Procedural Extensions of SQL: Understanding their usage in the wild

Surabhi Gupta
Microsoft Research India
t-sugu@microsoft.com

Karthik Ramachandra
Microsoft Azure Data (SQL), India
karam@microsoft.com

ABSTRACT

Procedural extensions of SQL have been in existence for many decades now. However, little is known about their magnitude of usage and their complexity in real-world workloads. Procedural code executing in a RDBMS is known to have inefficiencies and limitations; as a result there have been several efforts to address this problem. However, the lack of understanding of their use in real workloads makes it challenging to (a) motivate new work in this area, (b) identify research challenges and opportunities, and (c) demonstrate impact of novel work. We aim to address these challenges with our work.

In this paper, we present the results of our in-depth analysis of thousands of stored procedures, user-defined functions and triggers taken from several real workloads. We introduce *SQL-ProcBench*, a benchmark for procedural workloads in RDBMSs. SQL-ProcBench has been created using the insights derived from our analysis, and thus represents real workloads. Using SQL-ProcBench, we present an experimental evaluation on several database engines to understand and identify research challenges and opportunities. We emphasize the need to work on these interesting and relevant problems, and encourage researchers to contribute to this area.

PVLDB Reference Format:
Surabhi Gupta and Karthik Ramachandra. Procedural Extensions of SQL: Understanding their usage in the wild. PVLDB, 14(8): 1378 - 1391, 2021.
doi:10.14778/3457390.3457402

1 INTRODUCTION

Declarative SQL is the de-facto standard for querying relational data, and it has proven its relevance time and again. Optimization and efficient evaluation of SQL queries has been one of the important focus areas of the DB community over the years. While this has resulted in quite mature and sophisticated techniques, it still remains an active area of research.

In addition to the declarative SQL interface, most RDBMSs provide extensions that allow users to write programs using procedural extensions to SQL [7, 10, 20]. They usually appear in the form of Stored Procedures (SPs), User Defined Functions (UDFs) and Triggers. Many RDBMSs also allow procedures to be written in other languages such as Java, C++, C# etc. These procedural extensions augment declarative SQL with imperative constructs and enable the procedural programming style when interacting with RDBMSs. Such extensions offer several benefits when used in conjunction with declarative SQL, such as code reusability, modularity, readability and maintainability.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 14, No. 8 ISSN 2150-8097.
doi:10.14778/3457390.3457402

1.1 Motivation

While procedural extensions of SQL have existed for many years, they are known to be inefficient, especially when operating on large datasets. The poor performance of procedural extensions in many RDBMSs has been observed over the years by practitioners and application developers. They have realized that the benefits of such procedural extensions come with a rather huge performance cost [6, 16]. Therefore, they have had to make a hard choice between performance on one side, and benefits such as modularity and reusability on the other. In addition to the performance overheads, procedural extensions of SQL also lack many features that are usually expected in programming languages. Over the years, the language surface has remained largely stagnant and has not evolved much. Versioning, debugging support, unit testing support, diagnostics are some key tools that are mostly not offered, or not as sophisticated as necessary. As a result, practitioners have to rely upon third party tools or build their own.

Although the poor performance of procedural SQL extensions is well-known, their optimization and efficient evaluation has not received enough attention. One reason is perhaps that most of the standard database benchmarks such as TPC-H [21], TPC-DS [18], SSB [46] and JOB [43] do not include UDFs or stored procedures. TPC-C and TPC-E [19] contain some stored procedures; but from our study, we find that it is an under-representation of both the number and complexity of procedures in real workloads. Real world workloads are typically far more complex, with a mix of SQL queries, views, stored procedures, UDFs, triggers etc. Another important reason for the lack of attention in this area is that very little is known about their usage patterns and the magnitude of their use in real-world applications.

To substantiate this and to gain more insights into the procedural complexity of real-world workloads, we conducted a detailed analysis which we present in this paper. Procedural extensions to SQL have several unique advantages that make them an attractive choice for many application developers. Therefore, despite all their inefficiencies and limitations, our study indicates that procedural extensions are quite widely used. Furthermore, there are several scenarios where users prefer to write procedural code in conjunction with declarative SQL.

Based on the increasing complexity of database applications and the feedback received from practitioners, this area has now started receiving more attention. Many of the efforts to address this problem have explored the use of optimizing compilers and code generators [4, 28, 40–42, 44, 45, 54, 55, 57]. Some other works

have explored techniques such as program analysis and program synthesis to translate procedural code into equivalent declarative SQL [29, 30, 36, 39, 48, 52]. These are discussed in Section 2.2.

A common hurdle that all these efforts face is the lack of understanding around the magnitude of use of procedural constructs in real-world enterprise applications. This often makes it hard to motivate and appreciate the potential impact of work in this area. Another important challenge that researchers face is the lack of clarity around the usage patterns and complexity of code written by application developers 'in the wild'. Furthermore, many of these procedural extensions used in enterprises are proprietary in nature. This makes it hard to identify specific research opportunities and focus efforts where it really matters. Our present work is motivated by these hurdles and is an attempt to address them.

1.2 Contributions

We believe that the above challenges hinder innovation in this interesting and practically important area. To address this gap, we make the following contributions in this paper.

(1) We conduct an in-depth analysis of more than 6500 procedures across many real-world applications, and derive insights about the kinds of procedures found in the wild. We present the methods and results of this analysis in Section 3.
(2) We introduce SQL-ProcBench (Section 4), a benchmark for procedural workloads in RDBMSs, which has been developed using the insights derived from our analysis. We ensure that SQL-ProcBench represents real-world workloads while preserving the confidentiality and privacy of the proprietary code used for our analysis. To the best of our knowledge, SQL-ProcBench is the first benchmark that focuses specifically on procedural workloads in RDBMSs.
(3) Using SQL-ProcBench, we present an experimental evaluation on 4 database engines with different configurations (Section 5).
(4) Based on our analysis and experiments, we identify key research challenges and opportunities to encourage the community to contribute to this line of work (Section 6).

2 BACKGROUND

Procedural extensions to SQL have been in existence for almost 3 decades [12]. Oracle's PL/SQL was first released in 1991 basing the language on the Ada programming language. Procedural extensions formally became an ISO standard named SQL/PSM (SQL/Persistent Stored Modules) in 1996 as an extension to SQL92. Since then, it has been part of the SQL:1999 standard, including SQL:2016 [12].

2.1 Procedural extensions and their evaluation

Broadly, procedural extensions are of 3 kinds: Stored Procedures (SPs), User-Defined Functions (UDFs) and triggers. We now describe each of these, and briefly explain how they have been evaluated traditionally. Note that we only provide a conceptual explanation of the traditional evaluation techniques. Each database engine might vary in the details and optimizations that may be performed.

2.1.1 User-Defined Functions (UDFs). User-defined functions provide a mechanism for users to add a function that can be evaluated as part of SQL statements. The SQL standard distinguishes between scalar and table functions. A scalar function returns only a single value (or NULL), whereas a table-valued function returns a table comprising zero or more rows, each row with one or more columns. Once created, a user-defined function may be used in expressions in SQL statements. For example, it can be invoked from within the various clauses of SQL statements such as the SELECT/FROM/WHERE/GROUP BY/HAVING, etc. These functions can also access data stored in tables in the database using embedded SQL queries. Note that UDFs are not allowed to make changes to the persistent state of the database.

The traditional evaluation technique used in most disk-based RDBMSs for UDFs is explained in [48]. In summary, the query optimizer does not look inside the procedural component during the optimization of the query that invokes the UDF. During execution of the query, the UDF is evaluated once per qualifying row. During this evaluation, the body of the UDF is treated as a batch of statements, and this batch is interpreted and executed statement-by-statement, sequentially. Each statement might be compiled and optimized during the first execution of the UDF in some systems, but it is still iterative and sequential in most cases.

2.1.2 Stored Procedures. Stored procedures are similar to UDFs with few key differences. One difference is that UDFs can be used like any other expression within SQL statements, whereas stored procedures must be invoked using the CALL or EXECUTE statement. UDFs could be used in SELECT statements, but stored procedures cannot. A stored procedure can return multiple values using the OUT parameter, or return no value. Stored procedures can contain transactions, and can therefore mutate the persistent state of the database, unlike UDFs. The execution of stored procedures is similar to UDF evaluation explained earlier.

Usually, we find that stored procedures are used to implement application logic that includes updating the database state, whereas UDFs are primarily used for data retrieval and computation-oriented tasks. As we show in this paper, stored procedures are usually found to be more complex than UDFs because they allow a broad set of operations to be incorporated in them.

2.1.3 Triggers. A database trigger is a procedure that is automatically executed in response to certain events on a particular table or view in a database. Triggers are usually used to maintain the integrity of the information on the database. Triggers can also be used to log historical data or perform related operations. Triggers can be associated at different granularities such as schema-level, row-level, column-level, etc. These offerings vary between different database engines; more details can be found here [2]. Traditionally, triggers are treated very similar to stored procedures, and hence are compiled and evaluated the same way as procedures.

2.1.4 Foreign language interfaces. In addition to procedural SQL extensions, most RDBMSs allow procedures to be written in other languages such as Java, C++, C# etc. These foreign language interfaces provide some advantages w.r.t. tooling and IDE support. But they come with their own set of limitations and pitfalls. Debugging support, diagnostics and troubleshooting are still a challenge. Furthermore, these run in a sandboxed environment with restrictions on the kinds of operations allowed. They are mostly suitable

for compute-only operations which purely operate on their input parameters. Accessing persistent data requires opening new connections to the DBMS, like a client application. This not only affects performance, but also complicates transaction semantics and could have potential security risks. This paper focuses on procedural extensions to SQL as defined in the SQL/PSM ISO standard, as it represents the full scope of possible operations in such procedures.

2.2 Efficient Evaluation of Procedural code

We now briefly describe known efforts to efficiently evaluate procedural extensions in RDBMSs. At the outset, we note that this is not an exhaustive literature survey, but is meant only as a glimpse into this area of research.

2.2.1 Compilation to assembly/machine code: These approaches aim to take procedural SQL as input and efficiently compile them into assembly or machine code directly. Many of these efforts have explored compiling procedures into native machine code using a compiler backend such as LLVM [17] while performing several optimizations along the way [34, 40–42, 44, 45, 49, 50, 54–57]. This kind of compilation breaks the volcano-style iterator model of query execution and instead results in highly efficient code. It leverages the large body of modern compiler optimization techniques, architecture-specific instructions and optimizations such as vectorization and SIMD as part of this compilation process. These techniques are more common in main-memory database engines. Some real-world implementations include Hyper [45], SQL Server Hekaton [28], Oracle [8], MemSQL [4], etc.

2.2.2 Translation of procedural SQL to declarative SQL:. An alternative that has emerged in the last few years is the technique of transforming procedural code into set-oriented declarative SQL or relational algebra. Using program-rewriting techniques to achieve this goal was initally proposed in [36, 37]. A technique to decorrelate UDF invocations was first given by Simhadri et. al. [52]. This has then been followed and extended by many others [11, 29, 30, 35, 39, 48]. Other related works include [26, 33, 47]. All these approaches are based on the principle that database engines have over the years built mature and sophisticated techniques to optimize complex SQL queries. Therefore, if procedural code can be expressed in a declarative form that database query optimizers can consume, existing relational optimization techniques can be leveraged to come up with an efficient query plan for procedures.

There have also been techniques proposed for optimizing procedures written in non-SQL languages. Related to this, we note that many developers also choose to implement application logic on the client, interacting with the database using libraries/APIs such as JDBC [22]. Similar limitations exist in this scenario, and therefore SQL translation techniques using program transformation and program synthesis have been considered here as well [26, 31, 32].

2.2.3 Other approaches. Database engines have used several different techniques to improve performance of procedures. For instance, stored procedures can often be annotated with properties such as such as *deterministic*, *side-effect-free*, or *order-independent* that enable certain optimizations like parallel execution. Some databases employ techniques such as sub-program inlining [15], function result caching [9] and interleaved execution [3]. Recent techniques have also tried to build specialized indexes or statistics [38, 51] for UDFs and procedures to improve performance in specific scenarios.

Table 1: Chosen procedural workloads

Workload	Description	Object Count
W1	Configuration Management System	3770
W2	Incidents & issue management software	845
W3	Document & data management tool	625
W4	Ecosystem inventory	575
W5	Retail platform	345
W6	CRM application	296
W7	Supply chain management tool	226
Total		6682

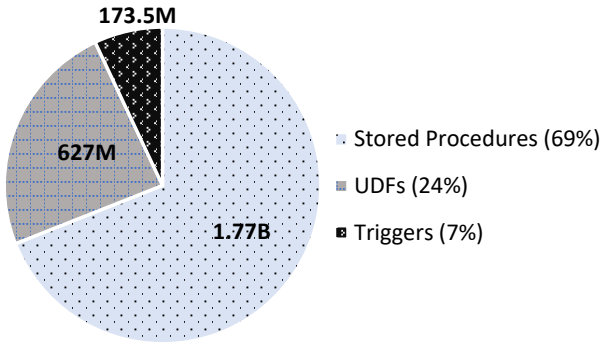


Figure 1: Object-type distribution in Azure SQL Database

3 PROCEDURAL SQL WORKLOAD ANALYSIS

Procedural extensions of SQL are quite widely used in the wild. Figure 1 gives a distribution of the number of stored procedures, UDFs and triggers that are in use in the Microsoft Azure SQL Database Service [5]. Note that these procedures result in multiple billions of daily invocations; Figure 1 only shows the number of defined procedures. This gives an indication of the magnitude of usage of these procedural objects in real-world workloads.

We have conducted a deeper analysis on a set of real workloads collected from various sources. In this section, we first describe the workloads chosen for the analysis and then present the methods and key insights we gained from this analysis. We also detail out the distributions of various statistically interesting properties collected on these workloads.

3.1 Choice of Workloads

We have considered 6682 objects from 7 diverse, proprietary real workloads (under NDA), belonging to three different categories: stored procedures, UDFs and triggers. A brief description about the nature of these workloads and the number of objects collected from each is shown in Table 1. These workloads include a mix of operational, analytical and hybrid applications across different industry verticals. Since we analyzed *all* the objects that we could gain access to from these 7 randomly chosen workloads, we believe that our selection process did not introduce biases.

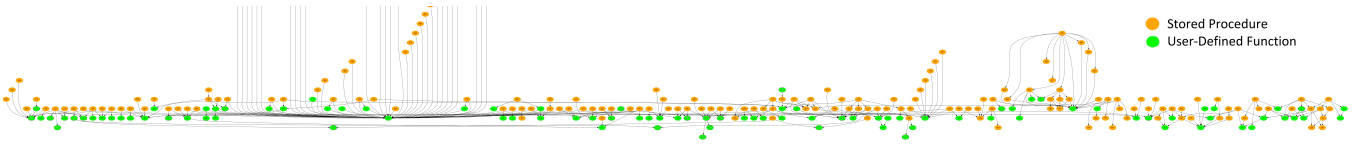


Figure 2: Object dependency subgraph from a real procedural SQL workload

These procedures vary greatly in their complexity and inter-object dependencies. For instance Figure 2 shows a subgraph of the object dependency graph from one of the workloads we have considered. Each node in this graph represents an object – orange represents a stored procedure and green represents a UDF. An edge between two objects x and y means that x invokes y. Note that this subgraph does not include triggers. We emphasize that Figure 2 is shown to give an idea of the complex inter-object dependencies that exist in the wild.

3.2 Feature Extraction

To gain more understanding of the procedural code present in real-world workloads, we have performed a quantitative analysis of several characteristic properties of these objects. We identified a set of 38 features or properties that provide insights into the nature of these procedures and help understand their usage. These 38 features have been categorized into 10 buckets or categories. The categories and the features are listed in Table 2.

These features were extracted using static analysis of the definitions of these objects i.e., the source code of these procedures, UDFs and triggers. We parsed these object definitions, performed type-derivation and built necessary data structures such as the abstract syntax tree and control flow graph. We also captured inter-object dependencies. Using these data structures, we computed the frequency distributions and other complexity metrics of all the features listed in Table 2.

3.3 Results

We now present some key insights and takeaways from this analysis based on different feature categories.

3.3.1 Code Complexity. We analyze the objects on 4 metrics to capture code complexity. First is the number of statements in the body of a procedure. Second is the cyclomatic complexity which is a standard metric to measure code complexity [1]. It is used to quantify the number of linearly independent paths through the object's source code. The presence of control flow altering statements increases the cyclomatic complexity. In addition to these two metrics, we capture the maximum depth of nested *if*-blocks and nested *loop*-blocks.

Figures 3(a) through (c) show the frequency distributions of the number of statements inside stored procedures, UDFs and triggers respectively. The y-axes in all these plots show the percentage of objects having the statement count specified by the x-axis bins. For Figure 3(a), the x-axis bin-width is 4, which means that the first bar shows the number of procedures having statement count in the range [1,4], second in the range [5, 8] and so on. For the other two graphs, the bin width is 1. All these graphs are skewed towards the left, implying that a large fraction of these objects have a rather small statement count. We also observe that stored procedures are more complex than UDFs and triggers. This is also corroborated from the fact that unlike UDFs and triggers which have a higher concentration around one statement, stored procedures have a higher concentration around two statements (48% of the stored procedures in the first bin have 2 statements as opposed to (16-18)% for each of 1, 3 or 4 statements.)

Figure 4 shows the distribution of cyclomatic complexity in these object types. We see a large percentage of these objects have a cyclomatic complexity value less than or equal to 5, with stored procedures being relatively more complex. Table 3 shows the average, 90th percentile and maximum values for these features.

3.3.2 Statement and Query complexity. A distinguishing characteristic of procedural extensions of SQL w.r.t procedural code in general, is the presence of SQL queries and DML/DDL statements intermixed with procedural code. One way to quantify the intra-statement complexity for each statement in the object is by measuring the number of nodes in the syntax tree of every statement. For SQL queries, we capture the number of operators in the query, which acts as a decent indicator of the query complexity. SQL statements usually turn out to be more complex in comparison with procedural constructs according to this metric.

Figures 7(a) - (c) show the distribution of the average number of nodes per statement in stored procedures, UDFs and triggers respectively. The y-axes in all these plots show the number of objects having the average number of nodes per statement as specified by the x-axis bins. The bin-width for x-axes in all these plots is 5. The graphs show that in comparison to procedures and UDFs, triggers tend to have a higher statement complexity. This is also evident from the fact that roughly 56% stored procedures, 45% UDFs and 78% triggers have an average number of nodes per statement greater than 10 (see Table 3). This can be attributed to the fact that triggers almost always contain SQL statements (SELECT/INSERT/UPDATE/DELETE etc.). In comparison, we find that many stored procedures and UDFs are computation-oriented – without SQL statements in their definition. Note that since triggers are invoked in response to some event and are usually used for maintaining the integrity of information in the database, or log certain events, they generally contain more *insert*, *update*, *delete* statements than *select* statements. This can also be seen in the *Number of Select Statements* feature in Table 3 which shows triggers having lesser number of select statements than stored procedures and UDFs.

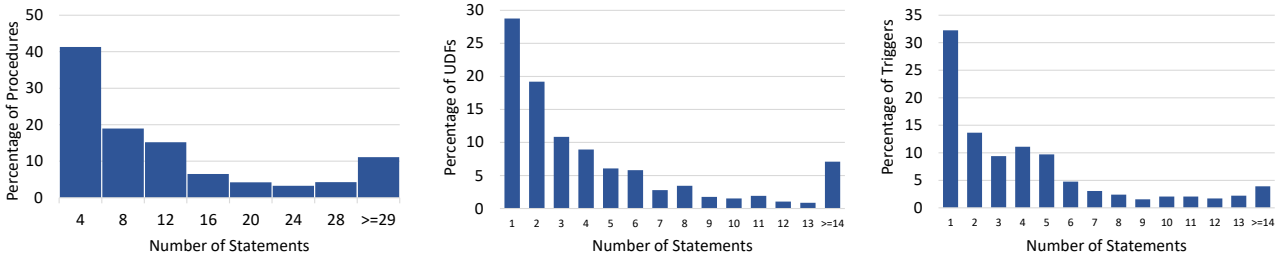
To quantify query complexity, we also analyzed the maximum length of join chains across all statements in the objects. Consider the following example to understand this metric: for the query '*SELECT * FROM T1, T2, T3*', the length of join chain is 3. We find the maximum value of the join chain length across all joins in the

Table 2: Properties collected from real-world workloads

Feature Category	Features	Count
Code complexity	Statement count, Cyclomatic complexity, Max. depth of nested *if-block* and nested *loop-block*	4
Statement & Query complexity	Maximum length of join chain, Average number of tree nodes per statement	2
Unconditional Control Flow constructs	Break, Continue, Goto, Return, Raise error	5
Conditional Control Flow constructs	If, Else, Case, While loop, Cursor loop	5
Sequential Imperative constructs	Set, Select with assignment, Print	3
Object dependencies	Table references, Table variable references, UDF calls, Intrinsic function calls	4
DML statements	Select, Insert, Update, Delete, Merge	5
DDL Statements	Create table, Create table variable, Create view	3
Transaction-related statements	Begin transaction, Commit, Abort, Execute	4
Parameter information	Parameter count, Table-valued parameters, User-defined type parameters	3
	Total number of features:	**38**

Table 3: Summary Statistics for key features

Feature	Stored Procedures			UDFs			Triggers		
	Avg.	90th Percentile	Max.	Avg.	90th Percentile	Max.	Avg.	90th Percentile	Max.
Number of Statements	13.09	30	343	5.12	11	74	4.47	10	62
Avg no. of Nodes Per Statement	19	40	394	18.94	34.76	946	23.58	48	318
Max. Length of Join Chain	2.1	4	25	2.19	4	40	2.15	3	9
Number of *IF* Statements	3.08	7	121	1.36	3	34	0.99	3	22
IF Nesting Depth	1.73	3	90	1.49	3	8	1.41	2	7
Number of Table References	6.47	14	343	2.70	7	162	6.79	14.6	64
Number of Set Statements	2.34	6	254	1.27	3	41	0.18	0	24
Number of Select Statements	2.23	5	65	0.65	2	16	0.36	1	5
Cyclomatic Complexity	4.85	11	146	2.87	6	182	2.11	4	26
Number of Intrinsic Function Calls	3.49	9	257	1.85	4	95	1.67	5	37



(a) Stored Procedures **(b) UDFs** **(c) Triggers**

Figure 3: Distribution of Number of Statements

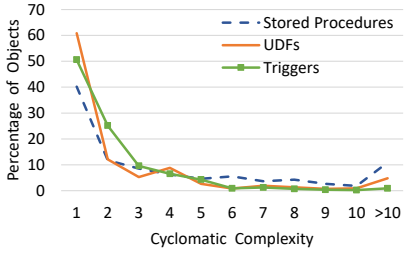


Figure 4: Distribution of Cyclomatic Complexity

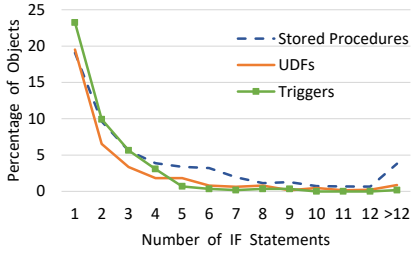


Figure 5: Distribution of *IF* statements

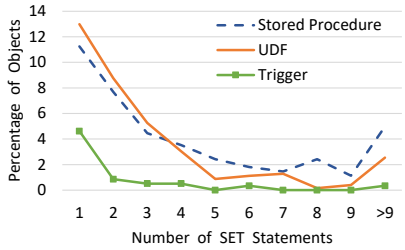
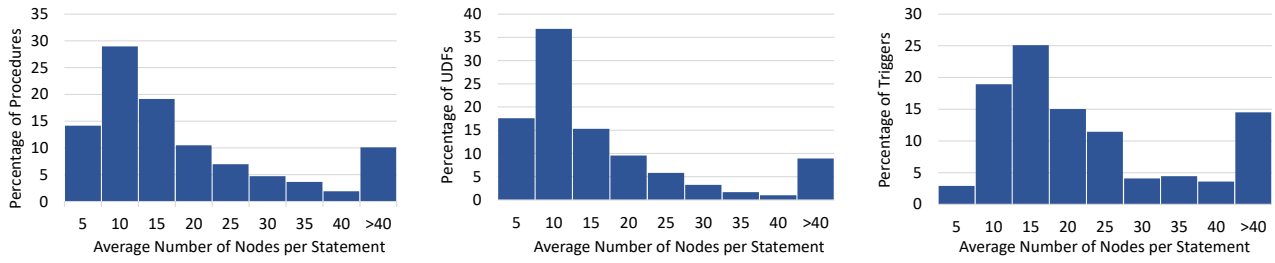


Figure 6: Distribution of *Set* Statements

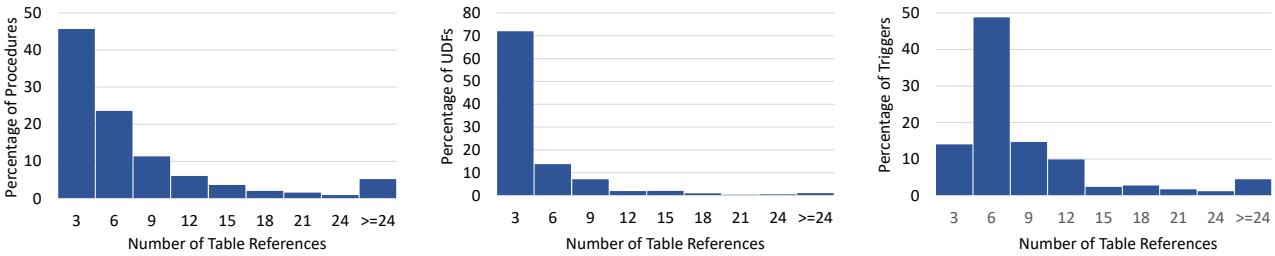
object code to get the maximum length of join chain for an object. The average max length of join chains across all stored procedures, UDFs and triggers which reference at least one table is 2.10, 2.19 and 2.15 respectively. The maximum value for this metric is as high as 25 for stored procedures, 9 for triggers and 40 for UDFs as shown in Table 3 along with other summary statistics.

3.3.3 Distribution of Control-Flow constructs. Figure 5 shows the distribution of the number of *IF* statements in our workloads. We observe that all these procedural objects make extensive use of



(a) Stored Procedures **(b) UDFs** **(c) Triggers**

Figure 7: Distribution of average number of nodes per statement



(a) Stored Procedures **(b) UDFs** **(c) Triggers**

Figure 8: Distribution of Table References

conditional statements. This is evident from the fact that 55% of stored procedures, 37% of UDFs and 44% of triggers in our analysis pool have at least one *if*-block. In fact, we also observe a high value for the average number of *if* statements per object – stored procedures: 3.08, UDFs: 1.36, triggers: 0.99. See Table 3 for details.

A large number of objects having multiple *if* statements also have them in a nested hierarchy – 21% stored procedures, 11% UDFs and 13% triggers contain at least one nested *if*-block. Further, we analyze the maximum nesting depth among all nested if-blocks present in an object and found it to be quite high: 8 in UDFs, 7 in triggers, and 90 in stored procedures! In addition to *if* statements, many procedural objects also contain loop constructs. Our analysis reveals that roughly 11% procedures, 7% UDFs and 8% triggers contain at-least one loop *(cursor/while)*.

3.3.4 Object References. Figures 8(a)-(c) show the distributions of the number of table references in stored procedures, UDFs and triggers. The x-axes in all these plots have bin width of 3. Note that these distributions are not cumulative of references inside nested invocations to UDFs and procedures. We observe that triggers on an average access more database tables than procedures and UDFs (see Table 3). This observation is in alignment with our earlier observation in Section 3.3.2 that triggers almost always have SQL statements in them.

3.3.5 Sequential imperative constructs. Figure 6 shows the frequency distribution of the number of *SET* statements in the 3 object types. We observe that 42% of stored procedures and 27% UDFs have at-least one set statement. Triggers on the other hand contain set statements comparatively less frequently – with roughly 8% triggers having at-least one set statement. This again can be attributed to our earlier observation that triggers usually contain more SQL statements as opposed to imperative statements.

3.3.6 Distribution of DML/DDL statements. Stored procedures and triggers often contain DML/DDL statements. 33% of stored procedures contain at-least one *INSERT* statement, 15% contain at-least one *DELETE* statement and 22% contain at-least one *UPDATE* statement. For triggers, we see that 40%, 41% and 28% triggers contain at-least one *INSERT*, *DELETE* and *UPDATE* statements respectively.

3.3.7 Parameters. The stored procedures from our analysis pool have on an average 2.63 parameters each and UDFs have 1.41 each. Triggers do not accept parameters as they are invoked automatically when an event that they are associated with occurs.

4 THE SQL-PROCBENCH BENCHMARK

We now present the SQL-ProcBench benchmark for procedural workloads in RDBMSs. We begin by describing its design philosophy, then describe the benchmark and provide some examples. The full benchmark specification is available at [13].

4.1 Design philosophy

RDBMSs offer a wide range of design choices for users at different levels of granularity, such as storage mechanisms and their corresponding storage layouts. Stored procedures, UDFs and Triggers end up operating on these various configurations at different scales and complexities. They are found in all kinds of workloads as well – transactional, analytical and hybrid.

Evaluation techniques for stored procedures and UDFs may result in very different characteristics w.r.t performance and resource utilization, depending upon database design choices and nature of the workload. Therefore, any benchmark that aims to model these

procedural extension workloads needs to consider this broad landscape. Keeping this in mind, we have incorporated the following choices in SQL-ProcBench.

4.1.1 Language. As described in Section 2.1, database systems support procedures and functions to be written in different programming languages in addition to procedural extensions of SQL. For procedures written in languages other than SQL, there are usually certain restrictions on the kinds of operations allowed. This benchmark focuses on procedural extensions to SQL as defined in the SQL/PSM ISO standard - as it represents the full scope of possible operations in such procedures.

4.1.2 Storage layouts. Both row and columnar storage layouts are extremely common in real-world scenarios, and users make this choice based on the nature of the workload. Usually row-stores are preferred for transactional/operational workloads and columnar storage layouts are preferred for analytical workloads. HTAP scenarios that are increasingly being adapted recently, include a combination or a mix of row and columnar layouts. A single database might have some tables in row format while others in columnar format. We include both kinds in SQL-ProcBench.

4.1.3 Storage Mechanism. Disk-based and in-memory databases are also very common in real-world scenarios. Again, there are also designs that incorporate a mix of these mechanisms – where some tables are disk-based and others are memory-resident. SQL-ProcBench is designed with this aspect in consideration, and some of our experiments show results in both these configurations.

4.1.4 Query types and complexity. Procedural extension of SQL are usually interspersed with declarative SQL queries in two ways. (a) Embedded queries: queries found inside the body of procedures, UDFs and triggers, and (b) Calling queries: UDFs invoked as part of an SQL query in the FROM clause (table functions) or in the SELECT/WHERE clauses (scalar functions). Further, the queries found inside procedures can be read-only (SELECT) queries or DML (INSERT/UPDATE/DELETE/...) depending upon the type of the object. We have incorporated all these query types in SQL-ProcBench. Based on the results of our analysis, we also vary the complexity of both embedded and calling queries.

4.1.5 Modeled on real business domain scenarios. The stored procedures, UDFs and triggers in SQL-ProcBench have been designed such that they not only represent real world procedural workloads in terms of their statistical properties, but they also model real-world business scenarios. This makes SQL-ProcBench more realistic and helps the community understand the scenarios better. This is in-line with standard database benchmark scenarios such as TPC-H, TPC-DS etc.

4.2 Benchmark Description

The SQL-ProcBench benchmark has been carefully designed to incorporate the various aspects described above, while faithfully reflecting real-world procedural SQL workloads. To achieve the latter, we have made use of the results of our analysis of many real-world workloads as presented in Section 3. We now describe the schema, procedures and queries, and then give a few examples.

4.2.1 Schema. The SQL-ProcBench schema is based on an *augmented* TPC-DS [18] schema which models the sales and returns process of a retail business through 3 different sales channels: store, catalog and web. The original TPC-DS schema consists of 7 fact tables and 17 dimension tables. Six fact tables store the sales and returns data for each of three channels and one fact table stores the inventory of items for the catalog and web channels. We chose to build the SQL-ProcBench schema based on TPC-DS due to the following reasons:

- TPC-DS schema is well known and is widely used in the community; it enables easy data generation and convenient scaling.
- The schema models a real-world business scenario.
- The data distributions and skew properties reflect real-world enterprise datasets.
- Table cardinalities scale realistically - with fact tables scaling linearly and dimension tables scaling sub-linearly.

For SQL-ProcBench, we augment the TPC-DS schema with 7 new fact tables namely *store_sales_history*, *catalog_sales_history*, *web_sales_history*, *store_returns_history*, *catalog_returns_history*, *web_returns_history* and *inventory_history* to store historical sales and returns data. The attributes of these history tables are identical to their non-history counterparts from the original schema. The DDL statements for these tables are also given in [13].

This augmentation enables the following scenario: all the data pertaining to sales and returns of the current year is stored in the non-history tables, and the corresponding history tables store all past data. This design is inspired by system-versioned tables or temporal tables introduced in the SQL:2011 standard [14]. In this model, the non-history tables will undergo frequent updates and inserts; and are thus stored as row-store tables in the database. The history tables are rarely updated and are used to perform analytical queries on the history data. These are therefore stored using a columnar layout. The dimension tables are relatively much smaller and are stored as row-store tables. Note that these storage layouts are just provided for guidance and represent one possible combination that we used for our experiments; users are free to use any layout as desired.

As it can be observed, the SQL-ProcBench schema can model different kinds of workloads (transactional, analytical and hybrid), and various database configurations. This helps us to cover the gamut of usage scenarios encountered in real workloads. Depending upon the aspect that needs to be measured, parts of the benchmark can therefore be used as necessary. For instance, to focus on transactional workloads, a slice of SQL-ProcBench that only targets rowstore tables (non-history tables and dimension tables) can be used.

4.2.2 Indexes. All the non-history fact tables and the dimension tables are stored in row-store format and have clustered indexes on their primary-key attributes. This is because they store the most recent 1 year data and are updated frequently. The history fact tables on the other hand are stored in columnar format with the corresponding index structures. This is done to enable efficient analytical operations on history tables.

In addition to these, we have also created non-clustered B-tree indexes on the sold_date_sk attributes of the 3 sales tables: store_sales, catalog_sales, web_sales; and the inv_date_sk attribute of the inventory table. Note that indexes and their implementations may

vary across databases, so we only recommend the above indexes to be present. The index creation statements are also included in [13].

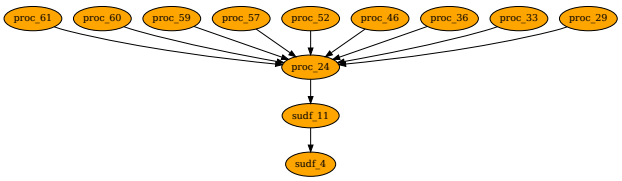


Figure 9: An object dependency graph from SQL-ProcBench

4.2.3 Objects and queries. SQL-ProcBench consists of 103 objects belonging to 3 categories: Stored Procedures, User Defined Functions (UDFs) and Triggers. The UDFs are further sub-categorized into Scalar UDFs and Table Valued UDFs. The calling queries and statements that are used to invoke these objects, along with the object definitions can be found in [13]. The break-up for each of these object types is: 63 stored procedures, 24 scalar UDFs, 10 table valued UDFs and 6 triggers.

This object-type distribution in SQL-ProcBench is based on the object-type distribution seen in Azure SQL Database shown in Figure 1. To reflect real world usage scenarios, the distribution of various other properties described in section Section 3.3 in SQL-ProcBench also mirror those seen in the real workloads. Moreover, like real-workloads, SQL-ProcBench also has complex inter-object dependencies; one such dependency graph is shown in Figure 9.

The schema definitions, object definitions and queries are all available at [13] in three SQL dialects: PL/SQL [10], PL/pgSQL [7] and T-SQL [20]. The naming convention we use is as follows. Stored procedures are named as *proc_<i>_<name>*, scalar UDFs are named as *sudf_<i>_<name>*, table valued functions named as *tvf_<i>_<name>* and triggers are named as *trig_<i>_<name>*; where *<i>* is a number identifying the object and *<name>* is the name of the object as created inside the database.

Listing 1: Definition of procedure AssessItemQuality

```
create procedure AssessItemQuality as
begin
  declare @maxRetItems table(itemNo int, manId int);
  insert into @maxRetItems
    select * from dbo.MaxReturnItems();
  delete from item where i_item_sk in
    (select itemNo from @maxRetItems);
  update item set i_item_desc = 'HIGH_RISK_ITEM'
    where i_manufact_id in
    (select manId from @maxRetItems);
end
```

4.3 Examples

To illustrate some objects present in SQL-ProcBench, consider a scenario that aims to assess the quality of items sold through the retail business. This scenario is modeled through a stored procedure *AssessItemQuality* whose definition is shown in Listing 1 in the T-SQL dialect. The PL/SQL and PL/pgSQL variants of this scenario can be found in [13]. This procedure first finds all items that are returned the maximum number of times by calling a user-defined function *MaxReturnItems()*, and inserts the result in a variable *@maxRetItems*.

Listing 2: Definition of UDF MaxReturnItems

```
create function MaxReturnItems()
returns @maxRetItems table (itemNo int, manufactId int) as
begin
  declare @itemNo int, @manufact int;
  declare @recDate  date;
  declare @itemTbl table(itmNo int,cnt int);

  insert into @itemTbl
    select top 1000 cr_item_sk,count(cnt) tCnt
    from
      (select cr_item_sk, count(*) cnt
      from catalog_returns group by cr_item_sk
      union all
      select wr_item_sk, count(*) cnt
      from web_returns group by wr_item_sk
      union all
      select sr_item_sk, count(*) cnt
      from store_returns group by sr_item_sk)t
    group by cr_item_sk order by tCnt desc

  declare c1 cursor for select itemNo from @itemTbl;
  open c1;  fetch next from c1 into @itemNo;
  while(@@FETCH_STATUS=0) begin
    set @recDate = (select i_rec_start_date
                from item where i_item_sk=@itemNo);
    set @manufact = (select i_manufact_id
                from item where i_item_sk=@itemNo);
    if(DATEDIFF(day,@recDate,'2000-01-01')>0)
      insert into @maxRetItems
        values(@itemNo, @manufact);
    fetch next from c1 into @itemNo;
  end
  close c1;  deallocate c1;
  return;
end
```

Listing 3: Definition of a Trigger on the Item table

```
CREATE TRIGGER delUp_item ON item AFTER DELETE, UPDATE AS
begin
  if(update(i_item_sk)) begin
    raiserror('Operation not allowed', 16, 10);
    rollback transaction;
  end
  if exists (select * from inserted) begin
    insert into logTable values
      ('logging updation to item table', GETDATE());
  end
  else begin
    delete from catalog_sales
    where cs_item_sk in (select i_item_sk from deleted);
    delete from catalog_returns
    where cr_item_sk in (select i_item_sk from deleted);
    delete from store_sales
    where ss_item_sk in (select i_item_sk from deleted);
    delete from store_returns
    where sr_item_sk in (select i_item_sk from deleted);
    delete from web_sales
    where ws_item_sk in (select i_item_sk from deleted);
    delete from web_returns
    where wr_item_sk in (select i_item_sk from deleted);
    delete from promotion
    where p_item_sk in (select i_item_sk from deleted);
    delete from inventory
    where inv_item_sk in (select i_item_sk from deleted);
  end
end
```

Table 4: Database engines and configurations used

Name	Fact & Dimension tables		History tables	
	Layout	Mechanism	Layout	Mechanism
D1	Row	Disk	Columnar	Disk
D2	Row	Disk	Row	Disk
D3	Row	Disk	Columnar	In-Memory
D4	Row	In-Memory	Row	In-Memory

Then, it deletes these items and marks all other items from the manufacturers of these *highly returned items* as 'high risk'.

Listing 2 shows the code for the user defined function *MaxReturnItems()*. This function first finds out the 1000 most returned items across all the sales channels and inserts them into a table variable *@itemTbl*. Next, it loops over this *@itemTbl* to find items which are outdated (received prior to 2000); these items and their manufacturers are then inserted into the table *@maxRetItems* which is finally returned to the calling procedure.

These updates and deletes in the item table also invoke a trigger which logs the updates into a logging table and performs necessary deletions from several other database tables to maintain referential integrity. The definition of this trigger *delUp_item* is shown in Listing 3. These examples give a glimpse of the kind of procedural code that is present in the SQL-ProcBench benchmark.

5 EXPERIMENTS

Using SQL-ProcBench, we have conducted experiments on multiple database engines with different configurations. In this section, we present some of the results and our observations. The broad goals of these experiments are (a) to understand the characteristics regarding evaluation of procedural SQL across different database engines and configurations, and (b) to uncover some of the insights we get from running SQL-ProcBench, thereby identify and highlight opportunities for future research in this area.

5.1 Setup

We have conducted experiments on four database engines and configurations. The characteristics of these four engines D1-D4 are given in Table 4. The names of these systems have been withheld for anonymity. All these databases have been set up on machines with identical hardware and software configurations. The machines were equipped with Intel(R) 2.10 GHz dual processors with 8 hyper threaded cores each, resulting in 32 logical cores. They had 256 GB of RAM, and a 1TB SSD. All of them used Windows Server 2019 as the operating system.

We have used the default database settings for all our experiments, and have not attempted to manually tune any engine based on our workloads. We set up the SQL-ProcBench schema as described in Section 4.2 with 3 different scale factors: 1GB, 10GB and 100GB. All our experiments have been conducted on these 3 scale factors. Since different database engines expose different levels of diagnostics and monitoring information about time and resources spent in executing procedural components, we show results based on the availability of this information.

We present results in this paper that cover stored procedures, UDFs and triggers of varying complexities. Table 5 gives some key properties about the chosen UDFs and their complexity. We show the number of statements, describe the logic of the UDF briefly, and also describe the complexity of the query that invokes this UDF. Table 6 summarizes key properties about the chosen stored procedures, showing the number of statements, object dependencies, and some information about the operations in the procedure. Similarly, Table 7 shows properties about triggers. We show the DML statement that is used to invoke the trigger with the number of rows affected by the DML statement in brackets, the affected table, some notes on the complexity of the logic inside the trigger and the number of statements in the trigger code. As it can be observed from these three tables, we have picked a mix of simple and complex procedures with different characteristics.

5.2 Where is time spent?

Using SQL-ProcBench, We conduct a series of experiments to answer the following question: *What fraction of time/resources is spent executing the procedural SQL component in a given query/workload that involves a procedural component?* This is important to understand and emphasize the importance of efficient evaluation of procedural components.

5.2.1 User Defined Functions. UDFs are usually invoked from an SQL query. The overall performance/resource utilization of such queries will depend both on the complexity of the invoking query as well as the complexity of the UDF, in addition to data sizes. Note that the query that invokes the UDF might itself be simple or arbitrarily complex in many scenarios, involving large tables, joins, aggregates etc. and the UDF can also vary in complexity.

Therefore, we conduct the following experiment to understand how the relative complexities of the UDF and its calling query might impact the time spent in the UDF. First, we consider a simple, single-statement UDF *sudf_20a_GetManufactSimple* that just looks up a small table and returns a value. Please refer to Table 5 for details about this UDF. We consider two scenarios for executing this UDF. First, where the invoking query is very simple, and second, where the invoking query is very complex. The calling queries can be found in the benchmark specification [13]. We run these two queries that invoke the same UDF with varying data sizes.

Figure 10(a) shows the results of this experiment on D1. The x-axis indicates the data size. We vary the data size and hence the number of UDF invocations by using the LIMIT clause of SQL (or equivalent). The y-axis shows the percentage of the total execution time of this query that was spent in evaluating the UDF. The orange (circle marker) and blue (triangle marker) lines show the results for the simple and complex calling queries respectively.

We observe that if the query that invokes this UDF is simple, almost 100% of the time is spent in executing the UDF, irrespective of the data size. On the other hand, if the calling query is complex, we see that at smaller data sizes, the UDF is not the bottleneck – more time is spent in executing the calling query. However, the larger the data gets, the more the time spent executing the UDF. For the simple UDF that we chose, more than 70% of the time was spent in the UDF when executing over 100k rows.

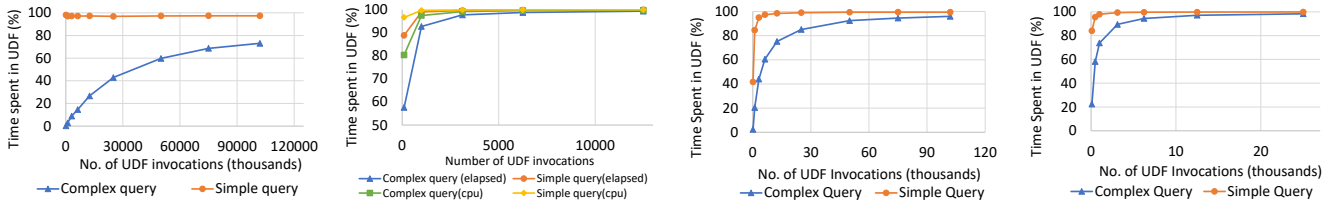
The key takeaway from this graph is that even though the procedural component (the UDF in this case) is very simple, it can in fact be the main bottleneck when operating on large datasets. This

Table 5: UDFs used in our experiments

Name	Num. Statements	Remarks on Calling Query	Remarks on UDF Code
sudf_1_TotalLargePurchases	1	Complex - with table joins, Group by, Order by	Return output of select statement
sudf_5_MorEveRatio	4	Contains subquery with *distinct* filtering	Mutliple assigns with aggregate queries
sudf_6_TotalDiscount	3	Table Scan with *Distinct* Filtering	Multiple assigns with aggregate queries
sudf_7_ProfitableManager	4	UDF called from the *where* clause	Aggregate query with conditionals
sudf_12_IncWebSpending	10	Performs set intersect.; UDF call in *where* clause	Multiple assigns with conditionals
sudf_13_MaxPurchaseChannel	12	Table Scan	Contains nested *if-else* blocks
sudf_15_IncomeBandOfMaxBuy	13	UDF called from where clause; Presence of *Order by*	Multiple conditional statements
sudf_20a_GetManufactSimple	1	Expt. with both simple and complex calling queries	Table lookup
sudf_20b_GetManufactComplex	8	Expt. with both simple and complex calling queries	Access multiple large fact tables
tvf_4a_BestPromoWeb	2	Called from procedure proc_23	Insert into table variable
tvf_4b_BestPromoCatalog	2	Called from procedure proc_21	Insert into table variable
tvf_4c_BestPromostore	2	Called from procedure proc_22	Insert into table variable
tvf_8_MaxReturnItems	15	Called from procedure proc_63	Contains cursor loop

Table 6: Stored Procedures used in our experiments

Name	Num. Statements	Dependencies	Remarks
proc_21_ActivatePromoCat	8	tvf_4b, trig_3	Updates promotion table in a loop
proc_22_ActivatePromoStore	8	tvf_4c, trig_3	Updates promotion table in a loop
proc_23_ActivatePromoWeb	8	tvf_4a, trig_3	Updates promotion table in a loop
proc_24_CreateRandomString	6	sudf_11	Generates random string from specified character set
proc_25_DelCatalogPage	5	trig_5	Deletes and updates in conditional blocks
proc_43_DeleteCustomer	9	trig_6	Delete using cursor loop
proc_52_CatalogOrderCancellation	16	proc_24	Multiple assignments, conditionals and error generating statements.
proc_63_AssessItemQuality	4	tvf_8, trig_4	presence of multiple DML statements



(a) Simple UDF (D1) **(b) Complex UDF (D1)** **(c) Simple UDF (D4)** **(d) Complex UDF (D4)**

Figure 10: Percentage of time spent executing a UDF when called from a query

is because the overhead introduced due to iterative execution of UDFs eventually outweighs the complexity of the calling query.

Now we consider a more complex UDF: *sudf_20b_GetManufact-Complex* and repeat the experiment by invoking it from a simple and complex query. The results of this experiment are shown in Figure 10 (b). The x-axis indicates data size and y-axis shows the percentage of the total time spent in evaluating the UDF. The orange (circle marker) and blue (triangle marker) lines show the percentage of elapsed time in the UDF for the simple and complex calling queries respectively. The yellow (diamond marker) and green (square marker) lines show the percentage of CPU time spent in the UDF for the simple and complex calling queries respectively.

For the simple calling query, we see that the percentage of time spent in the UDF is quite high right from low cardinalities and remains high. For the complex calling query, the percentage of time spent in the UDF starts off low, but quickly reaches a high value and remains high. This shows that having complex UDFs invariably means that the bottleneck will most likely be the UDF, irrespective of the complexity of the calling query or the size of the data. We have observed similar trends in other databases as well; Figure 10(c) and Figure 10(d) show the results for D4. While there have been recent efforts towards optimizing UDFs [30, 35, 48], there is still a huge optimization opportunity as described in Section 6.

5.2.2 Triggers. As mentioned earlier, triggers are often used in operational or transactional workloads to maintain integrity of the data, log information, enforce business rules, etc. To measure the overhead introduced by triggers on transactional workloads, we execute DML statements (INSERT/UPDATE/DELETE) on tables that have triggers associated with them. Then, we measure the time taken to execute these statements with and without the triggers enabled. Note that these statements were run individually (no concurrent execution). There are a total of 6 triggers in SQL-ProcBench and we show results for all of them. Table 7 explains the DML used in these experiments, and also gives some information about the complexity of the logic in the trigger.

The results of this experiment for D2 and D3 are shown in Figure 11 and Figure 12 respectively. The y-axis shows the percentage of time spent in the trigger code while executing the DML statement. From the figure it is clear that triggers often consume a significant chunk of the overall query execution. For 3 statements, triggers

Table 7: Triggers used in our experiments

Name	DML (rows affected)	Table	Remarks on Trigger Complexity	Lines of Code
Trig1	INSERT (1)	date_dim	Conditional checks to validate the legality of the insertion	20
Trig2	UPDATE (15)	customer_address	Check validity; insert in log table based on multiple conditional checks	17
Trig3	UPDATE (200)	promotion	Logging the update in a log table	1
Trig4	DELETE (47, in loop)	item	Deletions in other tables to maintain referential integrity	14
Trig5	DELETE (34)	catalog_page	Conditional validity checks; multiple updates using cursor loop	13
Trig6	DELETE (1)	customer	Multiple deletes in cursor loop	14

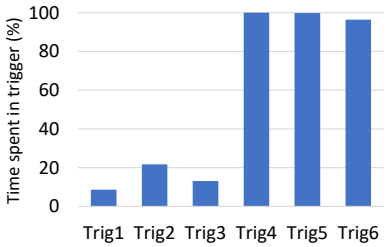


Figure 11: Trigger Overhead (D2)

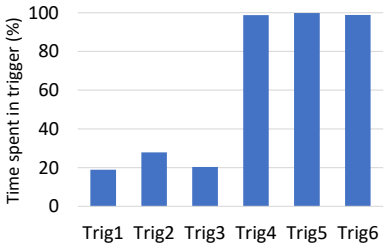


Figure 12: Trigger Overhead (D3)

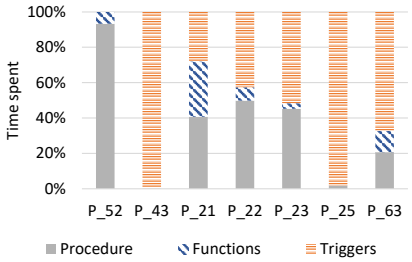


Figure 13: Break-up of stored procedure execution time (D1)

took around 20-25%, whereas for the other 3, we found that most of the time was spent in the triggers (>95%). This depends on the complexity of the procedural code within the trigger and the number of rows affected by the DML operation (which determines the number of times the trigger gets invoked). For instance, we see that triggers Trig4, Trig5 and Trig6 all contain looping constructs. But we also observe that although Trig3 affects 200 rows, it is still a fairly simple operation of logging the update into a log table, so it is relatively cheap. as shown in Figure 12, D3 shows similar behavior as D2 w.r.t trigger overheads. It should also be noted that none of the SQL translation based optimization techniques described in Section 2.2 can handle triggers currently.

5.2.3 Stored Procedures. Stored procedures are the most general form of procedural code, which can perform all kinds of permissible database operations such as data retrieval, transactions etc. In order to understand how time is spent in stored procedures, we collected information about the percentage of time spent in the various modules (UDFs/triggers and other procedures) as part of execution of a stored procedure. Figure 13 shows the results for 6 procedures from SQL-ProcBench. Except for Proc_52, we see that for other procedures, significant amount of time is spent in triggers, and some time in nested UDFs. This implies that any efforts to optimize triggers and UDFs will have a cascading effect on improving the performance of the calling procedures and functions. While techniques mentioned in Section 2.2 can help, but they provide limited coverage or performance gains. For instance, the SQL translation techniques cannot handle DML and exception handling constructs, which are fairly common in stored procedures.

5.3 Impact on Transaction Throughput

We conducted an experiment to study the impact of procedures on transaction throughput. We considered 3 relatively simple stored procedures that perform transactions with inserts and updates, and ran a mix of these transactions concurrently. These concurrent threads were run from a separate machine and connected to the database server through high speed LAN. Figure 14 shows the result of this experiment for D1 with both Serializable and Read committed isolation levels. We increase the number of concurrent threads, shown on the x-axis, and measure the throughput in terms of transactions per minute, shown on the y-axis in log scale. The red(minus) and green(cross) lines show the results with triggers disabled, and the yellow(square) and violet(diamond) lines show results with triggers enabled.

For both the isolation levels, we see an order of magnitude degradation in throughput when triggers are enabled, for all the concurrency levels. These triggers are quite simple, but run into contention-related delays as they update the same database log table. This degradation increases with more concurrency, and is observed for both isolation levels. We observed similar trends in other database engines as well, although the exact numbers vary. This shows that optimizing stored procedures and triggers in transactional workloads can directly improve transactional throughput.

Table 8: Scalability experiment for D1

Name	Data growth Factor	Execution time growth factor	
		1G-10G	10G-100G
sudf_5	3,8	1.22 (sub-linear)	3.32 (sub-linear)
sudf_6	10	22.6 (super-linear)	10.99 (super-linear)
sudf_7	10	6.06 (sub-linear)	23.67 (super-linear)
sudf_12	9	63.8 (super-linear)	9.42 (linear)
sudf_13	10	123.8 (super-linear)	16.07 (super-linear)
sudf_15	10	12.59 (super-linear)	14.76 (super-linear)

5.4 Impact of Scale

To understand the impact of data scale, we ran SQL-ProcBench over 3 different scale factors: 1GB, 10GB and 100GB. The results of this experiment for D1 is given in Figure 15 and Table 8. We show the results for 6 UDFs. Figure 15 shows scale factor on the x-axis and execution time (log scale) on the y-axis. Each line represents a UDF.

Table 8 gives more details about how performance of these UDFs degrade at larger scales. It shows the variation in execution times

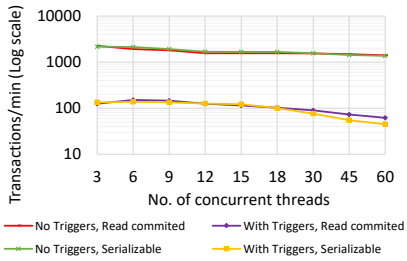


Figure 14: Transaction throughput (D1)

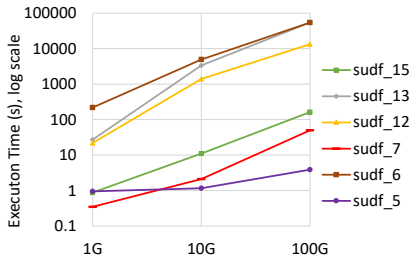


Figure 15: Scalability in D1

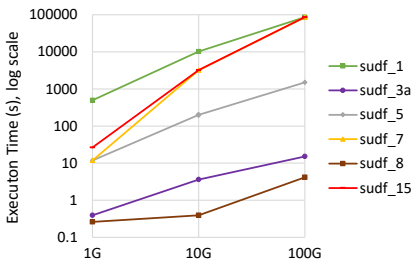


Figure 16: Scalability in D3

Table 9: Scalability experiment for D3

Name	Data growth Factor	Execution time growth factor	
		1G-10G	10G-100G
sudf_1	9	20.67 (super-linear)	>11 (super-linear)
sudf_3b	10	9.18 (sub-linear)	38.38 (super-linear)
sudf_5	3, 8	16.95 (super-linear)	127 (super-linear)
sudf_7	10	270 (super-linear)	>13.6 (super-linear)
sudf_8	8	1.49 (sub-linear)	15.94 (super-linear)
sudf_15	10	121.85 (super-linear)	>13.4 (super-linear)

with data growth. The data growth factor column indicates the total increment factor across all the tables accessed by a UDF as we scale from 1GB to 10GB and then to 100GB. Note that since roughly 90% of the data scales linearly (fact tables) and the rest scales sub-linearly (dimension tables), the total data accessed by the UDF increments by slightly less than 10x in most cases. For sudf_5, the values 3,8 denote that the total data accessed increased by 3 times from 1-10G and by 8 times from 10-100G. The growth-factor numbers shown in the table are rounded off to the nearest integers.

We see a super linear execution time growth in a majority of cases. Some extreme cases show particularly high increments such as 123.8 times (for UDF 13). This shows that performance problems of procedural extensions are disproportionately exacerbated at larger data sizes. We observed similar trends on D3, shown in Figure 16 and Table 9. In most cases, we see a super linear increase in execution time as the data grows.

6 OPPORTUNITIES

We now discuss some interesting challenges and opportunities for research in this area.

Language support. We see two broad future directions in the area of language surface. First, procedural SQL extensions need to evolve to include more complex data structures and data types such as collections, trees etc. that are common in programming languages. Second, the current optimization techniques for procedural extensions are limited in their support – for instance, none of the existing techniques optimize DML statements or exception handling constructs. Further, optimization of procedures written in non-SQL languages is also a promising direction, as explored in [26, 32].

Handling large procedures and large data. Both compilation and translation-based techniques have limits w.r.t the size of procedures that they can optimize. Compilation often results in huge binaries and long compile times when procedures are large. SQL-translation techniques end up in huge, extremely complex SQL queries for large UDFs. This can push the query optimizer to its limits and result in bad plans or optimizer timeouts. We have observed this phenomenon in D1, D2 and D4.

A related interesting direction here is to extend cardinality estimation techniques to handle procedural SQL extensions, building upon the work of [23]. Based on experiments in Section 5.4, we have also seen how the performance of procedural programs degrade super-linearly in most cases as the scale of data increases. As the size of data increases in database applications, these problems start to surface and become more prominent. Any new techniques in this space must take this into consideration.

Combining compilation and translation. As mentioned in Section 2.2, there are two broad directions that have evolved w.r.t optimizing procedural extensions of SQL viz. compilation and translation. These two have evolved independently so far, but there is a promising space where these two could potentially be combined.

Automatic Parallelization/Vectorization. Through our experiments, we observed that in general, most DB engines do not use parallelism to execute procedural SQL. While it is not straightforward to parallelize arbitrary procedures, it is definitely not a conceptual limitation. Therefore, automatically parallelizing and vectorizing complex SQL procedures is an interesting direction.

Testing and proving correctness. Testing and verifying correctness of techniques that optimize procedural SQL is challenging. Switching between declarative/relational and procedural semantics can often be tricky in terms of guaranteeing semantics. Testing and proving correctness of such techniques is very important. Techniques such as RAGS [53], XData [24], Cosette [27], MutaSQL [25] are attempts to achieve this for SQL queries but this needs to be extended to cover procedural SQL.

7 CONCLUSION

Procedural extensions of SQL are widely used in real-world applications. Users prefer the flexibility to express their intent using a mix of declarative and imperative code. In this work, we have performed a detailed analysis of thousands of real world procedural workloads and found that users write moderate-to-complex logic in these procedures. Based on our analysis, we have created SQL-ProcBench, an easy-to-use benchmark that represents real procedural workloads. Our experiments using this benchmark on multiple database engines highlight the opportunities that lie untapped in this area. Through our work, we hope to bring these challenges to the attention of the research community and encourage more contributions. We believe that our work enables novel contributions in this interesting and relevant area of research.

REFERENCES

[1] [n.d.]. Cyclomatic complexity. https://en.wikipedia.org/wiki/Cyclomatic_complexity.

[2] [n.d.]. Database Triggers. https://en.wikipedia.org/wiki/Database_trigger.

[3] [n.d.]. Interleaved execution for MSTVFs in Microsoft SQL Server. https://docs.microsoft.com/en-us/sql/relational-databases/performance/intelligent-query-processing?view=sql-server-ver15#interleaved-execution-for-mstvfs.

[4] [n.d.]. MemSQL SQL compiler. https://www.singlestore.com/blog/full-fledged-sql-compiler-faster-query-processing/.

[5] [n.d.]. Microsoft Azure SQL Database. https://azure.microsoft.com/en-us/services/sql-database/.

[6] [n.d.]. Performance overhead of SQL user-defined functions. http://glennpaulley.ca/conestoga/2015/07/performance-overhead-of-sql-user-defined-functions/.

[7] [n.d.]. PL/pgSQL: SQL Procedural Language for PostgreSQL. http://www.postgresql.org/docs/8.2/static/plpgsql.html.

[8] [n.d.]. PL/SQL compilation in Oracle. http://www.dba-oracle.com/t_compiled_pl_sql.htm.

[9] [n.d.]. PL/SQL Function Result Cache. http://www.oracle.com/technetwork/issue-archive/2010/10-sep/o57plsql-088600.html.

[10] [n.d.]. PL/SQL: Oracle's Procedural Extension to SQL. http://www.oracle.com/technology/tech/pl_sql.

[11] [n.d.]. Scalar UDF Inlining. https://docs.microsoft.com/en-us/sql/relational-databases/user-defined-functions/scalar-udf-inlining?view=sql-server-ver15.

[12] [n.d.]. SQL Persisted Stored Modules. https://en.wikipedia.org/wiki/SQL/PSM.

[13] [n.d.]. SQL-ProcBench Benchmark Specification: Schema, Queries and Objects. https://aka.ms/sqlprocbench.

[14] [n.d.]. SQL:2011 or ISO/IEC 9075:2011. https://en.wikipedia.org/wiki/SQL:2011.

[15] [n.d.]. Subprogram Inlining in Oracle. https://docs.oracle.com/cd/B28359_01/appdev.111/-b28370/inline_pragma.htm.

[16] [n.d.]. T-SQL User-Defined Functions: the good, the bad, and the ugly. http://sqlblog.com/blogs/hugo_kornelis/archive/2012/05/20/t-sql-user-defined-functions-the-good-the-bad-and-the-ugly-part-1.aspx.

[17] [n.d.]. The LLVM Compiler Infrastructure. http://llvm.org/.

[18] [n.d.]. The TPC-DS Benchmark Specification. http://www.tpc.org.

[19] [n.d.]. TPC-C and TPC-E. TPC-C and TPC-E Benchmark Specification. http://www.tpc.org.

[20] [n.d.]. Transact SQL. https://docs.microsoft.com/en-us/sql/t-sql/language-elements/language-elements-transact-sql.

[21] 2005. The TPC-H Benchmark Specification. http://www.tpc.org.

[22] 2020. The Java Database Connectivity API. https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/.

[23] Nicolas Bruno, Sameer Agarwal, Srikanth Kandula, Bing Shi, Ming-Chuan Wu, and Jingren Zhou. 2012. Recurring Job Optimization in Scope. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 805–806. https://doi.org/10.1145/2213836.2213959

[24] Bikash Chandra, Amol Bhangadia, Bhupesh Chawda, Biplab Kar, K. Reddy, Shetal Shah, and S. Sudarshan. 2014. Data Generation for Testing and Grading SQL Queries. *The VLDB Journal* 24 (11 2014).

[25] Xinyue Chen, Chenglong Wang, and Alvin Cheung. 2020. Testing Query Execution Engines with Mutations. In *Proceedings of the Workshop on Testing Database Systems* (Portland, Oregon) *(DBTest '20)*. Association for Computing Machinery, New York, NY, USA, Article 6, 5 pages. https://doi.org/10.1145/3395032.3395322

[26] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing database-backed applications with query synthesis *(PLDI)*. 3–14. https://doi.org/10.1145/2462156.2462180

[27] Shumo Chu, Daniel Li, Chenglong Wang, Alvin Cheung, and Dan Suciu. 2017. Demonstration of the Cosette Automated SQL Prover. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1591–1594. https://doi.org/10.1145/3035918.3058728

[28] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *ACM SIGMOD (SIGMOD '13)*. 12.

[29] Christian Duta and Torsten Grust. 2020. Functional-Style SQL UDFs With a Capital 'F'. In *ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1273–1287. https://doi.org/10.1145/3318464.3389707

[30] Christian Duta, Denis Hirn, and Torsten Grust. 2019. Compiling PL/SQL Away. *arXiv e-prints* (Sep 2019). arXiv:1909.03291 [cs.DB]

[31] K. Venkatesh Emani, Tejas Deshpande, Karthik Ramachandra, and S. Sudarshan. 2017. DBridge: Translating Imperative Code to SQL *(ACM SIGMOD)*. 1663–1666.

[32] K. Venkatesh Emani, Karthik Ramachandra, Subhro Bhattacharya, and S. Sudarshan. 2016. Extracting Equivalent SQL from Imperative Code in Database Applications *(ACM SIGMOD)*. 16.

[33] Sofoklis Floratos, Yanfeng Zhang, Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2018. SQLoop: High Performance Iterative Processing in Data Management. In *38th IEEE International Conference on Distributed Computing Systems, ICDCS 2018, Vienna, Austria, July 2-6, 2018*. 1039–1051.

[34] Craig Freedman, Erik Ismert, Per-Åke Larson, et al. 2014. Compilation in the Microsoft SQL Server Hekaton Engine. *IEEE Data Eng. Bull.* 37, 1 (2014), 22–30.

[35] Surabhi Gupta, Sanket Purandare, and Karthik Ramachandra. 2020. Aggify: Lifting the Curse of Cursor Loops Using Custom Aggregates. In *ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 559–573. https://doi.org/10.1145/3318464.3389736

[36] Ravindra Guravannavar. 2009. *Optimizing Nested Queries and Procedures*. PhD Thesis. Indian Institute of Technology, Bombay, Department of Computer Sc. & Engg.

[37] Ravindra Guravannavar and S Sudarshan. 2008. Rewriting Procedures for Batched Bindings. In *Intl. Conf. on Very Large Databases*.

[38] Wenjia He, Michael R. Anderson, Maxwell Strome, and Michael Cafarella. 2020. A Method for Optimizing Opaque Filter Queries. In *ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1257–1272. https://doi.org/10.1145/3318464.3389766

[39] Denis Hirn and Torsten Grust. 2020. PL/SQL Without the PL. In *ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2677–2680. https://doi.org/10.1145/3318464.3384678

[40] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything You Always Wanted to Know about Compiled and Vectorized Queries but Were Afraid to Ask. *Proc. VLDB Endow.* 11, 13 (Sept. 2018), 2209–2222. https://doi.org/10.14778/3275366.3284966

[41] A. Kohn, V. Leis, and T. Neumann. 2019. Making Compiling Query Engines Practical. *IEEE Transactions on Knowledge and Data Engineering* (2019), 1–1.

[42] K. Krikellas, S. D. Viglas, and M. Cintra. 2010. Generating code for holistic query evaluation. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. 613–624.

[43] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (Nov. 2015), 204–215. https://doi.org/10.14778/2850583.2850594

[44] Prashanth Menon, Amadou Ngom, Lin Ma, Todd C. Mowry, and Andrew Pavlo. 2020. Permutable Compiled Queries: Dynamically Adapting Compiled Queries without Recompiling. *Proc. VLDB Endow.* 14, 2 (2020), 101–113. https://doi.org/10.14778/3425879.3425882

[45] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (June 2011), 539–550. https://doi.org/10.14778/2002938.2002940

[46] Pat O'Neil, Betty O'Neil, and Xuedong Chen. 2009. (2009). https://www.bibsonomy.org/bibtex/26dc770318eb757ebc65ac4c3bce019d2/christophv

[47] Kisung Park, Hojin Seo, Mostofa Kamal Rasel, Young-Koo Lee, Chanho Jeong, Sung Yeol Lee, Chungmin Lee, and Dong-Hun Lee. 2019. Iterative Query Processing Based on Unified Optimization Techniques. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 54–68. https://doi.org/10.1145/3299869.3324960

[48] Karthik Ramachandra, Kwanghyun Park, K. Venkatesh Emani, Alan Halverson, César Galindo-Legaria, and Conor Cunningham. 2017. Froid: Optimization of Imperative Programs in a Relational Database. *PVLDB* 11, 4 (2017), 432–444.

[49] Maximilian E. Schüle, Jakob Huber, Alfons Kemper, and Thomas Neumann. 2020. Freedom for the SQL-Lambda: Just-in-Time-Compiling User-Injected Functions in PostgreSQL. In *32nd International Conference on Scientific and Statistical Database Management* (Vienna, Austria) *(SSDBM 2020)*. Association for Computing Machinery, New York, NY, USA, Article 6, 12 pages. https://doi.org/10.1145/3400903.3400915

[50] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. 2016. How to Architect a Query Compiler. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1907–1922. https://doi.org/10.1145/2882903.2915244

[51] Sourav Sikdar and Chris Jermaine. 2020. MONSOON: Multi-Step Optimization and Execution of Queries with Partially Obscured Predicates. In *ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 225–240. https://doi.org/10.1145/3318464.3389728

[52] V. Simhadri, K. Ramachandra, A. Chaitanya, R. Guravannavar, and S. Sudarshan. 2014. Decorrelation of user defined function invocations in queries. In *ICDE 2014*. 532–543.

[53] Don Slutz. 1998. *Massive Stochastic Testing of SQL*. Technical Report MSR-TR-98-21. 9 pages. https://www.microsoft.com/en-us/research/publication/massive-stochastic-testing-of-sql/

[54] Juliusz Sompolski, Marcin Zukowski, and Peter Boncz. 2011. Vectorization vs. Compilation in Query Execution. In *Proceedings of the Seventh International*

Workshop on Data Management on New Hardware (Athens, Greece) *(DaMoN '11)*. Association for Computing Machinery, New York, NY, USA, 33–40. https://doi.org/10.1145/1995441.1995446

[55] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. 2018. How to Architect a Query Compiler, Revisited. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 307–322. https://doi.org/10.1145/3183713.3196893

[56] S. D. Viglas. 2014. Just-in-time compilation for SQL query processing. In *2014 IEEE 30th International Conference on Data Engineering*. 1298–1301.

[57] S. D. Viglas. 2017. Processing Declarative Queries through Generating Imperative Code in Managed Runtimes. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. 1610–1611.