Snel: SQL Native Execution for LLVM

Marcelo Mottalli
Universidad de Buenos Aires

Gustavo Ajzenman
Grandata, San Francisco

Carlos Sarraute
Grandata Labs, Buenos Aires

Abstract

Snel is a relational database engine featuring Just-In-Time (JIT) compilation of queries and columnar data representation. Snel is designed for fast on-line analytics by leveraging the LLVM compiler infrastructure. It also has custom special methods like resolving histograms as extensions to the SQL language. "Snel" means "SQL Native Execution for LLVM".

Unlike traditional database engines, it does not provide a client-server interface. Instead, it exposes its interface as an extension to SQLite, for a simple interactive usage from command line and for embedding in applications. Since Snel tables are read-only, it does not provide features like transactions or updates. This allows queries to be very fast since they don't have the overhead of table locking or ensuring consistency.

At its core, Snel is simply a dynamic library that can be used by client applications. It has an SQLite extension for seamless integration with a traditional SQL environment and simple interactive usage from command line.

Contents

1 Introduction

1.1 A Bit of History

Snel is a relational database engine featuring Just-In-Time (JIT) compilation of queries and columnar data representation. It is designed for fast on-line analytics by leveraging the LLVM compiler infrastructure [1]. It also has custom special methods like resolving histograms as extensions to the SQL language.

What is the motivation for building Snel? What motivates the design decisions? Here is a brief outline of the history of Snel.

The R&D team of the company was working on a product that leverages anonymized datasets coming from mobile phone companies and banks to analyze multiple aspects of the human dynamics of users, such as their consumption patterns, and how these patterns correlate with their economic status [2, 3, 4]. The team generated inferences of users' income [5] and analyzed how income is tied to the location in the communication graph topology [6]. Call networks and calling behavior were also used to infer credit risk [7] and even lifestyles of users [8].

The information concerning the mobility of users was used to predict mobility patterns [9] and large social events [10]. This information can complement traditional urban planning tools [11, 12, 13]. Universal laws concerning the regularity of users' mobility were observed in [14]. Mobility was also shown to be correlated with data traffic usage, which in turn was used to characterize users [15, 16, 17]. The mobility information has also applications in epidemiology, providing a cost-effective tool to analyze the spatial spread of diseases such as the Chagas disease [18, 19]. Finally the mobility information is the input used to generate predictions of the users' location, using methods such as the ones described in [20, 21, 22].

The social graph constructed from the communications between mobile phone users is rich in information that was leveraged to make inferences such as dynamic communities [23], inferences of age, gender and other socio-demographic attributes [24, 25, 26, 27, 5]. In particular it provides an effective tool for predicting customer churn [28, 29, 30], along with a variety of other predictions [31, 32, 33, 34, 35].

At that time, the Data Processing Infrastructure (DPI) team was working on a front-end which displays the output of queries made on millions of records containing the results of the previous analyses, and making those queries on a regular MySQL database took minutes, whereas our objective was to give the result to the user in less than a second.

The DPI team came up with the following proposal: to build a tiny engine that would have all the data in memory (since the data could fit in the RAM of the server) and eliminate the I/O latency problems. To simplify things, they decided to:

1. Use the SQLite engine that has an API to incorporate external data sources.
2. Program everything in C, since this is the language of the SQLite API.

It should be noted that, at that time, the set of queries made by the original product were very limited, so the idea was to hard-code those queries in our engine to accelerate everything.

So the team started working on that. What took them more time was to understand the API of external (or "virtual") tables of SQLite. Basically the only thing it did was bring the rows one by one, but luckily it would not bring the whole row, it only brought the columns that were needed, so it was possible to make the engine columnar. This way the new engine would be a simple interface to a list of vectors (arrays) where each vector would be a column. Since the SQLite API calls this "virtual tables", the name of the engine would be "virtual table of vectors". We decided to call it VTor, to shorten it.

1.2   Our Own SQLite Engine

When we finished implementing VTor, there was a BIG problem: It was slow. Very slow.

The problem was that the only thing that the SQLite API did was bring the rows one at a time, and the SQLite queries engine was in charge of resolving the queries. This is the classical iterator model for query evaluation which was proposed back in 1974 [36], and was made popular by the Volcano system [37]. It is still today the most commonly used execution strategy, as it is flexible and quite simple. As long as query processing was dominated by disk I/O the iterator model worked fine. The problem is that, for example, if we have a table with 3 million records, the SQLite API is going to make 3 million calls to the xNext function, which is totally useless. We did the calculations and discovered that doing a scan of 1 million rows took approximately 1 second, just to advance the cursor. Without even taking into account the extra cost of processing the query.

As often happens, despair was the mother of inspiration. We got fully into the SQLite code and tried to understand how the queries resolution mechanism worked. It turns out that SQLite has a kind of "virtual machine" JVM style called VDBE (Virtual Database Engine), and each query is represented as a series of opcodes that are then interpreted by SQLite to solve the query.

Then we came up with the idea: we could add new instructions to the VDBE that would allow to yield the query execution control, effectively bypassing the SQLite mechanism. In this way we could really hardcode the queries and execute them all within our own engine, while maintaining some compatibility with SQLite. This way we would not need for example to make our own SQL parser since we could "hook" the SQLite parser.

So we added new opcodes to VDBE. We called these opcodes "custom queries", since they allowed the external engine to execute the query in a customized way. We also extended the SQLite API to allow the external engine to provide more information to the SQLite query planner. The functions that we added were:

xCanOptimizeSelect: SQLite passes the tokenized tree of the SQL query to the module and the module responds with a true or false if that particular query can be executed entirely by the engine. If so, a new VDBE opcode named OP_VTRunSelect is generated that calls a function of the API.

xExecuteCustomSelect: It re-passes the tokenized tree to the module but this time it tells it to execute that query. The module returns a cursor that will return a new row each time xNext is called. The difference between this and the "traditional" mechanism of the SQLite API is that the module can do anything with the query that is passed

4

to it. What VTor does is to analyze the query and in each call to xNext it makes the aggregation of all the data using OpenMP and other optimizations.

xExplainCustomSelect: It simply asks the module to describe what it will do with the query. It returns a string with a description of the "query plan" that the module will execute internally. VTor does not return anything useful, but Snel does, as we will see later.

An additional change we did to make everything run faster was to add indexes. The indices are very simple: they are simply a vector of tuples (value, rowid) ordered by ascending value, one for each column that you want to index. In this way, each time a query is made with a constraint of type WHERE x > c, a binary search is made in the index to reduce the search space.

From here things were looking nice. There was only a major inconvenient, and that was that ALL the information should reside in memory. Each time the product started, it had to load the data of a CSV file into the memory and generate the indexes. This did not take too long but it was annoying.

Shortly before the first release of the product all this collapsed. Basically because for the release, information was generated for ALL the social graph of a country, which included more than 100 million records, so the information no longer fitted into memory.

Again pressing the panic button, the solution arose: have the data saved on disk. The data was saved in "raw" mode on disk, one file per column, so that you could make mmap of each file to load the data of each column, with the additional advantage that the operating system is responsible for caching everything automatically .

And this is roughly how VTor was born.

But as it usually happens, the requirements started to grow, and to grow:

- We need to make histograms.
- We need to filter the outliers of the histograms, so we need support to calculate percentiles.
- We need support for String columns, even though we had sworn at first that we were not going to need.
- We need a COUNT (DISTINCT).
- We need to be able to join tables!
- Can we optimize such and such graph?
- Why does not VTor support the OR comparison?
- Can we add NULLs?

So what we predicted to be a few hundred lines of code began to grow and grow. Remember that at this point all the queries were hard-coded, so any new feature that we wanted to include was a lot of work since the architecture was not intended for anything more than a very punctual pair of use cases. Every time we had to add something, we suffered.

1.3 The Birth of Snel

In addition to the rigidity of VTor, there were two things that were very bothersome:

1. It was done in C. Segmentation faults were occurring every day.
2. There were many macros everywhere to handle different types of data in the same way. For example, it is not the same to solve something that executes SUM (column_int32) than something which executes SUM (column_int64). This not only causes problems when programming because you have to duplicate a lot of code, but it also had an impact on execution time, since for each value that is generated you have to call the function with the corresponding data type. Basically the same as a virtual table in C++ (not to be confused with the "Virtual Tables" of SQLite that are a completely different thing).

It was at this point that we learned how a database engine really works. It turns out that what you learn in college on relational algebra and those kind of things are not an invention and can be used in real life!

We studied the paper "Efficiently Compiling Efficient Query Plans for Modern Hardware" [38]. The main problem, they say, is the dispatch of methods in run time. The same problem we mentioned earlier with the data types! This causes thousands or millions of virtual function calls in the execution of a query plan, which breaks the execution pipeline of the CPU.

The solution proposed by this paper was to compile the queries to machine code using LLVM [1], which solves the problem of virtual functions by directly generating the necessary code for each data type. As an experiment, we started to implement the ideas of this paper. The main ideas of [38] are:

1. Processing is data centric and not operator centric. Data is processed such that we can keep it in CPU registers as long as possible. Operator boundaries are blurred to achieve this goal.
2. Data is not pulled by operators but pushed towards the operators. This results in much better code and data locality.
3. Queries are compiled into native machine code using the optimizing LLVM compiler framework.

An additional advantage was that using relational algebra to solve queries gave a lot more flexibility to solve queries than VTor did. So, we went on developing and developing.

It took one year but in the end we were able to finish it. We called this new engine "SNEL" as an acronym for "SQL Native Execution for LLVM". Well, that's the excuse, actually we chose Snel because that means "fast" in Dutch and the name seemed right to us.

When developing Snel we made the following decisions:

1. We were going to continue using the mechanism of SQLite virtual tables, which was already implemented.
2. We were going to use the same VTor tables. So only the switch of the virtual table module from VTor to Snel could be made and the results had to be the same, with an improvement in performance. Therefore, we could continue using the mechanism of mmaping the files with the data of the columns.
3. We did it in C++, since we were hating C and also the LLVM API is in C++. While C++ brought a few problems, the number of segmentation faults and memory leaks was greatly reduced. Now at least we had exceptions!

2 The Architecture of Snel

Broadly speaking, Snel has 3 important parts:

1. The storage system.
2. The SQL translator to relational algebra and optimizer.
3. The translator from relational algebra to IR. The Intermediate Representation (IR) is a low-level programming language similar to assembly, and can be considered as the "bytecode" of LLVM.

2.1 Storage System

Snel stores the tables in a very simple way: each column is stored in a single file, in "flat" format. That is, if for example we have an int32 column, the file will have 4 bytes for each value. This allows you to map the file to memory and use it as if it were an array directly, without any intermediate conversion. The big problem with this is that it wastes a lot of space, since the vast majority of data are NULLs or 0 and could easily be compressed. Each column file has a .snelcol extension.

The indexes are stored in a similar way to the columns, except that each value of the index is a tuple (value, rowid). The rowid is 64 bits, so each index entry is (sizeof (value) + 8) bytes. This also wastes a lot of space. Each index file has a .snelidx extension.

In addition to the column and index files, there is a file with extension .snel that is nothing more than a JSON with the description of the schema of the table: what type of data each column has, if they are indexed or not, if NULLs are allowed, etc.

Text Columns

What we said earlier makes sense for columns with fixed-length data types: integers and floating point. However, the text columns have a special format.

Each .snelcol file that stores a column of text has the following format:

```
String 1\0
String 2\0
...
String n\0
Sync bytes ('SB')
Offset to String 1 (−1 if NULL)
Offset to String 2 (−1 if NULL)
...
Offset to String n (−1 if NULL)
Offset to the offset of String 1 (yo dawg)
```

At first glance, this last value does not seem to make sense. However, it is necessary to know where the list of offsets begins within the file quickly. The indices of the text columns do

not include the value, they are only a list of rowids ordered according to the string value from smallest to greatest. That is, it is not a COVERING INDEX.

2.2 Translator and Optimizer

True to its VTor roots, Snel translates a SQL query provided by SQLite to a relational algebra tree. The SQL query comes as a tree of tokens and the module snel_sqlite (which is the interface between SQLite and Snel) translates that into a structure of queries understood by Snel.

This structure is quite simple. Viewing the src/query.hpp file you can see what SQL features Snel supports:

```
boost::ptr_vector<SQLExpression> fields;
ConstraintTree constraintTree;
std::set<const Table∗> sourceTables;
UniqueExpressionList groupBy;
std::vector<OrderClause> orderBy;
uint64_t limit = NO_LIMIT;
uint64_t offset = 0;
bool distinct = false;
```

The structure is:

fields: corresponds to the SELECT clause of SQL. It contains the list of expressions that are going to be solved (ex: col1, SUM (col2), COUNT (*), etc.)

constraintTree: corresponds to the WHERE clause of SQL. Example: (col1> 10 AND (col2 <20 OR col3 = 5)). It is represented by a tree where each node is a Boolean operator.

sourceTables: corresponds to the FROM clause of SQL, that is, the list of tables that will be queried.

groupBy: expressions by which GROUP BY is going to be done.

orderBy: as in SQL.

limit: as in SQL.

offset: as in SQL.

distinct: flag that indicates whether the query is DISTINCT or not, that is, whether it will return duplicate values or not.

This is basically the subset of SQL that Snel understands. Note that it does not support subqueries, at least not directly, although it would not be terribly difficult to add them.

Snel's translator will grab this structure and turn it into a tree of relational algebra operators. Without going into too much detail (that comes later), Snel converts the following query:

```
SELECT foo, COUNT(∗) FROM table1 WHERE bar > 2 GROUP BY foo
        ORDER BY foo DESC LIMIT 10 OFFSET 2
```

The optimizer will grab this query plan and apply several optimization steps to make the query run faster, such as verifying whether the table has indexes to hit only a subset of the rows.

2.3 Translator to IR

Once the optimized query plan is generated, it is translated into IR code, which in turn is compiled into machine code by LLVM and then the query is executed. I will explain this in more detail later.

2.4 The Snel Code and Class Hierarchy

The Snel code is divided into three parts:

1. The main library, libsnel.so. It is the dynamic library to which the other two parts link. The code is inside the src/ directory.
2. The command line. It allows to do several tasks related to the maintenance of tables (creation / import, generation of indexes, list contents, etc). Located in the directory cli/.
3. The interface with SQLite. As mentioned before, SQLite allows you to incorporate external engines, and this module allows queries to Snel tables from SQLite directly. The code is in the snel_sqlite/ directory.

Apart from these three main parts, we can also find:

1. Testing code in the tests/ directory.
2. A utility to generate Snel tables from Spark in the spark−snel−connector/ directory.
3. A script that allows you to add columns to Snel tables in the scripts/ directory.

We now describe the class hierarchy of the main library. The organization of the Snel code into classes corresponds to the architecture previously described and has three main parts: storage, relational algebra engine and IR translator.

The classes of the library are described in the following sections.

2.4.1 Storage

BaseVector: Represents a vector of fixed-size elements. It is similar to a std::vector<> but with the difference that the size of the elements is defined in runtime. This allows us to generate vectors from the JIT code. A vector has an associated allocator (see below), which allows the vector to reside in memory or to be saved to disk in a transparent form.

Allocator: It is the interface for vector allocators. There are two types of allocators.

MemoryAllocator: For vectors that are going to be generated in RAM.

MMapAllocator: For vectors that have a file in the filesystem as backup.

BaseColumn: It is an interface that represents a column of a table. An interface is used since afterwards there will be a subclass for each type of column (int [8, 16, 32, 64], float and string).

Column<T> and StringColumn: specific subclasses for each type of column.

BaseIndex: Interface that represents an index associated with a column. Note that this class descends from BaseVector, since all the indexes are vectors with elements of fixed size.

Table: Table with its columns.

2.4.2 SQL / Relational Algebra

Query: Structure that represents a SQL query.

QueryPlan: Represents the relational algebra tree of a query plan and contains the code to generates the query plan from a Query instance.

QueryPlanOptimizer: Takes a QueryPlan, applies several optimization steps, and returns another QueryPlan, theoretically more optimal. This class is VERY complex. As I read once: "Query optimization is not rocket science. When you flunk out of query optimization, we make you go build rockets."

QueryPlanParallelizer: This class is another step of optimization, but I put it in a separate file because it is a bit more complicated than the others. It analyzes the tree of a query plan and decides which parts can be parallelized to execute them in several threads.

*-Expression: All classes that end in -Expression that are in the subdirectory expressions/ represent SQL expressions. For example, the function SUM(column1) is a SUM expression composed with a column expression.

2.4.3 IR Translator

BaseBuilder: It is a subclass of IRBuilder of LLVM that has some useful extensions.

QueryPlanBuilder: IR Builder that has some specific utilities to generate IR code from query plans.

LLVMWrapper: Utilities that allow to call C++ functions from JIT code. For now it is only a wrapper for the methods of the BaseVector class.

QueryContext: A JIT execution context is simply a set of variables that are visible by an execution thread. When a query is divided into several threads, each thread will have its own execution context (all threads will have the same variables but with different values).

*-Operator: The classes that end in -Operator that are in the operators/ subdirectory represent the different relational algebra operators supported by Snel. Later each operator is explained in detail.

3 Implementation of Snel

Here we will explain in as much detail as possible how Snel is implemented. If you have to get your hands dirty and modify something, this is a good place to begin to understand how everything works and why things are made the way they are made.

3.1 First of All: How to Compile

First of all, it is necessary to install LLVM. The latest supported version is 3.6.2, since the API from 3.7 is not backwards compatible. At some point you would have to change the code to update it to the new API.

Since Ubuntu as of 16.04 comes with LLVM 3.8 installed by default, we will have to compile and install LLVM in a separate directory. The steps to download, compile and install LLVM in /opt/llvm−3.6.2 are:

```
$ wget http://llvm.org/releases/3.6.2/llvm−3.6.2.src.tar.xz
$ tar xf llvm−3.6.2.src.tar.xz && cd llvm−3.6.2.src
$ mkdir build && cd build
$ ../configure −−enable−jit −−enable−optimized −−enable−targets=host
        −−enable−assertions −−prefix=/opt/llvm−3.6.2
$ make −j3 && sudo make install
```

This takes a long time, so go get a coffee. Once LLVM is installed, you have to download the dependencies:

```
$ sudo apt−get install cmake g++ libgomp1 libboost−all−dev libreadline−dev
        zlib1g−dev mawk liblog4cxx−dev libyaml−cpp−dev
```

Now you have to compile the famous sqlite3-grandata. Assuming that the repository checkout is in ~/src/dpi:

```
$ cd ~/src/dpi/sqlite3−grandata
$ mkdir −p build/release && cd build/release
$ cmake ../.. −DCMAKE_BUILD_TYPE=Release
$ make −j3 && sudo make install
$ # Verify that everything works fine
$ sqlite3−grandata −−version
3.7.17 grandata
```

Now we proceed to compile Snel. We're going to do it in Debug mode to be able to... well, debug:

```
$ cd ~/src/dpi/projects/snel
$ mkdir −p build/debug && cd build/debug
$ cmake ../.. −DCMAKE_BUILD_TYPE=Debug
        −DLLVM_DIR=/opt/llvm−3.6.2/share/llvm/cmake
```

3.2 Testing

To check that everything is working well, we can run the tests. They are written using the Google testing framework and can be run together using CTest, a test running tool that comes with CMake:

```
$ cd $PATH_TO_SNEL/build/debug/test
$ ctest
Test project /home/marcelo/dpi/projects/snel/build/debug/tests
      Start 1: test_basics
1/6 Test #1: test_basics ...................... Passed 0.09 sec
      Start 2: test_single_table
2/6 Test #2: test_single_table ................ Passed 0.37 sec
      Start 3: test_null
3/6 Test #3: test_null ........................ Passed 0.29 sec
      Start 4: regression_tests
4/6 Test #4: regression_tests ................. Passed 1.22 sec
      Start 5: test_config
5/6 Test #5: test_config ...................... Passed 39.21 sec
      Start 6: test_sql
6/6 Test #6: test_sql ......................... Passed 5.28 sec

100% tests passed, 0 tests failed out of 6

Total Test time (real) = 46.48 sec
```

3.3 Creating Snel Tables

At the moment Snel tables can not be created using the SQL interface but must be created "from the outside". To create the tables you must use the command snel import, which will import data in plain text from some source to a new Snel table. Let's see:

```
$ snel import −−help
Usage: snel import [options] <table name> <schema file>
         <output directory> [input file]
Options:
    −h This help message
    −s <separator> Field separator character (default: "|")
    −b <size> Buffer size, in number of rows (default: 100,000)
    −−safe Check for invalid input (SLOWER)
    −−null−repr Representation for null fields
    −v Be verbose
```

We then need two things: a data source (usually a CSV file) and a file to describe the schema of the table to be created. The schema file has several lines with the following format:

```
<column name 1> <tipo> [NULLABLE] [INDEXED],
<column name 2> <tipo> [NULLABLE] [INDEXED],
```

```
...
<column name n> <tipo> [NULLABLE] [INDEXED]
```

Where the type of the column can be one of the following:

- BOOLEAN, BOOL or BIT: value 0 or 1
- INT8 or CHAR: 8-bit integer
- INT16 or SHORT: 16 bit integer
- INT32 or INT: 32-bit integer
- INT64 or LONG: 64-bit integer
- FLOAT: 32-bit floating point (Note: doubles are not supported)
- STRING or TEXT: variable length text

By default the columns DO NOT accept NULLs, unless the NULLABLE flag is specified. In the same way, the columns are not indexed unless the flag INDEXED is specified. By convention, the table schema specification files are saved in a file with the .snelschema extension. Going back to the snel import command, the most common use case is to grab a file with extension .csv.gz and to pipe it to that command. Something like this:

```
$ # If you do not have the pv command installed, I recommend it STRONGLY.
$ # You will thank me.
$ # In this case we use the tail −n +2 to remove the headers from the csv file
$ pv archivo.csv.gz | zcat | tail −n +2 |
        snel import table_name schema.snelschema /tmp/sarasa
```

If all goes well, you will see in /tmp/sarasa many files with the name

```
table_name−<column_name>.snel[col|idx]
```

and a file named table_name.snel which is the JSON that contains the data and metadata of the table. These files contain the raw data of the columns. So much so that if you read the files to an array of C and you cast it to the correct data type, you will be able to read all the values of the corresponding column (except for the columns of type text, whose format I already explained before).

3.3.1 Update values or add columns to a Snel table

In its origins, both VTor and Snel were read-only. But since the format is so simple nothing prevents us from updating the values directly or adding new columns to an existing table. The snel merge command allows you to "merge" two tables using a column as a merge key. Let's see a typical example: we have the column clients where the primary key is the MDN column, and we want to add a new variable to the table. Suppose that this variable is called eye_color and is a number between 0 and 3. Suppose we also want to update the height variable that already existed in the customer table but we did a super algorithm that inferred this variable with an accuracy of +/- 1 cm in comparison with the existing variable that has a precision of +/- 50 cm (yes, a crap). Usually these two variables come in a CSV that looks like this:

```
mdn|eye_color|height
B0A00F35B1F5DEAD84DB2D28BD883094|0|1.77
9386711E612687A86DF876B7D76FB514|2|1.52
3328EFF2A02D215A16F52F46A83CAE1B|3|1.85
```

And suppose we only have information for these three MDNs. The problem is that we can not create a table and put the new column "by hand" by copying the .snelcol file because nothing would indicate that the MDNs are in the same order. The solution is then to create a new table with this information using the snel import command:

```
$ cat new_variables.csv | tail −n +2 | snel import new_variables
        new_variables.snelschema /tmp/new_variables
```

Where the file new_variables.snelschema is:

```
mdn STRING INDEXED,
eye_color INT8 NULLABLE,
height FLOAT NULLABLE
```

This will create /tmp/new_variables/new_variables.snel and all the associated column files. Now let's see the snel merge command:

```
$ snel merge −−help
Usage: snel merge [options] <source table> <dest table> <key column name>
Options:
    −h This help message
```

This will merge the source table new_variables in the target table (clients) using the key column (mdn). The command for the merge is then:

```
$ snel merge /tmp/new_variables/new_variables.snel /path/to/clients.snel mdn
```

This will do the following:

1. It will create the column eye_color in the table clients and will set the values in the corresponding rows. The rest of the rows will be NULL.
2. It will update the corresponding values of the height column.

In order to simplify all these steps, there is a script in scripts/snel−merge−csv.py that automates all this. In this case it would be executed in the following way:

```
$ cat new_variables.csv | tail −n +2 |
        scripts/snel−merge−csv.py −c eye_color:int8 −c height:float
        /path/to/clients.snel mdn
```

This will create a temporary table in /tmp (make sure there is enough space!) And it will do the merge automatically.

Note that you can't append data to a Snel table. There is nothing to prevent it, it's just that that functionality is not implemented.

3.3.2 Generating tables with Spark

In the Snel repository there is a connector for Spark that allows generating Snel tables using Spark. This involves two steps:

1. Generation of partitioned tables in HDFS (Hadoop Distributed File System).
2. Merging the partitions from HDFS to where the table will be used.

So, how is a Snel table generated from Spark? There is an implicit function called saveToSnel that applies to Resilient Distributed Datasets (RDD) of type RDD[Seq[Option[Any]]]. The function looks like this:

```
def saveToSnel(destPath: String, snelSchema: SnelSchema,
        compress: Boolean=true): Unit
```

– destPath: Path where the table will be saved. For example: hdfs://host_hdfs.com:1234/user/pepe/tabla_snel
– snelSchema: The schema of the table. It is basically a table name and an array with the specification of each column. The specification of each column is an instance of one of the following classes:
  – BoolColumn
  – Int8Column
  – Int16Column
  – Int32Column
  – Int64Column
  – FloatColumn
  – TextColumn
– compress: Indicates whether the data will be compressed on HDFS. Unless there is a really strong motive, you usually have to leave this parameter as true.

Now, we must bear in mind that this is very nice when it comes to making type conversions. Each value within the RDD is a Seq that represents the values of the row, and there must be as many values as the number of columns. Each value can be NULL or non-NULL, that is why an Option is used. If the value is non-null, the value is cast to the correct data type, for example if the schema says that this value corresponds to an int16, it will do value.asInstanceOf[Short], and if this fails it will throw an exception.

So, in order to make the conversion, given a generic RDD, it is necessary to do the following steps:

1. Generate the Snel schema, doing something like this:

```
val snelSchema = SnelSchema(tableName = "una_tabla",
    columns = Array(
    Int16Column("una_columna_int16", isNullable=true, isIndexable=false),
    FloatColumn("una_columna_float"),
    ... etc ...
))
```

2. Convert all RDD values to an Option[Any] with the correct data type. This is the most expensive step. The mapping between types of columns and values is as follows:

Type of column	Type of Scala
BoolColumn	Boolean
Int8Column	Byte
Int16Column	Short
Int32Column	Int
Int64Column	Long
FloatColumn	Float
TextColumn	String

To do this, you can obviously map the RDD so that it has the desired format.

3. Once the RDD is generated with the desired format, call rdd.saveToSnel (...) with the appropriate parameters. Note that for this function to be available, we must first do import com.grandata.snel.spark.connector._ so that the Scala compiler puts the function as implicit.

This will trigger a Spark task that will generate the table in HDFS. When viewing the list of generated files, you can see the following:

```
bash−4.1# bin/hdfs dfs −ls /user/marcelo/test_table
Found 5 items
−rw−r−−r−− 3 [...] /user/.../test_table/_SUCCESS
drwxr−xr−x − [...] /user/.../test_table/test_table−floatcol.snelcol
drwxr−xr−x − [...] /user/.../test_table/test_table−int32col.snelcol
drwxr−xr−x − [...] /user/.../test_table/test_table−textcol.snelcol
−rw−r−−r−− 3 [...] /user/.../test_table/test_table.snel

bash−4.1# bin/hdfs dfs −ls /user/marcelo/test_table/test_table−floatcol.snelcol
Found 3 items
−rw−r−−r−− 3 [...] /user/.../test_table/test_table−floatcol.snelcol/part−r−00000.gz
−rw−r−−r−− 3 [...] /user/.../test_table/test_table−floatcol.snelcol/part−r−00001.gz
−rw−r−−r−− 3 [...] /user/.../test_table/test_table−floatcol.snelcol/part−r−00002.gz
```

As you can see, it creates a directory for each column with the parts of the columns inside. Snel can not access this information directly, so you have to run another Spark job to download these files and assemble them in the place where they will be used. For this, you have to call the function com.grandata.snel.HDFSMerge.mergeTable().

I strongly recommend seeing the tests in this package to see how everything works, mainly the test of the HDFSMergeTest class, which is an integral test.

3.4 Executing Queries

For now, the only way to execute SQL queries on one or more Snel tables is by using the SQLite interface, which translates SQL queries into the structure that represents a query in Snel.

The SQLite interface for Snel is a dynamic library libsnel−sqlite.so that should load automatically when loading the binary sqlite3−grandata. If this does not happen, you have to specify the path by hand with the command .load. For example, when Snel is compiled in debug mode, I do not want to install it in /usr/local so I load it directly from where it is generated:

```
$ sqlite3−grandata
SQLite version 3.7.17 grandata
VTor version vtor−1.6.7−release
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .load /home/.../snel/build/debug/snel_sqlite/libsnel_sqlited.so
Initializing Snel 0.9.1−debug (git: develop/8906b0f)...
[DEBUG] [0x7f53ad3ad7c0] ... snel − Initializing LLVM native target
[DEBUG] [0x7f53ad3ad7c0] ... snel − Finished initializing
[DEBUG] [0x7f53ad3ad7c0] ... snel − Setting locale to en_US.UTF−8
sqlite>
```

Once the module is loaded in SQLite, we must indicate that we want to create an external or virtual table using the Snel module:

```
sqlite> CREATE VIRTUAL TABLE tabla USING SNEL('/path/a/tabla.snel');
sqlite> SELECT COUNT(∗) FROM tabla;
123456
```

Here all the magic happened: the query was translated into relational algebra, the resulting tree was translated into IR code, that code was compiled into machine code and that code was executed.

Types of queries supported

Snel supports the following SQL functionality:

- SELECT of columns and aggregated columns.
- Aggregation functions:
  SUM, AVG, MAX / MIN, COUNT, COUNT (DISTINCT).
- Aggregation functions for histograms:
  BIN, BIN_MIN, BIN_MAX.
  These are unique features of Snel.
- WHERE clauses:
  - <column> <op> <expr>: where <op> can be >, >=, <, <=, = or !=, and <expr> can be a constant or the name of another column.
  - Boolean operators: AND, OR, NOT.
  - Clause <column> IN (<subquery>) where <subquery> is a query supported by Snel. This is the only type of subqueries that Snel supports for now.
- GROUP BY.
- ORDER BY.

This is an example of a complex query supported by Snel:

```sql
SELECT BIN(latitude, 10) AS bin_lat, BIN(longitude, 10) AS bin_long,
    BIN_MIN(latitude), BIN_MAX(latitude), BIN_MIN(longitude),
    BIN_MAX(longitude), COUNT(*), SUM(col1), AVG(col2),
    COUNT(DISTINCT col3) AS cnt
FROM table
WHERE
    (latitude BETWEEN -10 AND 10
    AND longitude BETWEEN -20 AND 20
    AND age > 30)
    OR age IN (SELECT ages FROM otra_tabla)
GROUP BY bin_lat, bin_long
ORDER BY cnt DESC
LIMIT 10
```

4   The Life of a Query

Let's see what happens between writing a query and returning its result. Broadly speaking, the following steps are executed:

1. The SQLite parser converts the string with the SQL query to an AST tree (Abstract Syntax Tree).
2. SQLite "asks" the Snel module if it is able to resolve that query – performed by sqlite3_module.cpp, function snel_xCanOptimizeSelect. Snel analyzes the query and, if the query has a supported structure, returns true.
3. Let's assume that the query can be executed by Snel. In that case, call the snel_xExecuteCustomSelect function that gives control of the execution to Snel.
4. Snel converts the AST to a relational algebra tree, the query plan. This is an almost direct translation. After doing this, it applies several optimization steps to the query plan to have an optimized query plan (see the Query::executeQuery function).
5. The query plan is translated into LLVM IR code. See function QueryPlanBuilder::generateTopLevelQueryPlanIR.
6. The LLVM execution engine is created with the aggressive optimization parameter so that the queries are fast.
7. The IR of the query plan is compiled to machine code using the execution engine. The result of the compilation is saved in an object of type ExecutedQuery, which is the one that will be the owner of the LLVM module and will provide a cursor to iterate the resulting rows.
8. The control is returned to SQLite. SQLite will then call the snel_xNext function that will advance the cursor row by row and snel_xColumn that will return the values of the resulting rows.
9. When snel_xNext indicates EOF, SQLite calls snel_xClose, which frees all resources. We will see these steps in more detail, with references to the code to make it easier to understand.

4.1 SQL Translator

The first step is done by SQLite and is simply the translation of the SQL query to an AST. This is a standard compiler process. Next, SQLite invokes the snel_xCanOptimizeSelect function, passing the AST as a parameter. The AST is an instance of the *Select* type of SQLite, which looks like this:

```
struct Select {
  ExprList *pEList; /* The fields of the result */
  u8 op; /* One of: TK_UNION TK_ALL TK_INTERSECT TK_EXCEPT */
  u16 selFlags; /* Various SF_* values */
  int iLimit, iOffset; /* Memory registers holding LIMIT & OFFSET counters */
  int addrOpenEphm[3]; /* OP_OpenEphem opcodes related to this select */
  double nSelectRow; /* Estimated number of result rows */
  SrcList *pSrc; /* The FROM clause */
  Expr *pWhere; /* The WHERE clause */
  ExprList *pGroupBy; /* The GROUP BY clause */
  Expr *pHaving; /* The HAVING clause */
  ExprList *pOrderBy; /* The ORDER BY clause */
  Select *pPrior; /* Prior select in a compound select statement */
  Select *pNext; /* Next select to the left in a compound */
  Select *pRightmost; /* Right−most select in a compound select statement */
  Expr *pLimit; /* LIMIT expression. NULL means not used. */
  Expr *pOffset; /* OFFSET expression. NULL means not used. */
};
```

As we can see, it is not a science. Each SQLite expression (Expr) contains several details about the types of SQL expressions, for example, if it is a constant, if it refers to a column, if it is an aggregation function, if it is a subquery, etc. Snel will try to translate this structure into a structure of the type snel::Query. This translation is done by traversing the AST recursively, section by section (SELECT fields, WHERE, GROUP BY, etc). At any time the translator may find an unsupported SQL feature such as subqueries, in that case the translation process ends with an exception. Even if the translation is successful, it may happen that the query for some reason is not supported by Snel. In this case, SQLite is informed that the query can not be processed internally by the engine, thereby giving control of the execution of the query to SQLite. If the query can be translated into a Snel query and is supported, the xCanOptimizeSelect function returns true. This tells SQLite that the execution of the query will be handled internally by Snel, which as explained before is a much faster mechanism.

4.2 Execution of the Query

If the query can be executed by Snel, SQLite will yield the query execution by calling the xExecuteCustomSelect function, where the AST of the query will be passed again and Snel will do the same translation as before. Once the translation of the AST is done, the function snel::Query::executeQuery is called, where ALL the magic happens: translation to relational algebra, optimization and compilation to machine code. The result of this whole

process is a cursor: basically an object that can be advanced with the function next() and from where the individual values of the columns can be brought.

4.3   Query to Relational Algebra

An object of the Query type is translated into a tree of relational algebra operators. The operators implemented in Snel are:

ACCUMULATE: It is an operator that simply stores the rows in memory for further processing by another operator. It is useful for example when we want to buffer the result of a sub-tree.

AGGREGATOR: It is the operator that is responsible for solving aggregation expressions such as SUM, COUNT, AVG, etc.

BINNER: It is a special operator of Snel. It is used to group values in different bins, generally used to calculate histograms of values.

CONSTRAINT: It is responsible for filtering the columns that do not satisfy the given constraints, for example "A = 1 OR (B> 2 AND c <3)".

DEBUG: Useful operator to debug, the only thing it does is to show the rows that pass through this operator by the standard output.

LIMIT: Limit the number of rows to a given value.

MERGEJOIN: Implements a sort-merge-join between tables. For now this is the only algorithm implemented, it would be good to implement a hash-join in the future.

PARALLEL_AGGREGATOR: Same as the aggregator but allows the aggregations to run in parallel using several CPUs, and is responsible for joining the results.

SORT: Sorts the rows that arrive according to some criteria (usually the value of a given column).

FULL SCAN: Full scan of a table. Run through all the columns sequentially.

INDEX SCAN: Scan a table through an index. The table is traversed non-sequentially.

XJOIN (or CROSS JOIN): Makes an "all against all" join, and is $O(n^2)$ (Cartesian product).

In a first step, the translation of the query to AR is quite direct. For example, a query like

SELECT a, COUNT ($*$) FROM table WHERE c <2 GROUP BY a

is translated simply as

FULL SCAN $->$ CONSTRAINT $->$ AGGREGATE.

The second step is to grab the AR tree and apply several optimization steps. The steps that are applied are (in order):

1. Optimization of JOINs: It evaluates whether the CROSS JOINS can be replaced by a MERGE JOIN. This occurs if there is a constraint of type table1.column1 = table2.column2 and both columns are indexed.

2. Relocation of CONSTRAINT: This step tries to "lower" the constraints that apply to a single table immediately after the SCAN associated with that table. This

serves two purposes: to quickly filter the rows that do not satisfy the constraints and to apply subsequent optimizations. In the literature this is known as predicate pushdown.

3. Indexing: It evaluates whether there is a FULL SCAN followed by a CONSTRAINT. If so, it checks whether an index can be applied, and if it can, it replaces the FULL SCAN with an INDEX SCAN (the CONSTRAINT is not replaced since it may have other constraints that do not apply to the index).

4. SORT Relocation: Try to pushdown the SORT operators. This step is necessary for the next one.

5. SORT Elimination: Check whether some SORT have become obsolete. The SORT is a costly operation and should be avoided as much as possible.

6. Parallelization: Finally, it checks whether the query can be parallelized. There are some conditions that a query must fulfill in order to be parallelized (basically, all the aggregation operators are parallelizable). If it is parallelizable, a new operator is added as the root of the query, the PARALLEL_AGGREGATOR.

Let's see a simple example query:

```
SELECT COUNT(∗) FROM table1 WHERE int8col1 > 3
```

Let's see how this query is translated into relational algebra. Opening an existing database, first disable multithread execution to simplify things, and let's see the translation:

```
sqlite> SELECT SNEL_SET_MAX_THREADS(1); −−Turns off multithreading
sqlite> SELECT COUNT(∗) FROM table1 WHERE int8col1 > 3;
48429032
sqlite> EXPLAIN QUERY PLAN SELECT COUNT(∗) FROM table1;
0|0|0|AGGREGATE (cost: 100000000) { CONSTRAINT [table1.int8col1 > 3]
        { FULL SCAN FOR TABLE 'table1' (100000000 rows) } }
```

The translation to AR is then

```
AGGREGATE (cost: 100000000) { CONSTRAINT [table1.int8col1 > 3]
        { FULL SCAN FOR TABLE 'table1' (100000000 rows) } }
```

This is read "from the inside out": first make a FULL SCAN of the table, filter by the specified column and then AGGREGATE the rows. The aggregation in this case is simply to count the number of rows that satisfy the constraint. In pseudocode:

```
num_rows = 0
while table1.nextRow(): # FULL SCAN
  if table1["int8col1"] > 3: # CONSTRAINT
      num_rows++ # AGGREGATE
```

When executing the previous query, the IR code will be stored in a file called queryplan.ll and the AR tree will be saved in the queryplan.dot file. Running xdot in this last file, we can generate a graphic representation of the query.

4.4 Translation to IR Code

Note that in order to understand this section, it is important to understand a little how LLVM works. I recommend following the Kaleidoscope tutorial to understand the API. Broadly speaking, we can say that the translation is "direct" in the sense that each relational algebra node is translated to IR independently of the others. Each AR tree is compiled to IR code inside a module that looks like this:

```llvm
; ModuleID = 'queryplan'

%TopLevel.ContextType = type { ... }
%TopLevel.ResultRow = type <{ ... }>

@contextInstance = global %TopLevel.ContextType zeroinitializer

@output.0 = global algun_tipo 0
@output.1 = global algun_tipo 0
...
@output.n = global algun_tipo 0

define private void @initializeContext_TopLevel.Context
        (%TopLevel.ContextType* %contextPtr) {
  ...
}

define void @queryPlanInit(%TopLevel.ContextType* %contextInstancePtr) {
  ...
}

define void @queryPlanDestroy(%TopLevel.ContextType* %contextPtr) {
  ...
}

define private void @staticInit() {
  ...
}

define private void @staticDestroy() {
  ...
}

define internal %TopLevel.ResultRow*
        @TopLevel.FetchRow(%TopLevel.ContextType* %contextPtr) {
  ...
}
```

This is the general structure of the module with the global variables and functions common to all the query plans. Let's see them one by one:

1. %TopLevel.ContextType: First let's talk about contexts. A context is a set of variables that will be visible to a particular part of the execution of a query plan. Thinking of it as an object-oriented programming model, a context would be the members of a class. In the execution of a query plan there are two contexts:
   - The global context, which contains the general variables of the query plan. There is a single global instance of this context.
   - The context of the thread, which contains the local variables of each execution thread. There are as many instances of this context as there are threads in the execution of a query plan.

   The variable % TopLevel.ContextType then represents the data type of the global context variables.

2. %TopLevel.ResultRow: The top-level in this case refers to the root of the execution tree. The rows returned by this node are going to be the rows of the result of the query. This structure represents the format of that row. For example, in a query like SELECT x, COUNT (x) FROM table, if we assume that x is of type int8, the TopLevel.ResultRow will have the type <int8, int64> representing x and COUNT respectively .

3. @contextInstance: It is the only instance of the global context, as explained above. This instance will be initialized when the query plan is initialized.

4. @output.[n]: These global variables are very important: they are the way to pass the rows "outside" of the LLVM module. Each of these variables will have the value of a column of the row that is being outputted. The reason for having a variable for each column is that LLVM has a function that, given a global variable in a module, returns a pointer to it in the module's execution space. In this way, we will be able to inspect the values from the outside.

5. @initializeContext_TopLevel.Context: This is the "constructor" of the global context. It creates the instance of the global context and initializes the variables.

6. @queryPlanInit: It is the constructor of the query plan. Among other things, it will call @initializeContext_TopLevel.Context. This function must be called from the outside to initialize the module.

7. @queryPlanDestroy: Destructor of the query plan. Called when you reach EOF and the query plan is not needed anymore.

8. @staticInit and @staticDestroy: These are functions called by the @queryPlanInit to set static variables (which are global outside of the context).

9. @TopLevel.FetchRow: This is the MOST important function. Each time it is called, it calculates the next row of the result of the query. Returns NULL when it reaches EOF.

All the modules will have this basic structure.

4.5   Compilation and Execution

Once the LLVM module is generated, the ExecutionEngine is created and the query is compiled. All this happens in a line of code:

```
std::unique_ptr <ResultCursor> cursor =
```

```
queryPlanIR.compile (engine, compileIRBenchmark);
```

The cursor will be the object that allows us to traverse the rows one by one. The module interacts with "the outside" (that is, everything that is not compiled by LLVM) through its public functions, which in this case are three:

1. The initialization function.
2. The FetchRow function.
3. The termination function.

In addition, there is a global variable for each output column, and these variables can also be accessed from the outside. In pseudocode, this is how a query is executed:

```
cursor = queryPlanIR.compile(engine, benchmark)
// The init function is called automatically
while cursor.next():
  row = cursor.values()
  // Process row[0], row[1], etc
cursor.delete()
```

4.6 Translation of Operators to IR

To translate an operator from AR to IR, each operator has a Builder associated with it. For example, the LIMIT operator has a LimitBuilder associated with it, which is responsible for the translation. Let's see the example of this particular operator. I use this operator because it is the simplest. We can think of the translation of an operator as a piece of code that does something with its input and generates an output. Let's take as example the query SELECT a, b FROM table LIMIT 10.

In the case of the LIMIT operator, it receives as input the values of columns a and b, and has the same values as output (that is, it does not make any transformation). The two SCAN operators, FULL SCAN and INDEX SCAN, are special since they do not have any input. If we think of it as a pseudocode, we can implement the two operators independently as follows:

```
// SCAN
class SCAN(tabla):
  def nextRow():
    tabla.next()
    if tabla.inEOF():
      yield EOF
    else:
      yield tabla['a'], tabla['b']
// LIMIT
class LIMIT(childOperator):
  count = 0

  def nextRow():
```

```
    while count < 10:
       (a, b) = childOperator.nextRow()
       count += 1
       yield a, b
    yield EOF
```

The interesting part of all this is that calling the function nextRow(), make yield, etc., has 2 major problems:

1. It is not very friendly with the instruction cache of the CPU, since it has to make calls to virtual functions.
2. It needs memory to store the intermediate values.

This makes the execution of the query suboptimal, since there is a lot of overhead. We will have to face this problem. Like the fantastic twins joining their rings, these two operators can also be joined in a super operator that we will call Query Plan:

```
class QUERY_PLAN(tabla):
  count = 0

  def nextRow():
    while count < 10:
      tabla.next()
      if tabla.inEOF():
        yield EOF
      else:
        a, b = tabla['a'], tabla['b']

        count += 1
        yield a, b

    yield EOF
```

Note that the values of the columns flow "from bottom to top" (seeing the query plan as a tree), that is, the values are generated in the leaves and go to the root. This is the main idea of the paper "Efficiently compiling efficient query plans" [38]. The traditional approach is the opposite, with the problems that we described earlier.

Now let's look directly at the code that translates the LIMIT operator to IR, which is the simplest of all. I put the code here and then I will be commenting part by part. The following code is in the function LimitOperatorBuilder::_ generateOperatorFunction, and I put it with color so it looks nicer:

```
void LimitOperatorBuilder::_generateOperatorFunction(
        QueryPlanBuilder& builder,
        QueryContext& queryContext,
        llvm::Value∗ contextInstance,
        bool /∗inlined∗/)
{
    const LimitOperator∗ limitOperator =
```

```
        dynamic_cast<const LimitOperator*>(this->_operator);
    _zero = builder.getConstant<LimitOperator::limit_type>(0);
    _counterVariable = queryContext.registerContextVariable(
        this->prefix() + ".currentPosition",
        builder.getType<LimitOperator::limit_type>(),
        _zero);

    llvm::Constant* limit = builder.getConstant<LimitOperator::limit_type>(limitOperator->limit);
    llvm::BasicBlock* produceChildBlock = builder.CreateBasicBlock("produceValues");
    llvm::BasicBlock* returnRowBlock = builder.CreateBasicBlock("returnRow");
    llvm::BasicBlock* reachedEOFBlock = builder.CreateBasicBlock("reachedEOF");
    // Operator begin
    // If we have outputted enough rows, finish.
    llvm::Value* counterVariablePtr = queryContext.getVariablePtr(contextInstance,
            _counterVariable, builder);
    llvm::Value* currentCounter = builder.CreateLoad(counterVariablePtr, "currentCounter");
    llvm::Value* reachedLimit = builder.CreateGenericCmp(QueryPlanBuilder::CMP_GE,
            currentCounter, limit, "reachedLimit");
    builder.CreateCondBr(reachedLimit, reachedEOFBlock, produceChildBlock);

    builder.SetInsertPoint(produceChildBlock);
    {
        // Create the function to produce the child values
        OperatorBuilder::OperatorFunction childProduceFunction = this->childBuilder->
                buildInlinedOperatorFunction(builder, queryContext, contextInstance);
        OperatorBuilder::OperatorFunction::OperatorFunctionResult childResult =
                childProduceFunction.invokeOperator(builder);
        // Increment the counter
        llvm::Value* next = builder.CreateAdd(currentCounter,
                builder.getConstant<LimitOperator::limit_type>(1));
        builder.CreateStore(next, counterVariablePtr);
        // Generate the requested expressions.
        // This operator is not able to generate expressions on its own, so it
        // just copies the expressions from the child node.
        for (const SQLExpression& expression : this->_operator->expressions)
            this->setOutputValue(builder, expression, childResult.expressionResolver.
                    getResolvedExpression(expression, builder));
        builder.CreateCondBr(childResult.inEOF, reachedEOFBlock, returnRowBlock);
    }
    builder.SetInsertPoint(returnRowBlock);
    {
        this->returnRow(builder);
    }
    builder.SetInsertPoint(reachedEOFBlock);
    {
        this->returnEOF(builder);
    }
}
```

As you will see it is not terribly complicated. Although this is the easiest operator, by far!
Let's see what is happening, and take the opportunity to introduce the concepts.

```
    const LimitOperator* limitOperator =
```

26

```
        dynamic_cast<const LimitOperator*>(this->_operator);
    // Nothing crazy here. You only get a pointer to the operator itself.
    _zero = builder.getConstant<LimitOperator::limit_type>(0);
    _counterVariable = queryContext.registerContextVariable(
        this->prefix() + ".currentPosition",
        builder.getType<LimitOperator::limit_type>(),
        _zero);
```

Here begins the interesting part. There is a reference to context variables. To understand
it, you have to keep in mind that the same portion of the code can be executed in several
processors in parallel. The context variables are then the variables of each execution thread.
Each operator will have a pointer available to its context, which in turn will have a set
of variables specific for each operator. In this case, the LIMIT operator is saying "I need
the context to store a variable of type LimitOperator::limit_type (alias for uint64_t), and
call it LimitOperator.CurrentPosition. The variable must be initialized with the value 0".
According to the pseudocode that we showed before, this is neither more nor less than the
line that says "count = 0", with the addition that this is an instance variable of the class, and
not a local variable.

```
    llvm::Constant* limit = builder.getConstant<LimitOperator::limit_type>(limitOperator->limit);
    // Here the value for which you are limiting yourself is loaded in a constant (in this case, 10).
    llvm::BasicBlock* produceChildBlock = builder.CreateBasicBlock("produceValues");
    llvm::BasicBlock* returnRowBlock = builder.CreateBasicBlock("returnRow");
    llvm::BasicBlock* reachedEOFBlock = builder.CreateBasicBlock("reachedEOF");
```

Here the BasicBlock that the operator will use are created. If you do not know what a
BasicBlock is, go right now and read the LLVM tutorial.

```
    llvm::Value* counterVariablePtr = queryContext.getVariablePtr(contextInstance,
            _counterVariable, builder);
    llvm::Value* currentCounter = builder.CreateLoad(counterVariablePtr, "currentCounter");
    llvm::Value* reachedLimit = builder.CreateGenericCmp(QueryPlanBuilder::CMP_GE,
            currentCounter, limit, "reachedLimit");
    builder.CreateCondBr(reachedLimit, reachedEOFBlock, produceChildBlock);
```

This reads the value of the environment variable counter (remember: in LLVM, a variable
and its value are two different things! A variable is simply a pointer to a memory address),
and compares it with the value of the limit. If it is greater or equal, it jumps to the block
that returns EOF, otherwise it jumps to the block that will produce the values of the row:

```
if counter >= limit:
    goto eof
else:
    goto produceChild

    builder.SetInsertPoint(produceChildBlock);
    {
```

Here we inject the code that will ask the child operator (in this case the FULL SCAN) to
produce rows.

```
// Create the function to produce the child values
OperatorBuilder::OperatorFunction childProduceFunction = this−>childBuilder−>
                buildInlinedOperatorFunction(builder, queryContext, contextInstance);
OperatorBuilder::OperatorFunction::OperatorFunctionResult childResult =
                childProduceFunction.invokeOperator(builder);
```

Perhaps the name buildInlinedOperatorFunction is contradictory: the code is either inline, or a separate function. Sorry about that. In this case, the code that produces the rows is injected directly into the current function. The struct OperatorFunctionResult allows you to abstract from where the child operator values come from. As I mentioned before, the child's code can be generated inline or in a separate function. This structure makes this transparent. The two most important parts of this structure are:

```
struct OperatorFunctionResult {
    llvm::Value∗ inEOF = nullptr;
    ExpressionResolver expressionResolver;
    ...
}
```

inEOF is simply a Boolean value that indicates whether the child returned EOF or not. expressionResolver works like a dictionary and allows to bring the values of the row that was generated. Continuing with the code:

```
// Increment the counter
llvm::Value∗ next = builder.CreateAdd(currentCounter,
                builder.getConstant<LimitOperator::limit_type>(1));
builder.CreateStore(next, counterVariablePtr);
```

This is quite trivial. We increase the current counter by 1 and store the result in the counter variable.

```
// Generate the requested expressions.
// This operator is not able to generate expressions on its own, so it
// just copies the expressions from the child node.
for (const SQLExpression& expression : this−>_operator−>expressions)
    this−>setOutputValue(builder, expression, childResult.expressionResolver.
                getResolvedExpression(expression, builder));
```

This piece of code "records" the output values of this operator. Here basically it copies the values from input to output. Note that when I say that it copies the values, it does not mean that it makes a true copy of the values. In this particular case, the output values are stored in the same registers as the input values. This means that there will be zero overhead for passing the values of the child operator to the parent operator (if there were any, in this particular case there is not). This is the most important concept of all!

```
builder.CreateCondBr(childResult.inEOF, reachedEOFBlock, returnRowBlock);
}
```

It checks whether the child operator returned EOF. In that case, this operator will also return EOF, otherwise it will "issue" the current row.

Two questions can be asked:

1. Why do we check for EOF after obtaining the values of the child operator?
2. What happens if the child arrived at EOF? What values will be put in the records that we generate?

The answers are:

1. The most logical thing would be to first check whether EOF was reached and then get the values of the child operator. However, it is done in the other order for simplicity, since checking whether the child arrived at EOF implies an additional if. It is not an optimization (since the compiler should automatically realize this), but rather to make the code simpler.
2. If the child arrived at EOF, the generated row will have garbage and that garbage will be copied to this operator. Since we are going to return EOF, it really does not matter if the generated row has garbage.

```
builder.SetInsertPoint(returnRowBlock);
{
    this->returnRow(builder);
}
```

This is the code that indicates that a row must be generated. It is simply a jump to the "parent operator". This is the equivalent of the yield.

```
builder.SetInsertPoint(reachedEOFBlock);
{
    this->returnEOF(builder);
}
}
```

As before, the EOF indication is made by simply jumping to a special block that transfers the EOF to the parent. And this is basically what an operator does: configure its context, receive the rows of the child operator, and generate the output for the parent operator.

5 Final Remarks

5.1 Discussion

If I had to do everything from scratch, here is what I would do:

1. I would not make sqlite3−grandata a dependency. In fact, it would not make it so dependent on SQL, since there are some queries that are difficult to express in SQL. For this we would have to make a DSL and a parser, which can be quite cumbersome. Or use Lua as an embedded language to express queries to the base.
2. Definitely not in C ++. There is a lot of boilerplate code that only serves to satisfy the gods of C ++. I would do it in Scala. The problem is that there are no bindings of LLVM for Scala, but it is not that difficult to do (see here).

29

3. I would eliminate the indexes and make the columns compressed. This would imply that all queries would be FULL SCAN, but they would be sequential rather than random scans and in most cases the queries would be faster. Perhaps a good idea would be to read directly files with the Parquet format, which already have compression included.

4. I would do it distributed, although this is very hard to do. Maybe with Zookeper or something that allows synchronization. This would allow to alleviate the problem of FULL SCAN.

5. I would do it client-server. Nowadays it is imbibed and this generates many problems and inconveniences.

5.2   Summary

In this paper, we presented Snel, a relational database engine featuring Just-In-Time (JIT) compilation of queries and columnar data representation. This engine was designed for fast on-line analytics by leveraging the LLVM compiler infrastructure.

Unlike traditional database engines, it does not provide a client-server interface. Instead, it exposes its interface as an extension to SQLite, for a simple interactive usage from command line and for embedding in applications. Since Snel tables are read-only, it does not provide features like transactions or updates. This allows queries to be very fast since they don't have the overhead of table locking or ensuring consistency.

References

[1] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, p. 75, IEEE Computer Society, 2004.

[2] Y. Leo, E. Fleury, J. I. Alvarez-Hamelin, C. Sarraute, and M. Karsai, "Socioeconomic correlations and stratification in social-communication networks," *Journal of the Royal Society Interface*, vol. 13, no. 125, 2016.

[3] Y. Leo, M. Karsai, C. Sarraute, and E. Fleury, "Higher-order correlations of consumption patterns in social-economic networks," in *NetMob 2017 (Fifth Conference on the Scientific Analysis of Mobile Phone Datasets)*, Apr 2017.

[4] Y. Leo, M. Karsai, C. Sarraute, and E. Fleury, "Correlations of consumption patterns in social-economic networks," in *2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pp. 493–500, IEEE, Aug 2016.

[5] M. Fixman, A. Berenstein, J. Brea, M. Minnoni, M. Travizano, and C. Sarraute, "A Bayesian approach to income inference in a communication network," in *2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pp. 579–582, IEEE, Aug 2016.

[6] S. Luo, F. Morone, C. Sarraute, M. Travizano, and H. A. Makse, "Inferring personal economic status from social network location," *Nature Communications*, vol. 8, May 2017.

[7] M. Óskarsdóttir, C. Bravo, C. Sarraute, J. Vanthienen, and B. Baesens, "The value of big data for credit scoring: Enhancing financial inclusion using mobile phone data and social network analytics," *Applied Soft Computing*, 2018.

[8] R. Di Clemente, M. Luengo-Oroz, M. Travizano, S. Xu, B. Vaitla, and M. C. González, "Sequence of purchases in credit card data reveal life styles in urban populations," *arXiv preprint arXiv:1703.00409*, 2017.

[9] N. Ponieman, A. Salles, and C. Sarraute, "Human mobility and predictability enriched by social phenomena information," *Proceedings of the 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining, ASONAM 2013*, pp. 1331–1336, 2013.

[10] N. B. Ponieman, C. Sarraute, M. Minnoni, M. Travizano, P. R. Zivic, and A. Salles, "Mobility and sociocultural events in mobile phone data records," *AI Communications*, vol. 29, pp. 77–86, sep 2015.

[11] S. Anapolsky, C. Lang, N. Ponieman, and C. Sarraute, "Exploración y análisis de datos de telefonıa celular para estudiar comportamientos de movilidad en la ciudad de Buenos Aires," *XVIII CLATPU*, 2014.

[12] C. Sarraute, J. Brea, J. Burroni, K. Wehmuth, A. Ziviani, and J. I. Alvarez-Hamelin, "Social events in a time-varying mobile phone graph," in *Fourth Conference on the Scientific Analysis of Mobile Phone Datasets (NetMob 2015), MIT Media Lab, Cambridge, USA*, 2015.

[13] C. Sarraute, C. Lang, N. B. Ponieman, and S. Anapolsky, "The city pulse of Buenos Aires," in *Fourth Conference on the Scientific Analysis of Mobile Phone Datasets (NetMob 2015), MIT Media Lab, Cambridge, USA*, 2015.

[14] E. Mucelli Rezende Oliveira, A. Carneiro Viana, C. Sarraute, J. Brea, and I. Alvarez-Hamelin, "On the regularity of human mobility," *Pervasive and Mobile Computing*, vol. 33, pp. 73–90, 2016.

[15] E. Fattori, P. Groisman, M. Minnoni, and C. Sarraute, "A new point process model for the spatial distribution of cell towers," in *Fifth Conference on the Scientific Analysis of Mobile Phone Datasets (NetMob 2017), Milan, Italy*, 2017.

[16] E. M. R. Oliveira, A. C. Viana, K. P. Naveen, and C. Sarraute, "Measurement-driven mobile data traffic modeling in a large metropolitan area," in *2015 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, IEEE, mar 2015.

[17] E. Mucelli Rezende Oliveira, A. Carneiro Viana, K. Naveen, and C. Sarraute, "Analysis and modeling of mobile data traffic in Mexico city," in *Fourth Conference on the Scientific Analysis of Mobile Phone Datasets (NetMob 2015), MIT Media Lab, Cambridge, USA*, 2015.

[18] J. de Monasterio, A. Salles, C. Lang, D. Weinberg, M. Minnoni, M. Travizano, and C. Sarraute, "Analyzing the spread of Chagas disease with mobile phone data," in *2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, IEEE, aug 2016.

[19] J. de Monasterio, A. Salles, C. Lang, D. Weinberg, M. Minnoni, M. Travizano, and C. Sarraute, "Uncovering the spread of chagas disease in Argentina and Mexico," in *Fifth Conference on the Scientific Analysis of Mobile Phone Datasets (NetMob 2017), Milan, Italy*, 2017.

[20] G. Chen, S. Hoteit, A. C. Viana, M. Fiore, and C. Sarraute, "Spatio-temporal predictability of cellular data traffic," tech. rep., INRIA Saclay-Ile-de-France, 2017.

[21] G. Chen, A. C. Viana, and C. Sarraute, "Towards an adaptive completion of sparse call detail records for mobility analysis," in *2017 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, IEEE, Mar 2017.

[22] L. Silveira, J. de Almeida, H. Marques-Neto, C. Sarraute, and A. Ziviani, "Mobhet: Predicting human mobility using heterogeneous data sources," *Computer Communications*, vol. 95, pp. 54–68, 2016.

[23] C. Sarraute and G. Calderon, "Evolution of communities with focus on stability," *arXiv preprint arXiv:1312.0912*, 2013.

[24] C. Sarraute, P. Blanc, and J. Burroni, "A study of age and gender seen through mobile phone usage patterns in Mexico," in *2014 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2014)*, IEEE, aug 2014.

[25] J. Brea, J. Burroni, M. Minnoni, and C. Sarraute, "Harnessing mobile phone social network topology to infer users demographic attributes," *Proceedings of the 8th Workshop on Social Network Mining and Analysis, SNAKDD 2014*, 2014.

[26] C. Sarraute, J. Brea, and J. Burroni, "Inference of users demographic attributes based on homophily in communication networks," in *Fourth Conference on the Scientific Analysis of Mobile Phone Datasets (NetMob 2015), MIT Media Lab, Cambridge, USA*, 2015.

[27] C. Sarraute, J. Brea, J. Burroni, and P. Blanc, "Inference of demographic attributes based on mobile phone usage patterns and social network topology," *Social Network Analysis and Mining*, vol. 5, no. 1, pp. 1–18, 2015.

[28] M. Oskarsdottir, C. Bravo, W. Verbeke, C. Sarraute, B. Baesens, and J. Vanthienen, "A comparative study of social network classifiers for predicting churn in the telecommunication industry," in *Advances in Social Networks Analysis and Mining (ASONAM), 2016 IEEE/ACM International Conference on*, pp. 1151–1158, IEEE, 2016.

[29] M. Oskarsdottir, C. Bravo, W. Verbeke, C. Sarraute, B. Baesens, and J. Vanthienen, "Churn prediction in the telecommunication industry using social network analytics," in *Fifth Conference on the Scientific Analysis of Mobile Phone Datasets (NetMob 2017), Milan, Italy*, 2017.

[30] M. Óskarsdóttir, C. Bravo, W. Verbeke, C. Sarraute, B. Baesens, and J. Vanthienen, "Social network analytics for churn prediction in telco: Model building, evaluation and network architecture," *Expert Systems with Applications*, vol. 85, pp. 204–220, 2017.

[31] G. Chen, S. Hoteit, A. C. Viana, M. Fiore, and C. Sarraute, "Relevance of context for the temporal completion of call detail record," tech. rep., INRIA Saclay, 2016.

[32] E. Mucelli Rezende Oliveira, A. Carneiro Viana, K. Naveen, and C. Sarraute, "Mobile data traffic modeling: Revealing temporal facets," *Computer Networks*, vol. 112, pp. 176–193, 2017.

[33] S. Jouaber, Y. Leo, C. Sarraute, E. Fleury, and M. Karsai, "Impact of university admission on student's egocentric network," in *Fifth Conference on the Scientific Analysis of Mobile Phone Datasets (NetMob 2017), Milan, Italy*, 2017.

[34] M. Fixman, M. Minnoni, and C. Sarraute, "Comparison of feature extraction methods and predictors for income inference," in *Simposio Argentino de GRANdes DAtos (AGRANDA)-JAIIO 46 (Córdoba, 2017)*, 2017.

[35] Y. Liao, W. Du, M. Karsai, C. Sarraute, M. Minnoni, and E. Fleury, "Prepaid or postpaid? that is the question. novel methods of subscription type prediction in mobile phone services," *arXiv preprint arXiv:1706.10172*, 2017.

[36] R. A. Lorie, *XRM: An extended (N-ary) relational memory*. IBM, 1974.

[37] W. J. McKenna, "Efficient search in extensible database query optimization: the volcano optimizer generator," *University of Colorado at Boulder*, 1993.

[38] T. Neumann, "Efficiently compiling efficient query plans for modern hardware," *Proceedings of the VLDB Endowment*, vol. 4, no. 9, pp. 539–550, 2011.