**4.6. Script optimization**

**4.6.1. Why its important**

Performance is quite an important part of any software solution, no matter if we’re talking about applications in which users click buttons to display data or if we’re writing a query directly into, let’s say SSMS. Nobody likes to click a button, go get a coffee and then hope the results are ready by the time they come back. As computers get faster and technology moves forward users get more impatient and want things right now; without having to wait.

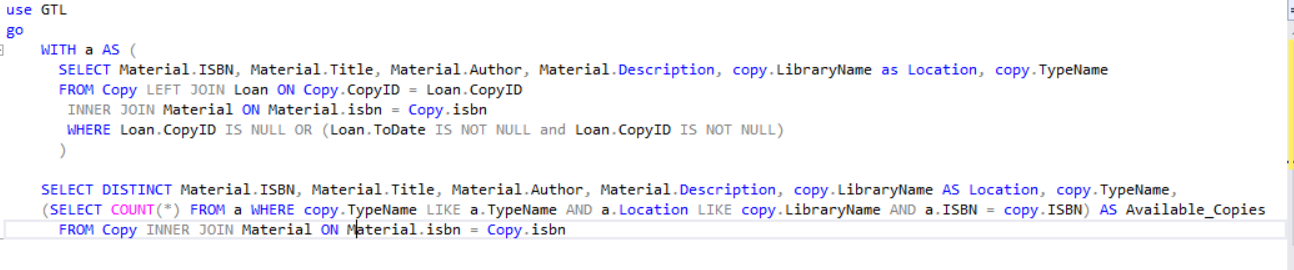
The SQL Select statement is the primary mechanism to retrieve data from a relational database. Often even clicking a single button requires query performance optimization because everything that’s actually happening under the hood is just SQL Server pulling the data from a database. Therefore, we need to make sure that our queries are performing well. This is something that we can design with a focus on the query performance, but we can also find and troubleshoot slow performance queries by identifying bottlenecks in them.

**4.6.2. How we did it**

We decided to improve the performance of our queries, by following a few simple rules, when composing different database interactions. And those simple rules are:

* Adding as many “WHERE” clauses as possible: This was done simply because the more filters we put in the query the less data will have to be processed in future actions.
* Selecting only columns that we need: By simply selecting the specific columns we needed, instead of performing a “SELECT(\*)” on the tables, we can drastically reduce the amount of data, the server will have to process in order to finish executing the query.
* Mindful use of the “JOIN” statement: Joining tables generally is a resource intensive action, because the server has to check each of the entries in both of the tables. Therefore, using as little “join” statements as possible, and even when they’re used, they’re applied on indexed columns, drastically improves the overall performance.
* Revisiting indexes: Creating an index on a table is like a double edged-sword. On one hand it can help reading information from that table, on the other, inserting or updating existing information can cause major problems, in terms of performance. Therefore, revisiting existing indexes and even temporarily removing them, while bulk data is inserted/updated, can be a great idea.
* Moving queries to stored procedures
* Add index on attributes that are used for = conditions; or are used for joins
* Refactoring queries that use “IN” to using “JOIN” instead

Those were only a few ways we used to improve our queries. But one other way we used, especially for some of the more complex queries, like the query seen in (Fig 1) was the use of execution plans. A query execution/explain plan AKA execution plan is an ordered set of steps used to access data in a SQL Server. It’s basically a map that the SQL Server is drawing to the shortest, ideally the most efficient, path to the data in our database. Such a plan is created when a query is accepted by the SQL Server and it’s coming from either an application or it’s coming from us when testing query performance.



Now there are two types of execution plans:

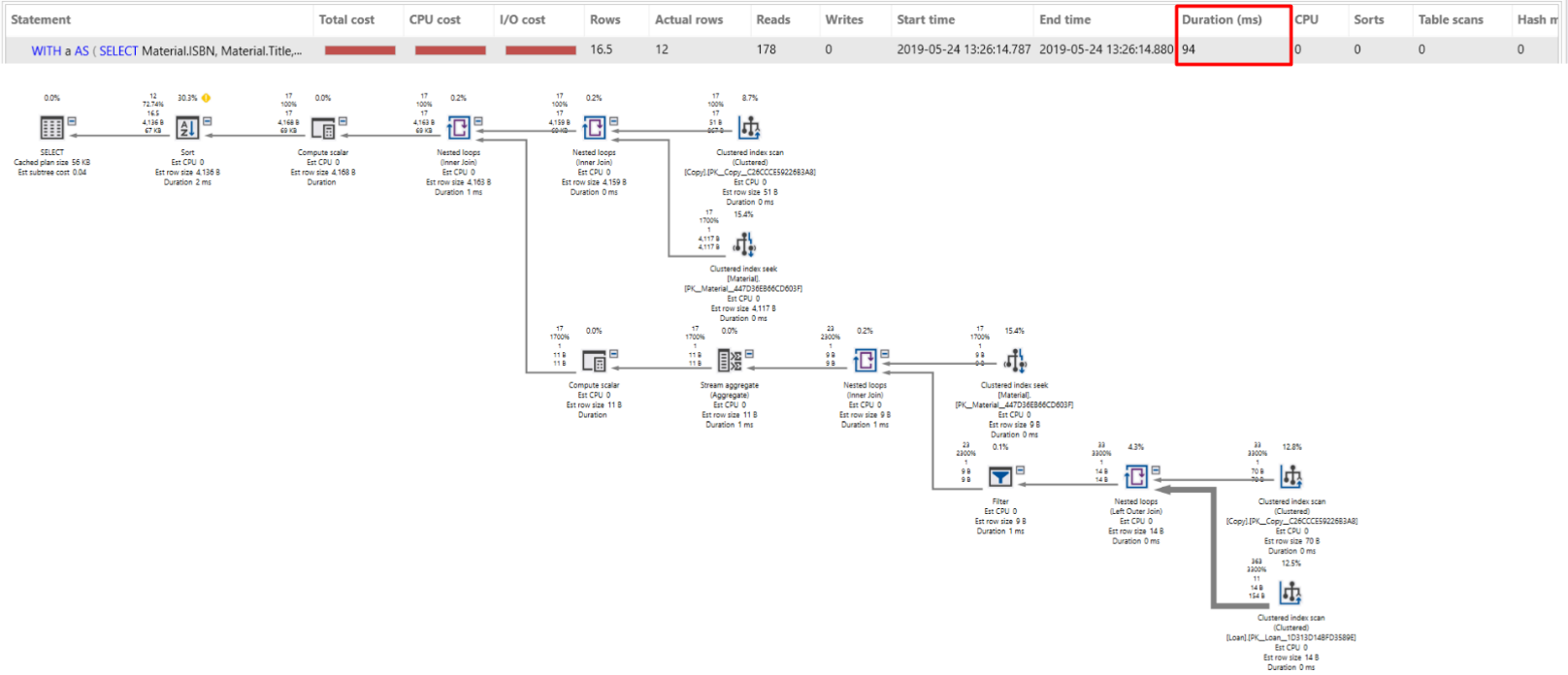
* Estimated execution plan: an execution plan generated by the SQL server using various statistical tools, in order to “guess-timate” the most efficient way this query can be converted into smaller tasks.
* Actual execution plan: an execution plan that is generated only after the query was executed against a specific database. This execution plan is way more useful when it comes to optimizing a query as this plan contains information such as: run time, returned number of entries, etc. But it also comes with the downside of requiring to actually execute the query first, which might not be the brightest idea if the only available database, is the one that the customers are currently using.

Since we only have one database, the one we used to develop this application, we decided that the actual execution plan would yield better results when trying to optimize our queries.

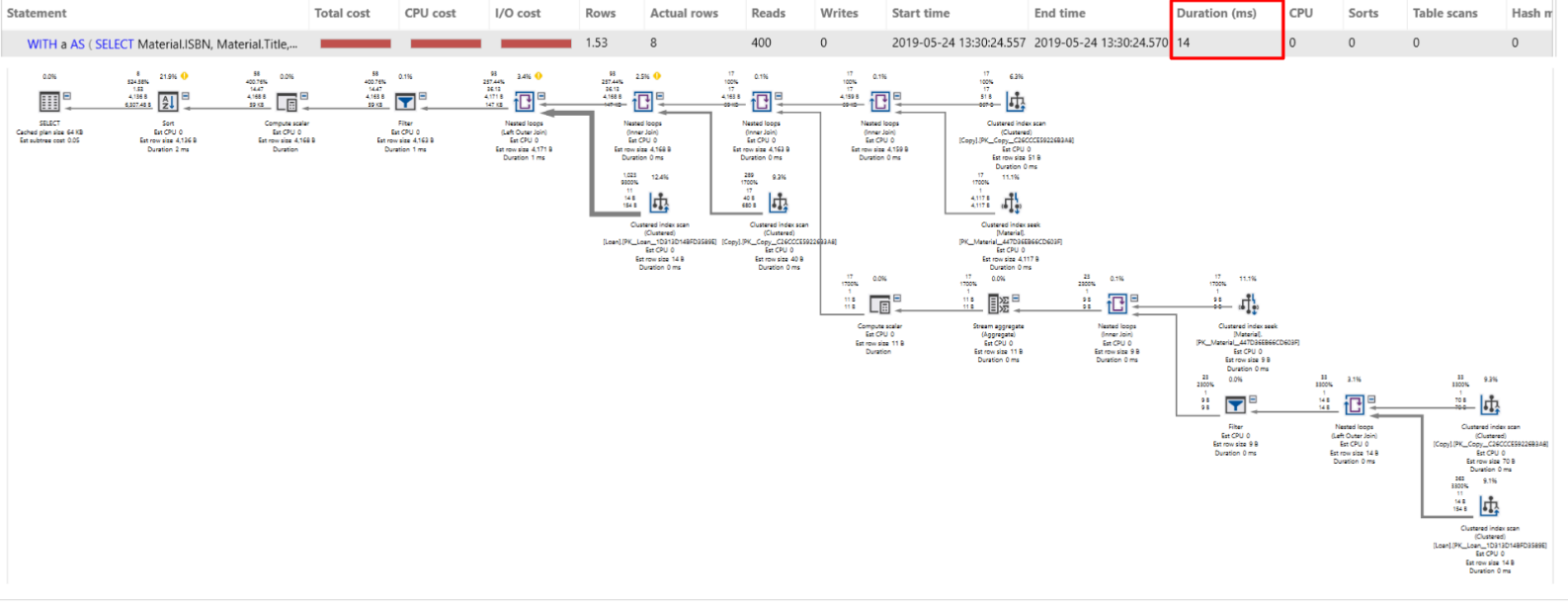
**4.6.3. Optimization process**

The following part would depict the process of our query optimization, with the help of a specific query, depicted in **Fig1.** This entire process was done using ApexSQL, a software solution for SQL query optimization, that smoothens the entire process with a slick and intuitive user interface.

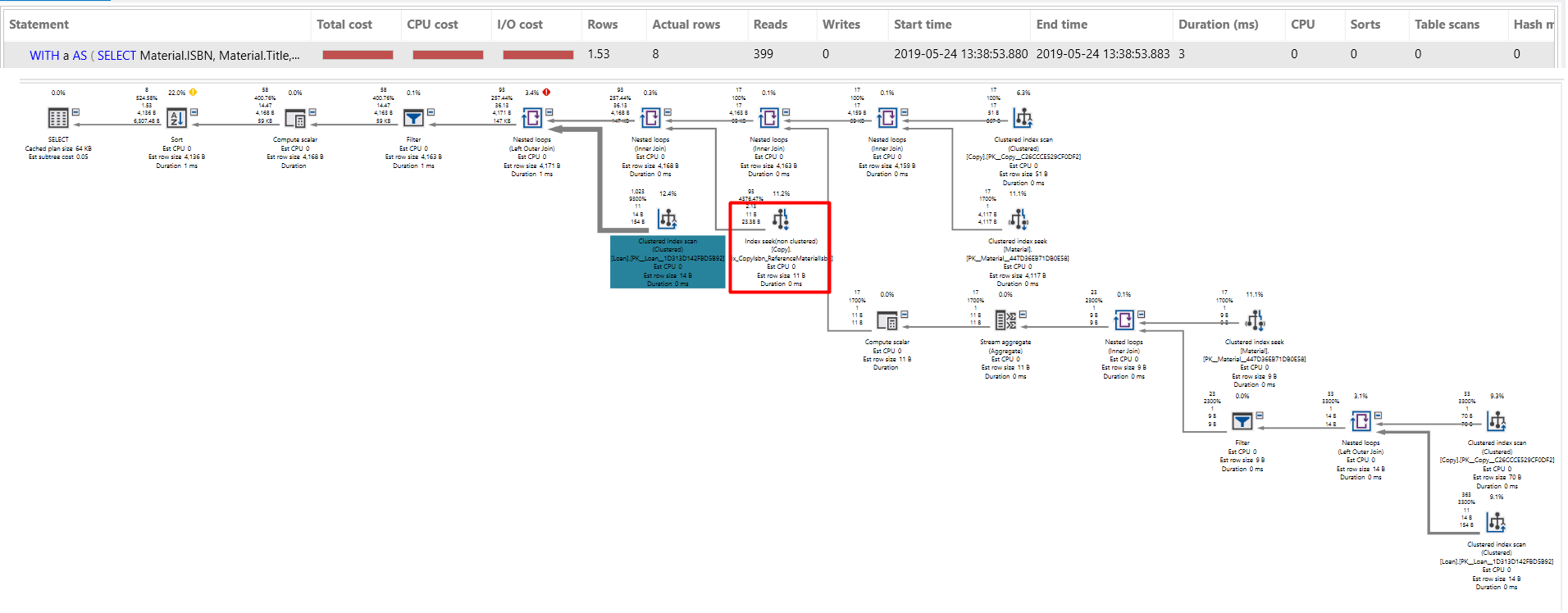
**Figure 2** represents the execution plan of the query presented in **Figure 1**. And we can see that it’s original execution time is at 94ms. Not too bad, but it could be improved.



After refactoring the query, by making it "JOIN” the already existing table “a” instead of re-joining with “Materials”, the execution time has been drastically reduced, as can be seen in **Figure 3**, to 14ms, even though the amount of reads performed has essentially doubled.



Now 14ms, is a pretty good time, and as far as we can see, there isn’t too much optimization that can be further performed on the query itself. However, we can still improve the execution time, by changing how the data inside some of the used tables, is saved. And in that regards, we can create an un-clustered index on the “Copy.Isbn” field, which would result in the following execution plan **(Fig 4).**



The simple creation of an index resulted in a 3ms execution time. Change done by the fact that a “Index scan” became an “Index seek” action inside the execution plan.

Now there are still some table scans inside the current execution plan, scans which perhaps could be changed to “seek” actions instead. Changes which will further reduce the execution time of this particular query. One good example being the “Index scan” performed on the “Loan” table, however due to the fact that the “Loan” table will be under constant “threat” of writing and updating actions; in fact, according to the problem statement, we expect around 300.000 writes and around 299.250 updates (5% of the overdue books are never returned), per year; we decided to keep the “Loan” table with as few indexes as possible.

**Used references**

<https://www.ibm.com/support/knowledgecenter/en/SSZLC2_9.0.0/com.ibm.commerce.developer.doc/refs/rsdperformanceworkspaces.htm>

<https://www.apriorit.com/dev-blog/381-sql-query-optimization>

<https://www.w3schools.com/sql/sql_create_index.asp>