Johannes Luong, Dirk Habich, Wolfgang Lehner

A Technical Perspective of DataCalc — Ad-hoc Analyses on Heterogeneous Data Sources

Erstveröffentlichung in / First published in:

IEEE International Conference on Big Data. Los Angeles, 9.-12. Dezember 2019, S. 3864–3873. IEEE. ISBN 978-1-7281-0858-2.

DOI: https://doi.org/10.1109/BigData47090.2019.9006029

Final edited form was published in "IEEE International Conference on Big Data". Los Angeles, 9.-12. Dezember 2019, S. 3864–3873. IEEE. ISBN 978-1-7281-0858-2
https://doi.org/10.1109/BigData47090.2019.9006029

A Technical Perspective of DataCalc — Ad-hoc Analyses on Heterogeneous Data Sources

1st Johannes Luong
Database Systems Group
Technische Universität Dresden
Dresden, Germany
johannes.luong@tu-dresden.de

2nd Dirk Habich
Database Systems Group
Technische Universität Dresden
Dresden, Germany
dirk.habich@tu-dresden.de

3rd Wolfgang Lehner
Database Systems Group
Technische Universität Dresden
Dresden, Germany
wolfgang.lehner@tu-dresden.de

***Abstract*—Many organizations store and process data at different locations using a heterogeneous set of formats and data management systems. However, data analyses can often provide better insight when data from several sources is integrated into a combined perspective. *DataCalc* is an extensible data integration platform that executes ad-hoc analytical queries on a set of heterogeneous data processors. The platform uses an expressive function shipping interface that promotes local computation and reduces data movement between processors. In this paper, we provide a detailed discussion of the architecture and implementation of *DataCalc*. We introduce data processors for plain files, JDBC, the MongoDB document store, and a custom in memory system. Finally, we discuss the cost of integrating additional processors and evaluate the overall performance of the platform. Our main contribution is the specification and evaluation of the *DataCalc* code delegation interface.**

I. Introduction

Big data defines the shape and challenges of modern day data processing. For instance, capable and cheap hardware architectures enable new applications that use physical and virtual sensors to produce unprecedented amounts of data. Therefore, many professions depend on large scale data analysis to drive research and to guide decision making. This "democratization" of data processing has led to a much more diverse set of requirements that have to be considered by state-of-the-art data management systems. In that sense, the influential *Beckman Report on Database Research* [Abadi et al., 2016] already identifies "coping with diversity in data management" as one of five primary challenges in big data. The reason is that novel big data applications often make use of data types that are not easily translatable to the traditional relational model. Therefore, data is stored in a large variety of formats and gets processed by a myriad of systems with special purpose application programming interfaces. In this environment it quickly becomes difficult to analyze data across the boundaries of specialized big data systems.

To tackle this challenge, we recently proposed the data integration platform *DataCalc* that enables a set of heterogeneous data processors to collaborate in the execution of analytical workloads [Luong et al., 2019]. *DataCalc* translates queries into the novel intermediate representation *DC* that provides a unified and extensible high-level representation for a large number of data analysis applications. *DC* is a small and highly structured functional programming language whose domain specific built-in functions and limited number of syntactic forms make it an excellent target for algebraic rewrites, such as relational optimizations. *DC* expressions are delegated to data processors which carry out computations on their local data stores. The term *data processor* encompasses any system that can execute at least one *DC* function and return its result to the *DataCalc* runtime. This broad description captures a diverse set systems, such as traditional DBMS, *big data* cluster processing runtimes, special purpose accelerators, and many other data processing system that offer some external application programming interface (API). Processors can accept or reject *DC* functions on a case to case basis. This affords a high degree of flexibility in the creation of new processors and in adapting the runtime behaviour of existing ones. A new processor can start off with support for basic data loading functions and if the need arises, additional capabilities such as filters and projections can be added later on.

Our Contribution and Outline.

In this paper, we present a technical in-depth discussion of our novel extensible data integration platform *DataCalc* and show how this platform can be used to execute SQL queries. SQL and the relational algebra are well known in the *big data* community and therefore we see it as a good starting point for our exploration of *DataCalc* and its novel intermediate representation *DC*. In particular, we make the following contributions:

1) We describe a flexible data integration model that delegates query processing to attached processors.
2) We introduce the novel intermediate program representation *DC* that facilitates domain specific optimization.
3) We evaluate our system with regard to the cost of developing additional data processors and its runtime performance on a set of star schema queries.

The remainder of the paper is structured as follows: In Section II, we discuss related works from industry and research. In Section III, we introduce the main components of *DataCalc*. Based on that, we investigate several data processors in Section IV. Afterwards, Section V contains the evaluation and in Section VI we conclude the paper with a summary of our contributions.

Final edited form was published in "IEEE International Conference on Big Data". Los Angeles, 9.-12. Dezember 2019, S. 3864–3873. IEEE. ISBN 978-1-7281-0858-2
https://doi.org/10.1109/BigData47090.2019.9006029

II. Related Work

Apache Calcite [Begoli et al., 2018] is a framework for query optimization and federated query execution with a strong emphasis on extensibility. The framework includes a SQL compiler that translates queries into an internal tree representation, planners that apply tree rewrites to optimize queries, and a lightweight execution model that delegates parts of queries to available execution engines. Each of these components is open to modification. The internal query representation can be extended with new functions and operators, optimization can be augmented with additional rewrites, and new engines can be integrated into the execution model. As will become clear in the subsequent sections, Calcite has had a direct and strong influence on the design of *DataCalc*. At this point, the primary difference between the two systems are the intermediate representations that are used to represent queries internally. Calcite's intermediate representation and query planner are currently hard-wired to static relational schemas. That is, operations have to be expressed in terms of relations and column expressions and data sources have to provide schema information for their data objects. *DC*, on the other hand, is not limited to any particular application domain and does not require static schemas. Instead, it can represent any abstract data type by including its functions as built-ins of the language. Typing is delegated to processors which can, optionally, use their internal catalogs to infer types for DC expressions. In this sense, we propose *DataCalc* as a research project that explores how a system such as Calcite could benefit from a more open-ended query representation.

The combination of several data processing domains in a unified system is a primary goal of *DataCalc*. A prominent example for the attention that this topic is receiving in the industry is the PartiQL query language that has been recently announced by Amazon AWS [Papakonstantinou et al., 2019] as "one query language for all your data". PartiQL is a SQL-compatible language that can access structured, semi-structured, and nested datasets on a growing number of AWS data storage and management systems. According to the announcement, the main objective for developing the language is to decouple application logic from format specific data accessing methods. Applications that use PartiQL can replace their physical data sources without modifying their source code. This facilitates the flexible evolution of applications in the AWS ecosystem. However, in contrast to *DataCalc*, PartiQL is *just* a query language that is supported by several independent data sources. Each new source has to provide an implementation of the language and ad-hoc cross-source data integration, a core feature of *DataCalc*, is not (yet) available.

Similar to PartiQL, the goal of Musketeer [Gog et al., 2015] is to break the coupling between applications and specific data processing technology. Many *big data* processing engines, such as Hadoop[1] and Spark[2], offer their own specialized higher-level programming abstractions that create a tight coupling with the applications that use them. Musketeer breaks this bond by providing an M to N mapping between system specific languages and execution engines. Using Musketeer, an application written in one of these languages can be executed on all available systems. To limit the complexity of this mapping, Musketeer relies on a typical compiler architecture with multiple front-ends, a common intermediate representation, and a set of back-ends that generate executable code. We see *DataCalc* as an alternative to Musketeer with an intermediate representation that is better suited for domain specific optimization and that does not depend on a data-flow execution model. For a more in-depth discussion of the relation between *DataCalc* and other approaches we refer to a previous article of our group [Luong et al., 2018].

[1]https://hadoop.apache.org
[2]https://spark.apache.org

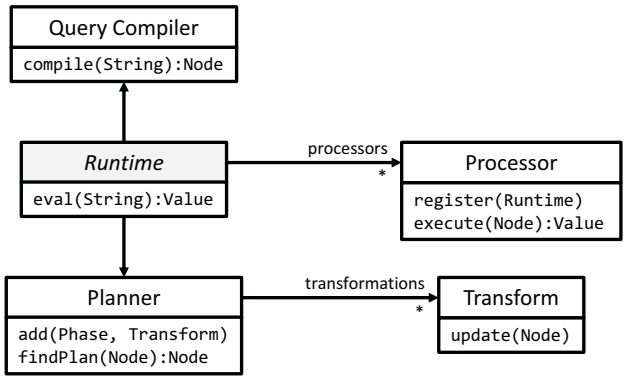


Fig. 1: DataCalc platform components

III. *DataCalc*

DataCalc is an extensible platform for ad-hoc data analyses on a set of heterogeneous data processors. It uses an expressive internal program representation and a flexible evaluation model to ship analytical workloads to various data processing engines. This allows *DataCalc* to use the native compute resources of connected processors and to reduce the amount of data that is transferred between those processors. Figure 1 shows the primary software components of the *DataCalc* platform in an UML style class diagram. The `Runtime` component serves as central coordinator of the platform and its `eval` function is the entry point for query execution. The first step of the execution process is the compilation of a SQL string into its *DC* representation. This is accomplished by the `compile` function of the `Query Compiler`. Internally, *DC* programs are represented as directed acyclic graphs and `Node` is the base type of this graph representation. Next, the `Runtime` passes the *DC* program to the `findPlan` function of the `Planner` which applies a set of transformations to adapt and optimize the query to the available data processors. One important type of transformation is the *Assignment* which assigns function applications of the *DC* program to available processors. These *Assignments* are added to the `Planner` by the `Processors` when they are first registered with the `Runtime`. Once `findPlan` returns, the `Runtime` schedules

Final edited form was published in "IEEE International Conference on Big Data". Los Angeles, 9.-12. Dezember 2019, S. 3864–3873. IEEE. ISBN 978-1-7281-0858-2
https://doi.org/10.1109/BigData47090.2019.9006029

assigned *DC* expressions for execution by passing them to the `execute` function of their `Processor`. Eventually, the outer-most function of the *DC* program is executed and the `Runtime` returns the result of that final `execute` call as result of the query.

A primary objective of *DataCalc* is to encourage the integration of many different data processors into the platform. All that is required to add a new processor, is an implementation of the `Processor` interface and a `Transform` that assigns program nodes to the new processor. The `Processor` and `Transform` interfaces are slim and abstract to enable a wide range of implementations and to allow processors fine grained control over which parts of a program they want to execute. For example, processors are not required to provide any kind of metadata, such as a schema. Further, the `execute` function enables a range of evaluation models by hiding data access behind an abstract `Value` type. Processors can return lightweight result values right-away and internally postpone processing until the contents of those values are actually required. Finally, the *Transform* interface gives processors detailed control over the program elements that they want to execute. A processors can even replace a call with an equivalent composite expression, assign itself one element of this composite, and leave the remaining elements for other processors. For example, a processor that only implements a subset of SQL's expression language can replace a `FILTER` that uses some supported and some unsupported sub-expression with two nested `FILTER`s, one of which is fully supported and can be assigned.

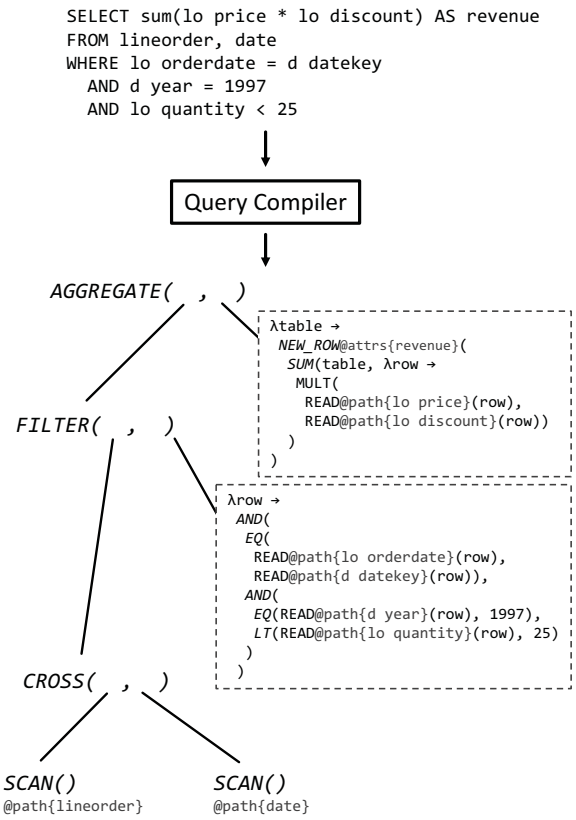


Fig. 2: Query compilation

A. Query compilation and the DC language

The query compiler accepts SQL queries and translates them into *DC*, the internal functional language of *DataCalc*. *DC* is a pure, highly structured functional language with a large number of built-in functions that can represent various processing domains, such as the relational algebra. The following table shows some examples of *DC* built-ins for several data types and processing domains. Please note that the graph, linear algebra, and recursion built-ins are future work, they are included here only to demonstrate the flexibility of the representation.

Boolean	`GT`, `EQ`, `LT`, `AND`, `OR`, `NOT`, ...
Math	`PLUS`, `MINUS`, `MULT`, `DIV`, ...
Relational	`SCAN`, `CROSS`, `FILTER`, ...
Property Graph	`MATCH`, `NODE`, `REL`, ...
Linear Alg.	`SOLVE`, `TRANSPOSE`, ...
Recursion	`ITERATE`, `UNFOLD`, `FOLD`, ...

The primary goal of *DC* is to provide an easy to adopt, expressive, and future proof basis for *code shipping* in data processing systems. We define code shipping as the process of moving application logic between independent processing systems, with the goals of preventing data movement and exploiting distributed compute resources. *DC* programs do not have a dedicated textual representation, but are directly constructed in-memory. However, if an external form is required DC programs can be serialized to - and parsed from JSON.

Figure 2 shows a simple SQL query and the *DC* code that the compiler generates for this query. In *DC*, the whole query is represented as a single nested expression. Most elements of that expression are applications of built-ins and some of these applications accept a local function definition as one of their arguments. To emphasize the logical structure of the query, we have pulled apart some elements of the expression into a tree like form. The local function definitions, on the other hand, are represented as lambda expressions with a nested body. Some of the function applications are annotated with additional static information. This is represented by an `@`-sign followed by the name of the annotation, and some values in curly braces. For example, many built-ins are annotated with static paths that identify the data objects and attributes that will be read by these calls.

The `Query Compiler` maps different SQL clauses to their corresponding *DC* built-ins. Figure 2 gives an impression of this process. The `FROM` clause is mapped to a `CROSS` call with two `SCAN` calls as arguments. Next, the `WHERE` clause is mapped to `FILTER` which takes a table and a predicate as arguments. The table is just the result of the `CROSS` call and the predicate is provided by a local function definition. The predicate function consists of a single parameter, `row`, and

Final edited form was published in "IEEE International Conference on Big Data". Los Angeles, 9.-12. Dezember 2019, S. 3864–3873. IEEE. ISBN 978-1-7281-0858-2
https://doi.org/10.1109/BigData47090.2019.9006029

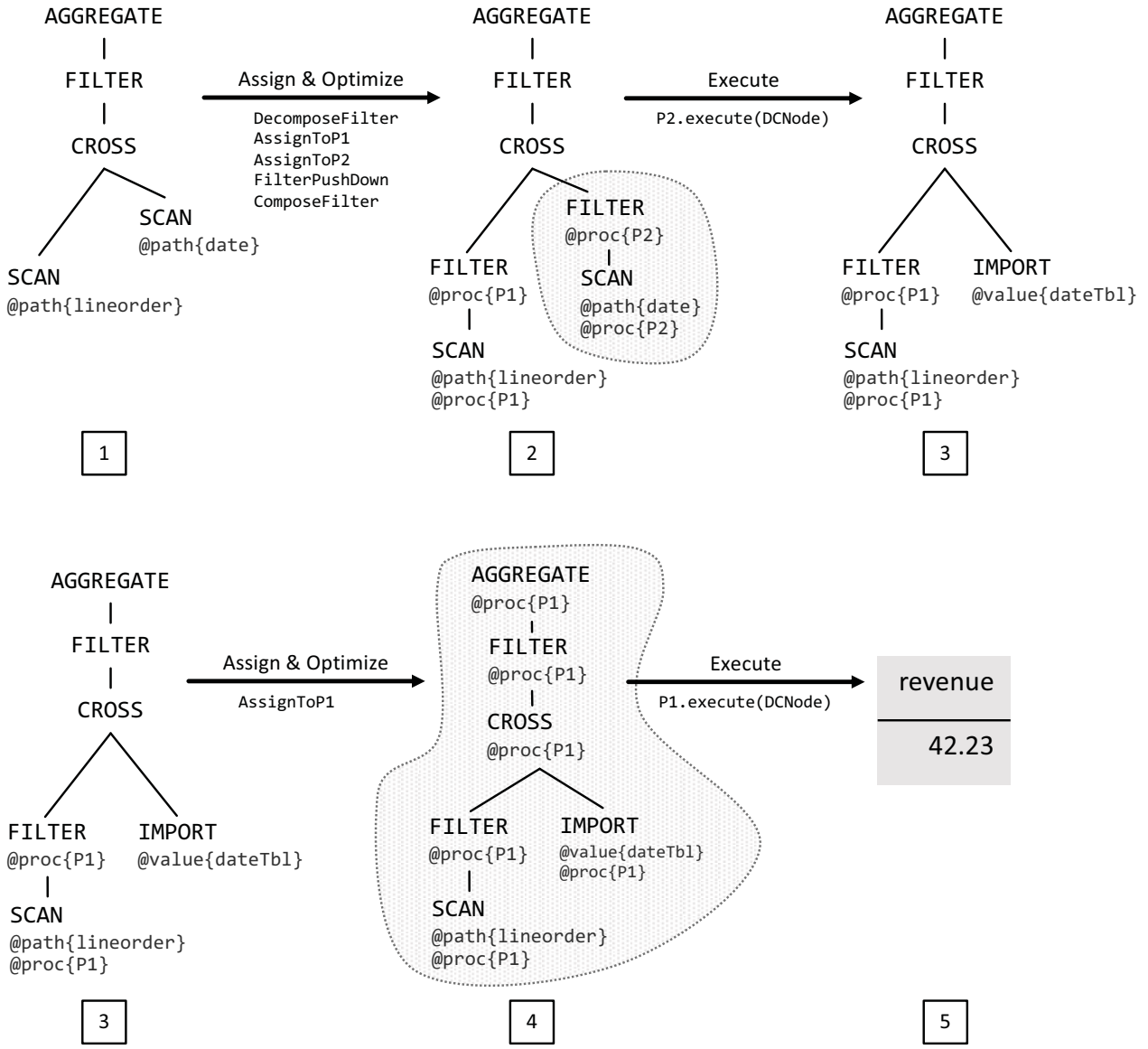


Fig. 3: Query planning and execution

a nested body expression that reads from `row` to compute a boolean which determines whether the row is accepted into the result of the `FILTER` call. Finally, the `SELECT` clause is mapped to the `AGGREGATE` built-in.

B. Planning

In *DataCalc*, planning is the part of query execution where a *DC* program is logically assigned and adapted to concrete available processors. The process of planning consists of applying rewrites to the internal DAG representation of a *DC* program. Common tasks of rewrites are to replace expressions with semantically equivalent but somehow preferred alternative expressions, to attach additional annotations to the program, or to perform checks on the program. Rewrites are applied to the *DC* program by the `Planner`. Some rewrites, such as `PushDownFilter` and `CombineJoins`, are independent of any particular processor and are added to the `Planner` by the `Runtime`. Others, such as the important `AssignTo`*X* rewrites, are added by processors when they are registered with the runtime. Currently, we use a simple fixpoint planner that repeatedly applies its rewrites until none of them produce further updates to the *DC* program. A more effective cost based planner remains future work.

The main objective of planning is to effectively use the compute abilities of available processors and to prevent large scale data shipping. Usually, this is accomplished by delegating as large as possible *DC* subexpressions in one go. An example of this process is depicted in Figure 3. The figure shows, in five stages, the planning and evaluation of the example program

Final edited form was published in "IEEE International Conference on Big Data". Los Angeles, 9.-12. Dezember 2019, S. 3864–3873. IEEE. ISBN 978-1-7281-0858-2
https://doi.org/10.1109/BigData47090.2019.9006029

that we have already discussed in Section III-A. For the sake of simplicity, we left out the lambda parameters and concentrate on the main processing functions. The first version of the program, in the top-left corner, shows the output of the query compiler as it is submitted to the planner. Then, the planner successfully applies five rewrites to generate the second version. The iterative planner distinguishes four phases: *Prepare*, *Assign*, *Optimize*, and *Clean-up* and it applies the rewrites of each phase repeatedly until they do not update the program any more. The `DecomposeFilter` rewrite belongs to the *Prepare* phase and it replaces `FILTER` calls that have a top-level conjunction in their predicate with two nested `FILTERS` without a conjunction. The purpose of this preparation is to improve the results of the subsequent `FilterPushDown`. `ComposeFilter` belongs to the *Clean-up* phase and implements the opposite effect of `DecomposeFilter` with the goal to reduce the number of traversals in processors with limited internal query optimization. The `AssignToX` rewrites are applied in the *Assign* phase. These rewrites mark *DC* nodes that can be executed by their processor by attaching a `@proc` annotation. In the case of `SCAN` calls, processors can use the `SCAN`'s `@path` annotation to determine whether they own the referenced object. For other built-ins, processors usually check whether they implement the function and whether the arguments of the application are already assigned to themselves. The fix-point planner applies rewrites in reverse order in which they were added. That is, if two processors can assign a node, the processor that was registered last, will always receive the call. This approach is another potential shortcoming of the fix-point planner that we leave for future works. The `@proc` annotation has to implement a `matches(Path):Bool` function that can be used to detect whether some `@path` annotation belongs to the processor. This path matcher is used by rewrites in the *Optimization* phase. For example, `FilterPushDown` uses the matcher to decide whether a `FILTER` call can be moved into a branch of a `CROSS` call. The specific semantics of path matching can be defined by each processor. The only requirement is that the matcher can detect all paths that belong to its processor. Some processors, such as SQL databases, can use their internal catalogs to implement convenient column name short-hand notations. Others will require paths that are qualified with some sort of processor ID.

In the second version of the program, no more nodes can be assigned because the arguments of the `CROSS` call are assigned to different processors. At this point, one of the assigned branches is selected and passed to the `execute` function of its processor. We will discuss execution in greater detail in subsequent sections. It is sufficient to notice that, in the third version of the program, the highlighted branch of Processor P2 is replaced with an `IMPORT` call that is annotated with the result of the execution. At this point, the planner restarts the rewriting loop which eventually generates the fourth version of the program. This time around, `AssignToP1` is the only rewrite that can be applied successfully. Fortunately, Processor P1 provides an implementation of the `IMPORT` function and therefore, `CROSS` and the remaining calls can all be assigned to P1 and the whole program can be executed in one final go.

C. Execution

Execution is triggered by the planner whenever no more nodes can be assigned. This is usually the case, if the whole program is assigned or if the arguments of some built-in calls are assigned to different processors. The case where no nodes are assigned after planning represents an error. In principle, the planner could execute all assigned subtrees in parallel. However, after some initial testing, we chose a different strategy: the planner executes a single assigned subtree, replaces that tree with an `IMPORT` call that is annotated with the result of the execution, and restarts the planning loop. This process is shown in Figure 3 in the transition from version 2 to version 3. For the example query, this approach is much more efficient than the parallelized alternative as the `date` table is much smaller than the `lineorder` table. Once P2 is finished, the small `date` table can be quickly imported into P1 and then the whole remaining query can be executed at once. The parallel approach, on the other hand, would schedule a very expensive `SCAN-FILTER` plan on P1 that results in a large intermediate value. Of course, if there is a choice, the planner has to decide which subexpression is executed first. Right now this is decided according to a manually defined processor priority list. The definition of a more elaborate optimization scheme is left for future work.

For the most part, the actual data processing happens in external systems and it is up to the processor components to translate *DC* subexpressions into executable code for their back-ends. We will discuss several examples of this process in Section IV. However, one important aspect of execution that is common to all processors is the movement of data between processors. We have already discussed how executed expressions are replaced with an `IMPORT` call that is annotated with the result of the expression and how processors can capture the `IMPORT` by assigning it to themselves. But how does the actual data movement happen? In general, of course, processors can implement data imports in any way they see fit. The `Value` type that is returned by `execute` does not define any operations, however, processors can use Java's runtime type information system to detect particular sub-types of `Value` and implement custom data movement logic for these sub-types. This can be especially useful when values have to be moved between systems that already define some common data exchanging system. However, to provide a general fallback solution, *DataCalc* defines the `Value` sub-type `TableValue` that has to be implemented by all result values of table processing built-ins. `TableValue` implements a lazy batch-iterator interface that uses Apache Arrow[3] as internal data format. Using this interface, processors can implement generic data importing logic that works for all table intermediate values. It should be noted that `TableValue`

[3]https://arrow.apache.org

Final edited form was published in "IEEE International Conference on Big Data". Los Angeles, 9.-12. Dezember 2019, S. 3864–3873. IEEE. ISBN 978-1-7281-0858-2
https://doi.org/10.1109/BigData47090.2019.9006029

```
class AssignToLocal extends Rerwite {
  boolean updateApplyBuiltIn(ApplyBuiltIn apply) {
    switch (apply.getName()) {
      case FILTER: case SORT: case LIMIT: {
        var matcher = Proc.getMatcher(apply.getArg(0));
        Proc.set(apply, Local.ID, matcher);
        return true;
      }
      case SELECT: case AGGREGATE: case GROUP_BY:
        // ...
    }
    return false;
  }
}

class LocalSelect implements TableValue {
  TableValue inTable;
  GraalFunc projectFunc;

  @Override
  BatchIterator iterator() {
    return new SelectIterator();
  }

  class SelectIterator implements BatchIterator {
    BatchIterator inIt = inTable.iterator();
    VectorSchemaRoot outBatch = ...;

    VectorSchemaRoot loadNextBatch() {
      VectorSchemaRoot inBatch = inIt.loadNextBatch();
      if (inBatch == null) return null;

      RowPointer rowPtr = new RowPointer(inBatch);
      while (rowPtr.next()) {
        // delegate to Truffle interpretation
        Object row = projectFunc.call(rowPtr);
        ArrowHelper.append(outBatch, row);
      }
      return outBatch;
    }
  }
}
```

Fig. 4: Components of the *Local* processor

currently requires all data to be moved through the runtime process. Direct data exchange between external engines is not supported in this approach.

IV. Data Processors

Data processors perform the actual computational work in *DataCalc*. Each processor is represented by a driver object that implements the `Processor` interface and that is registered with the `Runtime`. In addition, each processor has to provide a `Rewrite` that captures DC expressions during planning.

A. Local processor

The *Local* processor provides implementations of most DC built-ins as part of its driver component. That is, all processing takes place in the runtime process. *Local* does not store data and therefore does not implement the `SCAN` function. The runtime implicitly registers *Local* as first processor. Therefore, it has the lowest assignment priority and will only execute built-ins that are not accepted by any other processor.

Figure 4 shows parts of *Local*'s assignment rewrite and of its `SELECT` built-in. The rewrite uses the name property of the `ApplyBuiltIn` object, a sub-class of `Node`, to identify the function that is being called. Next, in case of a `FILTER`, `SORT`, or `LIMIT`, it retrieves the path matcher of the call's table argument and then sets a `@proc{Local}` annotation for the call. For other built-ins, the assignment logic only differs with regard to the path matcher. For example, a `SELECT` call defines a completely new set of paths and does not reuse the matchers of its arguments. Notably, *Local* does not check the processor assignment of its function arguments. This is the case because *Local* can operate on the result values of all processors. The `LocalSelect` class is *Local*'s implementation of the `SELECT` built-in function. The class implements the `TableValue` interface (see Section III-C) and accepts another `TableValue` as its input. The fact that the built-ins of *Local* operate on `TableValue`s explains why they can be directly applied to the results of any other processor, which all have to implement this interface. Actual data processing is postponed until a `SelectIterator` is created and consumed by some subsequent operation. At that point, the iterator reads rows from the input table, invokes the `projectFunc` to map input to result rows, and appends those rows to the result batch. The `projectFunc` object is an executable version of the function parameter of `SELECT`. *Local* uses the Truffle language implementation framework[4] to compile DC function definitions into efficient executable code.

B. File Processor

The *File* processor is an extension of the *Local* processor that reads data from files and streams them to *DataCalc* via HTTP. In the current version, the *File* processor can read CSV and Arrow files. Comma separated values (CSV) is a popular text based file-format for tabular datasets that is easy to use in ad-hoc application scenarios. The Arrow file format is a binary format that can be used to serialize Apache Arrow[5] datasets to permanent storage. The *File* processor consists of a lightweight server application that can be quickly started on any network connected machine and a *DataCalc* driver component. To reduce network traffic, the server executes DC built-ins such as `FILTER` or `SELECT` on its local datasets. It reuses *Local*'s function implementations, with the exception of `IMPORT`, and adds an implementation of `SCAN` for CSV and Arrow files.

The assignment rewrite of *File* accepts `SCAN` calls whose `@path` annotation starts with a "`file.`" prefix and other built-ins if their arguments are already assigned to the *File* processor. When the driver is called to execute an expression, it simply saves the unevaluated expression in a `TableValue` object and returns that object as result. Eventually, when a consumer creates an iterator on the `TableValue`, the stored DC expression is serialized to JSON and send to the server. The server deserializes the JSON document back into the original expression, translates that expression into a tree of function implementations, executes that tree and streams back the results to the iterator.

[4]https://github.com/oracle/graal/tree/master/truffle
[5]https://arrow.apache.org

Final edited form was published in "IEEE International Conference on Big Data". Los Angeles, 9.-12. Dezember 2019, S. 3864–3873. IEEE. ISBN 978-1-7281-0858-2
https://doi.org/10.1109/BigData47090.2019.9006029

C. JDBC processor

The *JDBC* processor uses the *Java Database Connectivity* API[6] to integrate relational database management systems into *DataCalc*. Most RDBMS support JDBC and with this processor all of these systems are made available in *DataCalc* at once. In a nutshell, JDBC provides a SQL client for the Java programming language. Programs send plain SQL queries to available databases and receive datasets that are encoded in standard Java data types. The *JDBC Processor* translates *DC* expressions into SQL queries that can be send off to JDBC. Depending on the locations of tables, whole DC programs can be converted into a SQL query that is processed by a single RDBMS.

The processor's assignment rewrite uses the JDBC metadata interface to check the paths of `SCAN` calls and to implement column reference shorthands in path matchers. For other built-ins, it checks whether the call's table arguments are already assigned to the *JDBC Processor*. Similar to the *File* processor, `execute` simply saves the raw *DC* expression in a `TableValue` and returns that value as result. Actual query processing is deferred until an iterator over the `TableValue` is consumed. At that point, the whole *DC* expression is translated into a SQL query, send to JDBC, and the result is transformed into Arrow batches that can be returned by the iterator.

For the most part, the *DC* to SQL translation is accomplished by recursively mapping *DC* built-ins to SQL snippets and concatenating those snippets into complete queries. However, the translation of relational functions such as `SELECT` and `WHERE` is slightly more complicated because DC allows arbitrary composition of these functions whereas SQL defines a limited set of valid query forms. To translate these functions we introduce a simple state machine that produces valid SQL queries for any sequence of relational function calls. Figure 5 gives an example of the translation process. In the graph, the arrows represent function calls and the nodes show the current state and the SQL query that would be generated at that point. The graph starts with a `SCAN` call that simply selects all attributes of the requested table. The subsequent `WHERE` adds some predicates to the query but does not change the state. The same is true for the third and fourth call which add another source table and an additional predicate expression. The fifth call, however, adds a set of projection expressions and moves the query into the SELECT state. Once a query is in this state most subsequent relational functions require nesting of the query. This can be observed in the seventh call that joins the projected query with another table. In this particular instance, the query could be unnested quite easily, but this is not true in general and we leave this kind of transformation to the optimizer of the target database system.

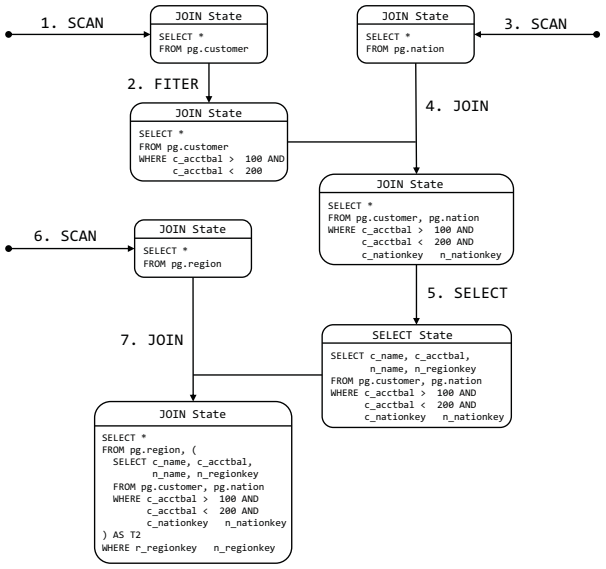


Fig. 5: Translating DC to SQL using states

D. Mongo processor

The *Mongo* processor integrates the NoSQL database system MongoDB[7] with *DataCalc*. Whereas traditional database systems are build around the *Relation*, MongoDB chooses *Collections* of *Documents* as primary abstraction. *Collections* are named sets of *Documents* and a *Document* implements a key-value mapping where keys are character strings and values can be primitives, arrays, or nested documents. MongoDB provides a query language that is optimized for the access of documents. The core primitive of that language are *Query Documents* which define pattern matching constructs for documents but also include expressions for data transformations. Besides the significant differences between Query Documents and SQL, it is possible to translate a number of important *DC* built-ins into MongoDB queries.

The *Mongo* assignment rewrite accepts `SCAN` calls whose paths are contained in the MongoDB collection catalog. This catalog holds the names of all collections but does not define any schema for the documents in those collections. This is the case, because MongoDB documents do not have to adhere to any kind of static schema and each document can define arbitrary attributes. Because of this, *Mongo* paths always have to include a collection name or an alias of such a name. Otherwise, *Mongo* would have to match any attribute path. Besides `SCAN`, *Mongo* currently also accepts the `SELECT` and `FILTER` built-ins. Query execution is implemented in exactly the same way as in the *JDBC* processor: the raw *DC* expression is stored in a `TableValue` and gets translated into a MongoDB query once an iterator is consumed. The results of the query are ransformed into Arrow batches that can be returned by the iterator's `nextBatch` function. Figure 6 shows an example of this translation for a SQL query that

[6]https://www.oracle.com/technetwork/java/javase/jdbc

[7]https://www.mongodb.com

Final edited form was published in "IEEE International Conference on Big Data". Los Angeles, 9.-12. Dezember 2019, S. 3864–3873. IEEE. ISBN 978-1-7281-0858-2
https://doi.org/10.1109/BigData47090.2019.9006029

```
SELECT (l_extendedprice / l_quantity) / (1.0 + l_tax)
FROM mongo.lineitem
WHERE l_extendedprice <= 10000.0
```

↓

```
db.lineitem.aggregate([
  {$match: {$expr: {$lte: ["$l_extprice", 10000.0]}}},
  {$project: {price:
    {$divide: [
      {$divide: ["$l_extprice", "$l_quantity"]},
      {$add: [1.0, "$l_tax"]}
    ]}}
  }
])
```

Fig. 6: SQL to MongoDB translation

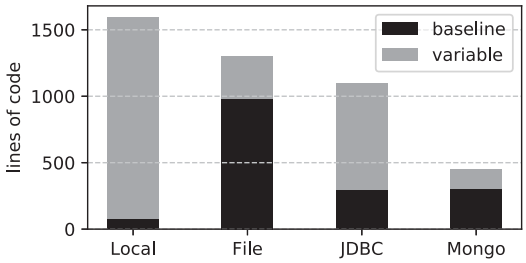


Fig. 7: Lines of code in each processor

can be fully mapped to a MongoDB query. The result of the translation is a MongoDB transformation pipeline that processes document collections by applying a sequence of transformations such as filters, projections, or aggregations.

V. Evaluation

In the previous sections, we have introduced our data integration platform *DataCalc*. On this platform, various data processors collaborate in the evaluation of SQL queries. Now that we have examined the individual components of *DataCalc* in some detail, we want to take a step back and try to get a better understanding of the overall approach. In particular we want to investigate the following questions: 1) how expensive is the development of additional *DataCalc* processors and 2) how do query execution times and the amount data transfer correspond to the placement of data objects.

A. Cost of developing processors

The primary goal of *DataCalc* is to provide a platform for the integration many different kinds of data sources and processors. One aspect that can influence the decision to create an additional processor is the implied development cost of that processor. In the following section we try to quantify this cost by discussing the development of the existing processors and by investigating how many lines of code (LOC) had to be written to implement those processors. Of course, the complexity of code can vary widely and therefore LOC numbers only provide a rough estimate of the "real" development costs. *DataCalc* is implemented in Java and the following LOC numbers always refer to Java code.

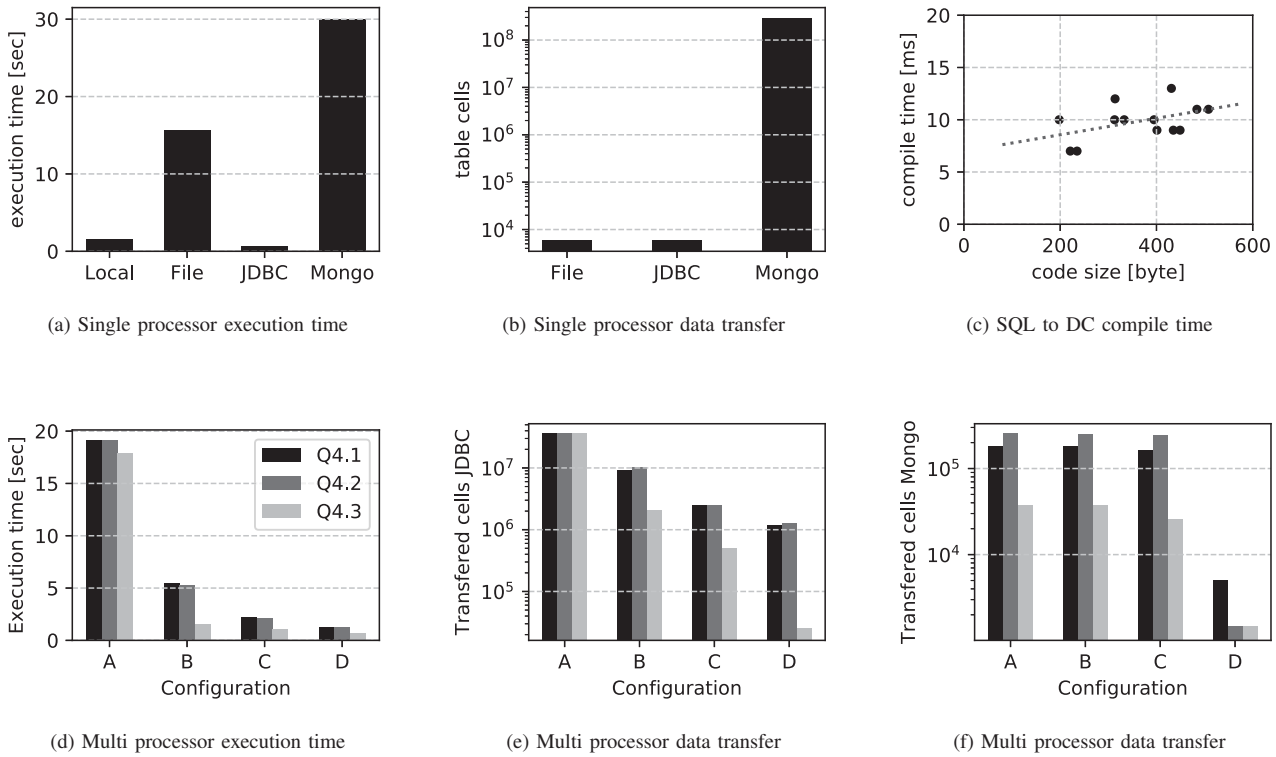
Each of the processors that we have developed in this paper offered unique challenges and required some creative problem solving. However, we also discovered some general rules that we expect to be true for all processors. First, development of a processor always implies a fixed baseline cost that is independent of the number of DC built-ins that are supported. For example, each processor has to implement an assignment rewrite and most processors also need data transformation code that can translate from native data formats to Arrow batches. Second, the function based delegation model enables an iterative approach to processor development where basic functionality is established quickly and advanced features are added later on. After each iteration, processor performance can be evaluated to identify the most promising next feature and developer time can applied efficiently. To give a better overall impression of development costs, we are going to take a closer look at each individual processor in the following paragraphs.

The *Local* processor runs in the same Java virtual machine as the `Runtime`, directly operates on Apache Arrow batches and does not hold its own data sets. In other words, *Local* does not require inter process communication or data transformations. Therefore, the baseline cost of *Local* consists of implementing a generic driver class and an assignment rewrite with about 80 lines of repetitive function recognition code. Unfortunately, the variable costs of *Local* are rather large. This is the case because the processor provides its own function implementations and each additional function requires some significant development effort. In total, the processor currently requires close to 1600 lines of code (LOC). However, the function implementations are still basic and leave much room for improvement. High quality implementations would certainly require a much greater effort.

The *File* processor delegates workloads to a remote server component that operates on various file formats. Compared to *Local*, this setup results in a significantly increased base development cost. In the driver component, networking code contributes the biggest chunk with about 300 LOC. In total, the driver component consist of roughly 500 lines of baseline code. The server component currently consists of a total of 800 LOC. About two thirds of that is baseline code that deals with networking and file access. Similar to *Local*, the variable cost of the *File* processor comes from implementing *DC* functions. However, in the current version `SCAN` is the only function with dedicated implementations. All other functions are shared with *Local*. This enables the *File* processor to implement most DC functions with very little additional cost. On the other hand, we have good reason to believe that the CSV parsing performance might be further improved with specialized implementations of functions such as `FILTER` and `SELECT`. If the *File* processor ever becomes a performance bottleneck for *DataCalc* we can allocate additional resources to investigate this possibility.

The *JDBC* processor uses the *Java Database Connectivity* framework to access relational database systems. In total, the

Final edited form was published in "IEEE International Conference on Big Data". Los Angeles, 9.-12. Dezember 2019, S. 3864–3873. IEEE. ISBN 978-1-7281-0858-2
https://doi.org/10.1109/BigData47090.2019.9006029



(a) Single processor execution time

(b) Single processor data transfer

(c) SQL to DC compile time

(d) Multi processor execution time

(e) Multi processor data transfer

(f) Multi processor data transfer

Fig. 8: *DataCalc* system performance

processor currently takes up 1100 LOC with about 300 lines of baseline code and 800 lines of variable cost code. The baseline code deals with managing JDBC connections, reading database metadata, and transforming JDBC result sets into Arrow batches. The first working version of the processor added a variable cost of almost zero. It only supported the `SCAN` function which can be easily translated into a simple `SELECT * FROM <table>` query. The next version added support for the `FILTER` function at a cost of roughly 300 LOC. Most of that code is introduced by a `Transform` that translates DC into SQL expressions. This transformation is required for the translation of filter predicates but can be reused in other functions that use expression. Eventually, at a cost of about 500 LOC, additional relational functions and support for arbitrary call sequences were added.

The *Mongo* processor uses a total of 450 LOC to execute *DC* queries on a MongoDB database. About 300 lines of baseline code manage connections and transform Mongo datasets into Arrow batches. A basic version, that only supports the `SCAN` function, can be achieved with very little variable cost. Additional functions require a 100 LOC transformation that translates *DC* into Mongo expressions. The remaining 50 LOC are used to match functions and to invoke the Mongo translator. Overall, the *Mongo* processor is the least expensive. One reason for this is the straightforward translation from DC to Mongo's transformation pipeline syntax. However, the *Mongo* processor also implements the smallest number of DC functions. Figure 7 gives a summary of the LOC numbers that we have cited in the previous paragraphs. The dark segment of the bars represents the baseline code and the light segment the variable code. Overall, we believe that the cited costs are rather small, especially for existing data processing systems such JDBC or MongoDB that require very limited upfront investment.

B. System performance

To investigate the influence of data placement on system performance we have experimented with queries of the star schema benchmark (SSB) [O'Neil et al., 2009]. SSB contains 13 analytical queries that are defined over one fact table and four dimension tables. All experiments have been performed on a single machine with a SSD, 16 GB DDR3 RAM and a 2.7 GHz Intel Core i5 CPU with two cores. The *JDBC* processor is connected to a PostgreSQL 11.4 database and the *Mongo* processor to a MongoDB 4.0.3 instance. Both databases run their default configurations. Test data has been generated with the SSB `dbgen` tool at a scale factor of 1 which yields about one gigabyte of data. This scale is, of course, far from any *big data* scenario. However, the primary motivation for the following experiments is to show that *DataCalc* works as intended with regard to code shipping and its implied reduction of data transfers. This effect can be shown independent from any particular data sizes. Each experiment is repeated 10 times and in the subsequent discussion we report the median value

Final edited form was published in "IEEE International Conference on Big Data". Los Angeles, 9.-12. Dezember 2019, S. 3864–3873. IEEE. ISBN 978-1-7281-0858-2
https://doi.org/10.1109/BigData47090.2019.9006029

of these runs. However, the variance between runs has been less than 10% and we deem it insignificant for our purposes.

Figure 8a shows the sum execution time of all SSB queries when all datasets are located on a single processor. In the last column, both *Mongo* and *Local* are enabled because *Mongo* does not support all required operations to run SSB. In the remaining columns only a single processor is active. For the purpose of this test, we extended *Local* with `SCAN` implementation that can load Arrow files from disk. *JDBC* runs all queries in less than a second which makes it the fastest processor by far. *Local* is about three times slower than *JDBC* and both *File* and *Mongo* are several times slower than the other processors. Figure 8b shows the total number of table cells, that is, scalar values contained in a table, that have been transferred from an external engine to the runtime. *File* and *JDBC* both have to move the same limited number of cells. In contrast, *Mongo* has to transfer four orders of magnitude more cells because it does not implement join and aggregation functions. These results confirm that *DataCalc* can, in fact, avoid movement of data and therefore works as intended.

Compared to accessing a DBMS directly, *DataCalc* adds certain overheads to the execution of queries. For example, SQL input has to be translated into *DC* and planning adds the cost of repeatedly applying transformations. To quantify this overhead, we compare the execution times of the *JDBC* processor with the time it takes to run the SSB queries on a plain JDBC connection. Indeed, all queries are slightly slower with the *JDBC* processor than with the plain connection. Query 1.1 has the largest difference with 38 ms but the median difference is only 13ms with a standard deviation of 14ms. To get a more detailed understanding of the overhead we also investigated compile times separately. Figure 8c shows the SQL to DC compile time in relation to code size. The median compile time over all queries is 10ms with a standard deviation of 2ms.

In our final experiment, we investigate the behaviour of *DataCalc* in an integration scenario. We test four data placement configurations using the *JDBC*, *Mongo*, and *Local* processors. We limit our discussion to the SSB queries 4.1, 4.2, and 4.3 because these queries join over all tables. In configuration *A*, the fact table `lineorder` is placed on *JDBC* and all dimension tables are placed on *Mongo*. In each subsequent configuration (*B*, *C*, and *D*) the next join partner of `lineorder` is moved from *Mongo* to *JDBC*. That is, in configuration *D*, all tables but one are located on *JDBC*. The three bottom graphs in Figure 8 show execution times and data movement of each configuration. Configuration *A* is the slowest by far and also requires the most data movement. The test queries do not define any predicate on `lineorder` and therefore the complete table has to be moved from *JDBC* to *Local* before it can be joined with the dimension tables. Switching to configuration *B* yields the largest overall improvement. Queries 4.1 and 4.2 finish more than 70% faster and, due to a higher selectivity predicate, query 4.3 even achieves a 90% improvement. Similarly, data movement from *JDBC* is reduced by 75%, 72%, and 94%. The data movement from *Mongo* changes very little because of the relatively small size of the newly placed table. The switch to configuration *C* and then to *D* shows similar improvements although at a smaller magnitude. The following table summarises the changes in execution time and data movement for all configuration changes.

	Execution time			Data JDBC			Data Mongo		
Switch	4.1	4.2	4.3	4.1	4.2	4.3	4.1	4.2	4.3
A → B	72%	72%	91%	75%	72%	94%	0%	0%	1%
B → C	60%	60%	29%	73%	76%	76%	10%	5%	32%
C → D	40%	44%	37%	53%	49%	95%	97%	99%	94%

VI. Conclusion

Big Data applications need a powerful and efficient infrastructure to meet the demanding processing requirements for data preparation and analysis. In this environment it quickly becomes difficult to analyze data across the boundaries of specialized big data systems. To overcome that, we discussed the novel extensible data integration platform *DataCalc* that enables a set of heterogeneous data processors to collaborate in the processing of SQL queries in this paper. The platform translates queries into an internal form that represents relational operators as function calls. These calls are delegated to data processors which carry out computations on their local data stores. In this way, *DataCalc* can access the data that is distributed across its processors.

References

[Abadi et al., 2016] Abadi, D., Agrawal, R., Ailamaki, A., Balazinska, M., Bernstein, P. A., Carey, M. J., Chaudhuri, S., Dean, J., Doan, A., Franklin, M. J., et al. (2016). The beckman report on database research. *Communications of the ACM*, 59(2):92–99.

[Begoli et al., 2018] Begoli, E., Camacho-Rodríguez, J., Hyde, J., Mior, M. J., and Lemire, D. (2018). Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *Proceedings of the 2018 International Conference on Management of Data*, pages 221–230. ACM.

[Gog et al., 2015] Gog, I., Schwarzkopf, M., Crooks, N., Grosvenor, M. P., Clement, A., and Hand, S. (2015). Musketeer: all for one, one for all in data processing systems. In *Proceedings of the Tenth European Conference on Computer Systems*, page 2. ACM.

[Luong et al., 2018] Luong, J., Habich, D., and Lehner, W. (2018). Design of a portable programming abstraction for data transformations. In *Proceedings of the 7th International Conference on Data Science, Technology and Applications, DATA 2018, Porto, Portugal, July 26-28, 2018.*, pages 400–408.

[Luong et al., 2019] Luong, J., Habich, D., and Lehner, W. (2019). DataCalc: Ad-hoc Analyses on Heterogeneous Data Sources. In *Appears in: IEEE Big Data 2019*.

[O'Neil et al., 2009] O'Neil, P., O'Neil, E., Chen, X., and Revilak, S. (2009). The star schema benchmark and augmented fact table indexing. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 237–252. Springer.

[Papakonstantinou et al., 2019] Papakonstantinou, Y., Goo, A., Ruppert, B., Wilsdon, J., and Varakur, P. (2019). *Announcing PartiQL: One query language for all your data*. https://aws.amazon.com/blogs/opensource/announcing-partiql-one-query-language-for-all-your-data/.