Next Generation Database Programming and Execution Environment

Dirk Habich, Matthias Boehm, Maik Thiele, Benjamin Schlegel, Ulrike Fischer, Hannes Voigt, Wolfgang Lehner
Dresden University of Technology
Database Technology Group
Dresden, Germany
{firstname.surname}@tu-dresden.de

ABSTRACT

The database research is always on the move. In order to integrate novel concepts, the significance of the database programmability aspect more and more increases. The programmability aspect focuses on internal components as well as on principle to push-down application logic to the database system. In this paper, we propose a novel database programming model and a corresponding database architecture framework enabling extensibility and a better integration of application code into DBMS. In detail, we present a scripting language *pyDBL* which is unified utilizable to implement physical database operators, query plans and even complete applications. We demonstrate the applicability of our approach in terms of a moderate performance overhead.

1. INTRODUCTION

> "The DBMS vendors (and the research community) should start with a clean sheet of paper and design systems for tomorrow's requirements, not to continue to push code lines and architectures designed for yesterday's needs." - Michael Stonebraker et al., [14]

Fundamentally, the requirements for tomorrow's database systems are manifold, whereas performance and usability are still the two most important requirements. Both requirements are not new and already well-established in the database community over long term. Nevertheless, most research work has been focused on solutions for satisfying the performance aspect, while the usability aspect has been singularly addressed in various works e.g. in several extension of the SQL standard. However, performance and usability are considered in an independent manner. The most challenging requirement for tomorrow's DBMS is the consolidation of performance and usability aspects under the term of programmability.

Without an opportunity to easy implement and to seamlessly integrate novel concept and methods addressing the performance aspect, most interesting and break-through concepts or methods will never be available as today. The reason is that new concepts—as an example the shift from row-systems to column-systems—are usually changing almost all layers of a highly complex system and such modifications are barely supported in today's DBMS with a layered architecture. In particular, the several abstraction layers complicate the implementation and integration. In order to support a better programmability with regard to the performance aspect, the established layered architecture must be revised offering a better extensibility and programmability of all layers in a comprehensive manner.

Beside the performance aspect, the usability requirement increases continuously. As already mentioned in the Claremont report on database research [1], the user base for DBMS is rapidly growing which implies new expectations with regard to programmability. Today, the corpus of the user base is unhappy (i) with offered user interfaces and (ii) with the heavyweight system architecture [1]. While the declarative way of SQL is usually too restricted, procedural opportunities like stored procedures are too complex. Furthermore, extensibility of traditional systems using the procedural opportunities is limited, which has an influence on usability and performance. Therefore, traditional DBMS are often degraded to storage units and enhanced functionality is implemented on top. However, with increasing data volumes, exporting massive data sets from the database and conducting data-intensive processing within the application is no longer a valid opportunity. Tomorrow's application will push-down their procedural logic to the raw data to execute the work near to the data.

Our Contribution

To tackle the performance and usability requirements in a unified way, we propose a novel database programming model and a corresponding database architecture framework enabling programmability regarding extensibility and a better integration of application code into DBMS. The key foundations of our concept can be described as follows:

Programming Model: We introduce a unified high-level procedural language for database programming—a database scripting language. This scripting language is uniformly utilizable to implement physical database operators, query plans and even complete applications. To establish this property, this scripting language is designed for data-

intensive applications by (i) providing db-specific language constructs/operators and (ii) an abstract view on the specific storage and access components. The advantages of this approach are extensibility and a reduced development effort.

Architecture Framework: We propose a database architecture framework with the ability to abstract heterogeneous hardware. This abstraction layer corresponds to a low-level virtualization serving as execution environment for our DB-scripts. We denote this virtualization layer as database low level virtualization machine (DBLLVM). To reduce the virtualization overhead, we present an optimization to integrate native operators for time-critical or highly-used operations.

Outline

In Section 2, we describe our novel database architecture including the resulting framework. While Section 3 gives an complete insight in our database scripting language, Section 4 take a close look at our virtualized execution environment. Finally, we presents some first evaluation results and further research work in Section 5, before we conclude the paper with a brief summary.

2. ARCHITECTURE FRAMEWORK

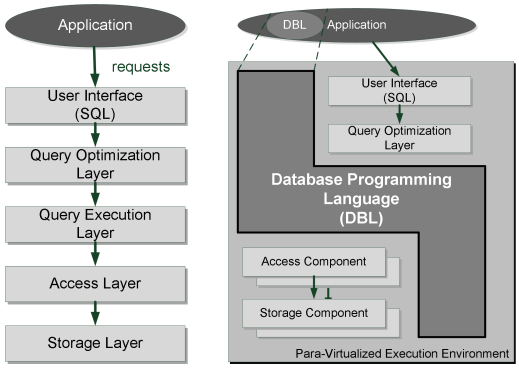
In this section, we are going to present (1) our novel database architecture and (2) core aspects of our constitutive framework.

2.1 Architecture

Database systems—open source as well as commercial—are usually a monolithic piece of complex software consisting of millions of code lines. Despite the availability of a layered database architecture, the complexity is nevertheless visible and hinders the extensibility. Figure 1(a) shows the well-known and often realized architecture for row-oriented database systems. Column-oriented database systems have a slightly adjusted architecture but follow the same principle [3, 11]. Each layer has its own functionality and the abstraction increases from the storage layer to the high-level declarative SQL interface. However, this layered architecture does not address the programmability neither for core operators nor for application logic.

To tackle the programmability issue for database developers as well as database users, we propose a revised architecture as illustrated in Figure 1(b). Our architecture does not fully correspond to a unified layered architecture. We describe our concept as DB programming-driven architecture form. In our architecture, the core component is a database programming language (DBL) being the gluing part between storage/access and and all other layers as e.g. application or query optimizer components—illustrated in Figure 1(b). Furthermore, our DBL is directly published for application developers, so that they are able to implement their application logic in our DBL. To establish the gluing aspect with an easy usability issue, our DBL has an abstract but specific view on various storage and access components.

Our DBL is uniformly utilizable to implement (1) specific operators for different storage or access components, (2) query execution plans and (3) application functionality. Therefore, our DBL has to correspond to a procedural programming language. Both points are resulting in the fact that we do not need to distinguish between external and internal logic, because everything is written in the same language. This allows us to include external application logic in all optimization decisions which is not possible with stored procedure or user-defined functions today. Moreover, declarative SQL statements are translated in our DBL using a standard query optimizer.



(a) Classic Layered Architecture (b) DB Programming-driven Architecture

Figure 1: Comparison of Database Architectures.

As execution environment, we propose to use a para-virtualization concept as second unique feature of our architecture. In order to efficiently glue DBL code and storage/access components together, we developed a specific DBL compiler which is based on virtualizing a process environment as proposed by the low-level virtual machine concept (LLVM) [5, 9].

2.2 Framework

Based on the presented architecture, four core frameworks parts are identifiable: (i) storage and access, (ii) SQL query optimizer, (iii) DBL, and (iv) virtualized execution environment. Our framework covers almost all parts except the first storage and access part, which have to specifically implemented using system programming languages such as C or C++. For the query optimizer several frameworks such as Starburst [13] or Volcano/Cascades [7] already exists and one of this frameworks could be used.

The role of our *Database Programming Language (DBL)* is to enable database system programmers as well as application programmers to implement their specific logic based on a certain storage or access components. In order to address this issue in an adequate manner, the DBL has to be a db-specific programming language. Instead of using a system programming language as foundation, we utilize a scripting language such as Python [8]. The expressive power of any scripting language is comparable to system programming languages, whereas scripting languages are typeless [12] and therefore, easier to learn and to use. The typeless property seems difficult at the first sight, but it is the best-fitting approach.

To specialize e.g., *Python* as specific DBL–further denoted as *pyDBL*—, we require a language concept to abstract various storage and access components. Our abstraction concept relies on maceration the typeless property by defining domain-specific types for each storage or access component. These language types offer the same properties at language level as the corresponding storage or access component at interface level. In this case, the core aspects of the under-

lying components have to be investigated and an adequate language type mapping has to be defined. In general, this guarantees an efficient extensibility on language level without making the language too complex.

Database operators, query execution plans and even applications are written in *pyDBL*. Our specific compiler transforms each *pyDBL*-script in some executable bytecode and executes it in a para-virtualized environment. Aside from regular compiler tasks such as loop unrolling, our compiler maps each activity regarding our included db-specific types with the corresponding storage and access interfaces. In this case, our *pyDBL*-scripts are skeletons and the compiler is responsible to glue *pyDBL*-scripts with storage and access components together.

3. DATABASE PROGRAMMING MODEL

In this section, we show how to use our *pyDBL* to script physical operations, query plans and even application logic. Furthermore, we give two examples of storage abstraction.

3.1 Operators

As already mentioned in Section 2, we abstract storage and access components using a language type definition concept. For example, a column storage component process a set of Binary Association Tables (BAT) on a conceptual level, whereas each BAT consists of two attributes `head` and `tail` [3, 11]. Moreover, each BAT is a multi-set of binary tuples [3]. The column storage component offers rich functionality to e.g. (i) create and delete a BAT, (ii) iterate over a BAT, (iii) access single binary units within a BAT. Normally, these interfaces are directly used to implement database operators such as selection or grouping. To overcome this issue, we create a new *pyDBL*-type `BAT` featuring a multi-set semantic and each record has two attributes, whereas these attributes are typeless at language level. Furthermore, this special type offers methods to add, insert and delete records. Listing 1 shows the *pyDBL*-script for the column-store specific `reverse`-operator on a storage independent level as we aimed with *pyDBL*. This operator gets a `BAT` as input and returns a `BAT` with swapped attributes [3]. As we can see, the script includes only necessary logic: (i) creating a new result `BAT` (line 2), (ii) iterating over input `BAT` (line 3), (iii) adding record with swapped attributes to result (line 4), and (iv) return result (line 5). In this case, a more efficient realization can be provided by storage component. We pick up this issue in Section 4 and propose a solution.

Listing 1: Column-Store Reverse-Operator

```
1 def col_reverse(BAT arg):
2  result = BAT()
3  for head, tail in arg:
4      result.append(tail, head)
5  return result
```

A more complex operator script is illustrated in Listing 2. This scripts represents the inner hash join algorithm for two BATs. According to [3], the join results in a BAT and consists of the outer attributes of the left and right BAT where their inner attributes matches. As depicted in Listing 2, the left BAT is hashed (line 3-5) and the right BAT is probed then (line 6-10).

Listing 2: Column-Store Hash-Join-Operator

```
1 def col_hashjoin(BAT arg1, BAT arg2):
2     result = BAT()
3     hashed={}    # hash table
4     for head1, tail1 in arg1:
5         hashed.put(tail1, head1)
6     for head2, tail2 in arg2:  # probing
7         values = hashed.get(head2)
8         if (values<>None):
9             for v in values:
10                result.append(v, tail2)
11    return result
```

Aside from this column-oriented approach, we are also able to abstract a row storage component. In this case, the storage component efficiently process a set of table elements, whereas each table consists of a number columns and each table is a multi-set of corresponding records. To abstract this row storage component in *pyDBL*, we constructed a specific `TABLE` type. This type has again a set semantic, whereas each record within this set is represented as array. The array representation is essential, then each table usually has different numbers of attributes—the attributes are typeless. Furthermore, the array representation enables an efficient method to directly access single columns of a record. Listing 3 shows the *pyDBL*-script of the select-operator evaluating a set of `or`-connected predicates. This operator gets a TABLE and a set of predicates—marked using *—as input.

Listing 3: Row-Store SelectOR-Operator

```
1 def row_selector(TABLE arg, *predicates):
2   result = TABLE()
3   for r in arg:
4     for p in range(len(predicates)):
5       if predicates[p](r):
6         result.append(r)
7         break;
8   return result
```

Listing 4 illustrates a realization of an equality predicate function in *pyDBL*. This predicate function gets an attribute position (`aPos`) and a reference value as input. In this case, we assume that by calling this predicate function, the necessary attribute position is known. How to ensure this assumption is described in Section 4. Using the `lambda` keyword, small anonymous functions can be created as well-known from functional programming languages.

Listing 4: Row-Store Equality Predicate

```
def pred_eq(aPos, value):
      return lambda x: x[aPos] == value
```

3.2 Queries

Aside from using *pyDBL* as operator programming language, we are able to utilize *pyDBL* as query execution language. In this case, *pyDBL* is similar to MIL [3], whereas operators and query scripts are written in the same language in our approach. Listing 5 shows an example SQL query, whereas Listing 6 illustrates the corresponding *pyDBL* script for a column-oriented database system. The operator definition are available at [3]. Either such *pyDBL* scripts are produced by the query optimizer or the application developer submits such scripts in NoSQL fashion [10].

Listing 5: Sample SQL-Query

```
SELECT category, brand, SUM(price)
FROM orders
WHERE data between '1-1-2010' and '31-03-2010'
GROUP BY category, brand
```

Listing 6: pyDBL - Query Execution Script

```
1 def example_query():
2   a = col_scan(open("date","year"),
3       pred_between('1-1-2010','31-03-2010'))
4   b = col_mark(col_reverse(a))
5   c = col_hashjoin(b,open("date","brand"));
6   d = col_hashjoin(b,open("date","category"))
7   e = col_hashjoin(b,open("date","price"))
8   f = col_group(c,d)
9   g = col_unique(col_mirror(col_reverse(f)))
10  h = col_sum([g, f, e])
11  col_emit(h, d, c) # output query result
```

3.3 Applications

As illustrated in Figure 1(b), our database programming language is exposed to applications, so that application programmers are able to realize parts of their procedural logic in *pyDBL* and this logic is then executed inside the database system. The biggest advantage is that every data-intensive procedural logic can be executed near to data and the data transfer between applications and database system can be massively reduced. To show the power of our *pyDBL*, Listing 7 shows the Dijkstra algorithm as column store-specific operator. Dijkstra is a graph search algorithm solving the single-source shortest path problem [6]. To represent a weighted graph, three BAT elements—source `bs`, cost `bc`, target `bt`—are necessary and this three BATs including a start node information are inputs of our operator. According to the well-known algorithm structure, we are able to transform this algorithm to a specific structure using column-oriented operators as shown in Listing 7.

Listing 7: Dijkstra Algorithm in *pyDBL*.

```
def col_dijkstra(BAT bs,BAT bc,BAT bt,start):
  visited={}
  queue = BAT([start],[0])
  while (len(queue)>0):
    current,cost=queue[0]
    queue=col_limit(queue,1,len(queue))
    g=visited.get(current)
    if (g==None) or (cost < g):
        visited[current]=cost
        a=col_scan( ba, pred_eq(current) )
        b=col_hashjoin( a.mirror(), bc )
        c=col_map(b, lambda x: x+cost )
        d=col_hashjoin( a.mirror(), bt )
        e=col_hashjoin( d.reverse(), c )
        queue=col_sort( queue + e, asc )
  result = BAT()
  for k,v in visited.iteritems():
        result.append(k,v)
  return result
```

Such application logic is usually not integral part of databases today. Using *pyDBL*, we are able to integrate this kind of application code as regular operator.

4. EXECUTION ENVIRONMENT

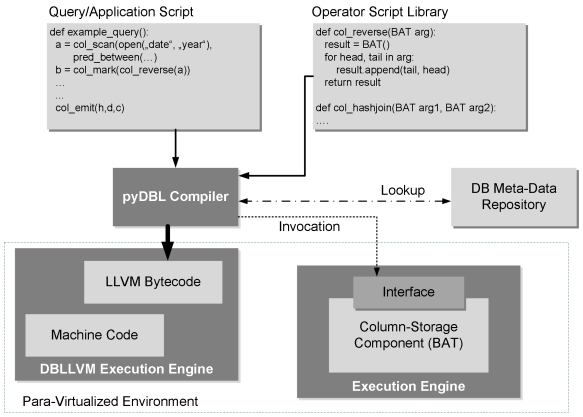


Figure 2: Compiler and Execution Infrastructure.

According to our proposed architecture, we have on the one hand storage and access components written in system programming languages. On the other hand, we have different *pyDBL*-scripts with an abstraction of storage and access components. The challenging task of our execution environment is now to glue everything together with regard to evolving hardware landscape. In order to tackle this issue, we utilize the low-level virtual machine concept (LLVM) [5, 9, 15], which is a compiler-backend infrastructure using a virtualized set of instructions.

4.1 Query and Application Compilation

Figure 2 illustrates the whole query/application compilation and execution procedure. While scripting language are usually interpreted at runtime [12], we utilize a compilation approach for our *pyDBL* language. One reason is the performance requirement which cannot be satisfied using language interpretation. Another reason is the included abstraction concept of storage and access components. As depicted in Figure 2, our *pyDBL* compiler receives a query or application script to be executed including all operator scripts as library. One major task of our *pyDBL* compiler is now to establish the connection with the corresponding interfaces of storage and access components. The second major task is to typify the scripts regarding the actual situation.

To describe these both special tasks more precisely, we assume a column-storage component with a ONC-interface (`Open, Next, Close`). This means, we can open a specific BAT column, iterate over the BAT using the `next`-operation and we are able to close the processing. Other interfaces are possible, but some adjustments on the following steps are necessary. Moreover, we want to execute the SQL statement of Listing 5 with the corresponding *pyDBL*-script as shown in Listing 6. As we can see in the *pyDBL*-script, the second operation is `reverse`-operator which is executed on a BAT determined by the first query script operation. This first method opens a BAT, where the *head*-column is by default of `long`-type (system-wide pre-definied) and the *tail*-column is a `date`-type in this case. These type information are determinable using script analysis in combination with variable tracing and accessing the meta-data repository of the database system (storing type information about created and available BATs). Therefore, we are able to extract type information for each `BAT`-variable in each script at

compile-time. If type information for intermediate variables are necessary (e.g. p-variable line 4 in Listing 3), then we can precisely estimate the types using some pre-defined rules as demonstrated in the related Python compiler approach [16]. In this case (`reverse`-operator at line 4 in Listing 6), this `reverse`-operator gets a BAT[long, date] as input and returns a BAT[date, long] as result. The `reverse`-operator in line 9 of the same query script process BATs with different types for head and tail attributes.

The correct derivation of types out of our typeless *pyDBL*-scripts is possible using script analysis, variable tracing, accessing meta-data repository and pre-defined rules. The next challenging task is to glue *pyDBL*-scripts with the corresponding storage interface together. In this task, we work with code replacement strategies in combination again with code analysis and variable tracing. Based on our introduced db-specific types, we are able to detect all code lines associated with variables of our db-specific types. Depending on the detected *pyDBL*-method, we replace the corresponding script methods with specific storage interface methods. Using our ONC-interface assumption and the determined BAT types, we construct a specific `reverse`-operator code for the invoked `reverse`-operator in line 4 of Listing 6. To illustrate the specific code, we show C++-like code lines in Listing 8.

Listing 8: Translated Reverse-Operator

```
BAT<date,long>
col_reverse(BAT<long, date> arg) {
  BAT<date, long> result =
                  new BAT<date, long>;
  Iterator<BAT<long, date>> iter =
                  arg->begin()
  while(iter->hasNext()) {
    Bun<long, date> b = iter->next();
    result.append((date)b.getTail(),
                  (long) b.getHead());
  }
  return result;
}
```

As derivable from Listings 1 and 8, the core structure of the `reverse`-script is kept, but some specific reformulation has been conducted. For example, the `for`-loop construct to iterate over the `BAT` language data object (line 3 in Listing 1), is replaced by a `while`-loop regarding the ONC interface of the column-storage component using the C++ iterator concept. Furthermore, the Listing 8 illustrates the inclusion of the already determined type information.

The type derivation and to establish the connection with arbitrary but well-defined interfaces of storage and access components are specific database compiler tasks in our DB programming-driven architecture. These compiler tasks have to be implemented at compiler level and the modular architecture of LLVM enables the efficient integration of those specific modules. Furthermore, standard optimization techniques such as loop unrolling are available and can be seamlessly integrated in our *pyDBL*-compiler.

4.2 Para-Virtualized Execution

As already illustrated in Figure 2, only query and application scripts are executed in our virtualized execution environment. Storage and access components run in a separate and specific execution environment being adjusted for their special needs. This so called para-virtualized execution is a core feature besides *pyDBL* compilation to satisfy the performance requirement and to reduce the overhead of the virtualization. To take more advantage of this para-virtualization, we are able to define rules to replace *pyDBL* operators with low-level operators. This property is crucial, when e.g. a specific operator is heavily used and a low-level implementation is more efficient then the *pyDBL* operator implementation. An outstanding example is the `reverse`-operator with a faster implementation on storage component where only two pointers have to be re-arrangement instead of processing all elements within a BAT (see Listing 1). Such low-level operators have to be made available at storage or access component and the replacement is part of *pyDBL* compiler. However, such specific storage optimization are not possible for all operators.

5. DISCUSSION

In this section, we (i) show first evaluation results, (ii) present future research work and (iii) discuss related work.

5.1 Evaluation

To evaluate our novel approach, we have prototypically implemented our described framework. Furthermore, we utilized this framework to realized a main-memory column-store database systems. In detail, we (1) implemented a column-storage component with ONC-interface using the system programming language C++, (2) scripted all relevant physical column-store operators [3] using *pyDBL*, (3) realized a sql parser with a standard query optimizer outputing a *pyDBL* query script, and (4) a rudimental version of our *pyDBL* compiler based on LLVM [15]. Aside from realizing a main-memory column-store database using our framework, we further implemented the same system using the classical approach using the same storage component. That means, tight coupling of storage interface with database operators–in this case, the column-store operators are written in C++— and the query optimizer outputs a C++ query program which are compiled using a regular g++ compiler (denoted as Classic).

For our evaluation, we used a reduced TPC-H benchmark where the fact table consisted of more than 6 millions entries. All experiments run a 64bit Linux machine with a Intel Core 2 Duo processor (3,06 GHz) and 8 GB main memory. We executed a number of OLAP-queries with increasing complexity, whereas the complexity is determined by the number of physical operators: Q1 (15 operators), Q2 (22 operators), Q3 (30 operators), Q4 (37 operators). As we can see in Figure 3, the runtimes (sum of compile and execution time) are slightly different. If we look at the pure query execution times, then we can only determine a negligible difference, because an optimal bytecode is executed in both cases. The difference originate in the compilation phase as illustrated in Figure 3. With our approach, we facilitate the implementation but we have to do more work in the compilation phase. However, the differences are only marginal and in contrast to the execution times of long-running queries, the moderate increased compile times are negligible.

5.2 Outlook

In this paper, we have proposed our database programming language *pyDBL* with an overall architecture framework in detail, whereas the description has been focused

Figure 3: Experimental Results.

on the extensibility aspect. The main drawback of current status is the storage dependency at *pyDBL* level for application developers. However, application code should be implemented in a storage independent way enabling further optimization opportunities and portability. Up to now, our database scripting language *pyDBL* includes only specific data types for different storage components. We are able to use the same concept to introduce specific data types for applications enabling an application-oriented implementation way. In order to execute such application scripts, we require a mapping of application data types to storage data types. Furthermore, this mapping has to be considered in the compilation phase like the inclusion of storage interfaces. The advantages of this approach are manifold, e.g. (1) application scripts are independent from any storage component and (2) reduced effort in comparison to today procedure. Today, application objects are transformed to relational entities using object-relational mapper first, and then processed by database system using a specific storage component. In our case, the object-relational mapper is eliminated. A further important benefit of our approach is that the application logic runs inside the database system and we do not further need a separate application server.

However, we are going to more precisely investigate the interplay between storage data types and application data types in our *pyDBL* in near future and how to integrate the mappings. In particular, we want to study the advantages and disadvantages of eliminating the relational view by transforming application objects directly to the storage layer. Furthermore, we want to explore the optimization potentials of our compiler when all application as well as database functionality is implemented in the same *pyDBL* language. A further topic of our ongoing research work, is to extend *pyDBL* with parallelism opportunities. In this work, we want to investigate how to script parallel operators in combination with parallelized applications.

5.3 Related Work

A number of projects have been addressing the needs of new applications by developing approaches to making a database system extensible; Exodus [4] or Genesis [2] are examples. However, all these projects have been proposed extensible concepts, whereas the programmability is restricted respectively not considered. Only the Exodus project includes a specific language to simplify the development. The so-called E language extends C++ with facilities for persistent systems programming focusing only on the implementation of new internal database components (e.g. new access methods or new query language operators).

To best of our knowledge, our *pyDBL* language and our architecture is the first integrated approach regarding programmability of databases. With our proposed language, different operators and application logic can be easy implemented and seamlessly integrated in the database without scarifying performance.

6. SUMMARY

Database research is always on the move, but today's systems are too complex and this hinders the integration of new research concepts. To tackle this programmability challenge, we have contributed a novel database architecture and framework with two distinct features of (i) an inherent database programming language and (ii) an extensible framework provided by a para-virtualized environment.

7. REFERENCES

[1] R. Agrawal et al. The claremont report on database research. *Commun. ACM*, 52:56–65, June 2009.

[2] D. S. Batory, J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twichell, and T. E. Wise. Readings in object-oriented database systems. chapter GENESIS: an extensible database management system, pages 500–518. 1990.

[3] P. A. Boncz and M. L. Kersten. MIL Primitives for Querying a Fragmented World. *VLDB J.*, 8(2):101–119, 1999.

[4] M. J. Carey, D. J. DeWitt, D. Frank, M. Muralikrishna, G. Graefe, J. E. Richardson, and E. J. Shekita. The architecture of the exodus extensible dbms. In *OODS'86*, pages 52–65, 1986.

[5] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization, 2002.

[6] T. H. Cormen, C. E. L. andRonald L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[7] G. Graefe. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[8] H. P. Langtangen. *A Primer on Scientific Programming with Python*. Springer, 2009.

[9] C. Lattner and V. S. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*, pages 75–88, 2004.

[10] N. Leavitt. Will NoSQL Databases Live Up to Their Promise? *Computer*, 43(2):12–14, Feb. 2010.

[11] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing database architecture for the new bottleneck: memory access. *VLDB J.*, 9(3):231, 2000.

[12] J. K. Ousterhout. Scripting: Higher-Level Programming for the 21st Century. *Computer*, 31(3):23–30, Mar. 1998.

[13] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in Starburst. In *SIGMOD*, volume 21, pages 39–48, 1992.

[14] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB*, pages 1150–1160, 2007.

[15] The LLVM Compiler Infrastructure. `http://llvm.org/`.

[16] Unladen Swallow - Python LLVM Compiler. `http://code.google.com/p/unladen-swallow/`.