EC Project 257859

Risk and Opportunity management of
huge-scale BUSiness communiTy cooperation

D2.1 Report on ROBUST Query Algebra and Core Operators

30 Oct 2011

Version: 1.4

Alexander Löser, Christoph Boden, Sebastian Schelter

[firstname.lastname]@tu-berlin.de

Database Systems and Information Management Group (DIMA), TU Berlin

Alexander Löser, TU Berlin DIMA

Christoph Boden, TU Berlin DIMA

Sebastian Schelter, TU Berlin DIMA

Falk Brauer, SAP AG

Adrian Mocan, SAP AG

Inbal Ronen, IBM Research – Haifa

Toby Mostyn, Polecat

Dissemination Level:

PU – Public

Executive Summary

This document is a technical report on the ROBUST Query Algebra comprising core and advanced operators as well as the community data model investigated in WP2. The aim of this work package is to explore an algebraic approach for designing and optimizing community-data specific operators required in WP1-WP8 as well as implements these operators using a functional programming model for efficient data-parallel execution. The task leading to the main results presented in this report was T2.1 “Identify an algebra and operators for community analytics”.

This report lays out the identified core and advanced operators on graph structures and textual data contained in community data as well as the provided data sets by the use case partner in WP 7-9 and their data model. It draws the connection to the related work packages requiring the individual operators and mentions aspects of the associated parallelization strategy. It also discusses the parallel processing platforms involved as well as the adapted existing open-source cloud computing library Mahout.

ROBUST Document Template

Table of Contents

1. Introduction

Work package 2 provides the base infrastructure for extracting, loading and transforming the huge amounts of community data that have to be processed for monitoring huge-scale business communities in the context of ROBUST. The main scope is to define an implement an extensible, algebraic model for complex queries over unstructured data, such as textual forum postings, and structured relational data, such as author information and graph data, such as or relationships between authors, postings and author feedbacks on postings.

This report provides a comprehensive description of the ROBUST algebra consisting of a data model, the platform and the community-data specific operators for the ROBUST infrastructure that have been identified in close cooperation with the application partners application partners (WP 7,8) and the community analysis partners in WP5. These concepts will be implemented and presented in the “D2.2 - Research prototype - Beta version of parallel analysis processor”. The robust algebra provides the common set of operators on the ROBUST community and textual data to be used by the analysis algorithms developed in WP5 driving the use-case analytics and monitoring applications developed in WP 7,8. The algebra contains core and advanced operators, where advanced operators potentially re-use some of the core operators identified. The operators that have been identified resemble analysis tasks and algorithms which could not be feasibly executed on single machines and thus need to take advantage of a large scale analytics engine.

The document is structured as follows: Starting from the data structures, provided by the use case partners (IBM, Polecat and SAP), we define a generic data model for the ROBUST infrastructure. This concerns both textual data as well as graph data. Next, we provide details on the platforms that will serve as the basis for the ROBUST cloud infrastructure, namely “Stratosphere PACT/Nephele” and “Apache Hadoop”. Based on these systems, the use case requirements and the community features provided by work package 5 (see “D5.1 - Report on feature selection and merging”), we derive a set of core and advanced operators for community monitoring. These form the framework of the ROBUST large scale analysis environment, which can easily be adapted to include new operators should the need arise during the remainder of the project.

2. Data Model

The main drivers of the ROBUST infrastructure are the requirements of the use cases. In this section we briefly introduce these use cases, introduce the data structures that have been provided by the use case partners and derive a common data model for the ROBUST infrastructure. In general, we can distinguish two main types of data structures that are of importance for community monitoring: textual content and social graphs. Respectively, the following section will lay out in detail each of these data structures and the corresponding monitoring scenarios separately.

2.1. Textual Content

An important aspect of the envisioned community monitoring infrastructure is the navigation over aggregated user generated contents. In particular, besides the social graph (see next subsection), extracted entities (e.g., persons, products, and companies), their co-occurrences and higher level aggregates such as discussion topics are important.

SAP Community Network

In the case of the SAP Community Network (SCN) of the WP 8 Business partner use case the majority of the user generated content is provided in the SCN Forums (see also “D8.1: Provisioning and preparation of the SAP Community Network Data”).

In order to make the SCN Forums data usable by the ROBUST infrastructure it has been necessary to harvest their content and metadata and make it available in a form that could be easily queried in further processed according to the ROBUST needs. In order to do this a crawler has been created able to retrieve all messages and associated data for a given forum channel. Besides metadata such as the title of the thread; the status (Answered or Unanswered) or the number of views per thread, which might be relevant for simpler monitoring purposes, SAP retrieved:

- the title of a thread,
- it’s actual textual content of a message,
- the author of a message,
- the time stamp when the message has been created,
- a set of entities and proper nouns that have been identified in a message by information extraction techniques.

At the time of writing this document approximately 200,000 threads containing 850,000 message posts have been crawled. Out of this, a number of around 80,000 corresponding to 350,000 messages have been provided to the consortium as a small test data set.

Next to extracting structured information from the documents, one major challenge is to identify so called “topics” in the document collection. Under the

term “topic”, we understand a mixture of terms (or entities) that frequently occur together. Thus, a topic describes the common terms (or entities) of a cluster of documents that are sufficiently similar. An interesting analytical scenario in the context of monitoring the SCN is the tracking of topics over time. That is, the community mangers (SCN mentors or forum moderators) can follow the distribution of topics across the community, the involvement of the members in discussions related to these topics and furthermore, can follow the emerging or fading of some of these topics during time. We will further detail algorithms for topic detection in Section 4.2.

IBM Connections

IBM Connections is a social collaboration platform sold to IBM customers and deployed on its intranet for use by its employees. The intranet deployment serves in ROBUST as the data set for the analysis of employee communities inside the enterprise in the context of the employee use case in WP 7.

IBM Connections exposes its data through APIs that can be accessed and crawled. The public entities are crawled and provided to ROBUST. Due to corporate confidentiality and privacy restrictions no content can be provided to the consortium. Still, a wide set of meta-data is exposed which enables building a rich social graph of relationships between the different entities, in particular entities (blogs, wikis, forums, etc.) and people.

Two sets are provided by IBM, the raw data of the services provided by Connections and the graph structure. The raw data includes the following information

- **Employees** with their id, country, organization, manager, friends (explicit network) and tags.
- **Blogs**, with their id, author, comments, commenter, recommenders and date information.
- **Wikis** with their id, tags, child pages, authors, editors, commenters and date information.
- **Files** with their id, author, downloader and sharing information and tags. All includes date information.
- **Discussion Forums** with their id, author, tags and replys with their authors and date information.
- **Communities** with their id, creator, members, included blog entries, forum entries, bookmarks, wikis and feeds.

The first drop of this data provided by IBM contains 534,519 employees profiles, 25,546 communities, 40,265 blog entries, 16,455 Wiki elements, 21,397 shared files, and 10,117 forum threads. More detailed descriptions of the data are available in Robust deliverable 7.1.

Tiddlywiki Community Data

The Tiddlywiki wikigroup community is a software development forum used by a team of developers at British Telecom’s open source development team Osmosoft. The data has been provided by the WP 9 public domain use case.

The original data was taken from a Google Group, and was available to Polecat in the form of an MBOX email format. This was then into a RDBMS model (MySQL) for the purpose for analysis by the Robust community. Interaction with this data is available directly, via an appropriate protocol (JDBC in the case of most partners). Partners have run analysis against the data, and stored the results of this analysis, where required, in the same database. Additional database objects have a naming convention showing the partner owner.

Each of the postings to the community has the following properties:

Receivers

Sender

Post Subject

Post Content

Date Sent

In terms of volume, there are 51,662 postings over a 3 year period, and the community has 3,194 contributing users.

There is also a separate, but related community: the Tiddlywiki development community (for core, rather than plug-in development). The structure of the data is the same as the core community. This has 14,703 messages over a shorter time period, and 791 contributing members.

2.2. Social Graphs

The aim of the SAP's envisioned demonstrator is to augment BI-style analysis of the community (e.g., built on top of log data) with support for graph based exploration and graph mining algorithms. In particular, two types of graphs will be tackled - a simpler one based on the reply structure and a more complex one that includes besides members also resources and content specific aspects as well.

The reply structure graph of the SCN reflects those interactions between the community members that are created when a member responds to a message that has been written by another member. That is, the nodes (i.e. vertices) of this graph are represented by community members while each of the edges indicates that there has been at least one interaction between the two members representing the nodes the edge connects. However, the reply structure graph does not capture at all the context in which the interactions between community members have taken place. Indeed, the user-contributed content and meta-information such as the forum details, characteristics of the thread or of the messages themselves can offer useful information about the conditions in which each interaction has been taken place.

As such, we propose a richer graph, with different types of nodes that capture meta-data, network structural information as well as the features of the actual

content (i.e. posts and message content). There is a range of analyses and algorithms that can be run on such an extended graph, in order to learn more about individual members or about the community itself as a whole. The nodes of the content-augmented graph can have the types exemplified in Figure 1. It consists of several types of nodes, where some of them can be interpreted as edge if supplied to a particular algorithm.

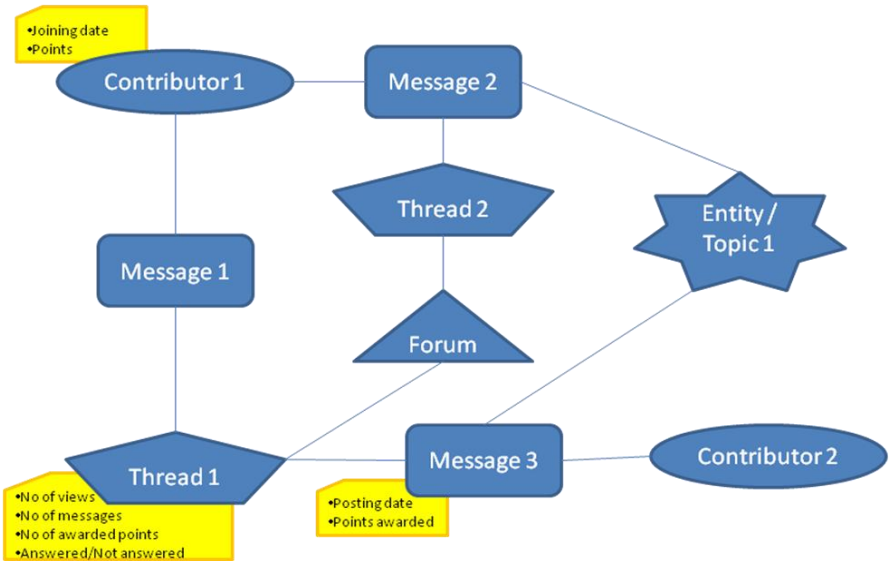

Figure 1: Graph data structure for the SAP Community Network

The most important nodes are users. On relation among them is that they exchange knowledge by means of contributing to the same forum thread messages. Another relation can be drawn by interests, e.g., if they “talk” about the same entities or topics (see Section 2.1.1.).

The edges of this graph will have no weights, except maybe those that connect a message with a certain topic or entity. That is, as entities and topics are extracted with a certain measure of confidence by automatic Information Extraction processes, this can be reflected in the above graph by associating weights to the edges connecting a Message type node and an Entity/Topic node. This graph consists of approximately 100,000 edges.

IBM Connections

In addition to the raw data described in section 2.1, IBM provides a social graph structure to the consortium. The nodes of the graph are all IBM employees and the vertices a wide set of relationships that can be extracted out of the entities described in section 2.1. These relationships include for example co-editing of wikis, being friends in the social network, tagging each other, commenting on each other’s blogs etc. Each relationship comes with a count of how often it happened between each pair of employees. More details can be found in Robust deliverable D7.1.

We extracted the graph of all available interactions that consists of 500,000 vertices connected by 107 million edges.

3. Parallel Analysis Processing Platforms

Extracting, loading and transforming millions of daily transactions of community data is a challenge. To enable the timely execution of community analysis and monitoring algorithms on top of these massive datasets one has to exploit parallelism based on a multitude of independent compute nodes on top of distributed file systems suitable for handling big data and large clusters of nodes. Section 3.1 introduces the most popular MapReduce paradigm pioneered by Google and its open source implementation Apache Hadoop [Had]. While the basic operators Map, Reduce and Combine of Hadoop enable the parallelization of quite a variety of analysis operations, it also has significant shortcomings that prohibit the truly efficient parallelization of particular algorithms. To overcome these shortcomings TU Berlin implemented the Stratosphere PACT/Nephele System, which is presented in section 3.2. The parallelization contracts of the Stratosphere system are an extension of the MapReduce paradigm that generally allows a more efficient implementation of analysis algorithms. These systems form the basis of the ROBUST scalable, parallel execution platform for core and advanced community analysis operators.

For the implementation of the operators we chose to adopt and enhance the existing open-source cloud computing analysis library Apache Mahout wich is introduced in section 3.3. Mahout allows for the scalable execution of complex analytical queries.

3.1. Apache Hadoop

MapReduce[DG04] has become a popular paradigm for data-intensive parallel processing on shared-nothing clusters. It is inspired by functional programming and has originally been developed to help Google create its index of the web.

Hadoop[Had] is a widely used open source implementation of this paradigm. The data is stored blockwise across machines of a cluster in a distributed file system and is usually represented as semi-structured (key,value) tuples. In order to efficiently parallelize the computation and offer tolerance against machine failures, data is replicated across multiple machines in a cluster. The computation needs to be happen where the data resides following the paradigm “ship code not data”. The runtime system has to be smart enough to assign processing tasks for data blocks to the machines already holding the replicas of these blocks. The computation code is embedded into two functions:

```
map: (key1, value1) -> list(key2, value2)
reduce: (key2, list(value2)) -> list(key3, value3)
```


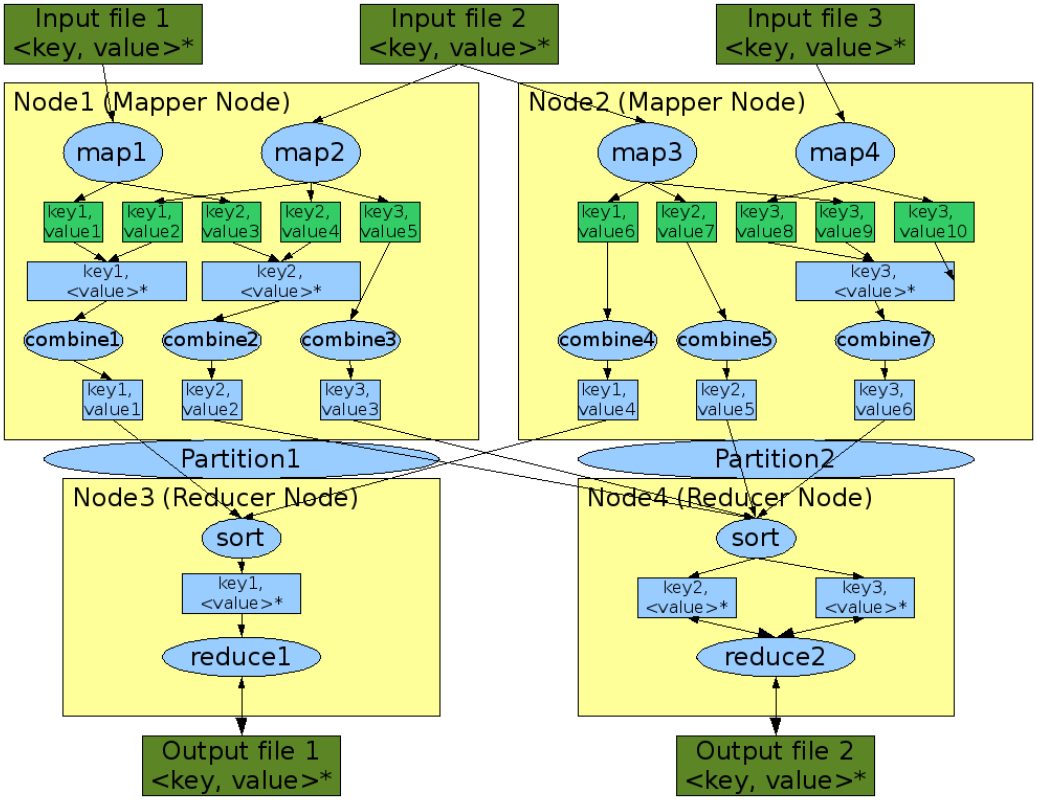

Figure 2 Data Flow inside a Map-Reduce system

At the beginning the map function is invoked on the input data in parallel on all the participating machines in the cluster. The output tuples are partitioned and sorted by their key and then sent to the reducer machines in the shuffle stage. The receiving machines merge the tuples and invoke the reduce function on all tuples sharing the same key. The output of this function is written to the distributed file system.

A third optional function called combine can be specified that. It is invoked locally after the map phase and is used to pre-aggregate the tuples in order to minimize the amount of data that has to be sent over the network (which is usually the most scarce resource in a distributed environment), therefore acting as a kind of local reducer.

```
combine: (key2, list(value2)) -> list(key2, value2)
```

Finally each map and reduce task offers initialize and tear-down functions that are invoked at the beginning and end of their phase which allow optimizations like in-mapper combining of small amounts of data.

3.2. Stratosphere PACT/Nephele

Large-scale data analysis applications require processing and analyzing of Terabytes or even Petabytes of data, particularly in the areas of web analysis or scientific data management. This trend has been discussed as web-scale data management in a panel at VLDB 2009 [ACH+09]. Formerly, parallel data

processing was the domain of parallel database systems. Today, novel requirements like scaling out to thousands of machines, improved fault-tolerance, and schema free processing have made a case for new approaches.

Among these approaches, the map/reduce programming model and its open-source implementation Hadoop have gained the most attention. Developed for simple logfile analysis, map/reduce systems execute sequential user code in a parallel and fault-tolerant manner, once it has been written to fit the second-order functions map and reduce. However, with the success of map/reduce, many projects have started to push more complex (e.g., SQL-like) operations into the programming model, thereby violating some of its initial design goals (i.e., separation of parallelization and user code) and paying significant performance penalties.

To eliminate these shortcomings TU Berlin has developed the Nephele/PACTs system[BEH+10], a parallel data processor centered around a programming model of so-called Parallelization Contracts (PACTs) and the scalable parallel execution engine Nephele. Our system pursues the same design goals as map/reduce and is highlighted by three properties:

1. A richer programming model than map/reduce that preserves the same abstraction level: Our programming model is based on Input and Output Contracts. Input Contracts are second-order functions which allow developers to express complex data analytical operations naturally and parallelize them independent of the user code. Output Contracts annotate properties of first-order functions and enable certain optimizations.

2. A separation of the programming model from the concrete execution strategy: The PACT programming model exhibits a declarative character. Data processing tasks implemented as PACT programs can be executed in several ways. A compiler determines the most efficient execution plan for a PACT program and translates it into a parallel data flow program.

3. The flexible execution engine Nephele: Nephele executes data flow programs modeled as directed acyclic graphs (DAGs) in a parallel and fault-tolerant way. Job annotations enable the PACT compiler to influence the execution in a very fine-grained manner.
The PACT programming model is a generalization of the map/reduce paradigm implemented for example in Apache Hadoop introduced in section 3.1 of this report. It is based on a key/value data model and the concept of Parallelization Contracts (PACTs). A PACT consists of exactly one second-order function which is called Input Contract and an optional Output Contract. An Input Contract takes a first-order function with task-specific user code and one or more data sets as input parameters. The Input Contract invokes its associated first-order function with independent subsets of its input data in a data-parallel fashion. In this context, the well-known functions map and reduce are examples of Input Contracts. The first-order function has to implement an interface that is specific to the PACT it uses, as it is known from map/reduce.

The programmer can attach optional Output Contracts to PACTs to denote certain properties of the user code's output data, which are relevant to the parallelization. The compiler can exploit that information and deduce in some cases, that suitable partitionings or orders exist and reuse them. Data processing tasks are implemented by providing custom code to PACTs and assembling them to a work flow graph. In the following we list an initial set of Input Contracts and describe them shortly:

Map The Map contract is used to process each key/value pair independently. Every key/value pair becomes an independent subset consisting solely of itself.

Reduce The Reduce contract partitions key/value pairs by their keys. Every group becomes an independent subset Similar to map/reduce we allow the use of a combiner.

Cross The Cross contract operates on multiple inputs and builds a distributed Cartesian product over its input sets. Each element of the Cartesian product becomes an independent subset.

CoGroup The CoGroup contract partitions each of its multiple inputs along the key. Independent subsets are built by combining equal keys of all inputs. Hence, the key/value pairs of all inputs with the same key are assigned to the same subset.

Match The Match contract operates on multiple inputs. It matches key/value pairs from all input data sets with the same key. All key/value pairs within an independent subset have the same key, but in contrast to CoGroup each subset contains only one key/value pair from each input. Hence, the Match contract associates key/value pairs from its inputs like an inner join on the key, without actually joining them.

While Input Contracts are mandatory components of PACTs, Output Contracts are optional. They allow developers to guarantee certain behaviors of the user code with respect to the properties of the output data. As an example, the developer can attach the Same-Key Output Contract to a function that returns the same key as it was invoked with. The PACT compiler exploits these guarantees provided by the output contract to generate more efficient data flow programs. With the mentioned Same-Key Output Contract, the compiler can for example deduce that any key-partitioning on the input data of the function still exists on the output data. The compiler can hence avoid unnecessary repartitioning and therefore expensive data shipping.

The Nephele/PACT system has a three tier architecture. To execute a PACT program it is submitted to the PACT Compiler. The compiler translates the program into a data flow program and hands it to the Nephele system for parallel execution. Input/output data is stored in the distributed File system

HDFS. We will briefly introduce the Nephele system and the PACT compiler in the following.

The Nephele system executes the compiled PACT programs in a parallel fashion. Similar to systems like Dryad, Nephele considers incoming jobs to be DAGs with vertices being subtasks and edges representing communication channels between these subtasks. Each subtask is a sequential program, which reads data from its input channels and writes to its output channels. The initial DAG representation does not reflect parallel execution. Prior execution, Nephele generates the parallel data flow graph by spanning the received DAG.

Thereby, vertices are multiplied to the desired degree of parallelism. Connection patterns that are attached to channels define how the multiplied vertices are rewired after spanning. During execution Nephele takes care of resource scheduling, task distribution, communication as well as synchronization issues. Moreover, Nephele's fault-tolerance mechanisms help to mitigate the impact of hardware outages. Unlike existing systems, Nephele offers to annotate jobs with a rich set of parameters, which inuence the physical execution. For example, it is possible to set the desired degree of data parallelism for each subtask, assign particular sets of subtasks to particular sets of compute nodes or explicitly specify the type of communication channels between subtasks. With respect to the PACT layer we leverage these parameters to translate optimization strategies of the PACT compiler into scheduling hints for the execution engine.
Currently, Nephele supports three different types of communication channels: Network, in-memory, and file channels. While network and in-memory channels allow the PACT compiler to construct low-latency execution pipelines in which one task can immediately consume the output of another, file channels collect the entire output of a task in a temporary file before passing its content on to the next task. As a result, file channels can be considered check points, which help to recover from execution failures.

The PACT compiler translates a PACT program into a Nephele DAG. In contrast to map/reduce, our system separates programming model and execution strategy. Due to the declarative character of the PACT programming model, the PACT compiler can choose from several execution plans with varying costs for a single PACT program. In the following, we discuss some optimization opportunities and describe the generation of Nephele DAGs.

Optimizing a single PACT. PACTs exhibit a declarative character. They define which independent subsets are generated from the input data and provided to separate instances of the user function, but do not define how that is actually achieved. A single Input Contract can be fulfilled by multiple execution strategies. Among the strategies we consider, several were devised by research on parallel database systems such as repartitioning, broadcasting, and symmetric-fragmentation-and-replication. As an example, the Match contract can be satisfied using either a repartition strategy which partitions all inputs by keys or a broadcast strategy that fully replicates one

input to every partition of the other input. Choosing the right strategy can tremendously reduce network traffic and execution time.

Optimizing a PACT program. Similar as in query optimization for relational DBMS, the optimal execution strategy for a PACT program cannot be found by combining the locally optimal choices for all PACTs. The evaluation of a PACT becomes significantly cheaper when existing properties of the data such as partitionings or sort orders can be exploited. Therefore, our PACT compiler works similar as a Selinger-style SQL optimizer which tracks so-called interesting properties. During optimization more expensive plans are spared from pruning if they provide an interesting property which can be utilized later. The PACT compiler exploits information provided by the Output Contracts to infer that the user code preserves certain properties. Our compiler's cost model considers data shipping costs. Starting with file size information, we use user annotations, such as factors for changing data volume and key cardinalities, to derive reasonable estimations. If reasonable estimates are impossible due to missing annotations, the compiler picks the strategy that performs best on large input sizes.

After choosing the execution plan for a PACT program, the PACT compiler must transform it into a Nephele DAG, which consists of sequential code blocks (vertices) and communication channels (edges). The compiler wraps the user function of each PACT with PACT code and maps it to a vertex in the Nephele DAG. The wrapping PACT code invokes the user code with an independent subset of data according to its Input Contract and receives its output. The execution strategy for a PACT is reflected in three aspects within the Nephele DAG: First, in the wiring pattern between the subtasks. Second, in the PACT code that calls the user code, and finally in the PACT code which receives and forwards the data in the preceding vertex.

3.3. Apache Mahout

The Apache Mahout™[Mah] machine learning library's goal is to build scalable machine learning libraries. It is an open source project maintained by the Apache Software Foundation.

Scalability refers to a variety of cases here: Scalable to reasonably large data sets. The core algorithms for clustering, classification and Collaborative filtering are implemented on top of Apache Hadoop using the map/reduce paradigm. The core libraries are highly optimized to allow for good performance also for non-distributed algorithms. Furthermore Mahout is scalable to support business cases as it is distributed under the commercially friendly Apache Software license. The last goal is a scalable community. Currently Mahout supports mainly four use cases: Recommendation mining takes users' behavior and from that tries to find items users might like. Clustering takes e.g. text documents and groups them into groups of topically related documents. Classification learns from existing categorized documents what documents of a specific category look like and is able to assign

unlabelled documents to the correct category. Frequent itemset mining takes a set of item groups and identifies which individual items usually appear together.

We chose Mahout as a basis for some of our algorithms that are tightly coupled to already existing functionality (liked distributed Linear Algebra operations). Therefore we use Mahout as an existing open-source cloud computing solution on top of Hadoop side-by-side with the Stratosphere system and decide for each particular research use case which system to base our solution on.

4. ROBUST Algebra Operators

In this section we give a concise presentation of the identified operators of the ROBUST algebra.

4.1. Core Operators

The core operators are going to be used as basis for the more advanced operators described in the following sections. In the case of the SAP SCN scenarios for instance, the fact prediction and fact retrieval algorithms can be also used in populating the graph described in Section 2.2. That is, some of the entities included the content-augmented graph could be in fact complex entities, represented by the occurrence (in a certain context) of two or more other elementary entity (or terms).

FactPrediction

Identifying and extracting relevant information such as entities and semantic relations amongst these entities from the textual community data is an essential processing step in analysing and monitoring communities such as SAP Community Network (SCN). Current relation extraction systems utilize a computational expensive execution stack of natural language processing operators such as part-of-speech tagging, phrase detection, named entity recognition and co-reference resolution. This machinery is needed to detect the exact borders of attribute values, to identify their correct type and to identify their relation within and across sentences. As a result, processing a document may take up to a few seconds. Higher precision is ensured with deep dependency parsing or semantic role analysis. These approaches require processing times of tens of seconds. Recently, extraction system vendors also offer extraction-as-a-service. These systems enable non-natural-language-processing experts to extract relations from community postings. This abstraction comes at the price that users can no longer tune the patterns or rules of the extractor. Moreover, additional costs may appear, such as costs for sending a page through the network or monetary costs.

It would thus be highly desirable to be able to filter out irrelevant postings that do not contain relations to speed up the extraction process and only forward relevant pages to the relation extractor.

We identified the **FactPrediction(Document, RelationType)** operator [BHL11], which predicts whether an extraction service will be able to extract a relation from a document. The relation predictor only forwards relevant pages which contain a textual representation of a relation to the actual extraction service. The relation predictor abstracts from extractor specific words and syntactic structures and is generally applicable to multiple extraction services. It avoids computationally intensive natural language processing techniques, such as part-of-speech tagging or deep dependency tree parsing.

The Input of the FactPrediction Operator is a textual document and type of a semantic relation. The output is the judgement whether the document contains a semantic relation of a given type

FactRetrieval

Collections of text documents such as forum postings or other community text data are often indexed with inverse indices. When these structures exist, one can leverage this information in identifying and retrieving documents that are likely to contain a semantic relation of a certain type that can be extracted by an information extraction service [BLN11]. In inverted indices, Words **w** out of a vocabulary **W** are mapped to documents **d** of a document collection **D** where documents are usually treated as bags of words. A keyword query **w** executed against an index **I** will thus return a set of documents which contain the keyword: $D_w = \{d \in D \mid w \in d\}$. Our goal is to leverage such an index to retrieve the set of facts

$$F = \bigcup_{d \in D} extract(d)$$

that can be extracted from the document collection while only processing a minimal subset of relevant documents $D_{min} \subseteq D$ containing all facts in **F** such that

$$F = \bigcup_{d \in D_{min}} extract(d) = \bigcup_{d \in D} extract(d)$$

When an inverted index structure returns a document for a given keyword, all that is known with certainty is that the keyword occurs at least once in this document. The challenge is to use this limited information as evidence about the otherwise unknown document collection and to somehow infer which documents are likely to contain a fact – to enable the system to efficiently extract all the facts(relations) **F** of a particular type contained in a document collection. To be able to solve this problem, one needs to estimate the probability that a document **d** actually contains a fact and process the most promising documents first. To infer this probability, we probe the index with a small set of extracted keywords and use the results as evidence to estimate the relevance of individual documents. This approach only requires a small set of handcrafted initial seed keywords W_{seed} as input to retrieve all facts from a document collection indexed with an inverted index **I**. In our notation, a fact is an instance of a semantic relation between two or more entities (e.g. *CompanyProduct(Company, Product)*; *PersonCareer(Person, Career)* …). The type of a fact is thus the type of semantic relation that a fact expresses.

Let **Y** be a random variable that encodes whether a given document actually contains a fact and **X** be another random variable from a joint distribution **P(X, Y)** encoding with which of the evaluated keywords a document has been found (thus encoding the occurrences of keywords in this document). Let us assume that the distribution is the same across the document collection. Our fact retrieval problem can then actually be expressed as inferring the (conditional) probability that a given document **d** contains a fact, given the

keyword queries with which it has been found. This means we have to estimate the conditional probability of **Y** (whether a document contains a fact) given the keywords **X** which returned the document **P(Y|X)**. By applying Bayes rule, the conditional probability can be rewritten as:

$$P(Y|X) = \frac{P(X|Y)\ P(Y)}{P(X)} \propto P(X|Y)P(Y)$$

By assuming that the keyword queries **X** with which the document has been found are conditionally independent given the label **y**, one can easily take the maximum likelihood estimates of the conditional probability P(X|Y) and the prior P(Y) from the evaluated keyword queries and then use these estimates to compute the probability that a document contains a fact for all unprocessed documents.

We see the sampled documents as a set of training examples (x,y) which have been generated from the joint distribution P(X,Y). Estimating the conditional probability is then similar to learning the decision function of a Naive Bayes classifier.

The input of the FactRetrival Operator is an inverted index over a document collection and a type of a semantic relation. The output of the operator is a list of documents that are likely to contain a relation of the specified type.

Any-K Operator

While the FactPrediction and FactRetrieval Operators can be applied to efficiently identify binary relations in community text data, one might also want to efficiently retrieve more complex relations from such textual data sets. To accomplish this task we identified the any-k operator [LNP+11], which enables the efficient processing of more complex relations – usually consisting of individual binary relations which are joined together based on common named entities such as persons or companies. Take the following query for instance

Product(Company,Product) ⋈ Career(Person,Company, Position) ⋈ Attribute(Person,B_Place, B_Date);

Where the three basic relations *Product(Company,Product), PersonCareer (Person,Company, Position)* and *PersonAttribute(Person,B_Place, B_Date)* provided by an information extraction service are joined together based on the underlined attributes. The any-k operator enables the efficient processing of such conjunctive queries over text data.

Graph Simplification

This operator removes loops (edges that connect a vertex to itself), duplicate edges and edge directions (optionally) from the input graph. It is a common preprocessing step for a variety of graph algorithms (such as triangle enumeration) for community analysis, that need to work on a cleansed version of the data. It uses operations like PROJECT, AGGREGATE and SELECT in this cleansing process.

The input to this operator is a set of edges where each edge is denoted by the numerical identifier of its start and end vertex.

The algorithm itself is rather non-complex; each edge of the input is processed in parallel in a mapper that can already remove loops. The edges will be sent to a reducer that will drop duplicates and write back the edges.

This algorithm requires a single pass over the data only and therefore scales linearly with the size of the input data and the number of machines in the cluster.

Its output consists of a set of edges representing a cleansed version of the input graph.

DegreeDistribution

This operator's aim is to count the degree of each vertex (the number of in- or outgoing edges) and aggregate and sort them to quantiles. The resulting distribution forms crucial information about a graph and can be used to estimate applicability and runtime of further graph algorithms. It is a common preprocessing step for before using more enhanced forms of community analytical reporting. The degrees and their distribution play an important role in WP7 where the in- and outdegree of users in a collaborative community network is used as a feature for predicting future activity.

The input to this operator is a set of edges where each edge is denoted by the numerical identifier of its start and end vertex.

In a first step each edge is examined and its start vertex identifier is sent to a reducer that will thereby count the degree per vertex. In a second pass over the data the degree counts are summarized and converted to quantiles.

The algorithm needs one pass over the input data and a second pass over the degree information that has the same cardinality as the input. It therefore scales linearly with the size of the input data and the number of machines in the cluster.

The output consists of degree numbers and their relative frequency (in descending order)

Adjacency Matrix

The adjacency matrix is a model of a graph based elements of community data in Linear Algebra. It allows many operations on the graph to be expressed in terms of mathematical operations. This model is the basis for various graph analysis algorithms like RandomWalks or Spectral Clustering. It is a square matrix with as many rows and columns as the graph has vertices. A cell m,n of the matrix contains a 1 exactly when vertex m in the graph has an edge pointing to vertex n and 0 otherwise. An additional requirement might be to substochastify this matrix so that it models markov chains.

The input to this operator is a set of edges where each edge is denoted by the numerical identifier of its start and end vertex. Additionally a list of all vertices needs to be supplied because there might be unconnected nodes in the graph.

In a first step the graph is transformed into a matrix by reading each edge in a mapper and keying it by start vertex. In this way the columns of the adjacency matrix are created in the reducer. This matrix is transposed parallely in a second pass over the data.

This algorithm requires two passes over the data and scales linearly with the size of the input data and the number of machines in the cluster.

Triangle Enumeration

Identifying densely-connected subgraphs or trusses [Coh08] within a large graph is a common task in many use-cases such as social network analysis. A typical preprocessing step is to enumerate all triangles (3-cliques) in the graph. For simplicity, we consider only undirected graphs, although the algorithm can be extended to handle more complex graph structures (like multigraphs) with the help of a simplifying preprocessing step. The algorithm requires a total order over the vertices to be defined, for example a lexicographical ordering of the vertex IDs. Triangles form an important information about a graph and give hints about communities, a piece of information of crucial knowledge for WP 5.

The graph is stored in the distributed file system as a sequence of edges (pairs of vertices). When generating the key/value pairs from the file, each edge will be its own pair. Inside the pair, both the key and value will consist of both the edge's vertices, ordered by the defined ordering.

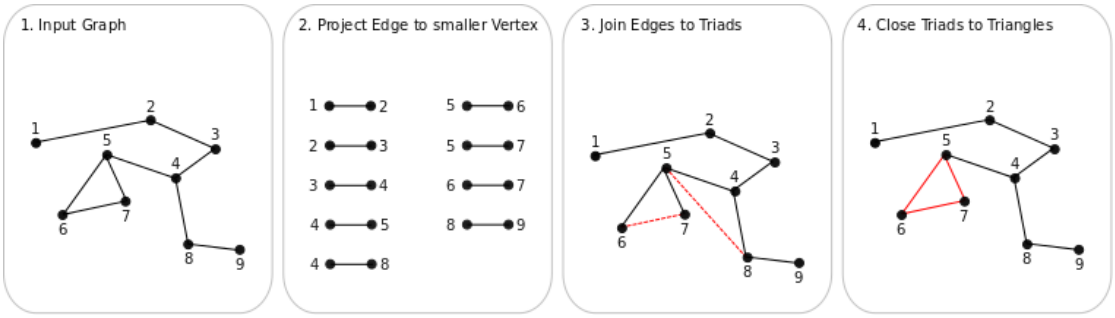

Figure 3 Indiviual Steps of the Triangle Enumeration Operator

The MapReduce approach to solve the edge-triangle enumeration problem was proposed by Cohen [Coh09]. It requires the graph to be represented as a list of edges, augmented with the degrees of the vertices they connect. The implementation comprises two successive MapReduce jobs that enumerate and process the so-called open triads (pairs of connected edges) of the graph. The first Map task sets for each edge the key to its lower degree vertex. The subsequent reducer works on groups of edges sharing a common lower-degree vertex and outputs each possible subset consisting of two edges, using the vertex pair defined by the ordering of the two corresponding higher-degree vertices as the key. The mapper of the second job takes two inputs – the original augmented edge list and the open triads from the preceding reducer. It sets the edge's vertices (in order) as the key for the original edges and leaves the open triads unchanged. The technique of adding a lineage tag

to each record is used, allowing the second reducer to separate its inputs again into sets of open triads and edges. Hence, it works on groups consisting of zero or more open triads and at most one single edge which completes the triads forming a closed 3-cycle.

The edge-triangle enumeration algorithm can be expressed as a PACT program. The first MapReduce job, enumerating all open triads, can be reused without any further change. The second MapReduce job is replaced by a Match contract with two inputs – the Reduce contract's result and the original edge list. Open triads and closing edges are matched by their key. The Match function closes the triad and outputs the three edges as a closed triangle.

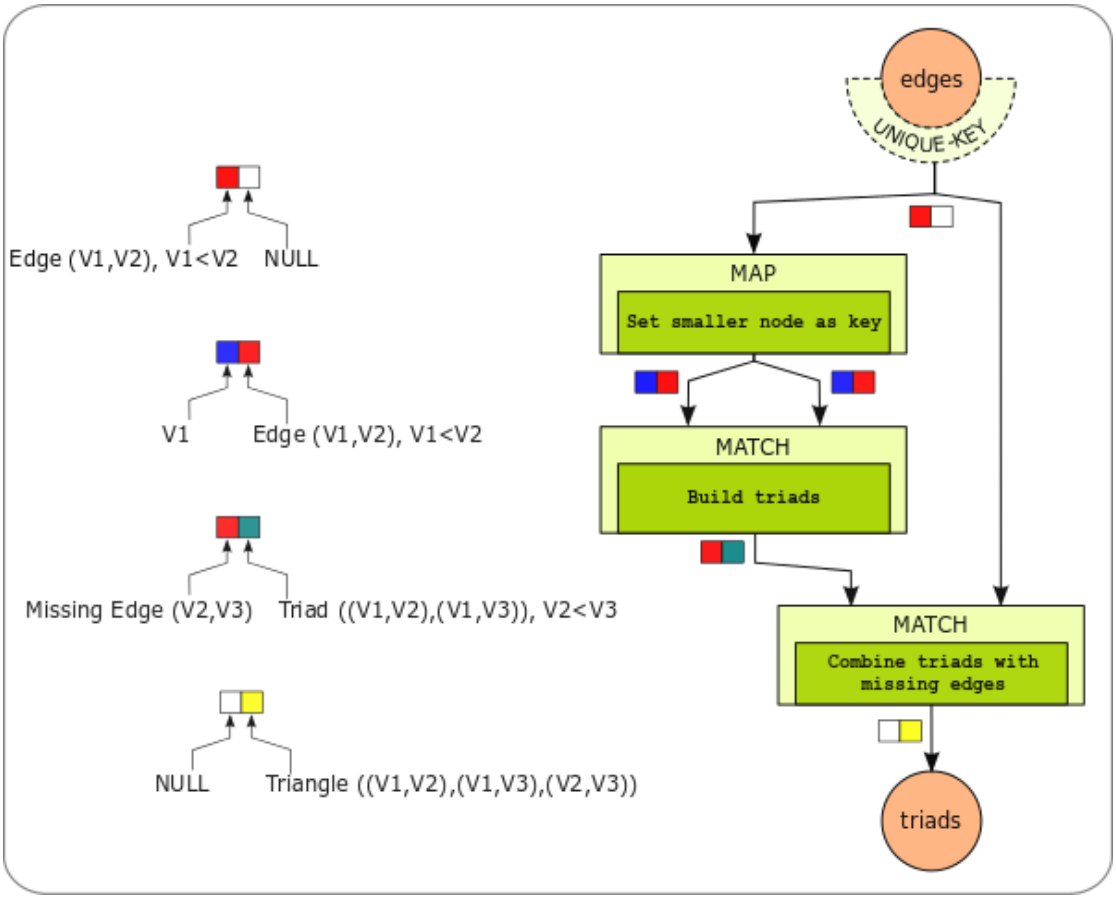

Figure 4 Data Flow of the Triangle Enumeration Operator in the PACT/Stratosphere system

The runtime of the job is dominated by the building of open triads and their join with the edges. The number of open triads as quadratic to the degree of a vertex, handing the enumeration down to the lower order vertex does not change the asymptotics of this. The runtime is clearly dependent on the shape of the graph, on its degree distribution to be precise. It will be a task for later research to see how well this algorithm can be applied to the graphs provided by the use case partners.

The output of the job is a list of the triangles of the graph denoted by triples of connected vertex identifiers.

4.2. Advanced Operators

The advance operators build on top for the core operators and offer functionality that will get reflected in the selected use case scenarios. For example, in the case of the SAP Developer Network (SDN), the LDA Topic Model and the Bayesian Topic classifier will contribute to the population of the SCN graph. The Author Authority (Page Rank) algorithm can be used to assess the relevance of certain message post, to identify the most popular forums and to bring insights into the overall content quality, as perceived the community members themselves. The Node Proximity can be used to understand the existing connections and influence patterns that exist among community members that go beyond the simple reply structure of the network. The Community Detection and Link Prediction algorithms could be used detecting clusters of users acting independently in the same forum as well as the connections existing between various topics and forums.

The advanced operators combine and reuse Core Operators to allow more refined analytical reporting on the community data. We plan to implement them in the context of the existing open-source cloud-computing library Mahout.

Latent Dirichlet Allocation (LDA) Topic Model

The Latent Dirichlet Allocation (LDA) operator infers a probabilistic topic model over a given collection of textual documents such as forum postings. It is used as a way to discover the hidden thematic structure in such documents collections, which is a task requested for example by the use case in WP 7 and 8. A topic in this context is basically a distribution over words (terms) where the most likely terms characterize the topic. Each document is seen as a mixture of these individual topics or distribution of words. This information can be used to cluster documents according to different topics or can be seen as features that annotate a social graph. LDA is a hierarchical Bayesian model and is usually inferred using either variational inference or sampling approaches.

The parallelization strategy for this operator is as follows. When using variational inference, which can be seen as a generalization of the expectation maximization algorithm for hierarchical Bayesian models, one has to infer the posterior probability of each topic for each word in each document by counting the occurrences and afterwards normalize the counts to arrive at an actual distribution over all terms in the collection. The first step can be implemented within a Map operator and the normalization can be executed as a reduce operator. These steps have to be repeated until convergence.

The input of this operator is a collection of text documents. The output is distributions over words (topics) as well as a distribution over topics for each document in the collection.

Naïve Bayes Topic Classifier

This text operator classifies textual documents such as forum postings into predetermined categories which can be topics, forum branches etc. The documents are treated as bags of words and represented through term frequency – inverse document frequency (*tf-idf*) feature vectors. The operator is trained on a set of labelled training data as a pre-processing step and can then classify documents on the fly. A Naïve Bayes classifier is a simple generative model for classification, which assumes conditional independence of the features X given the label. To be able to scale the feature extraction and training of the model to large collection of community data, this phase of the operator needs to be parallelized. The actual classification itself can most likely be accomplished without a massively parallel execution environment.

The training part of the operator requires a set of labelled training data, which means textual documents and an associated category. The trained operator can then be invoked with individual documents as input and returns a category label for this document.

Author Authority (PageRank)

PageRank is a function that assigns a real number to each node in a social graph (nodes might represent users for example). The intent is that the higher the pagerank of a node, the more “important” it is. It provides an intuitive way of ranking authors.

Think of a community of users as a social graph, where users are the nodes, and there is an edge between users if some interactions between them have happened, like user A read a forum post of user B or they form some kind of connection like a friendship. Suppose we randomly follow these connections with a probability defined by the number of outgoing edges per node (the connections of a user). The probability distribution for the location of a random reader can be described by a vector whose j^{th} component is the probability that the reader is reading posts of user j. This probability is the (idealized) pagerank function. In analyzing user communities as conducted in WP5, it’s of crucial importance to identify these influential users.

The input to this operator is a set of edges where each edge is denoted by the numerical identifier of its start and end vertex. Additionally a list of all vertices needs to be supplied because there might be unconnected nodes in the graph.

As there might be unconnected nodes in the graph or subgraphs with no out-connection we also need to introduce a teleportation probability 1- β making our random reader to jump to the content of any other user in the social graph. One iteration of the PageRank algorithm involves taking an estimated Page-Rank vector v and computing the next estimate v′ by

$$v' = \beta M v + (1 - \beta)\, e/n$$

This linear equation can be solved by iterative multiplication (power iterations). The matrix M is the matrix created by our core Adjacency Matrix operator. One iteration boils down to multiplying this matrix by the PageRank vector v which is done by broadcasting it through the cluster and reading M row wise in a mapper after that. The dot product of a row n with the PageRank vector forms the n-th component of v' and will be sent to a reducer that simply collects it. This is process is repeated until a fixed number of iterations has passed or a convergence criterion has been met.

The operation requires a single pass over the data per iteration its runtime is proportional to the number of iterations.

The output is a list of vertex ids with their particular PageRank attached.

Node Proximity (Random Walk with Restarts)

This operator is very closely related to the PageRank operator. It also performs a random walk through the graph but that walk is started from a particular start node and with a given probability we teleport back to that start node. So we don't get a measure for node importance (as in PageRank) but a measure for the proximity of every node in the graph to our start node. If our nodes represent users and edges represent interactions than this proximity can be used to form new friendships for example.

The input to this operator is a set of edges where each edge is denoted by the numerical identifier of its start and end vertex. Additionally a list of all vertices needs to be supplied because there might be unconnected nodes in the graph.

The parallelization strategy is very similar to that of PageRank. Again we solve the same linear system by power iterations we just use another teleportation probability vector. We need to use the AdjacencyMatrix operator first to create the matrix representation of our input graph. After that we iteratively perform matrix vector multiplication by broadcasting the vector and multiplying it with the rows of the matrix in the mapper. The reducers will collect the results and form the new version of the vector with the steady state probabilities of the random walk.

The operation requires a single pass over the data per iteration its runtime is proportional to the number of iterations.

The output is a list of vertex ids with their particular proximity attached.

Community Detection (k-trusses)

This operator aims to find densely connected subgraphs in social graphs. These subgraphs denote communities where a large fraction of the members interacts with each other. The theoretical construct to search for would be a maximum clique, a subgraph where each node is connected to each other node. However it has been proven that finding maximum cliques is NP-hard

and they don't seem to occur that often in real world networks. Therefore this operator aims for so called k-trusses, a relaxation of a clique. A k-truss is a subgraph in which each edge is contained in at least k-2 triangles.

The input to this operator is a set of edges where each edge is denoted by the numerical identifier of its start and end vertex.

This operator relies on the triangle enumeration operator. It will iteratively invoke the triangle enumeration and inspect each edge in a second pass over the data. If the edge is not contained in at least k-2 triangles it will be dropped. The algorithms terminates when no more edges are dropped, the remaining edges are the ones forming the trusses.

This operation is very costly and data intensive as it needs to iteratively invoke the triangle enumeration. The number of iterations depends on the k chosen. It will be an aim for future research to find strategies for choosing a near optimal k upfront.

The output of this operator is a set of edges denoted by start and end vertex id that are contained in a k-truss.

Link Prediction (Co-Occurrence)

This operators' aim is to mine similarities in social graphs by looking at co-occurring events. These similarities depend on the content of the graph and could denote content recommendations like "people who read this thread also read ..." or the prediction of new links like "people who are friends with this user are also friends with...". It will extract those co-occurring events per node from the graph and use a similarity measure like cosine or jaccard coefficient to compare their vectorized representations. Highly scoring nodes form the recommendations afterwards.

The input to this operator is a set of edges where each edge is denoted by the numerical identifier of its start and end vertex.

This operator uses the AdjacencyMatrix operator to create a matrix representation of the input graph. A row in this matrix represents a node and all its edges to other nodes in the graph. By pairwise comparison of all nodes that share at least one neighbor we can be sure to mine all non-zero similarities in the end. In a first pass over the data, a couple of statistics such as the number of edges per node are computed and written back to the distributed file system. In a second run the mapper emits all co-occurring nodes from an edges neighborhood by reading a row from the adjacency matrix. These pairs are collected and counted by the reducer afterwards which uses the counts together with the previously saved statistics to compute the final similarity value.

The runtime of this operator depends on the degree distribution of the input graph. The degree distribution operator can be used to examine this distribution in depth. Nodes with a very high degree might need downsampling as the number of co-occurring pairs that would need to be emitted from them

grows quadratically. If such down-sampling is applied, the algorithm scales linearly with the number of nodes and edges in the graph.

The outputs of this operator are pairs of nodes denoted by their ids together with their assigned similarity value.

Local Clustering Coefficient

The local clustering coefficient of a node in a graph is a measure indicating how close its neighbours are to being a complete graph (a so called clique). It therefore denotes the connectedness of a graph. It is computed by dividing the number of links between the nodes in its direct neighbourhood through the maxium number of links that could possibly exist between them.

The input to this operator is a set of edges where each edge is denoted by the numerical identifier of its start and end vertex.

This operator relies on the triangle enumeration operator. For each vertex we need to perform a join between its edges and the triangles containing one of its neighbors. In MapReduce the reducer would then need to count neighbours and containing triangles, in PACT this code could be embedded inside a Match.

The runtime and data intensity of this algorithm depends on the shape of the graph, to be more precise on the number of triangles that need to be joined.

The output is a list of vertex ids with their particular local clustering coefficient attached.

4.3. Combination of Operators resembling the Algebra

The algebra is formed by the interdependencies between the operators. Core operators provide means for data transformation and aggregation that forms the input for Advanced Operators.

There are basically two families of graph operators in our algebra. The first one correlates to the way graph algorithms are usually thought of in computer science: as operations on vertices and edges. Commonly the first operator invoked for all algorithms is the simplification operator that creates the graph representation as a set of vertices and edges from the raw input data.

Another very important operator is the triangle operator for mining 3-cliques of a graph that form the input for more advanced algorithms such as k-trusses or clustering coefficient computation.

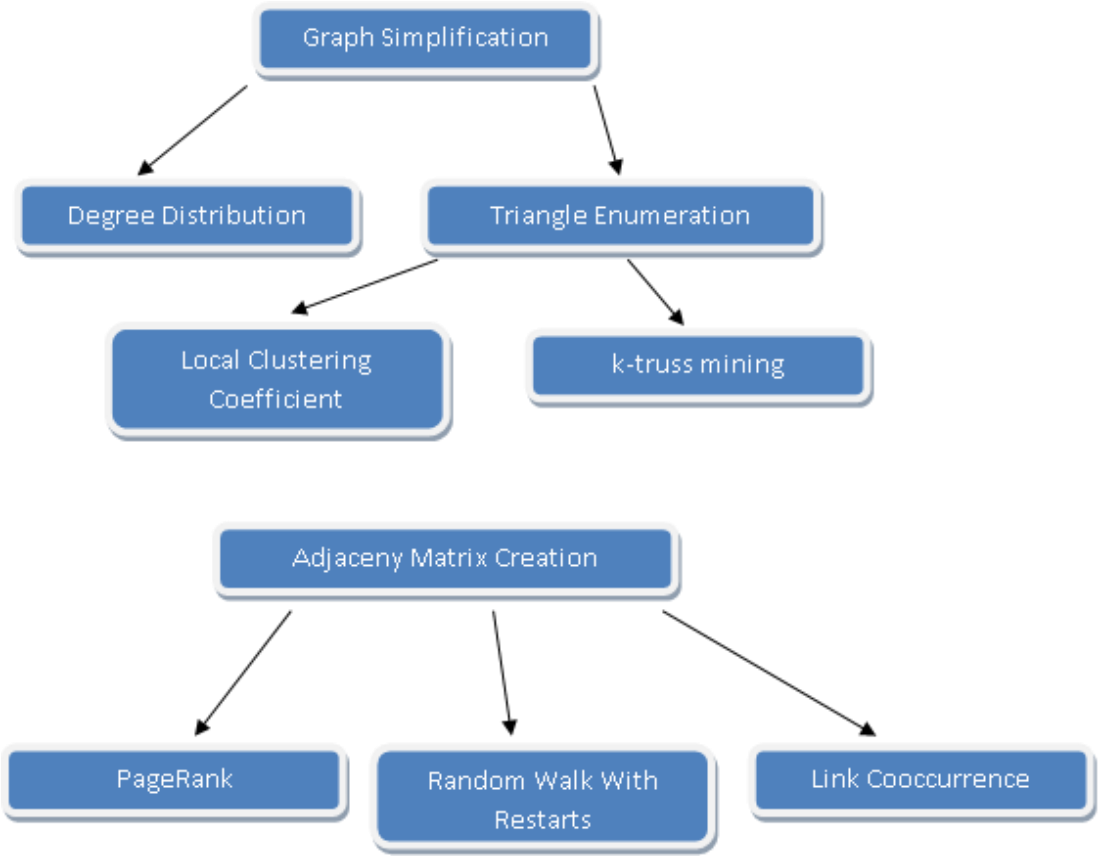

Figure 5 Schematic overview of the dependencies of core and advanced operators in the ROBUST Algebra

The second family of operators follows a deeper mathematical approach to graph mining. A graph is interpreted as a so called adjacency matrix, a square matrix in the space of all vertices where each component indicates the existence or non-existence of an edge. The common pre-processing step here is to transform the graph into that matrix representation and then use algorithms consisting of Linear Algebra operations on that.

5. Status of Implementation and experimental results

Although the actual implementation of the operators and the query algebra is not the subject matter of Deliverable 2.1, we include a short description of the related efforts here. A large number of operators has already been implemented as part of the Apache Mahout library. This framework is distributed under the business friendly open source Apache license. Therefore the code for these operators is already available for use case partners and the general public. It can be executed on the previously described Apache Hadoop platform which is also available under the Apache License and is currently experiencing rapid adoption in the industry. The operators that are currently implemented and have been commited to Mahout include AdjacencyMatrixCreation, PageRank, RandomWalkWithRestart, GraphSimplification, TriangleEnumeration, LocalClusteringCoefficient and DegreeDistribution. The code for these operators has been tested on TUB's local research cluster consisting of 6 machines running Hadoop 0.20.203 with each machine having 2 AMD Opteron 8-core CPUs, 32 GB of memory and 1 Terabyte hard disks. For each operator we are currently running experiments on two graph datasets. These graph datasets have been extracted from the community data supplied by the use case partners SAP and IBM and represent interactions between their respective community members. The IBM dataset consists of approximately 100 million edges and 500,000 vertices. The parallel implementation easily scales beyond this size however. To this extend, TUB will include larger graphs in future experiments (e.g. yahoo song ratings dataset with 400M interactions, snapshot of a twitter follower graph with 1.5B edges).

Table 1 Preliminary results of runtimes of the implemented operators on the two provided data sets from WP 7 and 8.

Algorithm	SAP (100K edges)	IBM (100M edges)	Wikipedia Pagelinks (130M edges)
Graph simplification	1 min	5 min	5 min
Adjacency matrix creation	2 min	6 min	8 min
Degree Distribution	1 min	3 min	4 min
PageRank (10 iterations)	11 min	37 min	39 min
RWR (10 iterations)	12 min	35 min	37 min
Link cooccurrences	5 min	53 min	35 min
Triangle enumeration	2 min	several hours*	-

* This algorithm had an average runtime of 15 minutes when implemented using the Nephele/PACT system.

6. Conclusions and further steps

This report gave an overview of the data sets and their data model dealt with in the ROBUST project, presented the identified core and advanced operators that comprise the ROBUST algebra and discussed the involved parallel processing systems.

Next steps will include a thorough evaluation to which extend the “Bulk Synchronous Parallel Model” might be more appropriate for processing massive graph data sets. In such a context vertices of a graph can receive messages sent by other vertices, send messages to other vertices, modify its own and its outgoing edges' states, and mutate the graph's topology in a series of global so called “supersteps” which are essentially one iteration of a programs are expressed as a sequence of iterations that actually constitutes the program.

Furthermore we will explore to which degree the inference or the sampling on statistical models can be parallelized in conjunction with WP1 as well as means to parallelize matrix multiplications on the investigated parallel processing systems.

A. List of Figures

B. List of Tables

C. List of Abbreviations

Abbreviation	Explanation
DAG	Directed Acyclic Graph
DoW	Decription of work, i.e. GA - Annex I
WP	Work package

D. References

[ACH+09] Daniel J. Abadi, Michael J. Cafarella, Joseph M. Hellerstein, Donald Kossmann, Samuel Madden, Philip A. Bernstein: How Best to Build Web-Scale Data Managers? A Panel Discussion. 35th International Conference on Very Large Databases (PVLDB) 2009

[BEH+10] D. Battre, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele PACTs: a programming model and execution framework for web-scale analytical processing. In Proceedings of the 1st ACM symposium on Cloud computing, pages 119-130. ACM, 2010.

[BHL11] C. Boden, T. Häfele, A. Löser: Classification Algorithms for Relation Prediction. DaLi Workshop at the IEEE International Conference on Data Engineering 2011

[BLN11] C. Boden, A. Löser, C. Nagel, S. Pieper: FactCrawl: A Fact Retrieval Framework for Full-Text Indices. 14th. WEBDB Workshop with SIGMOD 2011

[DG04] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplifed data processing on large clusters. In Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6, pages 10{10, Berkeley, CA, USA, 2004. USENIX Association.

[Had] Apache Hadoop, http://hadoop.apache.org/

[LNP+11] A. Löser, C. Nagel, S. Pieper, C. Boden: Self-Supervised Web Search for Any-k Complete tuples. BeWeb Workshop at the International Conference on Extending Database Technology 2011

[Mah] Apache Mahout, http://mahout.apache.org/

Version history

Version	Date	Author	Comments
0.1	20/09/2011	Christoph Boden	Initial Draft
0.2	23/09/2011	Falk Brauer	SAP Data
0.3	27/09/2011	Inbal Ronen	IBM Data
0.4	27/09/2011	Toby Mostyn	Tidly Wiki Data
0.5	27/09/2011	Sebastian Schelter	Core Graph Operators
0.6	28/09/2011	Christoph Boden	Core Text Operators
0.7	30/09/2011	Sebastian Schelter	Advanced Graph Operators
0.8	04/10/2011	Christoph Boden	Advanced Text Operators
0.9	05/10/2011	Adrian Mocan	Refinement SCN Data
1.0	05/10/2011	Christoph Boden	Finalized Preliminary Draft
1.1	07/10/2011	Alexander Löser	Preliminary Review
1.2	25/10/2011	Christoph Boden	Consilidated Draft
1.3	28/10/2011	Christoph Boden	Final edit after review
1.4	30/10/2011	Christoph Boden	Version for submission

Acknowledgement

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 257859, ROBUST

