EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik

Masterthesis Computer Science

Lateral Join for SQLite

Jonatan Braun

February 26, 2021

Supervisor

Prof. Dr. Torsten Grust
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen

Co-Supervisor

Prof. Dr. Klaus Ostermann
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen

Braun, Jonatan:
Lateral Join for SQLite
Masterthesis Computer Science
Eberhard Karls Universität Tübingen
Bearbeitungszeitraum: 01.09.2020 - 28.02.2021

Abstract

The popular RDBMS SQLite is held back by the lack of support for some modern database computation techniques. To adress parts of this issue this work presents an approach to add lateral joins to its list of features. This addition provides new possibilities for SQL queries and could make SQLite ready for fast in-database computation tools.

Contents

LIST OF ABBREVIATIONS

API	application programming interface
RDBMS	relational database management system
SQL	Structured Query Language
VDBE	Virtual Database Engine
yacc	yet another compiler compiler

1 INTRODUCTION

SQLite [5] is a relational database management system (RDBMS). Unlike most others it does not rely on a client-server model for communication with the Database. Instead it provides a lightweight C-Library which can be integrated directly into applications. This includes the RDBMS as a part of the application instead of a separate instance (process) communicating with it. Having the Database integrated into the same process like this comes with quite a few advantages like faster communication due to in-process function calls. The main argument however seems to be ease of use. In comparison to other major RDBMS like PostgreSQL [4] the set-up effort (especially for small development applications) is almost negligible. SQLite provides an application programming interface (API) that allows developers to create and process Structured Query Language (SQL) statements in their application similar to how other functionalities like `fopen` are used. Therefore, SQLite is used in a lot of applications, arguably it is the most used (at least when it comes to instances) database engine out there. This is due to its use in most browsers, many mobile applications and other embedded systems, just to name a few [5]. With all the reasons to use SQLite when designing an applications, there are also some drawbacks, some of which will be discussed in this work. For a lot of use cases there are strong reasons not to rely on SQLite. Scalability is one of SQLites most limiting factors. The vast selection of functionalities provided by modern full scale RDBMS like PostgreSQL is another one. Lateral Subqueries were introduced in PostgreSQL 9.3 and provide new ways to write SQL queries. They are one example of functionality not represented in SQLite.

1.1 Purpose

This thesis aims to explore the expandability of SQLite for additional powerful features introduced by other RDBMS on the basis of lateral subqueries. The goal is to implement a SQLite version of the **`LATERAL`** keyword without disrupting core functionality.

The thesis consists of 5 chapters. In chapter 2 the core concept of the **`LATERAL JOIN`**

is explained. The relational database management system (RDBMS) SQLite and its architecture are presented in chapter 3. It also examines some code details that are important for the implementation of lateral joins for SQLite, which is elaborated on in chapter 4. The final chapter 5 provides a short summary of this thesis content, including a short discussion of the findings as well as some aspects that are worth criticizing as a conclusion to complete this work.

1.2 Motivation

The support of lateral joins is not mandatory for a RDBMS to support conventional SQL database interactions but it can provide a tool to simplify queries. Tables, queries or calculations can be bound to variables to avoid duplications in the SQL statement and to provide a more intuitive programming approach. SQL queries can get pretty complicated very quickly for more complex computations. This is also where "imperative programs offer several benefits over SQL and hence are often preferred and widely used." [3] In tools to greatly enhance the performance of imperative programming in RDBMS the support of `LATERAL JOIN` can be a requirement [1, 2]. The addition of lateral queries to SQLite can therefore make it ready for these use cases as well.

2 LATERAL JOIN

The main purpose of this thesis is to implement the **LATERAL** keyword into the SQL syntax of SQLite. Therefore, it is important to know what a lateral subquery does and how it works.

This chapter gives an introduction into **LATERAL JOIN** and its applications in SQL queries. Usually SQL is evaluated independently for each subquery present in the **FROM** statement. This means that a query might access the result set of a subquery used as a source but the subquery, evaluated independently, does not know anything about its context.

PostgreSQL [4] introduced lateral subqueries in version 9.3 allowing subqueries in the **FROM** clause to reference other **FROM** items.

2.1 Semantic

The **LATERAL** keyword lets subqueries declared with it access variables bound by the keyword. A query of the Form

```
SELECT items FROM table1, table2, LATERAL ([Subquery]), table3 ...
```

would allow the `[Subquery]` to access columns of table1 and table2 but not table3 or others following.
The way the query will be interpreted is basically like a for-each-loop. In the example above the `[Subquery]` would be executed for every item in `table1` and `table2`. In each iteration it has access to the current row of `table1×table2`.

```
foreach element in (table1 x table2)
|  LATERAL Subquery
|  ⌊    query body
⌊
```

This results in all the items in the **FROM** clause that appear on the left side of the **LATERAL** keyword to be available inside the **LATERAL** `Subquery`. Note that these

items may be subqueries themselves which would make only their results available. Nested queries with **LATERAL JOIN** are also possible. In that case **LATERAL** not only allows for access to the preceding **FROM** term items of the same query but also those of the outer query (left of its declaration).

2.2 Syntax

The Syntax for lateral joins might vary depending on the RDBMS in use but they share the same basic structure. Since lateral subqueries were introduced with PostgreSQL [4] its syntax is used in this chapter. In non lateral queries, source tables (or queries) usually can be named in any arbitrary order without any impact on the result of the query. As stated earlier, lateral queries introduce order as an important factor in the **FROM** part of the query. All columns left of the **LATERAL** keyword become available variables inside of the related subquery.

```
SELECT res FROM (SELECT 42 AS x) AS q1(x),
    LATERAL (SELECT x+1 AS res) AS result;
```

Listing 2.1: LATERAL toy x+1 example

Looking at the toy example in Listing 2.1 the value 42 of the query q1 is bound to variable x and can be referenced in the lateral subquery which yields the result of x+1. This example may not seem very useful but it shows how columns can be bound to variables for use in a lateral query. As a developer, one might imagine the potential simplifications this simple concept brings into SQL. Repetitions of the same subqueries in more complex queries can be avoided using **LATERAL**.

3 SQLITE

"SQLite source code is in the public-domain and is free to everyone to use for any purpose."[5] It is, as the name suggests, a lightweight relational database management system (RDBMS). It comes with a compiled size of less then 1MB which is rather small compared to most other RDBMS. While a lot of other RDBMS also come with a large list of dependencies for installed libraries SQLite is a C-language-library that only requires a few standard core C calls that should be available on any operating system. It can be build from a single source file and no server setup is required. SQLite can store its Databases as a regular file on disk. To use SQLite in an application all a developer has to do is a single include of the library and everything is ready to go. The setup process is incredibly easy and fast compared to the setup process of for instance PostgreSQL. These advantages make SQLite the most used database engine in the world. Every smartphone, tablet, most wearables and all of todays common browsers as well as some web backend systems use SQLite in some way. In early development of applications SQLite is often used for local testing purposes and will be replaced later by a larger scale RDBMS. In most cases this is possible because SQLite, despite being lightweight, still features most SQL functionalities. The focus of SQLite is on local storage. It can be used as a comfortable alternative for `fread()/fwrite()`. The developers even claim it to be faster in some cases [5].

3.1 Internal Structure

SQLites internals are not always clearly separable which is probably due to its claim for compactness and backwards compatibility. Nevertheless there are 3 basic parts in its architecture which is illustrated in figure 3.1.

The main part is the core (green) of SQLite with the interface (API), the SQL command processor and the virtual machine. The SQL command processor contains the SQL compiler (red). Statements running through the tokenizer will be broken up into tokens representing SQL keywords. These will be given to the parser which evaluates the statement and calls functions to allocate table structures. The parsed

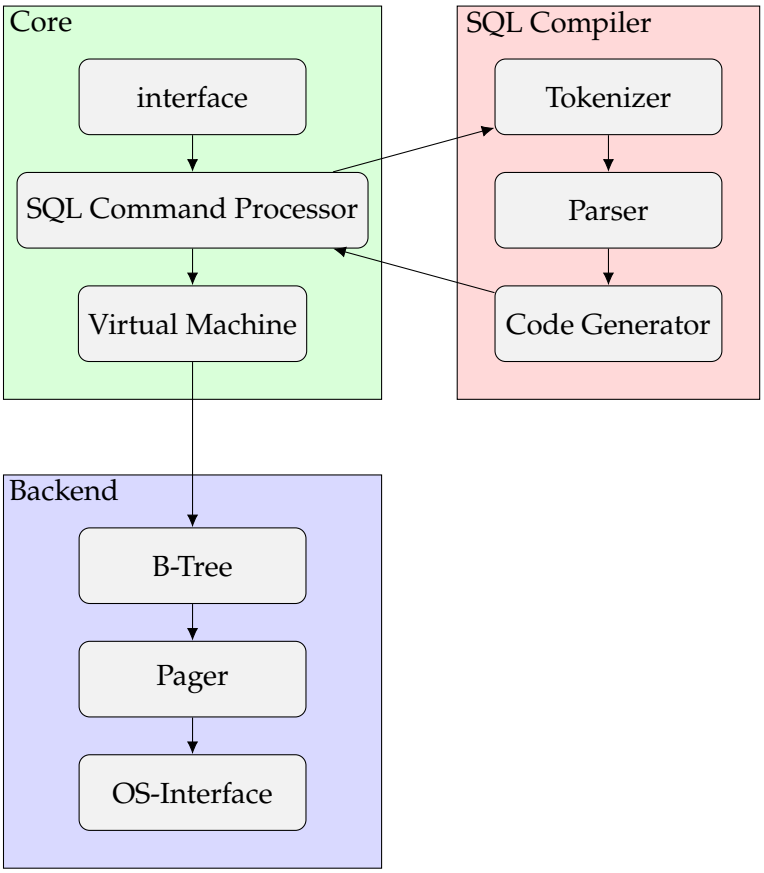

Figure 3.1: SQLite internal structure (rebuild according to [5, arch.html])

statement will then be converted to a virtual machine code with the code generator. SQLite uses its own Bytecode which can be run by its virtual machine (in SQLite called Virtual Database Engine (VDBE)). The Bytecode also functions as a way to describe what SQL statements are compiled into (`EXPLAIN`). Note that the VDBE might seem like a nice way to assemble or test new features but the bytecode is not meant to be written directly. Instead it is a result of the code generator and works on structures allocated mostly while parsing the sql statements `FROM` section. The investigation of the code did not yield any way of directly creating any working VDBE-code for the intended purpose without deeply interfering with the core functionality of SQLite.

Finally the backend (blue) is all about storing and loading the data. The database is stored in B-trees in a single file on disk. It also handles paging/caching and has an abstract interface to the operating system to provide cross platform compatibility.

There will be no focus on the backend in this thesis although it is important for the performance and purposes of SQLite. Since the backend is actually clearly separated and does not interfere with the core much, no further investigation of the backend is necessary for the scope of this work. The same thing applies for the interface in the core (API) which is meant to provide ways to implement SQLite into applications and offers functions to send queries.

3.2 Implementation

3.2.1 Parser

SQlite uses a parser generator similar to yet another compiler compiler (yacc) and bison called Lemon. From today's perspective, the developers created their own parser generator mostly due to historic reasons, since the alternatives did not provide the desired features and robustness at that time. The grammar of SQLite is written in a Lemon parser template. Lemon provides a C-code implementation of a parser for the given grammar.
The grammar consist of a lot of rules and special cases present in SQL. Some of the most relevant parts for this thesis are provided in the listings 3.2 and 3.3.

```
////////////////////////// The SELECT statement /////////////////////////
cmd ::= select(X).  {
  SelectDest dest = {SRT_Output, 0, 0, 0, 0, 0};
  sqlite3Select(pParse, X, &dest);
  sqlite3SelectDelete(pParse->db, X);
}
. . .
%type oneselect {Select*}
%destructor oneselect {sqlite3SelectDelete(pParse->db, $$);}
. . .
oneselect(A) ::= SELECT distinct(D) selcollist(W) from(X) where_opt(Y)
                 groupby_opt(P) having_opt(Q)
                 orderby_opt(Z) limit_opt(L). {
  A = sqlite3SelectNew(pParse,W,X,Y,P,Q,Z,D,L);
}
```

Listing 3.2: SQLite Parser snipped for `SELECT` [6, parser.y, 440-544]

To elaborate on this, since the documentation is sparse at best, the code describes the

SELECT statement of a query from a parser pont of view. Basically what happens is the definition of how a **SELECT** statement can be written and what happens for each case. Here it defines the construction of the select structure in `sqlite3SelectNew(...)` which allocates space and sets pointers to corresponding structures. Furthermore is the declaration for the result destination (and type) and the destructor also found here. Most importantly, this is where the function call for the outer **SELECT** statement, as well as for some cases found in the **FROM** statement, come from (`sqlite3Select()`). Its important to note that these are independent calls with separate structures. Listing 3.3 shows a snipped of the **FROM** statement which makes subqueries in the **FROM** statement possible. The subquery, represented by **select**(S), uses the same definition described above inside of parentheses. It may have an optional alias and/or a **HAVING** or **USING** part. `sqlite3SrcListAppendFromTerm()` appends tables to the list of sources that make up the **FROM** clause. The same function is called in case of a subquery. The link, responsible for the dependency, is established in this function as will be elaborated in section 3.2.2. For a regular table, the name of the table and its database would sit where the two zeros are located here. Instead a pointer to the SELECT structure of the subquery (S) is passed. Note that the subquery does not know its parent.
It should be mentioned that not the entire SQL statement is translated directly into C-code like this, on the contrary, most other things are put into an expression tree for further processing by the code generator.

```
. . .
seltablist(A) ::= stl_prefix(A) LP select(S) RP
            as(Z) on_opt(N) using_opt(U). {
    A = sqlite3SrcListAppendFromTerm(pParse,A,0,0,&Z,S,N,U);
}
. . .
```

Listing 3.3: Parser (Lemon) snipped for **FROM**-subqueries [6, parser.y, 656-659]

3.2.2 The FROM clause

The most relevant part for this thesis is the **FROM** clause. As indicated in section 3.2.1 SQLite builds a source list by appending tables or subqueries in `sqlite3SrcListAppendFromTerm()` with the important lines shown in listing 3.4. The function call `sqlite3SrcListAppend(...)` mostly allocates space if necessary, before appending a new source item with its names for database and table, if avail-

able. A pointer to the subquery is set in the last (new) item of the source list. This way SQLite can later make sure that all necessary **FROM** clause data is ready before evaluating the parent query.

```
. . .
p = sqlite3SrcListAppend(pParse, p, pTable, pDatabase);
. . .
pItem = &p->a[p->nSrc-1];
. . .
if( pAlias->n ){
    pItem->zAlias = sqlite3NameFromToken(db, pAlias);
}
. . .
pItem->pSelect = pSubquery;
pItem->pOn = pOn;
pItem->pUsing = pUsing;
return p;
. . .
```

Listing 3.4: `sqlite3SrcListAppendFromTerm` [6, build.c, 4613-4658]

The source list itself is a **struct** containing a counter to keep track of the number of items held in a dynamically growing array. `sqlite3SrcListAppend(...)` handles the growth of the source list to keep it from growing into other data. Each of the items in the list contain a specialized **struct Table** which holds information necessary to later locate and access its data in the database.

3.2.3 SELECT statement

The **SELECT** statement is the core of every SQL query. While most parts are optional, a query to the database should always contain **SELECT** (or synonyms in some RDBMS). A query can be as simple as **SELECT** 1;. For the most part, other SQL keywords refer to the selected results in some way. This is why SQLite uses the `sqlite3select(...)` function as the center piece of its query implementation. As has been shown in section 3.2.1, the parser itself calls it after evaluating the query text. At its core, this function itself already contains over a thousand lines of code while also branching into many subfunctions and all over SQLite in its role as the center piece. Therefore, the focus will only be on the essential parts for this thesis.
The first important thing happening is the call of the function in listing 3.5.

```
/*
** This routine sets up a SELECT statement for processing.  The
** following is accomplished:
**
**     *  VDBE Cursor numbers are assigned to all FROM-clauseterms.
**     *  Ephemeral Table objects are created for allFROM-clausesubqueries.
**     *  ON and USING clauses are shifted into WHERE statements
**     *  Wildcards "*" and "TABLE.*" in result sets areexpanded.
**     *  Identifiers in expression are matched to tables.
**
** This routine acts recursively on all subqueries within theSELECT.
*/
void sqlite3SelectPrep(
  Parse *pParse,         /* The parser context */
  Select *p,             /* The SELECT statement being coded. */
  NameContext *pOuterNC  /* Name context for container */
){
  assert( p!=0  pParse->db->mallocFailed );
  if( pParse->db->mallocFailed ) return;
  if( p->selFlags & SF_HasTypeInfo ) return;
  sqlite3SelectExpand(pParse, p);
  if( pParse->nErr  pParse->db->mallocFailed ) return;
  sqlite3ResolveSelectNames(pParse, p, pOuterNC);
  if( pParse->nErr  pParse->db->mallocFailed ) return;
  sqlite3SelectAddTypeInfo(pParse, p);
}
```

Listing 3.5: `sqlite3SelectPrep` [6, select.c, 5263-5276]

A lot of work is done to make the things mentioned in the comments happening, some of which will be examined in more detail later. `sqlite3SelectExpand()` unfolds the subqueries present and walks down their parse trees. This again leads to a lot of code, some of which is important, but most of it would contribute little towards the goal of this thesis and is therefore not elaborated in much detail. Figure 3.6 lists the callstack for a simple query with a **FROM** subquery to the `sqlite3SelectPrep` of the subquery. This callstack shows how the expansion of the subquery in the **FROM** part even leads to a recursive API call to compile the subquery, including the parser, ending up in the same spot again. This recursion will go as deep as the query is nested, possibly branching out for multiple **FROM** subqueries. While this works great for SQLite, one might already suspect this was not build with something like **LATERAL JOIN** in mind. The expansion of the subquery also assigns the VDBE cursor numbers to the source list items. They are used to read table rows later.

`sqlite3ResolveSelectNames()` is the next and probably most relevant call in the preparation function, if the goal is to make **LATERAL JOIN** possible. This is where "Identifiers in expression are matched to tables." [6, select.c 5259] Which means this is where an error would occur if access to a column or table was requested that is not known in this context. One parameter is `NameContext *pOuterNC` which is used to provide the name context of parent

```
sqlite3SelectPrep(...)
sqlite3Select(...) // SELECT
yy_reduce(...)
sqlite3Parser(...)
sqlite3RunParser(...) // PARSER
sqlite3Prepare(...)
sqlite3LockAndPrepare(...)
sqlite3_prepare_v2(...) // API
sqlite3_exec(...)
sqlite3InitOne(...)
sqlite3Init(...)
sqlite3ReadSchema(...)
sqlite3LocateTable(...)
sqlite3LocateTableItem(...)
selectExpander(...)
sqlite3WalkSelect(...)
selectExpander(...)
sqlite3WalkSelect(...)
sqlite3SelectExpand(...) // EXPAND
sqlite3SelectPrep(...)
sqlite3Select(...) // SELECT
yy_reduce(...)
sqlite3Parser(...)
sqlite3RunParser(...) // PARSER
sqlite3Prepare(...)
sqlite3LockAndPrepare(...)
sqlite3_prepare_v2(...) // API
shell_exec(...)
runOneSqlLine(...)
process_input(...)
main(int argc, char ** argv)
```

Figure 3.6: callstack for simple subquery

queries to subqueries (of some type).

SQLite supports standard SQL queries like the following with no issue.

```
SELECT col1 FROM table1 WHERE (SELECT 1 FROM table2 WHERE col1 = col2);
SELECT col1, (SELECT col2 FROM table2 WHERE col1 = col2) FROM table1;
```

Note that in both of the queries the subquery uses the outer name context. The name `col1` is available for subqueries in the `SELECT` and the `WHERE` statement. The subqueries will be resolved within the parent query using its `pOuterNC`.

This access however would not work for a subquery in the `FROM` part, especially not in the way required for the behavior of a `LATERAL JOIN`. A look at the callstack in listing 3.7 for the first of the two queries also reveals quite a difference compared to the `FROM` part subquery. In both cases a lot is happening that does not show in the callstacks and the stack in figure 3.7 might look different even for a slightly changed query, but the different approach is visible. Please note that there is some VDBE code created in this callstack. This is something occurring all over the place in most parts of the code generator. VDBE lines might be created, changed and expected to be there at various places starting in `sqlite3Select()`.

```
sqlite3Select(...) // Subquery SELECT
sqlite3CodeSubselect(...) // VDBE
sqlite3ExprCodeTarget(...) // VDBE
sqlite3ExprCodeTemp(...) // VDBE
sqlite3ExprIfFalse(...) // col1=col2
sqlite3WhereCodeOneLoopStart(...)
sqlite3WhereBegin(...) // WHERE
sqlite3Select(...) // Outer SELECT
yy_reduce(...)
... // same as above
```

Figure 3.7: callstack for `WHERE` subquery

3.3 Virtual Database Engine

SQLite implements its own virtual machine called Virtual Database Engine (VDBE). The code generator uses the provided interface to create and modify bytecode for the VDBE. This bytecode represents the compiled SQL statement and is run by the VDBE. The readability of the bytecode makes it suitable for the `EXPLAIN` command to describe what the database engine will do for a certain query.

A simple query to select one column from a table will look like listing 3.8.

The bytecode has an opcode and up to five parameters. Most of the time the parameters are used to address a certain database, table, column or register. With `EXPLAIN` SQLite also writes comments in some cases which might help to understand

addr	opcode	p1	p2	p3	p4	p5	notes
0	Init	0	7	0	0	-	start prog., jump to addr 7
1	OpenRead	0	4	0	0	-	open cursor 0 to read table 4
2	Rewind	0	6	0	0	-	iterate table with cursor 0
3	Column	0	1	1	0	-	read col1 at cursor 0 in reg1
4	ResultRow	1	1	0	0	-	from reg1 use 1 regs as result
5	Next	0	3	0	1	-	builds loop with Rewind
6	Halt	0	0	0	0	-	Exit and close all cursors
7	Transaction	0	0	8	1	-	begin read transaction on db
8	TableLock	0	4	0	table1	-	lock for shared cache
9	Goto	0	1	0	0	-	jump to addr 1

Listing 3.8: VDBE bytecode for `EXPLAIN SELECT col1 FROM table1;`

the bytecode of more complex queries. In the following a closer look will be taken at some of the more important commands, found in almost any query. `OpenRead` creates a reading cursor (p1) for the table p2 in database p3. `Rewind` in combination with `Next` builds a loop to iterate over a table. `Rewind` works on a cursor (p1) and jumps to p2 when there is no more line to read. `Next` (or `Prev`) is the tail of the loop. It advances the cursor (p1) and, if successful, jumps to the top of the loop body (p3). Please note that this basically represents a for-each-loop over a table. There are a lot more opcodes available in the byteode engine, including commands to make things like suboutines possible, but there are also several restrictions. The subroutines that have been mentioned already are very limited. "The bytecode engine has no stack on which to store the return address of a subroutine. Return addresses must be stored in registers. Hence, bytecode subroutines are not reentrant." [5, opcode.html#subroutines_coroutines_and_subprograms] Please note that simple manipulations to the bytecode generally result in errors because the VDBE works on established structures by the code generator, sometimes assuming things are set up a certain way. Please also consider, that "The bytecode engine is not an API of SQLite. Details about the bytecode engine change from one release of SQLite to the next. Applications that use SQLite should not depend on any of the details found in this document." [5, opcode.html#executive_summary]

3.4 Drawbacks

Parts of this section gave good reasons to use SQLite as a RDBMS a lot of purposes. While SQLite certainly excells at some of its core goals there are also some serious drawbacks. The focus of SQLite clearly is on single user and SQL core functionality

and compactness. Larger scale applications or databases should always get off better when they use a modern server based RDBMS like PostgreSQL. There is also a difference in comfort after the initial setup. SQLite is easy in its setup, but it often lacks detailed error, debug and help messages as well as more advanced database functionalities and some modern SQL language extensions like `LATERAL JOIN`. Considering implementation and documentation from the point of view of a developer, opinions may vary, but the investigations in the scope of this thesis led to the conclusion that SQLite lags behind in these aspects. It should be mentioned that this refers to the code base. SQLite left the impression of a grown project where comments similar to the one in listing 3.9 are no rare occurrence:

```
/* [...] countless applications developed over the years
** have made baseless assumptions about column names and will
** break if those assumptions changes. Hence, use extreme caution
** when modifying this routine to avoid breaking legacy. [...]}
```

Listing 3.9: Code comment example [6, select.c 1844-1847]

Backwards compatibility and adjustments to support former bug behavior or assumptions like the one mentioned here, are great efforts to keep old applications from breaking. At the same time, they are not great for the structure, comprehensibility and adaptability of the code.

4 LATERAL JOIN FOR SQLITE

The previous chapters have explained the concept of **LATERAL JOIN** and presented SQLite and it's code. The essence of this thesis is an implementation of **LATERAL JOIN** for SQLite. This chapter will elaborate on this implementation approach. To avoid duplications, some relevant details of SQLite will also be further examined in addition to chapter 3.

4.1 Implementation

The goal was to implement a SQLite version of **LATERAL JOIN** without disrupting existing functionalities. Since this is not an additional function that can be added in a module but is rather having an impact on core behavior of SQLite, a deep dive into the code was necessary. The basic idea is to extract the tables of the parents **FROM** clause, add them to the tables available to the subquery and force the correct behavior for a lateral join onto the code generator.

4.1.1 LATERAL Keyword

The keyword for **LATERAL JOIN** is not known to SQLite. To achieve the implementation goal, making the parser accept the wanted queries is a first step. The SQL keywords used in the parser are actually not found in the SQLite source code directly, instead an external, standalone tool is used to store them in a hash table and provide a function to translate them into parser tokens. It has to be compiled and run to make it available to the SQLite parser.

Listing 4.1 shows this keyword structure as well as the addition of the **LATERAL** keyword to the table. Thus it shows, to make the **LATERAL** keyword available it is added as a join token just like **LEFT** or other join keywords.
While this makes the parser recognize the LATERAL keyword as a join type, it does not make it ready for use yet. The SQLite parser implements `joinop` (Listing 4.2) as the separator to appear between the content listed in the **FROM** clause. A few options are available here. The default behavior of **INNER JOIN** is chosen when a comma or

```
struct Keyword {
  char *zName;     /* The keyword name */
  char *zTokenType; /* Token value for this keyword */
  int mask;        /* Code this keyword if non-zero */
  int id;          /* Unique ID for this record */
  int hash;        /* Hash on the keyword */
  int offset;      /* Offset to start of name string */
  int len;         /* Length of this keyword, not counting final \000 */
  int prefix;      /* Number of characters in prefix */
  int longestSuffix;/* Longest suffix that is a prefix on another word */
  int iNext;       /* Index in aKeywordTable[] of next with same hash */
  int substrId;   /* Id to another keyword this keyword is embedded in */
  int substrOffset; /* Offset into substrId for start of this keyword */
  char zOrigName[20];  /* Original keyword name before processing */
};
. . .
static Keyword aKeywordTable[] = {
. . .
{ "LATERAL",          "TK_JOIN_KW",      ALWAYS                },
{ "LEFT",             "TK_JOIN_KW",      ALWAYS                },
. . .
```

Listing 4.1: Keyword hash table item structure [6, ../tool/mkkeywordhash.c 35-49]

```
%type joinop {int}
joinop(X) ::= COMMA|JOIN.              { X = JT_INNER; }
joinop(X) ::= JOIN_KW(A) JOIN.
                {X = sqlite3JoinType(pParse,&A,0,0);  /*X-overwrites-A*/}
joinop(X) ::= JOIN_KW(A) nm(B) JOIN.
                {X = sqlite3JoinType(pParse,&A,&B,0); /*X-overwrites-A*/}
joinop(X) ::= JOIN_KW(A) nm(B) nm(C) JOIN.
                {X = sqlite3JoinType(pParse,&A,&B,&C);/*X-overwrites-A*/}
```

Listing 4.2: Keyword hash table item structure [6, ../tool/mkkeywordhash.c 35-49]

just the **JOIN** keyword is used. Alternatively the join type can be specified with up to three identifiers listed before the JOIN keyword. In listing 4.2 it can be seen that the same C-code function is called in each case with an identifier. This function sets flags for join description and decides if a join of this type is possible. Not all join types are available in SQLite. Appart from lateral joins, there will also be an error thrown for outer or right joins.

```
if( (jointype & JT_OUTER)!=0
    && (jointype & (JT_LEFT|JT_RIGHT))!=JT_LEFT ){
    sqlite3ErrorMsg(pParse,
        "RIGHT and FULL OUTER JOINs are not currently supported");
    jointype = JT_INNER;
}
```

Listing 4.3: Unsupported **OUTER** and **RIGHT** joins in SQLite [6, select.c 265-270]

The return value `jointype` is an integer used as the join descriptor containing all flags set. However, this might be missleading since there is only a `u8` (eight bit) in place to store the join type later [6, sqliteInt.h 2690]. Fortunately, there still is a single bit unused which can be claimed for the lateral join type, without enlarging one of the core structures for a single bit.

```
#define JT_LATERAL   0x0040    /* The lateral keyword is present */
```

With this flag, tables or rather source list items, can be identified as part of a lateral join later in the code. On a sidenote, while the join operators are handled by the right join partner throughout most of the code, they are initially attached to the left operand that is followed by the join description. This is only relevant in the buildup of the source list structure. With the flag defined and the deciding function identified, the keyword can be made available. Quite a few things in SQLite are implemented unintuitively to some degree, probably due to optimization purposes. Such a case can be seen in listing 4.4 where all allowed keywords are put into a single overlapping string. The position and length in that string, as well as the flags to be set if that keyword is found, are defined in an array (`aKeyword[]`). To fully make **LATERAL** a valid keyword for joins it has to be added to the `zKeyText[]` and `aKeyword[]` accordingly. Having achieved that, no error will occur for a query like
`SELECT tbl1.col1, tbl2.col2 FROM tbl1 LATERAL INNER JOIN tbl2;`
However, the behavior would still be that of a normal **INNER JOIN** (which is correct

```
static const char zKeyText[] =
    "naturaleftouterightfullinnercrosslateral";
static const struct {
  u8 i;        /* Beginning of keyword text in zKeyText[] */
  u8 nChar;    /* Length of the keyword in characters */
  u8 code;     /* Join type mask */
} aKeyword[] = {
  /* natural */ { 0,  7, JT_NATURAL                },
  /* left    */ { 6,  4, JT_LEFT|JT_OUTER          },
  /* outer   */ { 10, 5, JT_OUTER                  },
  /* right   */ { 14, 5, JT_RIGHT|JT_OUTER         },
  /* full    */ { 19, 4, JT_LEFT|JT_RIGHT|JT_OUTER },
  /* inner   */ { 23, 5, JT_INNER                  },
  /* cross   */ { 28, 5, JT_INNER|JT_CROSS         },
  /* lateral */ { 33, 7, JT_LATERAL                },
};
```

Listing 4.4: Addition of LATERAL to keyword check in `sqlite3JoinType()` [5, select.c]

in this case).

4.1.2 The Source List

Changing the fundamentals of SQLite to support **`LATERAL JOIN`** would lead to a huge followup of changes throughout the code, each with the risk of breaking intended (or unintended legacy) behavior. Therefore, a less invasive path was taken. SQLite internally represents the **`FROM`** clause by a structure called source list (`srcList`). Chapter 3 already touched the subject of the buildup of the source list. The source code for the `srcList` structure can be found in full length with documentation in the appendix listing 2. After investigating other options, expanding the source list to support lateral access has turned out to be the most promising approach. Efforts have been made to minimize impact on queries that do not use **`LATERAL JOIN`**. To achieve this, a recursive pointer was added to the list.

```
struct SrcList* pLateral; // recursive srcList for lateral, can be NULL!
```

This pointer would just be a `NULL` pointer for regular queries which should barely affect performance and memory requirements. For developers it might be important to mention that additions to the `SrcList` must be made above the `SrcList_item a[1]` array. Otherwise it will grow into and overwrite content in lists with more than

one item. This is due to the implementation of `sqlite3SrcListAppend(...)` which was already pointed out in chapter 3. It enlarges the allocated space for the entire `SrcList` and uses it for the array starting at its declaration. For lateral queries this `pLateral` pointer is intended to hold the list of sources that should be available in a subquery due to the **LATERAL** keyword. An advantage of reusing the entire source list structure is that some of the existing code for it can also be reused. This introduces a way to add sources to a queries **FROM** term that can be treated separately. Section 3.2.2 introduced the function that builds the list of sources for a query (`sqlite3SrcListAppendFromTerm(...)`). Listing 4.5 shows how the buildup for the lateral source list was added to this function. The process is only initiated if the

```
if(pSubquery && p->nSrc > 1 &&
        (p->a[p->nSrc-2].fg.jointype & JT_LATERAL) != 0){
  Token t;
  sqlite3TokenInit(&t, p->a[p->nSrc-2].zName);
  SrcList *pLateral = 0;
  Select* pCurSel;
  for(int i=0; i<p->nSrc-1; i++){
    pCurSel = p->a[i].pSelect;
    sqlite3TokenInit(&t, p->a[i].zName);
    pLateral = sqlite3SrcListAppend(pParse, pLateral, &t, pDatabase);
    pLateral->a[pLateral->nSrc-1].pSelect = pCurSel;
    pLateral->a[pLateral->nSrc-1].zAlias =
            sqlite3DbStrDup(pParse->db, p->a[i].zAlias);
    pSubquery->pSrc->latItems++;
    pSubquery->pSrc->srcFlags |= SLF_LateralInf;
    p->a[i].usedInLateral = 1;
  }
  pSubquery->pSrc->pLateral = pLateral;
  sqlite3SrcListInferLateralInfluence(pSubquery->pSrc);
}
```

Listing 4.5: Buildup of the lateral source list in `sqlite3SrcListAppendFromTerm(...)`

source that is currently added is a subquery, not the only element in the **FROM** clause and joined with the **LATERAL** keyword. The information about the sources leading up to the lateral join (to the left of it) is copied and added to the subqueries lateral source list. The flag `SLF_LateralInf` is set for the subqueries source list and also recursively propagated through to nested queries. This flag was added to the available flags for source lists and serves as a marker to recognize the possible influence of a **LATERAL** keyword to a queries behavior.

4.1.3 Cursors

SQLite manages access to the data of a table in the database file with cursors. A Cursor points to a line (table row) in the file and can be advanced forward or backward. There can be multiple cursors pointing independently at the same table or line but every item in the source list can only have one cursor (see `SrcList` in listing 2). The cursor related to a source list item is identified by its index(`iCursor`) which is part of each source list item. Call to mind that the desired behavior is to execute a lateral subquery for every row of the preceding (joined) sources. This requires reading from the same cursor. The following example should clarify this.

```
SELECT A.x,S.z FROM A LATERAL JOIN (SELECT B.z WHERE B.y = A.x) AS S;
```

This query should be executed with two nested loops. One iterating over table `A` and the inner loop iterating over table `B` while accessing the same row `A.x` of table `A`. Applying this to the cursors used by SQLite, the lateral subquery is supposed to use the cursors of the outer query to access its tables while executing its own code nested in the parents loop. Therefore, it is necessary to set the cursors for the lateral source list accordingly which is done in `sqlite3SrcListAssignCursors(...)`. This function would assign each source list item its cursor number and does this for subqueries recursively. It was extended to call `sqlite3LateralSrcListAssignCursors(...)` if there are lateral sources available. Listing 4.6 shows how the cursor numbers are copied from their outer query counterpart.

```
void sqlite3LateralSrcListAssignCursors(SrcList *pList,
                                        SrcList* pOuterSrc){
  int i;
  struct SrcList_item *pItem, *pOuter;
  assert(pList || pList->latItems >0);
  if( pList ){
    for(i=0, pItem=pList->a, pOuter=pOuterSrc->a; i<pList->nSrc;
        i++, pItem++, pOuter++){
      if( pItem->iCursor>=0 ) break;
        pItem->iCursor = pOuter->iCursor;
    }
  }
}
```

Listing 4.6: `sqlite3LateralSrcListAssignCursors()` to copy corresponding cursor ids

With the correct cursors in place it still has to be made sure they are open and available to read, as well as ensuring to run the query in a structure as it has been described above.

4.1.4 Name Resolution

Access to the namespace of the outer query is what **LATERAL JOIN** is for. A lot of the work is done to make the essential name resolution possible. The name resolution has its own dedicated file in the SQLite source code containing "routines used for walking the parser tree and resolve all identifiers by associating them with a particular table and column" [6, resolve.c]. Following several attempts the chosen approach in listing 4.7 was developed.

```
int latI = p->pSrc->latItems;
if(latI > 0 && p->pSrc->pLateral->nSrc > 0)
{
  struct SrcList *pLateral = p->pSrc->pLateral;
  for (i=0; i<latI; i++)
  {
    Table *pTab;
    if(pLateral->a[i].pSelect){
        sqlite3ExpandSubquery(pParse, &pLateral->a[i]);
        pLateral->a[i].pSelect = 0;
        pLateral->a[i].pTab->nTabRef++;
    }else{
      pTab = pLateral->a[i].pTab = sqlite3LocateTableItem(pParse,
                                    LOCATE_NOERR,&pLateral->a[i]);
      if( pTab ){
          pTab->nTabRef++;
      } // [...] explained below in with recursive
    }
  }
// [...] separate listing
}
```

Listing 4.7: Addition to name resolution for lateral sources

This code will initiate a search for the "in-memory structure" of a table in the database when lateral sources are available and it is not a query. Provided with the required information extracted earlier this can be done by `sqlite3LocateTableItem(...)` [6, build.c 403 ff.]. In case of a subquery, it is

made sure to be expanded which should then provide necessary resolutions allowing the pointer in the lateral source to be removed. With the table structures in place the actual name resolution can be done. Usually this would only happen within the scope of the regular SQL access rules. To get around this, lateral subqueries are pretended to have an outer query providing the requested tables. A new instance `NameContext lateralNC` will provide the required context for this. This namespace is set up as shown in listing 4.8 to use the lateral source list as the pretended outer queries sources. The pointer `pNext` for the next outer context is set to the actual parent query to allow for nestings. This is probably `NULL` for most queries.

```
  memset(&lateralNC, 0, sizeof(lateralNC));
  lateralNC.pParse = pParse;
  lateralNC.pWinSelect = p;
  lateralNC.ncFlags = NC_AllowAgg|NC_AllowWin;
  lateralNC.pSrcList = pLateral;
  lateralNC.pNext = pWalker->u.pNC;
  pOuterNC = &lateralNC;
```

Listing 4.8: Setup of the outer name context for lateral subqueries

Keep in mind that only the namespace of the outer query is made available through this setup, it will not actually behave like a regular subquery with dependencies. Listing 4.9 lists the most important resolve calls using this name context where `sNC` stands for the name context of the current query with `pOuterNC` set as the next outer context (`sNC.pNext = pOuterNC`).

```
  . . .
/* Recursively resolve names in all subqueries */
for(i=0; i<p->pSrc->nSrc; i++){
  . . .
    pItem->pSelect->pParent = p; // For lateral in WITH RECURSIVE
    sqlite3ResolveSelectNames(pParse, pItem->pSelect, pOuterNC);
  . . .
/* Resolve names in the result set. */
if( sqlite3ResolveExprListNames(&sNC, p->pEList) ) return WRC_Abort;
  . . . The following are to resolve names in Having and Where clauses
if( sqlite3ResolveExprNames(&sNC, p->pHaving) ) return WRC_Abort;
if( sqlite3ResolveExprNames(&sNC, p->pWhere) ) return WRC_Abort;
  . . .
```

Listing 4.9: Resolving with lateral context in `resolveSelectStep` [6, resolve.c 1418-1653]

While the first call is to recursively resolve entire subqueries, the rest resolves expression trees for parts of a query. When a colum name is to be resolved `sqlite3ResolveExprNames(...)` leads to a call of `lookupName(...)`. This function can take identifiers of the types common in SQL which in its full form of `database.table.column` is always a unique name. The main task of `lookupName(...)` is to use that information and what is available in the source list to fill essential details in the expression that will later be translated into VDBE bytecode to access a column. This includes the database id, the cursor number assigned earlier, the located table structure, and the number of the column in that table. With the provided outer name context for lateral queries the information of the outer query can be used to resolve names. Inner name contexts are looked through first and a successful find stops the search. Name ambiguities are therefore possible in SQLite and solved by prioritizing the innermost context that contains the name.

4.1.4.1 With Recursive

The additions described so far work for many query types but queries using `WITH RECURSIVE` are handled slightly different in some parts of SQLite. The function used to locate the table in the memory does not work in this case, nor is there any that would. Direct access from the parent query seems to be the only way to find the information. Therefore, the pointer in listing 4.9 is set before the recursive call for name resolution in the subquery happens. Furthermore, if the table could not be found as described above, a search for this special case is initiated, looping through all parents and their sources, prioritizing the innermost. The actual name comparison is provided in listing 4.10 where `parSel` is the currently searched parent. In addition, the flag for correlated queries can prevent materialization (see 4.1.5.1). These changes allow the name resolution in recursive queries with the added lateral support.

```
const char *zTabName = parSel->pSrc->a[srcNum].pTab->zName;
const char *zLatTabName = pLateral->a[i].zAlias ?
                    pLateral->a[i].zAlias : pLateral->a[i].zName;
if(sqlite3StrICmp(zTabName, zLatTabName) == 0){
  pTab = pLateral->a[i].pTab = parSel->pSrc->a[srcNum].pTab;
  break;
}
```

Listing 4.10: Locating table structures of recursive queries for name resolution

4.1.5 Optimizations

With the names resolved, the desired data can be accessed for lateral joins, provided that the access happens in right position, order and scope. None of this is guaranteed by SQLite, instead optimizations might outsource operations into subroutines without cross access, change the order of loops in such a way that the subquery might represent the outer loop or it might not even create a loop or load the necessary data. All of this can happen because the system still does not know how a lateral join works. There are pointers in place to access the information it should, but there are no rules that would enforce the correct behavior nor would the changes allow SQLite to setup and check everything the way it does for built in behavior. Forcing the correct order of execution can be done by setting dependencies similar to what a **LEFT JOIN** would do. The join type flag `JT_LATERAL` introduced in 4.1.1 can be used for this. The most important step for this is presented in listing 4.11, where the preceding items in the source list are marked as required for this. This does not allow to create **WHERE** loop variants with the wrong order for the purposes of **LATERAL JOIN**.

```
if( ((pItem->fg.jointype|priorJointype) &
     (JT_LEFT|JT_CROSS|JT_LATERAL))!=0 ){
  /* This condition is true when pItem is the FROM clause term on the
  ** right-hand-side of a LEFT, CROSS or LATERAL JOIN.  */
  mPrereq = mPrior;
}
```

Listing 4.11: Setting dependencies in `whereLoopAddAll(...)` [6, where.c 3568-3572]

To ensure that all requested columns are actually accessible, the following line will force all columns of a table to be loaded if it hase been made available via the **LATERAL** keyword. To recognize this property, the flag `usedInLateral` of source list items was introduced (see appendix listing 2) and set while building the source list shown in listing 4.5.

```
if(pTabItem->usedInLateral) n = pTabItem->pTab->nCol;
```

Listing 4.12: Load all columns inserted in `sqlite3WhereBegin(...)` [6, where.c]

4.1.5.1 Flattening

In the processing of regular SQL queries a full scan is not always needed or even inefficient. SQLite might therefore decide to simplify the structure. A subquery might for example end up like in listing 4.13 (also see listing 1 in the appendix).

addr	opcode	p1	p2	p3	comment
0	Init	0	22	0	Start at 22
1	Integer	10	1	0	r[1]=10
2	Once	0	10	0	materialize "subquery_1"
3	Explain	3	0	0	MATERIALIZE 1
4	OpenEphemeral	1	1	0	nColumn=1
5	Explain	5	3	0	SCAN CONSTANT ROW
6	Integer	42	2	0	r[2]=42
7	MakeRecord	2	1	3	r[3]=mkrec(r[2])
8	NewRowid	1	4	0	r[4]=rowid
9	Insert	1	3	4	intkey=r[4] data=r[3]
10	Return	1	0	0	end subquery_1
11	OpenRead	0	5	0	root=5 iDb=0; numbers
12	Explain	12	0	0	SCAN SUBQUERY 1
13	Rewind	1	21	0	
14	Explain	14	0	0	SCAN TABLE numbers
15	Rewind	0	21	0	
16	Column	0	0	5	r[5]=numbers.num
17	Column	1	0	6	r[6]=subquery_1.a
18	ResultRow	5	2	0	output=r[5..6]
19	Next	0	16	0	
...	...	...	...	...	...

Listing 4.13: `EXPLAIN SELECT num,a FROM numbers, (SELECT 42 AS a);`

What happens here is the materialization of the subquery. SQLite calculates the subquery once, before the main loop starts and provides the results in a temporary table `subquery_1`. In this case there is only one column and one row in this "ephemeral" table. This is not compatible with the way lateral subqueries are executed and therefore should not be allowed. It might be worth mentioning that this behavior only has to be prevented if the subquery should have access to a lateral source. While this is mostly suppressed by the dependencies set in listing 4.11, there are several similar code structures that would not allow accessing the parents columns or behave contradicting to the lateral rules. For some queries, there might not be a reason to have an extra loop. The example above could have been run in a single loop over the table `numbers` (which it effectively does due to the single execution of

the outer loop). The much shorter bytecode in listing 4.15 would yield the same results. The `Integer` could even be set in the initiation once (example in appendix listing 1).

addr	opcode	p1	p2	p3	p4	comment
0	Init	0	8	0		Start at 8
1	OpenRead	0	5	0	1	root=5 iDb=0;numbers
2	Rewind	0	7	0		
3	Column	0	0	1		r[1]=numbers.num
4	Integer	42	2	0		r[2]=42
5	ResultRow	1	2	0		output=r[1..2]
6	Next	0	3	0		
7	Halt	0	0	0		
8	Transaction	0	0	8	0	usesStmtJournal=0
9	TableLock	0	5	0	numbers	iDb=0 root=5 write=0
10	Goto	0	1	0		

Listing 4.14: Manually flattened bytecode for `SELECT num,a FROM numbers, (SELECT 42 AS a);`

The process of removing loops for queries in SQLite is called flattening. A higher level view on this would be the absorption of subquery code into the parent query.

```
SELECT a FROM (SELECT x+y AS a FROM t1 WHERE z<100) WHERE a>5; (orig.)
SELECT x+y AS a FROM t1 WHERE z<100 AND a>5;  (same query flattened)
```

Listing 4.15: Flattening explanation example [6, select.c 3562,3574]

This kind of optimization is done in `flattenSubquery(...)` which is called in `sqlite3Select(...)` (see 3.2). There is a long list of restrictions under which this will operate which can be found in the documentation of the function [6, select.c 3555-3707]. One of those restrictions is the requirement of a `FROM` clause in the subquery. This line is extended to bypass this restriction and allow flattening for queries with lateral influence. Adding the lateral sources will allow SQLite to use

```
if( pSubSrc->nSrc + pSubSrc->latItems ==0 &&
    (pSubSrc->srcFlags & SLF_LateralInf)==0) return 0;
```

Listing 4.16: Extended flattening restriction (7) [6, select.c 3756]

this powerful optimization even if the subquery does not have a `FROM` clause of its own. Furthermore, this allows access to code that updates cursor numbers in

subqueries. However, some adjustments have to be made for it to actually work with the added lateral sources. Since this would allow flattening on subqueries with empty source lists but lateral sources available, the handling of that case has to be added to the function. A lateral subquery should be executed once for every row of the preceding joined source items in the parent query. If the subquery had a source to be iterated, this would usually involve a loop. This is not true in this case and all of the identifiers either have to refer to the parents sources or are locally generated, which means most of the subquery can be deleted, shown in listing 4.17. Large parts of the flattening code either are not executed in this case due to the lack

```
if((pSubSrc->srcFlags & SLF_LateralInf) != 0 && pSubSrc->nSrc == 0) {
  isLateralJoin = 1;
  if( pSubitem->fg.isIndexedBy )
sqlite3DbFree(db,pSubitem->u1.zIndexedBy);
  if( pSubitem->fg.isTabFunc )
    sqlite3ExprListDelete(db, pSubitem->u1.pFuncArg);
  sqlite3ExprDelete(db, pSubitem->pOn);
  sqlite3IdListDelete(db, pSubitem->pUsing);
  if(pSrc->nSrc-iFrom > 0){
    doShiftSrc = 1;
  }
  pSrc->nSrc--;
}
```

Listing 4.17: Flattening for lateral queries

of source list items or work just fine with this setup. The only essential addition is to update cursor numbers in the lateral subquery with `substSelect(...)` [6, select.c 3527]. This is not necessary for non lateral queries since they do not access preceding sources. Before the remaining structures are deleted, all source list entries to the right are shifted to fill up the now empty slot as can be seen in listing 4.18. This can lead to some really efficient queries. The example in listing 4.13 will actually end up exactly as demonstrated in listing 4.15 if the query is written with a **`LATERAL`** instead of a **`INNER JOIN`** (normal comma) although nothing else changes. In this example **`LATERAL`** would just enable the modified flattening optimization.

```
  // do the shift for a removed lateral join
if(doShiftSrc) {
  assert(pSrc->nAlloc > pSrc->nSrc);
  for(i=0; i<pSrc->nSrc-iFrom; i++){
    pSrc->a[iFrom+i] = pSrc->a[iFrom+i+1];
  }
}
```

Listing 4.18: Left shift to move up the remaining items in the list

4.2 Achievements

The implementation approach presented above allows the use of `LATERAL JOIN` in SQLite for a lot of query types. It was developed testing several types of lateral queries with a small testing database, some of which can be found in the appendix listing 3. Furthermore, the modified SQLite powered a database driven website, using lateral queries for some time with no issues. At the same time it has little impact on the core functionalities of SQLite because most of the code only comes into play when the `LATERAL` keyword is actually used. The SQLite tests were run several times during development to ensure the core code would not break. In addition, the support for flattening can even achieve better optimization in some cases (see 4.1.5.1). The query in listing 4.19 is a complicated way of calculating $42 + 1 = 43$ using a nested and chained lateral query will hold up as an example. Besides from

```
SELECT res.*
FROM   (SELECT 42 AS "x") LATERAL JOIN
       (SELECT result
        FROM ( (SELECT 0 AS "y_1") AS "let0"
              LATERAL JOIN
              ( (SELECT x AS y_2) AS "let1"
               LATERAL JOIN
               ( (SELECT y_2 + 1 AS "y_3") AS "let2"
                LATERAL JOIN
                  (SELECT "y_3" AS "result") AS "return3"
                ON True)
               ON True)
              ON True)) AS res;
```

Listing 4.19: Lateral testing query $(42 + 1 = 43)$

yielding the correct result, the VDBE code created, provided in listing 4.20, shows that no unnecessary loop code was created. The coroutine overhead is not required for this query to work but represent a minor potential for optimization that also occurs like this on regular SQLite queries.

addr	opcode	p1	p2	p3	comment
0	Init	0	12	0	Start at 12
1	InitCoroutine	1	5	2	subquery_1
2	Integer	42	2	0	r[2]=42
3	Yield	1	0	0	
4	EndCoroutine	1	0	0	
5	InitCoroutine	1	0	2	
6	Yield	1	11	0	next row of subquery_1
7	Copy	2	4	0	r[4]=r[2]; subquery_1.x
8	Add	5	4	3	r[3]=r[5]+r[4]
9	ResultRow	3	1	0	output=r[3]
10	Goto	0	6	0	
11	Halt	0	0	0	
12	Integer	1	5	0	r[5]=1
13	Goto	0	1	0	

Listing 4.20: VDBE code for query in listing 4.19

4.3 Problems

The majority of problems that occurred during development could be solved or bypassed, but there still remain unsolved issues with this implementation approach of `LATERAL JOIN`. First of all, as a SQL language supplement it is integrated into the core code of SQLite but is not really part of it. The attachment with separate structures to handle circumvents a lot of code that is in place for reasons like reliability, stability and security. In addition, `LATERAL JOIN` is not part of the SQLite test suite and can therefore not be tested in the same way.

This approach also did not produce a working SQLite version of `LATERAL LEFT OUTER JOIN` in the scope of this thesis. With the advanced flattening for lateral joins disabled, basic queries that do not take advantage of the lateral access, would still work. Access to the lateral sources will result in either a run time error or incorrect results. Since it does not provide any added value in this state and to communicate the lack of support for this join type, a message preventing the unsupported use was put in place, similar to the already present one shown in listing 4.3.

5 CONCLUSION

The presented implementation approach of lateral joins into SQLite, explained in chapter 4, allows for a variety of lateral queries to work. Adapted optimizations ensure the new query types still produce compact and efficient bytecode. However, it does not support `LEFT OUTER JOIN` and it can not offer the standard SQLite reliability for all lateral queries due to the method of its implementation. SQLite did not reveal satisfying extensibility of its core code during the research of this thesis. This is to large parts, due to the structure of the code with strong interlocking and dependencies between parser, code generator and VDBE while the limited documentation of procedures like the invalidation of pointers due to reallocations pose an additional hurdle for developers that are not already involved.

5.1 Related Work

Native support for lateral queries is already offered by some established RDBMS such as PostgreSQL, Oracle or MySQL. Therefore, it also finds its way into tools working with database systems. For example the source to source compilation for PL/SQL into pure SQL uses recursive queries with lateral joins [2, 1] to get rid of context switches. SQLite itself provides an API to create and load extensions that can offer additional functionalities such as application defined functions [5, appfunc.html]. Extensions using the API can be powerful tools, but they can not add SQL language constructs to SQLite.

5.2 Future Work

While the query test suggested a good coverage of working query types, extensive testing for lateral queries would be necessary to ensure the high reliability SQLite strives for, as an addition to the SQLite test harnesses with almost 92 million lines of code. The additional support for `LEFT OUTER LATERAL JOIN` could provide further query options. It should be possible to add support for SQLite into tools that require lateral queries, with the result of this thesis.

APPENDIX: LARGER CODE SNIPPETS

addr	opcode	p1	p2	p3	...	p5	comment
0	Init	0	19	0			Start at 19
1	OpenRead	0	5	0			root=5 iDb=0; numbers
2	Explain	2	0	0	SCAN TABLE numbers		
3	Rewind	0	18	0			
4	Column	0	0	1			r[1]=numbers.num
5	Integer	13	3	0			r[3]=13; return address
6	Once	0	13	0			
7	Explain	7	0	0	SCALAR SUBQUERY 1		
8	Null	0	4	4			r[4..4]=NULL ...
9	Integer	1	5	0			r[5]=1; LIMIT counter
10	Explain	10	7	0	SCAN CONSTANT ROW		
11	Add	7	6	4			r[4]=r[7]+r[6]
12	DecrJumpZero	5	13	0			if (–r[5])==0 goto 13
13	Return	3	0	0			
14	Ne	4	17	1		54	if r[1]!=r[4] goto 17
15	Column	0	0	8			r[8]=numbers.num
16	ResultRow	8	1	0			output=r[8]
17	Next	0	4	0		01	
18	Halt	0	0	0			
19	Transaction	0	0	8		01	usesStmtJournal=0
20	TableLock	0	5	0			iDb=0 root=5 write=0
21	Integer	42	6	0			r[6]=42
22	Integer	1	7	0			r[7]=1
23	Goto	0	1	0		00	

Listing 1: `EXPLAIN SELECT num FROM numbers WHERE num = (SELECT 42+1);`

```
/*
** The following structure describes the FROM clause of a SELECT statement.
** Each table or subquery in the FROM clause is a separate element of
** the SrcList.a[] array.
[...] */
struct SrcList {
  int nSrc;        /* Number of tables or subqueries in the FROM clause */
  int latItems;    /* Number of lateral tables, 0 if not allocated */
  u8 srcFlags;
  u32 nAlloc;      /* Number of entries allocated in a[] below */
  struct SrcList* pLateral; /* recursive srcList for lateral, can be NULL! */
  struct SrcList_item {
    Schema *pSchema;  /* Schema to which this item is fixed */
    char *zDatabase;  /* Name of database holding this table */
    char *zName;      /* Name of the table */
    char *zAlias;     /* The "B" part of a "A AS B" phrase.  zName is the "A" */
    Table *pTab;      /* An SQL table corresponding to zName */
    Select *pSelect;  /* A SELECT statement used in place of a table name */
    int addrFillSub;  /* Address of subroutine to manifest a subquery */
    int regReturn;    /* Register holding return address of addrFillSub */
    int regResult;    /* Registers holding results of a co-routine */
    struct {
      u8 jointype;      /* Type of join between this table and the previous */
      unsigned notIndexed :1;    /* True if there is a NOT INDEXED clause */
      unsigned isIndexedBy :1;   /* True if there is an INDEXED BY clause */
      unsigned isTabFunc :1;     /* True if table-valued-function syntax */
      unsigned isCorrelated :1;  /* True if sub-query is correlated */
      unsigned viaCoroutine :1;  /* Implemented as a co-routine */
      unsigned isRecursive :1;   /* True for recursive reference in WITH */
    } fg;
    int iCursor;      /* The VDBE cursor number used to access this table */
    Expr *pOn;        /* The ON clause of a join */
    IdList *pUsing;   /* The USING clause of a join */
    Bitmask colUsed;  /* Bit N (1<<N) set if column N of pTab is used */
    int usedInLateral; /* >0 if if the table is available in a lateral join */
    union {
      char *zIndexedBy;    /* Identifier from "INDEXED BY <zIndex>" clause */
      ExprList *pFuncArg;  /* Arguments to table-valued-function */
    } u1;
    Index *pIBIndex;  /* Index structure corresponding to u1.zIndexedBy */
  } a[1];             /* One entry for each identifier on the list */
};
```

Listing 2: Extended for `LATERAL`, original in `SrcList` [6, sqliteInt.h 2657-2708]

```
SELECT * FROM tbl1, prehistoric LATERAL JOIN
              (SELECT * FROM person WHERE person_id < tbl1.two) AS person;
SELECT * FROM (SELECT * FROM prehistoric) as dinos LATERAL JOIN
              (SELECT * FROM person WHERE dinos.legs = person_id);
SELECT * FROM tbl1 LATERAL JOIN (SELECT * From prehistoric
                   LATERAL JOIN (SELECT * FROM person,
                                (SELECT "hello!" as a WHERE a = tbl1.one)
                                  WHERE person_id < legs) AS person
                                WHERE prehistoric.legs < tbl1.two);
SELECT c.id, p.main_page_id
    FROM Content AS c LATERAL
           JOIN (SELECT p.title, p.main_page_id
                 FROM Content AS p
                 WHERE p.id=c.parent_id) AS p
         WHERE c.main_page_id IS NULL
             AND p.main_page_id IS NOT NULL;

WITH RECURSIVE
  cnt(x, z) AS (
     SELECT 1, 1
     UNION ALL
     SELECT cnt.x+1, y FROM cnt LATERAL JOIN (SELECT x AS y LIMIT 1)
      LIMIT 10
  )
SELECT x,z FROM cnt;

WITH RECURSIVE
  cnt(x, z) AS (
     SELECT 1, 0
     UNION ALL
     SELECT cnt.x+1, y.result FROM cnt LATERAL JOIN (SELECT res.*
FROM  (SELECT x AS "x") LATERAL JOIN
      (SELECT result
        FROM ( (SELECT 0 AS "y_1") AS "let0"
              LATERAL JOIN
              ( (SELECT x AS y_2) AS "let1"
               LATERAL JOIN
               ( (SELECT y_2 + 1 AS "y_3") AS "let2"
                LATERAL JOIN
                 (SELECT "y_3" AS "result") AS "return3")))
      ) AS res) AS y
      LIMIT 10
  )
SELECT x,z FROM cnt;
```

Listing 3: A small selection of tested development queries that all work on the modified SQLite to demonstrate what is possible. Their purpose is to test certain types or aspects of queries. The content of the used databases tables is not provided here.

Bibliography

[1] Denis Hirn, Torsten Grust. PL/SQL Without the PL. *In Proc. SIGMOD*, 2020.

[2] Denis Hirn, Torsten Grust, Christian Duta. Compiling PL/SQL Away. *In Proc. CIDR*, 2020.

[3] K. Ramachandra, K. Park, K.V. Emani, A. Halverson, C. GalindoLegaria, and C. Cunningham. Froid: Optimization of Imperative Programs in a Relational Database. *Proc. VLDB, 11(4)*, 2018.

[4] The PostgreSQL Global Development Group. PostgreSQL 12 Documentation, 2020. URL https://www.postgresql.org/docs/12/static/index.html. (visited on 09/26/2020).

[5] The SQLite Development Group. SQLite Documentation, 2020. URL https://www.sqlite.org/docs.html. (visited on 09/26/2020).

[6] The SQLite Development Group. SQLite Source Code (Version 3.33.0), 2020. URL https://sqlite.org/src/dir?ci=trunk&name=src. (visited on 09/26/2020).

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Abschlussarbeit selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Aussagen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, als solche gekennzeichnet habe. Diese Arbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Ort, Datum Unterschrift