AgilData-Stream Data Processing Execution Planner

Colorado School of Mines
Summer Field Session 2015

Colin Marshall
Nick Miller
Colton Thomas

TABLE OF CONTENTS

INTRODUCTION
Page 3

REQUIREMENTS
Page 3

SYSTEM ARCHITECTURE
Page 5

TECHNICAL DESIGN
Page 6

DESIGN AND IMPLEMENTATION DECISIONS
Page 12

RESULTS
Page 13

APPENDIX A: SOFTWARE INSTALLATION
Page 15

APPENDIX B: RELATIONAL ALGEBRA
Page 16

INTRODUCTION

Client Description

Our client, Dan Lynn, is the CEO of CodeFutures Corporation. CodeFutures is a company that provides a number of products related to database management and solutions, particularly for “big data”, or data that is too large to be processed using traditional means.

CodeFutures is currently developing AgilData, a big-data platform that processes streams of data in near real-time. It offers support for various SQL statements. Prior to the time of this project, AgilData already had several components completed, including a tokenizer, parser, and other basic necessities for the data product they are trying to achieve. They wanted our team to help them upgrade their query processing with a query optimizer.

Product Vision

The query optimizer/execution planner created for the client will be easily integrated with the existing AgilData framework. It was written in Java and forged through a rigorous test-driven development.

The execution planner developed during this project was created (somewhat) independently of the AgilData framework; however, it is in a state where integrating the product into the AgilData framework will not be too taxing on the client. The product must be able to take in a variety of SQL statements and determine how the statements will be executed. These plans vary in terms of order of streams operated on, ordering of operators (such as selections, joins, etc.), and in some cases syntax changes.

In six weeks’ time, the product must be functional with a reasonable amount of SQL elements supported. It was very unlikely that every SQL element would be fully optimized by this planner within six weeks, but a sizeable amount of elements are supported with varying degrees of improvement/optimization.

REQUIREMENTS

The goal of this project is to research, design, and build a stream execution planner that will translate and optimize a query involving streams of data (as

opposed to static relational tables). This project has three major phases. The first is the construction of example queries. The second is designing the query optimizer/execution planner. Finally, we will have to benchmark the planner/test the output rates we achieve by optimizing the streams.

Listed below are the functional requirements and the non-functional requirements. Meeting the functional requirements specified by our client are considered a success. The non-functional requirements are desirable additions to our basic definition of success, but can be viewed as “bonus” improvements on the product.

Functional requirements

- Example queries must be created in order to ensure test driven development of the query planner.
- A benchmark system is needed in order to ensure the optimizer actually works.
    - Part of this will come from test cases in the actual Java project, as there are methods in place to calculate the theoretical output rate of both the original tree and the optimized tree.
    - Other benchmarks should be used to ensure improvement through the query optimizer. Apache Calcite was used for this purpose.
- The query planner will take the example queries and run them through the existing AgilData framework (which consists of a tokenizer, parser, etc.)
    - Once the queries are run through the existing framework, they are turned into relational algebra trees.
    - The aforementioned relational algebra trees are changed based on methods in our query optimizer. Some of the changes will include moving nodes further up or further down the tree, changing the order in which streams are joined, and simply changing syntax in special cases.
    - Parts of the SQL expression/relational algebra that MUST be supported include selections (ex: “WHERE __”), projections (ex: “SELECT * ”) and external sorting (ex: ORDER BY, GROUP BY).

Non-functional requirements

- JOIN clauses will be as fully supported as time allows.
- A D3 document may be created to show a visualization of both the logical and execution plans of executing a query.

SYSTEM ARCHITECTURE

The overall flow of executing an SQL statement is dependent on not only the query optimizer, but rather the sum of many parts. Ultimately, the AgilData framework can be simplified as a series of steps (these steps are shown in **Figure 1** below.

1. A user creates an SQL statement.
2. This statement is run through the AgilData parser.
3. Once parsed, the query optimizer optimizes the execution plan of the query.
    a. For the streams that use AgilData, rate-based optimizations (i.e. maximizing output rate of a stream) are much more important to overall optimization than cost-based optimization (i.e. reducing machine usage). Cost-based optimization is calculated using relevant information and variables found in the database dictionary.
    b. This is discussed in great detail in the “Technical Design” chapter.
4. Tuples (rows) are taken in once the best execution plan has been decided.
5. This plan is executed and returns the result(s) to the user.

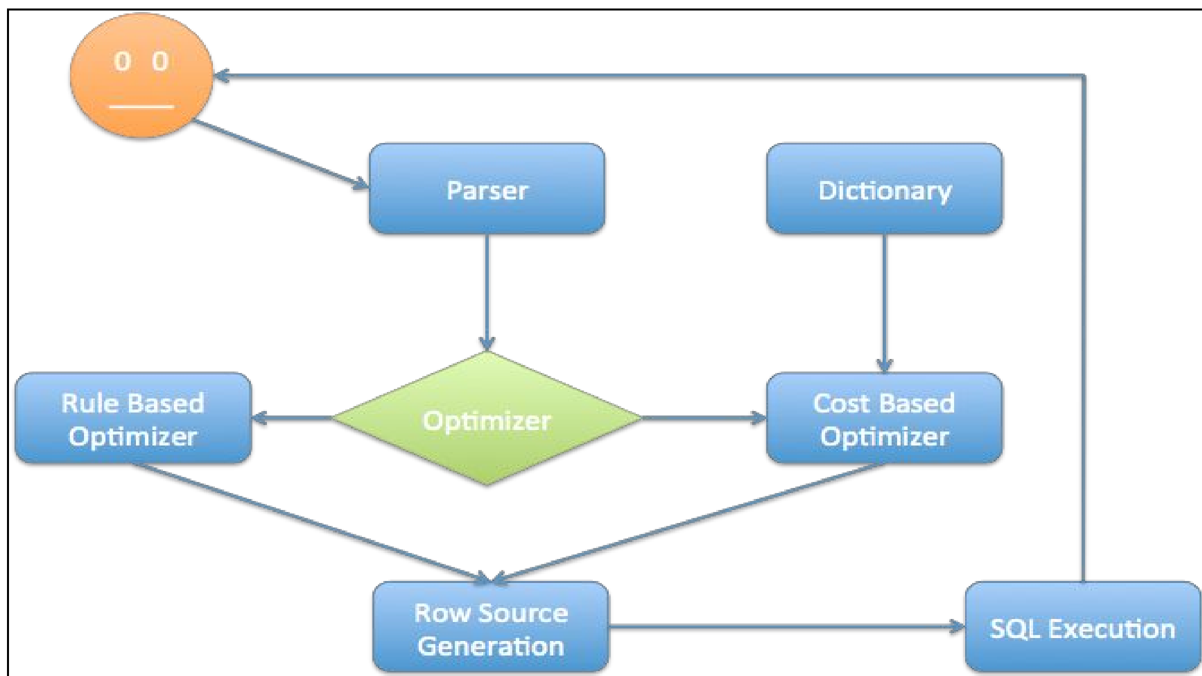

Figure 1. An overall view of an SQL statement being executed. The query optimizer developed in this project is represented as a green diamond.

The query optimizer itself was built using input from the already existing AgilData parser, and returns an optimized statement as well as an optimized tree. It does

NOT actually perform the SQL executions; that is, it does not physically join any streams nor does it select any columns; it merely tells a later component (SQL Execution on the figure) what order to perform these operations in.

The query optimizer itself is built in Java. The team briefly discussed whether or not Java, C++ or another object-oriented language would be best for the job at hand. Java won by overwhelming consensus for two main reasons: everybody in the group was comfortable with Java prior to the job beginning, and the components of AgilData that already existed prior to this project were also written in Java. By selecting Java, the team was able to worry less about setup and more about actual coding, thanks in no small part to building the existing code with Maven, allowing for easy usage of the existing AgilData parser.

TECHNICAL DESIGN

The Building Blocks of the Execution Planner

The execution planner has three primary steps in its job:

1. Take in an SQL query.
2. Optimize the SQL query.
3. Pass along an optimized SQL query to be executed.

On the surface, this seems fairly simple, as it seems as though the only tweaks that need to happen would come from changing the syntax of the query. While syntax does play a role (this is discussed in the sub-chapter “Projections and Selections”), the majority of optimization tweaks are under-the-hood, so to speak. The majority of these optimizations deal with ordering and tweaking of relational algebra.

Relational algebra is a form of algebra that deals specifically with databases and the processing of them. The foundation operations of relational algebra include selections (also known as “where” clauses in SQL), projections (“select” clauses), joins (“join” clauses), and external sorting (“group by” and “order by” clauses). As with standard algebra, order of operations plays a large role, though not only in the “Parentheses before exponents, exponents before multiplication” kind of way.

The order there operations are carried out in is where the vast majority of optimizing an SQL statement comes from.

Bearing the above paragraph in mind, the execution planner should not just keep track of syntax. It must keep track of the order in which relational algebra operations are being processed and executed. For this reason, the query optimizer is based heavily on a tree built from a parsed SQL statement. The tree's nodes consist of various relational algebra operations; some nodes are projections, others joins, others selections, and still others "group by" and "order by" clauses. A left-deep tree is constructed from the nodes that represent the original SQL statement, and is then optimized in various ways by the query optimizer, resulting in rearranging nodes on the tree so relational algebra is carried out in the most efficient way possible. The optimizer then passes the tree along with the new syntax for the expression itself, resulting in a truly optimized execution plan.

What should Optimization be Focused on?

While query optimizers have existed for a very long time for static relational tables (i.e. "standard" table-based databases), the development of query optimizers for big-data/stream databases is in its infancy. As such, a great deal of this project was research of both the implementation of streams in databases and the mathematical methods used to decide the optimal execution plan. This means a sizeable portion was deciding what mathematical formulas/algorithms to use in the calculation of output rates of stream operations.

The first major question that needed to be answered to design the software was whether or not stream operations should be calculated using cost-based optimization or rate-based optimization. Cost-based optimization aims to use as few system resources as possible to produce a good output rate. In theory, this can still lead to a great execution plan, but in practice this tends to equate to low system usage with a low output rate. This is not necessarily a bad thing in

standard relational databases, as a limited number of tuples will be generated so the latency would not be nearly as noticeable.

For streams, however, this can lead to disastrous latency. Because streams represent big-data, it makes more sense to optimize output rate FIRST and to whittle down the usage of system resources after that. This maximizes the amount of tuples produced per unit of time, which in turn reduces the latency of query execution to a reasonable amount.

Optimizing “Join” clauses

While “join” clauses were among the last to be OPTIMIZED, they were among the first to simply have groundwork laid. Join clauses are unique in that there are multiple algorithms that can happen under the hood to perform a join despite the keywords “join” or “left join” or so forth not indicating which join algorithm is used. The three main join algorithms are the nested-loop join, the hash join, and the sort-merge join. The query optimizer not only rearranges the order in which streams are joined, but also decides which of the aforementioned algorithms is used based on a variety of factors.

Mathematical research and reasoning led to one major formula. This formula is used to calculate the ultimate output rate of both a nested-loop join and of a hash join. This formula is shown in **Figure 2**.

$$\frac{f \cdot r_l \cdot r_r \cdot t}{r_l \cdot C_l + r_r \cdot C_r}$$

Figure 2. The formula used for both hash and nested-loop joins.

The output rate formula involves $\mathbf{r_l}$, $\mathbf{r_r}$ (the rates of input for the left stream and right stream respectively), the selectivity **f**, and the time **t** the stream is being viewed/processed. It also involves $\mathbf{c_l}$ and $\mathbf{c_r}$ which are the cost of processing the left stream and right stream respectively. $\mathbf{c_l}$ and $\mathbf{c_r}$ differ depending on whether a hash join rate is being calculated or a nested loop join is being calculated. If a

nested loop is being calculated, these values are $\mathbf{c_l = move + r_l{*}t + comp}$ and $\mathbf{c_r = move + r_r{*}t + comp}$ where ‘move’ is the cost of moving an input object from input buffers into main memory, and ‘comp’ is the cost of performing an in-memory comparison between two objects. If a hash join is being calculated, these values are $\mathbf{c_l = c_r = move + hash + probe}$ where ‘hash’ is the cost of hashing an object onto a hash table, and ‘probe’ is the cost of probing a hash table for output.

The formulas described in the previous paragraph are only used in the software after the software decides that a sort-merge join is NOT the best method of joining. There is no formula to determine if the sort-merge join is best; the usage of the sort-merge join is dependent on something much simpler. If an “Order By” clause is present that orders the column(s) the streams are being joined on, a sort-merge join is used because no sorting is required because the Order By clause has already done so. This makes sort-merge joins lightning fast. Sort-merge joins are rarely viable regarding streams if no such clause is present. A flowchart that summarizes how the software determines which join algorithm to use is shown in **Figure 3**.

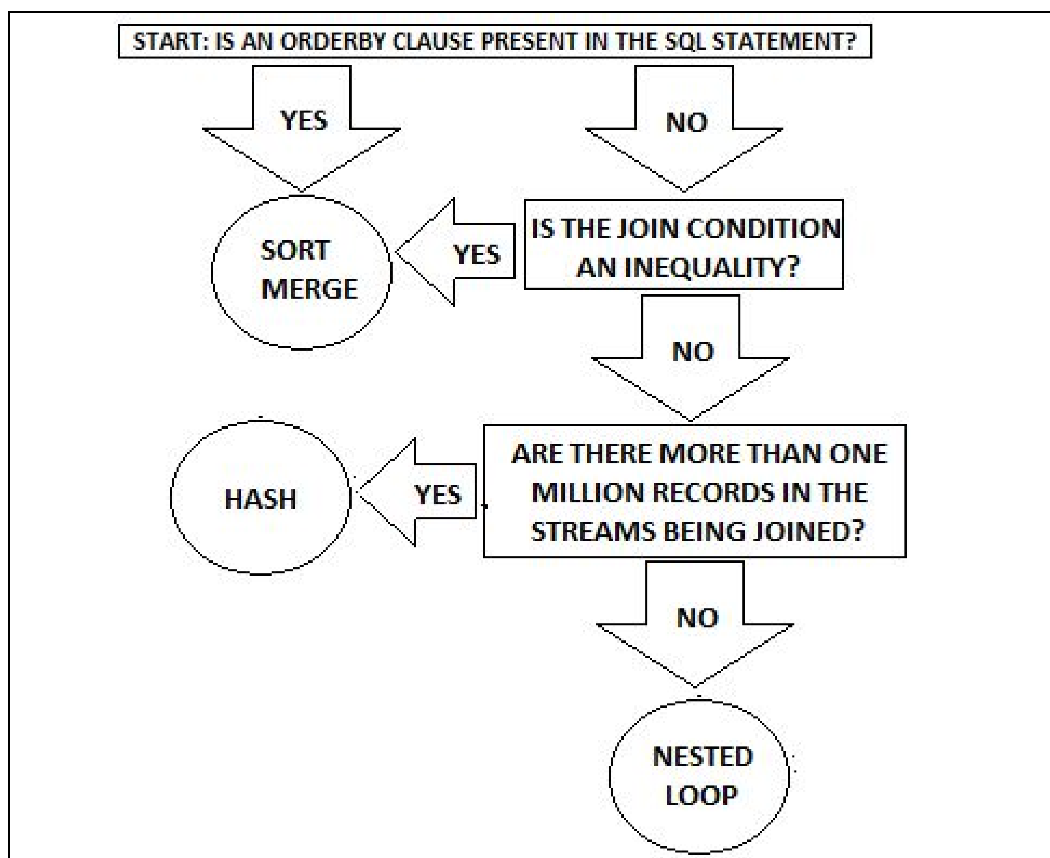

Figure 3. This is a simplified process of how the query optimizer decides which type of join to use when a join clause is present in the original SQL statement.

In addition to deciding which TYPE of join to use, deciding the order in which streams are joined is also very important. This is not really applicable when only one join clause and two streams are present, but when more than two streams are joined there needs to be consideration on which two streams are joined first.

Optimizing “Group By” and “Order By” clauses

“Order by” clauses are closely related to join optimization. In the previous section, the decision of using one of three well-known join types (nested loop, hash, sort-merge) was discussed. As was stated there, sort-merge joins are only used if an order by clause is present and is ordering by the relevant column/attribute. Thus, in “step 1” of order by optimization, a join is assigned the type of “sort-merge”. After this, the order by clause (represented as a node in the tree talked about in the beginning of this chapter) is re-visited. It searches the tree for the join node where a sort-merge flag was set (an enumerated value

“MERGED” of type “MergeType”), and repositions itself to just underneath that node. This allows it to be executed just before the sort-merge join is, maximizing efficiency of that particular join process.

If a sort-merge join is NOT present, order by and group by clauses are automatically optimized by the program when a left-deep tree is created; they are placed as the penultimate nodes in the execution plan tree (right before the projection node). Multiple sources of research agreed that this was the right place for these types of statements, as did logic itself (generally speaking, the less data there is to sort, the faster the sorting will be).

Projections and Selections

Oddly enough, selections are something of a misnomer. Selections are represented as the “where” clause in an SQL expression, NOT the “select” clause. The “select” clause represents projections.

Despite being the very first clause in nearly all SQL statements, projections are actually the last nodes to be optimized, nearly without exception. This makes sense given that it selects the columns/attributes to be displayed in the end for the end user; it is much faster to select columns that for all intents and purposes have their data finalized than it is to select columns that have had no filtering done to rule out certain tuples.

Apart from understanding the placement of projections at the end of the tree, not much can be done to optimize a projection. The one thing that can be done past node placement is a simple syntax change on the original SQL statement. Oftentimes, when a user wants to see all columns/attributes for the tuples produced by the query, their projection will have the syntax, “SELECT *.” This is convenient for the user as “*” is shorthand for selecting every single column in the tuples produced. The system, however, is actually faster if the user types out every single column. Naturally, it is oftentimes far too time-consuming for a user to type out every single column, so the query optimizer will detect if “SELECT *” has been used in the query, and replaces “*” with the relevant columns/attributes.

Selections are represented by the “where” clause, and they select which tuples will be processed and operated on. Much like projections, one of their most important optimization features is proper placement in the tree. In fact, many queries would not even be functional if selections were placed in the wrong spot. For example, if a join clause happens before the selections that “feed” the join with the proper tuples from input streams, there will be no streams to join. This strongly affirms the fact that selections need to be placed into the tree as deep as possible to ensure being executed first.

DESIGN AND IMPLEMENTATION DECISIONS

1. Rate-based optimization (R.B.O.) vs. Cost-based optimization (C.B.O.)
   We chose to run the system with a larger emphasis on rate-based optimization because the client was already aware and accepting of the fact that their large data sets will come with bigger costs. It is vital to optimize a stream for the output rate of processed tuples. While other aspects of research in this project featured clashing opinions in papers and write-ups, every paper on the topic of optimizing queries regarding streams agreed that rate-based optimization is far more important.
2. Dictation of table and stream statistics of the softwares databases
   a. We chose to model our system off of an open source database query optimizer apache calcite to implement the most realistic numbers for our tests as possible
   b. Since our client wants to implement the system into their existing program themselves, we also made a few “example tables” or “example streams” and a “database” class that holds information we would expect to see from their database. This makes our functions have access to some number we would expect to actually see.
3. Order of optimization
   The time allotted for this project was six weeks. This limited timeframe restricted us from optimizing every possible element of an SQL statement, so we had to decide which order to optimize elements in. Elements that absolutely needed to be completed were tackled first. Elements that we wanted to lay framework for and optimize as much as possible but not prioritize were tackled later. In order, the following were worked on:
   a. Selections (“where __” clauses)
   b. Projections (“select *” clauses)
   c. Group by clauses

   d. Order by clauses
   e. Execution order
   f. Join clauses and merging streams
4. Housing the relational algebra functions: how the nodes were set up, why a tree may or may not be necessary, etc.
   a. We decided to build an abstract class for the nodes. This way all nodes would have the same basic properties but they could still hold important information relative to their specific type of node. For example, select statements hold their conditions and join statements hold the two tables they are joining.
   b. We build the AST tree from the company's SQLSelect because it gives vital insight for the JOIN property of merging streams. Also it makes for an easier way to integrate into the system and we didn't have to make guesses about the type of information that was available after the SQL statement was parsed.
   c. We decided a tree would of nodes would be the easiest way to rearrange the order of the operators and operands. Another reason behind this is so that we can easily see a progression on which operations should be performed first and which should be performed last.

RESULTS

During this project our team completed several requirements asked for by our client. Our project, to build a query optimizer for streams, was broken up into several smaller tasks where some were mandatory and some were for if we had time to implement them.

Mandatory Tasks:

- Generate example queries
- Determine how to measure cost
- Generate relational tree
- Optimize projection, order by, group by, and selection nodes

Non Mandatory Tasks

- Optimize join nodes
- Create D3 visualization
- Further optimize tree

Out of these tasks we accomplished all of the ones that were mandatory and partially implemented optimization for join nodes. Cost estimation was determined through extensive research, since query optimization for streams is a relatively new topic. Our product takes in an ASTNode generated by the AgilSQLParser created by the clients and outputs the root node of the binary tree that is the most optimal order of optimization for the query. Example queries were made within tests and then sent through the parser and then through our optimizer to return the root of the binary tree. The optimizer successfully generates nodes from the ASTNode and puts them into an initial, unoptimized tree. From there the optimizer orders SELECT statements relative to JOIN statements and ORDER BY statements relative to merge joins. Currently it is very near accomplishing determining the most optimal order for JOIN nodes to be ordered in and will be able to rearrange the nodes in that order soon. Overall, this makes our optimizer able to optimize the three basic parts of SQL queries: projections, selections, and joins.

Lessons Learned

Our group had zero issues in terms of slacking off and laziness/putting off difficult segThere were a few hiccups along the way where we researched before tackling the next segment, but did not research thoroughly enough because we did not want to get behind schedule. A couple of hours may have been saved if we had researched more thoroughly in those times. In short, we were almost too ambitious in some cases.
Documentation is very important; the more documentation we included in our individual segments of code, the easier it was for other teammates to jump in and start using those functions/classes.
Communication with the client is key. This was not a regret or a mistake as we had good communication with the client overall, but as taxing as this project was it was noticeable how much of a difference it made to ask technical questions of the client.ments. On the other hand we were sometimes too eager to dive into coding.

APPENDIX A: SOFTWARE INSTALLATION

This software uses a variety of existing libraries from the AgilData framework, most noticeably a parser. These important libraries require a maven build to be used, otherwise the code will not work properly.

Other than this, there is no wrapped-up application as our client requested the source code as our final product. As such, we kept the source code in a private git repository and granted access to the client.

A .jar file was created strictly for a demonstration of the software, but not for the client.

APPENDIX B: RELATIONAL ALGEBRA

Relational algebra forms the backbone of any database system. While it was briefly mentioned in the report, uses of explicit notation for relational algebra were avoided to keep the flow of the report smooth. In this brief section, the relational algebra functions will be defined and have their respective symbols shown with them. The following **Figure 4** shows a basic relational algebra tree:

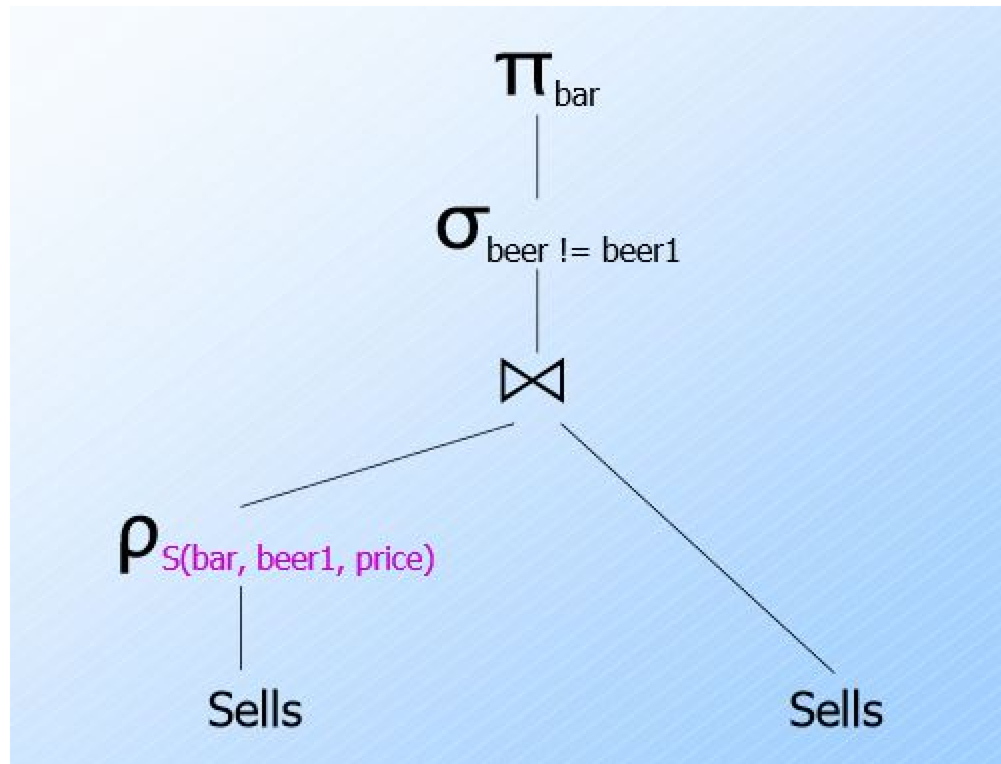

Figure 4: A relational algebra tree. It is not necessarily optimized.

In the above figure, the following relational algebra operations are present:

- Projection (π): selecting columns from a stream or table
- Selection (σ): selecting tuples (rows) from a stream or table
- Join (⋈): joins together two streams or tables based on the value(s) of certain column(s). This is also called a “natural join.”