Database systems architecture
Report

Implementation of a Postgre extension

KAHLA Michael WEI Hongdu LEGGIERI Michele BALDARI Antonio

10/12/23

Université Libre de Bruxelles

Ecole Polytechnique de Bruxelles

Database systems architecture

Contents

ECOLE POLYTECHNIQUE DE BRUXELLES

1 Introduction

1.1 Aim of the project

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.

2 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.

2.1 Implementation of chess extension functions and input/output

In this section, we provide an overview of the implementation of key functions within our PostgreSQL extension for storing and retrieving chess games :

1. Input/Output for the 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, given 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. This function

2. Input/Output for the chess board

1. 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.

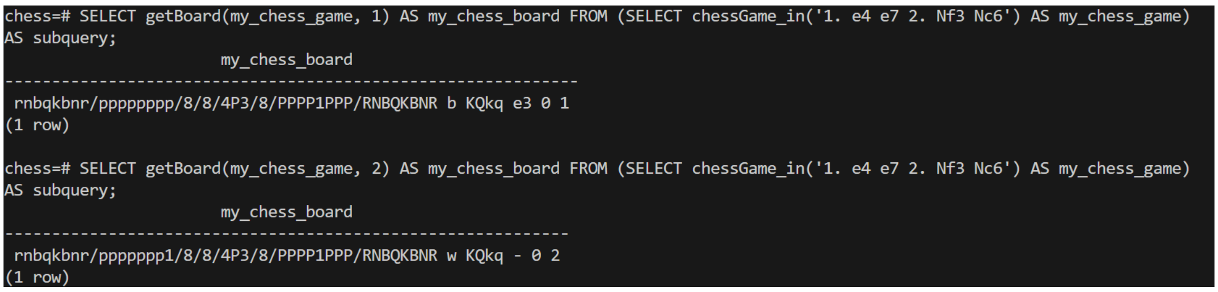
2. The function "Fen_from_chessboard" :

The Fen_from_chessBoard function converts the SCL_Board structure back into a FEN string, returned to our output function.

<u>Some useful functions</u>

1. getBoard function

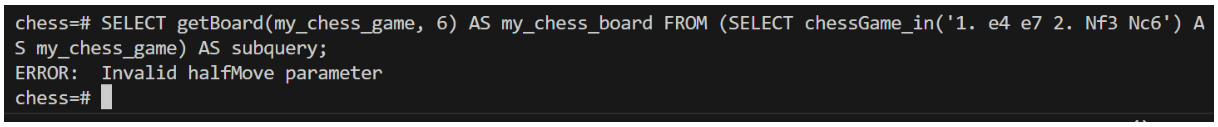
   The getBoard function extracts the board state at a specified half-move from a given chess game.



```
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/ppppppp1/8/8/4P3/8/PPPP1PPP/RNBQKBNR w KQkq - 0 2
(1 row)
```

Figure 1: Example of a getBoard function with two different half moves

In the above exemple, the getBoard function was tested with two different half moves : when the half move is 1, the fen that is returned is : "rnbqkbnr/pppppppp/8/8/4P3/8/PPPP1PPP/RNBQKBNR b KQkq e3 0 1". The interpretation of the fen is simple. Lowercase letters represent black pieces, and uppercase letters represent white pieces. Each letter corresponds to a specific piece: 'r' for rook, 'n' for knight, 'b' for bishop, 'q' for queen, 'k' for king, and 'p' for pawn. In a chess in general, the white will begin, and so, the letter 'b' indicates that it is the black's turn to move. The "1" in the string "PPPP1PPP" means ther is an empty square in the middle of the row for the white pieces. That can be verified from the PGN string with "1. e4 e7", which indicates that a white pawn begin its move to the e4 position, and thus keeping its initial square empty.
Same reasoning when the half move is equal to 1 (turn of blacks) and now the FEN returned is "rnbqkbnr/ppppppp1/8/8/4P3/8/PPPP1PPP/RNBQKBNR w KQkq - 0 2" where "ppppppp1" means that the black pawn moved to e7.

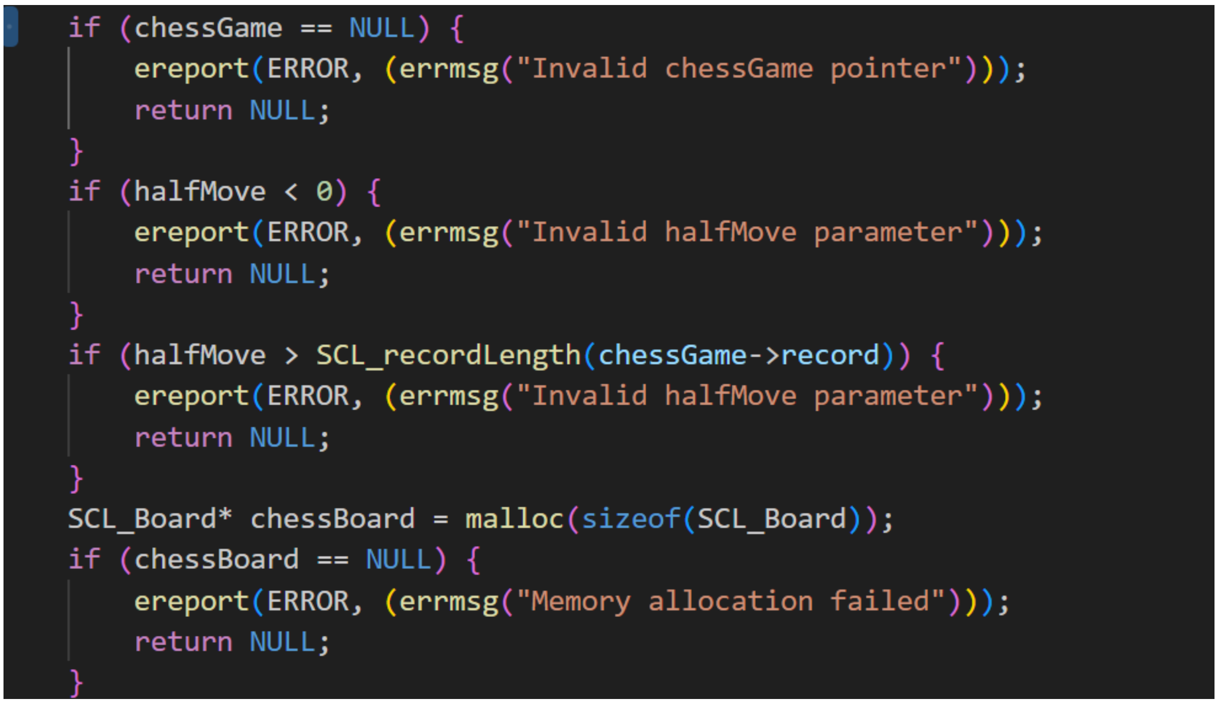


```
chess=# SELECT getBoard(my_chess_game, 6) AS my_chess_board FROM (SELECT chessGame_in('1. e4 e7 2. Nf3 Nc6') A
S my_chess_game) AS subquery;
ERROR:  Invalid halfMove parameter
chess=# 
```

Figure 2: getBoard function returning error

In the above picture, we can see that we tested 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".

Similarly, different errors where handled as shown in this picture :

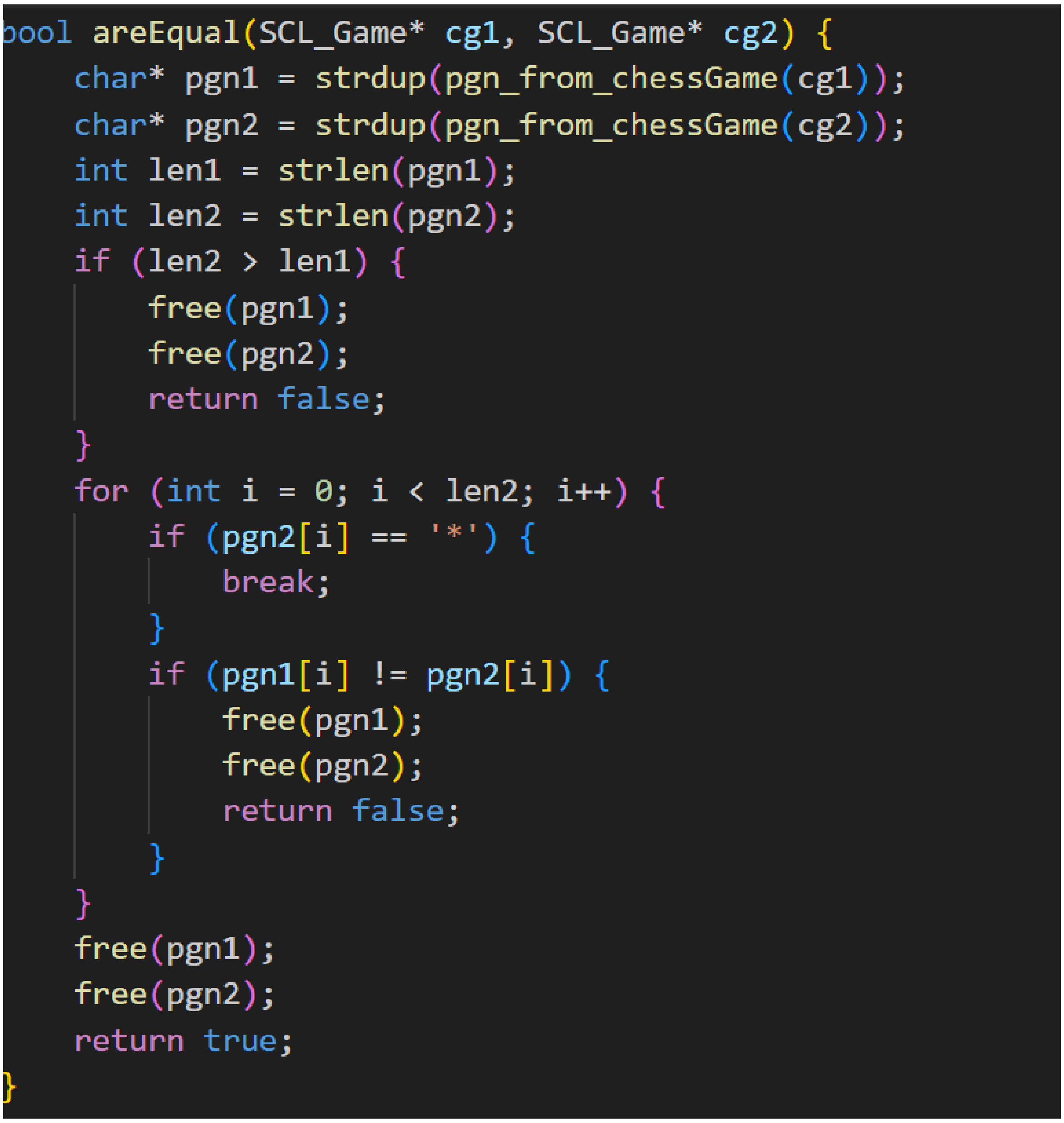


```
if (chessGame == NULL) {
    ereport(ERROR, (errmsg("Invalid chessGame pointer")));
    return NULL;
}
if (halfMove < 0) {
    ereport(ERROR, (errmsg("Invalid halfMove parameter")));
    return NULL;
}
if (halfMove > SCL_recordLength(chessGame->record)) {
    ereport(ERROR, (errmsg("Invalid halfMove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

Figure 3: Handling errors with getBoard functions

2. hasOpenning function

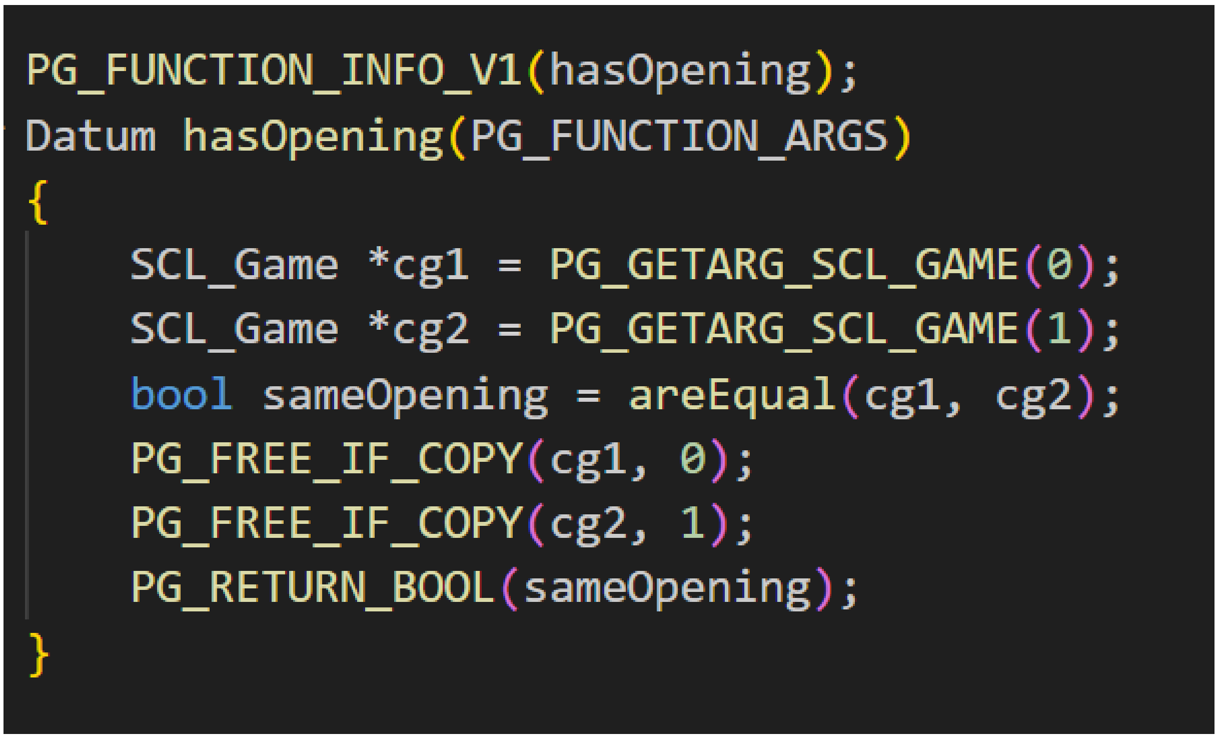
   The hasOpening function compares two chess games to determine if the second game's opening moves match the starting moves of the first game. If they match, the function returns True. The code of this function is shown below :



```c
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;
}
```

Figure 4: hasOpening function

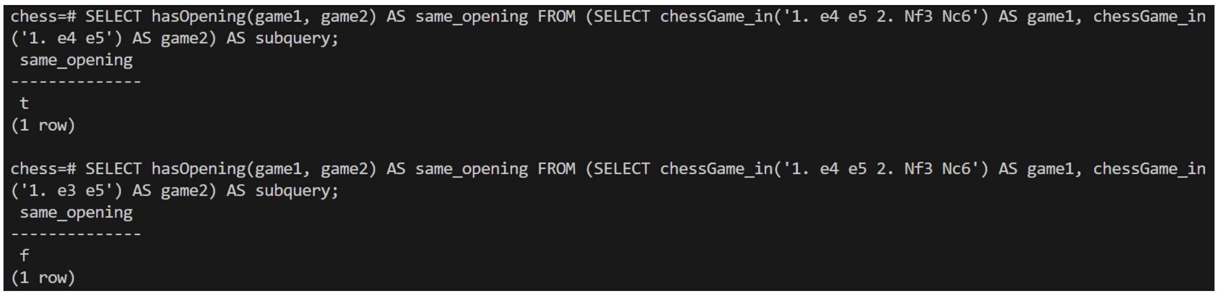
Here is an explanation of the code : the function takes two pointers as argument of type SCL_Game structures (cg1 and cg2). It retrieves the PGN strings for each chess game using pgn_from_chessGame function. Then, the lengths of the two PGN strings are calculated. If the second PGN is longer than the first, the function returns false because they cannot be equal. It then compares each characters up to the length of the second PGN. If a "*" is encountered in the second PGN (the shorter one), the comparison stops. If each characters at the current position are different, the function returns false. Finally, the dynamically allocated memory for the PGN strings is freed, and the function returns true.



```
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);
}
```

Figure 5: hasOpening function using the function ”areEqual”

Let’s try now to run some queries using this function :



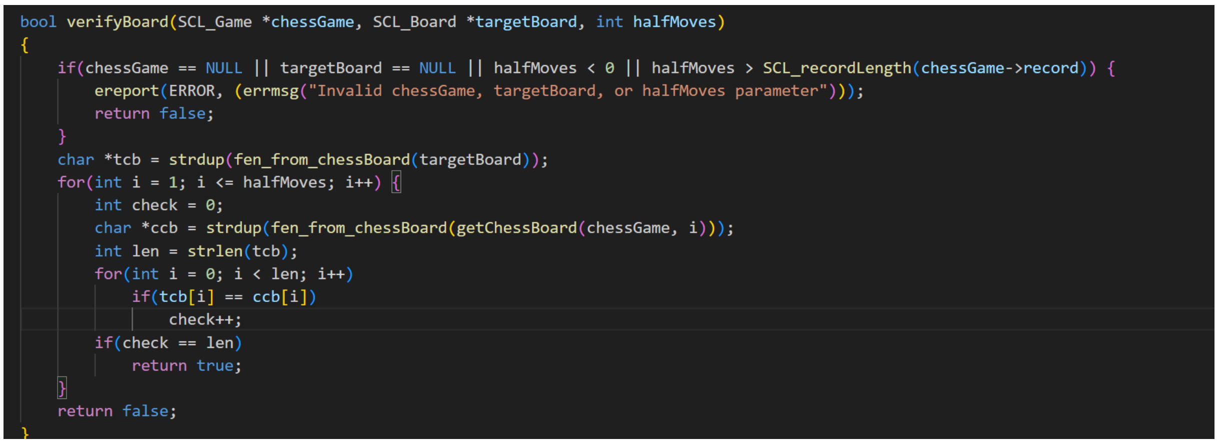
```
chess=# SELECT hasOpening(game1, game2) AS same_opening FROM (SELECT chessGame_in('1. e4 e5 2. Nf3 Nc6') AS game1, chessGame_in
('1. e4 e5') AS game2) AS subquery;
 same_opening
--------------
 t
(1 row)

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 6: Example of the hasOpening function

In the given queries, we employed 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 differed from the starting moves of the first game.

3. hasBoard function

The function "hasBoard" takes three parameters : a chessGame, representing our game, a chessBoard, which represents a specific board state, and an integer, which specifies the number of half-moves. This function returns True if the given chess board state is found within the first N half-moves of the input chess game.



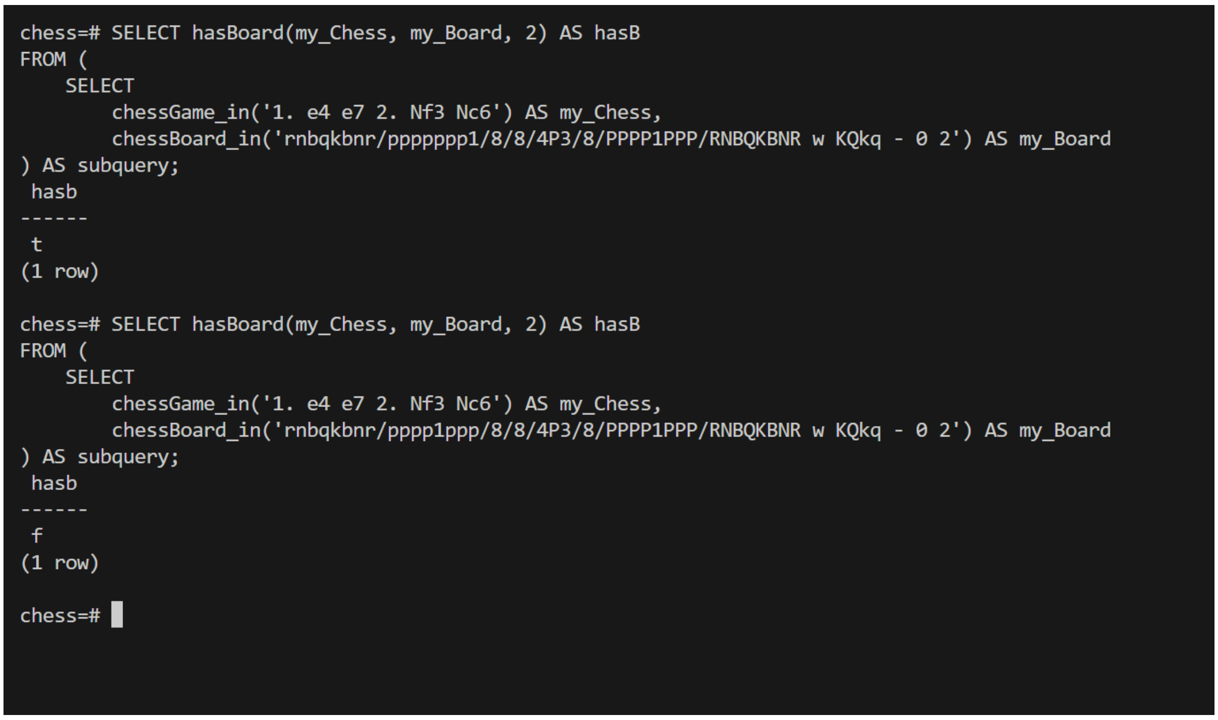
```c
bool verifyBoard(SCL_Game *chessGame, SCL_Board *targetBoard, int halfMoves)
{
    if(chessGame == NULL || targetBoard == NULL || halfMoves < 0 || halfMoves > SCL_recordLength(chessGame->record)) {
        ereport(ERROR, (errmsg("Invalid chessGame, targetBoard, or halfMoves parameter")));
        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 7: Boolean function verifyBoard used in hasBoard function

Explanation of the code :

The function starts by checking the validity of the input parameters: chessGame (a pointer to a chess game structure), targetBoard (a pointer to a chess board structure), and halfMoves (an integer specifying the number of half-moves to consider).

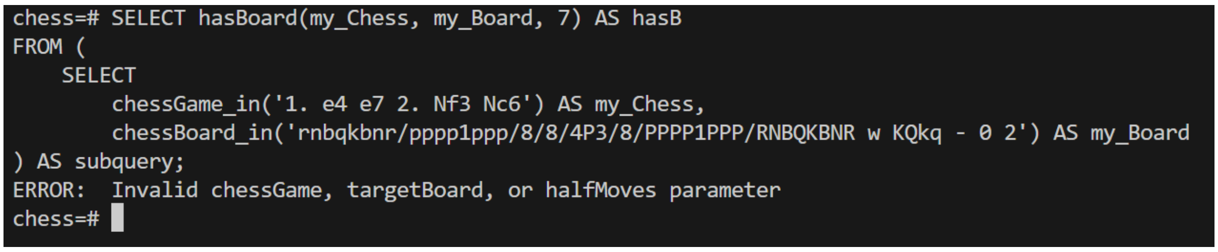
Then, after converting the targetBoard to a FEN string using the fen_from_chessBoard function, we iterate through the half-moves in the chessGame, and for each half-move, we converts the chess board at that half-move to a FEN string, and compare character by character with the FEN representation of the targetBoard. The "If" conditions checks if the two the FEN representations match. If no match is found after iterating through the specified half-moves, the function frees the memory and returns false, else, it returns True.



```
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 8: hasBoard with N = 2 and two queries

In the presented queries, we executed the hasBoard function in order to determine whether the given chess game, represented by the my_Chess variable, at the first two half moves, contains the specified chessboard state, represented by the my_Board variable. The first query resulted in a positive match (hasB = True), indicating that the board state (represented by the FEN string) after the second half move was present in the chess game (represented by the PGN string). With the same intuition, the second query yielded a negative result (hasB = false), signifying that the specified board state did not occur within the first two half moves of the given chess game. Indeed, one can take a closer look to the FEN in that query and see that it was modified so that the second half move (e7) doesn't match the given board.

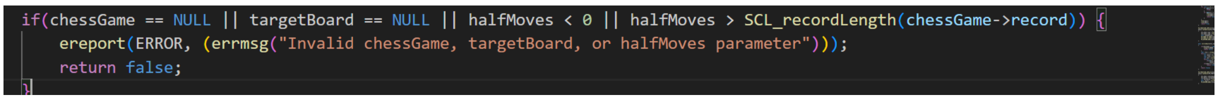


```
chess=# SELECT hasBoard(my_Chess, my_Board, 7) AS hasB
FROM (
    SELECT
        chessGame_in('1. e4 e7 2. Nf3 Nc6') AS my_Chess,
        chessBoard_in('rnbqkbnr/pppp1ppp/8/8/4P3/8/PPPP1PPP/RNBQKBNR w KQkq - 0 2') AS my_Board
) AS subquery;
ERROR:  Invalid chessGame, targetBoard, or halfMoves parameter
chess=#
```

Figure 9: hasBoard function if the number of half moves is greater than the one in the given PGN

In the above 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”.

Similarly, different errors where handled as shown in this picture :

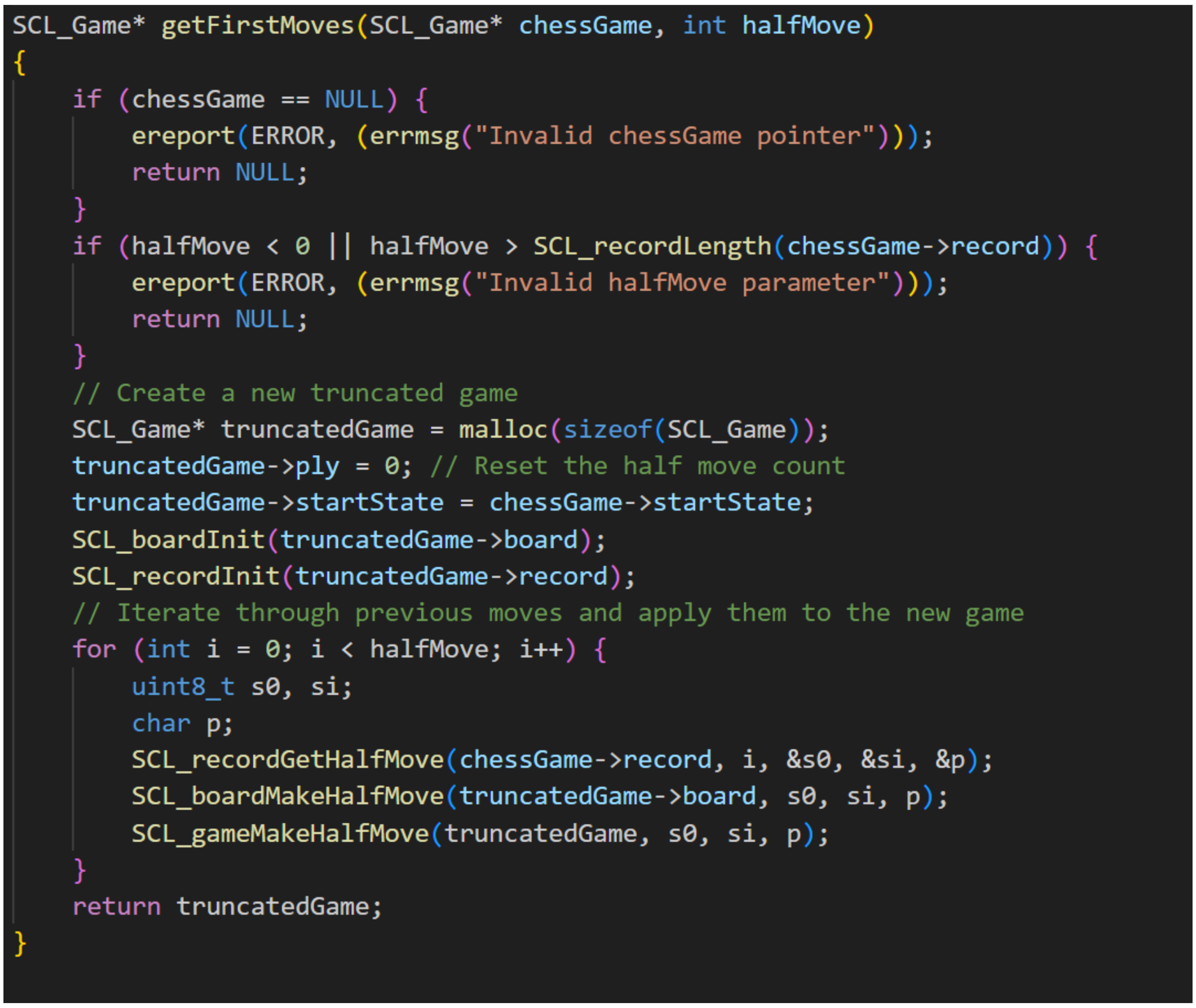


```
if(chessGame == NULL || targetBoard == NULL || halfMoves < 0 || halfMoves > SCL_recordLength(chessGame->record)) {
    ereport(ERROR, (errmsg("Invalid chessGame, targetBoard, or halfMoves parameter")));
    return false;
}
```

Figure 10: Handling errors with hasBoard

4. getFirstMove function

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.



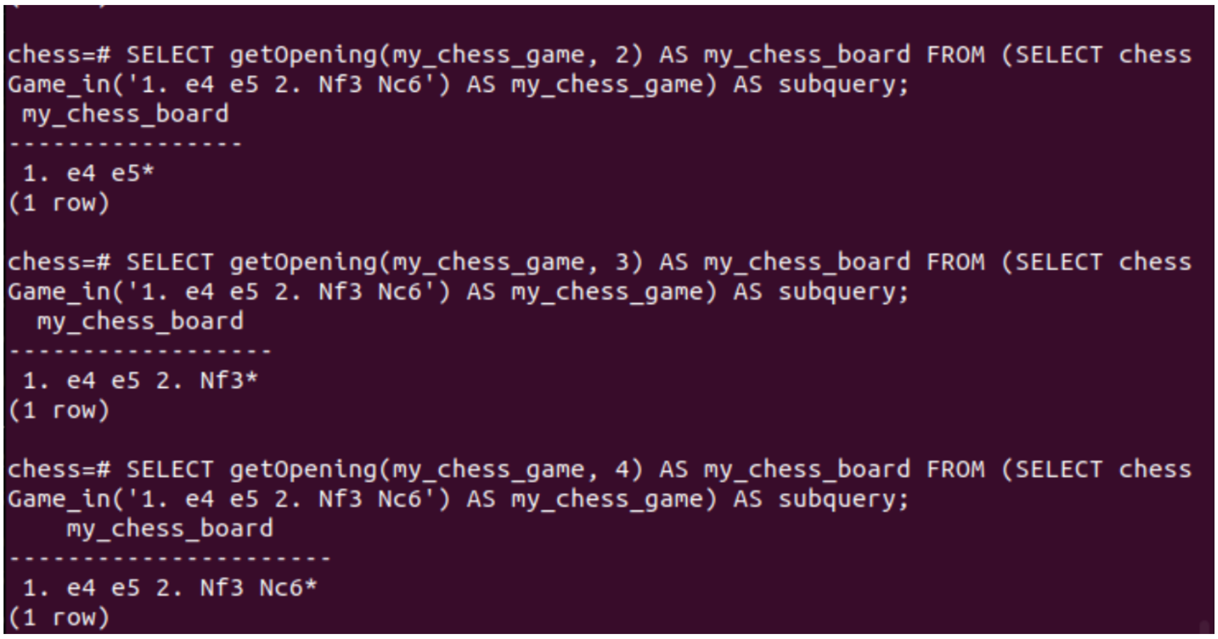
```c
SCL_Game* getFirstMoves(SCL_Game* chessGame, int halfMove)
{
    if (chessGame == NULL) {
        ereport(ERROR, (errmsg("Invalid chessGame pointer")));
        return NULL;
    }
    if (halfMove < 0 || halfMove > SCL_recordLength(chessGame->record)) {
        ereport(ERROR, (errmsg("Invalid halfMove 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;
}
```

Figure 11: getFirstMove function

Explanation of the code :

The function starts by checking whether the chessGame pointer is valid (not NULL) and whether the halfMove parameter is within a valid range based on the length of the record in the original game. If either condition fails, the function reports an error and returns NULL. Then, we allocate memory for a new chess game, which represents a truncated version of the original game. After that, the half move count (ply) in the truncated game is reset to 0, and the start state is copied from the original game to the truncated game. After initializing the board and the record of the truncated game, we iterate through the the first half moves of the game, given in parameter of the function. For each half-move, the function retrieves the source square (s0), destination square (si), and piece (p) from the original game's record. It then updates the board and game state of the truncated game using

SCL_boardMakeHalfMove and SCL_gameMakeHalfMove functions.



```
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 12: Example of queries with getFirstMove function

2.2 The indexing structure

In the context of our extension used for storing and retrieving chess games, the design and implementation of indexing structures play a crucial role in optimizing query performance. Indeed, in the context of databases, an index is a data structure that enhances the speed of data tuple retrieval operations, by storing a mapping between the values of one or more records in a table, and a pointer, which points to their corresponding physical locations in the table. Thus, Indexes are used to quickly locate data without having to search every row in a database table. All of this comes at a cost : indeed, While indexes enhance read performance, they can have bad implications on write performance as each modification to indexed columns may require to update the index. In this report, we will focus on two types of indexes : the B-Tree index, and the GIN index. First of all, we implemented a B-Tree index for our hasOpening function. Indeed, B-tree indexes are suitable for string comparisons and provide efficient range queries. In our hasOpening function, we compare two given chess game, which can be seen as a string (a PGN).

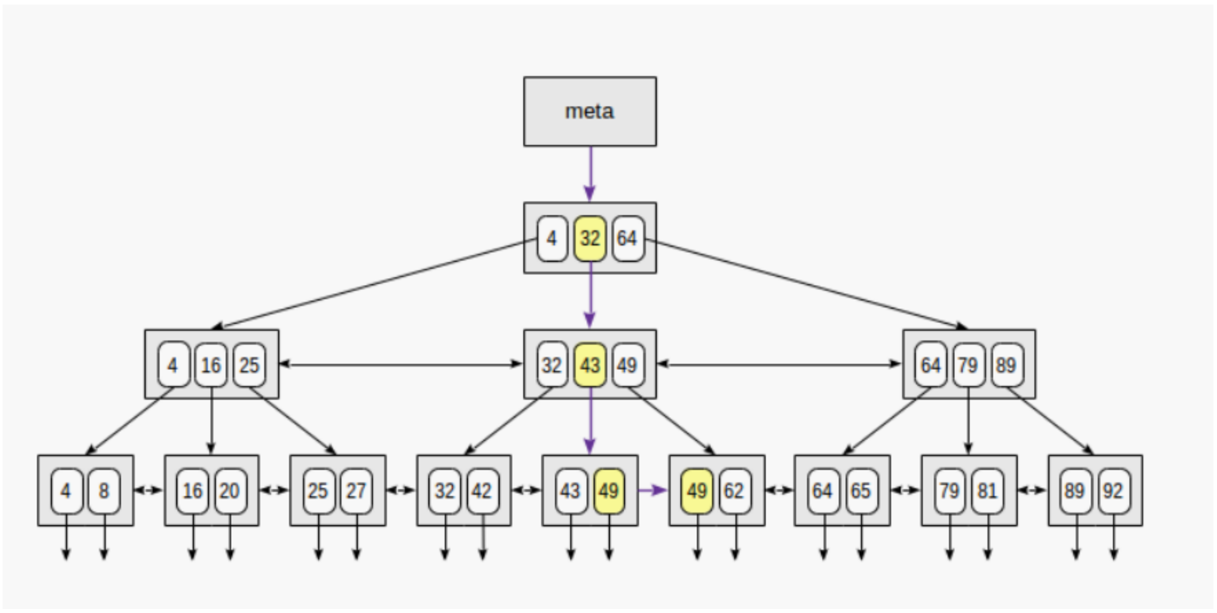


Figure 13: Example of a B-Tree index in order to search 49

Second, we implemented the GIN index for our hasBoard function.

An index can only be created on the column of a table, so in order to implement our B-tree index and test it, we created tables and add some data to it. (All the queries and their results for the steps bellow are shown in the appendix)

1. Creating our DB schema

Since our index should be used on a query that contains the 'hasOpening' predicate, we started by creating two tables, "Games" and "Openings", with a VARCHAR(255) column as the primary key, respectively "game" and "opening" storing our games and our openings. Games table is used to store chess game data, and Openings table is intended to store chess openings.

2. Insertion of data in our tables

Second, we inserted multiple chess game records into the Games and Opening table, each represented by a unique pgn.

3. Testing the hasOpening Function

A SELECT query was executed to count the number of games associated with each opening. The query used the hasOpening function, joining the Games and Openings tables.

4. Execution Plan Analysis

An EXPLAIN ANALYZE command was crucial to analyze the execution plan of the

SELECT query. We could see that the query was executed without using the index, but a sequential scan of our columns. (Indeed, the best optimized way is using this scanning instead of the index if our table does not contain lots of data)

5. Disabling Sequential Scan

In order to check if our index worked well, the command SET enable_seqscan = OFF was used to disable sequential scans, and to use the index.

6. Execution Plan Analysis

Another EXPLAIN ANALYZE command was executed to analyze the updated execution plan after disabling sequential scan. COST ?? ?? ? ?? ?? ?

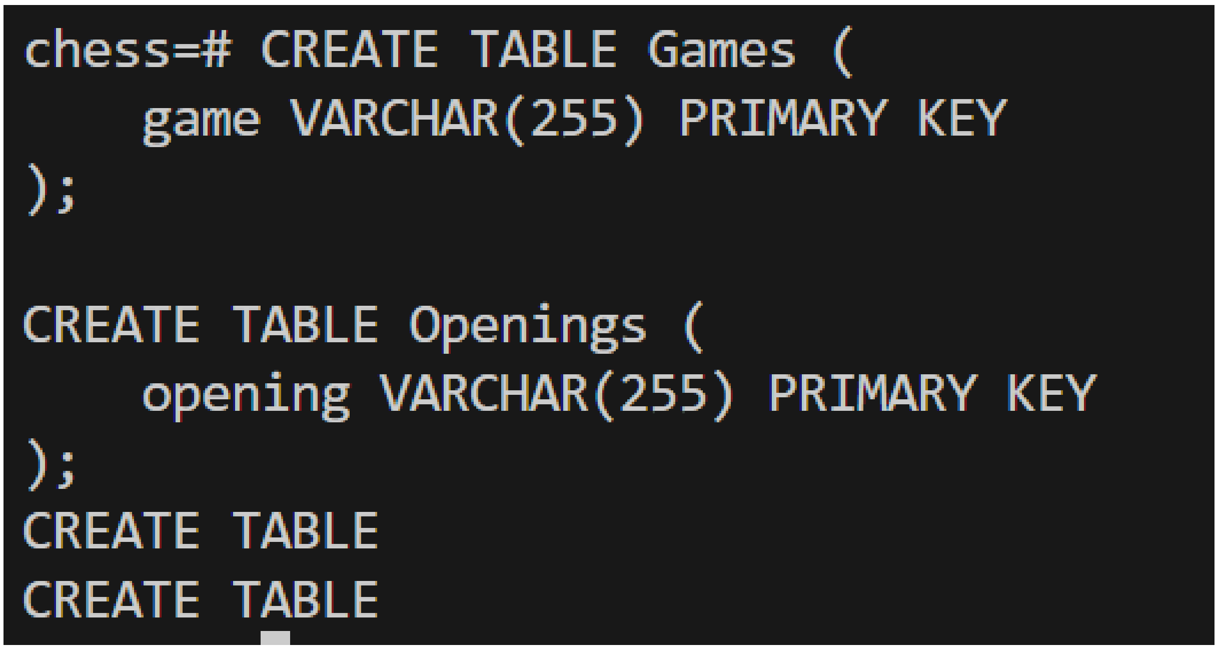
3 Conclusion

In conclusion, our project aimed to create a PostgreSQL extension for efficiently storing and retrieving chess games. We learned that databases can be extensible, allowing us to use them for specific domains and applications. We familiarized ourselves with the notion of implmentation of data types, functions, and indexes based on the chess notations. The extension allows users to play with chess games using various functions, such as retrieving board states at specific half-moves, or truncating games to their first N half-moves.

To optimize query performance, we introduced two indexes to support the hasOpening and hasBoard function. The B-tree index was chosen for the hasOpening predicate, considering the comparison of chess games made by this function. For the hasBoard predicate, a GIN index was chosen.

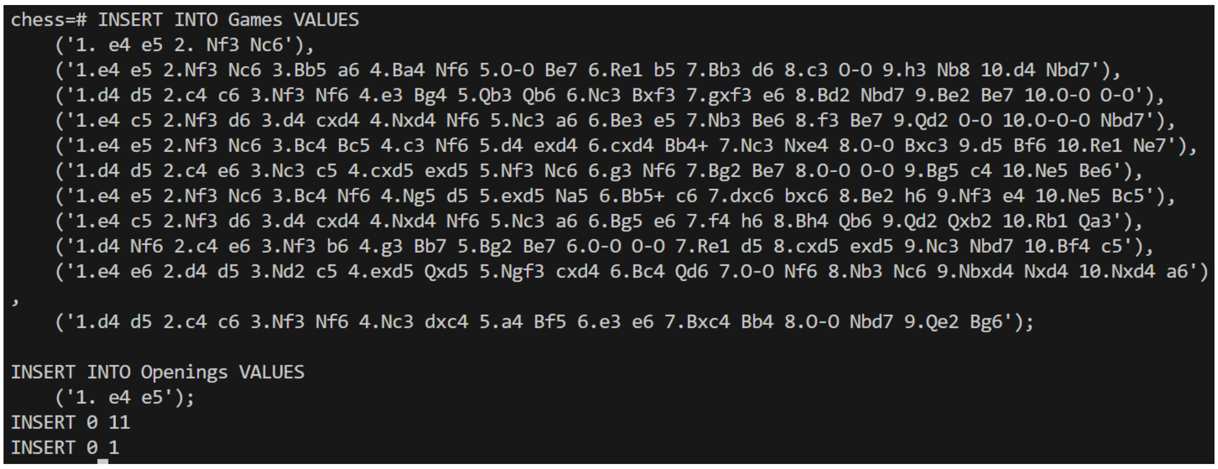
4 Appendix

4.1 Annexe A : B-Tree index



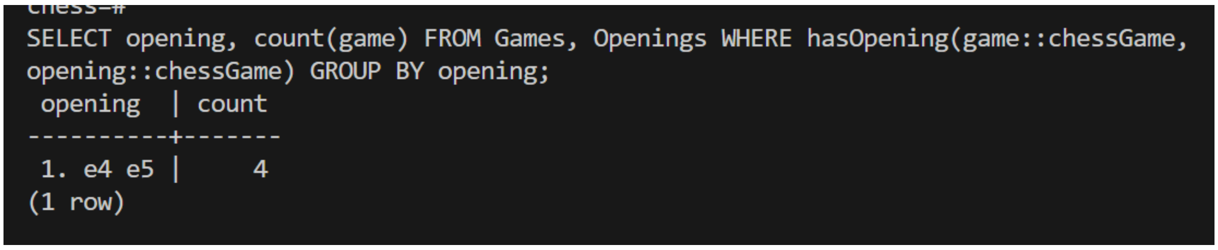
```
chess=# CREATE TABLE Games (
    game VARCHAR(255) PRIMARY KEY
);

CREATE TABLE Openings (
    opening VARCHAR(255) PRIMARY KEY
);
CREATE TABLE
CREATE TABLE
```

Figure 14: Query for creating the tables



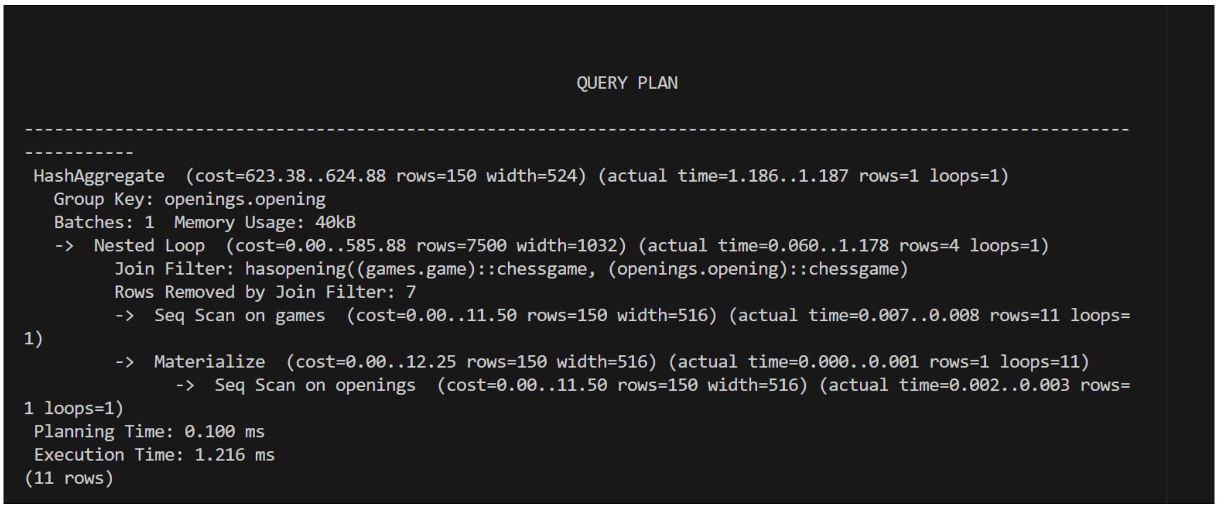
```
chess=# INSERT INTO Games VALUES
    ('1. e4 e5 2. Nf3 Nc6'),
    ('1.e4 e5 2.Nf3 Nc6 3.Bb5 a6 4.Ba4 Nf6 5.O-O Be7 6.Re1 b5 7.Bb3 d6 8.c3 O-O 9.h3 Nb8 10.d4 Nbd7'),
    ('1.d4 d5 2.c4 c6 3.Nf3 Nf6 4.e3 Bg4 5.Qb3 Qb6 6.Nc3 Bxf3 7.gxf3 e6 8.Bd2 Nbd7 9.Be2 Be7 10.O-O O-O'),
    ('1.e4 c5 2.Nf3 d6 3.d4 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 15: Inserted 1 PGN in "Openning" column and 11 PGN in "Games" columns



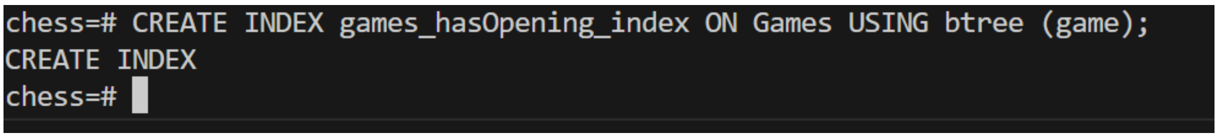
```
chess=#
SELECT opening, count(game) FROM Games, Openings WHERE hasOpening(game::chessGame,
opening::chessGame) GROUP BY opening;
 opening  | count
----------+-------
 1. e4 e5 |     4
(1 row)
```

Figure 16: Query that counts the number of games for a given openning, with its result



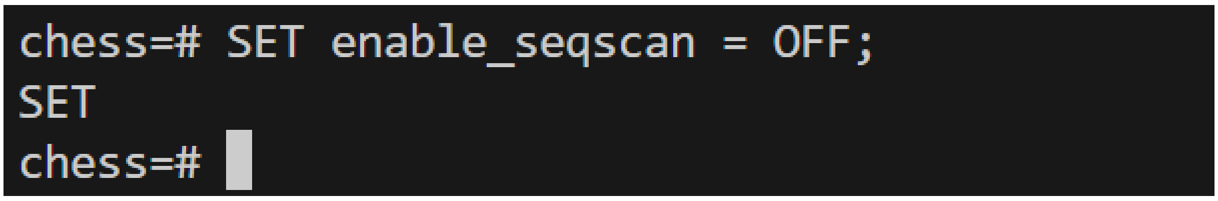
```
                                                   QUERY PLAN
-----------------------------------------------------------------------------------------------------------
----------
 HashAggregate  (cost=623.38..624.88 rows=150 width=524) (actual time=1.186..1.187 rows=1 loops=1)
   Group Key: openings.opening
   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 17: Query plan of the query above using sequential scan



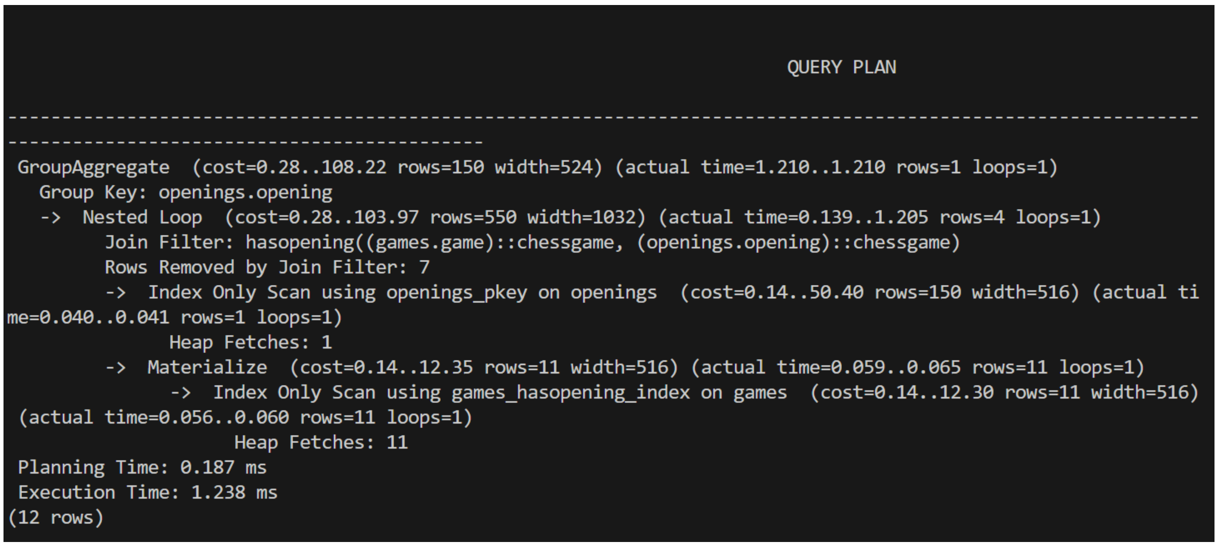
```
chess=# CREATE INDEX games_hasOpening_index ON Games USING btree (game);
CREATE INDEX
chess=#
```

Figure 18: Creation of the b-tree index



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

Figure 19: Command use to force our query to use our index instead of the sequential scan



```
                                                                         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 ti
me=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 20: Query plan of the query above which successfully use our index on the game column

4.2 Annexe B : The GIN index