイルを読み込む機能のため、独自構文を利用している。しかし、SQL に比べてはるかに簡単なものなので、実装はきわめて小さい。

[10] Grand Unified Configuration Scheme の略。コンフィグで設定できるパラメータを扱う機構。
[11] SQL文からトークン(予約語やid、データ部分などの塊) を切り出す
[12] 構文規則に従って、パーズ木を生成する
[13] テーブル名や属性名が実際にＤＢに定義されているかをチェックしたりする

3 catalog:
システムカタログを実装したモジュール。システムのメタ情報を保持しているのだが、PostgreSQL の場合、この部分を使って、データ型や関数、その他いろいろなものを定義できるようにしてある。

8 main:
PostgreSQL のサーバプログラムのエントリポイントである。PostgreSQL をコンパイルすると、postgres, postmaster というの２つのプログラムがあるが、これらはファイル名が異なるだけで同一のプログラムである。main には、これらのプログラムが最初に呼び出す main() 関数が含まれている。

14 postmaster:
main から postmaster として呼び出されたときの実装。クライアントからの接続を待ち、接続があった場合に子プロセスを fork する。

18 tcop:
main から postgres で呼び出されたとき、および postmaster から fork されたあと呼び出される PostmasterMain() の実装が入っている。

6 lib:
ライブラリとして使用するようなユーティリティ的ものが入っているディレクトリ。

7 libpq:
サーバ側の libpq プロトコルの実装。主にユーザ認証などである。

19 utils:
汎用モジュールのようにまとめてあるが、データ型に関する実装である 19.1 adt、リレーションやカタログデータなどのキャッシュを行う 19.2 cache、メモリ管理を行う 19.9 mmgr などの非常に重要なモジュールも含まれている。

4. ソースコードの流れ

ここでは、ソースコードを読み始める糸口として、C プログラムの先頭である main()からの流れを紹介する。
PostgreSQL のサーバを上げるためのコマンド postmaster および スタンドアロンで DBMS を実行するためのコマンド postgres は、main/main.c の main()関数から始まる。

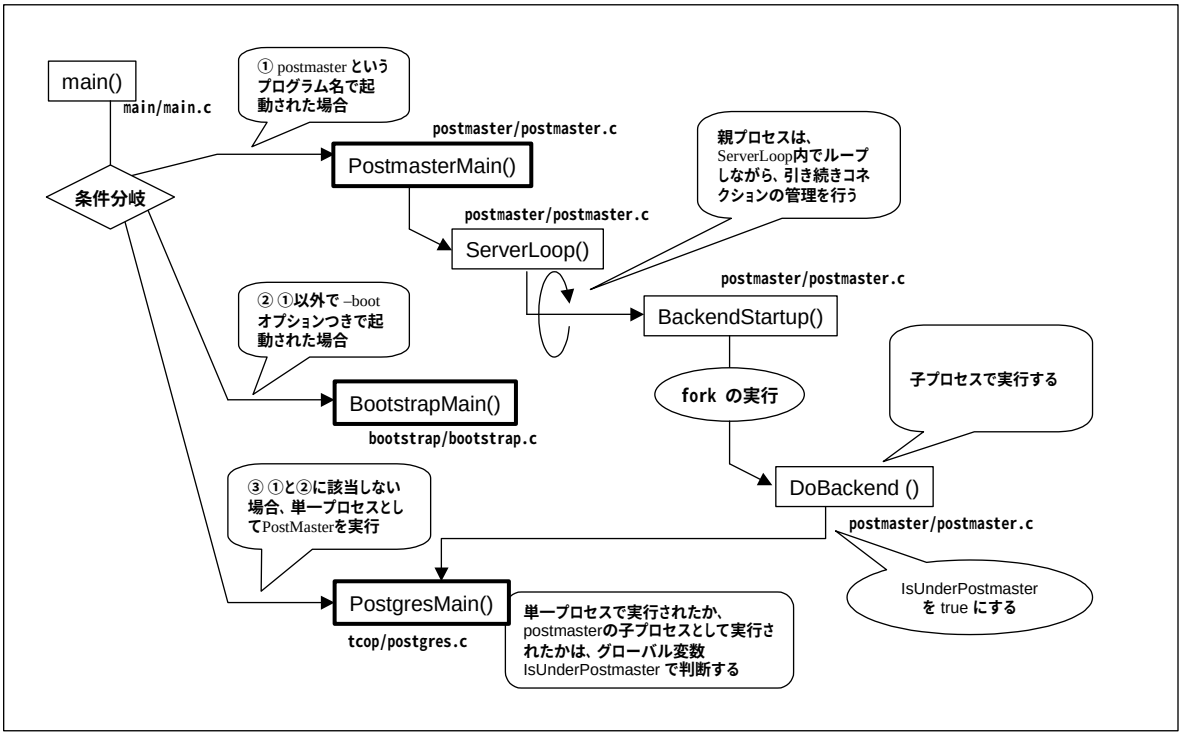

図 6. main() 関数からの流れ

main()関数の中で、postmaster という名前で呼ばれていたら、postmaster として起動する（①のルート）。-boot オプション付で呼ばれると、BootstrapMain という関数を呼び出し初期化専用プロセスとして動作する（②のルート）。その他の場合は、スタンドアロンの PostgreSQL のデータベースアクセスプログラムとして動作する（③のルート）。
コマンドラインから postgres として呼び出されたときの動作は、同時実行制御を省略していることとプロセスの初期化を行うこと以外は、postmaster から fork されている postgres と同じである。コード上はまったく同じ物で、コード内で postmaster から fork されたかどうかで、処理の分岐が若干入っている。

PostgresMain() 以降の処理は、図７のようになる。図４のキュエリ処理の流れが左側に示してあり、それぞれのエントリポイントが分かると思う。

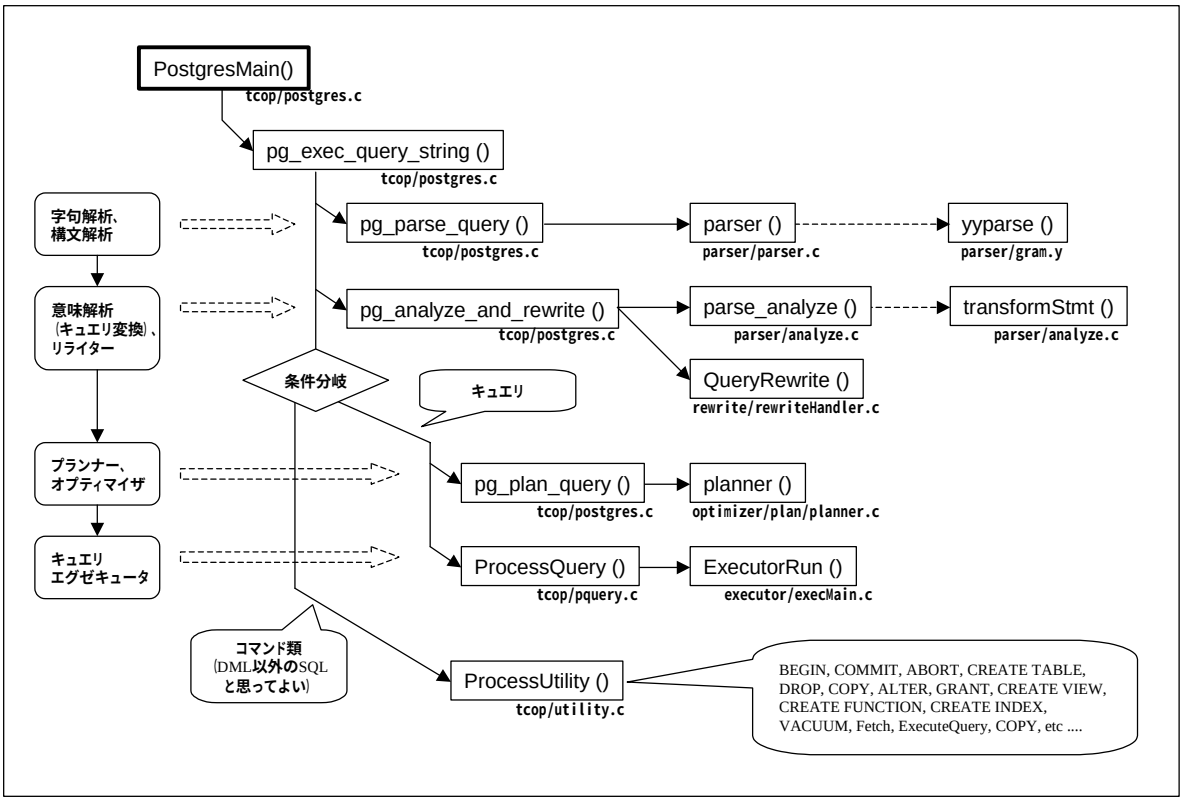

図 7. PostgresMain() 関数からの主な処理の流れ

以上が、これまでに紹介した DBMS のアーキテクチャのエントリポイントとなる。これらの関数から PostgreSQL の backend の処理の流れを追うことができると思う。

＜EOF＞