

# Implementation of Databases

## Assignment 6

Participants:  
(sorted in last name order)  
Ulfet CETIN  
Shreya KAR  
Samuel ROY

## Exercise 6.1 (Cost Estimation using Spark)

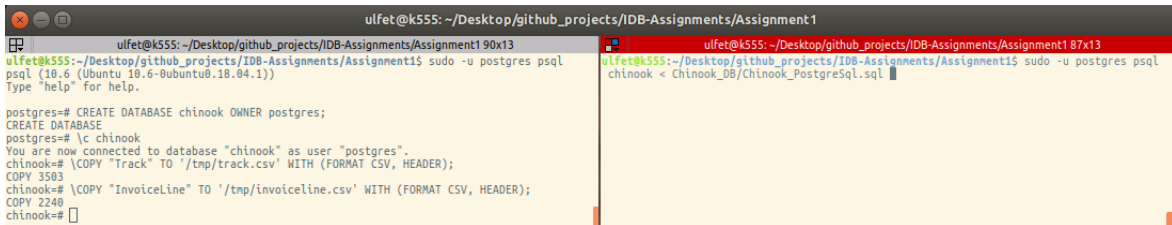
Consider the Chinook database from Exercise 1.2 and the 5th Query Q5.

1. Install Spark 2.4. In Exercise 1.2 you have data stored in PostgreSQL database. Export the data from tables Track and InvoiceLine individually to CSV files. Load these two CSV files into Spark. Alternatively, you can also load the data frames directly from PostgreSQL. Provide your codes.

We used Ubuntu 18.04 LTS, clean installation, for both PostgreSQL and Spark.  
Here is how we do it:

- PostgreSQL CSV Export

1. Install PostgreSQL using terminal:  
`sudo apt update`  
`sudo apt install postgresql postgresql-contrib`
2. Connect to PostgreSQL, create DB named "chinook":  
`sudo -u postgres psql`  
(psql shell) `CREATE DATABASE chinook OWNER postgres;`  
(psql shell) `\c chinook`
3. Import data from Assignment 1.  
`sudo -u postgres psql chinook < Chinook_DB/Chinook_PostgreSQL.sql`
4. Export data from inside psql shell.  
`\COPY "Track" TO '/tmp/track.csv' WITH (FORMAT CSV, HEADER);`  
`\COPY "InvoiceLine" TO '/tmp/invoiceline.csv' WITH (FORMAT CSV, HEADER);`  
(the files put in tmp folder to not cause privilege errors, retrieved from that folder)



```
ulfet@k555: ~/Desktop/github_projects/IDB-Assignments/Assignment1
ulfet@k555:~/Desktop/github_projects/IDB-Assignments/Assignment1$ sudo -u postgres psql
psql (10.6 (Ubuntu 10.6-0ubuntu0.18.04.1))
Type "help" for help.

postgres=# CREATE DATABASE chinook OWNER postgres;
CREATE DATABASE
postgres=# \c chinook
You are now connected to database "chinook" as user "postgres".
chinook=# \COPY "Track" TO '/tmp/track.csv' WITH (FORMAT CSV, HEADER);
COPY 3503
chinook=# \COPY "InvoiceLine" TO '/tmp/invoiceline.csv' WITH (FORMAT CSV, HEADER);
COPY 2240
chinook=#
```

Figure 1: CSV Extraction via PostgreSQL

- Spark CSV Import

1. Install Spark & Java8 (required for Spark)  
`sudo apt-add-repository ppa:webupd8team/java`  
`sudo apt-get update`  
`sudo apt-get install oracle-java8-installer`  
<https://spark.apache.org/downloads.html> (download spark-2.4.0-bin-hadoop2.7.tgz)  
`tar -xzf spark-2.4.0-bin-hadoop2.7.tgz`  
`cd spark-2.4.0-bin-hadoop2.7/bin`  
`./pyspark` (run pyspark once)

2. We would use Spark through Python.  
Here is the code for importing CSV files:

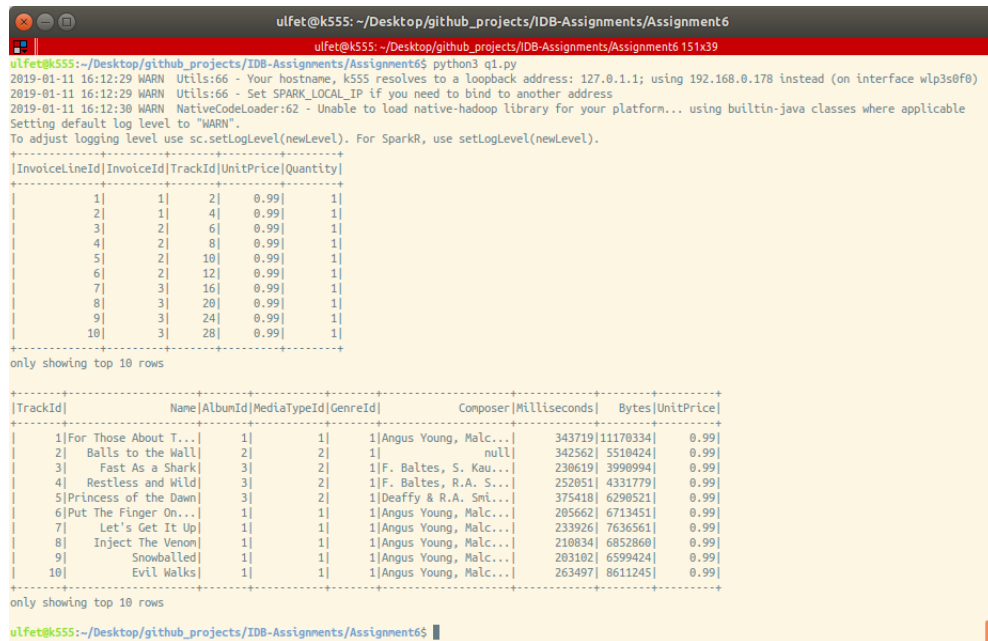
```
# Spark is not installed system-wide
# that is why findspark pip package is needed
import findspark
findspark.init()

from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("IDB Assignment 6 Query Runner") \
    .getOrCreate()

# read CSV files into DataFrames
invLineDF = spark.read.csv("csv_files/invoiceline.csv",header=True);
trackDF = spark.read.csv("csv_files/track.csv",header=True);

# Displays the content of the DataFrame to stdout
invLineDF.show(10)
trackDF.show(10)
```



```
ulfet@k555: ~/Desktop/github_projects/IDB-Assignments/Assignment6
ulfet@k555: ~/Desktop/github_projects/IDB-Assignments/Assignment6 151x39
ulfet@k555:~/Desktop/github_projects/IDB-Assignments/Assignment6$ python3 ql.py
2019-01-11 16:12:29 WARN Utils:66 - Your hostname, k555 resolves to a loopback address: 127.0.1.1; using 192.168.0.178 instead (on interface wlp3s0f0)
2019-01-11 16:12:29 WARN Utils:66 - Set SPARK_LOCAL_IP if you need to bind to another address
2019-01-11 16:12:30 WARN NativeCodeLoader:62 - Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
+-----+
|InvoiceLineId|InvoiceId|TrackId|UnitPrice|Quantity|
+-----+
|1|1|2|0.99|1|
|2|1|4|0.99|1|
|3|2|6|0.99|1|
|4|2|8|0.99|1|
|5|2|10|0.99|1|
|6|2|12|0.99|1|
|7|3|16|0.99|1|
|8|3|20|0.99|1|
|9|3|24|0.99|1|
|10|3|28|0.99|1|
+-----+
only showing top 10 rows
+-----+
|TrackId|Name|AlbumId|MediaTypeId|GenreId|Composer|Milliseconds|Bytes|UnitPrice|
+-----+
|1|For Those About T...|1|1|1|Angus Young, MaLc...|343719|11170334|0.99|
|2|Balls to the Wall|2|2|1|null|342562|5510424|0.99|
|3|Fast As a Shark|3|2|1|F. Baltes, S. Kau...|230619|3990994|0.99|
|4|Restless and Wild|3|2|1|F. Baltes, R.A. S...|252051|4331779|0.99|
|5|Princess of the Dawn|3|2|1|Deaffy & R.A. SmL...|375418|6290521|0.99|
|6|Put The Finger On...|1|1|1|Angus Young, MaLc...|205662|6713451|0.99|
|7|Let's Get It Up|1|1|1|Angus Young, MaLc...|233926|7636561|0.99|
|8|Inject The Venom|1|1|1|Angus Young, MaLc...|210834|6852860|0.99|
|9|Snowballed|1|1|1|Angus Young, MaLc...|203102|6599424|0.99|
|10|Evil Walks|1|1|1|Angus Young, MaLc...|263497|8611245|0.99|
+-----+
only showing top 10 rows
ulfet@k555:~/Desktop/github_projects/IDB-Assignments/Assignment6$
```

Figure 2: CSV Importing in Spark

2. Run the previous query Q5 from Exercise 1.2.5 in Spark and provide the logical plan and the physical plan. Note if you have directly connect to PostgreSQL in last task, then for this task note that the query has to be processed within Spark. Provide your codes.

Here is the Python code for running and displaying the query.

Note that, the code below has to be run after the code given above.

```
# to run SQL queries, save DataFrames as Temporary Views
invLineDF.createOrReplaceTempView("InvoiceLine")
trackDF.createOrReplaceTempView("Track")

# query taken from Assignment 1 - Official Solution
# https://www3.elearning.rwth-aachen.de/ws18/18ws-186186/assessment/
# Lists/LA_SampleSolutions/A01/Exercise_1_withSolutions.pdf

# List the top 5 most purchased tracks over all.
queryText = \
    "SELECT t.Name, count(t.Name) as PurchaseCount \
    FROM Track t, InvoiceLine i \
    WHERE t.TrackId = i.TrackId \
    Group By t.Name \
    Order By PurchaseCount \
    Desc Limit 5"

# run the query & display the answer
sqlDF = spark.sql(queryText)
sqlDF.show()
```

```
ulfet@k555: ~/Desktop/github_projects/IDB-Assignments/Assignment6
ulfet@k555: ~/Desktop/github_projects/IDB-Assignments/Assignment6 151x17
ulfet@k555:~/Desktop/github_projects/IDB-Assignments/Assignment6$ python3 q1.py
2019-01-11 16:19:50 WARN Utils:66 - Your hostname, k555 resolves to a loopback address: 127.0.1.1; using 192.168.0.178 instead (on interface wlp3s0f0)
2019-01-11 16:19:50 WARN Utils:66 - Set SPARK_LOCAL_IP if you need to bind to another address
2019-01-11 16:19:51 WARN NativeCodeLoader:62 - Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
-----+-----+
|      Name|PurchaseCount|
-----+-----+
|The Trooper|             5|
|Hallowed Be Thy Name|         4|
|      Untitled|         4|
|      Eruption|         4|
|The Number Of The...|         4|
-----+-----+
ulfet@k555:~/Desktop/github_projects/IDB-Assignments/Assignment6$
```

Figure 3: Spark Query Results

```

ulfet@k555: ~/Desktop/github_projects/IDB-Assignments/Assignment6
ulfet@k555: ~/Desktop/github_projects/IDB-Assignments/Assignment6 145x44
ulfet@k555:~/Desktop/github_projects/IDB-Assignments/Assignment6$ python3 q1.py
2019-01-12 12:50:10 WARN Utils:66 - Your hostname, k555 resolves to a loopback address: 127.0.1.1; using 134.61.167.65 instead (on interface wlp3s0f0)
2019-01-12 12:50:10 WARN Utils:66 - Set SPARK_LOCAL_IP if you need to bind to another address
2019-01-12 12:50:11 WARN NativeCodeLoader:62 - Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
+-----+
|      Name|PurchaseCount|
+-----+
|The Trooper|          5|
|Hallowed Be Thy Name|          4|
|      Untitled|          4|
|      Eruption|          4|
|The Number Of The...|          4|
+-----+

== Parsed Logical Plan ==
'GlobalLimit 5
+- 'LocalLimit 5
  +- 'Sort ['PurchaseCount DESC NULLS LAST], true
  +- 'Aggregate ['t.Name], ['t.Name, 'count('t.Name) AS PurchaseCount#48]
  +- 'Filter ('t.TrackId = 'i.TrackId)
  +- 'Join Inner
    :- 'SubqueryAlias 't'
    :- 'UnresolvedRelation 'Track'
    +- 'SubqueryAlias 'i'
    :- 'UnresolvedRelation 'InvoiceLine'

== Analyzed Logical Plan ==
Name: string, PurchaseCount: bigint
GlobalLimit 5
+- LocalLimit 5
  +- Sort [PurchaseCount#48L DESC NULLS LAST], true
  +- Aggregate [Name#31], [Name#31, count(Name#31) AS PurchaseCount#48L]
  +- Filter (TrackId#30 = TrackId#12)
  +- Join Inner
    :- SubqueryAlias 't'
    :- SubqueryAlias 'track'
    :- Relation[TrackId#30,Name#31,AlbumId#32,MediaTypeId#33,GenreId#34,Composer#35,Milliseconds#36,Bytes#37,UnitPrice#38] csv
    +- SubqueryAlias 'i'
    +- SubqueryAlias 'invoiceLine'
    +- Relation[InvoiceLineId#10,InvoiceId#11,TrackId#12,UnitPrice#13,Quantity#14] csv

```

Figure 4: Spark Plans - Screenshot 1

In order to get all the related plans on Spark, we added a line of code to q1.py (our Spark runner given above):

```

# run the query & display the answer
sqlDF = spark.sql(queryText)
sqlDF.show()
sqlDF.explain(True)

```

```

== Optimized Logical Plan ==
GlobalLimit 5
+- LocalLimit 5
  +- Sort [PurchaseCount#48L DESC NULLS LAST], true
  +- Aggregate [Name#31], [Name#31, count(Name#31) AS PurchaseCount#48L]
  +- Project [Name#31]
  +- Join Inner, (TrackId#30 = TrackId#12)
    :- Project [TrackId#30, Name#31]
    :- Filter isNotNull(TrackId#30)
    :- Relation[TrackId#30,Name#31,AlbumId#32,MediaTypeId#33,GenreId#34,Composer#35,Milliseconds#36,Bytes#37,UnitPrice#38] csv
  +- Project [TrackId#12]
  +- Filter isNotNull(TrackId#12)
  +- Relation[InvoiceLineId#10,InvoiceId#11,TrackId#12,UnitPrice#13,Quantity#14] csv

== Physical Plan ==
TakeOrderedAndProject(limit=5, orderBy=[PurchaseCount#48L DESC NULLS LAST], output=[Name#31,PurchaseCount#48L])
+- *(3) HashAggregate(keys=[Name#31], functions=[count(Name#31)], output=[Name#31, PurchaseCount#48L])
  +- Exchange hashpartitioning(Name#31, 280)
  +- *(2) HashAggregate(keys=[Name#31], functions=[partial_count(Name#31)], output=[Name#31, count#60L])
    +- *(2) Project [Name#31]
      +- *(2) BroadcastHashJoin [TrackId#30], [TrackId#12], Inner, BuildRight
        :- *(2) Project [TrackId#30, Name#31]
        :- *(2) Filter isNotNull(TrackId#30)
        :- *(2) FileScan csv [TrackId#30,Name#31] Batched: false, Format: CSV, Location: InMemoryFileIndex[file:/home/ulfet/Desktop/github_projects/IDB-Assignments/Assignment6/csv_files/..., PartitionFilters: [], PushedFilters: [IsNotNull(TrackId)], ReadSchema: struct<TrackId:string,Name:string>
        +- BroadcastExchange HashedRelationBroadcastMode(List(input[0, string, true]))
      +- *(1) Project [TrackId#12]
        +- *(1) Filter isNotNull(TrackId#12)
        +- *(1) FileScan csv [TrackId#12] Batched: false, Format: CSV, Location: InMemoryFileIndex[file:/home/ulfet/Desktop/github_projects/IDB-Assignments/Assignment6/csv_files/..., PartitionFilters: [], PushedFilters: [IsNotNull(TrackId)], ReadSchema: struct<TrackId:string>

```

Figure 5: Spark Plans - Screenshot 2

3. Compare the query plans of Q5 by Spark and PostgreSQL (Exercise 1.2.5), and itemize the differences.

```

ulfet@k555: ~
-----
Limit (cost=237.91..237.93 rows=5 width=24) (actual time=4.781..4.784 rows=5 loops=1)
-> Sort (cost=237.91..243.51 rows=2240 width=24) (actual time=4.780..4.781 rows=5 loops=1)
    Sort Key: (count(t."Name")) DESC
    Sort Method: top-N heapsort  Memory: 25kB
-> HashAggregate (cost=178.31..200.71 rows=2240 width=24) (actual time=3.735..4.240 rows=1888 loops=1)
    Group Key: t."Name"
-> Hash Join (cost=123.82..167.11 rows=2240 width=16) (actual time=1.638..2.712 rows=2240 loops=1)
    Hash Cond: (i."TrackId" = t."TrackId")
-> Seq Scan on "InvoiceLine" i (cost=0.00..37.40 rows=2240 width=4) (actual time=0.009..0.281 rows=2240 loops=1)
-> Hash (cost=80.03..80.03 rows=3503 width=20) (actual time=1.603..1.603 rows=3503 loops=1)
    Buckets: 4096  Batches: 1  Memory Usage: 219kB
-> Seq Scan on "Track" t (cost=0.00..80.03 rows=3503 width=20) (actual time=0.004..0.695 rows=3503 loops=1)

Planning time: 0.272 ms
Execution time: 4.882 ms
(14 rows)

chinook=#

ulfet@k555: ~/Desktop/github_projects/IDB-Assignments/Assignment6
-----
ulfet@k555:~/Desktop/github_projects/IDB-Assignments/Assignment6 160x17
== Physical Plan ==
TakeOrderedAndProject(limit=5, orderBy=[PurchaseCount#48L DESC NULLS LAST], output=[Name#31,PurchaseCount#48L])
+- *(3) HashAggregate(keys=[Name#31], functions=[count(Name#31)], output=[Name#31, PurchaseCount#48L])
   +- Exchange hashpartitioning(Name#31, 200)
      +- *(2) HashAggregate(keys=[Name#31], functions=[partial_count(Name#31)], output=[Name#31, count#60L])
         +- *(2) Project [Name#31]
            +- *(2) BroadcastHashJoin [TrackId#30], [TrackId#12], Inner, BuildRight
               :- *(2) Project [TrackId#30, Name#31]
               :  +- *(2) Filter isNotNull(TrackId#30)
               :  +- *(2) FileScan csv [TrackId#30,Name#31] Batched: false, Format: CSV, Location: InMemoryFileIndex[file:/home/ulfet/Desktop/github_projects/IDB-Assignments/Assignment6/csv_files/..., PartitionFilters: [], PushedFilters: [IsNotNull(TrackId)], ReadSchema: struct<TrackId:string,Name:string>
               +- BroadcastExchange HashedRelationBroadcastMode(List(input[0, string, true]))
                  +- *(1) Project [TrackId#12]
                     +- *(1) Filter isNotNull(TrackId#12)
                        +- *(1) FileScan csv [TrackId#12] Batched: false, Format: CSV, Location: InMemoryFileIndex[file:/home/ulfet/Desktop/github_projects/IDB-Assignments/Assignment6/csv_files/..., PartitionFilters: [], PushedFilters: [IsNotNull(TrackId)], ReadSchema: struct<TrackId:string>
ulfet@k555:~/Desktop/github_projects/IDB-Assignments/Assignment6$

```

Figure 6: Query Plan Comparison

Postgres' plan differs in the ways:

- Postgre starts with sequential scanning Track and hashing its rows into buckets.
- Then it sequential scans the InvoiceLine table and applies HashJoin on TrackId field(s) of those two tables.
- Then it groups the resulting combined rows on Track table's "Name" field and counts how many times a name has appeared.
- Finally, Postgre sorts the combined table on PurchaseCount and returns the first 5 rows (top 5 tracks).

Spark's plan differs in the ways:

- Spark starts with scanning CSV file of InvoiceLine table, filters out rows that has their "TrackId" fields as null and projects <TrackId> values of rows. Then it broadcasts those TrackId values.
- Then, Spark, this time, scans CSV file of Track table, filters out rows that has their "TrackId" fields as null and projects <TrackId, Name> values of rows.
- Then it applies BroadcastHashJoin on TrackId fields of two resulting projection. On the resulting rows of BroadcastHashJoin, it projects <Name> values of rows.
- After that, Spark HashAggregates on "Name" field and counts how many times a specific name has appeared and names this count as PurchaseCount. Result of this step is rows containing <Name, PurchaseCount>
- Finally, Spark sorts the <Name, PurchaseCount> rows on "PurchaseCount" field, finds the greatest 5 rows, projects the "Name" field of those 5 rows as result.

## Exercise 6.2 (Datalog)

1. Given the following extensional database:

- (a) teamNBA(X): X is a NBA basketball team
- (b) coach(X,Y): X is a coach of team Y
- (c) player(X,Y): X is a basketball player of team Y
- (d) taller(X,Y): X is taller than Y.

Please formalize the following rules using Datalog and the given predicates (notably, there might be also non-NBA teams in this database):

- (a) shorterTeammate(X,Y): X and Y are players in the same NBA basketball team, and X is shorter than Y.

Solution:

playersOfNBATeams(X,Y):-player(X,Y),teamNBA(Y)

shorterTeammate(X,Y):-playersOfNBATeam(X,Z),playersOfNBATeam(Y,Z),taller(Y,X)

- (b) coachOrPlayer(X,Y): X is a coach or a player of a NBA basketball team Y.

Solution:

coachOrPlayer(X,Y):-coach(X,Y),teamNBA(Y)

coachOrPlayer(X,Y):-player(X,Y),teamNBA(Y)

- (c) coachTallest(X,Y): X is a coach of a NBA team Y, and X is taller than all the players in the team Y.

Solution:

Solution:

shorterCoachThanPlayer(X):-coach(X,Z),player(Y,Z),teamNBA(Z),taller(Z,X)

coachTallest(X,Y):-coach(X,Y),teamNBA(Y),NOT shorter(X)

2. You have given the below facts F, rules R, and a query Q.

- (a) Decide if the program given by the rules R is stratifiable. State why or why not it is stratifiable (corresponding graph with strati and statement).
- (b) Given the facts F and rules R do a fixpoint computation with all intermediate steps. Based on the computation, write down all answers to query Q.

$F : s(a,b)s(b,c)s(c,b)$

$r(a)r(c)r(d)$

$R : p(X,Y) \leftarrow s(X,Y), NOT r(Y)$

$q(X,Y) \leftarrow p(Y,X), s(X,Y)$

$q(X,Y) \leftarrow q(Y,X), r(X)$

$t(X,Y) \leftarrow r(X), q(X,Y)$

$Q : t(X,Y)$

Solution:

- (a) The given program is stratifiable because there is no negative predicate in the body of a recursive rule.  
i.e. the rule

$q(X,Y) \leftarrow q(Y,X), r(X)$  has no negative dependency

Or we can say that no two predicates of the same stratum depend negatively on each other as seen in the below graph

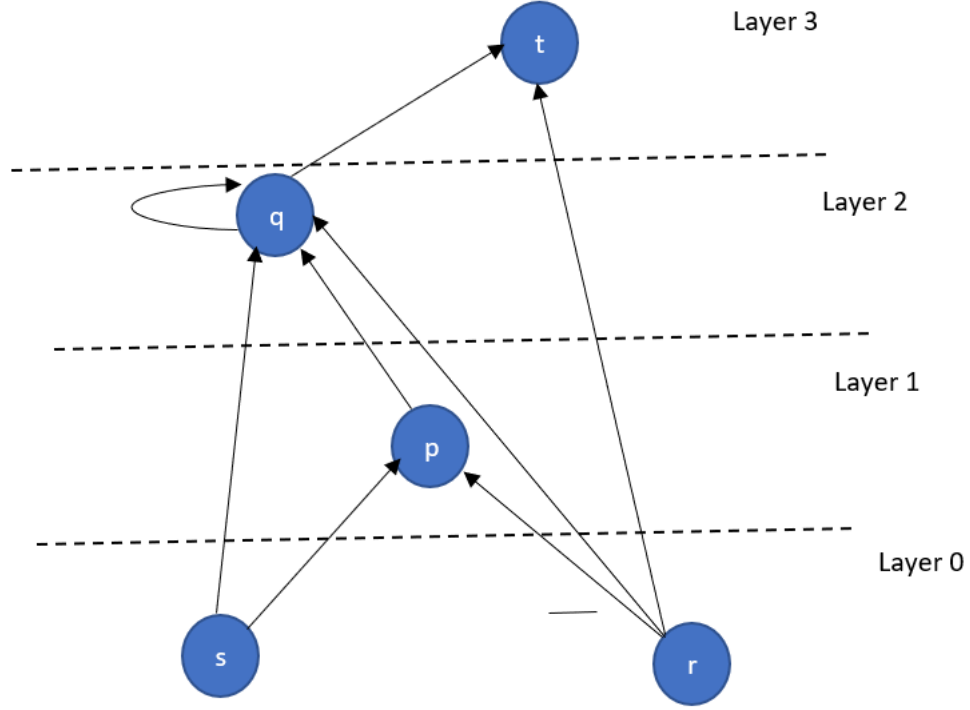


Figure 7: Query Plan Comparison

(b) Starting with F (F is all set of facts)

i. Adding Td(F0)

$$F1 = F \cup \{p(a,b), p(b,c)\}$$

ii. Adding Td(F1)

$$F2 = F \cup \{p(a,b), a(b,c), q(b,c)\}$$

$$F2 = F \cup \{p(a,b), a(b,c), q(c,b)\}$$

iii. Adding Td(F2)

$$F3 = F \cup \{p(a,b), a(b,c), q(b,c), t(c,b)\}$$

Another application of Td(F3) yields the same result, therefore the least fixed point is reached.

Answer of query  $q:t(X,Y)$  is (c,b)



### Exercise 6.3 (Data Integration)

Given is the following global schema with three relations:

Hospital(HospitalName; Street;CityName; Postcode)

Doctor(DoctorName;HospitalName;Disease)

Patient(PatientName; Age; AttendingDoctorName) There are four data sources:

One quick way is to do it manually for each bullet:

- DS1: AachenHospital(HospitalName, Street,Postcode): hospitals in Aachen.
  - DS2: DoctorsAndPatients(DoctorName, PatientName, Disease): doctors and their patients.
  - DS3: TeenagePatients(PatientName, Age): patient information with the patient age less than 18.
  - DS4: Hospital(HospitalName, Street, Postcode, CityID), City(CityID, CityName, Country): hospitals and cities.
1. Provide the LAV mappings between the source DS1 and the global schema.
  2. Provide the LAV mappings between the source DS2 and the global schema.
  3. For DS3 and the global schema, which mapping is more precise, a GAV mapping or a LAV mapping? Why? Also provide the mapping you think is more precise.
  4. Consider DS4 and the global schema. Rewrite the below query on the global schema to a query on the schema of DS4.  
q(CityName) :- Hospital('FrancisHospital',-,CityName; ):

Solution:

1. AachenHospital(HospitalName, Street, PostCode):-  
Hospital(HospitalName,Street,CityName,PostCode),  
CityName='Aachen'
2. DoctorsAndPatients(DoctorName,PatientName,Disease):-  
Doctor(DoctorName,\_,Disease),  
Patient(PatientName,\_,AttendingDoctorName)
3. LAV:  
TeenagePatients(PatientName,Age):-  
Patient(PatientName,Age,\_), Age<18

GAV:

Patient(PatientName,Age,AttendingDoctorName):-  
DoctorsAndPatients(DoctorName,PatientName,\_),  
TeenagePatients(PatientName,Age)

Here LAV mapping is more precise, because in GAV mapping the Age does not map to patients who are older than 18 years. Here Source centric Mapping is used (As constraint is on source level). But global schema does not define any constraint.

4. q'(CityName):- Hospital('FrancisHospital',\_,\_,CityID), City(CityID,CityName,\_)

### Exercise 6.4 (Answer questions briefly)

1. What is the goal of systems like Pig Latin or Hive?
2. When do you need to do shuffling in Spark? What are narrow and wide dependencies?
3. Sketch a data integration architecture. What is a mediator? What is the task of a mediator in a data integration architecture?
4. What is the Herbrand Base and the Herbrand Model?

Solution:

1. The main goal of high level languages like pig or hive is to reduce the workload of the user by preventing the need of writing complex map-reduce jobs in java. Pig uses a procedural style approach and hive uses a SQL style approach which makes data manipulation of Hadoop data easy.
2. We need shuffling in spark i.e. distribution of RDD between different nodes of a cluster when the operation can't be performed within a single partition. Examples of such operations are join, distinct, groupByKey. When a partition is dependent of one or two parent partitions it is called as "Narrow dependencies". However, when a partition is dependent on multiple parent partitions it is termed as "wide dependencies".
3. A mediator maintains a global schema and mappings between the global and source schemas. It is the task of the mediator to consult the mappings to decide which data to retrieve from the sources and how to combine them appropriately in order to form the answer to the user's query.

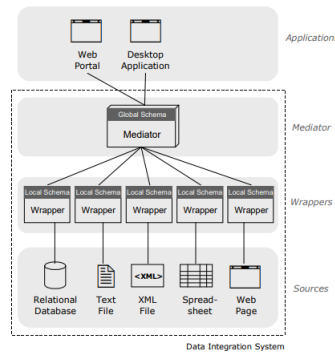


Figure 8: Data Integration System Architecture

4. Herbrand Base of D: All positive ground literals are constructable from predicates in D and constants in D.  
Herbrand Model of D: A Herbrand Model is every subset M of the Herbrand Base of D, such that:  
Every fact from F is contained in M. For every ground instance of a rule in D over constants in D, if M contains all literals in the body, then M contains the head as well.  
A minimal model does not properly contain any other model