Exploratory Database Programming for Oracle MLE

Jonas Grunert
jonas.grunert@student.hpi.de

ABSTRACT

With the introduction of the Oracle Multi-Language Engine into the Oracle database it is now possible to execute JavaScript and Python within the database. Access to the data is possible using callouts. Executing a writing performant code is a complex task and requires an in-depth knowledge of the database.

This paper proposes a solution using exploratory programming from the browser. This allows to write code in the browser, execute it from there and working in the browser with the results. The solution uses a server, which transforms requests from the browser into SQL statements, which execute the written code within a prepared environment in the database.

The solution provides a fast feedback loop, while outperforming classical exploratory database programming setups.

KEYWORDS

exploratory programming, database programming, multi language engine

ACM Reference Format:
Jonas Grunert. 2021. Exploratory Database Programming for Oracle MLE. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In the space of exploratory programming the behavior of the program is not well defined and the code goes through many variations to find the best approach through experimentation. A similar approach often is used to explore complex data. The query often changes and as such it should be easy to adapt the implementation. The data is often stored in relational databases, which are accessible by the SQL (Structured Query Language). While SQL is a highly expressive language for retrieving and updating data, it suffers from the loss of expressiveness in comparison to Turing-complete languages. Especially exploring a dataset with SQL is quite different from ideating code in higher level languages like Python or JavaScript.

Oracle offers such a relational database with a broad industry adoption. A recent experimental addition to the database is the inclusion of Oracle Multi Language Engine (MLE) [5]. It allows writing code either in JavaScript or Python, which gets executed within the database and allows access to the data from a higher level language. This opens up a lot of possibilities for exploratory programming, the only issue being a complex setup to get the code executed inside of the database. This paper demonstrates an approach to use exploratory programming for Oracle MLE in JavaScript using the Browser and transferring data from the database to the browser to visualize it. The aim is to reduce the amount of distribution boundaries and amount of time needed to explore a database.

1.1 Background

The Oracle MLE is embedded into the Oracle database and allows the execution of JavaScript or Python. The provided code can access the database through callouts. A callout is a specially loaded module provided by the database. It contains a function to synchronously execute SQL Queries and returns a lazily evaluated result. The JavaScript Engine embedded has neither access to

a browser environment runtime library nor a Node.js runtime library.

Despite those limitations it is entirely possible to write complex algorithms like the Available-To-Promise algorithm. A clear application of the capabilities are data interaction heavy procedures, which can be improved significantly by colocating data and execution code. To call JavaScript or Python code it has to be exposed with a SQL interface. It then may be used to provide field validations, execute CRON jobs or complex data object update logic.

To get the code to be executed into the database a series of dependent steps have to be executed:

- Loading textual representation of the code into the database
- Registering the textual presentation as dynamic module
- Register functions as executable functions from SQL with type annotations
- Calling the function from SQL

Accessing the surrounding database and its data is achieved in JavaScript using a callout module. The module allows the synchronous execution of SQL and returns a result set. The set has multiple meta properties of the executed query and a rows property when performing SELECT statements. Those rows are a lazily evaluated iterable, which allows execution of code per row. While Oracle offers the opportunities to binding external variables it does not permit the usage of the Oracle specific PL/SQL, which is a superset of SQL. As such many of the database internally are not available to be used from within the database.

1.2 Motivation

Relational databases are able to store large volumes of collected data. The database querying language SQL though, does not lend itself to exploratory programming. The ability of MLE to allow JavaScript to be executed within the database is a great opportunity to enable a better approach to exploratory programming with relational data. As JavaScript is able to be run within a browser and on the database it allows for simple transitioning from writing code for the visualization to writing code for accessing the data. This reduces the amount of distribution boundaries from two to one and writing both code fragments within one file.

As shown above the steps to execute the code in database are lengthy and do not lend itself towards exploratory programming as fast iterations are not quite possible. The amount of intermediate steps necessary are breaking the immediate feedback loop. Abstracting that complexity away and interfacing with the database by merely writing code and deploying it should increase the speed of the feedback cycle and as such development speed. Finally the lazy evaluated result set has its advantages while writing performant code, but it introduces an overhead while writing the code. To decrease the overhead and allow for an easier API, which does not loose out on performance allows the developer to concentrate on exploring the data. This increases productivity and leverages the concentration on the cor task at hand.

2 RELATED WORK

Oracle themselves provide a low-code platform called APEX, which centerpiece is the Oracle Database and their PL/SQL superset. In that project they push business logic as much as possible into the database. As such the project most recently integrated the MLE JavaScript and Python adapters to allow for more complex business logic than possible with PL/SQL [4].

Furthermore the work described in this paper is based off the work in two previous seminars. During the first seminar the problem of the complex deployment was addressed using a self written CLI (Command Line Interface) program, to deploy TypeScript projects [1]. While it worked in a constrained environment there was still a disconnect between the authoring and execution of the project. In a following seminar an integration of the functionality of the CLI into a server, which is connected to Lively [2]. Lively is an "Explorative, Self-supporting, Web-based Development Environment" developed and maintained at the HPI (Hasso-Plattner-Institute) [3]. This setup allowed for the integration of custom components into Lively to develop for the database. The scope of the seminar was explicitly not to integrate data from the database into the browser. As such there was still a barrier between the execution of the code and the visualization of the returned data. This was a natural extension point for the work presented in this paper.

3 SOLUTION

Building on top of the previous seminars work this section will describe the architecture, implementation details and runtime benefits of the project as well as describing the frontend integration.

3.1 Architecture

The architecture consists of a a dockerized database with MLE enabled, a proxy server, which relays code from the browser to the database and relays data from the database back to the browser. The final piece is the browser interface, which allows code editing and integrates the data to be used in the DOM (Document Object Model).

The communication with the server uses Websockets as a relic from the previous seminar. The extension of the protocol to include dynamically executed code was easier, than rewriting the entire server. To get around the specification of types for the SQL statements the code as well as the results are encoded as a string. This allows for simple transmission of the data to and from the database as none of the types need to be specified by the developer.

Figure 1 shows the setup in depth. While the browser UI (User Interface) handles the code regardless of where it gets executed. When the users decides to execute some arbitrary code it sends a message of the test type, which get translated into a SQL Query by the relay server and which then executes and returns a cleaned result from the database. All dtaa transformation is handled within the database. The relay server merely unpacks the returned data from the oracle driver. The generated SQL query takes advantage of statement bindings to easily integrate any code without worrying about pattern escapes. This additionally improves the execution speed of identical code. Reiterating on past code is way faster due to this feature.

3.2 Implementation details

To execute arbitrary code inside the database a helper function is registered, which is written is JavaScript. This helper function takes in a string of the code to be executed and returns a string containing the result.

The code is evaluated using the eval function available in JavaScript. Code passed as binding into the executed statement is eval and the result of that eval is converted into a string using the JSON module and passed back to the server using PL/SQL out bindings. As such the function does not have limitations, which are not already present in JavaScript. While Oracle does not specify the version of JavaScript running it safe to assume that it is higher than EcmaScript Version 6 and as such eval might have access to different scopes of the module. To not have an arbitrary scope, the enclosing JavaScript binds eval always to the global scope with this. To allow for exploratory programming in the database the function maintains a state between executions, which allows to execute complex commands incrementally to explore results and adapt based on the intermediate data. Variables bound to added to the special *this* object are added to the *global* object and be available for future executions via the *this* object.

3.3 Runtime library

In this context runtime library is defined as additional code, which is not included with the JavaScript Engine or the callout module provided by Oracle, but by the solution proposed in this paper. The code is embedded into the wrapping JavaScript function in the global scope, which allows it to be accessed from anywhere. This might seem like polluting the *global* namespace, which is a JavaScript antipattern, but within an exploratory programming paradigm, which should not be used in production, the solution strikes the balance between simplicity and functionality.

The parts of the runtime library together abstract away the process of calling out to the database and returning the data. The primary user interfacing component is a template literal function called sql. It allows the execution of arbitrary SQL, with interpolated variables being passed into the query as binding variables. An example for user written code and the corresponding executed SQL statement can be seen in Figure 2. All variables are substituted and then passed with an array. Furthermore when using a *SELECT* statement the data type of all selected fields is checked and when there is not a way for the JavaScript Engine to convert the database type to a JavaScript type a reasonable conversion will be injected into the SQL statement before execution. This allows for different representations of a date for example. While within a table the data and time for example could be specified as either a timestamp or an ISO-formatted date string both get converted to JavaScript Date Object by injecting the

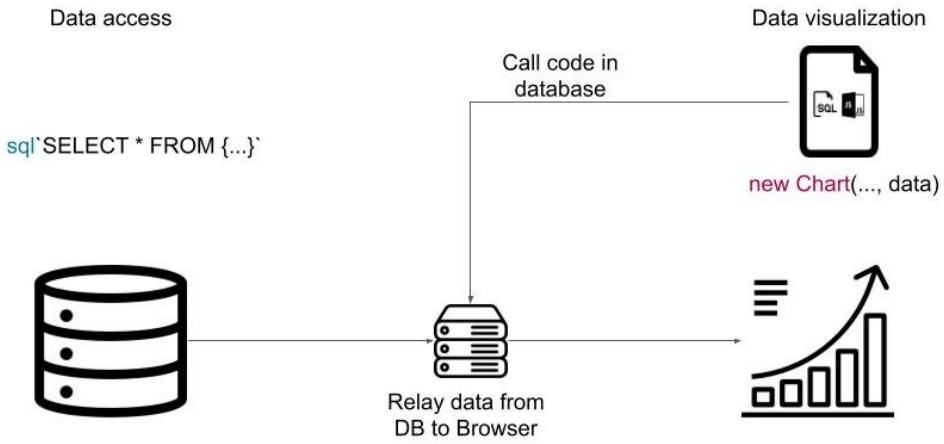

Figure 1: Architectural diagram of the implemented solution

```
SELECT                          SELECT
  id,                             id,
  kind,                           kind,
  emitTime                        cast(emitTime as date) AS emitTime
FROM logs                       FROM logs
WHERE kind = ${"failure"}       WHERE kind = :1
```

Figure 2: SQL code comparison. (Left: User Query; Right: Executed Query)

conversion. When the executed statement is data mutating a string containing the amount of altered rows is returned. When it is a data querying statement a special type of helper object is returned.

This helper object class is also part of the additional runtime library and is a thin wrapper around the result set iterable. An instance of the class wraps one rows iterable and augments it with bound functions, which allow to make use of the Map/Reduce programming paradigm. The map function takes a function as argument, which is then stored in an object internal array of mapping functions for later use. The single argument of the mapping function is an object presentation of the result from the SQL statement or the return value of the previous map function. While the result set returns returns rows it also contains metadata to allow the conversion into an object, where the keys are the selected columns with the corresponding values. To get the result from the helper object a developer can decide to either use the reduce or eject function. While eject takes no parameter it returns the dataset as an array of objects after applying all stored mapping functions to it. The reduce function on the other hand executes all mapping functions for an object and combines the result with the previous value. In that way it iterates over all values in the result set and returns the reduced value. The two parameters of the function are the previous accumulated value and the current value off the result set.

3.4 Frontend

Within the Lively Environment the exploratory environment is used by creating multiple workspaces. When creating such a workspace the developer has to decide if he wants to program in the database or use the DOM environment.

The code written is persisted across browser sessions. From within DOM workspaces one can access the return values of any database workspace using special variables. Upon executing the code within a MLE Workspace the code from that workspace is submitted to the database using Websockets to communicate with the relay server. Upon receiving a result it is transformed into a JavaScript Object and then added to the global scope and logged as string for inspection. In a DOM workspace the gloabl scope is accessible and as such access to the result from the database.

3.5 Future development

During the implementation one pain point arose when executing long running queries. Without intermediate

results or a progress tracker a developer is unable to estimate the time needed for a task. This hurts productivity and is bad for estimating implementation deadlines. There are two possible solutions, which both were unable to get implemented during this research. Anyhow the approaches will be described here. Both allow for incremental results, but differ in the control of result emittance.

One approach utilizes a helper table within the database. Control of when an incremental result is emitted lies with the database. Whenever a new result is ready it will get written to the table using a unique auto incrementing GUID (globally unique ID). The relay server listens to changes within that database table and emits events to the browser upon a new entry using websockets. This closely resembles the Event Sourcing Pattern, where data is appended only. The browsers task is it to assemble the current result and make it available to the developer in a usable fashion.

The other approach hands the control to the browser using JavaScript generator functions. A generator function allows to continuously return results, as long as there are results. Using the approach of a shared state between multiple executions such an approach could be taken to implement a generator on the browser side, which continuously asks for results from the database, which it returns it through its state one at a time. This requires a special state variable which is going to be provided by the runtime library. This allows the browser to incrementally show more complex results or even entirely different was of working, for example streaming.

4 EVALUATION

This section will describe the advantages, a small pilot benchmark and the limitations of the approach.

4.1 Advantages

The advantages of the exploratory approach allow an interactive development and exploration of the data in a higher level language. The feedback loop is shorter and error messages are more helpful than the traditional development for Oracle MLE. The integration of the execution result into the DOM environment allow for easy visualization of data. The advantages of the helper methods allow for Object-Oriented Programming Paradigm due to connecting state and behaviour, while SQL can only capture state. For more complex algorithms advanced data structures like Sets, Maps or Linked Lists may be used. Additionally the abstract away the complexity of using callouts to interact with the database.

4.2 Benchmark

To get a feeling for the performance difference between the MLE Setup and a traditional application server setup one complex query was executed in both setups. Figure 3 shows the setups next to each other. The query

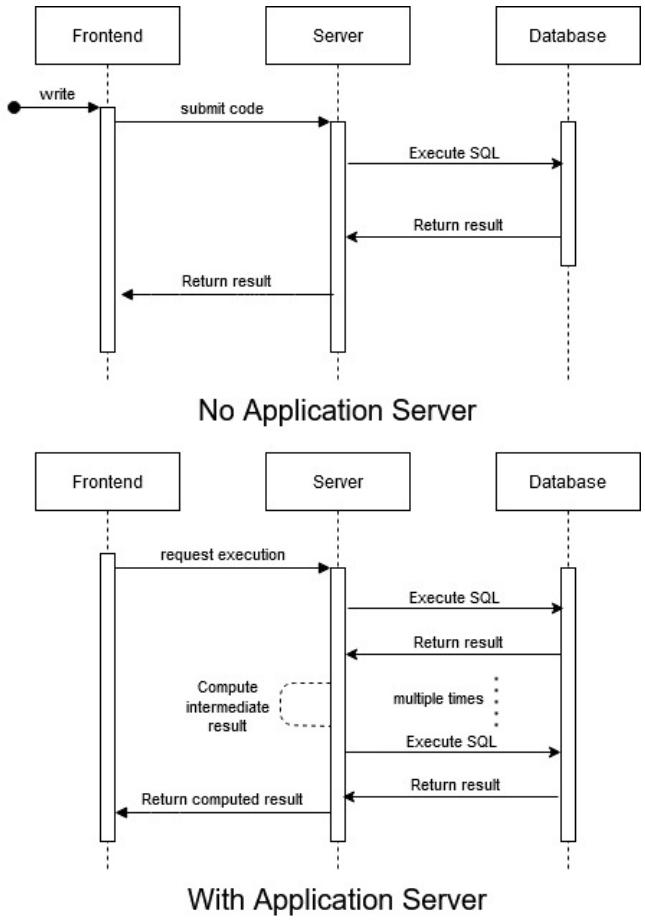

Figure 3: Comparison of the two benchmark setups

is taken from the Github Torrent and tries to find the ten projects on github, which take the longest to finalize a merge request. To execute this query three tables got imported from CSV-Files into the oracle database.

Together the data totals about 48 GB of data. When using the MLE approach the aforementioned helper functions are utilized, while the application server setup processes the data using a similar concept from the Oracle driver in Node.js. Both setups are executed upon a local machine with a database running in Docker and a native Node.js Server. The machine has 16 GB RAM and as such has no way to load all data into the RAM at the same time. The results are only an indicator of the possible performance, but are heavily in favor of the MLE approach. While it only took half the time it additionaly only occupied 3 GB of RAM compared to the Node.js server approach, which needed about 15 GB. This performance increase probably comes from the fact, that within MLE there is no need to data copy as it is in the server setup.

4.3 Limitations

While a promising step in the right direction there are still multiple limitations to the current approach. The distribution boundaries are now even smaller, but are still present within a file. The work spaces still do not allow to fully interweave MLE and DOM code. Additionally the approach does not allow for SQL bindings from the DOM code. This allows for unidirectional data code from the database to the browser, but not the otherway around, unless source code is modified. While the output as string allows for a broader application the current data output size of the database is capped at maximum 4000 bytes, what makes it impossible to transfer data heavy results. Finally the helper functions introduce a new paradigm in Map/reduce, still the execution time decreases with increased SQL knowledge with hurts developer productivity as one has to switch contexts quite frequently.

5 CONCLUSION

The approach presented within this paper allows to develop JavaScript for the Oracle MLE database. It lends itself to a exploratory style of programming without a complex setup to change code. The feedack loops are short and the state from previous executions is persisted across executions, which allows for incremental coding. The proposed solution allows to write database and browser code in one file and use JavaScript within the database. It seamlessly integrates code and results without the need of an application server.

REFERENCES

[1] Jonas Grunert. 2018. *DBPack*. Software Architrecture Group, Hasso Plattner Institute. https://github.com/jonasgrunert/dbpack

[2] Jonas Grunert. 2019. *DBPack Server*. Software Architrecture Group, Hasso Plattner Institute. https://github.com/jonasgrunert/dbpack-server

[3] Daniel Ingalls, Tim Felgentreff, Robert Hirschfeld, Robert Krahn, Jens Lincke, Marko Röder, Antero Taivalsaari, and Tommi Mikkonen. 2016. A World of Active Objects for Work and Play: The First Ten Years of Lively. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2016)*. ACM, New York, NY, USA, 238–249. https://doi.org/doi.org/10.1145/2986012.2986029

[4] Oracle. 2021. *Oracle APEX MLE*. Oracle. https://blogs.oracle.com/apex/mle-and-the-future-of-server-side-programming-in-oracle-apex

[5] Oracle. 2021. *Oracle Labs Walnut*. Oracle. https://labs.oracle.com/pls/apex/f?p=LABS:project_details:0:15