

Python製RDBMSで理解する、データベースのピース

～コードのステップ実行とヘックスビュアーで内部動作を追ってみよう～

PyCon JP 2025

@ k-kamijo

自己紹介

名 前 : k-kamijo

Github : @kei-kmj

所 属 : 株式会社DeltaX

趣 味 : 輪行
(電車旅 & サイクリング)



全国103,457教室から
ぴったりの塾が見つかる

塾選びから始まる、合格への第一歩

学習塾検索サイト

リアルな口コミ多数！

合格体験記も充実！



□ 塾名・志望校名から探す

Q 塾名、志望校名を入力

📍 全国の塾・予備校を探す

中国・四国

鳥取県 | 島根県 | 岡山県 | 広島県
山口県 | 徳島県 | 香川県 | 愛媛県
高知県

信越・北陸

新潟県 | 富山県 | 石川県 | 福井県
山梨県 | 長野県

北海道・東北

北海道 | 青森県 | 岩手県 | 宮城県
秋田県 | 山形県 | 福島県

九州・沖縄

福岡県 | 佐賀県 | 長崎県 | 熊本県
大分県 | 宮崎県 | 鹿児島県
沖縄県

関西

滋賀県 | 京都府 | 大阪府 | 兵庫県 | 奈良県 | 和歌山县

関東

茨城県 | 栃木県 | 群馬県 | 埼玉県
千葉県 | 東京都 | 神奈川県

東海

岐阜県 | 静岡県 | 愛知県 | 三重県

リアルな口コミで選ぶなら

享
丸
記
選
ブル



ジュクセン

はじめに

データベースとは

データを効率的に保存・検索・更新するためのシステム

データベースの種類

- 🍋 RDBMS → PostgreSQL, MySQL
- 🍋 NoSQL → MongoDB, Redis
- 🍋 グラフDB → Neo4j

RDBMS(Relational Database Management System)

- 🍋 データをリレーションナルなテーブルで管理
- 🍋 SQLで操作

自作RDBMS : KeiPyDBの紹介

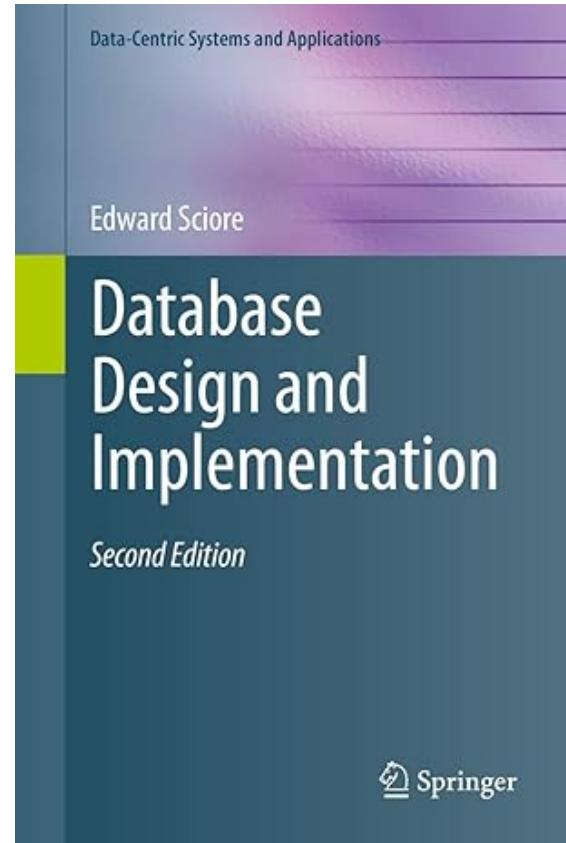
<https://github.com/kei-kmj/KeiPyDB>

「Database Design and Implementation : Second Edition」 Edward Sciore (著)

JavaでRDBMSを実装していく教科書っぽい洋書



Pythonで、標準ライブラリのみを使って実装



KeiPyDBの機能

SQL

- 🍋 CREATE TABLE
- 🍋 INSERT
- 🍋 SELECT
- 🍋 UPDATE
- 🍋 DELETE
- 🍋 WHERE
- 🍋 CROSS JOIN

機能

- 🍋 トランザクション
- 🍋 ハッシュインデックス
- 🍋 B-treeインデックス

話すこと

💡 SQLの1行の裏側で、たくさんの仕組みが動いていること

- CREATE TABLE
- INSERT INTO
- SELECT
- DELETE

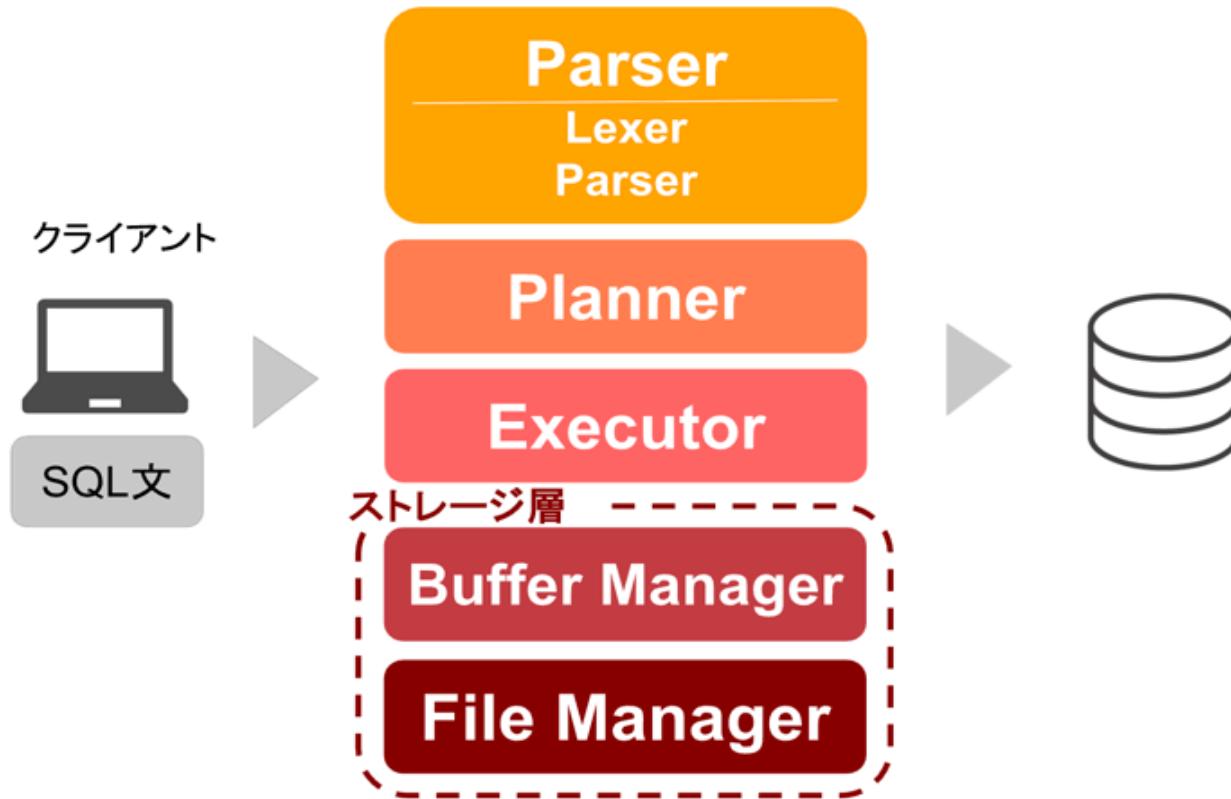
を使って、コードを追っていきます

話ないこと

- ✖ 特定のデータベース製品の性質や使い方
- ✖ SQL文の書き方
- ✖ インデックス戦略
- ✖ パフォーマンスチューニング
- ✖ テーブル設計や正規化

RDBMSのアーキテクチャ

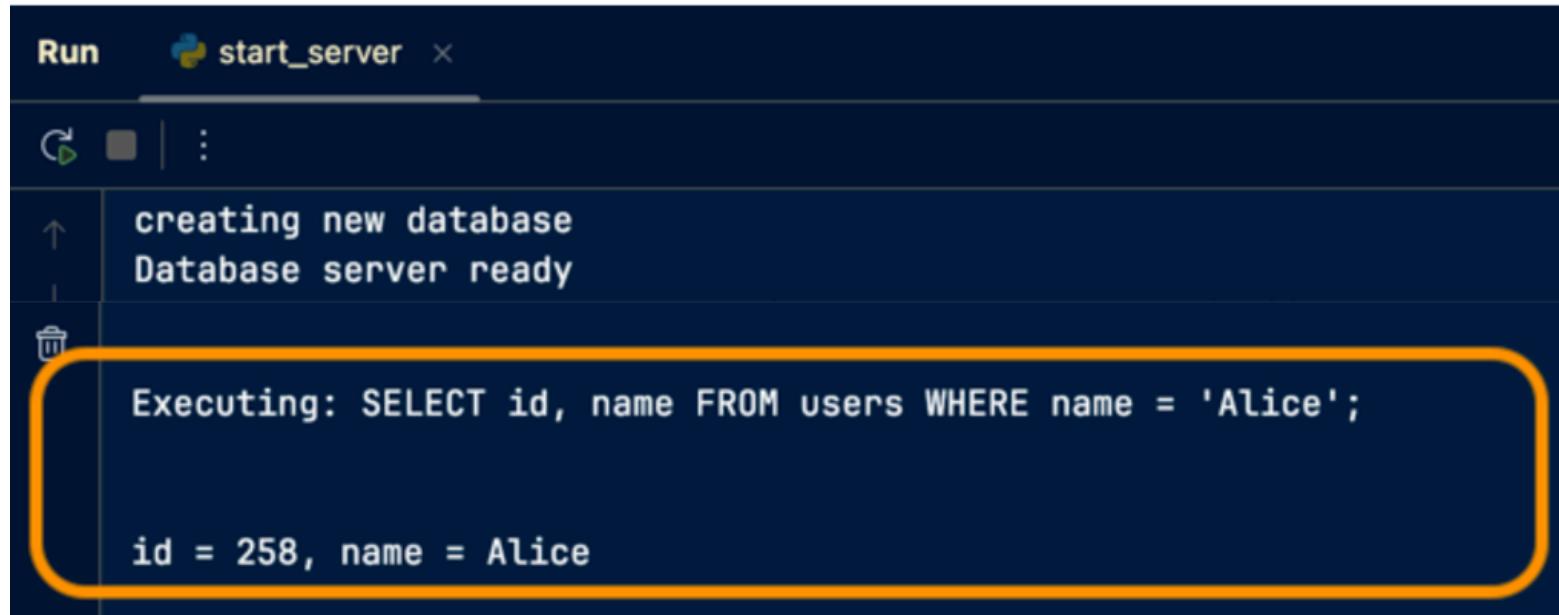
RDBMSのアーキテクチャ



SELECT 文

SELECT 文

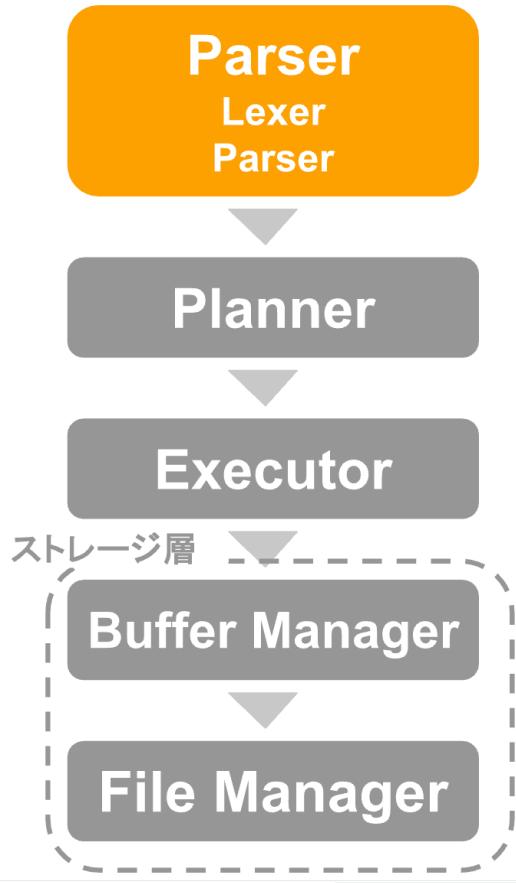
```
SELECT id, name FROM users WHERE name = 'Alice'
```



The screenshot shows a terminal window with a dark blue background. At the top, there is a tab bar with 'Run' and 'start_server'. Below the tabs are some icons: a play button, a square, and a vertical ellipsis. The main area of the terminal displays the following text:

```
↑ creating new database
Database server ready
Executing: SELECT id, name FROM users WHERE name = 'Alice';
id = 258, name = Alice
```

The line 'Executing: SELECT id, name FROM users WHERE name = 'Alice';' is highlighted with a thick orange rounded rectangle.



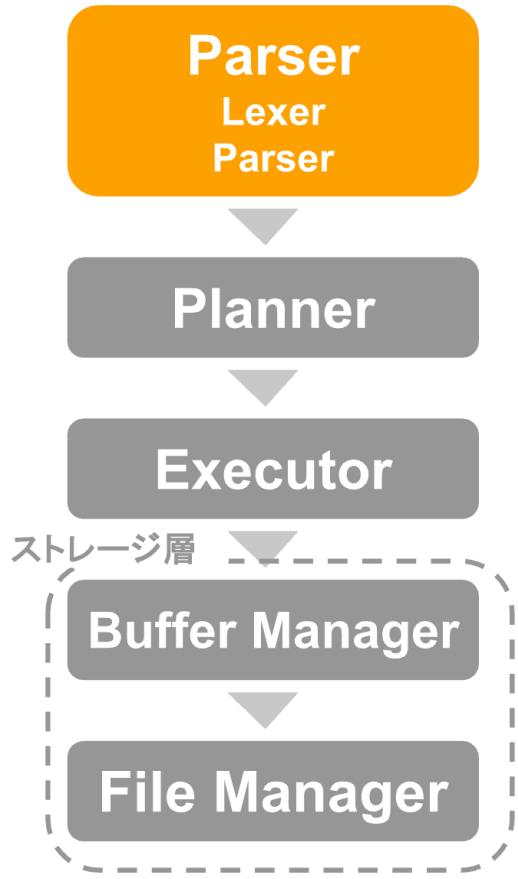
Lexer (字句解析)

SQLをトークン（意味のある最小単位）に分解する

```
SELECT id, name FROM users WHERE name = 'Alice'
```



```
SELECT id , name FROM users WHERE name = 'Alice'
```



Lexer (字句解析)

分解したトークンの種類を判定して分類

トークン	種類 (Lexerが判定)
SELECT , FROM , WHERE	キーワード (予約語)
id , name , users	識別子
,	デリミタ = 区切り文字
=	演算子
'Alice'	文字列リテラル

SELECT文の実行

```
class StartServer:

    @staticmethod
    def main() -> None:
        # SELECT
        tx_select = db.new_transaction()
        select_sql = "SELECT id, name FROM users WHERE name = 'Alice';"
        print(f"Executing: {select_sql}")
        plan = planner.create_query_plan(select_sql, tx_select)
        scan = plan.open()
```

Lexer(字句解析) - 初期化

```
class Lexer:
    def __init__(self, sql: str) → None:
        """SQL文を解析するための字句解析器"""
        self.keywords = {
            "select",
            "from",
            "where",
            "and",
            "insert",
            "into",
            "values",
            "delete",
```

Lexer(字句解析) - トークン分割

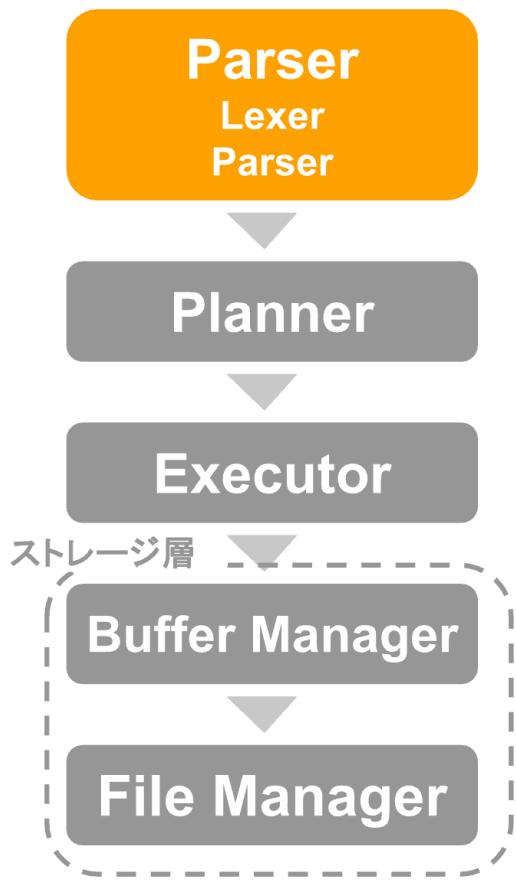
```
class Lexer:  
    def _tokenize(self, sql: str) → list[str]:  
        token_pattern = (  
            r"[a-zA-Z_][a-zA-Z_0-9]*" # 識別子  
            r"|[?:[^']|'']*\'''" # 文字列リテラル  
            r"\d+(?:\.\d+)??" # 数値（整数・小数）  
            r"[=,(),<>*+-/;]" # 演算子・区切り文字  
            r"\s+" # 空白文字  
            r"." # その他の1文字  
        )  
        # マッチするものをすべて抽出  
        token_list = re.findall(token_pattern, sql)
```

Lexer(字句解析) - キーワード(予約語)の判定

```
class Lexer:  
    def match_keyword(self, keyword: str) → bool:  
        """指定されたキーワードと現在のトークンが一致するかどうかを返す"""  
  
        return self.current_token.lower() == keyword.lower()  
  
    def eat_keyword(self, keyword: str) → None:  
        """指定されたキーワードを認識して次のトークンに進む"""  
        if self.current_token.lower() ≠ keyword.lower():  
            raise SyntaxError  
  
        self.next_token()
```

Lexer(字句解析) - 識別子の判定

```
class Lexer:  
    def match_id(self) → bool:  
        """現在のトークンが識別子かどうかを返す"""  
        return (self.current_token.isidentifier() and  
                self.current_token not in self.keywords)  
  
    def eat_id(self) → str:  
        """識別子を認識して次のトークンに進む"""  
        identifier = self.current_token  
        self.next_token()  
        return identifier
```



Parser (構文解析)

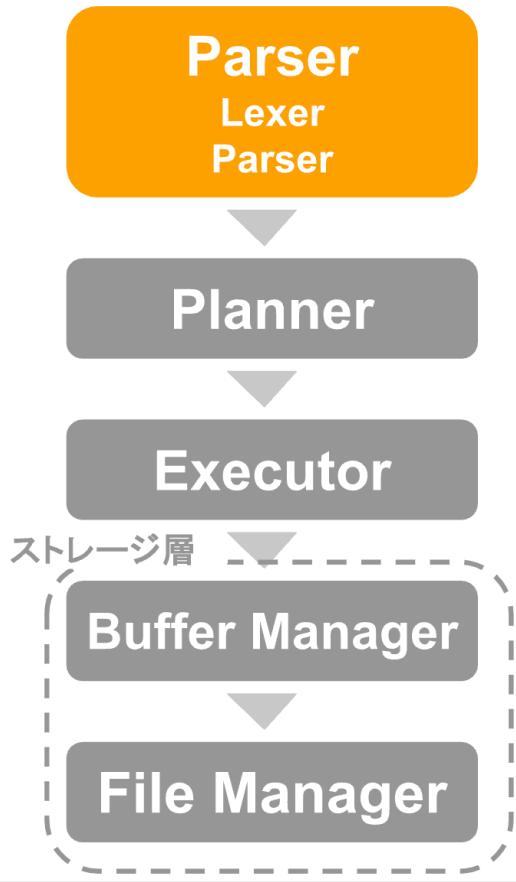
トークン列をコンピュータが理解できるよう構造化する

```
SELECT id , name FROM users WHERE name = 'Alice'
```



ASTを構築

```
QueryData
└─ SELECT: [id, name]
└─ FROM: users
└─ WHERE: (name = 'Alice')
```



Parser - 複雑な条件

```
WHERE price < 1000 AND  
( item = '牡蠣せんべい' OR item = 'もみじ饅頭' )
```



```
condition → AND [ price < 1000 ]  
                  |  
                  OR [ item = '牡蠣せんべい'  
                         |  
                         item = 'もみじ饅頭'
```

Parser(構文解析) - 構文解析の開始

```
class Planner:  
    def create_query_plan(self, query: str, tx: TX) → Plan:  
  
        parser = Parser(query)  
        parsed_query = parser.query()  
  
        self.verify_query()
```

Parser(構文解析) - query()メソッド

```
class Parser:  
    def query(self) → QueryData:  
        self.lexer.eat_keyword("select")  
        field_list = self.select_list()  
  
        self.lexer.eat_keyword("from")  
        table_list = self.table_list()
```

Parser(構文解析) - フィールドリストの作成

```
class Parser:  
    def select_list(self) → list[str]:  
        field_list = [self.field()]  
        while self.lexer.match_delimiter(", ", " "):  
            self.lexer.eat_delimiter(", ", " ")  
            field_list.append(self.field())  
        return field_list
```

Parser(構文解析)- query()メソッドの続き

```
class Parser:  
    def query(self) → QueryData:  
        self.lexer.eat_keyword("select")  
        field_list = self.select_list()  
        self.lexer.eat_keyword("from")  
        table_list = self.table_list()  
  
        predicate = Predicate()  
        if self.lexer.match_keyword("where"):  
            self.lexer.eat_keyword("where")  
            predicate = self.predicate()
```

Parser(構文解析) - テーブルリストの作成

```
class Parser:  
    def table_list(self) → Collection[str]:  
        table_name = self.lexer.eat_id()  
        table_list: list[str] = [table_name]  
  
        while self.lexer.match_delimiter(", "):  
            self.lexer.eat_delimiter(", ")  
            table_name = self.lexer.eat_id()  
            table_list.append(table_name)  
  
    return table_list
```

Parser(構文解析) - query()メソッドの続き

```
class Parser:  
    def query(self) → QueryData:  
        self.lexer.eat_keyword("select")  
        field_list = self.select_list()  
        self.lexer.eat_keyword("from")  
        table_list = self.table_list()  
  
        predicate = Predicate()  
        if self.lexer.match_keyword("where"):  
            self.lexer.eat_keyword("where")  
            predicate = self.predicate()  
  
        return QueryData(field_list, table_list, predicate)
```

Parser(構文解析) - 条件式の作成

```
class Parser:  
    def predicate(self) → Predicate:  
        predicate = Predicate([self.term()])  
        if self.lexer.match_keyword("and"):  
            self.lexer.eat_keyword("and")  
            predicate.conjoin_with(self.predicate())
```

Parser(構文解析) - 条件式(単体)の作成

```
class Parser:  
    def term(self) → Term:  
        left = self.expression()  
        self.lexer.eat_delimiter("=".  
        right = self.expression()  
        return Term(left, right)  
  
    def expression(self) → Expression:  
        if self.lexer.match_id():  
            return Expression(self.field())  
        else:  
            return Expression(self.constant())
```

Parser(構文解析) - 複数条件の連結

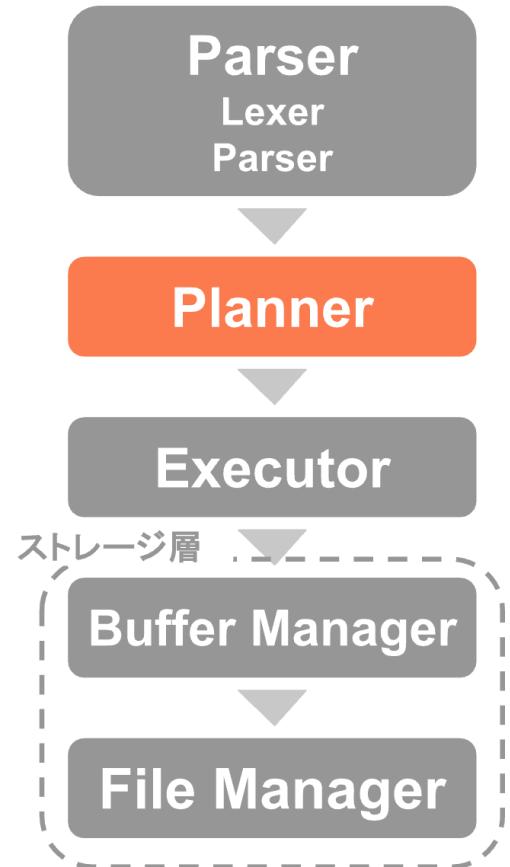
```
class Parser:  
    def predicate(self) → Predicate:  
        predicate = Predicate([self.term()])  
        if self.lexer.match_keyword("and"):  
            self.lexer.eat_keyword("and")  
            predicate.conjoin_with(self.predicate())
```

Parser(構文解析) - 最後にASTを返す

```
class Parser:  
    def query(self) → QueryData:  
        self.lexer.eat_keyword("select")  
        field_list = self.select_list()  
        self.lexer.eat_keyword("from")  
        table_list = self.table_list()  
        predicate = Predicate()  
        if self.lexer.match_keyword("where"):  
            self.lexer.eat_keyword("where")  
            predicate = self.predicate()  
  
        return QueryData(field_list, table_list, predicate)
```

Parser(構文解析) - ASTの確認

```
: Threads & Variables  Console  
> plan_list = {list: 0} []  
  query_data = {QueryData} SELECT id, name FROM users WHERE name = Alice  
  > fields = {list: 2} ['id', 'name']  
  > predicate = {Predicate} name = Alice  
  > tables = {list: 1} ['users']  
  > Protected Attributes  
> self = {BasicQueryPlanner} <db.plan.basic_query_planner.BasicQueryPlanner object at 0x1049f3d70>  
> transaction = {Transaction} <db.transaction.transaction.Transaction object at 0x104841ee0>
```



Planner (実行計画)

ASTを受け取って、実行方法を選択

```
ProjectPlan(  
    fields=['id', 'name'],  
    SelectPlan(  
        predicate="name='Alice'",  
        TablePlan('users')  
    )  
)
```

Planner(実行計画) - 作成開始

```
class Planner:  
    def create_query_plan(self, query: str, tx: TX) → Plan:  
        """クエリを実行するための計画を作成する"""  
        parser = Parser(query)  
        parsed_query = parser.query()  
  
        self.verify_query()  
  
        return self.query_planner.create_plan(parsed_query, tx)
```

Planner(実行計画) - create_plan()メソッド

```
class BasicQueryPlanner(QueryPlanner, ABC):
    def create_plan(self, query: Query, tx: TX) → Plan:

        plan_list: list[Plan] = []
        plan_list.append(TablePlan(tx, table_name, self.mdm))

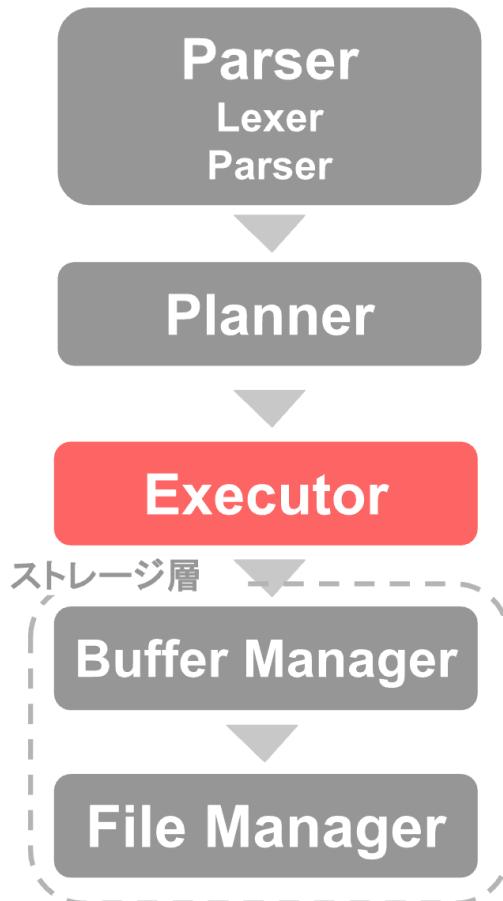
        plan = SelectPlan(plan, query.get_predicate())

        field_list = query.get_fields()
        return ProjectPlan(plan, field_list)
```

Planner(実行計画) - 実行計画の確認

```
Threads & Variables   Console

self = {ProjectPlan} <db.plan.project_plan.ProjectPlan object at 0x1031f2060>
  plan = {SelectPlan} <db.plan.select_plan.SelectPlan object at 0x104371e80>
    plan = {TablePlan} <db.plan.table_plan.TablePlan object at 0x1043bcc50>
      <protected> {Protected} names = None
      <protected> Protected Attributes
      <protected> Protected Attributes
```

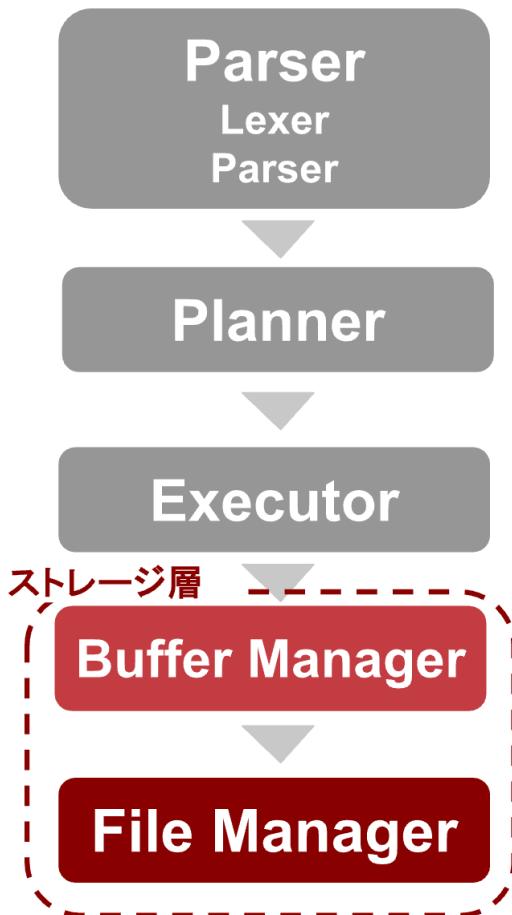


Executor (実行エンジン)

- ① Plan (実行計画) を受け取る
- ② Scan (実行オブジェクト) を作る

```
TableScan('users')          # テーブルから1行ずつ読む  
SelectScan("name='Alice'") # 条件に合うか確認  
ProjectScan(['id', 'name']) # 必要なカラムだけ取り出す
```

- ③ レコードを返す
- ⚠ 実行エンジンのしくみを理解するには
ストレージ層の理解が必要



ストレージ層の概要

担当

🍋 Buffer ManagerとFile Manager

解決したい課題

🍋 大量のデータを扱いたい ↔ 高速に処理したい

ディスクアクセスはメモリアクセスに比べて非常に遅い

役割

🍋 RDBMS自身でメモリ管理(OSにまかせない)

🍋 データの永続化

ストレージ



ブロック

スロット 0

スロット 1

スロット 2

用語



ストレージとのやりとりに用いる、

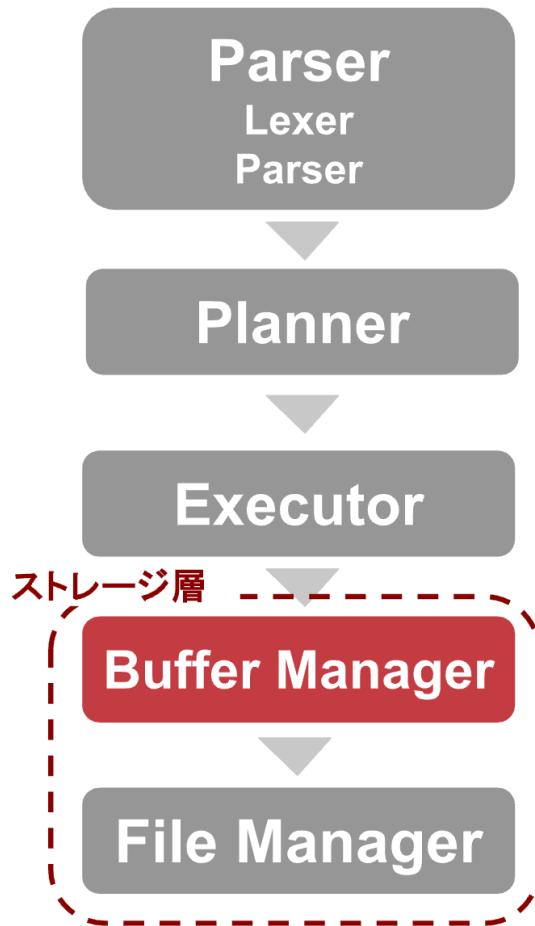
固定長のデータ単位 (KeiPyDBでは400バイト)



ブロック内でレコードを保存する固定長の領域
テーブル情報でサイズとレイアウトが決まる

スロット 0

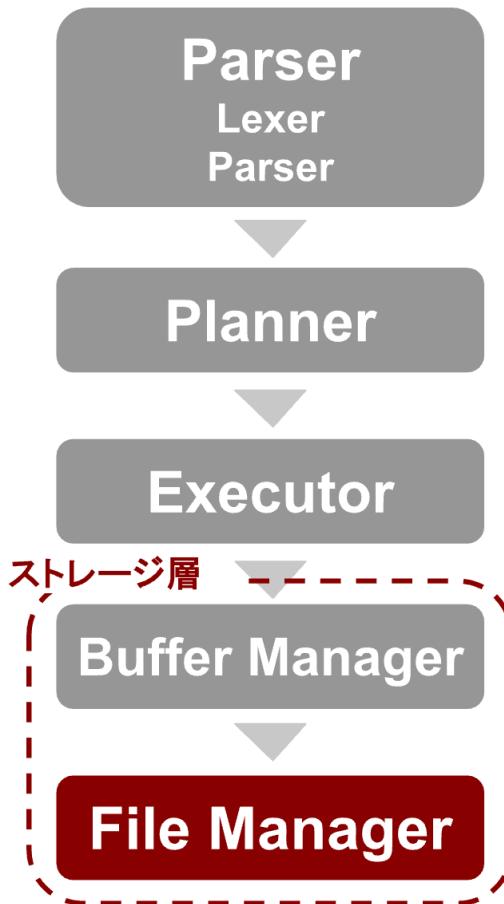
レコード



Buffer Manager

RDBMS自身でメモリ管理(OSにまかせない)

- 頻繁にアクセスされるブロックをメモリに保持
- 使用頻度が低いブロックのメモリからの追い出し



File Manager

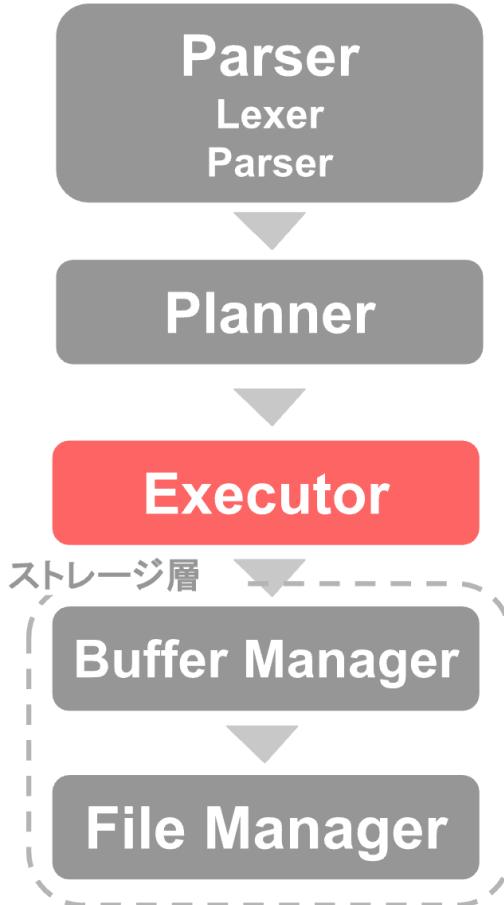
データの永続化

🍋 OSのファイルシステムとのやりとり

🍋 ストレージのデータをブロックの単位で読み書き

DeltaX





レコード取得のしくみ 1

```
ProjectPlan(  
    fields=['id', 'name'],  
    SelectPlan(predicate="name='Alice'",  
               TablePlan('users')))
```

- 🟡 **users**テーブルのブロックをストレージ層から
(トランザクションを介して) 1つずつ読む
- 🟡 ブロックの中をスロット単位で探す

スロットのサイズとレイアウト

- サイズ/レイアウトはテーブル情報で決まる

```
CREATE TABLE users (id int, name varchar(10))
```



スロットの最初の領域は状態フラグ

01:使用中、00:空き

状態フラグ 4 Bytes	id 4 Bytes	文字サイズ 4 Bytes	文字列 10 Bytes
01	258	5	Alice

レコード取得のしくみ 2

- 🍋 スロットのサイズとレイアウトがわかれれば、レコードにアクセス可能
- 🍋 スロット毎に、13バイト目から10バイト分を確認
- 🍋 nameがAliceかどうかの確認
- 🍋 レコードを取得して指定されたカラムを出力 { id: 258, name: 'Alice' }

状態フラグ 4 Bytes	id 4 Bytes	文字サイズ 4 Bytes	文字列 10 Bytes
01	258	5	Alice

4 8 12 22

Executor(実行エンジン) - レコード取得の開始

```
class StartServer:

    @staticmethod
    def main() → None:
        # SELECT
        tx_select = db.new_transaction()
        select_sql = "SELECT id, name FROM users WHERE name = 'Alice';"

        plan = planner.create_query_plan(select_sql, tx_select)
        scan = plan.open()
```

Executor(実行エンジン) - レコードの取得

```
class TableScan(UpdateScan, ABC):
    def get_value(self, field_name: str) → Constant:
        """現在のスロットの指定されたフィールドの値を返す"""

        field_type = self.layout.get_schema().get_type(field_name)

        if field_type == FieldType.Integer:
            return Constant(self.get_int(field_name))
        elif field_type == FieldType.Varchar:
            return Constant(self.get_string(field_name))
        else:
            raise ValueError(f"Unknown field type {field_type}")
```

Executor(実行エンジン) - 文字列型の値の取得

```
class TableScan(UpdateScan, ABC):
    def get_string(self, field_name: str) → str:
        """現在のスロットの指定されたフィールドの文字列を返す"""

        if self.record_page is None:
            raise RuntimeError

        slot = self.current_slot
        return self.record_page.get_string(slot, field_name)
```

Executor(実行エンジン) - 条件式の判定

```
class SelectScan(UpdateScan, ABC):
    def next(self) → bool:
        while self.scan.next():
            if self.predicate.is_satisfied(self):
                return True
        return False
```

Executor(実行エンジン) - 指定したカラムの値を取得

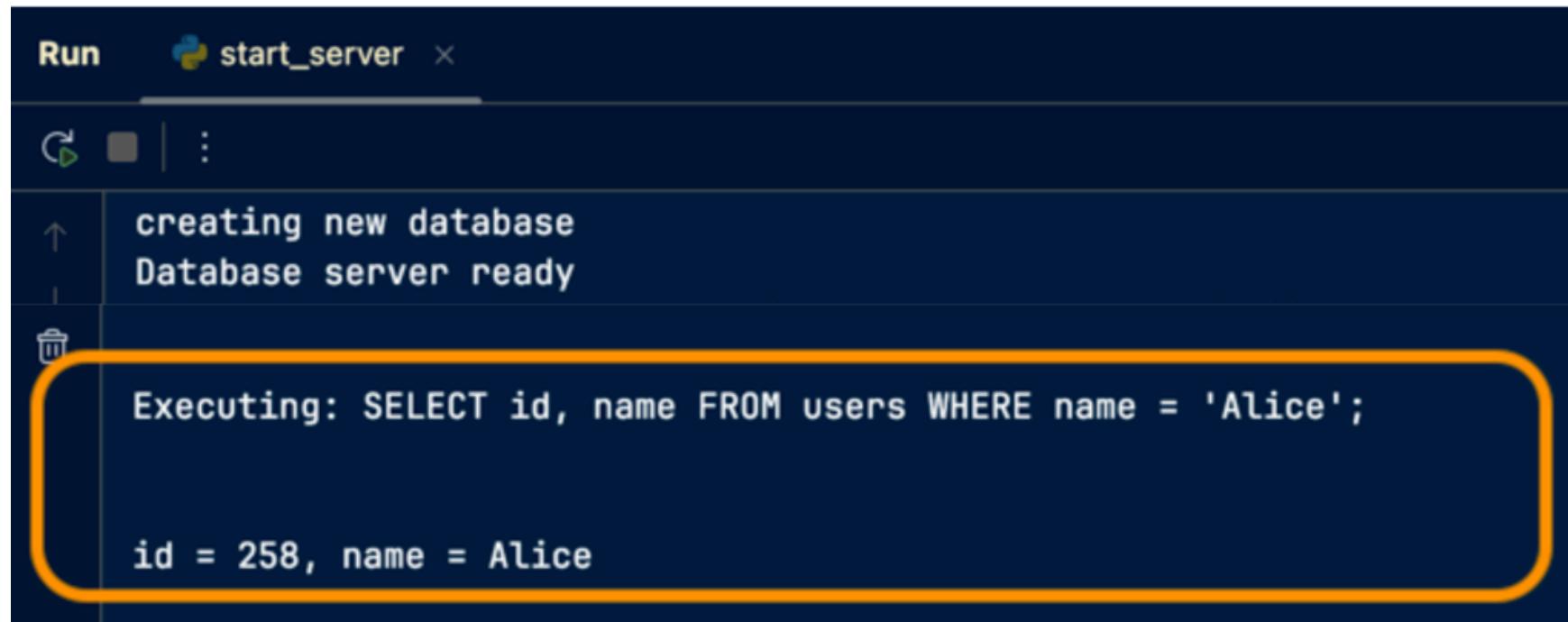
```
class ProjectScan(Scan, ABC):
    def get_value(self, field_name: str) → Constant:

        if self.has_field(field_name):
            return self.scan.get_value(field_name)
        else:
            raise RuntimeError
```

Executor(実行エンジン) - 処理終了

```
class StartServer:  
    @staticmethod  
    def main() → None:  
        # SELECT  
        tx_select = db.new_transaction()  
        select_sql = "SELECT id, name FROM users WHERE name = 'Alice';"  
        plan = planner.create_query_plan(select_sql, tx_select)  
        scan = plan.open()  
  
        scan.close()  
        tx_select.commit()
```

実行結果の確認



The screenshot shows a terminal window with the following content:

- Top bar: Run, start_server, X
- Icons: Refresh, Stop, More
- Logs:
 - ↑ creating new database
 - ↑ Database server ready
- Query execution:
 - Executing: `SELECT id, name FROM users WHERE name = 'Alice';`
 - Result:
`id = 258, name = Alice`

A yellow rounded rectangle highlights the "Executing" line and its result.

```
Run start_server X
↑ creating new database
↑ Database server ready
Executing: SELECT id, name FROM users WHERE name = 'Alice';
id = 258, name = Alice
```

INSERT 文 & DELETE 文

レコードの追加(INSERT)

```
INSERT INTO users (id, name) VALUES (259, 'Bob')
```

🍋 状態フラグが00のスロットを探す

🍋 Write-Ahead Logging/
ログを先に書き込んで障害対策

🍋 (トランザクションを介して)
空きスロットに書き込む

状態フラグ 4 Bytes	id 4 Bytes	文字サイズ 4 Bytes	文字列 10 Bytes
01	258	5	Alice
00			



ここに追加

レコードの追加(INSERT)

```
class BasicUpdatePlanner(UpdatePlanner, ABC):
    AFFECTED = 1

    def execute_insert(self, data: InsertData, tx: TX) → int:
        fields = data.get_fields()
        values = iter(data.get_values())

        for field_name in fields:
            value = next(values)
            scan.set_value(field_name, value)

        scan.close()
        return self.AFFECTED
```

FileManager - ディスク書き込み

```
class FileManager:  
    def write(self, block: BlockID, page: Page) → None:  
        """ブロックIDに対応するファイルにデータを書き込む"""  
        try:  
            f = self._get_file(block.file_name)  
            f.seek(block.block_number * self.block_size)  
            f.write(page.buffer)  
            f.flush()  
            os.fsync(f.fileno()) # 確実にディスクに書き込み
```

レコードの削除(DELETE)

```
DELETE FROM users WHERE id = 259
```

- 🍋 データはすぐには消さない
- 🍋 (トランザクションを介して)
 - スロットの状態フラグを00に更新
 - データはそのまま残る
- 🍋 後でレコードを追加する時に
 - 空きスロットとして再利用される

状態フラグ 4 Bytes	id 4 Bytes	文字サイズ 4 Bytes	文字列 10 Bytes
01	258	5	Alice
00	259	3	Bob

レコードの削除

```
class RecordPage:  
    EMPTY = 0  
    USED = 1  
  
    def delete(self, slot: int) → None:  
        """状態フラグを空にする"""  
        self._set_flag(slot, RecordPage.EMPTY)  
  
    def _set_flag(self, slot: int, flag: int) → None:  
        """状態フラグの書き込み"""  
        self.tx.set_int(self.block, self._offset(slot), flag, True)
```

DELETE文の実行結果の確認



```
Executing: INSERT INTO users (id, name) VALUES (259, 'Bob');
```

```
Executing: SELECT id, name FROM users
```

```
id = 258, name = Alice  
id = 259, name = Bob
```



```
Executing: DELETE FROM users WHERE id = 259;
```

```
Executing: SELECT id, name FROM users  
id = 258, name = Alice
```

ヘックスビューアーで確認

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
000	01	00	00	00	02	01	00	00	05	00	00	00	41	6C	69	63
010	65	00	00	00	00	00	00	00	00	00	03	01	00	00	03	00
020	00	00	42	6F	62	00	00	00	00	00	00	00	00	00	00	00
030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Alic
e
Bob
Bob

状態フラグ 4 Bytes	id 4 Bytes	文字サイズ 4 Bytes	文字列 10 Bytes
01	258	5	Alice
00	259	3	Bob

4

8

12

22

int型の並び順が逆

10進数	16進数
258	0x0102
259	0x0103

エンディアン

バイト列の並び順の設定の違い

リトルエンディアン

- 🍋 数値の最下位バイトがアドレスの低い方 → 258 (0x0102) は 02 01

ビッグエンディアン

- 🍋 数値の最上位バイトがアドレスの低い方 → 258 (0x0102) は 01 02

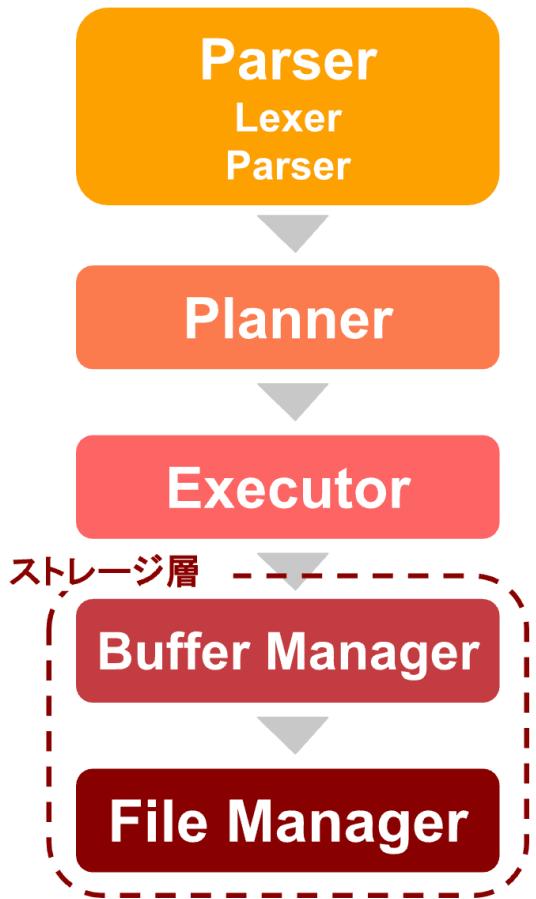
なぜ重要？

- 🍋 異なるシステム間でバイナリデータをやり取りする時
- 🍋 ネットワーク通信（ビッグエンディアン）

Pythonのstructモジュール

```
import struct  
# 16進数で0x0102をバイト列に変換  
data = struct.pack('<i', 258) # リトルエンディアン
```

<	リトルエンディアン
>	ビッグエンディアン
!	ネットワークバイトオーダー



まとめ

- **Lexer (字句解析)** - SQLをトークンに分解
- **Parser (構文解析)** - トークンからAST構築
- **Planner (実行計画)** - 最適な実行順を決定
- **Executor (実行エンジン)**
 - 実行計画に従ってデータにアクセス
- **Buffer Manager** - データのメモリ保持
- **File Manager**
 - エンディアン変換
 - ディスクへの読み書き

Appendix

最後に

インデックス、ログ、トランザクション...まだまだ話したいことが...

KeiPyDBのソースコードは、GitHubに公開済みです

<https://github.com/kei-kmj/KeiPyDB>

次にやりたいこと

🍋 ブラウザを自作して、KeiPyDBを組み込みたい!!

ご清聴ありがとうございました

