# An Empirical Study on the Possible Positive Effect of Imperative Constructs in Declarative Languages: The Case with SQL

Seyfullah Davulcu, Stefan Hanenberg[a], Ole Werger[b] and Volker Gruhn[c]
*University of Duisburg–Essen, Essen, Germany*

Keywords: SQL, Imperative, Declarative, Empirical Study.

Abstract: Today, imperative programming languages are often equipped with declarative constructs (such as lambda expressions in Java or C++). The underlying assumption (which is partly confirmed by experiments) is that imperative languages benefit from such constructs. This gives the impression that declarative programming languages are better suited for programming than imperative languages. However, the question is whether this statement holds vice versa as well, i.e., whether declarative languages benefit from imperative constructs. The present paper introduces a crossover trial where 24 students were equipped with an SQL extension that gives the illusion of imperative assignments. It turned out with high confidence ($p<.001$) that this construct -- although in principle already contained in a declarative fashion in SQL -- lets students solve a given task in only 52% of the time in comparison to the time required in standard SQL.

## 1 INTRODUCTION

Programming languages follow design principles and one of the fundamental principles is whether a language should rather be imperative or declarative. The traditional distinction between both is that code written in an imperative language describes step by step how an algorithm is executed. Code written in a declarative language rather specifies how the result looks like. Today's examples of declarative languages are functional programming languages such as Haskell, F#, Scala, etc., while today's examples of imperative languages are C, C++, or Java.

While imperative languages dominated the software market, the situation changed over the years: The functional construct "lambda expression" became integrated in different imperative languages. Moreover, tools such as IDEs also seem to see a benefit in lambda expressions. For example, already in 2009 the IDE IntelliJ visualized anonymous inner classes as lambda expressions. Hence, it looks like there is some common agreement that imperative languages benefit from the integration of declarative constructs – besides the fact, that there are only few studies that measured the effect of declarative features in imperative languages (see for example (Uesbeck et al., 2016; Lucas et al., 2019; Hanenberg and Mehlhorn, 2021; Mehlhorn and Hanenberg, 2022)). However, one could also ask whether declarative languages benefit from imperative language constructs. And among the set of declarative languages, there is one language that plays a major role in software development: the database language SQL.

The present paper introduces an experiment where SQL is equipped with a quasi–extension which gives the illusion to assign queries to variables – an extension that is not necessary, because it already exists in SQL to a certain extent. But by giving 24 subjects a task where a query consists of a number of subqueries, it turned out that the subjects required only 52% of the time they needed using standard SQL.

## 2 SQL, WITH AND SQL$_{Assign}$

SQL is an ISO standardized language for relational database systems[1] that is widely taught and applied today. In relational databases, the fundamental data structures are (named) tables having named (and typed) attributes where the actual data is represented by rows. The result of a SQL query is again a table.

The selection of data from tables is done via SE-

[a] https://orcid.org/0000-0001-5936-2143
[b] https://orcid.org/0009-0007-3226-1271
[c] https://orcid.org/0000-0003-3841-2548

[1] See iso.org/standard/63555.html

Davulcu, S., Hanenberg, S., Werger, O. and Gruhn, V.
An Empirical Study on the Possible Positive Effect of Imperative Constructs in Declarative Languages: The Case with SQL.
DOI: 10.5220/0012092300003538
In *Proceedings of the 18th International Conference on Software Technologies (ICSOFT 2023)*, pages 428-437
ISBN: 978-989-758-665-1; ISSN: 2184-2833

LECT statements where within a FROM statement tables are combined by a number of different JOIN operators. From the table that is the result of such operations certain rows can be selected via the WHERE clause and finally, the SELECT statement defines the resulting columns. Additional constructs are the GROUP BY and HAVING clauses that permit to define subtable. Moreover, SQL contains aggregate functions, i.e., functions that are applied on all elements of (sub-) tables.

SQL is a declarative language in the sense that developers specify the conditions that hold on the resulting table without specifying how the underlying data is iterated. An SQL query is one single expression. Queries can be quite difficult and it is possible that parts of queries need to be used multiple times within one SQL statement. This can be done, for example, using the WITH clause. The WITH clause is one expression consisting of a header where a name is given to one expression that later on can be used in the expression's body.

```
WITH Amounts AS (SELECT CustomerId , SUM(Amount) Res
                 FROM Payment
                 WHERE CustomerId != 42
                 GROUP BY CustomerId)
SELECT CustomerID
FROM Amounts
WHERE Res > 12345
```

Figure 1: The result of one SELECT statement is given the name `Amounts` that is used in the body of the WITH expression to refer to that statement.

An example for the WITH clause is given in Figure 1. One characteristic of the WITH clause is, that it is still one single expression. I.e., one reads the whole expression as one single execution step.

We are involved in SQL teaching and have the impression that students have problems with WITH. Our impression is, that WITH hinders students to think about the clause as a step-by-step introduction of names for expressions. As a consequence, we think that students do not use WITH and rather formulate SQL statements with subqueries – and we think this makes students less efficient.

An alternative way to formulate the above SQL statement is to use subqueries. Figure 2 describes the previous statement with a subquery. While the statement is shorter, the potential problem is that it no longer gives a clear description of the subquery: just finding the name of the subquery (Amounts) is not trivial, because it requires to identify the start and the end of the subquery. On the other hand, due to its shorter representation, it might seem attractive to students to use this presentation instead of using WITH.

```
SELECT CustomerID
FROM (SELECT CustomerId , SUM(Amount) Result
      FROM Payment
      WHERE CustomerId != 42
      GROUP BY CustomerId) Amounts
WHERE Result > 12345
```

Figure 2: Previous example using an inner statement.

Our idea was to give students a different construct that comes (syntactically) closer to the idea of imperative languages. Instead of writing a WITH clause, we give them the illusion of a variable assignment in a style known from languages such as Java, etc.: a variable name on the left-hand side to which an expression is assigned. Additionally, in order to give an impression of a sequence, we used (similar to Java) the semicolon to separate different statements.

```
Amounts = (SELECT CustomerId , SUM(Amount) Result
           FROM Payment
           WHERE CustomerId != 42
           GROUP BY CustomerId);
SELECT CustomerID
FROM Amounts
WHERE Result > 12345;
```

Figure 3: Previous example using SQL$_{Assign}$.

Figure 3 illustrates the previous example using the syntax of SQL$_{Assign}$. Obviously, it is possible to translate SQL$_{Assign}$ into SQL, where in front of an assignment only the keyword WITH has to be introduced, the equal operator has to be replaced by the keyword AS and the semicolon has to be deleted. Again, it is obvious that SQL$_{Assign}$ is only a different syntactical representation of SQL that actually does not add anything new. The only difference is, that it gives the impression that different statements are sequentially executed and it gives developers the illusion that the complete SQL statement can be read line by line.

The obvious research question is:

**RQ: Do Developers Benefit From The Introduction of SQL$_{Assign}$ in Comparison to Standard SQL?**

To answer this question, much more details are required in order to understand under what circumstances such difference might appear, etc.

## 3 RELATED WORK

We are not aware of experiments that focus on imperative features in declarative languages. But some studies analyze the effect of introducing declarative features in imperative languages (despite the fact that empirical studies in programming languages are rather rare in general, see for example (Kaijanaho, 2015)). Additionally, we consider studies that focus on the usability of SQL.

### 3.1 Studies on Imperative versus Declarative Language Constructs

Uesbeck et al. (Uesbeck et al., 2016) tested on 58 participants whether there are differences in development times for iterating collections using lambda expressions in C++ in comparison to traditional iterators. It turned out that lambda expressions required on average significantly more time. The study also took a look into error fixing times and the number of compiler errors. The results with respect to these dependent variables again favored traditional iterators.

Mehlhorn et al. studied the effect of Java's Stream API in comparison to the imperative loops on readability (Mehlhorn and Hanenberg, 2022) by measuring the time until the correct result for some code was detected. The study on 20 participants revealed the opposite effect of Uesbeck's study: the Stream API code required less time than the imperative loops.

Another study on lambda expressions in Java was performed by Lucas et al. (Lucas et al., 2019). The authors evaluated code snippets before and after the introduction of lambdas and compared it to some source code complexity measures. Additionally, they asked 28 developers to evaluate code snippets. The main results of the study was that about 50% of the developers considered the introduction of lambda expressions as an improvement. While the application of readability and understandability metrics did not find significant differences, developers perceived some code migrations towards lambda expressions negatively, especially when for-loops were replaced.

Pichler et al. (Pichler et al., 2011) studied differences between declarative and imperative business model languages. 28 students were given two types of tasks and four questions per task[2]. The models were formulated either using a declarative or an imperative business process language. The language had an effect on the response times as well as on the correctness of the answers ($p<.001$).

Although not explicitly meant to test whether a declarative or imperative specification is better (in terms of development time, answer time, or correctness), Kokuryo et al. studied how often imperative code was written using the configuration management tool Ansible (Kokuryo et al., 2020), a tool that is intended to define configurations in a declarative manner but that still provides imperative modules. The study revealed, that about 50% of the studied codes still used imperative modules.

### 3.2 Usability Studies on SQL

A study by Ahdi et al. tested common mistakes done by more than 2000 students across eight years on a data set consisting of more than 160 thousand snapshots of SQL queries. It turned out that a large percentage of errors was caused by syntactical errors – approximately 54% of faulty queries even on trivial statements contained syntax errors. The authors conclude from the study that "*a syntactic error and not a semantic error is what in most cases causes a student to abandon answering a question. Syntax errors [...] are the most common mistakes that novices make in writing their SQL statements*" (Ahadi et al., 2016).

A comparable study by Poulsen et al. (Poulsen et al., 2020) followed the approach of Ahdi et al. and studied a data set consisting of homework submissions from 286 students. Again, the SQL statements were classified according to their nature (joins, outer joins, etc.), but additionally, slightly more complex constructs (such as triggers or stored procedures) were considered as well. Again, it turned out that in almost all categories the number of syntax errors was higher than the number of semantic errors.[3] And even in the final submissions (where students had the ability to double check their statements), still 27% of the statements had syntax errors.

## 4 EXPERIMENT DESCRIPTION

### 4.1 Initial Considerations

SQL$_{Assign}$ was built upon the assumption that developers have the tendency to use inner statements in SQL instead of applying the WITH clause. We believe that this happens in situations where developers write rather complicated queries that are based on each other. For example, when it is clear from a task description that a statements such as S1 needs to be formulated whose results are required by a statement S2, we think that developers have the tendency to start with S1, but then, instead of giving it an explicit name, just use the whole statement S1 within the body of S2.

[2]From the paper, it cannot be derived what questions were asked or to what extent the questions were related to the declarative or imperative nature of the given model.

[3]There was one exception to this rule: in the group "Join and Where" 39% of statements were syntactically incorrect, while 43% of these statements had a semantic errors.

```
-- STEP1: Starting with some complicated query
SELECT a, b, c
FROM A JOIN B USING(x)
WHERE d=42
GROUP BY a, b, c
HAVING sum(e)>666

-- STEP 2: Using first Statement as Inner Statement
SELECT a, sum(b)
FROM (SELECT a, b, c
      FROM A JOIN B USING(x)
      WHERE d=42
      GROUP BY a, b, c
      HAVING sum(e)>666) complexQuery
GROUP BY a
HAVING sum(b)>999
```

Figure 4: Stepwise formulation of SQL queries using inner statements.

Figure 4 illustrates the described situation. If the task consists of multiple steps, we think that a number of people start with the first step and start with step 2 when they think they have finished step 1. But instead of just using the WITH clause, we think people rather continue writing the query by adding the additional code to the statement – as a consequence, the first statement becomes an inner statement. We think that this approach becomes more problematic if the inner statement requires some changes later on. From that we conclude that for the comparison of standard SQL and SQL$_{Assign}$ non-trivial queries are necessary that should be formulated in a step-wise manner.

Having the previous considerations in mind, we took two different approaches into account. First, our idea was to give people code to read that consists of inner queries, WITH clauses or the assignment expressions using SQL$_{Assign}$. Actually, we do believe that SQL$_{Assign}$ has already some (small) positive effect on readability, but we still think that such an experiment is not trivial, because one has to define under what circumstances a developers has actually understood the semantics of the query. Additionally, we do not think that the measured effect is large, because we think a larger, positive effect predominantly shows up when developers need to interact with the code (rewriting a query, replacing parts of a query, etc.).

We have chosen a different alternative. Instead of reading code, we let developers formulate a query. While one approach could be to give developers a finite amount of time and check how many developers have finished the task, it could also be an alternative to give developers multiple tasks in order to test, how many tasks each developer has fulfilled (an approach that has been used for example in (Uesbeck et al., 2016)), and finally, one could in principle check, what percentage of the task was fulfilled (an approach that was for example chosen in (Müller, 2005; Hanenberg, 2010)). Having the assumed benefit of SQL$_{Assign}$ in mind, we followed the approach to give developers one (or more) programming tasks and to measure how much time was required to fulfill the task.[4]

Next, the question is, what experiment setup is required. Although we had the impression that the differences between SQL and SQL$_{Assign}$ are not tiny, we were afraid that the deviation between developers could possibly hide this effect. However, so-called crossover trials (see for example (Senn, 2002; Kitchenham et al., 2003)) – experiments where developers are tested under different treatments – are an often applied technique to reduce this problem.

## 4.2 Task Description

As noted, we are not convinced that a variable assignment has a positive effect in all situations. We think the benefit shows up if queries are formulated step by step and if SQL tempts developers (for unknown reasons) to use inner statements. Hence, the task should be a query where its parts are defined step by step. Next, the query should be easily testable and the probability to achieve a correct query by luck should be low. Because of that we decided to ask for a statement that heavily relies on aggregate functions.

Figure 5 contains a corresponding task.[5] It is described in terms of subtasks – each one can be described by a corresponding query. The different elements of the query are articulated as distinct subtasks, each defined with a clear name (such as Quotient, Debt, etc.). Furthermore, the query is not trivial and it is plausible that even experienced SQL developers spend 20-30 minutes on the solution. And ultimately, the final query is based on aggregate functions (sums and averages). The underlying banking schema consists of five tables with rather trivial fields.

Figure 6 illustrates a possible solution using WITH. Although the task description consists of three steps, we think it is plausible to add some more steps to the solution, because, for example, the actual payment of customers can be more easily understood if defined in a separate variable. Figure 7 describes a different, possible solution using SQL$_{Assign}$. Actually, there are no major differences between both solutions: in the end, it is just a question where the variable

[4]Such approach was already applied in the first steps of experimentation in programming such as the experiment described by Lucas and Kaplan from 1976 (Jr. and Kaplan, 1976) and is up to today often applied.

[5]The original task description was in German – we translated the text to English in order to ease the readability of the present paper.

*Return all customers who are uneconomical from the point of view of a particular insurance agency.*
*Output: CustomerID, DegreeOfUneconomicalness.*
*Consider three variables:*

*Quotient: The fee defined for a contract added to the surcharge received by a customer divided by the fee of the contract. In this case, the quotient of a customer must be greater than the average quotient of all customers, otherwise the customer is economical.*

*Debt: The sum of the not yet paid requested money of a customer. In this case, the debt of a customer must be greater than his quotient multiplied by the sum of his paid requested money, otherwise the customer is economical.*

*Loss: The sum of the costs for all incidents of a customer. In this case, the loss of a customer must be greater than his debt added to the sum of the values of his securities, otherwise the customer is economical.*

*This insurance agency wants to terminate a customer, if his DegreeOfUneconomicalness fulfills (quotient * debt + loss) > 150,000.*

Figure 5: Task description of Task 1. The actual task description used in the experiment was not in English but in the language of the university where the experiment was performed.

names appear and how the end of an assignment looks like (semicolon versus brackets).

The experiment was defined as a crossover trial that potentially suffer from carry–over effects (see (Kitchenham et al., 2003; Hanenberg and Mehlhorn, 2021)). As a consequence, our goal was to define a second task that is quite close to the first task in terms of required steps, but where the computations of the different steps are different. We came up with a task from the field of formula one, where drivers should be selected. For reasons of completeness, Figure 8 illustrates the task description.

## 4.3 Measurement

We implemented a (small) tool for entering SQL statements. The tool used a Postgres database server version 11. Additionally, the tool showed the textual task description as well as the available tables in a tree view. We started the measurement once a participant articulated that he was willing to start with the task. The time measurement stopped once the result of a query matched the expected outcome. The measurement was done in seconds. Part of the measurement was all time spent on the task including potential searches on the Internet. After 80 minutes, the tool showed that the participant was permitted to go to the next task (respectively to stop the experiment).

We gave a short description of SQL$_{Assign}$ before the experiment and gave a simple task that should require 10 to 15 minutes. Finally, we screen recorded the sessions to have later on the opportunity to view again solution steps. Additionally, we observed the participants during the sessions.

## 4.4 Experiment Layout

The experiment had the following dependent and independent variables:

**Dependent Variable:** Time to completion (the time required by developers to define a requested query).
**Independent Variable:** Language (with the treatments SQL and SQL$_{Assign}$), where language is a within-subject variable.
**Random Variable:** Group Assignment, where subjects were either assigned to the group standard SQL first or SQL$_{Assign}$ first.

We defined a time limit (see (Feitelson, 2021)) for each task of 80 minutes assuming that 80 minutes are enough to solve the task. In case a participant did not finish within 80 minutes, we entered 80 minutes as his result. Since we assumed that this would happen only in SQL and not in SQL$_{Assign}$ this is a penalty for SQL$_{Assign}$, because the subject would have probably required more time using standard SQL.

## 4.5 Experiment Execution

We executed the experiment on 24 volunteers (students) based on purposive sampling (Patton, 2014).. The requirements for the volunteers was that they passed a database course at our institute. Hence, we assumed that they were familiar with SQL and the WITH clause. Each volunteer was tested in an individual session starting with an introduction into SQL$_{Assign}$ and software used in the experiment. We explicitly articulated that Internet search was permitted (and recommended) in case a participant had questions concerning SQL. We explicitly articulated that the used SQL dialect was Postgres. We did not define a time limit for the introduction and answered there all questions a student possibly had.

## 5 RESULTS

Table 1 contains for each participant the measured times. One participant (No 4) ran into the time limit. As explained before, we accepted that the time limit was reached and still included him in the data set.

```
WITH Quotient AS ( SELECT CustomerId, (Contract.Amount + Surchange.Amount) / Contract.Amount AS Quotient
                   FROM Customer NATURAL JOIN Contract JOIN Surchange USING(CustomerId)),
     Demand AS (SELECT CustomerId, SUM(DemandedAmout) AS Demand FROM Claim GROUP BY CustomerId,
     Paid AS (SELECT CustomerId, SUM(Payment) AS Paid FROM Payment GROUP BY CustomerId),
     Loss AS (SELECT CustomerId, SUM(Cost) AS Loss FROM Incident GROUP BY CustomerId),
     Value AS (SELECT CustomerId, SUM(Value) as Value FROM Asset GROUP BY CustomerId)
SELECT CustomerID, Quotient * (Demand - Payment) + Loss AS DegreeOfUneconomicalness
FROM Quotient
       NATURAL JOIN Demand NATURAL JOIN Paid NATURAL JOIN Loss NATURAL JOIN Value
WHERE Quotient > (SELECT AVG(Quotient) FROM Quotient
      AND Demand - Paid > Quotient*Paid
      AND Loss > (Demand - Paid) + Value
      AND Quotient * (Demand - Paid) + Value > 150000;
```

Figure 6: Possible solution for Task 1 using standard SQL using WITH.

```
AllCustomers = SELECT CustomerId, (Contract.Amount + Surchange.Amount) / Contract.Amount AS Quotient
               FROM Customer NATURAL JOIN Contract JOIN Surchange USING(CustomerId));
Average = SELECT AVG(Quotient) FROM AllCustomers;
QuotientT = SELECT CustomerId, Quotient FROM AllCustomers WHERE Quotient > Average;
Demand = SELECT CustomerId, SUM(DemandedAmount) AS Demanded FROM Claim GROUP BY CustomerId;
Paid = SELECT CustomerId, Sum(PayedAmount) AS Paid FROM Payment GROUP BY CustomerID;
DebtT = SELECT CustomerId, Demanded - Paid AS Debt FROM Demand NATURAL JOINT Paid NATURAL JOIN QuotientT
          WHERE Demanded - Paid > Quotient * Paid;
CustomerLoss = SELECT CustomerId, SUM(Value) as Loss FROM Incident GROUP BY CustomerId;
CustomerValue = SELECT CustomerId, SUM(Value) as Value FROM Asset GROUP BY CustomerId;
LossT = SELECT CustomerId, Loss, FROM CustomerLoss NATURAL JOIN CustomerValue NATURAL JOIN DebtsT
         WHERE Loss > Debts + Value;

SELECT CustomerID, Quotient * Debt + Loss AS DegreeOfUneconomicalness
FROM QuotientT NATURAL JOIN DebtT NATURAL JOIN LossT
WHERE Quotient * Debt + Loss > 150000;
```

Figure 7: Possible Solution for Task 1 using SQL$_{Assign}$.

Table 1: Measurements: Times for Task1 and Task2 in seconds. Group SQL started with standard SQL and solved the second task with SQL$_{Assign}$, group SQL$_{Assign}$ vice versa.

| Group | No | Training | Task1 | Task2 | Sum (Task1+Task2) |
|---|---|---|---|---|---|
| SQL | 2 | 656 | 3905 | 1778 | 5683 |
| SQL | 4 | 918 | 4800 | 2634 | 7434 |
| SQL | 6 | 1330 | 3546 | 2425 | 5971 |
| SQL | 8 | 590 | 3324 | 2102 | 5426 |
| SQL | 10 | 655 | 2164 | 1298 | 3462 |
| SQL | 12 | 943 | 2702 | 1305 | 4007 |
| SQL | 14 | 827 | 3612 | 1757 | 5369 |
| SQL | 16 | 860 | 4052 | 1785 | 5837 |
| SQL | 18 | 502 | 3502 | 1333 | 4835 |
| SQL | 20 | 757 | 4239 | 1748 | 5987 |
| SQL | 22 | 1138 | 4613 | 2077 | 6690 |
| SQL | 24 | 1314 | 3036 | 1442 | 4478 |
| SQL$_{Assign}$ | 1 | 873 | 2314 | 4183 | 6497 |
| SQL$_{Assign}$ | 3 | 671 | 1498 | 3465 | 4963 |
| SQL$_{Assign}$ | 5 | 730 | 2259 | 3143 | 5402 |
| SQL$_{Assign}$ | 7 | 812 | 2058 | 4269 | 6327 |
| SQL$_{Assign}$ | 9 | 645 | 1652 | 4273 | 5925 |
| SQL$_{Assign}$ | 11 | 508 | 1959 | 2916 | 4875 |
| SQL$_{Assign}$ | 13 | 748 | 1822 | 3593 | 5415 |
| SQL$_{Assign}$ | 15 | 604 | 1567 | 2081 | 3648 |
| SQL$_{Assign}$ | 17 | 648 | 1866 | 3674 | 5540 |
| SQL$_{Assign}$ | 19 | 653 | 1657 | 3316 | 4973 |
| SQL$_{Assign}$ | 21 | 777 | 1585 | 3014 | 4599 |
| SQL$_{Assign}$ | 23 | 653 | 1752 | 1633 | 3385 |

The first impression of the data is that times below 2000 seconds appear most often in task 2 for group SQL (where task 2 was solved with SQL$_{Assign}$) while the same phenomenon appeared mainly in task 1 for group SQL$_{Assign}$. One participant (No. 23) solved the task faster with SQL than with SQL$_{Assign}$.

## 5.1 Informal Observations

Since each participant was tested in an individual session, we were able to observe how they solved the tasks. This observation was informally done.

All participants except participant 4 were able to solve the tasks in time. The experimenter had the impression that participant 4 (who was not able to finish the task using standard SQL) was overwhelmed by the difficulty of the task (which was not the case for the task solved in SQL$_{Assign}$). Actually, one requirement of the different steps got lost in the participant's query and the participant was subsequently not able to detect this missing requirement in time.

No single participant used the WITH clause – neither in the group starting with standard SQL nor in the group starting with SQL$_{Assign}$. For almost all sub-

*Return all racers who, from the point of view of a particular F1 team, are considered worthy to be recruited as drivers.*
*Output: DriverID, LevelOfAttractiveness.*

*Consider three variables:*

*Significance: The points of a driver's team added to the points of the driver divided by the points of his team. In this case, the significance of a driver must be greater than the average significance of all drivers, otherwise the driver is unattractive.*

*Difference: The difference between the sum of the expected positions by analysts and the sum of the actual positions won by a driver. In this case, the driver's difference must be greater than the sum of his expected positions divided by his significance, otherwise the driver is unattractive.*

*PodiumPoints: The accumulated points by podium positions of a driver. In this case, the driver's podium points added with his difference must be more than the sum of the costs of the driver due to racing incidents / 1,000,000, otherwise the driver is unattractive.*

*This team is interested in a driver, if his LevelOfAttractiveness fulfills (significance * difference + podiumPoints) > 500.*

Figure 8: Task description of Task 2. The actual task description used in the experiment was not in English but in the language of the university where the experiment was performed.

jects using standard SQL the results were comparable: the different steps were formulated as individual queries and then either the resulting query was completely copied into a subquery or the conditions for each step were copied into the WHERE clause (and the tables were then joined).

Another observation was that some participants had problems when defining subqueries where either the time consuming problem was to give subqueries names or participants where confused what parts of a subquery should end up in the FROM or the WHERE clause, leading to problems in fixing errors caused by copying a query as a subquery (subject 1, 2, 3, 5, 7, 9, 16, 17, 19, 21, 22, 24). However, we also observed that for a number of subjects (subject 6, 10, 11, 12, 14, 15) the pure observation of the way how they solved the tasks did not reveal anything noticeable. However, for all these subjects, it still turned out that the use of SQL$_{Assign}$ required less time.

## 5.2 Quantitative Results

Before analyzing the results we check whether a carry-over effect could be detected. Thereto, we run a one-way ANOVA on the sum of times (dependent variable)[6] with the independent variable group (with treatments group 1 and group 2, see Table 2). For the sum of times for both tasks, the resulting ANOVA does not indicate a carry-over effect (F(1, 22)=.514, p=.481, $\eta_p^2$=.023), i.e. it is possible to consider the times in a combined analysis.

In order to study the effect of the language, we run a repeated measures ANOVA (see Table 2) with the within-subject variable language (with the treatments SQL and SQL$_{Assign}$) and the between subject variable group (with the treatments group 1 and group 2).[7] The difference between both groups is significant and the difference is almost factor 2 ($\frac{M(SQL)}{M(SQL_{Assign})} = \frac{3{,}460.25}{1{,}819.71} = 1.90$): Solving the task using SQL$_{Assign}$ required only 52% of the time required for standard SQL.

However, one subject (who started with SQL$_{Assign}$) required less time using SQL. We cannot exclude that this is the result of an individual learning effect. Or it could just mean that there are still single subjects to whom the introduction of the assignment operation is counterproductive. Or it could be just the result of luck.

It is noteworthy that the differences between standard SQL and SQL$_{Assign}$ are quite large. As a consequence, it is questionable, whether our rather cautious design using a crossover experiment was already larger than necessary. Consequently, we ask ourselves, whether the effect would have already shown up when the second round (i.e. where the crossover appeared) would not be considered.

In order to test this, we run a simple, unpaired t-test on the first round and get a similar result as the previously reported one. Again, the variable language is significant (t(13.676)=7.655; $p < .001$; M$_{SQL}$=3,624.58; M$_{SQL_{Assign}}$=1,832.42; $\frac{M_{SQL_{Assign}}}{M_{SQL}}$=50,56%)[8] and again SQL$_{Assign}$ required only approximately 50% of the time required by standard SQL.

## 6 THREATS TO VALIDITY

We see a obvious threats with respect to the generalizability of the results. First, we explicitly assume that a language construct as the proposed one has probably only a positive effect if the requested query is defined

[6]The analysis was performed using SPSS v27.

[7]Taking the result of the previous test into account, there is no need to use group as an independent variable any longer – we still do it here for reasons of completeness.

[8]The degrees of freedom were adapted, because the Levene test turned out significant (p<.001):

Table 2: Repeated measures ANOVA on the within-subject variable Language (L) and the between-subject variable Group (G). Confidence intervals (CI), means (M), and standard deviation (SD) are given in seconds.

| Variable | df | F | p | $\eta_p^2$ | Treatment | N | $\mathrm{CI}_{95\%}$ | M | SD |
|---|---|---|---|---|---|---|---|---|---|
| Language (L) | 1 | 141.416 | <.001 | .865 | SQL | 24 | [3,124.41; 3,796,84] | 3,460.625 | 794.61 |
| | | | | | $\mathrm{SQL}_{Assign}$ | 24 | [1,666.08; 1973.34] | 1,819.71 | 355.17 |
| Group (G) | 1 | .514 | .481 | .023 | SQL Start | 24 | [2,461.78; 2,969.80] | 2,715.79 | 1111.85 |
| | | | | | $\mathrm{SQL}_{Assign}$ Start | 24 | [2,310.53; 2818.55] | 2,564.54 | 948.57 |
| L * G | 1 | .982 | .327 | .022 | SQL \| SQL Start | 12 | [3,265.36; 3,983.81] | 3,624.58 | 765.15 |
| | | | | | SQL \| $\mathrm{SQL}_{Assign}$ Start | 12 | [2,937.44; 3,655.89] | 3,296.67 | 822.26 |
| | | | | | $\mathrm{SQL}_{Assign}$ \| SQL Start | 12 | [1,447.78; 2,166.22] | 1,807.00 | 437.15 |
| | | | | | $\mathrm{SQL}_{Assign}$ \| $\mathrm{SQL}_{Assign}$ Start | 12 | [1,473,19; 2,191.64] | 1,832.42 | 268.89 |

in a rather step by step manner. In other words, if a more imperative definition of a problem is given, we think a more imperative way to solve such problem is more appropriate. However, to what extent common SQL problems are rather formulated in an imperative or rather in terms of a declarative problem description is unclear to us. Second, we are aware that the query used in the experiment is rather difficult – at least, it is not a trivial SELECT–FROM–WHERE statement. However, to what extent this query is actually difficult and to what extent queries of such difficulty can be found in the real world is unclear to us. Our personal perception (having some background in writing SQL queries) is that real world queries easily have the difficulty of the given tasks. Third, it is unclear whether the more general claim – adding an imperative construct to a declarative language – actually holds to other languages as well. The here chosen language SQL is just one example of a declarative language and we do not know whether the effect is special to SQL.

In addition to the external threats, we see internal threats with respect to the chosen participants and their background. As discussed, it turned out that no single participant used SQL's WITH clause. We cannot exclude that this is just the results of the participants' background (since each participant was from the same institute): we cannot exclude that it is an institute-dependent characteristic of students that the construct WITH is rather not appreciated.

We see a more serious internal threat. It is commonly accepted to introduce a new language construct to participants in order to test the language construct in an experimental way. But actually, just the introduction of the construct could already introduce a bias: subjects could feel the need to use this language feature. Consequently, it is possible that the urge to use the assignments in the queries (when the experiment gives the opportunity to do so) is not the result of the language feature itself. And it is possible that the absence of the WITH clause in solutions delivered by subjects is no indicator for the inferiority of the WITH clause. Actually, we think that this is a serious threat that exists in general in experiments where new language features are tested (and introduced throughout the experiment). However, one also has to take into account that the test only on the first round (the last test in Section 5.2) also revealed a positive effect of the variable language. This reduces the potential threat that the experiment result was mainly driven by the additional teaching of the construct, because in the first round this is not relevant for the control group.

## 7 SUMMARY AND DISCUSSION

In the present study, we tested whether the introduction of the imperative language construct variable assignments (in addition to sequences) would provide a benefit to SQL developers. With this extension (that we called $\mathrm{SQL}_{Assign}$) we designed a crossover experiment where 24 students had to solve two queries. The task descriptions of both queries were given in multiple steps. The experiment revealed that the time to solve a task using $\mathrm{SQL}_{Assign}$ just required 52% of the time in comparison to standard SQL. And while 23 participants solved a task faster with $\mathrm{SQL}_{Assign}$, one participant experienced the opposit.

What makes the results of the experiment from our point of view surprising is, that in principle no such language construct is required. First, there is already a language construct such as WITH available in SQL that permits to define a named term as the result of an SQL expression – which is actually exactly what the proposed language construct does. Second, such terms and term replacements are not really required in SQL, because subqueries already provide something similar. However, while all participants used subqueries in standard SQL, not a single participant felt the need (or saw any reason) to use WITH.

Actually, we see two interpretations from the experiment in the context of SQL. First, it is possible that participants underestimate the difficulty of SQL while defining difficult queries. Once one query is done, participants think that just adding an additional query on top of it does not do any harm. While the proposed language construct is quite similar to the WITH clause, we think that it still gave participants the impression that it is easier to use than a WITH clause – probably, because the proposed syntax gave participants more the impression that the solution is executed statement by statement.

Of course, it could also be the case that the value of the WITH clause is just underestimated by students – and it could rather be interpreted as a critique of the students' education in SQL, where they are used to write difficult queries, but where they are maybe not well–trained in the reduction of a query's difficulty.

However, this leads to an even more general question. Actually, inner queries provide similar to the WITH clause an additional abstraction, because both permit to give a name to a query. It is possible that just the position of this name in a query is the result why it is not considered as a relief but rather as a burden – and it could be the case that the syntactical representation of the proposed assignments (where the variable names appear in the beginning of each line) is considered as a better and more explicit representation for such named queries.

Actually, the last argument seems to match to some extent the mentioned related work that indicates that SQL's syntax seems to be hard to understand (see (Ahadi et al., 2016; Poulsen et al., 2020)) which leads to the situation that most errors in SQL are syntax errors. On a more abstract level, this could be a hint that language designers should spend more effort into the design of syntax structures or even into the design keywords – which is in line with studies such as the one by Stefik and Siebert that revealed that even the choice of keywords in languages can make a large difference (Stefik and Siebert, 2013).

And on an even more abstract level, the designed experiment raises the question, whether declarative languages could benefit from the introduction of imperative language constructs. While today mainly the tendency in programming language design can be found that declarative language features appear in imperative language (such as the introduction of lambda expressions in main stream imperative languages), it is not often the case that people ask whether declarative languages could benefit from imperative features as well. However, we mentioned already in the related work section the study by Pichler et al. that showed that students have some tendency toward imperative language constructs (Pichler et al., 2011).

Our general conclusion with regard to this discussion is rather that the whole discussion about whether a language should be declarative or imperative is slightly misleading: there are works that have given evidence that the introduction of declarative features in imperative languages causes some benefit (see for example (Mehlhorn and Hanenberg, 2022)) while the present paper also gives evidence that a declarative language could benefit from imperative features. Probably, it is worth to think about whether the discussion should rather go into the direction of declarative versus imperative task descriptions – and the resulting question, what the effect of language constructs on the solution of such tasks actually is.

Altogether, we think that the present paper gives some food for thought for the design of languages. While language designers have the tendency to think a lot about language semantics, it might be the situation that rather simple syntactical differences in the representation of a language make a huge difference. And maybe the present study motives language designers to put additional effort into studying language features using empirical methods.

## 8 CONCLUSION

The here presented study gives evidence that the introduction of a quasi-imperative language construct in terms of a variable assignment reduces the time required to define a difficult query that was given to the experiment's participants in terms of a step by step description: the introduction of that language feature reduced the time effort for the definition of a query by about 48%.

While we see in the given experiment an indicator that SQL – which is today still the leading language for accessing databases – can be improved by relatively simple syntactical elements, we also see in the study a contribution to the discussion whether languages should be declarative or imperative: while in the literatures more and more evidence is gathered that imperative languages could benefit from declarative features, the present study gives first indicators that it is possible also the other way around.

Altogether, we think that the present study should motivate language designers to test their languages not only in terms of functionality, but also in terms of usability. Because it might turn out that even small syntactic differences have larger effects. And the study could also motivate language designers to think about improving existing languages. Again, not only in terms of functionality, but also in terms of usability – which could be just simple syntactical changes. And the evidence for the usability of a language constructs should not be based on plausibility but on measurements – and from our perspective the use of randomized control trials should be the first choice.

## REFERENCES

Ahadi, A., Behbood, V., Vihavainen, A., Prior, J., and Lister, R. (2016). Students' syntactic mistakes in writing seven different types of sql queries and its application to predicting students' success. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, SIGCSE '16, pages 401–406, New York, NY, USA. Association for Computing Machinery.

Feitelson, D. G. (2021). Considerations and pitfalls in controlled experiments on code comprehension. In *29th IEEE/ACM International Conference on Program Comprehension, ICPC 2021, Madrid, Spain, May 20-21, 2021*, pages 106–117. IEEE.

Hanenberg, S. (2010). An experiment about static and dynamic type systems: Doubts about the positive impact of static type systems on development time. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 22–35, New York, NY, USA. Association for Computing Machinery.

Hanenberg, S. and Mehlhorn, N. (2021). Two n-of-1 self-trials on readability differences between anonymous inner classes (aics) and lambda expressions (les) on java code snippets. *Empirical Software Engineering*, 27(2):33.

Jr., H. C. L. and Kaplan, R. B. (1976). A structured programming experiment. *Comput. J.*, 19(2):136–138.

Kaijanaho, A.-J. (2015). *Evidence-based programming language design: a philosophical and methodological exploration*. University of Jyväskylä, Finnland.

Kitchenham, B., Fry, J., and Linkman, S. (2003). The case against cross-over designs in software engineering. In *Eleventh Annual International Workshop on Software Technology and Engineering Practice*, pages 65–67.

Kokuryo, S., Kondo, M., and Mizuno, O. (2020). An empirical study of utilization of imperative modules in ansible. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*, pages 442–449.

Lucas, W., Bonifácio, R., Canedo, E. D., Marcílio, D., and Lima, F. (2019). Does the introduction of lambda expressions improve the comprehension of java programs? In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*, SBES 2019, pages 187–196, New York, NY, USA. Association for Computing Machinery.

Mehlhorn, N. and Hanenberg, S. (2022). Imperative versus declarative collection processing: An rct on the understandability of traditional loops versus the stream api in java. In *Proceedings of the 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 22–27, 2022, to appear*.

Müller, M. (2005). Two controlled experiments concerning the comparison of pair programming to peer review. *Journal of Systems and Software*, 78:166–179.

Patton, M. (2014). *Qualitative Research & Evaluation Methods: Integrating Theory and Practice*. SAGE Publications.

Pichler, P., Weber, B., Zugal, S., Pinggera, J., Mendling, J., and Reijers, H. A. (2011). Imperative versus declarative process modeling languages: An empirical investigation. In Daniel, F., Barkaoui, K., and Dustdar, S., editors, *Business Process Management Workshops - BPM 2011 International Workshops, Clermont-Ferrand, France, August 29, 2011, Revised Selected Papers, Part I*, volume 99 of *Lecture Notes in Business Information Processing*, pages 383–394. Springer.

Poulsen, S., Butler, L., Alawini, A., and Herman, G. L. (2020). Insights from student solutions to sql homework problems. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '20, pages 404–410, New York, NY, USA. Association for Computing Machinery.

Senn, S. (2002). *Cross-over Trials in Clinical Research*. Statistics in Practice. Wiley.

Stefik, A. and Siebert, S. (2013). An empirical investigation into programming language syntax. *ACM Trans. Comput. Educ.*, 13(4).

Uesbeck, P. M., Stefik, A., Hanenberg, S., Pedersen, J., and Daleiden, P. (2016). An empirical study on the impact of c++ lambdas and programmer experience. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 760–771.