UNI
FR
UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

MASTER IN COMPUTER SCIENCE

UNIVERSITÉ DE FRIBOURG

MASTER'S THESIS

Integration of DeepDive (Declarative Knowledge Base Construction) and Apache Spark

Author:
Ehsan FARHADI

Supervisor:
Pr. Philippe CUDRÉ-MAUROUX

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

eXascale Infolab
Department of Computer Science

August 18, 2017

Declaration of Authorship

I, Ehsan FARHADI, declare that this thesis titled, "Integration of DeepDive (Declarative Knowledge Base Construction) and Apache Spark" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: Ehsan Farhadi

Date: 17.08.2017

"Processed data is information. Processed information is knowledge. Processed knowledge is Wisdom."

Ankala V. Subbarao

Acknowledgements

I would like to express my very great appreciation to my thesis adviser Pr. Philippe Cudré-Mauroux for his patient guidance and enthusiastic encouragement. He also supported me in all my difficulties and problems during this thesis. I was not able to finish this thesis without his guidance and support throughout my thesis.

I would also like to offer my special thanks to Alisa Smirnova for all the time and effort she generous spent. Whenever I ran into a trouble during my thesis, the door to Alisa's office was always open.

Finally I must express my very profound gratitude to my parents for providing me with unfailing support and continuous encouragement throughout my entire life.

Thank you very much.

Université de Fribourg

Abstract

Faculty of Science
Department of Computer Science

Master of Science

Integration of DeepDive (Declarative Knowledge Base Construction) and Apache Spark

by Ehsan FARHADI

DeepDive is a newly introduced framework and data management system for creating a knowledge base from a set of unstructured and usually noisy information. Currently DeepDive is based on the relational databases and a powerful statistical analytics engine (Dimmwitted) for learning and inference.

Nowadays, one of the biggest challenges in computer science is dealing with tremendous amount of data; DeepDive is not an exception either. In this thesis we redesign DeepDive's core to integrate it with Apache Spark without losing DeepDive flexibility and power. We extend DeepDive to store the data on Hadoop file system and to process it with Spark and Spark-SQL for higher performance. By leveraging Spark's power of parallelization, we improve DeepDive's runtime up to 20% for large datasets.

Additionally we implemented a Gibbs sampler, similar to DeepDive's statistical analytics engine for a coherent implementation of DeepDive on Spark framework.

Contents

Softeware, Data and Results

- Spark-compatible version of DeepDive is available at:
  https://github.com/farhadie/deepdive
- The implementation of Gibbs sampler on Spark is separately available at:
  https://github.com/farhadie/ddEngine-spark
- The implementation of *birth place* application which we used for our experiments is available at:
  - Spark version: https://github.com/farhadie/birth_place_spark
  - Local/NUMA version: https://github.com/farhadie/birth_place_local
- All the results and log files produced by DeepDive in our experiments are available at: https://github.com/farhadie/dd_results

List of Figures

List of Tables

Chapter 1

Introduction

Creating a knowledge base from a collection of heterogeneous data is a process called knowledge base construction. The input of a KBC system can be in any form, type and volume ranging from a terabytes of noisy and unstructured pile of articles to an incomplete knowledge base. Extracting meaningful, structured information from these sort of data can answer a lot of key scientific question.

DeepDive is a framework for creating Knowledge base construction systems. It offers a declarative language alongside with many other features, that helps domain experts to define and build an end-to-end KBC system without worrying about the details of data management, scalability and underlying algorithms.

On the other hand we have Apache Spark: a general-purpose high-performance cluster-computing framework. Spark is a popular framework among data scientists because it runs applications up to 100 times faster than Hadoop [12] and handles terabytes of data.

The main goal of this thesis is to bring DeepDive and Spark together, leveraging the power of Spark in DeepDive's context. Considering Apache Spark popularity and its promising performance for big data, it is a good match for DeepDive to improve its performance. By integrating these two technologies, instead of relying on relational databases, we store the data on Hadoop file-system (HDFS) and perform all data-processing tasks with Spark.

One of the key challenges that we have to overcome is keeping DeepDive's flexibility, DeepDive sends the data to each function and receives the results. DeepDive supports user-defined function for data processing. Moreover DeepDive provides user with full control over execution of each and every step in a DeepDive application (which is a KBC system). These two feature are essential parts of DeepDive. Therefor, we mapped every DeepDive's underlying operation to an appropriate application for Spark and HDFS, in order to preserve DeepDive's core abilities.

We also implement a statistical analytics engine on Spark for learning and inference. Although we will observe that Dimmwitted engine (DeepDive's original high-performance Gibbs sampler) is very efficient, nevertheless it was interesting to have all the parts running on Spark.

By joining DeepDive and Spark, we manage to reduce DeepDive's data processing runtime up to 20%, which is a noticeable improvement considering the tremendous volume of a KBC system input.

Chapter 2

Preliminaries

In this chapter we address three important background materials for this thesis to set up the notations and definitions that the rest of this document consistently exploits:

1. **Knowledge Base Construction**: We briefly explain the structure of a knowledge base construction system and its operations.
2. **Factor Graph**: Factor graph is a probabilistic graphical model using by DeepDive for learning and inference in order to make prediction of the variable we need to extract from the input.
3. **Apache Spark**: We need to understand how Spark manages and distribute large data over cluster and parallelizes tasks and operations.

2.1 Knowledge Base Construction (KBC)

Knowledge base construction is the process of creating a structured knowledge base with the data extracted from a set of inputs. The input of KBC is usually a collection of heterogeneous and noisy data; it could be in any form, ranging from text documents and articles, tables, images, PDFs, audios, ..., or even another KB. The output of KBC is a structured knowledge base, populated with the facts extracted from the input. This process usually involves extraction, cleaning and integration of the data.

2.1.1 KBC Model

In a standard KBC model, there are four types of objects that we need to extract from the inputs:

- **Entity**: An entity is a distinct existence that could be anything in real world. For example it is a person, a place, an object, etc.
- **Mention**: A mention is any form of referring to an entity. A phrase, an abbreviation, an image, ..., all can be different forms of a mention. For example "Jon", "Jon Snow", "The white wolf" are three different mentions to the same entity of "Jon Snow".
- **Relation**: A relation is the way two (or more) entities are connected. Again, for example entity "Jon" is associated with entity "Sam" with a "friendship" relation.
- **Relation Mention**: A relation mention is the representation of a relation. For instance, the sentence "Jon and Sam are old friends.", is a representation of the relation "friendship" between two entities "Jon Snow" and "Samwell Tarly".

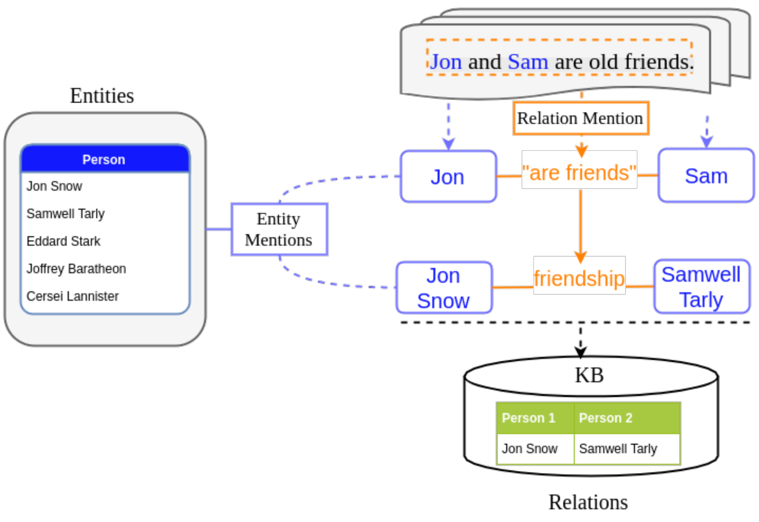


FIGURE 2.1: An illustration of the KBC model

2.1.2 KBC Process

The process in a typical KBC system consists of four consecutive tasks:

- **Data Preprocessing**: In this step KBC system cleans and prepares data be processed. For example if the input of the system is a set of PDF files, we need to convert PDF files into a text format, then we have to extract each sentence and apply natural language processing on it so that it is understandable for machine and processed.
- **Feature Extraction**: A feature is a measurable property or characteristic of a variable. Feature is an important concept in machine learning because by extracting the features of a phenomenon or a variable, we can measure and compare it, and in general use it as evidence to make prediction about the associated variable. Therefore feature extraction is one the main steps in a KBC system. For example given the sentence "Jon and Sam are old friends.", "...are...friends." is a feature that is extracted from the sentence and used to predict the relation between "Jon" and "Sam".

  One thing to notice is that different features might have different weights (or influence). For example, given two sentences "Jon and Sam are old friends." and "Sam have known Jon for 10 years.", "being friends" is a much stronger feature than "have known for 10 years" for predicting the relation "friendship" between two entities "Jon" and "Sam".
- **Supervision**: One important step in machine learning is collecting training data. Since in real-world problems there is not enough training examples for each relation, and even if there is, it is tedious and expensive task, we need to be able to create a training set from the existing data. One common solution to this problem is distant supervision. Distant supervision maps an incomplete and small knowledge base (training data) to the noisy existing data and creates a larger training set. For example, knowing that "Jon" and "Sam" are friends, we find all the sentences containing the mentions of these two entities and use them as training data.

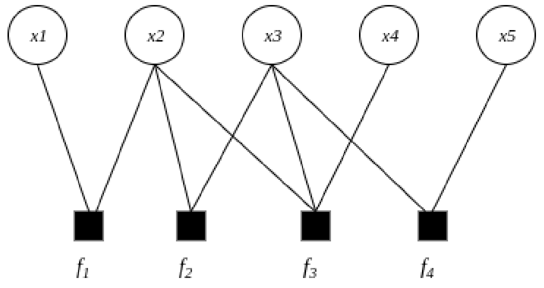


FIGURE 2.2: A factor graph representation of $g(x_1, x_2, x_3, x_4, x_5) = f_1(x_1, x_2).f_2(x_2, x_3).f_3(x_2, x_3, x_4).f_4(x_3, x_5)$

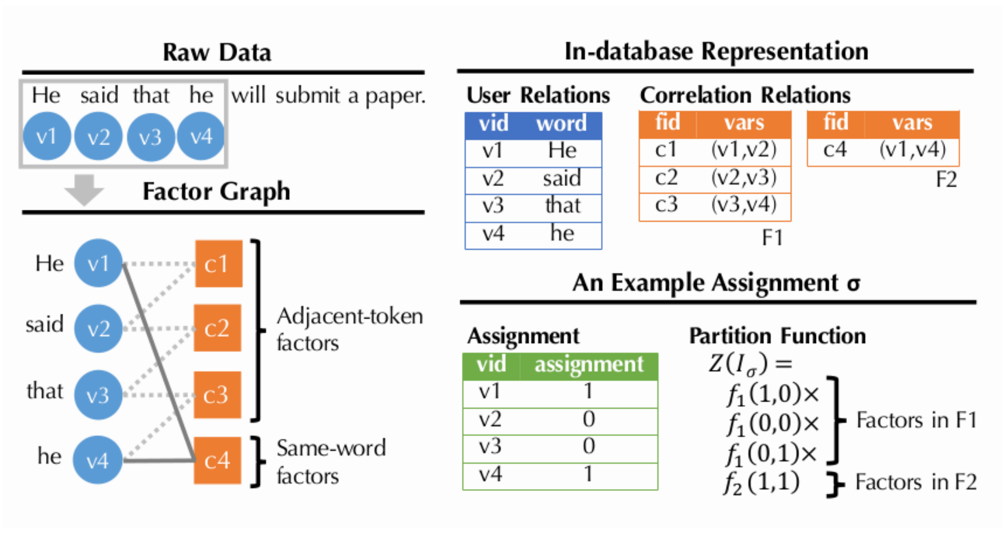
- **Statistical Inference and Learning**: After creating a training data, KBC system learns the weight of each feature and using these weights, it predicts the probability of each variable taking a particular value. This is the main step in a KBC system where system actually finds the relations between all entities and put them in a KB.
- **Iterative Refinement**: After performing four previous steps, we do not necessarily produce a correct and complete KB. Every system need to be tuned in order to achieve the best results, a KBC system is not an exception. One must observe the results from previous steps and improve the system by refining each step (for example by defining additional features, new inference rules, providing training data, changing user-defined functions and etc.) so that the result becomes as accurate as possible.

2.2 Factor Graph

As we will see later in chapter 4, DeepDive heavily relies on a factor graph, a probabilistic graphical model, for its statistical inference and learning. Here we introduce factor graphs in a brief.

In figure 2.2 we show an example of a factor graph. Factor graph is a bipartite graph representing the factorization of a function with two types of nodes, variable nodes for each variable and factor nodes for local factors. Edges in factor graph are only between one variable node and one factor node. An edge exists between variable node x_i and factor node f_j if and only if x_i is an argument of f_j [5].

In DeepDive context, we define a probabilistic database to be $\mathcal{D} = (\mathcal{R}, \mathcal{F})$ where $\mathcal{R}$ is the user schema, containing the random variable we defined in DeepDive application, and $\mathcal{F}$ is the correlation schema which holds the probability distribution over the factor graph by showing the correlation between random variable in user schema. In the user schema, each tuple has a unique ID taking values from the domain $\mathbb{D}$ and is associated with a value, taken from the domain $\mathbb{V}$. Each distinct variable assignment $\sigma : \mathbb{D} \mapsto \mathbb{V}$, defines a possible world $\mathcal{I}_\sigma$ [10]. We can see an illustration of this process in figure 2.3. In correlation schema $\mathcal{F}$, each correlation relation $F_j \in \mathcal{F}$ represents the correlation between variables in $\mathbb{D}$ and has the form $F_j(fid, \overline{v})$ where Fid is a unique ID taken values from domain $\mathbb{F}$, and $\overline{v} \in \mathbb{D}^{a_j}$ where a_j is the number of random variables that are correlated by each factor. To specify how F_j is associated with a function $f_j : \mathbb{V}^{a_j} \mapsto \mathbb{R}$ and a real-number weight w_j, given a possible world I_σ, for any $t = (fid, v_1, ..., v_{a_j}) \in F_j$, define $g_j(t, I_\sigma) = w_j f_j(\sigma(v_1), ..., \sigma(v_{a_j}))$.



vid	word
v1	He
v2	said
v3	that
v4	he

fid	vars
c1	(v1,v2)
c2	(v2,v3)
c3	(v3,v4)

fid	vars
c4	(v1,v4)

vid	assignment
v1	1
v2	0
v3	0
v4	1

$Z(I_\sigma) = f_1(1,0) \times f_1(0,0) \times f_1(0,1) \times f_2(1,1)$

FIGURE 2.3: An illustration of factor graph, representing user schema and correction schema, and random assignments in DeepDive; taken from [14].

2.2.1 Probability Distribution

Given the set of all possible worlds $\mathcal{I}$, the probability of a possible world $I \in \mathcal{I}$ is [14]:

$$Pr[I] = Z(I) \left(\sum_{J \in \mathcal{I}} Z(J) \right)^{-1}. \tag{2.1}$$

where $Z(I)$ is the normalizing constant (known as *partition function*) and it is defined $Z : \mathcal{I} \mapsto \mathbb{R}_+$ over any possible world $I \in \mathcal{I}$ as [14]:

$$Z(I) = \exp \left\{ \sum_{F_j \in \mathcal{F}} \sum_{t \in F_j} g_j(t, I) \right\} \tag{2.2}$$

In this factor graph which defines a probability distribution of all the variables, (marginal) inference refers to computing the probability of a random variable taking a particular value. For example in a factor graph with boolean variables, for each variable v_i let $\mathcal{I}_e^+$ be the all possible worlds where variable v_i is TRUE, and $\mathcal{I}_e^-$ be the all possible worlds where variable v_i is FALSE. The marginal probability of v_i is defined as follows:

$$Pr[v_i] = \frac{\sum_{I \in \mathcal{I}_e^+} Z(I)}{\sum_{I \in \mathcal{I}_e^+} Z(I) + \sum_{I \in \mathcal{I}_e^-} Z(I)} \tag{2.3}$$

2.2.2 Gibbs Sampling

Inference in factor graph is computed using sum-product algorithm [5], however exact inference is known to be intractable in large factor graphs [7, 11], thus a common solution for computing inference is to use Gibbs sampling. The main concept is that by taking samples from the probability distribution we estimate the probability of each variable with high accuracy [4].

Gibbs sampling algorithm is defined as the following three steps:

1. Let V_i be a randomly assigned value for variable v_i. We choose a random possible world $I_0 = \{V_1, V_2, V_3, ..., V_n\}$.

2. For each variable v_i we sample a new value for that variable according to the conditional probability $Pr[v_i|V_1, V_2, ..., V_{i-1}, V_{i+1}, ..., V_n]$ and update v_i with this new value. After iteration over all variables and updating all variables, we obtain I_1. For sampling next variables sampler should use updated values, otherwise the quality of samples and convergence rate decreases.

3. Repeating previous step for k times, which gives us k samples. Then we simply compute the marginal probability of v_i over these samples. Using the *Law of total probability*, the marginal probability of v_i is the sum of probabilities of possibles worlds that are consistent with v_i taking a particular value.

One interesting feature of Gibbs sampling is that for each sample, we actually don't need to compute the probability of V_i given all other variables $Pr[v_i|V_1, V_2, ..., V_n]$, instead the probability of v_i depends only on the variables that are correlated with v_i. This notion is call *Markov blanket*.

Markov Blanket: Defining $vars(fid) = \{v1, ..., v_{a_j}\}$ as set of variables associated with each factor, for each variable node in factor graph we define Markov blanket of v_i, denoted $mb(v_i)$, as:

$$mb(v_i) = \{v|v \neq v_i, \exists fid \in \mathbb{F} \text{ s.t. } \{v, v_i\} \subseteq vars(fid)\} \tag{2.4}$$

This equation simply means that $mb(v_i)$ is a set of variables that are connected to v_i with only one factor in between. For example in the sample factor graph in figure 2.2, $mb(x_2) = \{x_1, x_3, x_4\}$. DeepDive uses this technique to perform Gibbs sampling in parallel for the variables that do not have overlapping Markov blankets. This is the technique used in Dimmwitted engine, and the one we leverage in Spark for parallelizing learning and inference phase.

2.2.3 Weight Learning

Another important operation over the factor graph is learning the weights associated with each factor. We define $\mathcal{I}_\text{e}$ as set of possible worlds consistent with our evidence variables. For weight learning we need to maximize the probability of $\mathcal{I}_\text{e}$:

$$arg\, max_{w_j} \frac{\sum_{I\in\mathcal{I}_\text{e}} Z(I)}{\sum_{I\in\mathcal{I}} Z(I)} \tag{2.5}$$

There are multiple approaches for solving this problem. DeepDive uses stochastic gradient descent (SGD) and estimate the gradient with samples draw from Gibbs sampling.

2.3 Apache Spark

Spark is a highly-scalable, high-performance and general-purpose cluster computing framework. It is claimed to be up to 10x faster than Hadoop in iterative machine learning operations [13] and also it supports several programming languages. A Spark cluster uses master/worker architecture and consists of a *Driver Program*, a

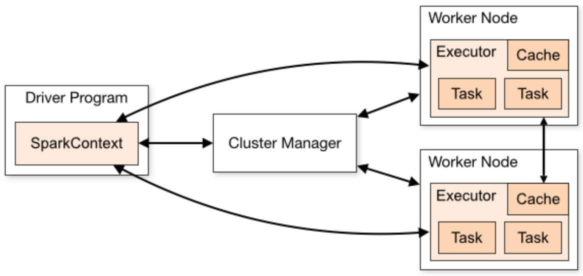


FIGURE 2.4: Spark cluster overview [9]

Cluster Manager and several *Worker Nodes*. As we can see in Figure 2.4, an Spark application runs as independent sets of processes called *Executors* (which are responsible for actual computations and storing the data) on the cluster, coordinated by driver program.

Spark uses several techniques for parallel computation which we briefly look into:

- **Resilient Distributed Dataset (RDD), DataFrame and DataSet**: Resilient Distributed Dataset is the main abstraction in Spark. It represents collection of objects partitioned over a cluster of machines as one object. RDDs are used to process huge amount of data in parallel, and if a partition of data is lost due to a machine failure, it is quickly rebuilt. RDD data can be stored on disk, cached on the memory for faster operation, or even replicated across the cluster; giving the user the choice between cost of space or access time.
  DataSet and DataFrame are improved versions of RDD. The key differences are: (1) they organize data into named columns, so it is more convenient for storing and processing relational queries. (2) Dataframe is faster for large data because it only transmit data without the schema between Spark nodes instead of serializing them.
  One important thing to have in mind, is that all three representations are immutable objects. Thus it is not possible to update a variable in an RDD, instead for every operation the whole RDD is recomputed.

- **Shared Variables**: Other than RDDs Spark lets users create two other types of variables. *Broadcast variables*, which are read-only variables replicated on every worker node for faster access time. *Accumulators*, which are variables that executors can only "add" to them, they are mostly used for implementing counters.

- **Parallel Operations**: There are two types of operation user can perform on RDDs: *Transformations* which create a new dataset from an existing one, and *actions*, which return a value to the driver program after running a computation over dataset. One should always have in mind that by performing an action on a large dataset, the returned data might exceed the memory of the driver program and crashes whole program.

Chapter 3

DeepDive Framework

In this chapter we review the structure and semantics of DeepDive framework. We see how one can design a KBC system with DeepDive, how it executes each step of a KBC system and finally we explain Dimmwitted engine and its optimization techniques to see how it is designed and how can we implement a similar Gibbs sampler in Spark.

3.1 DeepDive Application

For Defining a KBC system in DeepDive, we need to specify all the steps. DeepDive provides a Datalog-like language for defining a KBC system. Using this language we specify the data flow in our KBC system, feature extraction methods, entity-linking, supervision and learning and inference rules.

First, let's take a look at a DeepDive application structure. A DeepDive application consists of:

- **DeepDive Program**: In every application we have a main file called *app.ddlog*. This is where we actually define the data flow in our KBC system using the Datalog-like syntax called *DD-log*. In this file we define all the inputs, variables and relations, feature extraction, candidate generation, labeling and the statistical model that we're going to use for learning and inference.

- **DeepDive Configurations**: Before running the DeepDive application we have to configure DeepDive. We need to specify the Database engine that we are going to use, number of NUMA[1] nodes. We also need to configure the Dimmwitted engine according to our application (e.g. number of samples we need in Gibbs sampling).

- **User Defined Function**: DeepDive relies on relational databases for its operation, but it gives the user freedom of using functions written in any other language, for further data processing. This is what makes DeepDive very flexible. At each step, in addition to normal derivation rules that we define with DDlog, we can use external functions by specify the input and output of them. DeepDive integrates external functions and glue them together as a unified system, by automatically sending the appropriate data to each function and fetching the results.

Now Since we have an understanding of the structure of a DeepDive application we explain phases of a *DeepDive program*. A DeepDive program generally consists of several consecutive phases:

[1]Non-Uniform Memory Access: A memory architecture for multiprocessors where processors have access to a distributed shared memory.

```
sentences(
    doc_id         text,
    sentence_index int,
    sentence_text  text,
    tokens         text[],
    lemmas         text[],
    pos_tags       text[],
    ner_tags       text[],
    doc_offsets    int[],
    dep_types      text[],
    dep_tokens     int[]
).
function nlp_markup over (
        doc_id text,
        content text
    ) returns rows like sentences
    implementation "udf/nlp_markup.sh" handles tsv lines.

sentences += nlp_markup(doc_id, content) :-
    articles(doc_id, content).
```

FIGURE 3.1: An example of DeepDive program in DDlog language, defining "sentences" relation, extracted from "articles" after applying natural language processing in a user defined function

- **Data Preprocessing, Candidate Generation and Feature Extraction**: The first phase is to preprocess raw inputs and populate the database with clean and machine-understandable data. Then we need to generate candidates. Candidates are the possible mentions, entities and relation mentions in the noisy input. Afterwards, we need to extract the features correlated with each candidate using a user-defined function. For example, for a dataset of articles, we apply the *natural language processing (NLP)* on all sentences, and then use NLP tags as features. All these steps are translated into a collection of SQL queries and UDFs that user defines in the program and all the data is stored in the database.
- **Supervision**: DeepDive has two techniques for generating training data:
  - **Hand Labeling**: By labeling candidates we can add training data for our model. Labeling can be a manual task or a user-defined function finding candidates with a specific property.
  - **Distant Supervision**: This is a popular technique for generating training data using a small set of evidences and a large unlabeled data. In Distant supervision, given an evidence relation between multiple entities, we find all the relation mentions containing those entities and heuristically label them as true evidence
- **Learning and Inference**: In this phase, DeepDive uses the extracted features and inference rules defined in application to ground a factor graph. Grounding is the process of creating a factor graph from entities and relations extracted from the input, and storing the factor graph to disk so that it can be used to perform inference. After grounding DeepDive performs weight learning and marginal inference to predict the probability of variables having a specific value.

3.2 DeepDive Architecture

In this thesis we expand the underlying architecture of DeepDive to make it compatible with Spark. For this purpose, it is necessary to explain the underlying structure of it. We briefly mention important components of DeepDive framework and their role.

Compiler: Every DeepDive application needs to be compiled before its execution. DeepDive compiler takes an application code, convert it to a JSON file which describes the whole system like a blueprint. Then DeepDive uses this blueprint to create executable scripts specified for each step of application, according to configurations of the application. These executable files are stored in the same location as the application itself, and each of them can be executed by DeepDive user interface.

Database Connector: We already explained that DeepDive relies on relational databases. DeepDive has connectors for each type of database. These connectors translate DeepDive internal and underlying commands into the queries understandable for the database. This way DeepDive abstracts the database layer from other components by offering a unified interface for all supported databases.

Runner: A DeepDive application can run on a single core, several cores, or even several machines in parallel. The runner abstracts the execution resources from the other components. The runner consists of several *compute drivers* for each execution mode (i.e. local and cluster). Most of the scripts produced by compiler, are SQL queries that DeepDive passes to the database; but the other important part, user-defined functions, are normal application and the need to be executed in operation system environment. For each UDF, DeepDive has to (1) prepare the appropriate input by fetching them from database, (2) splitting fetched data to the number of process and passing them to UDF, (3) execute the UDF, and finally, (4) write back the output to the database. Moreover this is the component that handle the parallel execution of UDF on a cluster of machines. It partitions and sends the jobs to each machine over SSH and retrieves the results.

Inference: For learning and Inference DeepDive has a statistical analytic engine called Dimmwitted engine. This component takes the grounded factor graph and passes it to Dimmwitted engine for learning and inference and at the end writes the results to the database.

User Interface: DeepDive has a simple user interface for controlling the execution of each step of the application. Each step can be executed separately from rest of the application. additionally, A user can execute SQL queries on the database, using DeepDive UI. This is a great tool for debugging and analyzing an application.

3.3 Statistical Analytics

A factor graph is used in DeepDive for its ability to model complex correlations among random variables. But as we mentioned in chapter one, it is intractable to compute the exact inference over large factor graphs [11]. One of the popular methods to overcome this problem is Gibbs sampling. Combining factor graph and Gibbs sampling is not a new approach, for example OpenBUGS framework has been using this technique since 1980s [6]. Using Gibbs sampling for any factor graph, we can estimate the probability distribution of variables. Clearly, by getting more samples from the factor graph, we obtain results with higher quality and accuracy [8].

As we described in chapter 2, the input of Gibbs sampler is a bipartite factor graph $G = (X, Y, E)$ where X is the set of random variable we want to predict, Y is the set of factors representing the correlation between random variables X, and E is the set of edges connecting variable and factors. In Gibbs sampler we perform 3 steps for each variable v_i on each step of the computations:

1. Retrieve Markov blanket of the variable v_i.
2. Evaluate factors in $mb(v_i)$ given the assignments of variables in that step.
3. Aggregate evaluations of all factors in $mb(b_i)$, compute the probability of v_i and update the factor graph with new sample taken from $P[v_i|mb(v_i)]$.

These three steps are the *core operation* we loop over for every variable on each step, to achieve high-quality samples. Because 99% of the execution time is spent on core operation [14], we need to focus on optimizing this operation. There are three trade-offs for optimizing core operation [14]:

1. **Materialization trade-offs**: Materialization is the process of computing a query and storing the results for further use. During core operation we access the factor graph many times, not only reading from it, but also updating variables. One of the obvious optimization is to materialize only a portion of factor graph to minimize repeated accesses to it.
2. **Page-oriented layout**: Beside materialization, we have to optimize the variables and factors assignment to each page to minimize number of the page fetches from the storage (i.e disk). Since Gibbs sampling iterates through variables in a random pattern (for better convergence speed), we need to design a layout that has good average-case performance.
3. **Buffer-Replacement Policy**: One of the main challenges with large data, is to decide which page in memory has to be replaced when memory is full and we need to load another page from the storage. A good policy regarding the access pattern of our application helps avoid repeated page replacements.

3.4 Dimmwitted engine

in chapter 2, we introduced the user schema and correlation schema in chapter one and how we map them to a factor graph. Here we define two other relations for representing the factor graph,the edge relation:

$$E(v, f) = \{(v, f) | f \in \mathbb{F}, v \in vars(f)\} \tag{3.1}$$

and a sampled possible world:

$$A(\underline{v}, a) \subseteq \mathbb{D} \times \mathbb{V} \tag{3.2}$$

In equation 3.2, A is a set of pairs containing a variable and its assigned value and A is modified in each iteration over variables during Gibbs sampling.

According to these equations, the input of Gibbs sampler would be:

$$Q(v, f, v', a') \leftarrow E(v, f), E(v', f), A(v', a'), v \neq v' \tag{3.3}$$

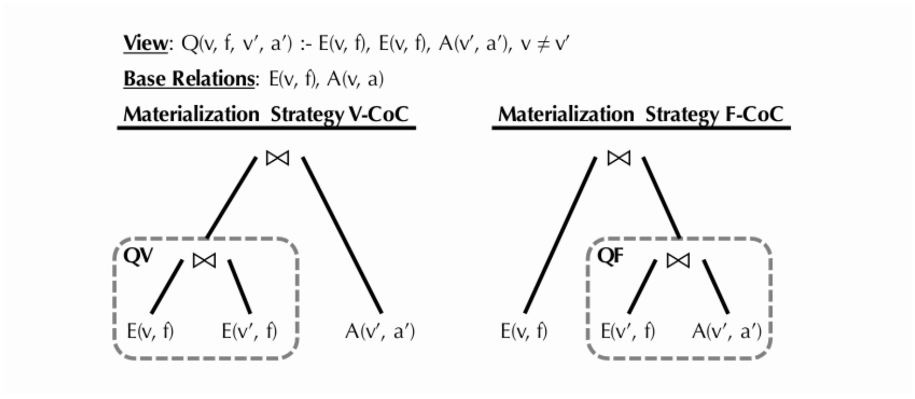

View: Q(v, f, v′, a′) :- E(v, f), E(v, f), A(v′, a′), v ≠ v′
Base Relations: E(v, f), A(v, a)
Materialization Strategy V-CoC
Materialization Strategy F-CoC
QV
QF
E(v, f)
E(v′, f)
A(v′, a′)

FIGURE 3.2: Strategies V-Coc and F-Coc. [14]

Dimmwitted groups all the Q tuples by their first field v. It simply put, each grouped tuples of Q, contains all the variables that exist in $mb(v)$ and their assignment, along with all the factors connecting them to v. This is all the information that we need to sample v by computing the probability $Pr\,[v|mb(v)]$.

For better convergence in Gibbs sampling, we need to sample variable randomly. Dimmwitted proceeds by randomly sampling variables in a group of Q, after one pass through Q, there will be a new possible world. Repeating this process gives us multiple samples and as we explain using samples we can compute marginal probability of v.

Now we briefly introduce Dimmwitted's optimizations for each of the mentioned trade-offs for accessing Q.

3.4.1 Materializing the factor graph

Materialization reduces number of random reads, but we have to be aware that the relation A is updated on every iteration over the Q groups, and this causes to random writes as well. There are two strategies that Dimmwitted is using for materializing [14]:

- **V-CoC**: If we consider Q as 3 joins between 2 relations E and A, we need to to improve performance of computing these joins. One strategy is to co-cluster on the variable side of the Q:

  $$QV(v, fv') \leftarrow E(v, f), E(v', f), v \neq v'.$$

  Meaning that we compute QV and store the result in memory for further use. Using this strategy we eliminate random reads on E, and we write to A once after passing over all Q tuples.

- **F-CoC**: like the previous strategy, except we co-cluster on factor side

  $$QF(f, v', a) \leftarrow E(v', f), A(v', a).$$

  In this strategy, we removed random reads on A, but we have to update A after each iteration on variables in Q.

So if we consider the cost as total number of random reads and writes we can see the final cost for each strategy in table 3.1. Before starting to sample, Dimmwitted

	Reads		Writes
	From A	From E	To A
V-Coc	| mb(v) |	0	1
F-Coc	0	d_v	d_v

TABLE 3.1: I/O cost for materialization strategies. For a variable v, d_v is the number of factors connected to v. The cost is total number of reads and writes.

decides which strategy is more efficient and uses that strategy for materializing the factor graph.

3.4.2 Page-oriented Layout

Even though the iterations over variables (either in E, or A) are random, if we capture the order of iterations over variables for each step and use it to pack them in the same order in pages, it shows that it has an order of magnitude improvement over a randomly packed variables [14].

3.4.3 Buffer-Replacement Policy

For large dataset that does not fit in the memory, we need a strategy for replacing pages in memory. We have to decide which page has to be evicted from the memory. When Dimmwitted passes over variables in Q for the first time, it records the full sequence of the iteration and using it to define a theoretically optimal eviction strategy: *evict the item that will be used latest in the future* [1].

For the factors, on the other hand, Dimmwitted logs the sequence of factor references in a file. Then, for each reference it passes over this log file and computes when each factor is going to be used next in the sequence and generates a log file consisting of pairs, and when they will appear next.

3.5 Conclusion

In this chapter we reviewed the underlying architecture of DeepDive framework and laid down the necessary details of DeepDive to know what parts we have to update in order to make it compatible with Spark. As we explained, DeepDive is a solid and very well designed framework, specially Dimmwitted engine which is one of the fastest Gibbs sampler among its competitors [14].

Chapter 4

DeepDive and Spark Integration

In previous chapter we explained DeepDive structure and important parts. In this chapter we go through our contribution to make DeepDive work on Spark as well. First we explain how can use Spark for general data processing, afterwards we explain how we create our own statistical analytic engine similar to Dimmwitted on Spark.

4.1 Adding Spark Compatibility to DeepDive

In order to add Spark Compatibility to DeepDive we make a several changes. Before going through the rest of this section, we mention challenges we faced through this thesis.

4.1.1 Challenges

We have two main challenge to integrate DeepDive with Spark:

- The main difference of Spark and a relational database that we have to have in mind, is that Spark is not a database. Spark processes the data like a database but it is not a storage system. Therefor, when we want to use Spark instead of a database we need to think of storing the data in an efficient and accessible way.
- Another challenge is that in Spark, the submitted application can not change on the fly. Each application should be compiled, packed with all its dependencies and sent to all worker nodes to be executed in parallel.

  On the other hand, a DeepDive application consists of several independent steps: each step can (and should, for debugging purposes) be executed separately. Plus there are user-defined functions that are autonomous basically applications. Therefore we simply can not bundle all the steps and UDFs together and send them to Spark as one application.

Keeping the mentioned points above in mind, we continue to explain our contribution to in this thesis.

4.1.2 Design Choice

Spark is compatible with several storage system but because the benefits of running Spark on top of YARN[1] and HDFS[2] we decide to store the data on a Hadoop file system and run Spark on top of YARN and HDFS. The advantages of running Spark with Hadoop are:

- Using HDFS means that there's no need to transfer data to the location of code. Since we use same node for storing the data and for processing them, the code and the data are in the same place.
- Each Spark job runs on the same node which is storing the data without any need to retransmit the data even between the nodes of the cluster (data locality).
- Leveraging existing Hadoop cluster.

DeepDive has two important features, user-defined functions and interactive interface. DeepDive allows user to process data in any way and any language it suits them and also to execute each step of DeepDive program separately for debugging and testing.

In DeepDive, compiler converts application code (App.ddlog) into a set of independent executable files, one for each step of DeepDive application. Each executable file contains a set of database queries and shell commands. Clearly our design also had to conform the two features, that is why we decided to keep DeepDive compiler and user interface unchanged. To do so, we only changed the underlaying parts of DeepDive and mapped database queries and shell commands to an Spark application. This way each step of DeepDive application is sent to Spark and executed separately. The trade-off for this design is that in Spark, for executing each step of DeepDive application you have to allocate resources, load data from HDFS, process the data, save results on HDFS and release resources afterwards. This put a constant overhead on every step separately disregarding the volume of the data. Therefore for DeepDive applications with small input, the Spark overhead is considerable, but for large datasets this constant is rather negligible.

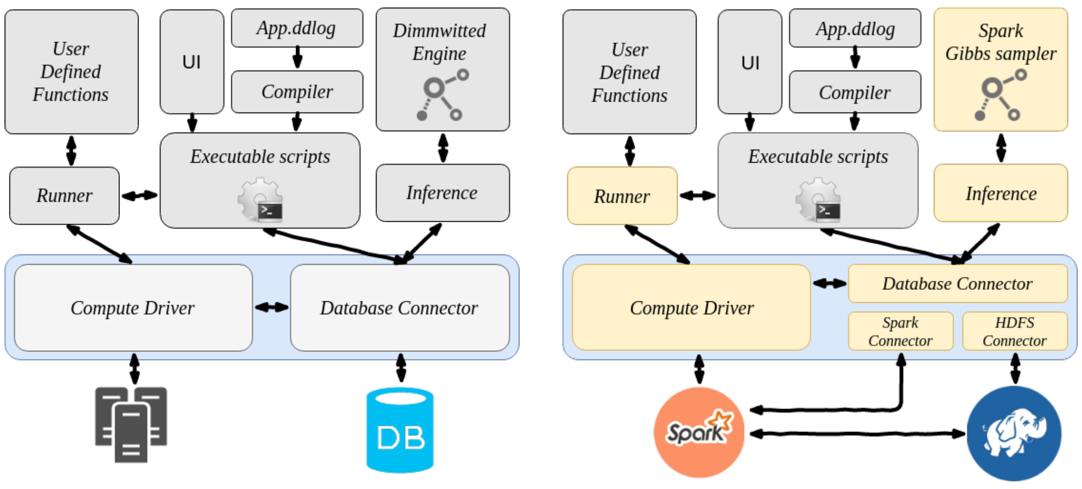
4.1.3 Spark/HDFS connector

In order to replace Spark with a database, an special database connector should be deigned to store data in Hadoop and process them using Spark. First thing to notice is that not every database query can be passed to the Spark directly. Some queries can be parsed and passed to HDFS instead of Spark (e.g. dropping a table in a relational database, is simply a delete in HDFS). Therefore, we divide DeepDive commands into two groups, create a mini-connector for each: (1) HDFS-connector and (2) Spark-connector. Additionally we use a switching script to send each query to the corresponding mini-connector. The first group of queries that do not need Spark processing, are sent to HDFS-connector, they are parsed, transformed and executed directly by calling an HDFS command. The second group on the other hand, get passed to Spark-connector which itself is a Spark application for further parsing and executing on Spark cluster.

In the Spark-connector which most of the work is done, the data from an HDFS or a local directory is transformed into Datasets and after processing, stored in HDFS.

[1]Yet Another Resource Negotiator: http://yarnpkg.com/

[2]Hadoop Distributed File System: http://hadoop.apache.org/



(A) Original Architecture of DeepDive

(B) modified version of DeepDive with Spark support (Yellow boxes are modified components)

FIGURE 4.1: An illustration of DeepDive architecture

There are multiple data formats that we can use for storing data (e.g. Parquet, HBase, kudo, etc.), we discuss the two main data formats:

- **Parquet format**: The most important feature of parquet is that it is a columnar data storage format, thus, it has better read and write speed when it is needed to access only a few columns of a relation and it makes parquet an efficient data format for data analytics. Additionally, parquet includes optimization encodings (i.e. RLE, Dictionary, Bit packing) for better data compression.

- **Avro format**: Unlike parquet, avro is a row-oriented format. Avro has lightweight and fast data serialization and deserialization, and even though it does not have internal index it uses directory-based partitioning techniques for fast random accesses.

Parquet has better overall performance over other storage formats including avro. With Parquet and Snappy compression[3], we can reduce total volume of data by factor of 10 [2].

Most of the SQL queries (e.g. select, join, etc.) that are created by DeepDive's compiler and passed to Spark-connector and, are processed by Spark-SQL. Each query has to be parsed to be compatible with Spark-SQL syntax (which is Hive SQL). Spark-SQL executes these queries against a Dataframe in parallel and returns the result in a new Dataframe.

4.1.4 Runner

As we described earlier, an important feature of DeepDive is user defined functions. DeepDive has to be able to provide the right input for each function, and fetch the result of each as well. In order to do that, we add a *compute-driver* to Runner called

[3]A fast data compressor/decompressor: https://google.github.io/snappy

```scala
import diuf.exascale.deepdive.udf.wrapper.Deepdive

object PersonMention extends Deepdive{
  def main(args: Array[String]): Unit = {
      val spark = SparkSession.builder() //defning a Spark Session

      load_tables(args,spark)
      ...//user data processing
      save(Output_DataFrame)
  }
}
```

FIGURE 4.2: An example of using Spark wrapper in user-defined functions in *Scala*

Spark-driver. This driver is compatible with other components of DeepDive and submit each one to the Spark cluster, letting them to be executed directly on Spark and in parallel.

In a locally executed DeepDive application, inputs should be fetched from the database, prepared and passed to each UDF. But in this case, considering that data is already stored on HDFS, this driver only has to point to the input data. This makes DeepDive more efficient and faster specially for large data.

Because these UDFs are executed separately, they should be compatible with Spark. User defined functions can be in any language that Spark can support (i.e. Java, Scala, Python), and like local UDFs, they have to implement a common API for communicating with DeepDive.

Just like DeepDive original design, we create a wrapper *Class* for user-defined functions in Scala and Python languages. User should extend this class when they are writing a UDF. This class have two methods: (1) *load_table*: which helps DeepDive to find the right table on HDFS and load them into Spark Session, so Spark-SQL can run queries on them. (2) *save*: which helps DeepDive to store the output of a UDF in the correct manner in HDFS. We can see an example of DeepDive wrapper in figure 4.2

4.1.5 Extra components

DeepDive ships with two extra component for two major common tasks in most KBC systems: natural language processing engine and feature extraction library.

- **Natural Language Processing**: One of the most common operation in a KBC system is to parse and process text documents. DeepDive includes *Stanford NLP Core* for this task. NLP core gets a whole article as input, strips sentences, add multiple NLP markups (e.g. NER tags, POS tags, etc.) to them, and produces a clean and structured table. To make NLP Core compatible with Spark, we use a NLP Core wrapper developed by *databricks* [3].

- **Feature Extraction Library**: Every KBC system needs a specific feature extraction algorithm. But DeepDive has a general feature extraction library written in *Python* (called DDlib) for obtaining features of a sentence, using the NLP markups generated by Stanford NLP core. Since Spark supports python, we add a wrapper for this library to make it compatible with Spark.

4.2 Gibbs Sampling on Spark

Dimmwitted engine is a high-throughput Gibbs sampler designed and tailored for DeepDive and its statistical analytic operations. Dimmwitted engine is written mostly in C++, therefor as we explained, it can directly manages memory for better performance. On the other hand, Spark is a general purpose framework. So it is reasonable to assume Dimmwitted performance surpasses any Gibbs samplers implemented on Spark. Although we can still use spark for processing data in all the previous steps of DeepDive (e.g. Feature extraction, candidate generation, etc.), and then pass the generated factor graph to Dimmwitted engine for best performance. Nevertheless, we implement a Gibbs sampler on Spark in order to make our contribution coherent. In this section we discuss how we implement a Gibbs sampler similar to Dimmwitted, compatible with Spark.

In this section first we introduce the challenges we are facing in implementation of Gibbs sampler on Spark, then we go through trade-offs we introduced in previous chapter, and we describe our implementation.

4.2.1 Challenges

One of the Spark limitations is lack of a shared memory between its worker nodes. RDDs and DataFrames are immutable objects and updating them are costly, specially when we're dealing with large data. Although there are some shared variables like broadcast variable and accumulator that we can use, but they are not flexible enough for a shared read and write variable between nodes.

The other issue is that Spark framework only offers limited control over memory and storage management, so obviously we can not implement every strategy used in Dimmwitted on Spark and we have to find a workaround for this problem.

4.2.2 Spark Tools

We quickly review the configurations and tunning techniques in Spark that we can leverage for our purpose:

- **Partitioning**: As we described in chapter 2, RDDs, DataFrames and DataSets are the main abstractions of Spark over the data. RDDs are partitioned among worker nodes, and operations are applied on each partition separately and parallel. In Spark we can control the partitioning process. We can group our data and put each group in different nodes.
- **RDD Persistence**: By default each RDD is recomputed each time we execute an action on it. However we can *persist* an RDD to avoid re-computations. There are several persisting levels in spark:
  - **Memory Only**: Persist RDD as deserialized objects on memory. If RDD does not fit on memory some partitions simply will not be cached and will be re-computed on the fly.
  - **Memory and Disk**: Persist RDD as deserialized objects on memory. If RDD does not fit in memory, some partitions will be stored on disk.
  - **Memory Only Serialized,**: Same as *Memory Only* levels, except the objects are serialized; therefor, they are space-efficient but more CPU intensive for reading.

  - **Memory and Disk Serialize**: Same as *Memory and Disk Only* levels, except the objects are serialized.
  - **Disk Only**: Store RDD only on disk.

4.2.3 Materializing the factor graph

In previous section we described the strategy for materialization in Dimmwitted, we use the same strategy in Spark as well. First we turn all the data into DataSets with specifically defined class: *variable*, *factors*, *weights*. Then as we explained in Chapter 3, we create *edge* out of factors and variable. After these steps, based on the cost of two strategies, V-CoC and F-CoC, we decide how to materialize our factor graph for better performance. Based on that, we compute QV or QF and cache it on memory to minimize the cost.

One important feature of DataFrame and RDDs in Spark is that they are not mutable. Unlike NUMA, there is no shared memory between Spark worker nodes. Therefor, after materializing, we can not directly update A on the Spark cluster.

After materializing QV or QF in memory, Because A is the the DataSet that we updates repeatedly, we replicate A (or QF in case we choose F-CoC strategy) on every executor, as a broadcast variable. A broadcast variable is a read-only variable therefor we can not write updated values of A, directly back into it. Thus, each executor, using a *MapPartition* transformation[4] iterates through local variables in each node, finds corresponding values of A using a lookup in the broadcast variable and updates the corresponding values in the local copy of A. After iterating over all variables in Q, the driver program collects new values of A into driver program, and re-broadcast them to all executors as the new sampled possible world for computing next sample. This way we guarantee the quality of samples, but collecting and re-broadcasting the data on each step is costly, especially for large data. We can see the pseudo code of core operation in Spark in algorithm 1.

Algorithm 1 Gibbs sampler core operation in Spark

```
1: function SAMPLING($QV, possible\ world_0, iterations$)
2:     $samples \leftarrow DataFrame(possible\ world_0)$
3:     $A \leftarrow broadcast(possible\ world_0)$
4:     for $iteration$ do
5:         for all $v\ in\ QV$ do                                  ⎫
6:             $new\_value = sample(v, mb(v), A)$                  ⎬ $mapPartitions$ transformation
7:             $A.update(v, new\_value)$                           ⎭
8:         $A.collect$
9:         $A \leftarrow broadcast(A)$
10:        $samples.join(A)$
11:        $samples.cache$                                         ▷ cache for next join
12:    return(samples)
```

4.2.4 Page-oriented Layout, Buffer-Replacement Policy

In Spark we simply can not specify the layout for memory pages or buffer-replacement policy on each node. Instead we assume that page layout in Spark is equivalent to

[4]A transformation which passed each partition of a RDD through a function

partitioning layout in Spark; In Spark we assign each Q to a specific RDD partition. Each partition plays the role of a memory page in Spark.

Again buffer-replacement policy equivalent in Spark is the partition-replacement policy, Spark is using Least Recent Used algorithm (LRU) for partition eviction. The twist is that even though we can control partitioning layout, we can not explicitly define which partitions should be evicted and which should not. The solution is to remove unnecessary partition from memory by unpersisting them manually in our implementation and only cache high-complexity DataFrames in memory to minimize the number of partition replacements.

4.3 Conclusion

In this chapter we reviewed our contribution in this thesis. Adding Spark extension for DeepDive in a way that the flexibility of DeepDive is kept and user feels no difference between running an application locally or on a Spark cluster.

We know that Dimmwitted engine is very well designed to have maximum throughput, and we tried to use its techniques in our implementation.

One thing to notice is that in our design, for every SQL query or UDF, one or multiple Spark jobs are submitted to Spark cluster through Spark-connector/HDFS-connector. Thus, for each step of DeepDive application, resources on the cluster should be allocated. This create a constant overhead over the application, but with large datasets, this overhead are insignificant compare to the actual computations.

Chapter 5

Experiments

In this chapter we validate that integrating DeepDive with Spark, actually improves the performance. Then, we compare Dimmwitted engine with our Gibbs sampler on Spark to see how much we can get close to Dimmwitted engine in terms of throughput and accuracy.

5.1 Experiment setup

To correctly measure the efficiency of our contribution, first we have to have a single DeepDive application to run on both NUMA machine (the default version of DeepDive) and on Spark. We designed a rather simple KBC system for finding birth place of every person mentioned in a given dataset of articles. This KBC takes simple text articles and has following steps:

1. Apply natural language processing on each article and add NLP tags to each word.
2. Find all mentions of all places, persons and their nationalities.
3. Link nationalities to countries as possible birth place.
4. Filter sentences containing a person mention and a place mention (or nationality mention).
5. Extract relation candidates between a person and a place pair.
6. Label each candidate based on some simple rules. For example if the verb "born in" in a sentence is between a person mention and place mention, that relationship is labeled as true.
7. Distant supervision, which find sentences with true relationships and use them as training dataset.
8. Extract features of sentences containing pairs of person and place mentions.
9. Perform learning and inference on the graph created by all the possible relations (as variables), extracted feature (as factors) and simple inference rules (e.g. if a person was born in a place, then he could not be born in another)

Datasets: For our experiments, we use a dataset with around 600,000 articles from Wikipedia's biography portal. We use different portions of this dataset for efficiency experiments. Additionally we use a small knowledge base of names of countries and their adjective forms for entity liking.

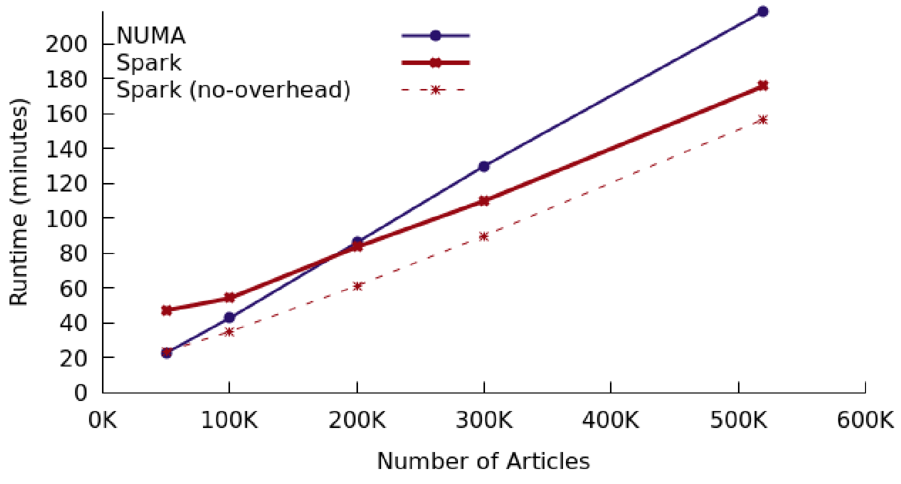


FIGURE 5.1: DeepDive's run-time (without inference) on NUMA and Spark with 8 parallel processes/executors

Metrics: We use two metrics for experiments: (1) runtime of the DeepDive application for comparing the performances. (2) the F1 score of final result of Dimmwitted engine and our inference engine on Spark to compare the quality of both Gibbs samplers.

Computational systems: We execute DeepDive on a local machine with 4 NUMA nodes (each with 6 CPUs and 16Gb of memory) and a standalone *postgreSQL* server. Our Spark cluster consists of 21 nodes (with 168 virtual cores and 220Gb memory in total) which is running on Spark v2.1.0 on YARN v2.6.0.

5.2 Efficiency

5.2.1 Time Efficiency

In the first set of experiments we measure how much Spark leverages the performance of DeepDive on similar systems. We run our application with different datasets, and configure DeepDive to run on 8 parallel process (and 8 executors for Spark). First we measure run-times of DeepDive without "learning and inference" step.

As we see in figure 5.1, DeepDive on NUMA nodes performs much better for small datasets, because executing functions and queries on the local database is instant and almost without any overhead. By increasing the volume of datasets, DeepDive on Spark shows better performance and throughput. User defined functions which are the important part of the system have better performance on Spark. Also it is interesting to notice Spark overhead. As we explained before the overhead of Spark is due to allocating and releasing resources cluster and it is independent from dataset volume, therefore, it is a constant value for each DeepDive application as it is clear in figure 5.1. By knowing the number steps of a DeepDive application we estimate this value: for n steps in an application we have

$$overhead \approx n \times 80 \; seconds.$$

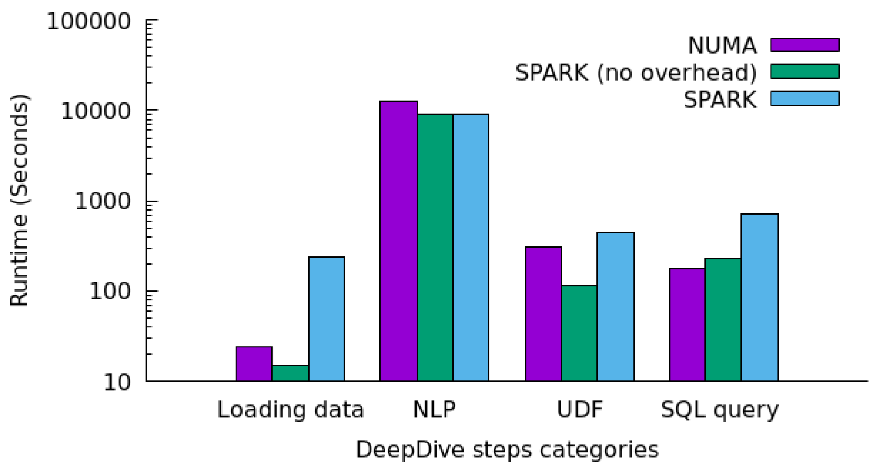


FIGURE 5.2: DeepDive's run-time separated by steps for 500K articles

Stored data volumes			
Dataset (#articles)	25K	50K	100K
PostgreSQL (Mb)	2580	4640	9010
Parquet + Snappy (Mb)	250	420	800

TABLE 5.1: Stored data sizes on PostgreSQL and HDFS in megabytes.

We break down the total runtime of the DeepDive application into 4 categories: loading data into database (or HDFS), natural language processing, user-defined functions and SQL queries. In figure 5.2 we see that Spark (without its overhead) is faster than "NUMA + PostgreSQL" in all external functions. As we see in figure 5.2, natural language processing is the most time consuming step in the KBC and it is much faster on Spark. The difference between other steps are not really significant compare to NLP step. In Spark, the data is stored on HDFS and has to be loaded into a RDD or DataFrame before applying a query using Spark-SQL. Thus, SQL queries are measured faster on a local database. It is needless to mention that Spark is a highly scalable framework. With more executors for our application, the performance improves drastically. For example in table **??** we can see the effect of the number of executors on runtime:

5.2.2 Space Efficiency

One interesting advantage of Spark is parquet format and snappy compression. The data can be compressed up to 10 times on disk when we use parquet format, we see the difference between the volume of stored data on the PostgreSQL and on HDFS in table 5.1. Space efficiency is not data scientists main concern, but when it comes to storing terabytes of data it is an issue to consider.

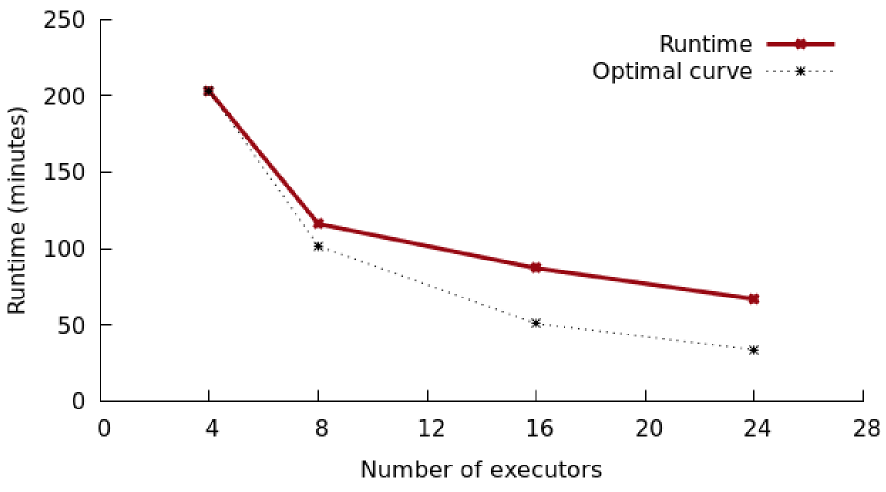


FIGURE 5.3: Scalability with different number executors.: DeepDive's run-time on 200k articles

5.2.3 Scalability

We compared the performance of DeepDive on NUMA and on Spark on similar setups in previous section. Here we check Spark scalability by adding more executors to the cluster. We use a dataset with 200,000 articles and measure runtime of DeepDive (without inference). As we see in figure 5.3, the speedup of of Spark is not too far from optimal curve. The optimal curve represents optimal speedup by parallelization: Doubling number of executors leads to double speedup ($2 \times n_{executors} \longrightarrow 2 \times speedup$) At the beginning, because the dataset is large for 4 and 8 executors, by doubling the number of executors, the speedup is nearly doubled as well. The effect of overhead is negligible for large datasets. By adding more executors, each executor has less computations to do (because the dataset is distributed among executors), instead the overhead shows itself: the velocity of slope reduces to zero slope and the total runtime equals to the constant overhead.

5.3 Gibbs sampler

We explain in chapter 4 that because there's no shared memory in Spark, we collect and re-broadcast new values of a sampled possible world for generating next samples. This is a costly operation and there is no workaround to this limitation in Spark. As we can see in table 5.2 the performance of Dimmwitted is better since Dimmwitted directly manages memory. Plus, in NUMA architecture processors have access to a shared memory, so there is no need for transmitting updated values of A to each node unlike Spark.

Although the performance of Dimmwitted surpassed our Gibbs sampler, the quality of our implementation was comparable to Dimmwitted engine. We used F1-score to measure the accuracy of our sampler:

$$F_1 = 2 \cdot \frac{percision \cdot recall}{precision + recall}.$$

	10K	25K	50K	100K
#variables	22865	47652	96009	190607
#factors	13769	2539555	5118857	10176100
Dimmwitted runtime (sec)	23	44	84	170
Spark sampler runtime (sec)	330	510	945	2140

TABLE 5.2: Number of variables and factors, and runtime of Dimmwitted engine and implemented Gibbs sampler on Spark for 1000 samples.

	Recall	Precision	F1-score
Dimmwitted	0.62	0.93	0.74
Spark sampler	0.58	0.9	0.70

TABLE 5.3: F1-score for a 100 random articles, with 1000 samples.

As we explained in chapter 4, since we don't have a write access to a shared memory among executors, we can not update the values of variable. Updates are local during each step and only at the end of a step, new value are broadcasted to all executors. Therefore the as we can see in table 5.3, convergence rate of our sampler is slower than Dimmwitted engine. On the other hand, Dimmwitted writes updates instantly on the shared memory so all NUMA nodes have access to the most updated value, thus it has better convergence rate.

Chapter 6

Conclusion and Future Work

In this thesis, we integrate DeepDive: a framework for knowledge base constructions, with Apache Spark: a general purpose data-processing framework. We redesign parts of DeepDive's structure to support Spark on all steps, while preserving DeepDive's flexibilities and features. A user can run the same DeepDive application which is written for a local machine, on Spark with only changing the configurations of the application. Finally even though DeepDive is very well designed, we manage to improve the performance of DeepDive using the parallelization power of Spark. This improvement may not be significant for small datasets, but it is considerable for larger ones. Ideally A user is should develop a DeepDive application, test and debug it locally, and afterwards use the same application to process large datasets with Spark. Of course Dimmwitted engine is one of the fastest Gibbs samplers because of its low-level design, and our Gibbs sampler on Spark was not as fast as Dimmwitted on NUMA. Therefore, the best performance is obtained by using Spark for all the steps, except learning and inference phase, and then pass grounded factor graph to Dimmwitted engine for inference. With this technique we achieve 20% of improvement for overall performance.

6.1 Future Work

In the future work, we continue to reduce the overhead over Spark applications. By changing DeepDive's compiler we might combine many of DeepDive underlying commands as one Spark application to reduce the overhead. Although this may need to change the compiler and fabric of DeepDive itself.

Another goal can adding the feature of generating a packed Spark Application corresponding to a whole KBC system. So that user can run the whole DeepDive application without any overhead on Spark. This task needs to define a new compiler for DeepDive to combine and convert all the executables and queries into one Spark Application.

6.2 Lessons Learned

Overall, integration of DeepDive and Spark is a good practice for running a KBC system on large amounts of data, except for inference engine. Gibbs sampling needs a huge number of writes and updates in memory and since there is no shared memory in Spark, it is not feasible to implement a Gibbs sampler faster that Dimmwitted on Spark. Spark is not designed for such computations. It was not beneficial to implement a Gibbs on Spark. Instead, we could go deeper on integration of data processing phases and reduce the overheads more. The other option might be changing DeepDive's model for inference, instead of using factor graph and Gibbs sampler maybe we can use a model more adaptable with Spark, or the models which are already implemented in *MLlib*[1] of Spark.

[1]Machine Learning Library of Spark

Bibliography

[1] Laszlo A. Belady. "A study of replacement algorithms for a virtual-storage computer". In: *IBM Systems journal* 5.2 (1966), pp. 78–101.

[2] Cloudera. *Benchmarking Apache Parquet: The Allstate Experience*. 2016. URL: `http://blog.cloudera.com/blog/2016/04/benchmarking-apache-parquet-the-allstate-experience/` (visited on 07/24/2017).

[3] databricks. *Stanford CoreNLP wrapper for Apache Spark*. 2016. URL: `https://github.com/databricks/spark-corenlp` (visited on 07/24/2017).

[4] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning*. The MIT Press, 2009. ISBN: 0262013193, 9780262013192.

[5] Frank R Kschischang, Brendan J Frey, and H-A Loeliger. "Factor graphs and the sum-product algorithm". In: *IEEE Transactions on information theory* 47.2 (2001), pp. 498–519.

[6] David Lunn et al. "The BUGS project: evolution, critique and future directions". In: *Statistics in medicine* 28.25 (2009), pp. 3049–3067.

[7] Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 2014.

[8] Jorge R Sobehart et al. "Moody's public firm risk model: A hybrid approach to modeling short term default risk". In: *Moody's Investors Service, Global Credit Research, Rating Methodology, March* (2000).

[9] Apache Spark. *Apache Spark Documentation: Cluster Mode Overview*. 2016. URL: `https://spark.apache.org/docs/latest/cluster-overview.html` (visited on 07/24/2017).

[10] Dan Suciu et al. "Probabilistic databases". In: *Synthesis Lectures on Data Management* 3.2 (2011), pp. 1–180.

[11] Martin J Wainwright, Michael I Jordan, et al. "Graphical models, exponential families, and variational inference". In: *Foundations and Trends® in Machine Learning* 1.1–2 (2008), pp. 1–305.

[12] Matei Zaharia et al. "Fast and interactive analytics over Hadoop data with Spark". In: *USENIX Login* 37.4 (2012), pp. 45–51.

[13] Matei Zaharia et al. "Spark: Cluster computing with working sets." In: *HotCloud* 10.10-10 (2010), p. 95.

[14] Ce Zhang and Christopher Ré. "Towards high-throughput Gibbs sampling at scale: A study across storage managers". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM. 2013, pp. 397–408.