Towards Programmability of a NUMA-aware Storage Engine

Dirk Habich, Johannes Schad, Thomas Kissinger, and Wolfgang Lehner

Technische Universität Dresden, Database Systems Group,
{dirk.habich,johannes.schad,thomas.kissinger,wolfgang.lehner}@tu-dresden.de
WWW home page: https://wwwdb.inf.tu-dresden.de

Abstract. The SQL database language was originally intended for application programmers. However, after more than 20 years of language extensions, SQL can only be generated by software components and is no longer suitable for an increasing user base like knowledge workers or data scientists, who want to work with data in an interactive fashion. The original idea of declarative query languages, telling the system what information to retrieve and not how to retrieve it, is still relevant. However, procedural elements are extremely worthwhile and have to be part of a next generation database programming language without compromising performance and scalability. To tackle this challenge, we are going to present our overall approach consisting of a highly-scalable NUMA-aware storage engine *ERIS* and a novel appropriate procedural programming approach on top of *ERIS* in this paper.

1 Introduction

In the era of *Big Data*, the requirements for data management systems are manifold, whereas performance and scalability are still the two most important technical requirements. Aside from these technical requirements, the relevance of the usability aspect increases. Generally, all these aspects are not new and already well-established in the database community over a long period. While most research work has been focused on solutions for satisfying the performance and scalability aspects, the usability aspect has been singularly addressed in various work as well e.g., the map-reduce programming concept or the PACT programming model [3] come to mind. From our point of view, the most challenging issue for tomorrow's data management systems is the consolidation of technical and usability aspects, taking programmability into account.

As already mentioned in the Claremont [2] as well as in the Beckman [1] report on database research, the user base for DBMS is rapidly growing and new users bring new expectation with regard to programmability or programming

Copyright © 2015 by the paper's authors. Copying permitted only for private and academic purposes. In: R. Bergmann, S. Görg, G. Müller (Eds.): Proceedings of the LWA 2015 Workshops: KDML, FGWM, IR, and FGDB. Trier, Germany, 7.-9. October 2015, published at http://ceur-ws.org

abstractions against very large data sets. Today, most of the users are unhappy (i) with the offered user interfaces and (ii) with the heavyweight system architecture. While the declarative way of SQL is often perceived as too restricted, procedural opportunities like stored procedures are too complex. Furthermore, the extensibility of traditional database systems using procedural opportunities is limited, which has an influence on usability as well as performance. Therefore, traditional DBMS are often degraded to storage units today and enhanced functionality is implemented on top or in competing data processing systems like Hadoop. However, with increasing data volumes, exporting massive data sets from the database and conduction data-intensive processing in user applications is no longer a valid opportunity. Tomorrow's applications will have to push-down their procedural logic into the database to work close to the data.

In order to tackle technical and usability requirements for database systems in a unified way to satisfy a growing user base like knowledge workers and data scientists, we are pursuing a holistic approach with novel and unique features:

NUMA-aware Storage Engine: The ever-growing demand for more computing power forces hardware vendors to put an increasing number of multiprocessors into a single server system, which usually exhibits a non-uniform memory access (NUMA). Based on that hardware foundation, we developed a new NUMA-aware in-memory storage engine *ERIS* that is based on a data-oriented architecture [7]. In contrast to existing approaches that focus on transactional workloads on a disk-based DBMS, *ERIS* aims at tera-scale analytical workloads that are executed entirely in main memory. *ERIS* uses an adaptive data partitioning approach exploiting the topology of the underlying NUMA platform and significantly reduces NUMA-related issues. As we have shown in [5], we achieve *more than a linear* speedup for index lookups and scalable parallel scan operations.

Programmability: Our NUMA-aware in-memory storage engine currently supports three basic storage operations that are required to run analytical queries: scan, lookup, and insert/upsert. Aside from reading operations, fast writing operations are necessary, especially to materialize large intermediate results. Based on those storage operations, we are currently developing a new procedural user interface, so that users are able to program their analytical tasks. In this paper, our primary goal is to introduce the procedural user interface and show how it can be used to add support for SQL functionality on *ERIS*. On the one hand, this new user interface facilitates easy and abstract programming in a procedural fashion and for a broad user base. On the other hand, the procedural analytical tasks are efficiently executable on our NUMA-aware storage engine *ERIS*.

The remainder of the paper is structured as follows: In the following section, we briefly review our NUMA-aware in-memory storage system. Then, we introduce our programmability concept using partitioning schemes as first-class citizen operators in Section 3. In Section 4, we discuss some execution perspectives. We conclude the paper, with some remarks regarding related work, future work and short conclusion.

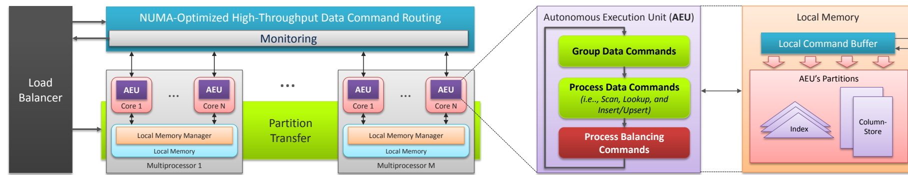

Fig. 1. Architectural Overview of ERIS and AEU Details.

2 The *ERIS* Storage Engine

In [5], we introduced our NUMA-aware all-in-memory storage engine for tera-scale analytical workloads. Fundamentally, NUMA systems consist of several interconnected multiprocessors (also denoted as nodes). Each multiprocessor contains multiple processing units (cores) and an integrated memory controller (IMC). Although the installed main memory is distributed across the IMCs of the different multiprocessors, each multiprocessor can access each memory location of the system. Therefore, latency and bandwidth of memory accesses depend on the distance between the requesting multiprocessor and the multiprocessor which has the data in local memory. The local memory associated with each multiprocessor is accessed with low latency at a high bandwidth. In contrast, remote memory is accessed via point-to-point connections between the multiprocessors that add latency and limit the achievable bandwidth. As we have shown in [5], the latency of remote access is approximately 10 times higher and the bandwidth is limited to about 11% compared to local accesses in the worst case.

2.1 *ERIS* Architecture Design

To efficiently tackle the NUMA-specific properties from a performance perspective for analytical workloads, our in-memory storage engine *ERIS* is treated like a distributed system using a data-oriented architecture [7] where each data object is logically partitioned. Figure 1 shows the overall architecture of *ERIS* and its individual components. The central components are the worker threads, which we call *Autonomous Execution Units (AEU)*. Each core, respectively hardware context, of a NUMA system runs exactly one AEU and an AEU never leaves its core. All AEUs that are pinned on the same multiprocessor use a common memory manager, because they share the same local main memory. Every AEU gets assigned a set of disjoint data partitions —each belonging to a different data object— which it stores in local memory. Further, every AEU holds exclusive access rights to its partitions. This approach restricts memory accesses of an AEU to the multiprocessor's local main memory and data objects do not have to be protected against concurrent accesses via latches. ERIS primarily uses range partitioning to split data objects (e.g., tables) into partitions.

On the right hand side of Figure 1, we depict the AEU loop as well as the local memory organization of an AEU. Each AEU maintains local data command

buffers and the actual data object partitions (either stored as a column-store, a row-store, or an index). In the first stage of the loop, the AEU scans its data command buffer (i.e., scan, lookup, or insert/upsert), which is periodically filled by the routing layer, and groups commands by the accessed data object and the command type. This optimization step allows to coalesce the same type of access to the same partition. For instance, an AEU is able to execute multiple scan commands on the same partition with a single scan and is thereby implementing scan sharing in combination with MVCC to ensure isolation. Moreover, the command grouping allows us to execute multiple index lookup or insert/upsert operations in a single batch operation to hide the main memory latency. Following the grouping step, the AEU actually processes its data command buffer, which is the most time consuming part of the loop. Afterwards, the AEU checks its command buffer for pending balancing or transfer commands. Such commands force an AEU to grow or shrink its partition or to transfer a range of its partition to another AEU.

NUMA-Optimized High-Troughput Data Command Routing: As shown in Figure 1, the data command routing is the most essential part of ERIS, because AEUs have to be supplied with data commands just in time. Especially during the execution of analytical queries, large amounts of data commands have to be routed between AEUs (e.g., lookup operations during a join). Thus, the main goal of the data command routing is to distribute data commands at a high throughput. A data command consists of a storage operation type (i.e., scan, lookup, or insert/upsert), a data object identifier, a reference to a callback function, a data segment that contains all the necessary parameters for the storage operation (e.g., a batch of keys for the lookup or filters for a scan), and additional data that is necessary for the query processing.

Load Balancing Besides data command routing, *ERIS* owns a NUMA-aware load balancer component to adapt the data partitioning to a changing workload. Since *ERIS* aims at analytical workloads, the maximization of parallel processing is the main objective of the load balancer. Thus, there is no need for inter-data-object balancing, for instance to colocate certain partitions of different data objects on the same AEU as it is beneficial for transactional workloads. The *ERIS* adaption loop starts with the monitoring of different metrics on a per data object level. Based on the captured metrics, the load balancer periodically checks the load of *ERIS* for imbalances. If the standard deviation between the different AEUs exceeds a given threshold, the load balancer executes a load balancing algorithm that calculates a new target partitioning. With the help of the current and the targeted partitioning, the load balancer computes a series of balancing commands that are routed to the involved AEUs.

2.2 Summary

ERIS is a NUMA-aware purely in-memory storage engine for tera-scale analytical workloads that is based on a data-oriented architecture. *ERIS* uses a

NUMA-optimized high-throughput data command routing as well as a configurable NUMA-aware load balancing algorithm to achieve a maximum of parallelism to execute analytical queries with low latency. Our analysis in [5] shows that *ERIS* greatly improves memory locality and cache usage and thus scales even on large-scale NUMA platforms. Since *ERIS* only provides storage operation primitives, the next step is to implement a query processing framework on top of *ERIS* and to evaluate the performance of more complex queries. Query processing with *ERIS* requires techniques for distributed systems and poses additional challenges for load balancing. In particular, data partitioning is a key aspect to enable data parallelism for processing.

3 *ERIS*-Programmability for SQL Functionality

In order to support the creation of complex analytical workloads, *ERIS* needs a flexible but easy to use interface for its storage and processing capabilities. Instead of specializing *ERIS* to one specific database interface, we equip it with a general purpose programming interface which can be used as the basis for an implementation of different high-level interfaces like SQL. Many state-of-the-art databases rely on a set of hardcoded physical operators to execute SQL statements. Thereby, a SQL statement is transformed into a graph representation, the query plan, and that graph representation is subsequently interpreted to call adequate physical operators. *ERIS*, on the other hand, does not limit data processing to a fixed set of relational operators. Instead, *ERIS* provides a means to run user provided code on data but enforces a certain structuring of user code and control flow in order to enable efficient and parallel execution. In order to ideally exploit *ERIS*' data processing capabilities, the *ERIS* programming framework's main design objective is to support data locality and parallel execution of computations. To enable that, we require for each user function, an explicit description of the data partitions the user function has to be granted access to.

3.1 Partitioning Schemes

The *ERIS* programming framework is built around three data structures for the desired SQL functionality: `tuples`, `relations`, and `partitions`. Just like in typical databases, `tuples` are flat containers of data types like numbers, strings or dates. Both `relations` and `partitions` are sets of tuples with a fixed schema; but only `relations` can be named and loaded from *ERIS* by name. There is no direct way to access the tuples of a relation. Instead `relations` have to be partitioned first and subsequently the individual parts of the relation can be processed by accessing their tuples. Requiring relations to be partitioned before their tuples can be accessed, is the key to enabling data parallel processing in *ERIS*.

The current version of our *ERIS* programming framework supports three partitionings from a programming perspective as first-class citizens, which are necessary to support SQL functionality.

Individuals: The first partitioning scheme is called *individuals* partitioning and it simply turns every tuple of the target relation into its own partition. In equation 1, we define the individuals partitioning as a family of sets over a base relation $R_1 \times \ldots \times R_n$ where each element of the base relation forms its own subset.

$$\mathbf{Ind}(R) \;=\; \{\, \{t\} \,|\, t \in R\,\} \tag{1}$$

Equivalents: The second partitioning groups tuples having the same value in a fixed set of attributes. Again, the *equivalents* partitioning is a family of sets. This time, a subset contains all tuples which are equivalent to a representative tuple. The representative tuple is drawn from a selection on the base relation of the partitioning. Two tuples are considered equivalent if they have the same value in all attributes that they have in common.

$$\mathbf{Equiv}(R, a_1, \ldots, a_k) \quad = \quad \{\, \{t \,|\, t \;\in\; R \wedge t \;\equiv\; c\,\} \,|\, c \;\in\; \pi_{a_1,\ldots,a_k}(R)\,\} \tag{2}$$

All: The final partitioning does not partition its input relation at all but simply declares the whole relation as a single partition. *All* is needed because some algorithms do not lend themselves to the form of data parallelism promoted by partitionings. This is especially true for all kinds of aggregations that fold a complete relation into a single tuple or value.

$$\mathbf{All}(R) \;=\; \{\, R\,\} \tag{3}$$

3.2 Processing Functions for Partitioning Schemes

Once a relation is partitioned using one of our three partitioning schemes, the individual parts can be used for processing. The processing of tuples has to be encapsulated in a pure function, the so called *user function* which takes one or multiple partitions as parameter and returns a single partition. The framework implements this concept by embedding tuple processing capabilities in $process(P_1, \ldots, P_n, f\backslash n)$, a second-order function which is parametrized with a list of input partitionings and a first-order *user function*. The arity of the *user function* always has to match the number of partitionings supplied to *process*.

In the simple case, $process(P, f\backslash 1)$ is applied to a single relation P and a *user function* $f\backslash 1$ with arity 1. In that case, *process* applies $f\backslash 1$ to each subset of partition P, creates a new relation by unioning the results of each application of $f\backslash 1$, and finally returns the new relation as overall result of the second-order function.

$$process(P_1, \ldots, P_n, f\backslash n) = \bigcup_{p_1 \in \mathbf{P_1}} \ldots \bigcup_{p_n \in \mathbf{P_n}} f(p_1, \ldots, p_n) \tag{4}$$

Equation 4 shows the generalization of *process* to multiple input partitions. In this case, $process(P_1, \ldots, P_n, f\backslash n)$ applies $f\backslash n$ to all possible combinations of subsets of the input partitionings. Again, the results of all applications are

Listing 1.1. Function for computing the relational projection

```
def projection(r: Relation, attrs: Attribute*): Relation = {
  val d = process(r.individuals) { partition =>
    val tuples = for (tuple <- partition) yield {
      new Tuple(attrs.map(a => tuple(a))
    }
    return new Partition(tuples)
  }
  process(d.equivalents(attrs)) { partition =>
    // eliminate duplicates
    return new Partition(partition.randomTuple())
  }
}
```

unioned into a new relation which is the final result of $process(P_1, \ldots, P_n, f\backslash n)$. Independent of the number of partitionings, $f\backslash n$ has to map its inputs to a single output schema or it will break the unioning of output partitions. In this way, $process(P_1, \ldots, P_n, f\backslash n)$ completes the story of data-parallelism in *ERIS* by enabling the parallel processing of multiple elements of one or more partitionings.

3.3 Realization of the Relational Algebra for SQL functionality

To demonstrate the adequacy of our presented framework, we use it to outline an implementation of the relational algebra on *ERIS*, whereas our language approach is based on Scala.

Projection: $\mathbf{P} = \boldsymbol{\pi_{a_1,\ldots,a_k}(R)}$ The relational projection requires a two step approach: first unwanted attributes are removed from the individual tuples and then duplicates are removed from the resulting relation.

Listing 1.1 shows the Scala code for that operation. The first *process* is applied to the individuals of the target relation R. The user function simply iterates the partition, extracts the relevant attributes from each tuple and uses those attributes to create a new tuple. Finally, all tuples are combined into a new partition and that partition is returned as a result of the user function. It has to be noted that in this case, because the partition only ever contains a single tuple, the more convenient parameter to the user function would be a simple tuple instead of a partition. But for the sake of generality we avoid the introduction of special cases at this point.

The second *process* application is necessary to eliminate duplicates which can be introduced when attributes are removed from relations. Duplicate elimination is not an automatic feature of the underlying `relation` data structure as it mapped onto *ERIS* containers which do not exhibit this feature. The idea is to perform an equivalents partitioning on all attributes of the relation. In this setup each partition can only hold either a single tuple or multiple duplicated tuples

Listing 1.2. Function for computing the cartesian product

```
def cross(r: Relation, s: Relation): Relation =
  process(r.individuals, s.individuals) { (p1, p2) =>
    val tuples = for (tuple1 <- p1; tuple2 <- p2) yield {
      new Tuple(tuple1.attributes ++ tuple2.attributes)
    }
    return new Partition(tuples)
  }
```

as all tuples in the partition have to be equal in all attributes. Most of the work of removing duplicates is of course done by the partitioning function itself. The user function simply selects a random element of the input partition and creates a new partition with the one element as only content.

Selection: $S = \sigma_P(R)$ The selection can be performed on a per tuple basis and therefore, a single processing of individuals is sufficient. The user function tests for each tuple whether it satisfies the predicate and creates the output partition with all tuples that have passed the test.

Cartesian Product: $P = R \times S$ Due to *process*' support for multiple input partitionings, the implementation of the cartesian product in listing 1.2 is straightforward. *process* is applied to the *individuals* partitionings of the relations R and S. Therefore, the *user function* is applied once on every possible pair of input tuples and merely has to combine the attributes of these tuples into a single output tuple and pack that tuple into a new partition.

Union: $U = R \cup S$ Set union is a typical example of an operation that is hard to parallelize with the partitioning approach as there is minimal independent per element computation. Listing 1.3 shows a simple solution relying on the *all* partitioning and simply concatenates all tuples of R with all tuples of S. As with the relational projection, this process can introduce duplicate entries which have to be removed in a second step.

Difference: $D = R \backslash S$ The set difference operator can be implemented as multiple parallel set inclusion tests using an *individuals* and an *all* partitioning. In listing 1.4 each tuple of R is combined with all tuples of S and the user function performs a single set inclusion test to check if the tuple has to be included in the output relation.

4 Execution and Complex Query Example

Our *ERIS* programming framework follows recent proposals [4, 6] to connect best performance with high programmer productivity by relying on domain specific

code generation. As already shown in the previous section, users of our framework write their programs in Scala and rely on a set of framework operations to interact with *ERIS*' storage and processing resources. The framework's operations are our partitioning schemes. In Section 4.1, we introduce our execution approach, followed in Section 4.2 by a short example of a complex query.

4.1 Execution

The overall compilation of programs written in Scala and using our *ERIS* framework is driven by the translation of partitionings and the $process(P_1, \ldots, P_n, f\backslash n)$ function as these constructs are directly mapped onto invocations of *ERIS*' native storage operations like `ScanTable` to scan a table. The `ScanTable` operation is responsible for all reading activitivies in *ERIS*, whereas the `ScanTable` can be parameterized with a single data container and a callback function being called whenever it has collected a chunk of records from the target container. The callback processes the records and pushes the resulting data to a new container using an insert operation. `ScanTable` invokes the callback with chunks of records until all records of the target container have been processed.

A single execution of `ScanTable` in its basic form is sufficient to translate an *execute* over a single *individuals* partitioning. The `ScanTable`'s callback is created with a loop that iterates the individual records of the input chunk. The body of the loop contains the translation of the *user function*, which is thereby applied on each individual record. At the end of each loop, an insert message materializes the result of the loop body into an output container.

The translation of an *execute* over an *equivalents* or an *all* partitioning of a single relation is of a similar character. `ScanTable` can be parametrized to call its callback either with all records that fulfill certain requirements or simply with all records of a container. These parametrization options exactly match the semantics of the *equivalents* and *all* partitionings, so the translation of both partitionings is a straightforward generation of `ScanTable` parameters. Requiring the grouping of certain or all records of a container has of course a negative impact on the locality of data access and should be used as sparingly as possible. Because they are already called with the correct set of records, the callback func-

Listing 1.3. Function for computing the set union

```
def union(r: Relation, s: Relation): Relation = {
  val d = process(r.all, s.all) { (p1, p2) =>
    return new Partition(p1.tuples ++ p2.tuples)
  }
  process(Ud.equivalents(r.attributes ++ s.attributes))
    { partition =>
      return new Partition(partition.randomTuple())
    }
}
```

Listing 1.4. Function for computing the set difference

```
def difference(r: Relation, s: Relation): Relation =
  process(r.individuals, s.all) { (p1, p2) =>
    val tuples = for (tuple <- p1 if !p2.contains(p1)) yield {
      tuple
    }
    new Partition(tuples)
  }
```

tions for *equivalents* and *all* do not need an inner loop to iterate over records. They simply contain the cross-compiled code of the respective user function and at the end an insert message to materialize the output tuples of the user function.

The code for a multi relation *execute* is a little bit more involved because a `ScanTable` will only ever touch one single *ERIS* container. Combining data from containers $C_1, \ldots, C_n$ requires n sequential `ScanTable` runs, one on each of the containers. As an example, we will examine a *process* over three *individuals* partitionings P_1, P_2, P_3. Processing will start with a `ScanTable` on P_1 with a callback that iterates over the individual records. For each record the callback will request a new `ScanTable` over P_2 and send the current record as additional data to the callback of the new scan. Each callback over P_2 will in turn iterate over the individual records of P_2, request a third `ScanTable` on P_3, and send the records from P_1 and P_2 as additional data to the callback of the scan over P_3. The callback of the final scan iterates once again over the records of its container, combines each one with the other two records, and hands them to the code of the user function in its iterator loop. Similar to the earlier examples, the result records produced by the user code have to be inserted into a new container at the end of the loop. The algorithm for three *individuals* partitionings can be easily generalized to other types of partitionings and relation counts, so we abstain from a more detailed description at this point.

4.2 Complex Query Example

We finish our discussion of the *ERIS* programming framework with a high level example that combines multiple relational operators to implement the simple SQL query depicted in listing 1.5.

Listing 1.5. A simple SQL example

```
SELECT student.name grade.course grade.grade
FROM student, grade
WHERE student.id = grade.studentID AND student.age > 30;
```

Listing 1.6 shows a straightforward translation from SQL to relational operators. Each operator is annotated with the *process* calls it requires and with the number of *ScanTable* operations necessary to execute the *process* calls in *ERIS*.

Listing 1.7. Optimized SQL translation. 3 x ScanTable

```
// Two input relations, 2 x ScanTable
val nonUnique = process(students.individuals, grades.individuals) {
    // join student and grade
    // select on joined
    // project on selected
}

// One input relation, 1 x ScanTable
val result = process(nonUnique.equals(name, course, grade) {
    // filter duplicates
}
```

In total, the direct translation of the SQL statement requires 5 *ScanTable* operations: 2 for the *cross*, 1 for the *select*, and 2 more for the *projection*.

Listing 1.6. Naive SQL translation. 5 x ScanTable

```
// 2 x ScanTable
// c = process(students.individuals, grades.individuals)
val c = cross(students, grades)

// s = process(c.individuals), 1 x ScanTable
val s = select(c, { t: Tuple =>
    return t("student.id") == t("grade.studentID")
        && t("student.age") > 30})

// nu = process(s.individuals), 1 x ScanTable
// result = process(nu.equivalents(...)), 1 x ScanTable
val attrs = ["student.name", "grade.course", "grade.grade"]
val result = project(s, attrs)
```

The straightforward translation provides a good opportunity for an important domain specific optimization. The first *process* joins the students and grades tuples and the two following *process* invocations process the *individuals* partitioning of the crossed relations. The individual tuples of the crossed relations are already materialized in the first *process* invocation. Therefore, we are able to append the bodies of the second and third *process* to the body of the first *process* and do all work in a single *process* invocation. This optimization is a good example of what is possible with a compiler that can be extended with domain specific knowledge. Listing 1.7 shows an outline of the optimized code, where the cross, select, and the first part of the project operations are fused into a single *process* invocation. The optimized version of the code reduces the number of required *ScanTable* operations to three and avoids two costly materializations of intermediate data containers.

5 Future Work and Conclusion

In this paper, we have presented our programmability idea for an in-memory storage engine which is based on a data-oriented architecture. Instead of relying on hardcoded physical operators, our approach introduces data partitionings as first-class citizens on the programming layer as second-order functions, which can be parameterized using any kind of first-order functions. Fundamentally, this approach is similar to the PACT programming model [3], whereas our concept is not limited to the key-value data format. Based on our overall idea, we want to extend our work in different directions. First, we want to build a more user friendly domain specific language (DSL) based on our partitioning schemes for SQL. Second, we are going to map different languages like SQL or Pig to our DSL to support these higher-level languages in an efficient way. Third, we will investigate further application domains to find additional specific partitioning schemes.

References

1. Abadi, D., Agrawal, R., Ailamaki, A., Balazinska, M., Bernstein, P.A., Carey, M.J., Chaudhuri, S., Dean, J., Doan, A., Franklin, M.J., Gehrke, J., Haas, L.M., Halevy, A.Y., Hellerstein, J.M., Ioannidis, Y.E., Jagadish, H.V., Kossmann, D., Madden, S., Mehrotra, S., Milo, T., Naughton, J.F., Ramakrishnan, R., Markl, V., Olston, C., Ooi, B.C., Ré, C., Suciu, D., Stonebraker, M., Walter, T., Widom, J.: The beckman report on database research. SIGMOD Rec. 43(3), 61–70 (Dec 2014)
2. Agrawal, R., Ailamaki, A., Bernstein, P.A., Brewer, E.A., Carey, M.J., Chaudhuri, S., Doan, A., Florescu, D., Franklin, M.J., Garcia-Molina, H., Gehrke, J., Gruenwald, L., Haas, L.M., Halevy, A.Y., Hellerstein, J.M., Ioannidis, Y.E., Korth, H.F., Kossmann, D., Madden, S., Magoulas, R., Ooi, B.C., O'Reilly, T., Ramakrishnan, R., Sarawagi, S., Stonebraker, M., Szalay, A.S., Weikum, G.: The claremont report on database research. SIGMOD Rec. 37(3), 9–19 (Sep 2008)
3. Alexandrov, A., Battré, D., Ewen, S., Heimel, M., Hueske, F., Kao, O., Markl, V., Nijkamp, E., Warneke, D.: Massively parallel data analysis with pacts on nephele. PVLDB 3(2), 1625–1628 (2010)
4. Brown, K.J., Sujeeth, A.K., Lee, H., Rompf, T., Chafi, H., Odersky, M., Olukotun, K.: A heterogeneous parallel framework for domain-specific languages. In: Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on. pp. 89–100. IEEE (2011)
5. Kissinger, T., Kiefer, T., Schlegel, B., Habich, D., Molka, D., Lehner, W.: ERIS: A numa-aware in-memory storage engine for analytical workload. In: International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures. pp. 74–85 (2014)
6. Klonatos, Y., Koch, C., Rompf, T., Chafi, H.: Building efficient query engines in a high-level language. Proceedings of the VLDB Endowment 7(10), 853–864 (2014)
7. Pandis, I., Johnson, R., Hardavellas, N., Ailamaki, A.: Data-oriented transaction execution. PVLDB 3(1-2), 928–939 (2010)