Master Project

Nested data types for the Sempala Property Table

Albert-Ludwigs-Universität Freiburg im Breisgau

31.03.2017

Polina Koleva & Matteo Cossu

Table of Contents

1. Introduction

This project stands in the domain of the Sempala project[1]. Sempala is a software consisting of two parts: a component that loads RDF data into a database stored in HDFS and a such that translates SPARQL queries into the Impala SQL or Spark SQL[2]. Different data models are used as a way to store RDF triples, the focus of our work was built on the Property Table concept.
The Property Table is a storage schema for RDF data that, for some type of queries, can be very effective. The Property Table was already implemented in Sempala. However, the lack of nested data structures as columns values in the previous versions of Impala leads to huge tables and a significant overhead. Therefore, we implemented a new version of the software to support the new features of Impala, that allow to store multivalued properties in a single row.
The new variant greatly reduces the size of the table and achieves good performance in terms of query execution.
We implemented two version of the translating component with the same logic but using two distinct technologies: Spark and Impala. And thanks to this choice, we could evaluate the Complex Property Table model more independently from the underlying implementations.
In this paper we will introduce the concept of Complex Property Table and we will explain how we implemented it. Moreover, we will show the evaluation results of different queries and datasets and we will try to give them some interpretation. Lastly, we will talk about the limitations of the technologies we used and how the system could be improved in the future.

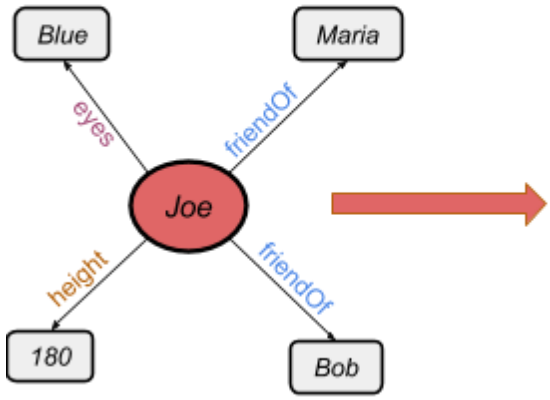
2. Complex Property Table Concept

The **Property Table** is a data scheme for storing RDF graphs. The table contains a row for every resource which occurs at least once as a subject in a triple. The other columns are all the possible predicates, that we will refer also as ‘properties’. A value contained in a column represents the object of a triple which has the predicate specified by the column name.
Each property can be *simple* or *complex*. A property is *simple* when for every possible subject there is at most only one possible value. A property instead is *complex* when at least two different values exist for a single subject.

In the previous versions of Sempala when a complex property was present, the problem of multiple values was solved by replicating the row for every possible value. With our work we

[1] "Sempala: Interactive SPARQL Query Processing on Hadoop ...." Accessed April 2, 2017. http://link.springer.com/chapter/10.1007/978-3-319-11964-9_11.
[2] The previous version the Sempala Property Table was implemented only in Impala

introduce the **Complex Property Table**, that follows the same concept as before but where the duplications are avoided.
The multiple values of a complex property are stored in a single cell using nested data structures, in particular Arrays. The following figure illustrates an example of how an input graph is stored, using the old version of Property Table and the new model.



Property Table

s	eyes	friendOf	height
Joe	Blue	Maria	180
Joe	Blue	Bob	180

Complex Property Table

s	eyes	friendOf	height
Joe	Blue	[Maria,Bob]	180

The left graph in the figure represents a subset of a rdf graph, to the right instead we can see how the same input would be translated into the normal Property Table and into the Complex one. As we can see *friendOf* is a complex property: it generates a duplicate in the Property Table but it is translated into a single row by using the nested data type.

Advantages of the Complex Property Table

The main difference between the Complex Property Table and the old version is the absence of duplication. Therefore, the number of rows is drastically decreased.
This fact has relevant positive consequences:

- The actual stored size of the table is remarkably smaller. In many cases, it could become possible to keep the entire table in memory during execution.
- The reduction of rows consequently decreases the complexity of joins.
- Filtering or joining on simple properties are faster because there is no need to scan through all the duplicates, generated by properties that are not involved in the operations.

The only drawback can be that, in case of queries that contain only complex properties, the performance should be the same or slightly worse than the simple property table model. The necessity of unnesting these properties should theoretically introduce a small overhead.

3. Implementation

As we already anticipated in the introduction, the implementation of the system consists of two parts: the loader and the translator.
The loader's job is to create the Complex Property Table from a rdf graph given as input. The translator instead converts SPARQL queries into Impala SQL or Spark SQL. The queries translation can be executed on these two supported systems to retrieve the requested results.

Input Data

Initially, the input data is provided as a RDF file which contains a list of triples. Each triple consists of a subject, a predicate and an object and it represents a relation between a subject and an object. The file is parsed down to triples and a new table ***triple_table*** is derived from them. There exists a row in the table for each rdf triple. The schema of the table is as follow:

triple_table	
s	*STRING*
p	*STRING*
o	*STRING*

A subject of a triple is stored in the column ***s,*** a predicate and an object in column ***p*** and column ***o*** respectively. For example, for the triple subject - predicate - object <Jonh> - <isFriendOf> - <James>, new tuple is saved in the table where column ***s*** is "Jonh", column ***p*** is "isFriendOf" and column ***o*** - "James". The proceeding analysis is done over the created ***triple_table*** and the RDF file will not be used anymore.

Data preparation

As we already mentioned in Chapter 2, we differentiate two types of properties. To repeat, each property matches to a unique predicate. So the number of properties is the same as the number of unique predicates in the input RDF file as well as the unique values in column ***p*** of ***triple_table.*** If a property is simple, it means that for each subject there is only one triple which contains this predicate. Or in other words, for each subject there is only one tuple in ***triple_table*** where ***p*** column is equal to this property. On the other hand, if a property is complex, it means that there is at least one subject for which there are at least two triples with contain this predicate and for at least one subject there are at least two tuples where p column is equal to this property.

The information that consists of which of the properties are complex and which are simple is really valuable, not only during the loading process but also for the translation of queries afterwards, so we need to collect and store it beforehand. Therefore, we build a table ***properties*** which contains one row for each property and its type. The table consists of two columns ***p****:STRING* and ***is_complex****: Boolean*. While the former stores the name of each property, the latter represents its type. The schema of the table follows:

properties	
p	*STRING*
is_complex	*BOOLEAN*

If *is_complex* is 1 it means that a property is of a complex type. For example, if we have a subject which has three outgoing edges with a predicate *isFriendOf*, its *is_complex* value is 1. On the other hand, *is_complex* is 0 if there is no subject which has more than one outgoing edge labelled with this property.
To collect the information needed for ***properties*** table, we execute two queries over the ***triple_table***. Firstly, all complex properties are collected.Initially, we group rows by the pair (s, p) and for each group we take the number of rows in it. If it is greater than 1, then the predicate exists more than once for the same subject, which is enough to call it complex. So, in such a way we can filter all complex properties.
The following code presents the full query used:

```
DataFrame multivaluedProperties = this.hiveContext.sql("
SELECT DISTINCT(p) AS p FROM
(SELECT s, p, COUNT(*) AS rc FROM triple_table GROUP BY s, p
HAVING rc > 1) AS grouped ")
```

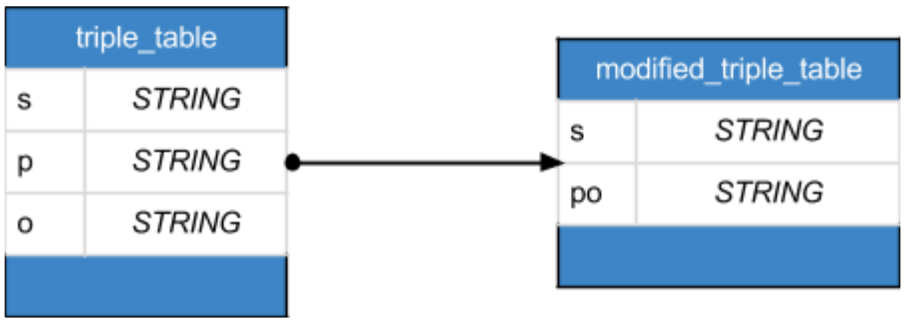
Secondly, to collect simple properties, we just select all properties and remove the complex ones from them - subtraction of complex properties from a set with all properties.

```
DataFrame singledValueProperties =
allProperties.except(multivaluedProperties);
```

The ***properties*** table is not deleted after loading finishes because it will be used during translation process.

Loading

The loading starts when information about different properties and their type has been already extracted. The process of loading the *Complex Property Table* mainly contains execution of one user-defined aggregation function over the ***modified_triple_table.***
Because aggregator functions can operate only over one column at the time, we need to concatenate a predicate column ***p*** and object column ***o*** of the ***triple_table***. Therefore, we build a new table with two columns ***s:STRING*** and **po:STRING**. The former column stores a subject while the latter concatenation of its predicate and object. So for each tuple in ***triple_table*** we produce one corresponding tuple in ***modified_triple_table***.

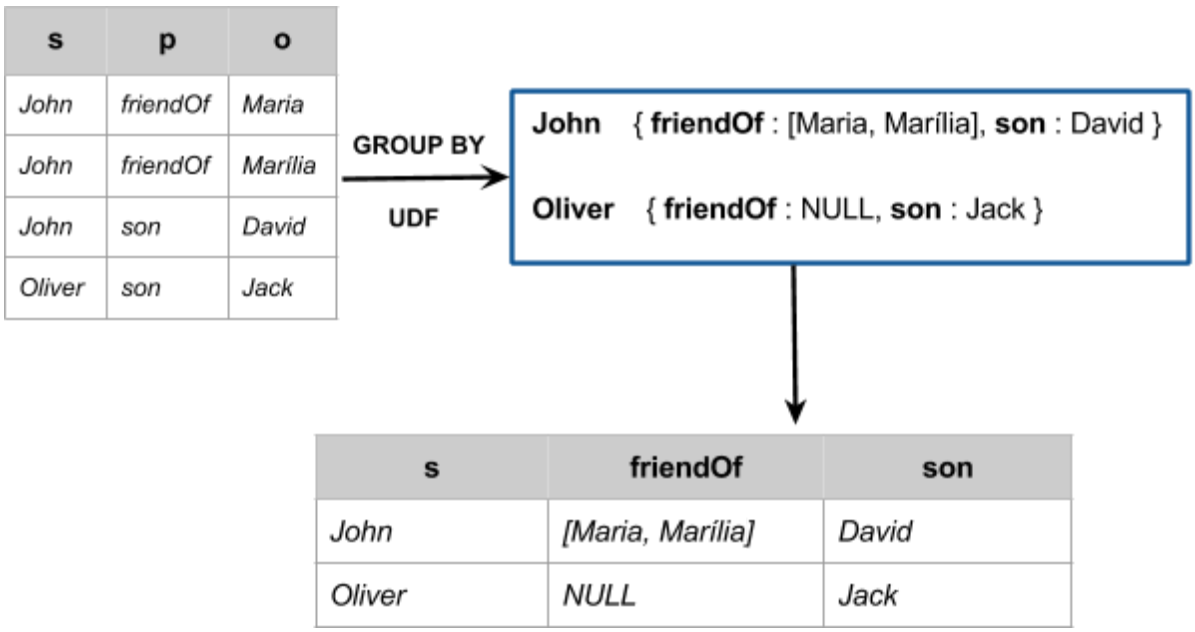


To concatenate the **po** string we use a custom delimiter string that will be used afterwards to distinguish again the two columns. Before applying the aggregator function we group our tuples by subject (***s*** column). So our aggregator function is applied over groups of tuples all with the same subject.

The User-defined aggregate function (*PropertiesAggregateFunction* class) is applied over ***modified_triple_table.*** It operates over groups where we have a group per subject. Besides, it has information about all properties of the triple table. In our case it contains a list of all properties extracted from the ***properties*** table. Their order is important because after the aggregation a list of properties for each subject is returned and their order is the same as the global list of properties which is set initially.
For instance, one group for subject “subject1” can be:

Predicate	***Object***
is_friend_of	*Adam*
name	*Eric*
is_friend_of	*Adele*
is_friend_of	*Gabi*

As a result of the function for this subject we will have two lists - one which contains <Adam, Adele, Gabi> and another one only with <Eric>. At the moment, the UDF will return a list of lists for each subject. The order of these lists depends on the already defined order that we pass as an input parameter to our function. For example, for each subject the value for its ***name*** predicate will be on position 5 in its corresponding list in the result set. If such predicate does not exists, this element will be NULL.
The following figure illustrates in general the loading process:



We described all the operations that are performed during the aggregation phase, and we still need to explain how we implemented them using Java and Spark. The class that contains the user-defined aggregate function needs to implement the methods required from the Spark interface(*UserDefinedAggregateFunction)*. The types have to be specified using wrappers around the Scala original types provided from the framework. Spark works internally with Scala and considers the implementation of the aggregate function like a black-box.
The methods that need to be specified are:

- The structure of the data input (*inputSchema*) .
- The structure of the internal buffer (*bufferSchema)* .
- The type of the values returned from the aggregate function (*dataType)*.
- The method that is called over every row in the same group, that save the partial results in the internal buffer (*update)*.
- The method that is called by Spark when the rows of a single group are split in two parts and their buffers need to be merged (*merge)*.
- The method that is called over every group, after the aggregation is finished, and that produces the result value (*evaluate)*.

In our case the information of every group is stored in the internal buffer as Hash Map. For each row of the same group we separate the property from the object and we add this into data the map, using the property as key and the object as value.
When the aggregation is finished, we produce for each subject an array of arrays containing all the values for all the properties.

After we have explained in details how we produce the necessary data, we still need to create the definitive table. The final step over the intermediate results is to extract the values and store them into a final complex*_property_table*. So for each subject, we iterate over its lists which represent values for different properties. If a property at index *n* is complex, we just extract the list at this position and store it as a list. Otherwise, if a property is simple, we extract the first element of the list and store it in a single valued column. If we take the example above, for our subject we have two predicates. Let us assume that in total we have 4 different predicates in the initial data set. So for each subject we will receive as a result a list with 4 elements.
Moreover, let us assume that the order of our predicate is *<name, age, gender, is_friend_of*>. We have already seen that for our subject there is no information for age and gender so the returned list for it will be:

```
<<Eric>, <>, <>, <Adam, Adele, Gabi>>
```

Because we have information which predicate is complex and which is simple for the *name* predicate we will extract the value and store it into singled value column for the predicate *name*. Therefore, after the transformation the result for a subject will be:

```
<Eric, NULL, NULL, <Adam, Adele, Gabi>>.
```

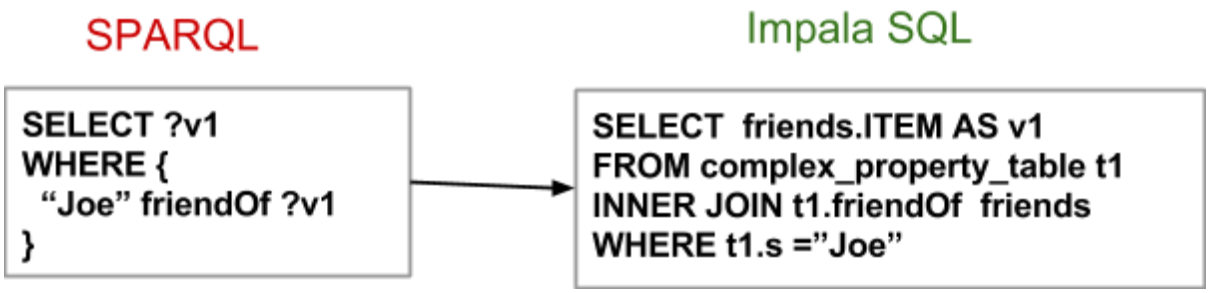
Translating

The core functionality of the translator is to convert SPARQL into valid Impala or Spark queries. This consists of a straightforward operation that translates the string syntax without altering the semantic. However, the Complex Property Table contains two different types of columns, and we need to treat them differently. Therefore, before starting the real translation, we must fetch the list of all properties and their type from the table called **properties,** stored in the HIVE metastore. After having which properties are complex and which simple, for both Impala and Spark the logic of translating the query is identical. However, they differ slightly in syntax, especially regarding the complex types.
Regarding the code implementation, we have adjusted the already existing logical structure by adding a ComplexSelect type and ComplexTripleGroup. The latter wraps the translation of triples which contain complex property, while the former the exact translation of a select statement when it includes complex column. Moreover, we have added a singleton component

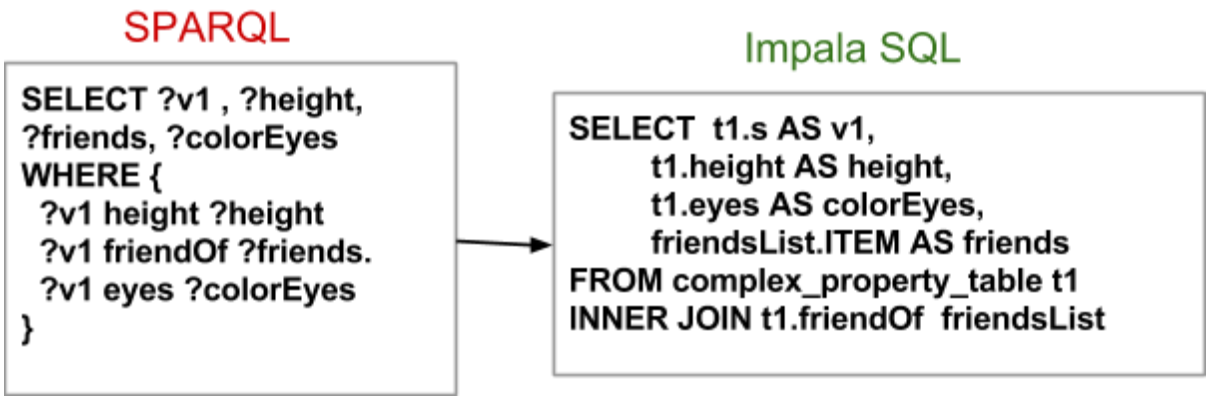
ComplexPropertyTableColumns which extracts which properties are complex and which are simple.

Impala

Impala can not select directly complex properties and it can perform joins directly only on simple attributes. Before a join on a complex type is performed, we must unfold the values contained in that column.
Impala's way to fetch values contained in complex columns is to use inner self joins.
The following figure shows an example for a simple selection:



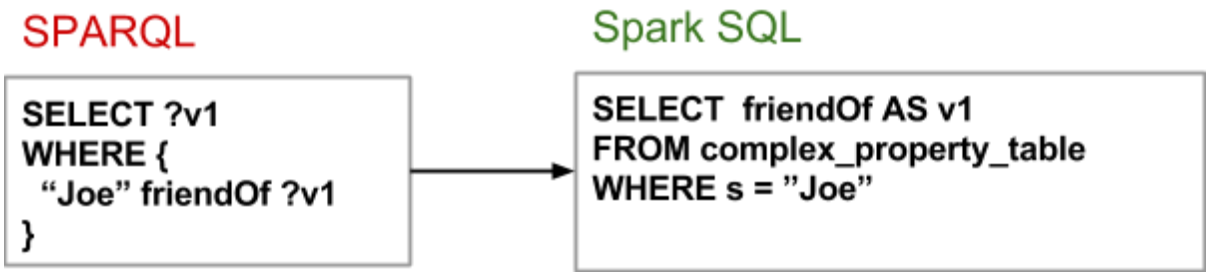
The column corresponding to the complex property is treated as another table and the values contained are selected with the *.ITEM* notation.
A limitation of Impala is that it is not allowed to check if a complex column is equal to *NULL*[3]. This problem is avoided by the usage of inner joins that simply do not produce any row if the column is empty.
The Complex Property Table model shows great performances on star queries, where many properties are selected for the same subject.
The following figure demonstrates, using an example, how star queries are translated in Impala SQL:



[3] using NULL instead of empty arrays saves considerable amount of space in the property table.

Spark

Spark as one of the most popular general engine for large-scale data processing provides also a feature for execution of SQL queries. Similarly to Impala, it supports multivalued columns but it can retrieve them without need of unnesting. Moreover, a function array_contains(<array>,<value>) which checks if an array contains a specific value is available. Therefore, the process of translating SPARQL queries to Spark SQL is really straightforward. Unfortunately, a join on multivalued column is not possible natively and unnesting of the arrays is a necessity when such is presented. The following figure shows an example for a simple query with a selection of a complex property:



Spark does not have the same limitation as Impala for NULL checks. It can be checked if a column is NULL no matter if it is a multivalued or single valued. It retrieves the simple and complex columns equivalently, the only difference is when a search for particular value is done. When it is over a complex column - array_contains function is used; otherwise an equal sign.
Because a join on complex column is not automatically supported, but it is an important aspect of the project, the following star query shows how it will be implemented:

SPARQL

```
SELECT  ?v0 ?v2 ?v3
WHERE
 {
   ?v0 subscribes Site164 .
   ?v2 caption ?v3 .
   ?v0 likes ?v2
 }
```

Spark SQL

```
SELECT BGP1.v0 AS v0, BGP1.v2 AS v2,
BGP1.v3 AS v3
FROM (
      SELECT BGP1_0.v0 AS v0, BGP1_1.v2
AS v2, BGP1_1.v3 AS v3
FROM (
       SELECT s AS v2, caption AS v3
       FROM complex_property_table
       WHERE s IS NOT NULL
       AND caption IS NOT NULL
  ) BGP1_1 JOIN (
      SELECT s AS v0, lve_1_likes AS v2
      FROM complex_property_table
      LATERAL VIEW
      EXPLODE(complex_property_table.like
      s) lve_1 AS lve_1_likes
      WHERE s IS NOT NULL
      AND subscribes IS NOT NULL
      AND array_contains(subscribes,
'Site164')
AND likes IS NOT NULL) BGP1_0
ON(BGP1_1.v2=BGP1_0.v2) ) BGP1 )
```

From the initial query, two TripleGroups by a subject are formed . One for the subject <v0> and another for <v2>. For each triple group a select statement is constructed. The interesting select statement in this particular query is this for <v0>:

Spark SQL

```
SELECT s AS v0, lve_1_likes AS v2 FROM complex_property_table
LATERAL VIEW EXPLODE(complex_property_table.likes) lve_1 AS
lve_1_likes
WHERE s IS NOT NULL
AND subscribes IS NOT NULL
AND array_contains(subscribes, 'Site164')
AND likes IS NOT NULL
```

Because <*likes*> is a complex property and we have a join on it, we have to “unnest” it by using LATERAL VIEW EXPLODE Spark function. Moreover, we have to search for a specific value in a complex column <*subscribes*>, that is why array_contains function is used. Because Spark supports NULL/NOT NULL checks of complex and simple columns, for each column contained in the query we have added such a check.

4. Evaluation

In this section we will evaluate the performance of Sempala Complex Property Table. Initially, we will present the environment that we use during evaluation process. Afterwards, we will introduce the variety of test datasets. Finally, we will introduce loading performance and then the execution times of different type of queries on two types of datasets, with several sizes.

Benchmark Environment

The cluster we used for our evaluation consists of 10 machines connected via a Gigabit Ethernet connection. Each machine is equipped with 32GB main memory. The cluster is Cloudera's Distribution for Apache Hadoop Version 5.10.0 containing Apache Spark Version 1.6 and Impala Version 2.8.

Spark uses only 9 machines of the cluster as worker nodes, since it needs one machine as a master node. After extensive testing, we have decided to change only specific configurations of Spark for optimal results. The following are used:

- *spark.driver.memory* = 2g

The default value is 1g. It represents the amount of memory to use for the driver process i.e where SparkContext is initialized.

- *spark.executor.memory* = 16g

The default value is 1g. It represents the amount of memory to use per executor process.

- *spark.sql.inMemoryColumnarStorage.batchSize* = 20 000

The default value is 10 000. It controls the size of batches for columnar caching. Larger batch sizes can improve memory utilization and compression, but risk OOMs when caching data.

- *spark.sql.shuffle.partitions* = 5/25/100/200

The default value is 200. This setting defines the number of partitions used for shuffling data executing join and aggregation operations. We change this setting depending on the query sets and datasets.

Datasets

We use the WatDiv[4] Data provided by the Waterloo SPARQL Diversity Test Suite 7. They developed it to measure how an RDF data management system performs across a wide variety of SPARQL queries with varying structural characteristics. We used the following four datasets:

[4] "Waterloo SPARQL Diversity Test Suite (WatDiv) v0.6." Accessed April 2, 2017. http://dsg.uwaterloo.ca/watdiv/.

- WatDiv10 - contains 1 Million triples
- WatDiv100 - contains 10 Million triples
- WatDiv1000 - contains 100 Million triples
- WatDiv10000 - contains 1 Billion triples

All datasets contain the same number of unique predicates (86), but the number of subjects and objects is different for each of them. The WatDiv entities are generated in such a way that mimics the real types of distributions on the Web.
Moreover, we have tested also with YAGO[5] (Yet Another Great Ontology) dataset. It is a knowledge base project developed jointly by Max Planck Institute for Informatics and the Telecom ParisTech University. The information it contains is automatically extracted from Wikipedia and other sources. It contains more than 10 million entities and more than 120 million facts about these entities.

Queries

WatDiv Basic
The WatDiv Basic query set is provided by Waterloo SPARQL Diversity Test Suite 7. It contains queries of varying shape and selectivity. The queries are grouped into the following subsets:

- **C** (C1,C2,C3) Complex Shaped queries.
- **F** (F1, F2, F3, F4, F5) Snowflake shaped queries.
- **L** (L1, L2, L3, L4, L5) Linear shaped queries.
- **S** (S1, S2, S3, S4, S5, S6, S7) Star shaped queries.

WatDiv Increasing Linear
This set contains path shaped queries. The length of the path increases from 5 (**IL-5**) to 10 (**IL-10**). These queries are also divided in three types: **1** and **2** where the paths begin or end from or to a specific node in the graph, for the type **3** instead all the subjects and objects are variables. Because of the high number of possible paths, the type **3** queries require a significant workload.

Yago Queries
The set for Yago consists of only 15 queries. They have different types and shapes. In particular, **Q9** yields a huge number of results, instead **Q12** and **Q13** need long execution times because they mix long paths and high complexity.

Loading

Loading times are usually not crucial for the system, but since we got significant improvement, we decided to highlight these results.
In the following tables we compare sizes and loading times of some of the datasets:

[5] "YAGO (database) - Wikipedia." Accessed April 2, 2017. https://en.wikipedia.org/wiki/YAGO_(database).

Dataset	Input Size (triple table)	Size Complex Property Table	Loading Time(s)
WatDiv10	49 MB	14.5 MB	62
WatDiv100	507 MB	74.3 MB	120
WatDiv1000	5.2 GB	392 MB	363
WatDiv10000	53.5 GB	4.1 GB	1778
YAGO	15.2 GB	2.1 GB	210

As we can see from the results, we can load even the biggest dataset in less than 30 minutes. Our model drastically reduces the size of the stored table, this can lead to the opportunity of loading the entire Complex Property Table into memory.

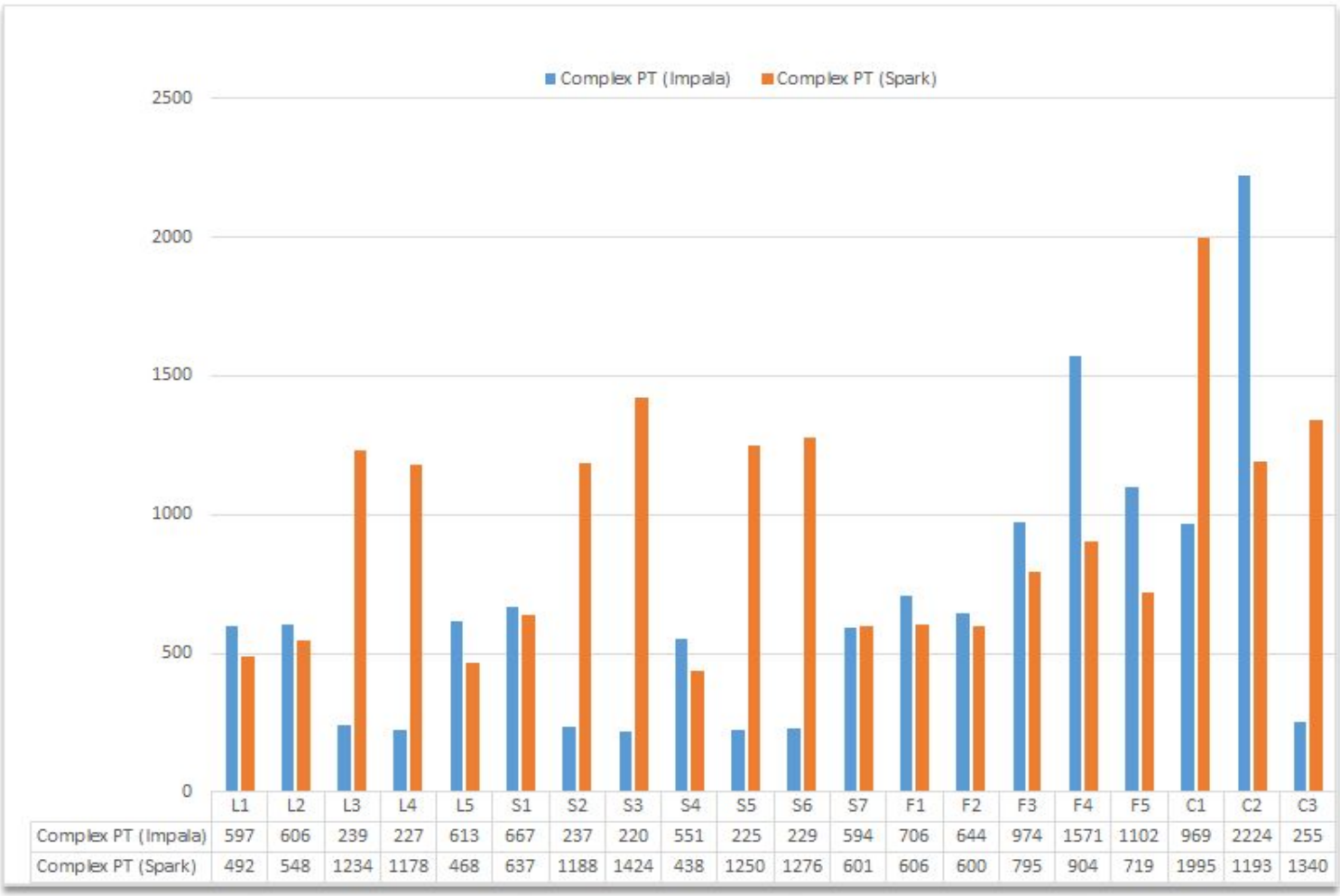
Queries Evaluation

For WatDiv Datasets we use 2 different groups of queries: Basic and Increasing. The difference and their specifications are explained in the section above. Both subset of queries are executed with the four WatDiv datasets. We have only one set of queries for YAGO dataset.
Because we try to achieve optimal execution time for each query, we change the setting spark.sql.shuffle.partitions and the number of partitions for the input dataset depending on the database size and subset of queries. Table below shows an overview over the partition numbers. All configurations values are chosen after tuning Apache Spark and executing queries.

Dataset	Query Set	Partitions
WatDiv10	**Basic**	5
WatDiv10	**Increasing**	25
WatDiv100	**Basic**	5
WatDiv100	**Increasing**	100
WatDiv1000	**Basic**	5
WatDiv1000	**Increasing**	100
WatDiv1000	**Basic**	200
WatDiv1000	**Increasing**	100
YAGO	**-**	100

Basic Queries

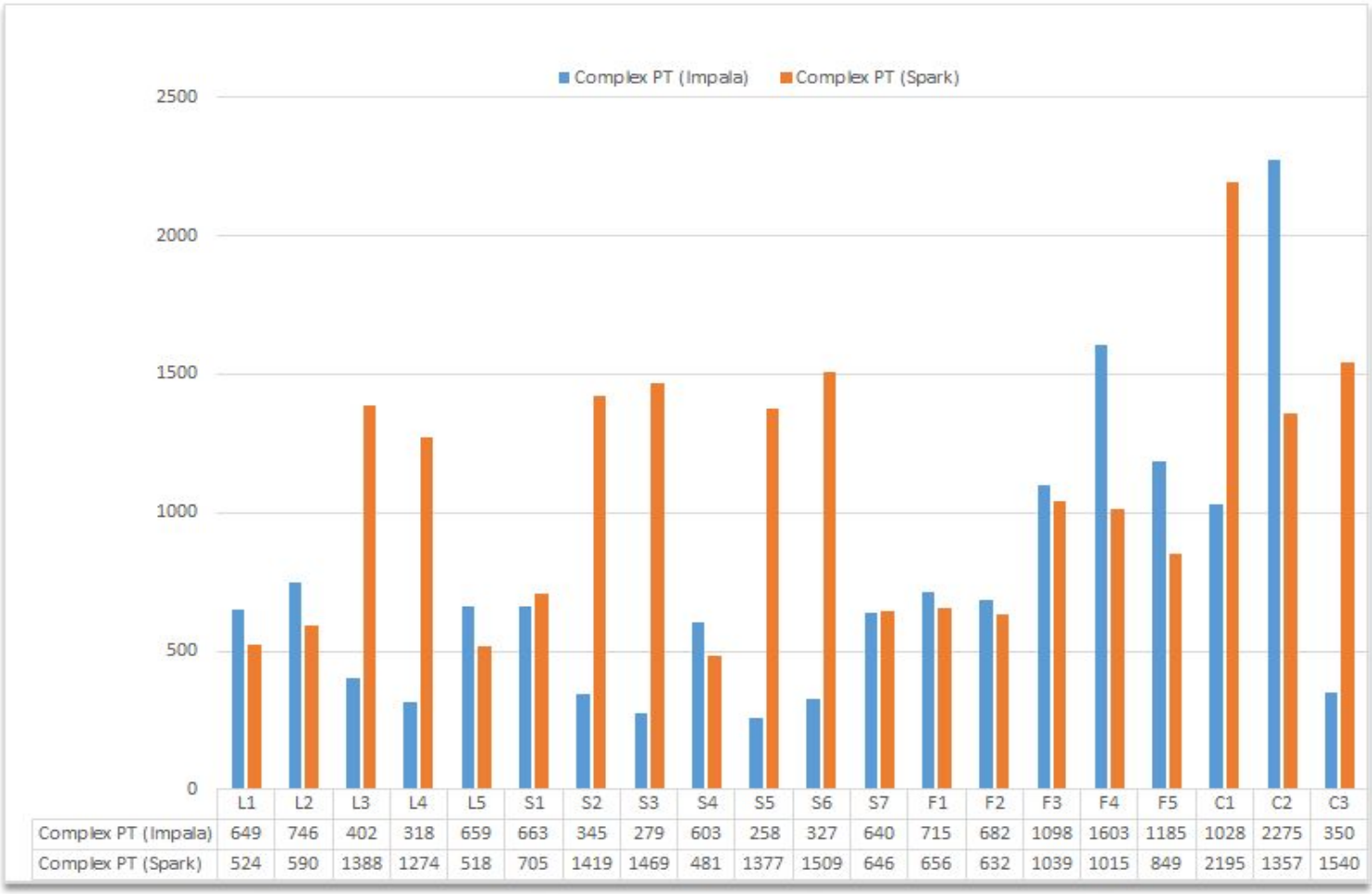
WatDiv 10



	L1	L2	L3	L4	L5	S1	S2	S3	S4	S5	S6	S7	F1	F2	F3	F4	F5	C1	C2	C3
Complex PT (Impala)	597	606	239	227	613	667	237	220	551	225	229	594	706	644	974	1571	1102	969	2224	255
Complex PT (Spark)	492	548	1234	1178	468	637	1188	1424	438	1250	1276	601	606	600	795	904	719	1995	1193	1340

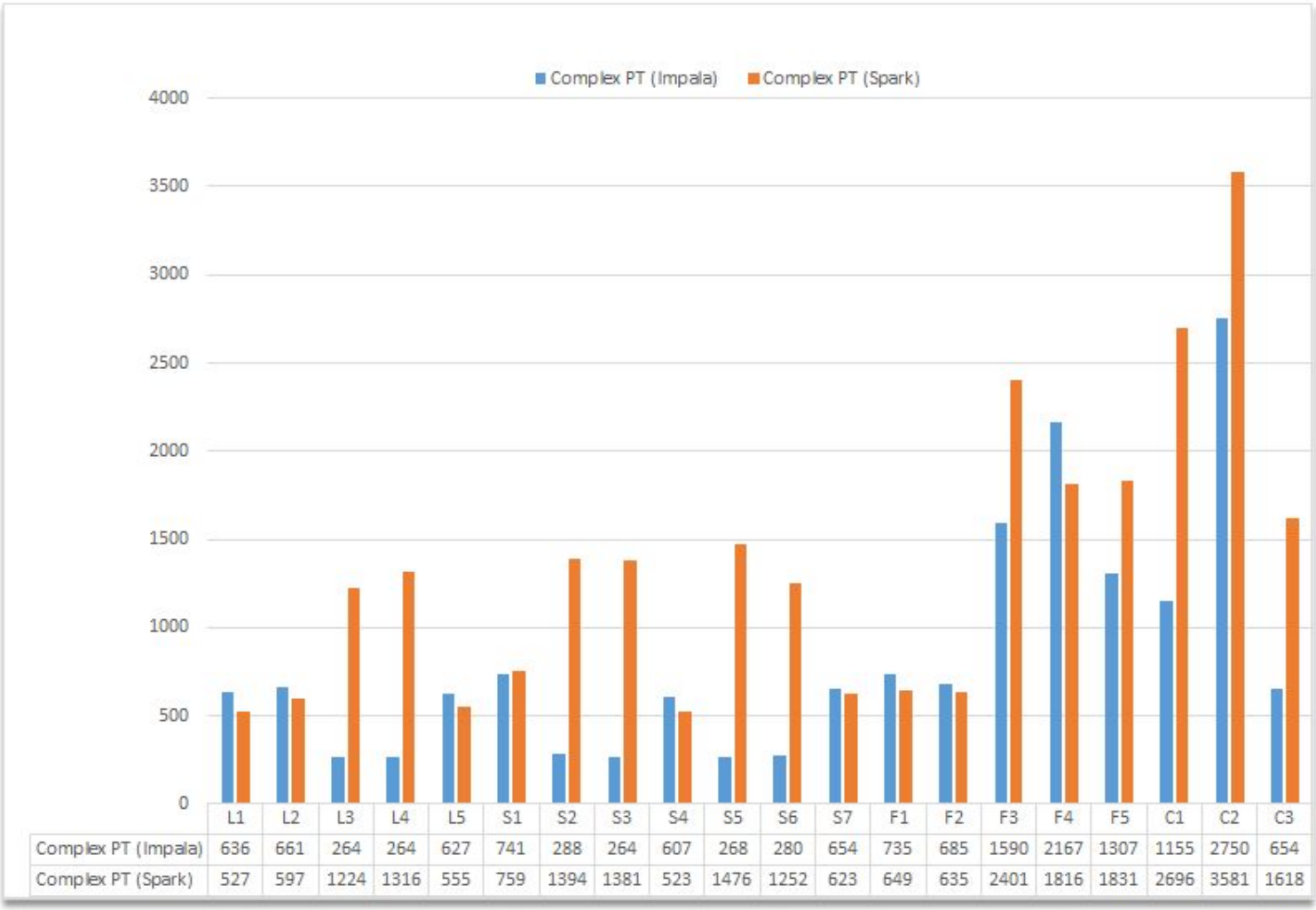
From WatDiv 10 and WatDiv 100 we can see that in most of the star shaped and linear queries Impala outperforms Spark.
Snowflakes and complex queries show similar behaviour for both systems with some exception in favor of Spark.
These datasets are probably too small to be very significant in the evaluation of the model, but they can be used to compare the two different systems used.
Impala can benefit from a more intelligent Join reordering, but it has no clear advantage on some queries where the location of the data is more important.

WatDiv 100



	L1	L2	L3	L4	L5	S1	S2	S3	S4	S5	S6	S7	F1	F2	F3	F4	F5	C1	C2	C3
Complex PT (Impala)	649	746	402	318	659	663	345	279	603	258	327	640	715	682	1098	1603	1185	1028	2275	350
Complex PT (Spark)	524	590	1388	1274	518	705	1419	1469	481	1377	1509	646	656	632	1039	1015	849	2195	1357	1540

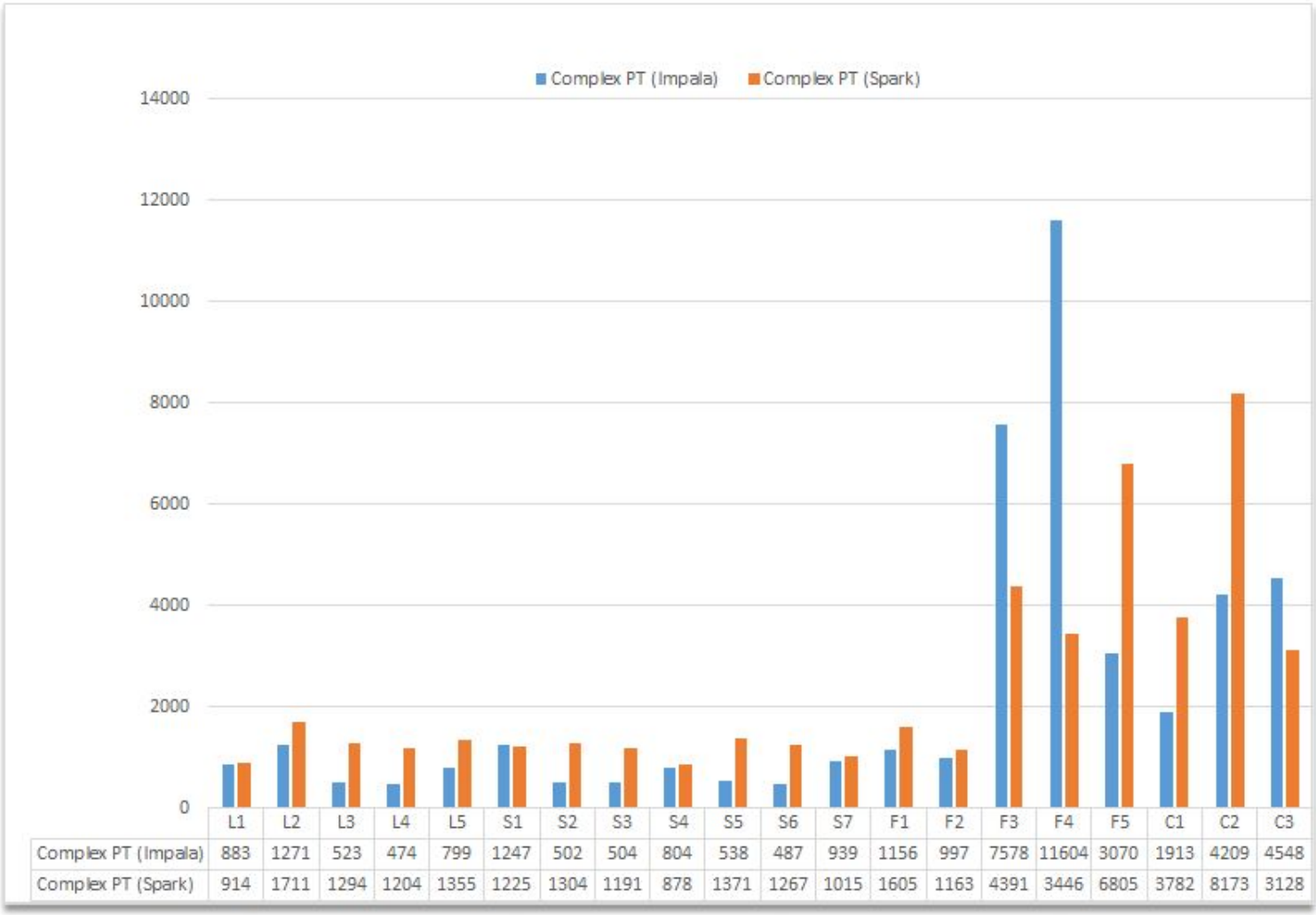
WatDiv 1000



	L1	L2	L3	L4	L5	S1	S2	S3	S4	S5	S6	S7	F1	F2	F3	F4	F5	C1	C2	C3
Complex PT (Impala)	636	661	264	264	627	741	288	264	607	268	280	654	735	685	1590	2167	1307	1155	2750	654
Complex PT (Spark)	527	597	1224	1316	555	759	1394	1381	523	1476	1252	623	649	635	2401	1816	1831	2696	3581	1618

WatDiv 10000

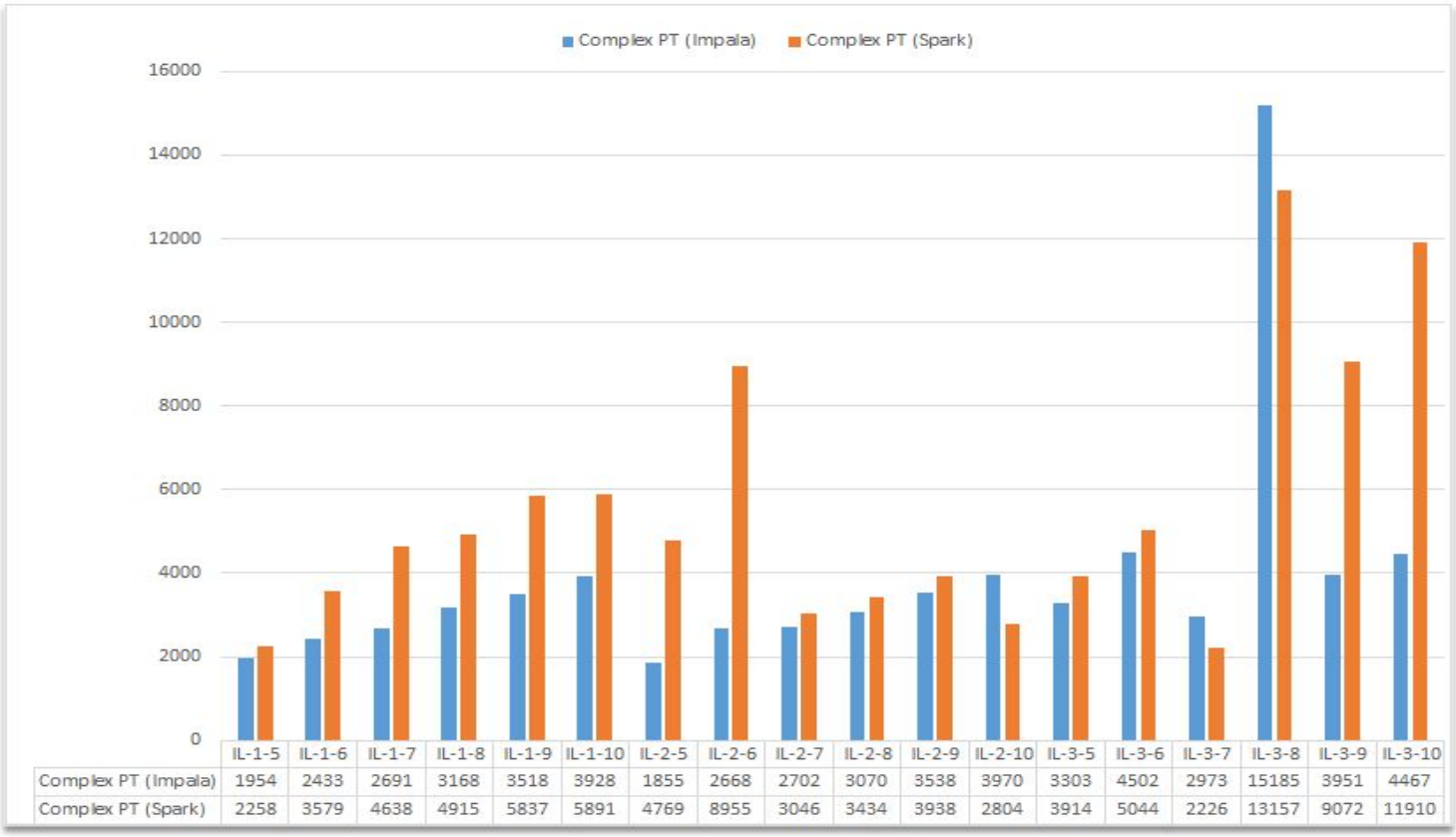
WatDiv 10000 is the biggest dataset and maybe the real protagonist of the evaluation.
These results confirm approximately the same trend as the previous datasets. Impala behaves better on average but shows really bad performances for some queries, in particular **F3** and **F4.** We believe that in some cases the Impala optimizer fails to find a good join type and order, especially when there are joins on complex properties.



	L1	L2	L3	L4	L5	S1	S2	S3	S4	S5	S6	S7	F1	F2	F3	F4	F5	C1	C2	C3
Complex PT (Impala)	883	1271	523	474	799	1247	502	504	804	538	487	939	1156	997	7578	11604	3070	1913	4209	4548
Complex PT (Spark)	914	1711	1294	1204	1355	1225	1304	1191	878	1371	1267	1015	1605	1163	4391	3446	6805	3782	8173	3128

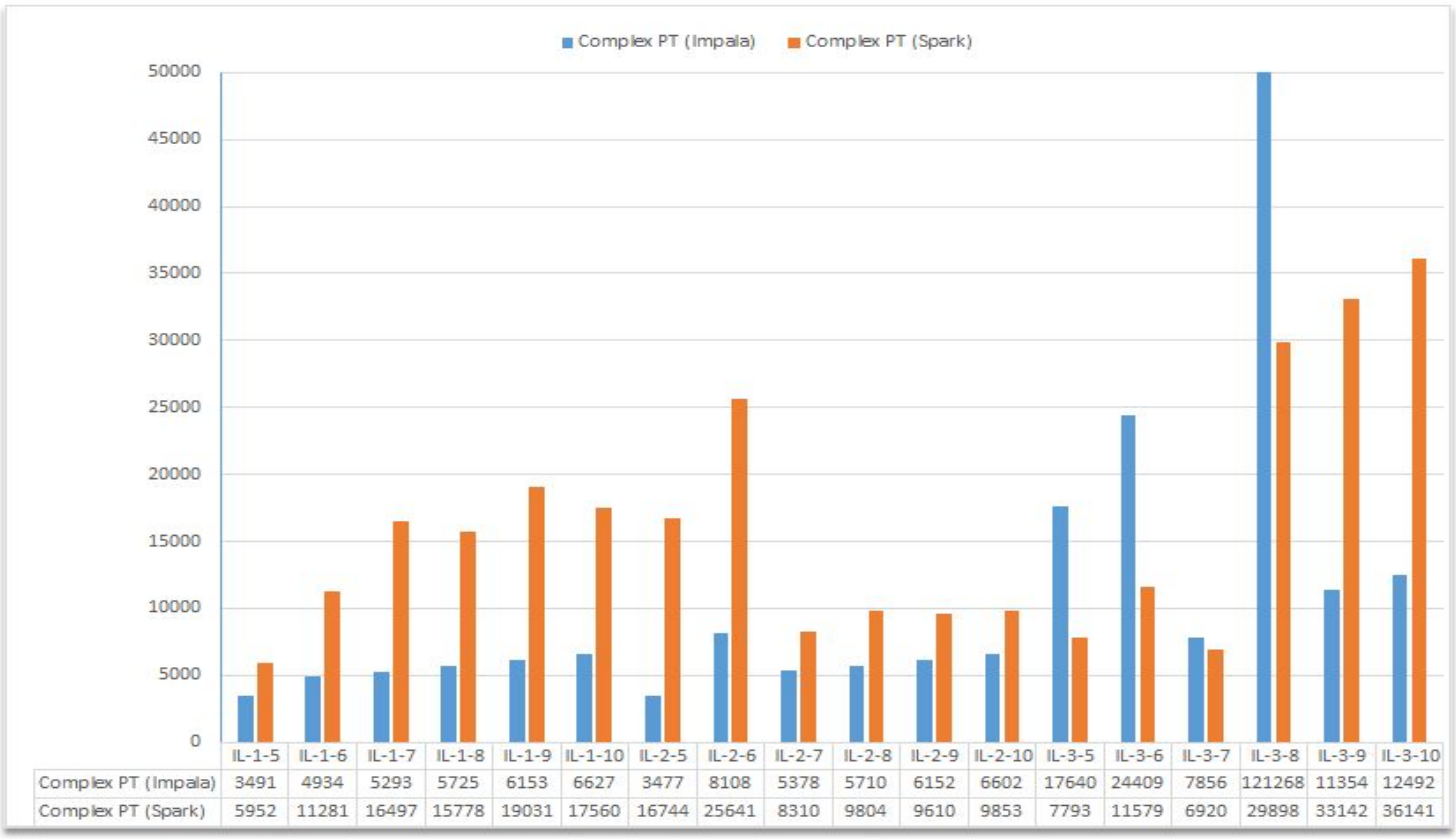
Increasing queries

WatDiv 10



	IL-1-5	IL-1-6	IL-1-7	IL-1-8	IL-1-9	IL-1-10	IL-2-5	IL-2-6	IL-2-7	IL-2-8	IL-2-9	IL-2-10	IL-3-5	IL-3-6	IL-3-7	IL-3-8	IL-3-9	IL-3-10
Complex PT (Impala)	1954	2433	2691	3168	3518	3928	1855	2668	2702	3070	3538	3970	3303	4502	2973	15185	3951	4467
Complex PT (Spark)	2258	3579	4638	4915	5837	5891	4769	8955	3046	3434	3938	2804	3914	5044	2226	13157	9072	11910

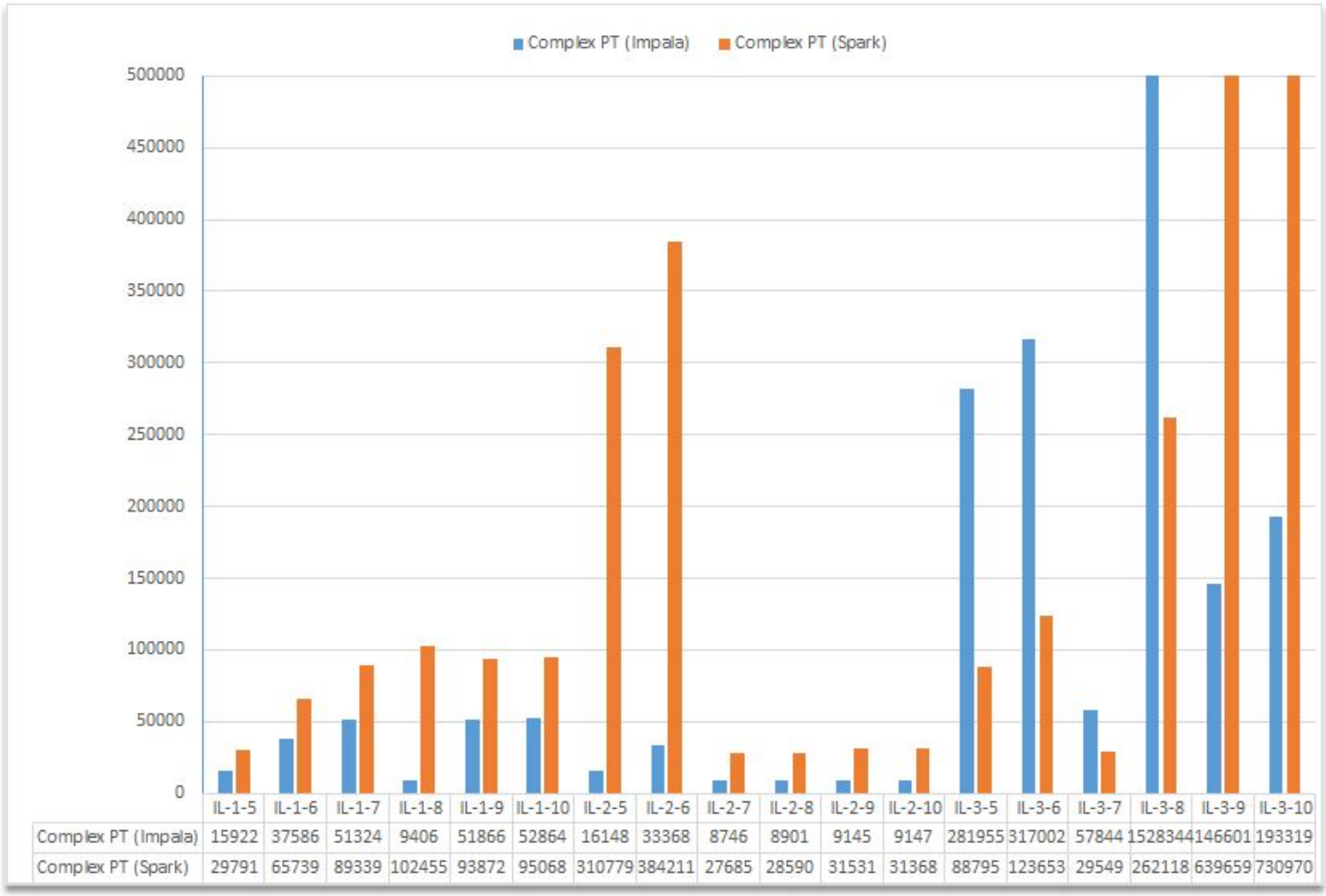
WatDiv 100



	IL-1-5	IL-1-6	IL-1-7	IL-1-8	IL-1-9	IL-1-10	IL-2-5	IL-2-6	IL-2-7	IL-2-8	IL-2-9	IL-2-10	IL-3-5	IL-3-6	IL-3-7	IL-3-8	IL-3-9	IL-3-10
Complex PT (Impala)	3491	4934	5293	5725	6153	6627	3477	8108	5378	5710	6152	6602	17640	24409	7856	121268	11354	12492
Complex PT (Spark)	5952	11281	16497	15778	19031	17560	16744	25641	8310	9804	9610	9853	7793	11579	6920	29898	33142	36141

As expected, in the graphs for increasing queries we can see that Complex Property Table is not an efficient model. It takes more time for these types of queries compared to star-shaped queries. Moreover, with the increasement of the length of a path in a query, the execution time also increases. The same tendency can be seen for the increase of database size.
To sum up, the analysis for increasing queries shows that no matter of the execution platform Impala or Spark, both engine experience disadvantages of the data model showing deterioration in the performance of specific type of queries.

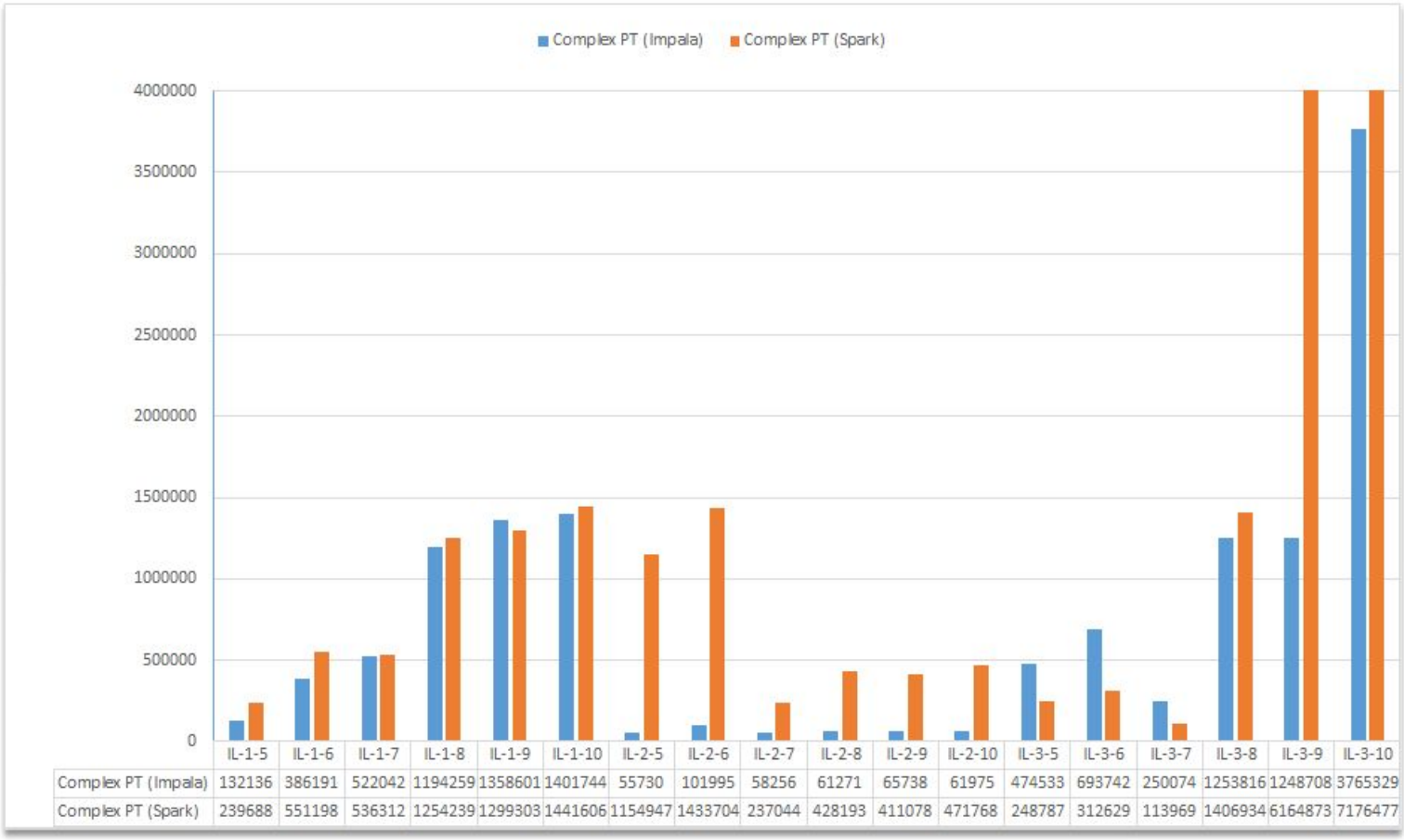
WatDiv 1000



	IL-1-5	IL-1-6	IL-1-7	IL-1-8	IL-1-9	IL-1-10	IL-2-5	IL-2-6	IL-2-7	IL-2-8	IL-2-9	IL-2-10	IL-3-5	IL-3-6	IL-3-7	IL-3-8	IL-3-9	IL-3-10
Complex PT (Impala)	15922	37586	51324	9406	51866	52864	16148	33368	8746	8901	9145	9147	281955	317002	57844	1528344	146601	193319
Complex PT (Spark)	29791	65739	89339	102455	93872	95068	310779	384211	27685	28590	31531	31368	88795	123653	29549	262118	639659	730970

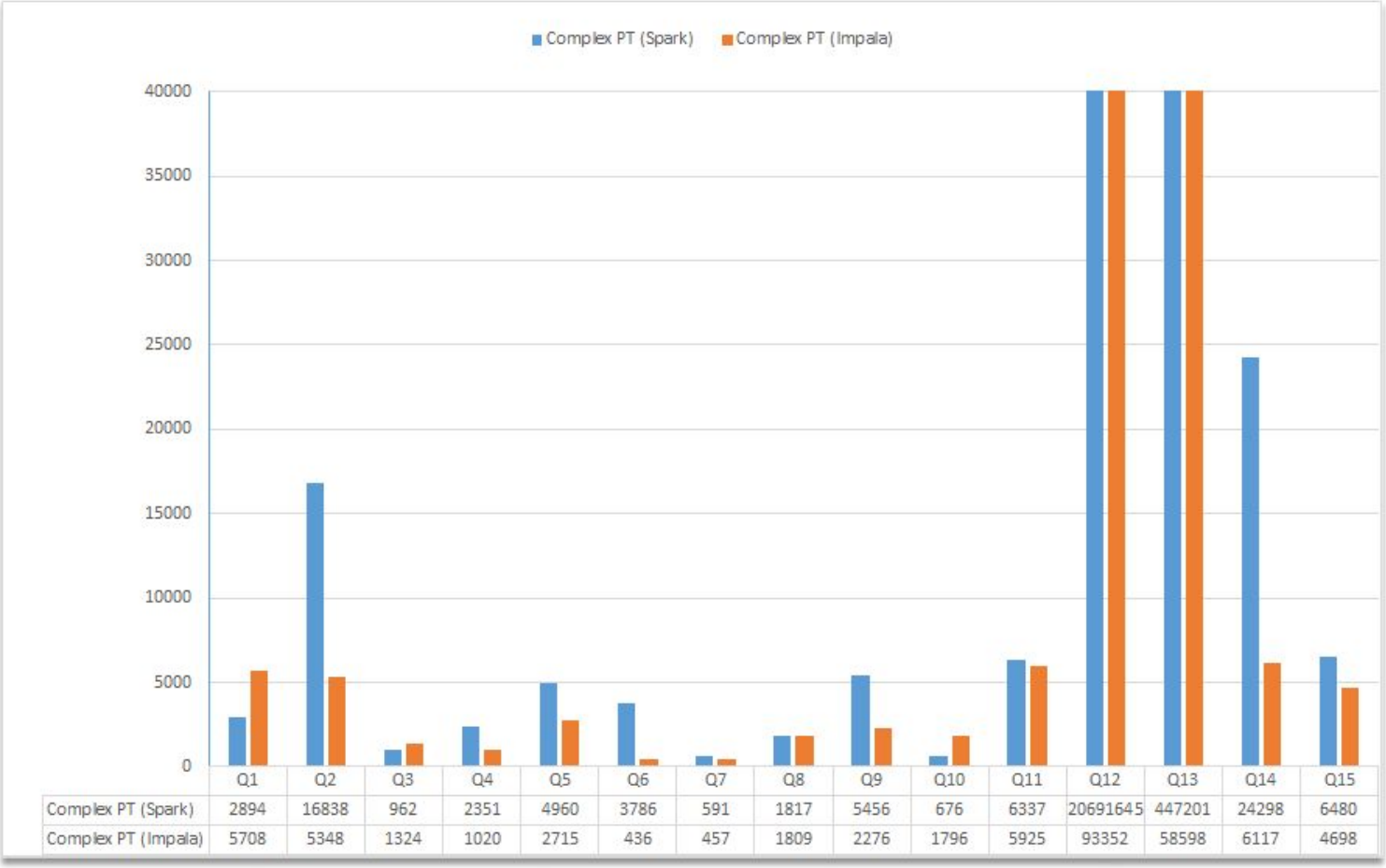
Impala execution engine usually performs better than Spark no matter for which type of queries we are executing. Something interesting is that for some specific queries Impala has more than two times worse results. We believe that the reason for the untypical situation is the way Impala uses the collected statistics about a database.

WatDiv 10000



	IL-1-5	IL-1-6	IL-1-7	IL-1-8	IL-1-9	IL-1-10	IL-2-5	IL-2-6	IL-2-7	IL-2-8	IL-2-9	IL-2-10	IL-3-5	IL-3-6	IL-3-7	IL-3-8	IL-3-9	IL-3-10
Complex PT (Impala)	132136	386191	522042	1194259	1358601	1401744	55730	101995	58256	61271	65738	61975	474533	693742	250074	1253816	1248708	3765329
Complex PT (Spark)	239688	551198	536312	1254239	1299303	1441606	1154947	1433704	237044	428193	411078	471768	248787	312629	113969	1406934	6164873	7176477

Yago



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15
Complex PT (Spark)	2894	16838	962	2351	4960	3786	591	1817	5456	676	6337	20691645	447201	24298	6480
Complex PT (Impala)	5708	5348	1324	1020	2715	436	457	1809	2276	1796	5925	93352	58598	6117	4698

Queries on Yago dataset corroborate again the same conclusions as WatDiv. Impala is faster and the model in general performs very well on star-shaped queries. Some of the queries, in particular **Q12** and **Q13**, with long paths and high number of results, show the worst performances.
The huge amount of intermediate results is very problematic, notably for Spark where they can trigger complicate memory issues.

5. Future improvements

Possible future improvements using Impala

As we showed in the evaluation, not supporting statistics for complex types can lead to really bad performances. The main reason is the misguided joins optimization.
Information about complex columns are never used. It is not possible to manually modify the table containing the statistics and the system does not leave other parameters to be tuned.

Columns without statistics are treated as zero-size, when in some cases a single cell could contain hundreds of values.
Some relations are broadcasted because they are believed to be very small, where instead they have a more significant size and this kind of operation should be avoided.
At the moment, the only way to solve this problem within Impala is to provide the exact joins order using the *STRAIGHT_JOIN* keyword. A possible future work could be to retrieve precise statistics about the graph and implement join reordering inside the translator.

Possible future improvements using Spark

Spark is quite complex engine containing components with different purposes. The most used one in this project is Spark SQL. While the deeper studying was done for the Spark SQL component, Spark in general was also explored in order to receive optimal results. One drawback that we have experienced working with Spark was multiple configurations available. This can give to a developer a lot of freedom, but on the other hand if not used right can deteriorate the results tremendously. Therefore, to achieve good results with Spark you need to spend time into understand the system in deep. Hopefully, once you have explored the most important configurations, you can improve the baseline of your queries by tuning them. We believe that there is still room for improvements in testing different combinations of configurations and maybe a possibility to automate the process.
Completely opposite to Impala, Spark is not only engine for SQL querying. Therefore, its SQL component is not so advance. There is still a lot of things that can be improved. Luckily, the addition of custom-based enhancement is really easy. So you can extend the strategies or add yours, but of course to come up with them is not an easy task. The things that are not supported natively from the platform and can increase the performance significantly is a join order and a join type. Collecting statistics and therefore reordering of joins can be seen as another good consecutive task of the project.
Lastly, it is worth it to mention a common problem that we have experienced with memory consumption. When a query retrieves a huge number of results, we had out of memory exceptions and uncontrolled increasement of consumed memory. It makes this an interesting topic where more experiments can be conducted.

6. Conclusion

We showed how to implement the Complex Property Table model and we evaluated it using Spark and Impala, one of the two most popular SQL engines on Hadoop.
Using two different systems helped us to isolate the limitations of the model from the problems of the systems.
The evaluation was performed on a small cluster using different datasets and various kinds of queries.
At the end, the results were quite expected: the model behaves very well on star queries but it suffers from long path queries.
The loading phase is noticeably fast and the compression greatly benefits from the characteristics of the table created. This can become useful in situations where the time or the size of the input data are critical to the purpose. Moreover, we faced many technical challenges, not only in configuring the systems optimally but also connected with limitations of execution possibilities.
Therefore, we identified some directions where the system and the model can be improved further, and we believe that in the future could be interesting to explore these ways.

7. Appendix

Evaluation Results

Arithmetic Mean Impala WatDiv Basic queries

Query	SF10	SF100	SF1000	SF10000
L1	597	649	636	883
L2	606	746	661	1271
L3	239	402	264	523
L4	227	318	264	474
L5	613	659	627	799
AM-L	456	555	490	790
S1	667	663	741	1247
S2	237	345	288	502
S3	220	279	264	504
S4	551	603	607	804
S5	225	258	268	538
S6	229	327	280	487

S7	594	640	654	939
AM-S	389	445	443	717
F1	706	715	735	1156
F2	644	682	685	997
F3	974	1098	1590	7578
F4	1571	1603	2167	11604
F5	1102	1185	1307	3070
AM-F	999	1057	1297	4881
C1	969	1028	1155	1913
C2	2224	2275	2750	4209
C3	255	350	654	4548
AM-C	1149	1218	1520	3557
AM	748,25	818,75	937,5	2486,25

Arithmetic Mean Spark WatDiv Basic queries

Query	SF10	SF100	SF1000	SF10000
L1	492	524	527	914
L2	548	590	597	1711
L3	1234	1388	1224	1294
L4	1178	1274	1316	1204
L5	468	518	555	1355
AM-L	784	859	844	1296
S1	637	705	759	1225
S2	1188	1419	1394	1304
S3	1424	1469	1381	1191
S4	438	481	523	878
S5	1250	1377	1476	1371
S6	1276	1509	1252	1267
S7	601	646	623	1015
AM-S	973	1087	1058	1179
F1	606	656	649	1605
F2	600	632	635	1163
F3	795	1039	2401	4391
F4	904	1015	1816	3446
F5	719	849	1831	6805
AM-F	725	838	1466	3482

C1	1995	2195	2696	3782
C2	1193	1357	3581	8173
C3	1340	1540	1618	3128
AM-C	1509	1697	2632	5028
AM	997,75	1120,25	1500	2746,25

Geometric Mean Impala WatDiv Basic queries

Query	SF10	SF100	SF1000	SF10000
L1	597	646	636	883
L2	606	738	661	1138
L3	239	392	264	523
L4	227	301	263	463
L5	611	659	627	799
GM-L	413	517	449	721
S1	667	663	740	1238
S2	236	342	288	502
S3	220	276	264	504
S4	551	602	606	803
S5	225	255	268	538
S6	228	322	279	487
S7	594	639	653	938
GM-S	345	411	401	672
F1	700	714	734	1156
F2	640	681	685	997
F3	974	1097	1589	7578
F4	1569	1602	2167	11604
F5	1102	1184	1284	2281
GM-F	945	1002	1173	2970
C1	969	1028	1155	1913
C2	2224	2275	2750	4209
C3	255	350	654	4548
GM-C	819	935	1276	3321
GM	576,2636348	667,9637963	720,5014348	1478,537914

Geometric Mean Spark WatDiv Basic queries

Query	SF10	SF100	SF1000	SF10000
L1	492	518	525	909
L2	548	561	565	1355
L3	1234	1380	1216	1291
L4	1178	1272	1286	1202
L5	468	514	544	1335
GM-L	712	765	759	1206
S1	637	689	739	1207
S2	1188	1416	1388	1299
S3	1424	1442	1362	1189
S4	438	479	518	874
S5	1250	1368	1453	1369
S6	1276	1500	1240	1264
S7	601	616	601	999
GM-S	893	977	966	1159
F1	606	643	640	1298
F2	600	628	623	1155
F3	795	1024	2347	4385
F4	904	1006	1813	3444
F5	719	844	1745	6718
GM-F	716	811	1242	2732
C1	1995	2195	2696	3782
C2	1193	1357	3581	8173
C3	1340	1540	1618	3128
GM-C	1472	1662	2500	4590
GM	904,7697484	1001,848306	1228,344333	2046,119673

Arithmetic Mean Impala WatDiv Increasing queries

Query	SF10	SF100	SF1000	SF10000
IL-1-5	1954	3491	15922	132136
IL-1-6	2433	4934	37586	386191
IL-1-7	2691	5293	51324	522042

IL-1-8	3168	5725	9406	1194259
IL-1-9	3518	6153	51866	1358601
IL-1-10	3928	6627	52864	1401744
AM-IL-1	2949	5371	36495	832496
IL-2-5	1855	3477	16148	55730
IL-2-6	2668	8108	33368	101995
IL-2-7	2702	5378	8746	58256
IL-2-8	3070	5710	8901	61271
IL-2-9	3538	6152	9145	65738
IL-2-10	3970	6602	9147	61975
AM-IL-2	2967	5905	14243	67494
IL-3-5	3303	17640	281955	474533
IL-3-6	4502	24409	317002	693742
IL-3-7	2973	7856	57844	250074
IL-3-8	15185	121268	1528344	1253816
IL-3-9	3951	11354	146601	1248708
IL-3-10	4467	12492	193319	3765329
AM-IL-3	5730	32503	420844	1281034
AM-5	2370,666667	8202,666667	104675	220799,6667
AM-6	3201	12483,66667	129318,6667	393976
AM-7	2788,666667	6175,666667	39304,66667	276790,6667
AM-8	7141	44234,33333	515550,3333	836448,6667
AM-9	3669	7886,333333	69204	891015,6667
AM-10	4121,666667	8573,666667	85110	1743016

Arithmetic Mean Spark WatDiv Increasing queries

Query	SF10	SF100	SF1000	SF10000
IL-1-5	2258	5952	29791	239688
IL-1-6	3579	11281	65739	551198
IL-1-7	4638	16497	89339	536312
IL-1-8	4915	15778	102455	1254239
IL-1-9	5837	19031	93872	1299303
IL-1-10	5891	17560	95068	1441606
AM-IL-1	4519,666667	14349,83333	79377,33333	887057,6667
IL-2-5	4769	16744	310779	1154947

IL-2-6	8955	25641	384211	1433704
IL-2-7	3046	8310	27685	237044
IL-2-8	3434	9804	28590	428193
IL-2-9	3938	9610	31531	411078
IL-2-10	2804	9853	31368	471768
AM-IL-2	4491	13327	135694	689455,6667
IL-3-5	3914	7793	88795	248787
IL-3-6	5044	11579	123653	312629
IL-3-7	2226	6920	29549	113969
IL-3-8	13157	29898	262118	1406934
IL-3-9	9072	33142	639659	6164873
IL-3-10	11910	36141	730970	7176477
AM-IL-3	9091,25	26525,25	415574	3715563,25
AM-5	3647	10163	143121,6667	547807,3333
AM-6	5859,333333	16167	191201	765843,6667
AM-7	3303,333333	10575,66667	48857,66667	295775
AM-8	7168,666667	18493,33333	131054,3333	1029788,667
AM-9	6282,333333	20594,33333	255020,6667	2625084,667
AM-10	6868,333333	21184,66667	285802	3029950,333

Geometric Mean Impala WatDiv Increasing queries

Query	SF10	SF100	SF1000	SF10000
IL-1-5	1953	3490	14306	106299
IL-1-6	2432	4934	37569	386185
IL-1-7	2691	5293	51323	522040
IL-1-8	3168	5725	9405	698404
IL-1-9	3518	6151	51860	772192
IL-1-10	3928	6626	52864	814912
GM-IL-1	2870,595501	5263,603729	29876,87733	459545,4971
IL-2-5	1853	3467	16146	55075
IL-2-6	2668	7846	33364	94881
IL-2-7	2702	5372	8746	57432
IL-2-8	3070	5707	8899	60261
IL-2-9	3537	6148	9145	64880
IL-2-10	3966	6600	9146	60926
GM-IL-2	2883,890651	5687,271608	12325,8849	64422,79905

IL-3-5	3303	17640	281955	474533
IL-3-6	4502	24409	317002	693742
IL-3-7	2973	7856	57844	250074
IL-3-8	15185	121268	1528344	1253816
IL-3-9	3951	11354	146601	1248708
IL-3-10	4467	12492	193319	3765329
GM-IL-3	4774,645423	19684,73264	246427,2808	886485,7589
GM-5	2286,451948	5976,211375	40233,48536	140577,7867
GM-6	3079,77008	9812,940046	73517,36228	294029,4834
GM-7	2785,680584	6067,546653	29611,46823	195723,1007
GM-8	5285,825011	15823,76627	50385,67036	375081,4791
GM-9	3663,357546	7544,135704	41119,80549	396977,3038
GM-10	4113,202084	8174,774488	45382,51774	571792,9146

Geometric Mean Spark WatDiv Increasing queries

Query	SF10	SF100	SF1000	SF10000
IL-1-5	2070	5620	22186	150324
IL-1-6	3553	11168	58879	426247
IL-1-7	4391	16434	77357	421114
IL-1-8	4734	15606	86520	841681
IL-1-9	5415	18087	82116	877018
IL-1-10	5461	17246	85874	1006286
GM-IL-1	4066,348972	13085,79928	62852,73402	521188,0706
IL-2-5	4684	14838	209958	916207
IL-2-6	8678	22657	376643	1085683
IL-2-7	2766	8237	27270	211170
IL-2-8	3142	9583	27955	374324
IL-2-9	3559	9525	31163	353247
IL-2-10	2794	9804	30079	456133
GM-IL-2	3898,895062	11632,8493	61946,71333	482826,4854
IL-3-5	3914	7793	88795	248787
IL-3-6	5044	11579	123653	312629
IL-3-7	2226	6920	29549	113969
IL-3-8	13157	29898	262118	1406934
IL-3-9	9072	33142	639659	6164873
IL-3-10	11910	36141	730970	7176477

GM-IL-3	**6299,152504**	**16784,83746**	**184748,1699**	**905646,6948**
GM-5	**3360,490596**	**8661,746173**	**74507,49845**	**324800,3865**
GM-6	**5377,7009**	**14309,23078**	**139968,9359**	**524966,0419**
GM-7	**3001,328906**	**9784,519003**	**39649,93822**	**216407,7846**
GM-8	**5805,822708**	**16474,48104**	**85906,1556**	**762470,3742**
GM-9	**5591,690993**	**17873,24514**	**117852,4242**	**1240710,262**
GM-10	**5664,176527**	**18282,29096**	**123597,0808**	**1487903,415**

YAGO

Name	Spark time(ms)	Impala time(ms)	Results
Q1	2894	5708	15
Q2	16838	5348	412
Q3	962	1324	73656
Q4	2351	1020	712
Q5	4960	2715	20236
Q6	3786	436	431
Q7	591	457	64719
Q8	1817	1809	104
Q9	5456	2276	374024187
Q10	676	1796	32
Q11	6337	5925	35752
Q12	20691645	93352	2946469
Q13	447201	58598	159600
Q14	24298	6117	360377
Q15	6480	4698	24
AM	**1414419,467**	**12771,93333**	
GM	**8419,079956**	**3513,66186**	

8. Bibliography

1. Simon Skilevic, “S2RDF: Distributed in-memory execution of SPARQL queries using Apache Spark SQL and Extended Vertical Partitioning”, http://www.vldb.org/pvldb/vol9/p804-schaetzle.pdf

2. A. Schätzle, M. Przyjaciel-Zablocki, A. Neu, and G. Lausen, “Sempala: Interactive SPARQL query processing on hadoop,” in The Semantic Web–ISWC 2014. Springer, pp. 164–179, http://link.springer.com/chapter/10.1007/978-3-319-11964-9_11
3. Manuel Schneider, Erweiterung des Projektes Sempala um das Datenformat Single Table
4. Antony R. Neu, Distributed Evaluation of SPARQL queries with Impala and MapReduce, http://www.antony-neu.com/wp-content/uploads/2014/06/thesis.pdf
5. Apache Spark - Lightning-Fast Cluster Computing, https://spark.apache.org/
6. Apache Impala - open source, analytic MPP database for Apache Hadoop, https://impala.incubator.apache.org/
7. The WatDiv data, http://dsg.uwaterloo.ca/watdiv/
8. YAGO - huge semantic knowledge base, http://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago/#c10444