Rialto: A Knowledge Discovery Suite for Data Analysis

Giuseppe Manco[a,*], Pasquale Rullo[b], Lorenzo Galucci[c], Mirco Paturzo[c]

[a] *ICAR-CNR, Via Bucci 41c, 87036 Rende (CS) - Italy*
[b] *Dip. di Matematica e Informatica, Universitá della Calabria, Via Bucci 30b, 87036 Rende (CS) - Italy*
[c] *Exeura S.r.l., Via P.A. Cabrai, 87036 Rende (CS) - Italy*

Abstract

A Knowledge Discovery (KD) process is a complex inter-disciplinary task, where different types of techniques coexist and cooperate for the purpose of extracting useful knowledge from large amounts of data. So, it is desirable having a unifying environment, built on a formal basis, where to design and perform the overall process. In this paper we propose a general framework which formalizes a KD process as an algebraic expression, that is, as a composition of operators representing elementary operations on two worlds: the *data* and the *model* worlds. Then, we describe a KD platform, named Rialto, based on such a framework. In particular, we provide the design principles of the underlying architecture, highlight the basic features, and provide a number of experimental results aimed at assessing the effectiveness of the design choices.

Keywords: Knowledge Discovery Process, Data Mining, Business Analytics Platforms

1. Introduction

Knowledge Discovery (KD) is the process of extracting novel, potentially useful, and ultimately understandable patterns in data (Han and Kamber, 2001). A KD process involves several steps, including data acquisition, data pre-processing,

*Corresponding author
Email addresses: manco@icar.cnr.it (Giuseppe Manco), rullo@mat.unical.it (Pasquale Rullo), lorenzo.gallucci@exeura.eu (Lorenzo Galucci), mirko.paturzo@exeura.eu (Mirco Paturzo)

data mining and data post-processing. In general, each step requires a variety of algorithms and techniques, such as SQL queries to manipulate data in the pre-processing phase, different learning methods in the data mining step, or model visualization metaphors in the post-processing phase.

Though the main efforts in the KD community have been so far devoted to the development of efficient mining algorithms and techniques, some standardization initiatives, aimed at solving the KD problem as a whole, have been proposed. As an example, the Cross Industry Standard Process (CRISP) methodology (Shearer, 2000), where data mining is set in the larger context of KD, is considered a *de facto* standard methodology used by industry data miners. A more formal attempt to define a general and unified framework is the 3-World model proposed in (Johnson et al., 2000). Here, a KD process is seen as a "multi-step process where the input of one mining operation can be the output of another" and a data mining algebra is proposed.

Thus, in the perspective of a unified approach, a data mining system should provide support to the entire KD lifecycle within a unique, integrated environment, providing suitable features to enable the construction of a KD analysis as an interactive and an incremental process. To this end, there is a number of requirements that an integrated KD environment should meet, including:

- capability to gather data from diverse data sources, such as databases, Excel, CSV files, etc. (*data acquisition*)
- availability of a wide variety of techniques to reduce data preprocessing efforts (nearly 80% of the time that an analyzer spends on analytics projects is during preprocessing)
- richness of mining algorithms and techniques
- availability of effective model and data visualization tools, for decision making and informed data manipulation, respectively
- *extensibility*, i.e., open architecture to be easily customizable with new specific tasks.

Additional requirements that a KD system should fulfill in order to cope with real-world applications include:

- *Usability*: a KD process can be extremely complex with several interacting components. In order to efficiently design and build such a process, the supporting KD environment should provide a GUI allowing for a simple and interactive creation of the analysis process, obtained by assembling a number of elementary tasks according to a given logical flow. Furthermore, it should enable the user to visually explore data and models.

- *Multiple data types*: while classical KD approaches mainly deal with relational data, the need to deal with unstructured data is lately growing (Sebastiani, 2002). Indeed, textual data represent the large majority of stored digital information and is growing at a much faster rate than business quantitative data. As a consequence, text mining applications have been steadily increasing in the last few years (see, e.g., (Vilares et al., 2015), (Hristovski et al., 2008)). Besides texts, other data types like time series (chung Fu, 2011), graphs (Borgelt, 2009), streams (Gaber et al., 2005; S., 2005) have lately gained increasing popularity.

- *Big Data*: a KD tool should be able to efficiently deal with mass-memory resident data and provide mechanisms to scale up to large real-world domains.

Today, a large number of KD tools exists, with both commercial and open source licenses, e.g., (Hall et al., 2009; Berthold et al., 2009). KD systems are diverse in design and implementation. Many commercial KD tools available in the market result from the integration of data mining methods into commercial statistical tools, e.g., SPSS and SAS, while others, such as Oracle Data Mining, result from the integration of data mining solutions into business intelligence products. A few tools, such as KNIME (Berthold et al., 2009) and IBM SPSS Modeler (formerly Clementine), were natively designed to provide full support to the KD lifecycle, giving the possibility of executing the entire KD process

within a unique, integrated environment, thus eliminating the need for context switching. Surveys on KD tools can be found in (Mikut and Reischl, 2011; Chen et al., 2007).

In this paper we present Rialto, a KD platform assisting the designer during the entire KD lifecycle. Rialto relies on a visual interaction which enables the construction of a KD process as a workflow representing a composition of elementary operators on two worlds: the *data* and the *model* worlds (the 2W Model). Thus, a workflow consists of nodes that are either tables (data), or models or tasks (operators). Tables are unnormalized relations, as *Event Collections* for managing complex data types, such as texts, itemsets, etc., are allowed. Operators are implemented as plug-ins of the platform. In order to facilitate the work of the analyst, Rialto offers a broad variety of pre-packaged plug-ins covering all phases of the KD process (from data acquisition to model deployment). In addition, new *ad hoc* Java-based plug-ins can be created whenever needed (*extensibility*). To this end, Rialto provides an integrated development environment which enables and simplifies code generation.

Currently, Rialto works on both relational data and texts. Data mining is supported by plug-ins for classification (including decision trees, Naive Bayes, rule induction), regression, clustering and association mining. As far as texts is concerned, text-specific plug-ins are available for preprocessing, classification (Complement Naive Bayes (Rennie et al., 2003), Olex (Rullo et al., 2009) , OlexGa (Pietramala et al., 2008) and Max Entropy (Nigam et al., 1999)), and topic detection based on the Latent Dirichlet Allocation (LDA) model (Blei et al., 2003). Ongoing work is devoted to extensions to deal with data streams as well as spatio-temporal data.

Rialto is also equipped with plug-ins allowing the analyst to inspect both data and models, thus taking advantage of descriptive statistics, explanatory charts, and models visualizations metaphors.

Finally, Rialto exhibits features, such as the *bridge* mechanism, for the efficient and transparent manipulation of large amounts of data. Efficiency and scalability are ensured by a workflow execution engine relying on parallel exe-

cution strategies along with effective buffering policies and management of data in main memory.

In short, the contributions of the paper can be summarized as follows.

- We propose an algebraic model, namely the 2W model, for specifying data mining queries and, more in general, a KD process. The 2W Model is a variant of the 3W model which emphasizes the underlying philosophy of combining several forms of knowledge and the cooperation among solvers of different nature, in a more general setting.

- We introduce an effective architecture which implements the 2W model and visually models data mining queries as interactions among the two worlds. Rialto relies on two key aspects: closure and extensibility. These aspects are directly derived from the specification of the 2W model, and they are the essential tools to achieve the combination of deductive inferences along with inductive mechanisms and statistical methods. However, Rialto also exhibits some important aspects beyond those: some architectural choices are specifically introduced to enable the optimized execution of data mining queries. In addition, a strong emphasis is given to visual inspection and support for the choice of the optimal data/model manipulation operator.

The paper is organized as follows. In Section 2 we provide a description of the 2W Model. In Section 3 we give an overview of Rialto. In Section 4 we describe the architecture of Rialto. In Section 5 we show two case studies based on Rialto. In Section 6 we evaluate Rialto against a number of both functional and architectural requirements, and perform a number of experimental analyses aimed at assessing the effectiveness of the proposed architecture. In Section 7 we describe a number of features we are currently working on and, finally, in Section 8 we provide some concluding remarks.

2. The Foundation: Modeling KD Processes as Algebraic Processes

There is a sort of standard framework upon which complex data analytics applications can be approached (Manco et al., 2008). Notably, analytical questions posed by the end user need to be translated into several tasks such as choose analysis methods, prepare the data for application of these methods, apply the methods to the data, and interpret and evaluate the results obtained. The interaction of these tasks requires combining several forms of knowledge and the cooperation among solvers of different nature: we need to support deductive inferences along with inductive mechanisms, in conjunction with statistical methods. Historically, there has been an effort to express this combination in a unified framework (Imielinski and Mannila, 1996): either by integrating data mining algorithms with the underlying database systems (Chaudhuri et al., 2002; Giannotti et al., 2004); or by providing ad-hoc extensions of SQL (Imielinski and Virmani, 1999; Wang et al., 2003; Ortale et al., 2008). We can devise however a unifying picture where we consider the Knowledge Discovery process as an algebraic process, with some primitive data types and instances which can be manipulated by means of a set of basic operators.

The impact of such an algebraic framework can be summarized in two main aspects which we expect the framework will support. First, in an effort to combine reasoning and mining, it is important to treat results of data mining on a par with data objects which can be further manipulated. Second, suitable combinations and compositions of known mining tasks can enable more poweful mining computations and findings. In essence, the principle upon which these aspects can be enabled is the *closure* capability.

The current literature has studied the foundational aspects of this algebraic view of the KD process. In particular, the 3-Worlds Model (3W in the following) (Johnson et al., 2000; Calders et al., 2006) introduces and develops this abstract view as a mathematical structure where three entities, namely data, intensional models and extensional models, can interact by means of algebraic operators. In the following we will describe a variant of the 3W, which we refer to as the

2-World Model (2W). In this model, a KD process can be especified as the interaction between *two* apparently separate worlds, the *data world* (D-world) and the *model world* (M-world).

The D-world consists of a set of entities to be analyzed, with their properties and mutual relationships. Following (Johnson et al., 2000; Calders et al., 2006), we assume that the D-world D is made of all possible nested relational tables that can be defined over an infinite attribute set $\mathcal{A} = \{A_1, \ldots, A_m, \ldots\}$, where each A_i is associated with a domain $dom(A_i)$.

The M-world consists of the intensional representations of the patterns relating the elements of the D-World, expressed through specific languages (rules, mathematical properties, etc.). In (Johnson et al., 2000; Calders et al., 2006), M is populated by region tables, i.e., relational tables where attributes can only be associated with the special *region* domain. In particular, a region can be described in terms of a description of its members, i.e., region membership criteria. The simplest criteria can be devised as constraints over a subset of attributes in $\mathcal{A}$. For example, $A \leq 5 \wedge B > 6$ intentionally represents the set of all tuples t defined over a relation $r[A, B]$ such that $t[A] \leq 5$ and $t[B] > 6$. Thus, an example model can be defined over the scheme given by the attributes $\{\texttt{Condition}, \texttt{Classification}\}$, and a possible instance is represented by the table r defined as:

`Condition`	`Classification`
$A \leq 5 \wedge B > 6$	$C = 1$
$A < 5$	$C = 0$

It is easy to see how the above table encodes a decision tree, where each row represent a path and in particular the `Condition` attribute represents the path, whereas the `Classification` attribute represents the label associated with that path.

Within the 2W, we deliberately choose a more general setting, and we do not specify directly models as region tables. Rather, we assume that a model m is any mathematical structure characterized by

- two schemas $S, S' \subset \mathcal{A}$ and $S \subset S'$; intuitively, S is the set of attributes upon which m is built, and $S' \setminus S$ is the set of attributes describing the result of the application of m to a tuple t over S;

- an *entailment* $\vdash_S$ operator such that, for each tuple t defined over S, $m \vdash_S t$ if t satisfies m;

- an *extension* operator τ such that, for each t entailed by m, $\tau_m(t) = t'$, where t' is a tuple defined over S', such that $t'[S] = t$.

We notice that the above two operators subsume the basic properties that models should have: the capability of bounding tuples (entailment), and the capability to annotate tuples according to these bounds (extension).
The table r illustrated above is a special case of this formalization, where $S = \{A, B\}$ and $S' = \{A, B, C\}$. Also, the entailment $\vdash$ holds if there is any region in `Condition` which is satisfied by t, and $\tau_r(t)$ extends t with the value of the region associated with `Classification`. For example, if $t = \langle 3, 6\rangle$ then $\tau_r(t) = \langle 3, 6, 0\rangle$ because the entailment is satisfied by the second row of r, and the corresponding `Classification` region is $C = 0$. In the following we refer to $\langle S, S', \vdash_S, \tau\rangle$ as the schema of a model.

Thus, in our formalization a 2W database is a tuple $\langle \mathbf{D}, \mathbf{M}\rangle$, where $\mathbf{D}$ is a set of relations in D, and $\mathbf{M}$ is a set of models in M. A knowledge discovery process can hence be formalized as a transition from a 2W database $\langle \mathbf{D}, \mathbf{M}\rangle$ to another one $\langle \mathbf{D'}, \mathbf{M'}\rangle$, through a number of suitable operators. In particular, the closure between the two worlds D and M is achieved by means of the following set of classes of operators (see Figure 1):

- *Filtering* operators are self-injecting relationships. They take a set of entities as input and produce a new set of entities, generating data from data (for pre-processing purposes) and models from models. Thus, a generic data filter operator is of the form

$$\phi_D : D^n \to D^m$$

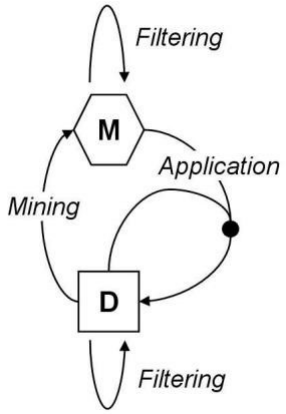

Figure 1: The 2W Model

while a generic model filter operator is of the form

$$\phi_M : M^n \to M^m.$$

- *Mining* operators relate data entities to model entities, thus enabling transitions from the D-world to the M-world. A generic mining operator (representing a mining algorithm) is of the form

$$\mu : D \to M.$$

The result of the application of μ to a data entity d is a model, which describes a pattern holding over d.

- *Application* operators can be seen as a sort of reverse mining operators, generating a data entity from the application of a model to another data entity, i.e.

$$\alpha : D \times M \to D$$

Applying a model m to a data entity d essentially means making the intensional properties of m explicit in extensional form: e.g., by associating each tuple in a table with the most likely target class according to the model, or by enumerating the frequent patterns appearing within the

tuple. The only requirement is that schemes are compatible, i.e., given $d \in D$ with scheme X and $m \in M$ with scheme $\langle S, S', \vdash, \tau \rangle$, then $S \subseteq X$.

Now, every data mining query can be represented by means of a composition of the above operators, and a KD process as a transition of 2W databases through data mining queries. For an instance, a typical learn-and-test process can be represented as the transition

$$\begin{aligned} &\langle \{d\}, \emptyset \rangle \\ &\quad \rightarrow \langle \{d, d_{Train}, d_{Test}\}, \emptyset \rangle \\ &\quad \rightarrow \langle \{d, d_{Train}, d_{Test}\}, \{m\} \rangle \\ &\quad \rightarrow \langle \{d, d_{Train}, d_{Test}, d_c\}, \{m\} \rangle, \end{aligned} \tag{1}$$

where each transition is determined by the application of some of the above described operators, notably:

$$\begin{aligned} \langle d_{Train}, d_{Test} \rangle &:= \phi_D^{(split)}(d) \\ m &:= \mu(d_{Train}) \\ d_c &:= \alpha(m, d_{Test}) \end{aligned}$$

Here, $\phi_D^{(split)}(d)$ represents a data filtering operator which splits d into two partitions. In turn, m is the model learned by the execution of a mining algorithm μ over d_{Train} (e.g., a decision tree), and d_c is the data entity obtained by applying the model m on the test set d_{Test}.

As another example, a process inducing a decision tree DT from a data set d, by using the learning algorithm μ^{DT}, and then transforming DT into a set of rules, is synthetically represented by the expression $\phi_M^{DT2Rules}(\mu^{DT}(d))$, where $\phi_M^{DT2Rules}$ is a suitable model filtering operator.

Compared to the 3W model, the 2W model exhibits some substantial similarities and differences, as shown in table 1. First, the Model world does not entail an extensional representation in our simplified view. As a matter of fact, the extensional representation of a model corresponds to a set of tuples associated with the regions they belong to. Since we implicitly embed these associations

	3W	*2W*
Data	Nested relational tables	Nested relational tables
Models	Region tables Both intensional and extensional representation	Any mathematical structure with entailment and extension No extensional representation
Data operators	Nested relational algebra	Generic (any function $D^n \mapsto D^m$)
Model operators	Dimension algebra	Generic (any function $M^n \mapsto M^m$)
Mining operators	Regionize, loop	Generic (any function $D \mapsto M$)
Model to data	Populate	Generic (any function $D \times M \mapsto D$ with compatible schemes)

Table 1: Comparison of 3W and 2W.

within the entailment and extension operators, we don't need to differentiate between intensional and extensional representation. The differentiation, although useful from a theoretical point of view, is of little interest from a practical point of view.

Second, we regard the operators ϕ_D, ϕ_M, μ and α as black boxes, whereas the 3W world only allows a pre-specified set of basic operators, upon which to derive more complex operators by means of compositions. In particular, the operators ϕ_D in 3W represent all the nested relational algebra operators, and ϕ_M is any operator composing and decomposing regions. Also, the only mining operators in (Calders et al., 2006) are the *regionize* and *loop* operators. These operators are useful from a theoretical point of view, since they allow to investigate the expressive power of the overall algebra: that is whether any data mining algorithm can be expressed by means of a combination of this minimal set of operators. However, it is extremely unpractical to rely on such operators from an application viewpoint. This is the main reason why we decided to keep the operators at an abstract level, as functionals within the two worlds. The key point in the 2W is that objects in different worlds can interact, and that there

is a way of mapping an object within D to an object within M, and viceversa.

Finally, another substantial difference, is the choice of the mining models. As we already mentioned, the 3W focuses on region tables, and regions are only expressible as sets of linear inequalities. As Calders et al. (2006) declare, "*this limitation means that the results of some data mining operations might not be expressible, as they require more complex mathematical objects. Straightforward examples are, e.g., non-linear regression methods, support vector machines, and clustering methods resulting in non-linear regions, etc' [...]. Therefore, the description of the output of some data mining tasks requires more sophisticated constraints than the ones used in the paper*". By contrast, in the 2W we choose to describe models in a more general setting, i.e., as mathematical structures equipped with the entailment and extension operators, the former providing the capability of bounding tuples, and the latter to annotate tuples according to these bounds. In this respect, the 2W model should be able to express data mining queries involving different models.

Rialto was designed with the 2W philosophy in mind, and the architectural choices are aimed at injecting the underlying algebra within a visual and interactive environment. In the following we explore the basic features of such an environment.

3. An Overview of Rialto

The framework we propose is a graphical environment for designing and managing KD processes as algebraic expressions based on the 2W Model. These are defined by means of workflows, whose nodes represent either *tables*, *models* or *tasks*:

- *Tables* represent entities of the D-world. A table is a set of tuples, characterized by attributes. Attribute domains can be either primitive or complex data types. In particular, an attribute can be an *Event Collection* (see Section 4.2.1), used to efficiently handle complex objects, such as itemsets, sequences, time series, texts, etc. In general, Event Collections

can be used for the efficient storing of sparse data sets (i.e., with only a few nonzero features).

- *Models* represent entities of the M-world.
- *Tasks* represent the algebraic operators of the 2W Model - thus, we have tasks of type *filtering*, *mining* and *application*.

The screenshot of an example learn-and-test workflow is shown in Figure 2. It essentially represents a KD process similar to the one showin in equation 1 above. This workflow starts with an *acquisition* task from a CSV file (a kind of filtering task), whose data is imported within Rialto and stored in table *AcquiredDataTable*. Each row in this table represents a pre-classified example. Then, a number of *pre-processing* steps are performed. In particular, the above table is first submitted to a *filter* node in charge of managing *null* values. The resulting table *NoMissingValuesTable* is then processed by another filter node, namely, *Discretize*, in charge of discretizing the values of attribute *Age*, so generating *DiscretizedAgeTable* (note that both the above filters transform tables into tables). Now data is ready for *mining* and, then, the EC4.5 algorithm (an extension of C4.5 (Quinlan, 1987)) is run by using the holdout method. To this end, the input table *DiscretizedAgeTable* is split into a training set and a test set (*HoldOut Testset*). The former is used to induce the model (a decision tree, in this case) represented in the workflow by the hexagonal shaped node. The latter (i.e., the test set) is used to test the accuracy of the induced model. This is done by using the *application* task Prediction Join (visually, the model node and the test table *HoldOut Testset* are fed to the prediction join node). The resulting table *Predicted*, enabling comparison of the predicted labels with those of the ideal classification, is finally *exported* as a CSV file.

As mentioned above, every task node in a workflow represents an operator of the 2W Model (filtering, mining, application), and is implemented as a Java *plug-in* (so any transformation on tables and models can be performed, as no predefined operators are in the 2W Model). However, in order to make it easier for the analyst the construction of workflows, Rialto offers a broad variety of

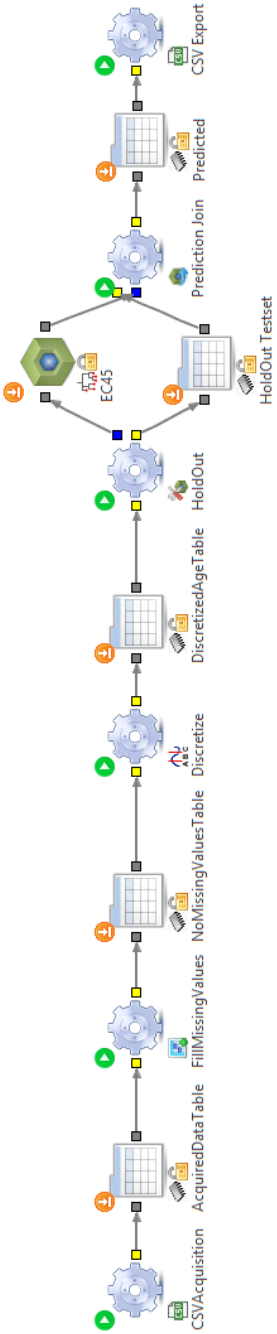

Figure 2: An example learning workflow

pre-built plug-ins covering all phases of the KD process. For an instance, for the purpose of data acquisition, Rialto provides plug-ins for different data types and format, namely, Arff, CSV, Xls, texts, and tables from relational databases through JDBC. Among the pre-processing plug-ins, SQL filters can be used to create tables from other tables. Data mining is supported by a number of plug-ins for classification, regression, clustering and association mining. Since Rialto works also on textual data, text-specific plug-ins are available for pre-processing (feature extraction, feature selection, concept recognition, etc.), text classification (Complement Naive Bayes (Rennie et al., 2003), Olex (Rullo et al., 2009) , OlexGa (Pietramala et al., 2008) and Max Entropy (Nigam et al., 1999)), and topic detection based on the Latent Dirichlet Allocation (LDA) model (Blei et al., 2003).

One peculiarity of Rialto is that, in the logic of an open and extensible architecture, not only tasks of the 2W Model are implemented as plug-ins, but *every* functionality in Rialto is implemented as a plug-in. For instance, Rialto offers several plug-ins (not appearing in workflows - for this reason called *behind-the-scene*, as opposed to tasks that are *on-the-scene* plug-ins) whereby the analyst can effectively and easily examine and inspect both data and models, taking advantage of descriptive statistics, explanatory charts, models visualization metaphors.

One of the fundamental behind-the-scene plug-in types in Rialto is that of *bridges*. A bridge is a component in charge of supporting the mapping of the table logical primitives into their physical level, thus providing a decoupling mechanism between logical data and its physical storage.This way, the bridge mechanism makes plug-ins independent from table implementations. A bridge is either a main memory bridge, used for "small" tables, or a mass memory bridge, the latter relying on a relational DBMS. The bridge mechanism ensures both transparency and efficiency of data retrieval and storing.

Table 2 provides a summary of the different classes of plug-ins. In its current release, Rialto features over 100 pre-packaged plug-ins.

This notwithstanding, new *ad hoc* plug-ins may be needed from time to time

Table 2: Summary of the different classes of plug-ins

Plug-in type	**Note**
Task	Implements 2W Model operators - Appears in workflows
Model visualizer	Applies to models - Does not appear in workflows
Data visualizer	Applies to tables - Does not appear in workflows
Statistical tool	Applies to tables - Does not appear in workflows
Bridge	Applies to tables - Does not appear in workflows

to fulfill the specific needs of the application at hand, e.g., an acquisition task to get data from a legacy system, or a new model visualizer. At this aim, Rialto provides an exhaustive Java-based application programming interface allowing for quick access to the main methods. This way, a user can easily implement, say, a new model induction algorithm, a new data visualization metaphor or a new bridge, and store it in the plug-in archive of Rialto.

Finally, efficiency and scalability of the Rialto platform are ensured by a workflow execution engine relying on parallel execution strategies along with effective data handling policies. Rialto provides support for three kinds of parallelism: parallel execution of independent workflow branches, parallel table scan and pipelining. Data handling relies on techniques for the efficient management of data in main memory and effective buffering strategies.

4. System Architecture

The general architecture of the system is shown in Figure 3. Here, we can observe two basic modules, namely, GUI (Graphical User Interface) and CORE. The former implements the required functionalities for visual workflow composition, as well as data/model explorations. The latter is responsible for the efficient execution of workflows.

Besides these two modules, there are several other components that can be *plugged* into the system. Data and model *visualizers* are bound to tables and models, respectively, and are aimed at performing data and model explorations.

Bridges are bound to tables, and are in charge of implementing a decoupling mechanism between logical data and physical memory (see Section 4.2.4 for more details). *Tasks* are implementations of the 2W Model operators.

The *extensibility* is provided through Rialto SDK, a software development kit allowing users to implement, build and integrate their own plug-ins into their repositories. Plug-ins in Rialto are based on Java and can be developed through the Rialto integrated development environment (IDE) which enables and simplifies code generation. The current release of Rialto includes over 100 plug-ins, covering all phases of the KD process.

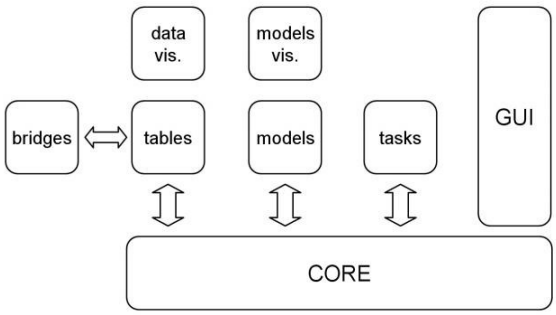

Figure 3: Rialto Architecture

We next analyze each architectural component in turn, starting from the GUI.

4.1. Graphical User Interface

The basic functionalities of the user interface are aimed at modeling KD analyses by means of workflows. To this purpose, Rialto implements a visual interaction metaphor for representing data, models and tasks, along with their interactions. In addition, Rialto provides suitable features to enable the construction of a KD analysis as an interactive and an incremental process, e.g., statistical visualization facilities to inspect, during pre-processing, data properties, as well as tools for models inspection and performance measurement

(like classification accuracy, or statistical relevance) during post-processing. In

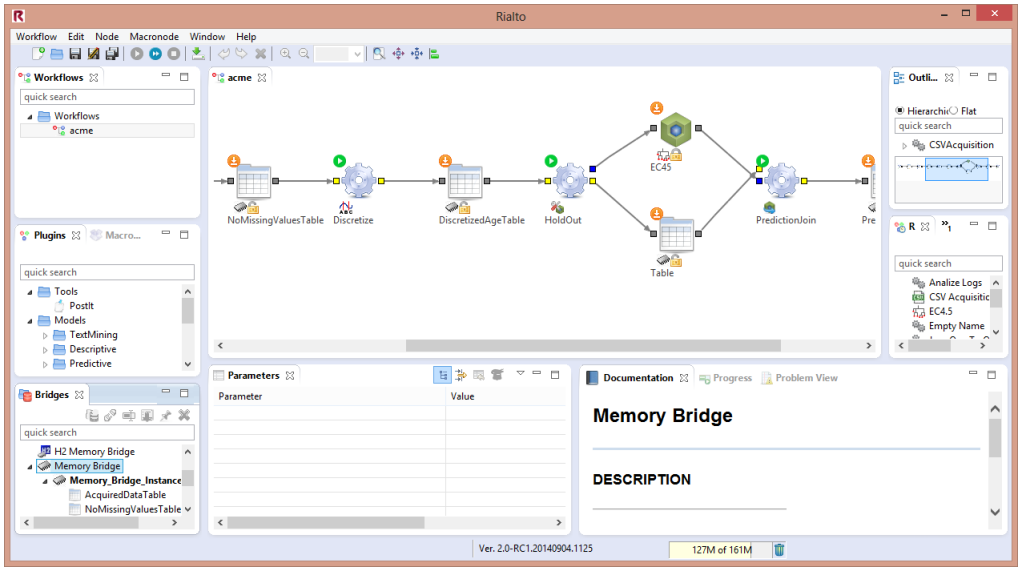

Figure 4: Workbench

particular, the user interface provides three different perspectives:

- *workflow perspective*, used to create, configure and connect nodes;
- *data perspective*, which enables exploratory analyses;
- *model perspective*, which allows inspecting and visualizing models.

4.1.1. Workflow Perspective

Figure 4 shows a screenshot of the Rialto workbench. The Workflow Editor Area (WEA) is the heart of the workbench. Here, a workflow is designed by exploiting a graphical editor, where a user can create, position, move and connect nodes. Nodes can also be moved around in the WEA with a drag-and-drop operation. Additional functionalities available in the Workflow Perspective are: the *workflows navigator* allowing the user to navigate and display on the WEA the workflows which are currently open; the *plug-ins view* which lists all

the available nodes that can be instantiated in a workflow through drag-and-drop operations; the *macronodes view*, which lists all available macronodes (i.e., pieces of workflows); the *bridge view* allowing to select a bridge to be attached to a given table (recall that each table has a bridge associated - see Section 4.2.4 for more details).

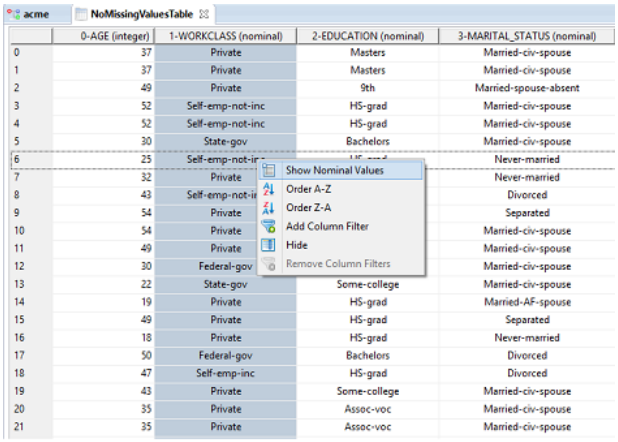

Figure 5: Table View

4.1.2. Data Perspective

In the Data Perspetive, the analyst can get insights into the elements of the D-world, i.e., tables (see Figure 5). To this end, he/she can perform exploratory analyses by means of *ad hoc* visualization and statistical tools as well as simple data manipulation tasks (such as ordering or filtering values). The data perspective can be accessed by double-clicking on a table node.

4.1.3. Model Perspective

Similarly to data, models can also be explored by clicking on the respective nodes in a workflow. Model visualizers allow for a visual inspection of the

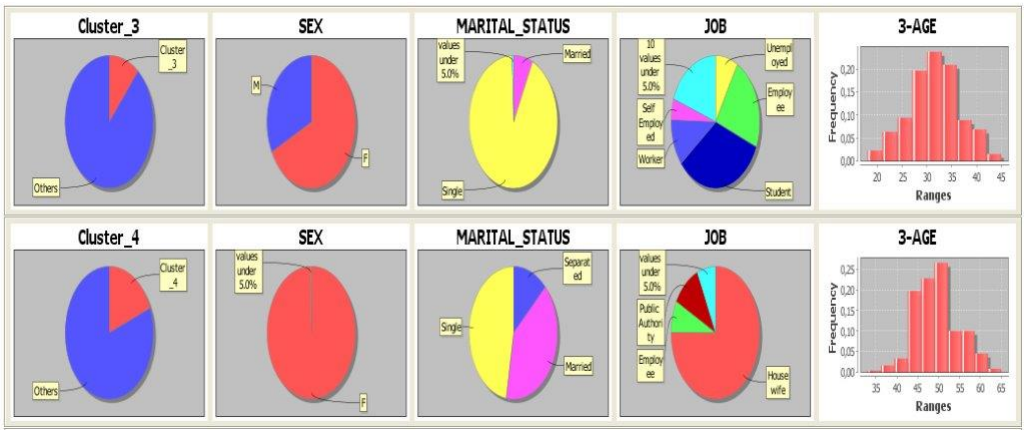

Figure 6: Clusters representation

underlying model in order to understand its features. An example visualization for a clustering model is shown in Figure 6, where the properties of Cluster_3 and Cluster_4 are represented in terms of population frequencies (leftmost pie charts), and attribute value distributions (sex, marital_status, job, 3-age).

4.2. Tables

Data in Rialto is stored within tables. From an abstract point of view, a table represents a set of entities characterized by properties. Properties are associated with a data type, which can be simple (nominal, ordinal, numeric) or complex (string, *event collection*). Nominal properties are characterized by an enumerable set of admissible values, which exhibit no order. By contrast, ordinal properties exhibit a standard order, and they are internally represented as integers. Finally, numeric properties can be exploited for representing results of measurements of real-life phenomena, and are implemented as double. Strings are exploited in Rialto to represent information that is not relevant to the analysis: in practice, a string is any sequence of bytes that are not manageable by the core system.

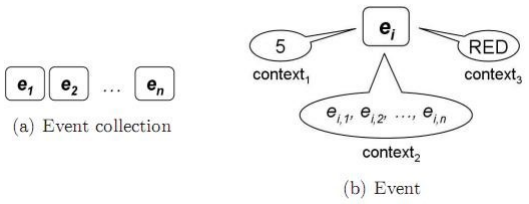

Figure 7: Event Collections

4.2.1. Dealing with complex data: Event Collections

Modeling of complex properties is usually mandatory in real-world applications, which typically require to deal with complex structures, e.g., itemsets, texts, sequences, time series. Within Rialto, we introduced a data type which can subsume several complex properties: the Event Collection (EC). An EC is essentially a set of events, where each event is a complex structure characterized by a set of predefined contexts, and a context is associated again to a data type. Thus, ECs in principle embody a recursive structure (a table within a table), where events correspond to sub-entities, and contexts represent the related sub-properties. Figure 7 illustrates the above ideas: each event has three contexts, namely, *context1* which is numeric, *context2* which is an EC, and *context3* which is nominal. It is easy to see how an EC can be used to model, e.g., itemsets (which are essentially event collections were events expose a single nominal context), time series (where events are characterized by a timestamp and a value) or even textual data. In the latter case, a document can be mapped to an EC of terms. Textual events can be characterized by contexts such as n-gram length, frequency within the document, and so on.

4.2.2. Manipulating Tables

Following the philosophy of an open architecture, tables can be accessed and managed by means of system APIs. In practice, a table can be created by specifying the metadata information and the underlying storage manager (or

bridge, explained later in this section). The following code fragment specifies a table with 4 attributes, and then generates a row which is added to the table:

Code Fragment 1: Table specification

```
TableMetadataBuild metadataBuilder =
        new TableMetadata.Builder();
metadataBuilder.add(new StringAttribute("Name"));
metadataBuilder.add(new DoubleAttribute("Height"));
metadataBuilder.add(new IntegerAttribute("Age"));
metadataBuilder.add(new NominalAttribute("SEX", "F", "M", "F"));
final TableMetadata tableMetadata = metadataBuilder.build();
Table table = bridge.generateTable(tableMetadata);
final Row newRow = table.getNewRow();
newRow.setValue(0, "John");
newRow.setValue(1, "1.74");
newRow.setValue(2, 30);
newRow.setValue(2, "M");
table.appendRow(newRow);
```

We can notice from the example that the library functions for creating and manipulating tables are quite straightforward.

4.2.3. Table Visualization and Statistical Tools

Besides the pre-defined tools, new user-defined visualization and statistical tools can be built over tables thank to the open architecture of Rialto. For an instance, a statistic relative to a table is specified by instantiating its evaluation. Specifically, a statistic implements the following Java interface:

```
public interface Statistic extends RialtoPlugin {
    void check(TableMetadata)
        throws RialtoException;
    void evaluate(Table);
    Object getValue();
}
```

The first method is aimed at checking the compatibility of the statistic with the metadata associated with the underlying table. The second method computes

the statistic, which can be fetched by using the third method. Notice how the interface exploits the TableMetadata, Table and Value components from the core, which are detailed later in this paper. As an example, Code Fragment 2 is a simplified version of the Average statistic.

Code Fragment 2: the Average statistic

```
public class Average extends ParametersOwner implements Statistic {

  @ParameterName("Attribute index")
  @ParameterAttributeIndexDecorator(acceptableTypes = { AttributeType.DOUBLE,
      ↪ AttributeType.INTEGER })
  @ParameterMandatoryDecorator
  public final PluginParameter<Integer> attributeIndex = createParameter();

  private transient double value;

  @Override
  public void check(TableMetadata metadata) throws RialtoException {
    for (final Attribute attribute : metadata) {
      if (AttributeType.DOUBLE == attribute.getAttributeType()
          || AttributeType.INTEGER == attribute.getAttributeType())
        return;
    }
    throw new RialtoException("Table has no numeric attributes.");
  }
  @Override
  public void evaluate(Table table) {
    int count = 0;
    value = 0;

    for (final Row row : table) {
      value += row.getValue(attributeIndex.getValue());
      count++;
    }
    value = value / count;
  }

  @Override
  public Object getValue() {
    return value;
  }
}
```

Within Code Fragment 2, we can see how parameters are specified within plug-ins by means of the Java Annotation technology. The same technology is used to map the specific user interface components to the parameters: in our case, the annotation concerning acceptable types forces the interface to allow the sole selection of either integer or double attributes.

Table charts can be embedded within Rialto in a similar fashion, by providing classes which implement the following interface:

```
public interface Chart extends Visualizer {
    void check(Metadata) throws RialtoException;
    JComponent getChart();
    DataCharts getDataChart();
    void setTable(Table);
}
```

In practice, a chart in Rialto is an object capable of providing both a DataChart and a JComponent. The first element represents the relevant summary information which can be extracted from the associated table, whereas the JComponent is a Java Swing Object implementing the visualization metaphor for the information contained within DataChart. As an example, the DataChart associated with a Pie Chart subsumes a table with the frequency of each value relative to the attribute under consideration, and the JComponent represents the panel where the Pie is built starting from the DataChart. One key feature of the Data Perspective is the capability to ease the filtering process by materializing data manipulations, which are directly performed within the data perspective, in a visual fashion.

4.2.4. Table Bridges

Bridges are used to store tables and guarantee efficient and transparent access to their data. In practice, a bridge can be regarded as a decoupling mechanism between logical data and its storage, thus making tasks independent of the physical implementation of tables. Every table in a workflow has a bridge attached to it and, vice versa, each bridge may have one or more attached tables. The scenario is exemplified in Figure 8, where three different bridges are used

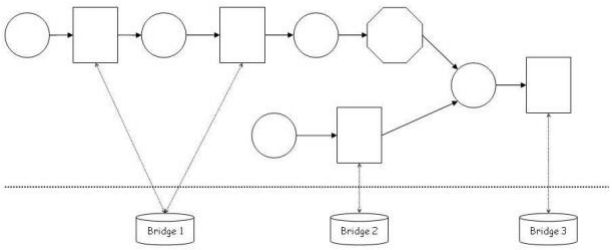

Figure 8: Bridegs within a workflow

behind the scene to physically store the contents of the four tables available within the workflow. Each bridge is implemented as a plug-in of the platform.

Looking at Rialto from a data perspective, we can see that bridges create a two-level architecture: a first, abstract logical level where tables and models are semantically represented and manipulated. A second, physical level where tables are physically hosted in *ad hoc* structures. Rialto supports both main memory and mass memory bridges, thus making it possible to handle data sets of very large size.

A main memory bridge is essentially a component which provides a physical representation of a table as a matrix. Thus, a table bridged on main memory is a collection of rows represented as arrays of double values, where each value represents either a numeric type or an index to a complex (nominal, string, EC) value, hosted in a specialized data structure. With reference to the Code Fragment 1 shown in Section 4.2.2, the *bridge.generateTable* method creates one such a table in main memory. The operations on the row access the array representation and store the values accordingly. By contrast, a mass memory bridge relies on a relational DBMS and exploits relational tables for storing and accessing data. Current mass memory bridges include both commercial DBMSs, namely, Oracle and MS-SQLServer, and Open DBMSs, namely, Postgres, MySQL and H2. Again, with reference to the data manipulation primitives of Section 4.2.2, operations on the table are translated into SQL statements, which are mediated

by an appropriate buffering strategy. Within a bridge, data structures and indexing methods are used to guarantee efficient implementation of the primitives for accessing and manipulating table components. For example, the access to an Event Collection (which can be seen as a non-normalized relation) can be optimized by relying on *ad hoc* structures based, for example, on inverted index structures.

The internal bridge behavior is exposed via easy APIs, enabling "fluent" declaration of complex filtering/sorting/randomization/etc. operations and, at the same time, optimal execution with respect to resources from the underlying storage medium. As an example, suppose that you want to manipulate table *customerTable*, by (a) selecting only rows having attribute *age* lower than *30*, (b) excluding rows with one or more event in event collection *debts*, and (c) sorting the resulting rows by descending values of attribute *income*. In Rialto this is expressed as in the following code snippet:

```
Table customerTable = ...
TableMetadata metadata = customerTable.getMetadata();
IntegerAttribute age = metadata.getAttribute("age");
EventCollectionAttribute debts = metadata.getAttribute("debts");
DoubleAttribute income = metadata.getAttribute("income");
TableView myView = customerTable.having(age).lessThan(30)
    .having(income).missing().sortBy(descending(income));
```

Here, we build a *TableView* object, which can be seen as a "normal" table in a workflow, except it is not materialized.

4.3. Models

Semantically, a model is an mathematical structure which subsumes intensional properties of a set of objects. Within the 2W, the objects are identified through entailment, whereas the properties are described through extension. Essentially, such properties can be induced by means of a specific mining algorithm, and can also be externalized by characterizing an entity according to

whether such properties hold for it. Models can be categorized as follows: Descriptive Models, which include Clustering and pattern models, and Predictive Models, which include classification, regression and outlier models.
In Rialto, a model exhibits a specific signature as follows:

- A table schema, which characterizes the set of objects to which the model itself applies;
- A build operator, which implements a specific induction strategy;
- A PMML representation, which can be used to instantiate the model itself.

The hierarchy of all possible models which can be instantiated is shown in Figure 9. Here, we can see that each node (representing a type of model) provides a number of methods which actually represent the extension property, and that can be exploited by the application tasks in order to apply a model to a table (as detailed later in the section).

4.3.1. Model Visualizers

Besides the pre-defined visualizers, new additional visualization tools can easily be integrated by exploiting an abstract representation of model features based on an extended PMML representation. That is, models are described by using an appropriate XML representation, and visualizers take this representation as input to produce the desired visualization. This way, new visualizers can be created (as plug-ins of Rialto), independently of the algorithms producing a specific model. The general interface that a custom plugin should implement is similar to the Chart interface, with the only difference that the checking is relative to a model rather than a table:

```
public interface ModelVisualizer extends Visualizer {
  void check(Model) throws RialtoException;
    JComponent generateComponent();
}
```

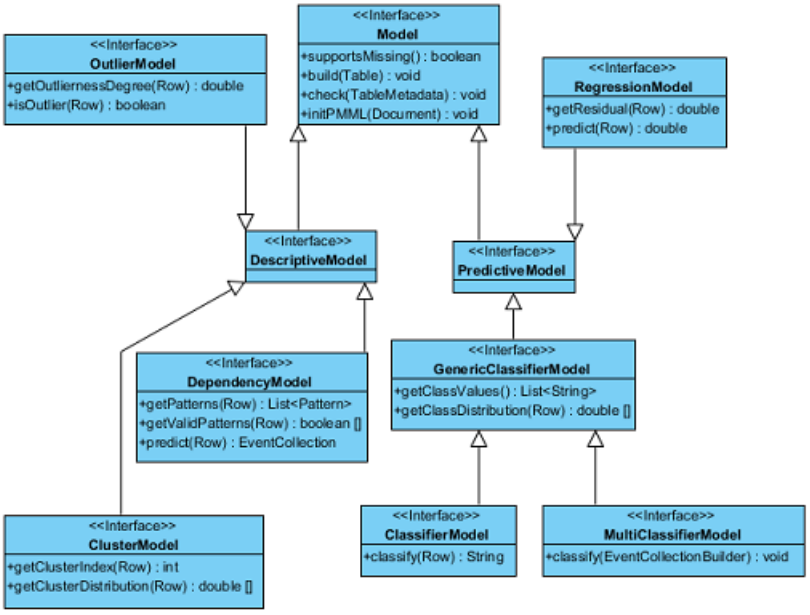

Figure 9: Model Hierarchy

4.4. Tasks

As we have seen in the previous sections, the algebraic operators of the 2W Model appear in a workflow as nodes of type *tasks*, that are implemented as Java plug-ins. Next we give some details about the different types of tasks.

Filtering tasks. They allow to acquire, export and modify tables and models. More specifically:

- *Acquisition* for importing tables or models from external sources. For example, we can exploit a CSV Acquisition task in order to acquire the content of a CSV file within a Rialto table, or a PMML representation of a decision tree into a model.

- *Export* for exporting tables and models to external destinations. For example, a CSV export task can pour the content of a table into a CSV

file, and a PMML model export can produce an XML file containing the abstract representation of a given model.

- *Table filtering*, for modifying tables into new tables with derived information. Example filtering tasks are selections of rows or columns, transformation of values, joining of multiple tables according to some criteria. SQL operations are supported natively within a SQL Filter, which performs a specified SQL query on the input table to produce a table where the result of the query is stored.

- *Model filtering* used to perform model post-processing. For example, a tree-based model can be transformed into a rule-based model.

Mining tasks relate tables to models. Mining tasks are responsible for invoking the *build()* method of the Model class (see top node in the hierarchy of Figure 9) according to a specific building strategy, namely, by simple mining, hold-out or cross-validation.

Application tasks enrich a table of entities with the properties which can be derived from the given model. By default, we devise two main application tasks, namely *Prediction Join* and *Description Join*. The basic idea is that a table and a model can only join if the underlying schemas are compatible (that is, the schema of the model is a subset of the schema of the table). The Prediction/Description distinction depends on the operators which can be used to enrich each entity with new properties. For example, a prediction join can be applied to a table and a classifierModel produces a new table where the metadata associated with the table is enriched by a further nominal attribute, which corresponds to the class which can be computed by the model on the tuples of the table. By contrast, a dependency join applies to a table and a DependencyModel returns a table with a series of binary attributes, where each attribute corresponds to the pattern associated with the model, and a true/false value which denotes whether the pattern holds for the specific row under consideration.

Again, the open architecture enables the definition of a*d hoc* tasks which can implement user-specific operations. Structurally, the signature of a task requires that a Task specifies some input and output ports, the metadata associated with each port and the implementation of the *execute()* method. Code Fragment 3 describes the code snippet for an example task that takes two tables as input, and returns as output a new table resulting as the merge of the input tables.

Code Fragment 3: An example Task

```
public class MergeTables implements Task {

    @Input(name = "First Table")
    @Order(0)
    private Table firstTable ;

    @Input(name = "Second Table", optional = true)
    @Order(1)
    private Table secondTable;

    @Output(name = "Output Table")
    @Order(0)
    private WriteableTable outputTable;

    @PreconditionCheck
    public void check() {
        if (secondTable != null
                        && !firstTable.getMetadata() != secondTable.getMetadata()) {
            throw new RialtoException("Metadata differ on the input tables");
        }
    }

    @Override
    public TableMetadata[] getAllTargetMetadata() {
            return new TableMetadata[] {
               firstTable.getMetadata().createBuilder().build()
            }
          }
    @Override
    public void execute() {
        for (Row row: firstTable ) {
            Row newRow = outputTable.getNewRow();
            newRow.copyFrom(row);
            outputTable.appendRow(newRow);
```

```
        }
        for (Row row: secondTable) {
            Row newRow = outputTable.getNewRow();
            newRow.copyFrom(row);
            outputTable.appendRow(newRow);
        }
    }
}
```

4.5. Core

The Core component provides three basic functionalities, namely, *Workflow Execution*, *Data Handling* and *Parallelization*.

Workflow execution. Rialto includes an engine for the efficient execution of workflows. The workflow execution layer implements scheduling policies and buffering strategies. The latter are aimed at "on the fly" processing of data streams.

Data Handling. Rialto includes techniques for the efficient management of data in main memory (under both main memory and mass memory bridges, in the latter case relative to the portions of data which is buffered into main memory). To this end, lossless data compression techniques, namely the LZ (Lempel-Ziv) algorithms (Ziv and Lempel, 1977), are used.

Parallelization. Rialto provides support for three kinds of parallelism: parallel execution of independent workflow branches, parallel table scan and pipelining. Parallel execution of workflow branches is quite straightforward, and it is enabled by static analysis of the workflow graph. Within parallel table scan, large tables are automatically partitioned and each packet is assigned to a different processor. The underlying paradigm here is that of processing single packets and then merging the produced results. To this purpose, Rialto provides a native object, namely *TaskExecutionSupport*, which can be exploited within tasks or statistics. This object autonomously implements the most suitable policy for processing a table, provided that a native method for processing single rows is provided. Code Fragment 4 shows a naive task implementing the copy of an input table to an output table. We can observe two components here: an

alternate version of the *execute* method, which uses a *TaskExecutionSupport* object, and a local object implementing a *RowProcessorBuilder* interface. The latter basically defines a method for processing an input row and producing an output row out from it. Within the *execute* method, a request is issued to the *taskExecutionSupport* object to scan the input table, process all rows using the *RowProcessor* object, and write the results to the output table. The policy for scanning the table and writing the results is totally transparent, as it is handled at system level by packetizing the table and processing each packet in parallel.

With reference to Code Fragment 3, it is possible to rewrite the *execute* method by exploiting the same row processor and a *TaskExecutionSupport* object. In practice, lines 29-38 of the method can be replaced by the lines

```
taskExecutionSupport.scan( firstTable ).with(new MyRowProcessorBuilder()).writingTo(outputTable);
taskExecutionSupport.scan(secondTable).with(new MyRowProcessorBuilder()).writingTo(outputTable);
```

thus demanding the management of the of the operations to the kernel (and thus exploiting the native parallelism).

Note that argument of *scan* can be any table which is "readable": for example a table view generated through the bridge APIs would fit as well. The following code snippet is a refinement of the previous one, where tables are randomized prior to start the scan.

```
taskExecutionSupport.scan( firstTable .randomize()).with(new
    ↪ MyRowProcessorBuilder()).writingTo(outputTable);
taskExecutionSupport.scan(secondTable.randomize()).with(new
    ↪ MyRowProcessorBuilder()).writingTo(outputTable);
```

Pipelining can be seen as an enhancement of the above strategy. Within pipeling, each row of a table is forwarded to the next task as soon as it has been processed by the current task. Pipes can be assigned to multiple processors.

Pipelining is relative to tasks, and not all tasks can support it. It is up to the user to specify (and guarantee) whether the task can be piped, by overriding the method *canBePiped*. With reference to Code Fragment 4, it is possible to enable pipelining simply by adding the following:

```
  @Override
  public boolean canBePiped() {
    return true;
  }
```

When Rialto detects that two or more task instances, having workflow connections, can work in pipe, the latter are automatically replaced by a single virtual task joining their behavior, but avoiding the burden of maintaining temporary tables working as "buffers" between tasks. This can be especially effective when such temporary, intermediate tables are very large.

Code Fragment 4: A simple task which seamlessly processes rows in parallel by exploiting Rialto's API

```
  ...
  @Override
  // Core task method: Rialto expects the whole task logic is called from here
  public void execute(final TaskExecutionSupport taskExecutionSupport) {
    // The following line asks for copying the whole input table
    // to output table via MyRowProcessorBuilder; Rialto automatically
    // handles input/output buffering, task parallelization, and so on.
    taskExecutionSupport.scan(getInputTable()).with(new
        ↪ MyRowProcessorBuilder()).writingTo(getOutputTable());
  }

  private static final class MyRowProcessorBuilder implements RowProcessorBuilder {
    @Override
    public RowProcessor buildNew() {
      return new AbstractRowTransformer() {
        @Override
        public void process(final Row inputRow, final Row outputRow) {
          // In this simple example output row is copied from input row, so it has the same content,
          // but Rialto API can be used to manipulate output accordingly
          outputRow.copyFrom(inputRow);
        }
      }
    }
  }
```

5. Expressivenes by examples: Rialto in Action

The purpose of this section is to show how Rialto works on different classes of applications. To this end, we provide one case study on customer segmentation, and another on text classification.

5.1. Case Study 1: Marketing Campaigns

ACME Household Cleaning Supplies manufactures and sells a wide array of household cleaning supplies over the Internet. They have a very large customer base for which they have gathered information about customers through product registration cards and customer satisfaction surveys. Annually, ACME mails holiday cards with a promotional voucher for a free sample product. In order to reduce mailing costs and estimate the total cost of the promotion, ACME wants to know which customers in their database are most likely to redeem the vouchers. To this end, the workflow depicted in Figure 2 was developed.
Data Acquisition. The first step in almost every data mining workflow is the importing of historical data. ACME's data is contained in a comma-delimited file, which can be imported using the CSV Acquisition task.
Data Preparation. Rialto comes prepackaged with many of the most common preprocessing tasks. For the purpose of this case study, a first data preparation task is concerned with missing values. To do so, we use the Missing Values statistical tool in Rialto. Once executed, Rialto determines that there are 431 missing values in the WORKCLASS attribute and 433 missing values in the OCCUPATION attributes. Now that we have identified that the data is missing values, we apply the Fill Missing Values task to automatically populate the missing data. To this end, Rialto uses the most common values found in WORKCLASS and OCCUPATION to replace the missing values. The second data preparation task is that of putting data in buckets. Indeed, ACME tracks the ages of their customers. ACME's sales department thinks of customers in the following age groups: under 25, 26 to 35, 36 to 60, and 61+. There are several ways to place this data into buckets, one of which is utilizing Rialto's

Custom Discretize task.

Data Mining. Now that data is prepared, we want to perform data mining on it to see if an accurate predictive classifier for ACMEs marketing campaign can be created. By using the Hold Out task, we separate data into a training set and a test set, the latter allowing us to validate the accuracy of results. The EC4.5 algorithm (an extension to the commonly used C4.5 algorithm) is then run to produce a predictive classifier for ACME customers.

Model application. To test the quality of the learned model, the Prediction Join task is applied to the test set. Then, the Confusion Matrix is computed, along with the statistical accuracy of the model. Also, the resulting table *Table5*, where the predicted labels are compared with those of the ideal classification, is *exported* as a CSV file.

Once the trained classifier is available, it can be applied to determine which new customers should receive the promotional mailer. To this end, a new workflow, to be deployed over Rialto server, is constructed. Of course, the predictive classifier expects the data to be prepared in the same manner in which it was trained. Therefore, data representing new customers must again be discretized using the same ranges as the original data. ACME can now use classification results to assess the cost of running a promotion with new customers. Now, instead of sending out the promotions to all the new customers, ACME can confidently send out the promotion to 169 of the total 1500 new customers. A savings of eighty-nine percent!

5.2. Case Study 2: Text Classification

The application scenario is as follows. We want to induce a model for classifying a set of medical abstracts from the OHSUMED data collection (Hersh et al., 1994), a subset of the bibliographic database MEDLINE consisting of peer-reviewed medical literature maintained by the National Library of Medicine. To this end, we have available a subset of pre-classified documents, along with two learning algorithms, namely, Complement Nave Bayes (Rennie et al., 2003) and MaxEntropy (Nigam et al., 1999). Our goal is to compare how well they

perform, in order to select the best one (over the given data).

The OHSUMED subset we use for training is the collection consisting of the first 13,929 documents, from the 50,216 medical abstracts of the year 1991, labelled by the 23 diseases categories of the C codes of the Medical Subject Headings (MeSH) thesaurus (REF).

The KD process consists in the acquisition of the training documents, their pre-processing, models learning by CNB and MaxEnt and, finally, comparison of the induced models. The collapsed learning workflow (i.e., where table nodes are hidden) implementing this process is shown in Figure 10.

Data Acquisition. The input documents are represented as lines of a CSV file. Each line contains the document text (string) and the class labels. Data are acquired through CSV Acquisition task. Since features spaces in text categorization are typically very large (the high dimensionality problem) and, in addition, each document contains only a small subsets of features, it turns out that a document vector is normally largely sparse (i.e., too many 0s occur). Thus, a standard feature vector representation of texts would be highly inefficient. To overcome this drawback, the terms of each document are stored by an event collection (see Section 4.2.1)

Data Preparation. For the purpose of this study, we consider the following pre-processing steps:

- *N-gram Extraction* includes both stop-words and stop-tokens removal, and n-grams of up to 3 stem words are generated. This choice originates from the observation of the presence, in the medical vocabulary, of long phrases such as "immunologic deficiency syndromes". Of course, the meaning of "immunologic deficiency syndromes" is very different from that of "deficiency" alone.

- *Concept Extraction* (CE) is performed by using the subset of the MeSH thesaurus made of the concept hierarchy having the 23 diseases categories (C codes) as roots. The advantage of CE is that, for example, terms like "arrhythmia" and "heart aneurysm" can merge into the most general

concept "heart diseases", thus getting a stronger feature, able to better discriminate between the MeSH "cardiovascular diseases" category and the others.

- *Feature Selection* (FS) is performed by selecting the 500 highest scoring terms (a term is either a ngram or a concept) according to the Information Gain function (Forman, 2003). The advantage of applying FS to both grams and concepts is that of selecting discriminating terms with both statistical and semantic qualities (Lewis, 1992).

Preliminarily to the above steps, the *Remove Attributes* filter is applied to eliminate the attribute *Document_Name* (it has no effect on classification performance). After *Ngram Extraction* and *Concept Extraction* have been executed in parallel, the ngram-based and the concept-based document representations are merged by the *Tables Join* node, so as each document is represented in terms of both ngrams and concepts (both stored by event collections). The *Ngram Extraction* filter is set through a configuration mask providing stop-words and stop-tokens lists, maximum ngrams length, etc.. The *Concept Extraction* filter is set through a configuration panel whereby the MeSH thesaurus is provided (as an XML file).

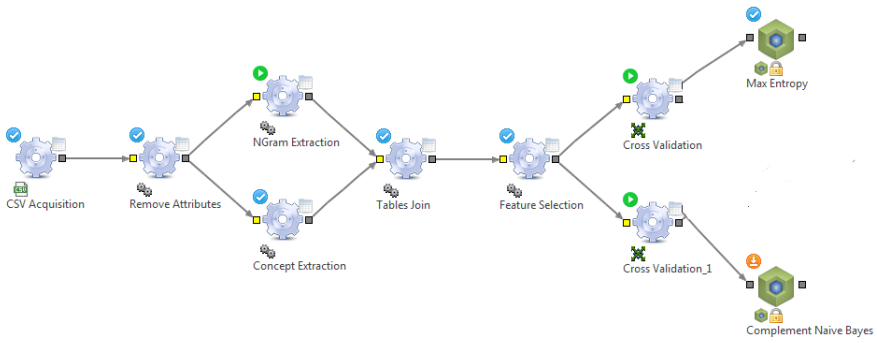

Figure 10: Compact representation of a workflow comparing two text classification algorithms

Data Mining. This step is performed by using both the Complement Nave

Bayes and MaxEntropy text classification algorithms. To validate and compare the predictive capabilities of the induced models, five-fold cross validation is performed. Overall classification performance are determined by calculating Precision, Recall, and F1-measure metrics.

5.3. Discussion

The two case studies above were aimed at giving a flavor of how KD processes can be accomplished by using Rialto. To this end, we used some of the pre-packaged plug-ins for data acquisition, pre-processing, mining, etc., and showed how they were connected to create KD workflows (one working on relational data and one on texts). As we have seen, both workflows were characterized by quite standard tasks: acquisition from CSV files, classification models induction performed by classical algorithms (i.e., C4.5, Complement Naive Bayes and MaxEntropy, the latter two for text classification), and data preparation, which confirmed to be the most time consuming phase of KD - in particular, text pre-processing in Case Study 2 consisted of five elementary tasks (from Attribute Removal to Feature Selection - see Figure 10). Though not reported in the above description for space reasons, the results of the two analyses can be inspected by several view plug-ins, displaying data and models in diverse ways. For an instance, one could display the table (not shown in Figure 10) obtained by joining the output tables of the Ngram Extraction and Concept Extraction tasks to inspect the bag of n-grams and the bag of concepts representing each processed document (both bags stored as attributes of type Even Collection).
Different variants of the above workflows could be created in order to improve the quality of the respective analyses. For instance, frequency analysis could be performed in Case Study 2, to keep the highest frequent terms, simply by adding a suitable node to the workflow of Figure 10 (Rialto provides plug-ins for the computation of the most important frequency measures). Or, for another example, more advanced feature extraction techniques, such as those based on dependency graphs (Abdulsahib and Kamaruddin, 2015), (Vilares et al., 2015), could be carried out to make machine learning methods better generalize. This

goal could be achieved by replacing in the workflow of Figure 10 the n-grams extractor node by a node implementing dependency-graph-based techniques. Both these examples show how the plug-in based architecture of Rialto makes the construction of a KD analysis a simple process, where tasks (nodes) can be assembled, removed, replaced, so allowing for quick and interactive changes to the analysis.

6. Discussion and Evaluation

We refer to (Chen et al., 2007), which identifies the following dimensions along which a data mining system should be evaluated (see also Section 1):

- *Managing large data volumes*: is the tool capable of manipulating large data volumes? Does this require changes in the architecture?
- *Acquisition from data sources*: can the tool gather data from diverse data sources, such as databases, Excel, CSV files, etc.?
- *Mining models*: is the choice of mining algorithms available in the tool rich enough to support real-world data mining applications?
- *Data preprocessing*: does the tool offer support to pre-process data before applying mining algorithms? Is there some guidance in the process?
- *Data visualization*: has the tool some efficient data visualizations, in order to support decision making and informed data manipulation?
- *Open Architecture*: is the tool extendable? Is the architecture open? Is there a development environment?
- *Open Community*: is there a thriving user community? Does it have the ability to develop new features and/or fix bugs?

Figure 11 provides a picture of the positioning of Rialto relative the above-mentioned dimensions.

	Description
Managing large amounts of data	Bridges paradigm allows Rialto to transparently use mass memory or databases, thus allowing the processing of large data volumes without changing the system architecture.
Acquisition from data sources	Rialto acquires dataset in CSV, Excel, ARFF, and others. Rialto also exploits the JDBC API, allowing import of data from database management systems (DBMSs) such as Oracle, MySQL, SQL Server, PostgreSQL, and many others.
Mining models	Rialto provides several algorithms for classification and clustering, like SSEM, Naive Bayes, Maximum Entropy, EC4.5, Rule Learner, CBOD, Travel CBOD, Dolphin, Linear Regression, EM, K-Means, Association Miner.
Data preprocessing	Rialto provides a wide variety of functionalities for data manipulation through a number of predefined plug-ins implementing filter tasks. In addition, it offers a number of facilities at Data Perspective level, where the choice of the available tools is guided through contextual menus.
Data visualization	The system offers a number of graphical tools for data and model visualization; new ad-hoc visualizers can be built and integrated, if necessary.
Open Architecture	Rialto provides a powerful and intuitive API for developing new plugins. Thus, new types of statistics, tasks, models, visualizers, etc., are easily implemented. The new plug-ins transparently take advantage of all the mechanisms of parallelism, pipelining, and memory management. The development environment of Rialto is based on opensource Eclipse IDE.
Open Community	Rialto does not have an open community, as it is not distributed in open source. However, it is widely used in academic environments, and this involves a continuous tool evolution, along with a growing availability of plugins.

Figure 11: Rialto Evaluation

6.1. Experimental Analysis

In this section we report the experimentation performed with the aim of assessing a number of architectural choices affecting the capability of Rialto to scale to large data volumes. To this end, there are some specific features we have analyzed:

- How bridges affect the performance. Since the architecture strongly relies on the idea of semantic abstraction, the coupling between the semantic part and the data engine has to be carefully analyzed. In particular, with data which are disk-resident, it is crucial to quantify the overhead with respect to memory-resident data, and to make sure that such an overhead is not affected by the data size.

- Pipelining. Since pipes can be assigned to multiple processors, it is worth measuring how enabling this feature within the tasks improves the overall performance.

- Scalability of the parallel table scan. The *TaskExecutionSupport* component provides mechanism for parallel processing of elementary user-defined operations on tables. However, it introduces some overheads that need to be quantified.

- Event Collections. These components are particularly suited for managing high-dimensional data. We perform some comparative measurements on memory usage and efficient support to native filtering operations.

It is worth noticing that these choices are a direct consequence of the underlying model, which enables modularity in the execution of data mining queries. Thus, the experiments are aimed at demonstrating that those architectural choices actually optimize the executions, and that they can be profitably exploited in suitable query evaluation plans based on the underlying algebra.

Bridges. In a first set of experiments, we exploit a simple workflow which loads a table and builds a model on such a table. We use a simple Naive Bayes Model, which scans the input table to collect statistics. Figure 12 reports the

performance (time in ms) of three different bridges for increasing size of the input data. We compare the Rialto's in-memory bridge, and two SQL-based bridges. One is based on the H2 open source java database engine[1]. We instantiate the bridge using the in-memory table storing of H2. Thus, the H2 bridge is just an alternative memory bridge which supports SQL queries natively. The other SQL-based bridge relies on the widely known commercial MySQL engine[2].

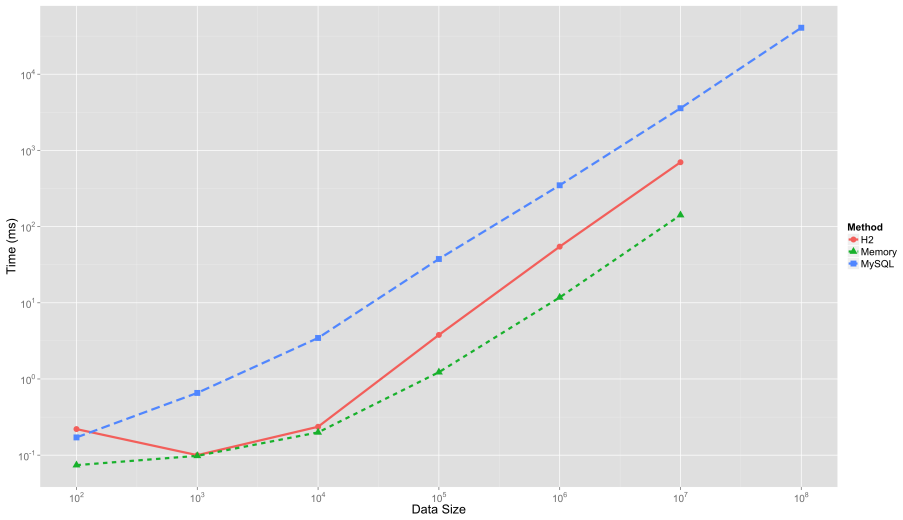

Figure 12: Bridges comparison - Rialto memory bridge (lower dotted line) vs H2 bridge (solid line) and MySQL bridge (upper dotted line)

How we can see from the graph, there is a constant factor which differentiates the time performances of the different bridges (so, the overhead is not affected by the data size). Also, notice that the in-memory bridges suffer of memory limitations, as with the current experimental configuration they are only capable of handling tables with at most 10^7 tuples, whereas the same limitation does not hold for the MySQL bridge.

Pipelining. In a second set of experiments, we measure the effectiveness of

[1] See `http://www.h2database.com` for details.
[2] See `http://www.mysql.com`.

pipelining. To this purpose, we setup a simple workflow which exploits the Naive Bayes model built in the previous set of experiments, and applies it to a new table. Thus, the piped tasks in this scenario are *data acquisition* followed by *predictive join* (Naive Bayes is applied to the acquired table). Figure 13 shows

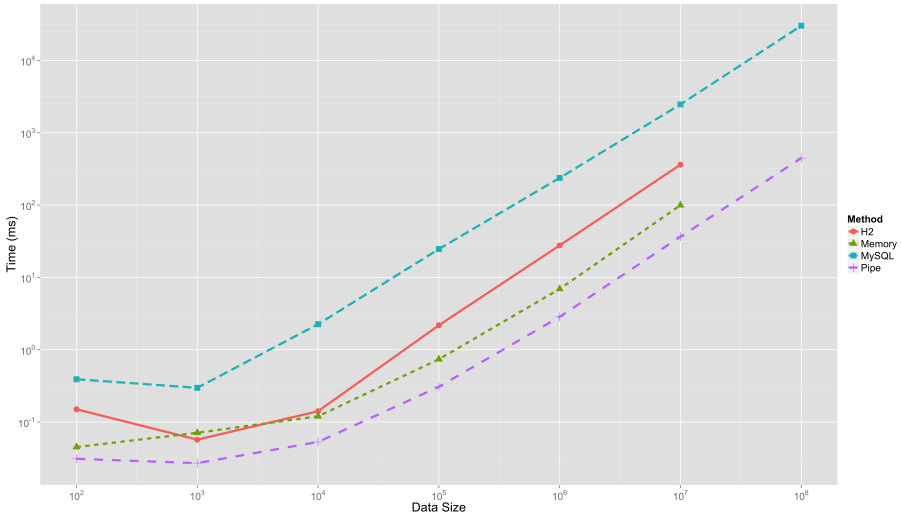

Figure 13: Time performance of piped execution (lower dotted line) vs sequential execution (the three upper lines are those reported in Figure 12).

the performance of the piped version of the tasks (bottom line) compared to the sequential versions (where the predictive join is run after the input table has been *entirely* created) obtained by using the same bridges of the previous experiments. There are a few aspects that are worth being mentioned. First, the pipelining clearly outperforms all other methods in terms of efficiency. Further, since intermediate tables can be deemed as volatile within a workflow, the fragment of the workflow which makes use of pipelining is essentially independent from the size of the input tables. This can be appreciated within the graph by noticing that the pipe (bottom line) does not suffer from memory limitations of the other in-memory methods (lines in the middle): tuples are passed away as long as they are processed, and hence the only memory requirement is the need to store a single tuple.

Parallelism. The next set of experiments measures the speed-up of the parallel table scan. To this purpose, we implemented a filtering task which takes a table in input, and produces the same table as output. Within the *RowProcessBuilder* we inserted a delay mechanism which simulates time consuming operations on a single row. The length of the delay is parameterized and ranges from $0\mu s$ to $300\mu s$ in our experiments.

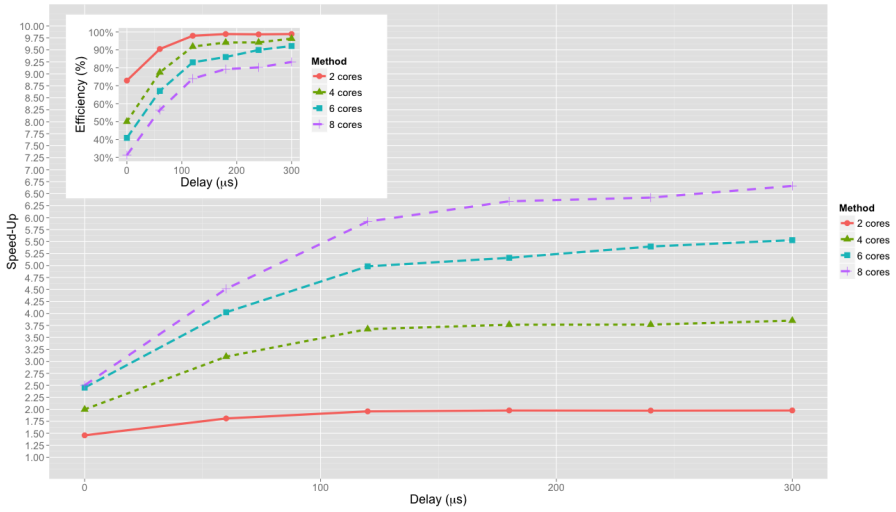

Figure 14: Parallel table scan speed-ups in main memory with 2-, 4-, 6- and 8-core processors

Figure 14 shows the speed-up achieved by running the task on different architectures. The baseline is a processor with a single core, and the figure shows the speed-ups with processors having number of cores equal to 2 (bottom line), 4, 6 and 8 (upper line). We can see that, in general, (1) the higher the delay, the higher the speed-up with a given core configuration, and (2) the higher the number of cores, the lower the speed-up. In particular, we can observe that the highest speed-up is obtained with a 2-core processor (varying from 1.50 to nearly 2), while the maximum speed-up of the 8-core configuration is 6.75. In the left upper corner, the figure shows the efficiency in the exploitation of the different core configurations. The efficiency represents the percentage of proximity to the ideal situation: intuitively, a 2-core configuration is 100% efficient if it is

twice faster than the single-core configuration (speed-up=2). Similarly, 4-cores should be 4 times faster, and so on. We can notice that efficiency increases as long as delay increases and, in general, efficiency values are quite satisfactory - ranging from 80% (8 cores) to 100% (2 cores).

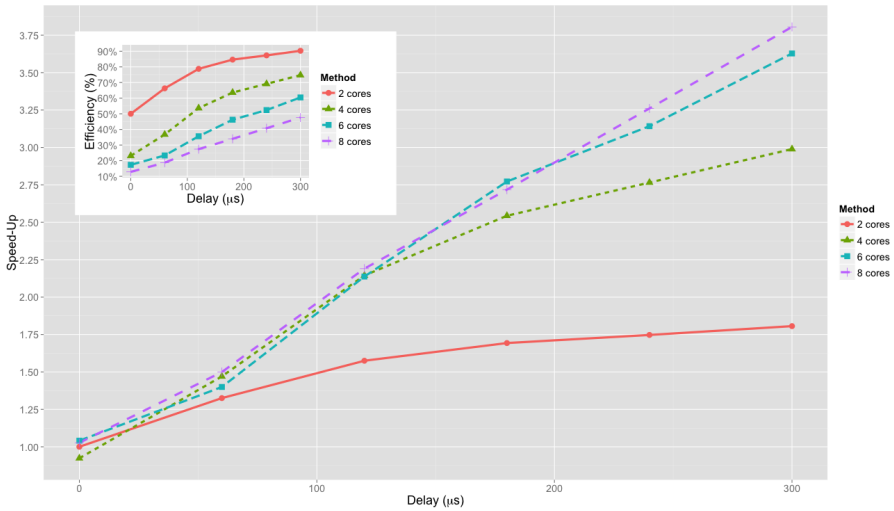

Figure 15: Parallel table scan speed-ups in external memory with 2-, 4-, 6- and 8-core processors

Figure 15 shows the speed-up when an out-of-core bridge is exploited. The situation in this case is more problematic, since the retrieval and saving of the tuples clearly requires access to external memory. The speed-ups in this case are halved and for increasing cores the efficiency cuts significantly.

Event Collections. There are essentially two aspects we need to face, namely, memory consumption and efficiency of native filtering operations. In order to stress the system on the above aspects and to measure its response, we exploited some synthetic datasets, containing $10,000$ transactions each. A transaction consists of a set of items in the form $\{i_1, \ldots i_n\}$ where i_j uniquely identifies an item picked from an initial set $\mathcal{I}$. These datasets simulate several real-life situations, like e.g. purchase transactions, where $\mathcal{I}$ represents the set of possible items which can be purchased, and each transaction represents a set of

purchases relative to an individual; or textual data, where $\mathcal{I}$ represents a vocabulary and a transaction represents a text document, irrespective of the order of words. In the generation process, we vary the size of $\mathcal{I}$ from 10^2 to 10^6. Then, each dataset is generated by repeatedly performing, for each transaction, the following steps: (1) sample the size of the transaction through a Poisson distribution with fixed λ parameter, (2) sample a Multinomial distribution on $\mathcal{I}$ from a Dirichlet distribution with fixed prior α, and (3) populate the transaction by repeatedly samping from the multinomial distribution sampled in the previous step.

The datasets were generated in a relational format as text files where each line represents the pair `(TransactionID, {ItemID})`. We then built a workflow which acquires the text file and builds a table T, where each row represents a transaction. There are two alternative ways of representing the items of a single transaction in T. Either as a binary tuple with fixed length $|\mathcal{I}|$, or as an event collection. We built two alternative versions of the workflow, were the acquisition task was customized to produce either of the two representations. Next, the workflow proceeds with a filtering task which aims at removing the top 10% more frequent items.

The rationale of this workflow is twofold. First of all, by monitoring the acquisition task, we can measure the memory usage in both representations. The binary representation (referred to as *Full Table* in the following) acts as a full matrix. By contrast, the representation of a transaction as an event collection (referred to as EC) allows us to measure the compression ratio and the possible overhead due to internal indices required for the event collection. Figure 16 plots the memory usage after the loading of the transactions for both representations. To this end, both the in-memory and the H2 bridges are used. We can observe that the memory usage of the EC version of the table is low and substantially independent from the size of the dataset $\mathcal{I}$, whereas the Full Table representation grows very fast with it (upper line). That is, ECs enable an effective and compact representation of very large sets of transactions.

Now, let us consider the filtering operation removing frequent items from

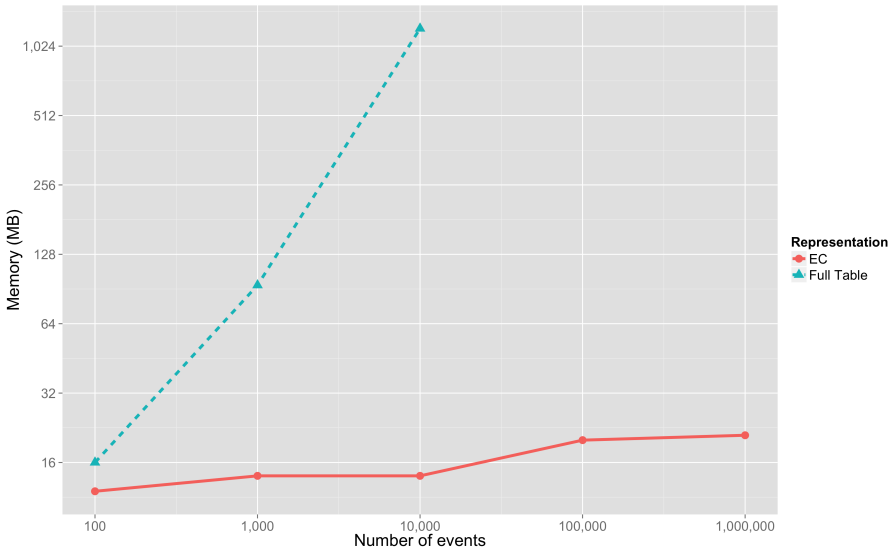

Figure 16: Memory usage when transactions are represented as (1) binary tuples with fixed length (dotted line) and (2) event collections (solid line).

$\mathcal{I}$. In principle, event collections should be more sensitive to such an operation: removing a column from a table should be straightforward, whereas removing an event from a collection requires searching for such an event. Thus, this simple filtering operation is a good test for assessing the performance of ECs manipulations. We can see the performance of the removal operation in Figure 17. Again, the scan of the EC representation is quite constant. By contrast, removing a column on the Full Table representation requires to restructure the metadata associated with the table, which is a costly operation when the number of attributes is overwhelming high.

7. Ongoing Work

Besides the above described features, there are also a number of features that we are currently working on in order to enhance the Rialto capabilities, notably:

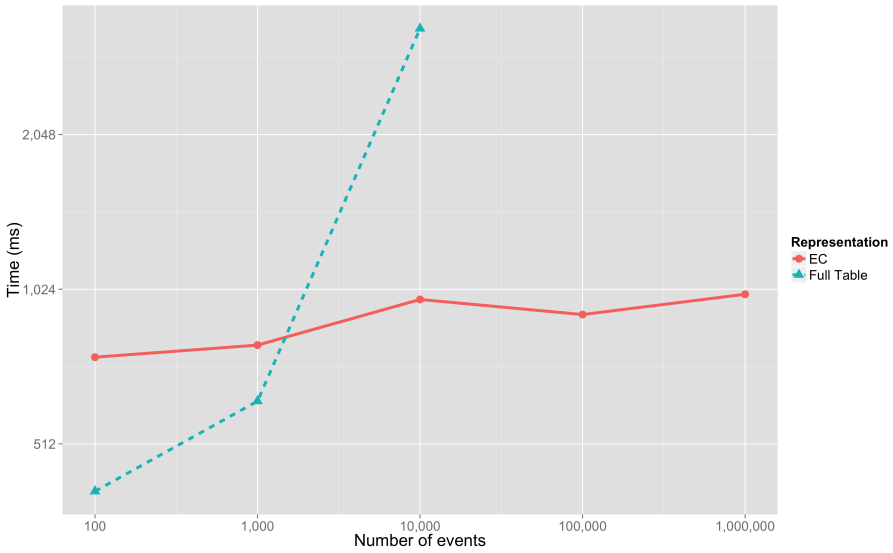

Figure 17: Filtering operation costs in case transactions are represented as (1) binary tuple with fixed length (dotted line) and (2) event collections (solid line)

- *Data Streams.* In the current implementation, we assume that data within Rialto is made of either tables or models. However, in the same paradigm, we can extend such basic types to comprise data streams, i.e., memoryless tables with a timestamp associated. In practice, in our model a stream has the same features of a table (metadata and rows). However, the content of the table changes over time and it is relative to a specific time window. Under this perspective, the paradigm of tasks has to be reconfigured as well, to enable transformation from streams to tables and models, and viceversa.

- *Spatio-temporal data.* Besides the current types, we are working on extending attributes types to support spatial and spatio-temporal data. This is especially important for complex applications that require the analysis of geo-reference data or moving objects.

- *Support to Hadoop.* The simple form of parallelism which are provided

to the user, namely parallel table scan and pipelining, can be further enhanced by considering the current technologies. We are currently working on extending the Rialto architecture to support the MapReduce programming model, by developing bridges under the Hadoop framework.

8. Conclusions

In this paper we have first proposed a general framework for modeling KD processes, named 2W Model, a variant of the 3W Model proposed in (Johnson et al., 2000). According to this model, the essence of a KD process can be regarded as the interaction between two separated worlds: the data and the model world. This way, any knowledge discovery process can be formalized as an algebraic expression, that is essentially a composition of operators representing (pseudo)elementary operations on the two worlds.

Then, we have described a KD platform, called Rialto, which relies on the 2W Model. To this end, it provides a visual interface whereby a KD process can be modeled as a workflow, where each node is either a table (data world), or a model (model world) or a task representing an operator allowing a transition between the two worlds. Using this metaphor, Rialto provides support to all steps of a KD process, from data acquisition to model exploration and deployment. In addition, in order to cope with real-world analytical problems, where new *ad hoc*, specific tasks may be needed, Rialto allows incorporation of new (Java) plug-ins within an open architecture.

Efficiency and scalability are achieved thanks to suitable design choices including bridges, event collections and parallelism. We have conducted a number of experiments showing the effectiveness of the proposed architecture to scale to large volumes of data.

Download of Rialto can be found at `http://www.exeura.eu/products/rialto`.

References

Abdulsahib, A. K., Kamaruddin, S. S., June 2015. Graph based text representation for document clustering. Journal of Theoretical and Applied Information Technology 76 (1).

Berthold, M., et al., 2009. Knime: The konstanz information miner. SIGKDD Explorations 11 (1).

Blei, D., Ng, A., Jordan, M., March 2003. Latent dirichlet allocation. Journal of Machine Learning Research 3, 993–1022.

Borgelt, C., 2009. Graph mining: An overview. In: Proc. 19th Workshop on Computational Intelligence. KIT Scientific Publishing.

Calders, T., Lakshmanan, L. V. S., Ng, R. T., , Paredaens, J., 2006. Expressive power of an algebra for data mining. ACM Transactions on Database Systems 31 (4), 1169–1214.

Chaudhuri, S., Narasayya, V., Sarawagi, S., 2002. Efficient evaluation of queries with mining predicates. In: Proc. 18th International Conference on Data Engineering (ICDE'03). pp. 529–540.

Chen, X., Ye, Y., Williams, G., Xu, X., 2007. A survey of open source data mining systems. Lecture Notes in Computer Science 4819, 3–14.

chung Fu, T., 2011. A review on time series data mining. Engineering Applications of Artificial Intelligence 24 (1), 164–181.

Forman, G., 2003. An extensive empirical study of feature selection metrics for text classification. Journal of Machine Learning Research 3, 1289–1305.

Gaber, M. M., Zaslavsky, A., Krishnaswamy, S., 2005. Mining data streams: A review. ACM SIGMOD Record 34 (2), 18–26.

Giannotti, F., Manco, G., Turini, F., 2004. Specifying mining algorithms with iterative user-defined aggregates. IEEE Transations on Knowledge and Data Engineering 16 (10), 1232–1246.

Hall, M., et al., 2009. The weka data mining software: An update. SIGKDD Explorations 11 (1).

Han, J., Kamber, M., 2001. Data mining: concepts and techniques. Morgan Kaufmann.

Hersh, W., Buckley, C., Leone, T., Hickman, D., 1994. Ohsumed: An interactive retrieval evaluation and new large text collection for research. In: Proc. 17th ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR'94). pp. 192–201.

Hristovski, D., Friedman, C., Rindflesch, T. C., Peterlin, B., 2008. Literature-based knowledge discovery using natural language processing. Information Science and Knowledge Management 15, 133–152.

Imielinski, T., Mannila, H., 1996. A database perspective on knowledge discovery. Communications of the ACM 39 (11), 58–64.

Imielinski, T., Virmani, A., 1999. MSQL: A query language for database mining. Data Minining and Knowledge Discovery 3 (4), 373–408.

Johnson, T., Lakshmanan, L. S., Ng, R. T., 2000. The 3w model and algebra for unified data mining. In: Proc Int. Conf. on Very Large Databses (VLDB'00). pp. 21–32.

Lewis, D. D., 1992. An evaluation of phrasal and clustered representations on a text categorization task. In: Proc. 15th ACM Int. Conf. on Research and Development in Information Retrieval (SIGIR'92). pp. 37–50.

Manco, G., et al., 2008. Querying and reasoning for spatiotemporal data mining. In: Mobility, Data Mining and Privacy - Geographic Knowledge Discovery. Springer, pp. 335–374.

Mikut, R., Reischl, M., 2011. Data mining tools. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 1 (5), 431–443.

Nigam, K., Lafferty, J., McCallum, A., 1999. Using maximum entropy for text classification. In: Proc. IJCAI-99 Workshop on Machine Learning for Information Filtering. pp. 61–67.

Ortale, R., et al., 2008. The DAEDALUS framework: progressive querying and mining of movement data. In: Proc. 16th ACM SIGSPATIAL Int. Symp. on Advances in Geographic Information System. p. 52.

Pietramala, A., Policicchio, V. L., Rullo, P., Sidhu, I., 2008. A genetic algorithm for text classification rule induction. In: Proc. European Conf. on Machine Learning and Knowledge Discovery in Databases (ECMLPKDD'08). Vol. 5212 of Lecture Notes in Computer Science. pp. 188–203.

Quinlan, J. R., 1987. Generating production rules from decision trees. In: Proc. 10th International Joint Conference on Artificial Intelligence (IJCAI'87). pp. 304–307.

Rennie, J. D., Shih, L., Teevan, J., Karger, D. R., 2003. Tackling the poor assumptions of naive bayes text classifiers. In: Proc. Int. conf. on Machine Learning (ICML'03). pp. 616–623.

Rullo, P., Policicchio, V., Cumbo, C., Iiritano, S., August 2009. Olex: effective rule learning for text categorization. IEEE Transactions on Knowledge and Data Engineering 21 (8), 1118–1132.

S., M., 2005. Data Streams: Algorithms and Applications. Now Publishers Inc.

Sebastiani, F., 2002. Machine learning in automated text categorization. ACM Computing Surveys 34 (1), 1–47.

Shearer, C., 2000. The crisp-dm model: The new blueprint for data mining. Journal of Data Warehousing 5, 13–22.

Vilares, M., Fernandez, M., Blanco, A., November 2015. Supporting knowledge discovery for biodiversity. Data and Knowledge Engineering 100, 34–53.

Wang, H., Zaniolo, C., Luo, C., 2003. ATLAS: A small but complete SQL extension for data mining and data streams. In: Proc. Int. Conf. on Very Large Databases (VLDB'03). pp. 1113–1116.

Ziv, J., Lempel, A., 1977. A universal algorithm for sequential data compression. IEEE Transactions on Information Theory IT-23, 337–343.