

# Presentation of a chess extension in Postgres

Database Systems Architecture Projet

*KAHLA Michael & LEGGIERI Michele & BALDARI Antonio & WEI Hongdu*

# CONTENT

## I. Introduction

## II. The extension's architecture

II. i. Implementation of chess extension functions and input/output

II. ii. The indexing structure

## III. Conclusion

## IV. Appendix

IV. i. B-Tree index

IV. ii. The GIN index

## I. Introduction

In this project, our team aims to create a *PostgreSQL extension* designed for storing and retrieving chess games. The extension utilizes various notations, including *Portable Game Notation (PGN)* for recording full games, *Standard Algebraic Notation (SAN)* for moves, and *Forsyth–Edwards Notation (FEN)* for storing board states.

The primary goals of the project include implementing *data types* (*chessgame*, *chessboard*), *specific functions*, and *indexes* to support various chess-related operations. In our implementation, we used the *smallchesslib.C* library for chess operations, ensuring the accuracy and efficiency of chess-related functions.

## II. The extension's architecture

The extension provides functions such as *'getBoard'*, *'getFirstMoves'*, *'hasOpening'*, and *'hasBoard'* in order to manipulate and query chess games efficiently. These functions are designed to work with SAN and FEN notations for chessgame and chessboard, respectively. and efficiency of chess-related functions.

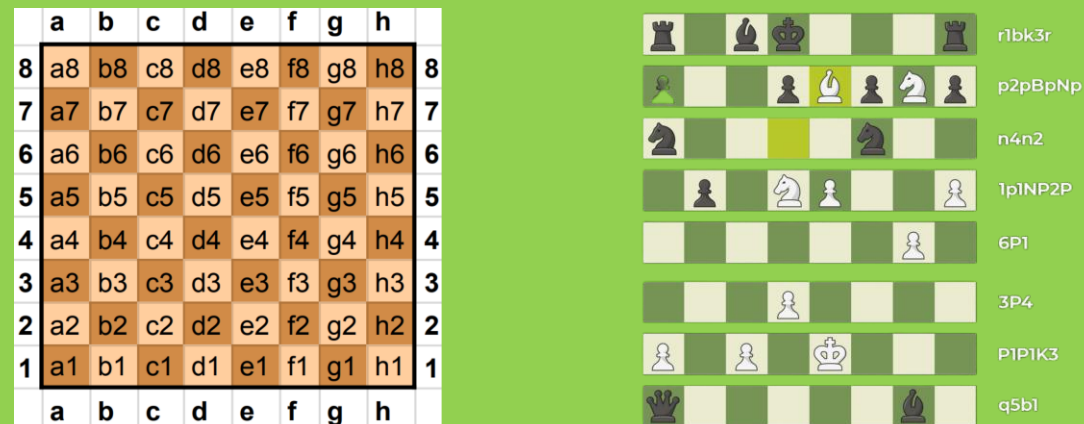


Figure 1 – SAN and FEN notations

## II. The extension's architecture

### *II. i. Implementation of input/output for chess game*

(a) The function "chessGame from Pgn" :

The chessGame from pgn function is used to return a chess game in our input function, and is thus responsible for creating a SCL GAME structure from a given PGN by the user.

(b) The function "Pgn from chessGame" :

The "Pgn from chessGame" function converts the given SCL Game structure back into a PGN string that can be read by the user, and returns it to our output function.

It utilizes the SCL printPGN function to extract moves from the game record and construct the PGN string.

## II. The extension's architecture

### *II. i. Implementation of input/output for chess board*

(a) The function "chessBoard from Fen" :

The chessBoard from fen function creates a SCL Board structure from a given FEN, and returns it to our input function. It initializes the board and uses the function SCL boardFromFEN.

(b) The function "Fen from chessBoard" :

The Fen from chessBoard function converts the SCL Board structure back into a FEN string, returned to our output function.

## II. The extension's architecture

### II. i. *Implementation of chess extension functions*

(a) The function "getBoard" :

This function extracts the board state at a specified half-move from a given chess game.

```

chess=# SELECT getBoard(my_chess_game, 1) AS my_chess_board FROM (SELECT chessGame_in('1. e4 e7 2. Nf3 Nc6') AS my_chess_game)
AS subquery;
          my_chess_board
-----
rnbqkbnr/pppppppp/8/8/4P3/8/PPPP1PPP/RNBQKBNR b KQkq e3 0 1
(1 row)

chess=# SELECT getBoard(my_chess_game, 2) AS my_chess_board FROM (SELECT chessGame_in('1. e4 e7 2. Nf3 Nc6') AS my_chess_game)
AS subquery;
          my_chess_board
-----
rnbqkbnr/pppppppp1/8/8/4P3/8/PPPP1PPP/RNBQKBNR w KQkq - 0 2
(1 row)
  
```

*Figure 2 – Test on getBoard*

## II. The extension's architecture

```

if (chessGame == NULL) {
    ereport(ERROR, (errmsg("Invalid chessGame pointer")));
    return NULL;
}
if (halfMove < 0) {
    ereport(ERROR, (errmsg("Invalid halfMove parameter")));
    return NULL;
}
if (halfMove > SCL_recordLength(chessGame->record)) {
    ereport(ERROR, (errmsg("Invalid halfMove parameter")));
    return NULL;
}
SCL_Board* chessBoard = malloc(sizeof(SCL_Board));
if (chessBoard == NULL) {
    ereport(ERROR, (errmsg("Memory allocation failed")));
    return NULL;
}

chess=# SELECT getBoard(my_chess_game, 6) AS my_chess_board FROM (SELECT chessGame_in('1. e4 e7 2. Nf3 Nc6') AS
S my_chess_game) AS subquery;
ERROR:  Invalid halfMove parameter
chess=#
  
```

*Figure 3 – Representation of error-handling system in getBoard*

In the left picture, we represent our getBoard function when the number of half moves was greater than the one in the given chess game. That resulted in an error, displaying on the screen : "Invalid halfMove parameter".



## II. The extension's architecture

### II. i. Implementation of chess extension functions

(b) The function "hasOpening" :

```

PG_FUNCTION_INFO_V1(hasOpening);
Datum hasOpening(PG_FUNCTION_ARGS)
{
    SCL_Game *cg1 = PG_GETARG_SCL_GAME(0);
    SCL_Game *cg2 = PG_GETARG_SCL_GAME(1);
    bool sameOpening = areEqual(cg1, cg2);
    PG_FREE_IF_COPY(cg1, 0);
    PG_FREE_IF_COPY(cg2, 1);
    PG_RETURN_BOOL(sameOpening);
}

bool areEqual(SCL_Game* cg1, SCL_Game* cg2) {
    char* pgn1 = strdup(pgn_from_chessGame(cg1));
    char* pgn2 = strdup(pgn_from_chessGame(cg2));
    int len1 = strlen(pgn1);
    int len2 = strlen(pgn2);
    if (len2 > len1) {
        free(pgn1);
        free(pgn2);
        return false;
    }
    for (int i = 0; i < len2; i++) {
        if (pgn2[i] == '*') {
            break;
        }
        if (pgn1[i] != pgn2[i]) {
            free(pgn1);
            free(pgn2);
            return false;
        }
    }
    free(pgn1);
    free(pgn2);
    return true;
}
  
```

The hasOpening function compares two chess games to determine if opening moves of the second game match the starting moves of the first game. If they match, the function returns True.

Figure 4 – hasOpening function

## II. The extension's architecture

In the following queries, we use the `hasOpening` function to check the similarity of opening moves between two chess games. The first query resulted in a positive match (`same opening = true`), indicating that the opening moves of the second game (`game2`) precisely matched the initial moves of the first game (`game1`). In the same way, the second query yielded a negative result (`same opening = false`), signifying that the opening moves of the second game is distinct.

```

chess=# SELECT hasOpening(game1, game2) AS same_opening FROM (SELECT chessGame_in('1. e4 e5 2. Nf3 Nc6') AS game1, chessGame_in('1. e4 e5') AS game2) AS subquery;
 same_opening
-----
t
(1 row)

chess=# SELECT hasOpening(game1, game2) AS same_opening FROM (SELECT chessGame_in('1. e4 e5 2. Nf3 Nc6') AS game1, chessGame_in('1. e3 e5') AS game2) AS subquery;
 same_opening
-----
f
(1 row)

```

*Figure 5 – Test on hasOpening*

## II. The extension's architecture

### II. i. Implementation of chess extension functions

(c) The function "hasBoard" :

```
bool verifyBoard(SCL_Game *chessGame, SCL_Board *targetBoard, int halfMoves)
{
    if(chessGame == NULL || targetBoard == NULL || halfMoves < 0 || halfMoves > SCL_recordLength(chessGame->record)) {
        ereport(ERROR, (errmsg("Invalid chessGame, targetBoard, or halfMoves parameter")));
        return false;
    }
    char *tcb = strdup(fen_from_chessBoard(targetBoard));
    for(int i = 1; i <= halfMoves; i++) {
        int check = 0;
        char *ccb = strdup(fen_from_chessBoard(getChessBoard(chessGame, i)));
        int len = strlen(tcb);
        for(int i = 0; i < len; i++)
            if(tcb[i] == ccb[i])
                check++;
        if(check == len)
            return true;
    }
    return false;
}
```

Figure 6 – hasBoard function

The function "hasBoard" takes three parameters : a chessGame, a chessBoard and an integer. It returns True if the given chess board state is found within the first N half-moves of input chess game.

## II. The extension's architecture

In the presented queries, the first query resulted in a positive match (`hasB = True`), indicating the board state after the second half move was present in the chess game. While the second query yielded a negative result (`hasB = false`), signifying the specified board state did not occur within the first two half moves of the given chess game.

```

chess=# SELECT hasBoard(my_Chess, my_Board, 2) AS hasB
FROM (
  SELECT
    chessGame_in('1. e4 e7 2. Nf3 Nc6') AS my_Chess,
    chessBoard_in('rnbqkbnr/ppppppp1/8/8/4P3/8/PPPP1PPP/RNBQKBNR w KQkq - 0 2') AS my_Board
) AS subquery;
hasb
-----
t
(1 row)

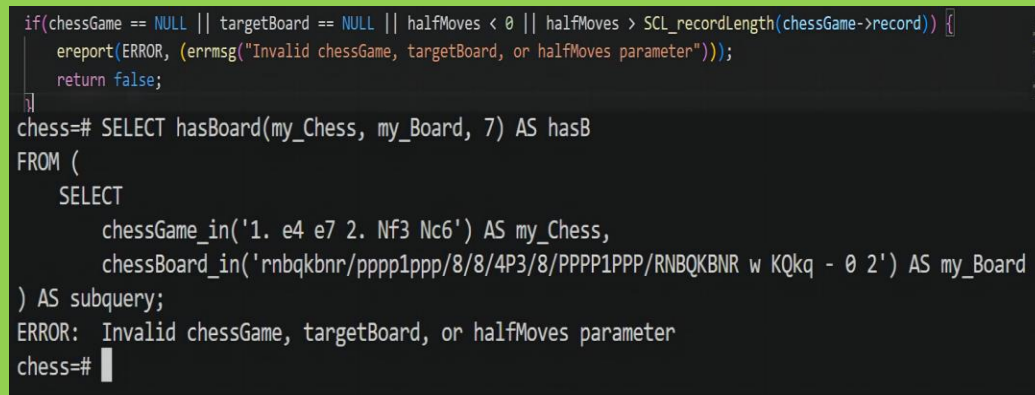
chess=# SELECT hasBoard(my_Chess, my_Board, 2) AS hasB
FROM (
  SELECT
    chessGame_in('1. e4 e7 2. Nf3 Nc6') AS my_Chess,
    chessBoard_in('rnbqkbnr/pppp1ppp/8/8/4P3/8/PPPP1PPP/RNBQKBNR w KQkq - 0 2') AS my_Board
) AS subquery;
hasb
-----
f
(1 row)

chess=#

```

*Figure 7 – Test on hasBoard*

## II. The extension's architecture



```

if(chessGame == NULL || targetBoard == NULL || halfMoves < 0 || halfMoves > SCL_recordLength(chessGame->record)) {
    ereport(ERROR, (errmsg("Invalid chessGame, targetBoard, or halfMoves parameter")));
    return false;
}
chess=# SELECT hasBoard(my_Chess, my_Board, 7) AS hasB
FROM (
    SELECT
        chessGame_in('1. e4 e7 2. Nf3 Nc6') AS my_Chess,
        chessBoard_in('rnbqkbnr/pppp1ppp/8/8/4P3/8/PPPP1PPP/RNBQKBNR w KQkq - 0 2') AS my_Board
    ) AS subquery;
ERROR: Invalid chessGame, targetBoard, or halfMoves parameter
chess=#
  
```

*Figure 8 – Representation of error-handling system in hasBoard*

In the left picture, we can see that we tested our hasBoard function when the number of half moves was greater than the one in the given chess game. That resulted in an error, displaying on the screen : "Invalid chessGame, targetBoard, or hafMoves parameters".

## II. The extension's architecture

### II. i. Implementation of chess extension functions

(d) The function "getFirstMove" :

```
SCL_Game* getFirstMoves(SCL_Game* chessGame, int halfMove)
{
    if (chessGame == NULL) {
        ereport(ERROR, (errmsg("Invalid chessGame pointer")));
        return NULL;
    }
    if (halfMove < 0 || halfMove > SCL_recordLength(chessGame->record)) {
        ereport(ERROR, (errmsg("Invalid halfMove parameter")));
        return NULL;
    }
    // Create a new truncated game
    SCL_Game* truncatedGame = malloc(sizeof(SCL_Game));
    truncatedGame->ply = 0; // Reset the half move count
    truncatedGame->startState = chessGame->startState;
    SCL_boardInit(truncatedGame->board);
    SCL_recordInit(truncatedGame->record);
    // Iterate through previous moves and apply them to the new game
    for (int i = 0; i < halfMove; i++) {
        uint8_t s0, si;
        char p;
        SCL_recordGetHalfMove(chessGame->record, i, &s0, &si, &p);
        SCL_boardMakeHalfMove(truncatedGame->board, s0, si, p);
        SCL_gameMakeHalfMove(truncatedGame, s0, si, p);
    }
    return truncatedGame;
}
```

This function takes two parameters: a given chess game, and an integer to specify the number of half moves. This function is responsible to return a truncated version of the input chessgame containing the first half-moves, specified as arguments.

Figure 9 – getFirstMove function

## II. The extension's architecture

```

chess=# SELECT getOpening(my_chess_game, 2) AS my_chess_board FROM (SELECT chess
Game_in('1. e4 e5 2. Nf3 Nc6') AS my_chess_game) AS subquery;
 my_chess_board
-----
1. e4 e5*
(1 row)

chess=# SELECT getOpening(my_chess_game, 3) AS my_chess_board FROM (SELECT chess
Game_in('1. e4 e5 2. Nf3 Nc6') AS my_chess_game) AS subquery;
 my_chess_board
-----
1. e4 e5 2. Nf3*
(1 row)

chess=# SELECT getOpening(my_chess_game, 4) AS my_chess_board FROM (SELECT chess
Game_in('1. e4 e5 2. Nf3 Nc6') AS my_chess_game) AS subquery;
 my_chess_board
-----
1. e4 e5 2. Nf3 Nc6*
(1 row)

```

*Figure 10 – Test on getFirstMove*

## II. The extension's architecture

### II. ii. The indexing structure

- B-Tree for efficient string comparisons
- GIN for hasBoard function, indexing on a table column

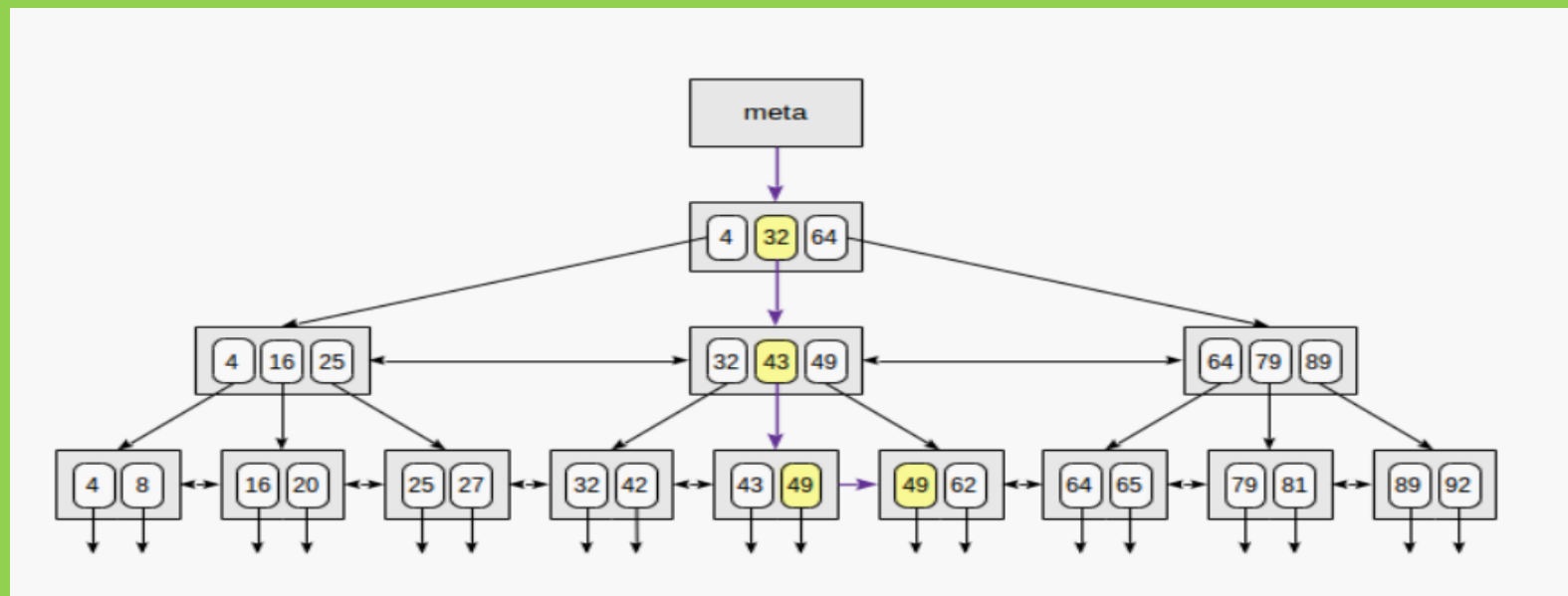


Figure 11 – Example of B-Tree index searching 49



## II. The extension's architecture

### *II. ii. The indexing structure steps*

- Creation of DB shema
- Insertion of data
- Test of hasOpening function
- Analysis of execution plan
- Verification of disabling sequential scan
- Analysis of execution plan

### III. Conclusion

In summary, our project aimed to devise a PostgreSQL extension for the adept storage and retrieval of chess games. We delved into the extensibility of databases, tailoring them to specific domains. We engaged with the intricacies of implementing data types, functions, and indexes rooted in chess notations. The extension empowers users to manipulate chess games, offering functions like retrieving board states and truncating games to specific half-moves. To bolster query efficiency, we introduced two indexes: a B-tree index for the `hasOpening` predicate, tailored for comparing chess games, and a GIN index for the `hasBoard` predicate.

## IV.i. Appendix of B-Tree index

```
chess=# CREATE TABLE Games (  
    game VARCHAR(255) PRIMARY KEY  
);  
  
CREATE TABLE Openings (  
    opening VARCHAR(255) PRIMARY KEY  
);  
CREATE TABLE  
CREATE TABLE
```

*Figure 12 – Query for creating tables*

## IV.i. Appendix of B-Tree index

```

chess=# INSERT INTO Games VALUES
('1. e4 e5 2. Nf3 Nc6'),
('1.e4 e5 2.Nf3 Nc6 3.Bb5 a6 4.Ba4 Nf6 5.O-O Be7 6.Re1 b5 7.Bb3 d6 8.c3 O-O 9.h3 Nb8 10.d4 Nbd7'),
('1.d4 d5 2.c4 c6 3.Nf3 Nf6 4.e3 Bg4 5.Qb3 Qb6 6.Nc3 Bxf3 7.gxf3 e6 8.Bd2 Nbd7 9.Be2 Be7 10.O-O O-O'),
('1.e4 c5 2.Nf3 d6 3.d4 cxd4 4.Nxd4 Nf6 5.Nc3 a6 6.Be3 e5 7.Nb3 Be6 8.f3 Be7 9.Qd2 O-O 10.O-O-O Nbd7'),
('1.e4 e5 2.Nf3 Nc6 3.Bc4 Bc5 4.c3 Nf6 5.d4 exd4 6.cxd4 Bb4+ 7.Nc3 Nxe4 8.O-O Bxc3 9.d5 Bf6 10.Re1 Ne7'),
('1.d4 d5 2.c4 e6 3.Nc3 c5 4.cxd5 exd5 5.Nf3 Nc6 6.g3 Nf6 7.Bg2 Be7 8.O-O O-O 9.Bg5 c4 10.Ne5 Be6'),
('1.e4 e5 2.Nf3 Nc6 3.Bc4 Nf6 4.Ng5 d5 5.exd5 Na5 6.Bb5+ c6 7.dxc6 bxc6 8.Be2 h6 9.Nf3 e4 10.Ne5 Bc5'),
('1.e4 c5 2.Nf3 d6 3.d4 cxd4 4.Nxd4 Nf6 5.Nc3 a6 6.Bg5 e6 7.f4 h6 8.Bh4 Qb6 9.Qd2 Qxb2 10.Rb1 Qa3'),
('1.d4 Nf6 2.c4 e6 3.Nf3 b6 4.g3 Bb7 5.Bg2 Be7 6.O-O O-O 7.Re1 d5 8.cxd5 exd5 9.Nc3 Nbd7 10.Bf4 c5'),
('1.e4 e6 2.d4 d5 3.Nd2 c5 4.exd5 Qxd5 5.Ngf3 cxd4 6.Bc4 Qd6 7.O-O Nf6 8.Nb3 Nc6 9.Nbxd4 Nxd4 10.Nxd4 a6')
,
('1.d4 d5 2.c4 c6 3.Nf3 Nf6 4.Nc3 dxc4 5.a4 Bf5 6.e3 e6 7.Bxc4 Bb4 8.O-O Nbd7 9.Qe2 Bg6');

INSERT INTO Openings VALUES
('1. e4 e5');
INSERT 0 11
INSERT 0 1

```

*Figure 13 – Inserted 1 PGN in "Opening" column and 11 PGN in "Games" columns*

## IV.i. Appendix of B-Tree index

```

chess=#
SELECT opening, count(game) FROM Games, Openings WHERE hasOpening(game::chessGame,
opening::chessGame) GROUP BY opening;
 opening | count
-----+-----
 1. e4 e5 |      4
(1 row)

```

*Figure 14 – Query counting num of given opening games, with results*

```

                                QUERY PLAN
-----
HashAggregate  (cost=623.38..624.88 rows=150 width=524) (actual time=1.186..1.187 rows=1 loops=1)
  Group Key: openings.opening
  Batches: 1  Memory Usage: 40kB
  -> Nested Loop  (cost=0.00..585.88 rows=7500 width=1032) (actual time=0.060..1.178 rows=4 loops=1)
    Join Filter: hasopening((games.game)::chessgame, (openings.opening)::chessgame)
    Rows Removed by Join Filter: 7
    -> Seq Scan on games  (cost=0.00..11.50 rows=150 width=516) (actual time=0.007..0.008 rows=11 loops=
1)
      -> Materialize  (cost=0.00..12.25 rows=150 width=516) (actual time=0.000..0.001 rows=1 loops=11)
        -> Seq Scan on openings  (cost=0.00..11.50 rows=150 width=516) (actual time=0.002..0.003 rows=
1 loops=1)
  Planning Time: 0.100 ms
  Execution Time: 1.216 ms
(11 rows)

```

*Figure 15 – Query plan using sequential scan*

## IV.i. Appendix of B-Tree index

```
chess=# CREATE INDEX games_hasOpening_index ON Games USING btree (game);  
CREATE INDEX  
chess=#
```

*Figure 16 – B-Tree index creation*

```
chess=# SET enable_seqscan = OFF;  
SET  
chess=#
```

*Figure 17 – Utilization of index preference over sequential scan*

## IV.i. Appendix of B-Tree index

```

                                QUERY PLAN
-----
GroupAggregate (cost=0.28..108.22 rows=150 width=524) (actual time=1.210..1.210 rows=1 loops=1)
  Group Key: openings.opening
    -> Nested Loop (cost=0.28..103.97 rows=550 width=1032) (actual time=0.139..1.205 rows=4 loops=1)
        Join Filter: hasopening((games.game)::chessgame, (openings.opening)::chessgame)
        Rows Removed by Join Filter: 7
        -> Index Only Scan using openings_pkey on openings (cost=0.14..50.40 rows=150 width=516) (actual time=0.040..0.041 rows=1 loops=1)
            Heap Fetches: 1
        -> Materialize (cost=0.14..12.35 rows=11 width=516) (actual time=0.059..0.065 rows=11 loops=1)
            -> Index Only Scan using games_hasopening_index on games (cost=0.14..12.30 rows=11 width=516)
                (actual time=0.056..0.060 rows=11 loops=1)
                Heap Fetches: 11
  Planning Time: 0.187 ms
  Execution Time: 1.238 ms
(12 rows)
  
```

*Figure 18 – Utilization of our index on game column with query plan*

## IV.i. Appendix of GIN index