M2 – DAC - LODAS - 2019

MU5IN860 Linked Open Data, Apprentisssage Symbolique

Bernd Amann

SU

5 novembre 2019

Systèmes RDF

Outline

Services de données RDF

Services RDF

- importation / exportation : fichiers, BD $\leftrightarrow$ graphes RDF
- stockage : graphes/ensembles des triplets RDF
- interrogation : « endpoint » SPARQL, service web/REST pour l'interrogation par SPARQL
- inférence : RDFS, OWL, règles

Défis

- hétérogénéité : données et des schemas
- volume : graphes avec des milliards de noeuds et d'arcs
- distribution : données, services (systèmes fédérés)

Classification des systèmes RDF

Systèmes RDF « non-natifs » : réutilisation

- réutilisation de technologies existantes (relationnel, noSQL) :
  - réutilisation au niveau logique : représentation de graphes dans des tables et traduction de requêtes SPARQL en requêtes SQL
  - réutilisation au niveau physique : structures d'index (arbres-B), opérateurs physiques (sélection, jointures, ...)
- exemples : RDF-3X, Jena, Virtuoso, Graph DB, HBase/MapReduce, SPARK/Sparql SQL

Systèmes RDF « natifs » : spécialisation

- systèmes spécialisés pour le traitement de données RDF
  - représentation physique spécialisée
  - opérateurs conçus pour le traitement de graphes RDF
- exemples : Sesame/rdf4j, WaterFowl, gStore

La séparation entre systèmes « natifs » et « non-natifs » n'est pas toujours évidente.

Critères de comparaison de systèmes

Comment comparer les systèmes ?

- coûts de stockage (disque/mémoire)
- coûts d'importation / pré-traitement de données (indexation, compression)
- coûts d'évaluation des requêtes (motifs de graphes, jointures)
- scalabilité : distribution des données, parallélisation, réplication
- complexité de déploiement : architecture, configuration du système

Outline

SPARQL et SQL

Objectif

Utiliser un moteur SQL pour évaluer des requêtes SPARQL :

- Schéma de stockage relationnel pour des données RDF
  - ensembles de tables pour stocker les triplets
  - schémas génériques (indépendant de RDFS) et spécifiques (dépendant du schéma RDFS)
- Evaluation de requêtes SPARQL :
  - traduction logique : optimisation et évaluation SQL
  - traduction physique : optimisation SPARQL et évaluation SQL

2 - Triple stores « SQL »

- Schémas de stockage génériques
- Schémas de stockage partitionnées
- Schémas de stockage étendus

Solution 1 : Table de triplets

Schéma de stockage **générique**

- une seule table principale : **Triples(subject, property, object)**
- tables « support » (dictionnaire) et index
- **Virtuoso**, **Jena**, Sesame, 3store, Kaon, RStar

Avantages

+ importation / exportation simple et rapide (triplet = nuplet)
+ traitement uniforme des données RDF et schémas RDFS
+ représentation « naturelle » de propriétés multivaluées et optionnelles

Inconvénients

- **beaucoup d'auto-jointures sur une grande table** $\rightarrow$ indexation exhaustive (RDF3-X)

Motifs simples = jointures SQL

Requête optional1

```
PREFIX  :  <http://example.org/ns#>
SELECT ?title ?price
FROM <bibliosem.ttl>
 WHERE { ?x :title ?title .
         ?x :price ?price . }
```

Expression algébrique de optional1

```
(prefix ((: <http://example.org/ns#>))
  (project (?title ?price)
    (bgp
      (triple ?x :title ?title)
      (triple ?x :price ?price)
    )))
```

SQL

```
select t1.object as title,
       t2.object as price
  from Triples t1, Triples t2
 where t1.property = ':title'
   and t2.property = ':price'
   and t1.subject = t2.subject;
```

ou

```
select t1.object as title,
       t2.object as price
  from Triples t1 join Triples t2
       on (t1.subject = t2.subject)
 where t1.property = ':title'
   and t2.property = ':price';
```

OPTIONAL = left outer join SQL

Requête optional1a

```
PREFIX  :  <http://example.org/ns#>
SELECT  ?title  ?price
FROM <bibliosem.ttl>
 WHERE { ?x :title  ?title  .
         OPTIONAL { ?x :price  ?price  . } }
```

Expression algébrique de optional1a

```
(prefix ((: <http://example.org/ns#>))
  (project (?title ?price)
    (leftjoin
      (triple ?x :title ?title)
      (triple ?x :price ?price)
    )))
```

- q1 : requête motif `?x :title ?title`
- q2 : requête motif `?x :price ?price`

SQL

```
select t.title, t.price
from (select t1.x, t1.title, t2.price as price
        from (q1) t1 left outer join (q2) t2 on (t1.x = t2.x)) t
```

ou

```
select t.title, t.price
from (select t1.x, t1.title, t2.price as price
      from (select subject as x, object as title
              from Triple where property=':title') t1
                  left outer join
           (select subject as x, object as price
              from Triple where property=':price') t2
                on (t1.x = t2.x)) t
```

OPTIONAL imbriqué

Requête optional5

```
PREFIX  :  <http://example.org/ns#>
SELECT ?title ?editor ?address
  FROM <bibliosem.ttl>
 WHERE { ?x :title ?title .
   OPTIONAL { ?x :editor ?editor
     OPTIONAL { ?editor :address ?address }}}
```

- q1 : requête motif `?x :title ?title`
- q2 : requête motif `?x :editor ?editor`
- q3 : requête motif `?editor :address ?address`

$$q1 ⟕ q2 ⟕ q3$$

SQL

```
select t.title t.editor t.address
from ( select t1.x, t1.title , t2.editor
         from q1 t1 left outer join
                (select t2.x, t2.editor , t3.address
                 from q2 t2 left outer join q3 t3
                      on (t3.editor = t2.editor) ) ta
              on (t1.x = t2.x) ) t
```

2 - Triple stores « SQL »

- Schémas de stockage génériques
- Schémas de stockage partitionnées
- Schémas de stockage étendus

Solution 2 : Tables partitionnées

Schéma de stockage **partitionné**

- une table par type de propriété : **Property$_i$(Subject, Object)**
- exemple : SW-Store [Abadi et.al. 07].

Avantages

+ import/export simple et rapide (triplet=nuplet)
+ représentation « naturelle » de propriétés multivalués et optionnelles
+ réutilisation de la technologie *column tables* (Vertica / Monet DB)
+ jointures efficaces : tables trié par le sujet et merge-join

Inconvénients

- beaucoup de jointures $\rightarrow$ column-store (MonetDB)
- nombre de tables importantes (beaucoup de « petites » tables)

Schéma spécifique : Tables partitionnées

Requête simple1

```
PREFIX  :  <http://example.org/ns#>
SELECT ?title ?price ?editor
FROM <bibliosem.ttl>
 WHERE { ?x :title ?title .
         ?x :editor ?editor .
         ?x :price ?price . }
```

SQL

```
select t1.title, t2.editor, t3.editor
  from title t1, editor t2, price t3
 where t1.subject = t2.subject
   and t2.subject = t3.subject
```

Solution 3 : Tables de propriétés

Schéma de stockage **partitionné**

- Regroupement (clustering) de sujets *partageant* des propriétés
- **Cluster$_i$(Sujet, Property$_1$, Property$_2$, ..., Property$_n$)**
  - une table par « cluster » de propriétés
  - chaque propriété apparaît dans une seule table.
- **Classe$_i$(Sujet, Property$_1$, Property$_2$, ..., Property$_n$)**
  - une table par classe RDFS
  - une propriété peut apparaître dans plusieurs tables.

Avantages

+ fragmentation des données en plusieurs tables
+ regroupement de propriétés qui sont souvent interrogées ensembles $\rightarrow$ moins de jointures

Inconvénients

- propriétés partagées entre tables génèrent des *unions*
- dénormalisation : propriétés multivalués ou optionnelles peuvent générer des valeurs NULL

Tables de propriétés : moins de jointures

Requête simple1

```
PREFIX  :  <http://example.org/ns#>
SELECT ?title ?price ?editor
FROM <bibliosem.ttl>
 WHERE { ?x :title ?title .
         ?x :editor ?editor .
         ?x :price ?price . }
```

Expression algébrique de simple1

```
(prefix ((: <http://example.org/ns#>))
  (project (?title ?price ?editor)
    (bgp
      (triple ?x :title ?title)
      (triple ?x :editor ?editor)
      (triple ?x :price ?price)
    )))
```

Schéma générique

```
select t1.object as title,
       t3.object as price,
       t2.object as editor
  from Triples t1, Triples t2,
       Triples t3
 where t1.property = ':title'
   and t2.property = ':editor'
   and t3.property = ':price'
   and t1.subject = t2.subject;
   and t1.subject = t3.subject;
```

Table de propriétés

```
select t1.title, t2.price, t1.editor
  from title_editor t1, price t2
 where t1.subject = t2.subject;
```

Tables de propriétés : plus d'unions

Requête simple2

```
PREFIX  :  <http://example.org/ns#>
SELECT ?x ?price
FROM <bibliosem.ttl>
 WHERE { ?x :price ?price . }
```

Schéma générique

```
select t1.subject, t1.object as price,
  from Triples t1
 where t1.property = ':price'
```

Schéma spécifique

```
select t1.subject, t1.price
  from livre_prix t1
union
select t1.subject, t1.price
  from article_prix t1
```

2 - Triple stores « SQL »

- Schémas de stockage génériques
- Schémas de stockage partitionnées
- Schémas de stockage étendus

Exemple : Oracle RDF

- application du “Spatial Network Data Model” (Spatial NDM)
- graphe RDF = réseau logique
- deux packages :
  - SDO_RDF : manipulation de graphes RDF (modèles)
  - SDO_RDF_INFERENCE : inférence et règles

Schéma RDF étendu dans Oracle 10g

Deux types abstraits

Interface SQL :

- **SDO_RDF_TRIPLE** (affichage)
- **SDO_RDF_TRIPLE_S** (stockage)

Plusieurs tables annexes

- **RDF_LINK$(LINK_ID, P_VALUE_ID, START_NODE_ID, END_NODE_ID, LINK_TYPE, ACTIVE, CONTEXT, REIF_LINK, MODEL_ID)**
- RDF_MODEL$(MODEL_ID, MODEL_NAME)
- RDF_NAMESPACE$(NAMESPACE_ID, NAMESPACE_NAME)
- RDF_VALUE$(VALUE_ID,VALUE_NAME, VALUE_TYPE, LITERAL_TYPE) :
  - les URIs et valeurs sont stockées dans une table séparée
  - VALUE_TYPE : URI, blank node, literal (plain, typed)
- RDF_NODE$(NODE_ID, ACTIVE)
- RDF_BLANK_NODE$(NODE_ID, NODE_VALUE, ORIG_NAME, MODEL_ID)

(invisibles pour le développeur)

Utilisation du package SDO_RDF

Fonctionnalités principales

Génération d'un graphe RDF :

- création d'une table *T* avec un attribut de type SDO_RDF_TRIPLE_S pour stocker interroger des triplets RDF.
- création d'un modèle (graphe) RDF *G* : CREATE_RDF_MODEL : génération des tables annexes.
- insertion / suppression de triplets RDF dans la table *T* : INSERT / DELETE
- interrogation motifs SPARQL : requête SQL sur des tables générées par SDO_RDF_MATCH sur *G*

Autres fonctionnalités

gestion d'espaces de noms, collections, noeuds blancs, réification

Exemple : Insertion triplets

```
INSERT INTO family_rdf_data VALUES (1,
 SDO_RDF_TRIPLE_S('family',
   'http://www.example.org/family/John',
   'http://www.example.org/family/fatherOf',
   'http://www.example.org/family/Suzie'));

INSERT INTO family_rdf_data VALUES (20,
  SDO_RDF_TRIPLE_S('family',
                'http://www.example.org/family/Male',
                'rdfs:subClassOf',
                'http://www.example.org/family/Person'));

               INSERT INTO family_rdf_data VALUES (25,
  SDO_RDF_TRIPLE_S('family',
                'http://www.example.org/family/siblingOf',
                'rdf:type',
                'rdf:Property'));
```

Évaluation de Motifs de Graphes

Fonction SDO_RDF_MATCH

```
SDO_RDF_MATCH(
      query VARCHAR2,
      models SDO_RDF_MODELS,  rulebases SDO_RDF_RULEBASES,
      aliases SDO_RDF_ALIASES,
      filter VARCHAR2
   ) RETURN ANYDATASET;
```

Exemple

```
SELECT m
FROM TABLE(SDO_RDF_MATCH(
     '(?m rdf:type :Male)',
     SDO_RDF_Models('family'), null,
     SDO_RDF_Aliases(SDO_RDF_Alias('','http://www.example.org/family/')),
    null));
```

Réponse

M

---

http://www.example.org/family/Jack
http://www.example.org/family/Tom

Utilisation du package SDO_RDF_INFERENCE

SDO_RDF_INFERENCE

Package pour la définition et l'exécution de règles RDF.

Bases de règles par défaut (Oracle 11gR2)

- RDFS : RDF/RDFS entailment
- RDFS++ : RDFS et owl:InverseFunctionalProperty, owl:sameAs
- OWLSIF : RDFS++ et owl:FunctionalProperty, owl:SymmetricProperty, owl:TransitiveProperty, owl:inverseOf, owl:equivalentClass, owl:equivalentProperty, owl:hasValue, owl:someValuesFrom, owl:allValuesFrom
- OWLPrime : OWLSIF + owl:disjointWith, owl:complementOf
- OWL2RL `http://www.w3.org/TR/owl-profiles/#OWL_2_RL`

Exemple : Interrogation avec Inférence

Règles utilisateurs

```
EXECUTE SDO_RDF_INFERENCE.CREATE_RULEBASE('family_rb');
INSERT INTO MDSYS.RDFR_FAMILY_RB VALUES(
        'grandparent_rule',
        '(?x :parentOf ?y) (?y :parentOf ?z)',
       NULL, '(?x :grandParentOf ?z)',
       SDO_RDF_Aliases(SDO_RDF_Alias('',
              'http://www.example.org/family/')));
```

Requêtes avec Inférence

```
SELECT m FROM TABLE(SDO_RDF_MATCH('(?m rdf:type :Male)',
   SDO_RDF_Models('family'),
   SDO_RDF_Rulebases('RDFS'),
   SDO_RDF_Aliases(SDO_RDF_Alias('',
     'http://www.example.org/family/')), null));
```

Outline

3 - Systèmes RDF « noSQL »

- RDF et noSQL
- Systèmes RDF/noSQL
- RDF/SPARQL en MapReduce

Systèmes **n**ot-**o**nly SQL

Données RDF

- données volumineuses, complexes et faiblement structurées
- requêtes complexes (motifs de graphes)
- cohérence faible (données web, schéma = description)

Systèmes « SQL »

- structuration forte : schéma = contraintes
- algèbre physique complexe $\rightarrow$ distribution et parallélisation difficile
- gestion de cohérence coûteuse (ACID)

Systèmes « noSQL »

- favoriser la scalabilité et la disponibilité
- structuration faible : schéma = description et validation des données
- cohérence à terme (eventual consistency) au lieu de cohérence forte (lecture/écriture)

Les systèmes noSQL sont bien adaptés pour le stockage et l'interrogation de très grands graphes RDF.

3 - Systèmes RDF « noSQL »

Column Stores

Modèle et systèmes

- tables avec familles de colonnes / couples (clé,valeur) imbriqués
- indexation distribué
- Bigtable/Google, Hbase/Apache, Cassandra/Apache, SimpleDB

Avantages et Inconvénients

+ favorisent la cohérence et la tolérance aux pannes
- complexité de la mise-en-oeuvre

Column Stores RDF

- schémas de stockage RDF génériques et partitionnés
- [20] Hexastore + HBase, CumulusRDF [21] (Cassandra), Stratusstore [22] (SimpleDB)

Document Stores

Modèle

- collection de documents *indépendants* et identifiés par des clés
- document = arbre étiqueté JSON
- Lotus notes, mongodb/10gen, couchDB/Apache, Terrastore, JSON stores

Avantages et Inconvénients

+ contrôle fin de la fragmentation de données (sharding) $\rightarrow$ partitionnement de graphes RDF
- fragmentation à la charge du développeur

Document Stores RDF

- dataset RDF = ensemble de documents JSON
- rdf-couchdb, MongoDB-RDF

Graph Stores

Modèle

- graphes de données dirigés et étiqueté
- Neo4j, InfiniteGraph, Trinity (Microsoft), FlockDB

Avantages et Inconvénients

+ implémentation « directe » du modèle graphe RDF : jointure par navigation, expressions de chemins
- parallélisation compliquée (partitionnement de graphes)

Graph Stores RDF

- dataset RDF = ensemble de graphes de données
- Neo4J (support natif de SPARQL), gStore, AMBER

3 - Systèmes RDF « noSQL »

- RDF et noSQL
- Systèmes RDF/noSQL
- RDF/SPARQL en MapReduce

Performance et passage à l'échelle

Objectif : Scalabilité

Implémenter des systèmes RDF qui sont capables d'adapter « *en proportion* » leurs besoins en ressources aux demandes (requêtes) et aux volumes de données RDF à gérer.

Scalabilité $\Leftrightarrow$ distribution et parallélisation

- Partitionnement et distribution des données :
  - schémas de stockage partitionnés (hash, graph)
  - indexation distribuée
- Évaluation parallèle de requêtes (jointures)

Problème d'optimisation

Deux problèmes équivalents :

- Minimiser l'échange de données (shuffling) entre les noeuds
- Maximiser la localité des données par rapport aux traitements

Data Parellelism and Map-Reduce

Data parallelism

Separate task on a big dataset into a set of synchronizes parallel tasks on smaller subsets.

Map-Reduce Program

Consecutive steps executed by m mappers M_i and r reducers R_j :

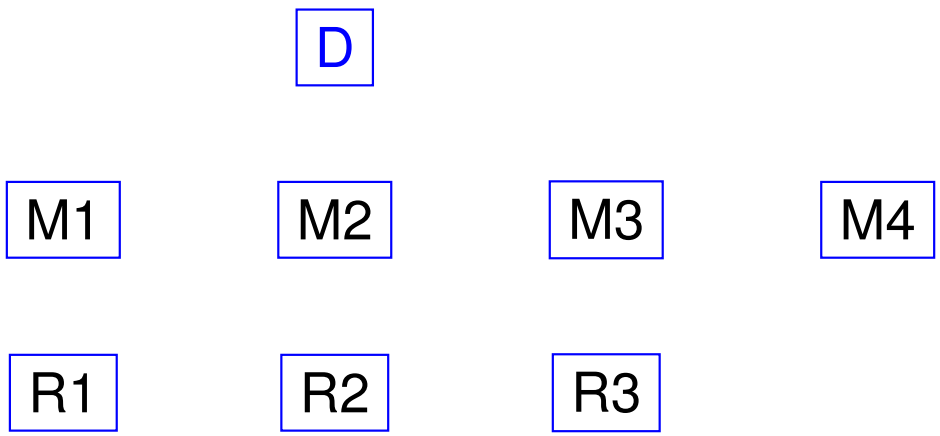
- **initialization** : data set D is distributed to m mappers M_i
- **map** : each mapper M_i **partitions** and **sorts** its local data set (tuples) D_i according to a **key-value** k to generate partitions $P_i(k)$.
- **shuffle** : all partitions $P_i(k)$ with the same key-value k are sent to the same reducer R_j.
- **reduce** : each reducer R_j aggregates for some given k the partitions $P_i(k)$ of all mappers.

Each **node** can play the role of one or several mappers and/or one or several reducers.

Map-Reduce Example

Consecutive steps executed by 3 mappers M_i and 3 reducers R_j :

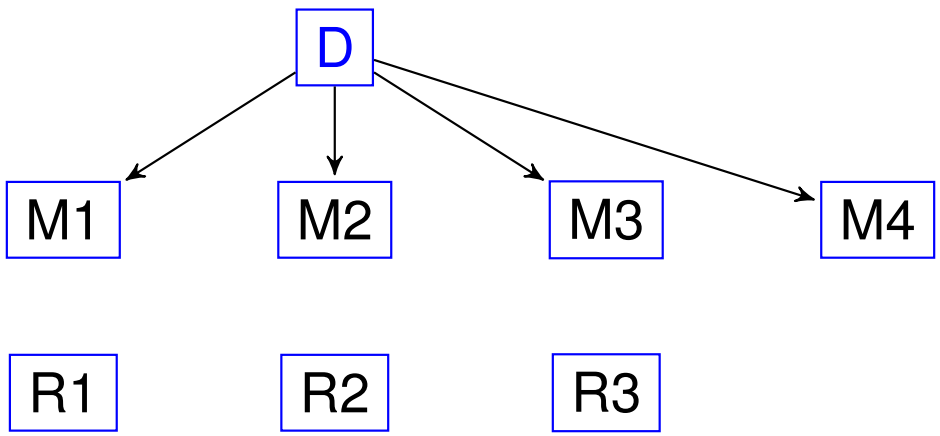
- **initialization** : data set D is distributed to m mappers M_i
- **map and shuffle** : all mappers use the same partitioning scheme and send all tuples t in partition v where
- **reduce** : each reducer R_j aggregates for some given k the partitions $P_i(k)$ of all mappers.



Map-Reduce Example

Consecutive steps executed by 3 mappers M_i and 3 reducers R_j :

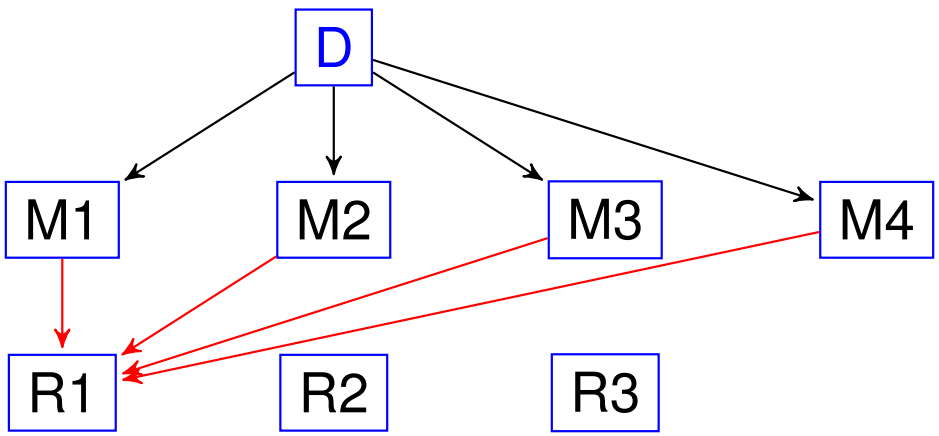
- **initialization** : data set D is distributed to m mappers M_i
- **map and shuffle** : all mappers use the same partitioning scheme and send all tuples t in partition v where
- **reduce** : each reducer R_j aggregates for some given k the partitions $P_i(k)$ of all mappers.



Map-Reduce Example

Consecutive steps executed by 3 mappers M_i and 3 reducers R_j :

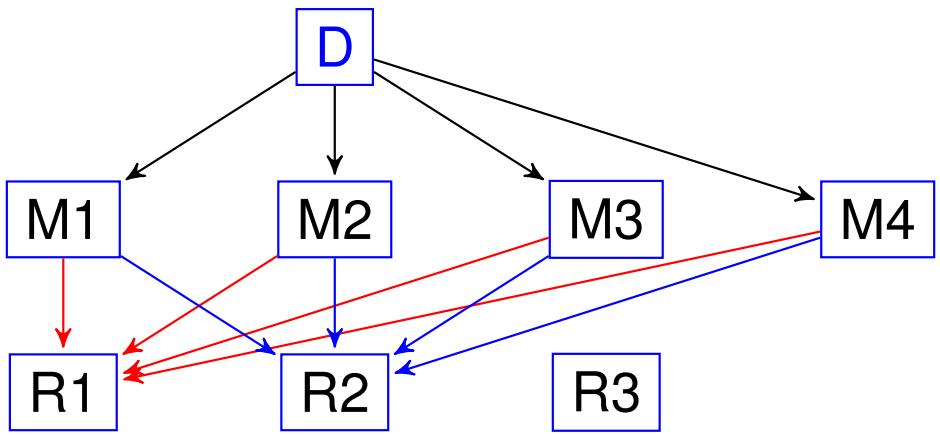
- **initialization** : data set D is distributed to m mappers M_i
- **map and shuffle** : all mappers use the same partitioning scheme and send all tuples t in partition v where $v \ mod \ 3 = 0$ to reducer $R1$,
- **reduce** : each reducer R_j aggregates for some given k the partitions $P_i(k)$ of all mappers.



Map-Reduce Example

Consecutive steps executed by 3 mappers M_i and 3 reducers R_j :

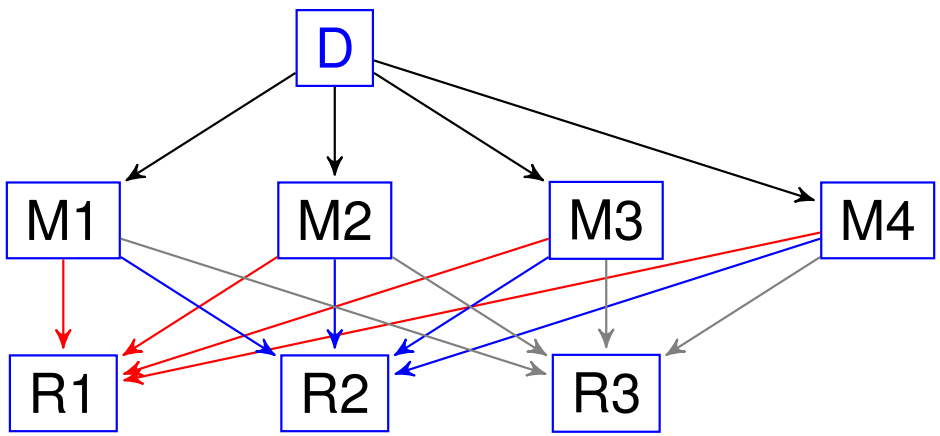
- **initialization** : data set D is distributed to m mappers M_i
- **map and shuffle** : all mappers use the same partitioning scheme and send all tuples t in partition v where $v \ mod \ 3 = 0$ to reducer $R1$, $v \ mod \ 3 = 1$ to reducer $R2$
- **reduce** : each reducer R_j aggregates for some given k the partitions $P_i(k)$ of all mappers.



Map-Reduce Example

Consecutive steps executed by 3 mappers M_i and 3 reducers R_j :

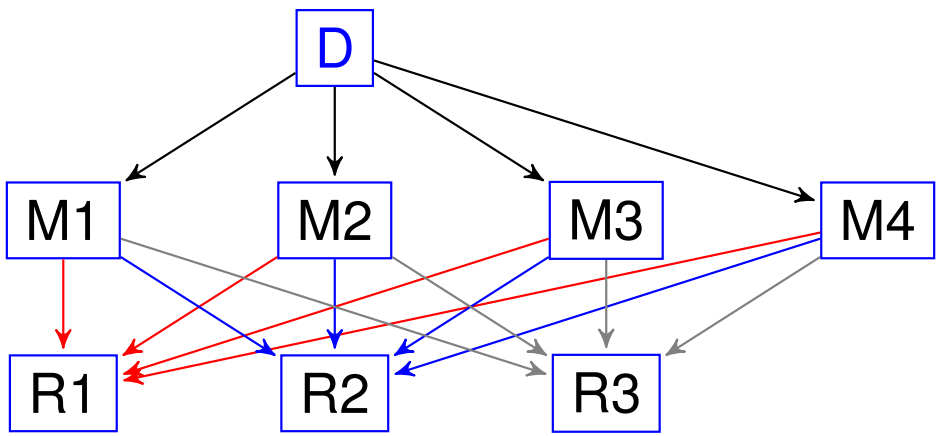
- **initialization** : data set D is distributed to m mappers M_i
- **map and shuffle** : all mappers use the same partitioning scheme and send all tuples t in partition v where $v \ mod \ 3 = 0$ to reducer $R1$, $v \ mod \ 3 = 1$ to reducer $R2$ and $v \ mod \ 3 = 2$ to reducer $R3$.
- **reduce** : each reducer R_j aggregates for some given k the partitions $P_i(k)$ of all mappers.



Map-Reduce Example

Consecutive steps executed by 3 mappers M_i and 3 reducers R_j :

- **initialization** : data set D is distributed to m mappers M_i
- **map and shuffle** : all mappers use the same partitioning scheme and send all tuples t in partition v where $v \ mod \ 3 = 0$ to reducer $R1$, $v \ mod \ 3 = 1$ to reducer $R2$ and $v \ mod \ 3 = 2$ to reducer $R3$.
- **reduce** : each reducer R_j aggregates for some given k the partitions $P_i(k)$ of all mappers.



Map-Reduce Joins and Graph Patterns

Map-Reduce and RDF

- RDF data set can be partitioned and distributed over a given set of computation **nodes**
- Graph pattern $\sim$ many joins

Observations

- Map-Reduce step naturally implements a « join » on the partition keys.
- joins with Map-Reduce is a well-known problem ;
- two main join processing strategies : partitioned join and broadcast (replicated) join

Partitioned Triple join

Join $t \bowtie_{?x} t'$

- **initialization** (executed once) : data set D is distributed to the mappers M_i.
- **map** : partition key k = binding of $?x$: partition triples matching patterns t and t' according to the **bindings** k of variable $?x$ to generate partitions $P_i(t,k)$ and $P_i(t',k)$.
- **shuffle** : send all partitions $P_i(t,k)$ and $P_i(t',k)$ with the same key-value k are the same reducer R_j.
- **reduce** : reducer R_j **joins** all tuples in partitions $P_i(t,k)$ and $P_i(t',k)$ received from all mappers M_i for a given join value k.

Data transfer cost (shuffle) : $\leq O(b+b')$ where b and b' is the number of binding tuples for patterns t and t'.

Partitioned Triple Join : Example

Example

t=(?x :p :r1), t'=(?a :q ?x)

- **initialization** (executed once) : data set D is distributed to the mappers M_i.
- **map and shuffle** : all triples matching patterns (?x :p :r1 and (?a :q ?x) by the same binding for ?x are sent to the same reducer which computes the join.
- **reduce** : reducer R_j **joins** all tuples in partitions $P_i(t,k)$ and $P_i(t',k)$ received from all mappers M_i for a given join value k.

Cost depends on data locality (**initialization** step)

For example, if all triples with the **same subject** s are distributed to the same mapper node $M(s)$ and each mapper node $M(s)$ is also the **reducer** $M(s) = R(s)$ of the triples with subject $s \Rightarrow$ **star** queries are executed *locally* (without any data transfert / shuffle cost)

Broadcast join

Join $t \bowtie_{?x} t'$

- **initialization** (executed once) : data set D is distributed to the mappers M_i.
- **map** : each mapper M_i partitions the triples matching patterns t according to the **bindings** k of variable $?x$ to generate partitions $P_i(t,k)$ and $P_i(t',k)$.
- **shuffle** : each mapper M_i broadcasts the partition corresponding to the smaller data set, say $P_i(t,k)$, to all computation nodes (reducers).
- **reduce** : each reducer R_j **joins** all received partitions $P_i(t,k)$ bindings with its local partition $P_j(t',k)$.

Data transfer cost (shuffle) : $O(b * n)$ where b is the size of bindings for t and n is the number of computation nodes.

Example

t=(?x :p :r1), t'=(?a :q ?x)

- for a given node (mapper) N, all bindings produced by N for triple pattern t (small) are sent to all nodes (reducers) which compute the join.

Example : RDF-3X + Hadoop [9]

Data distribution

- partition RDF graph : METIS graph partitioner
- each Hadoop node stores one or several partitions in a local RDF-3X instance
- (partial) replication to reduce inter-machine communication (N-hop guarantee)

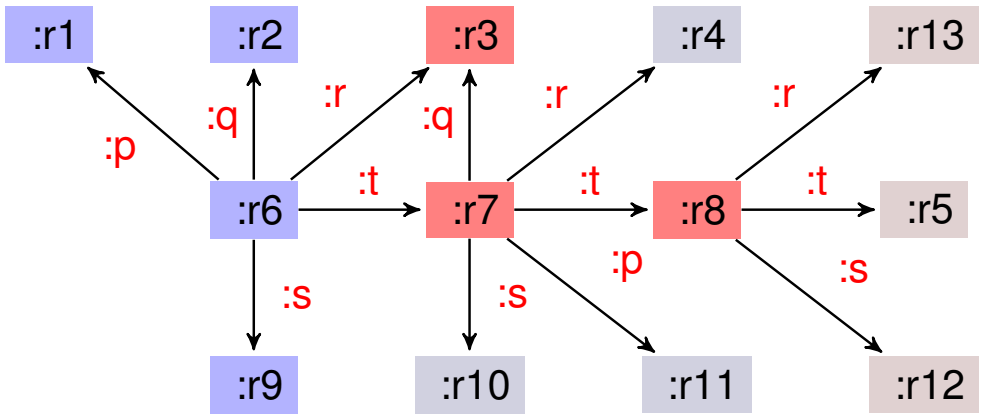
Parallel query processing

Query is parallelizable without communication ?

- yes : result = union of locally evaluated sub-queries
- no :
  1. decompose query pattern into subqueries (minimal edge partionining of query graph)
  2. evaluate subqueries
  3. integrate subquery results globally

RDF-3X + Hadoop : Node Cut Partitioning

Graphe RDF

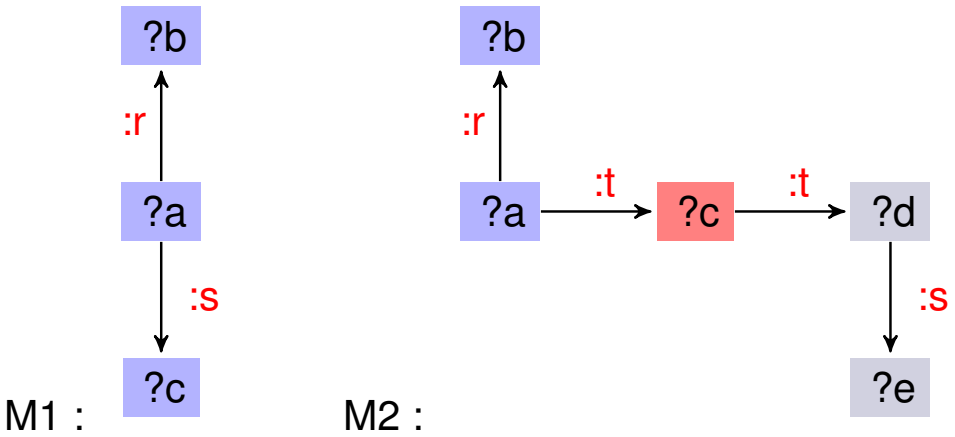


- trois partitions :

p1 p2 p3

- les noeuds repliqués sont en rouge (1-hop replication).

Motifs SPARQL



- M1 : évaluation localement dans les trois partitions et union des mappings obtenus
- M2 : décomposition en sous-motifs qui sont évalués localement ; leurs réponses sont joints globalement.

Bibliographie I

1. D. Abadi, A. Marcus, S. Madden and K. Hollenbach. SW-Store : a vertically partitioned DBMS for Semantic Web data management. VLDBJ 18(2), 2009.
2. Marcelo Arenas, Jorge Pérez, Querying semantic web data with SPARQL, Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, June 13-15, 2011, Athens, Greece
3. Bornea, Mihaela A., Julian Dolby, Anastasios Kementsietsidis, Kavitha Srinivas, Patrick Dantressangle, Octavian Udrea, and Bishwaranjan Bhattacharjee. Building an Efficient RDF Store over a Relational Database. In Proceedings of the 2013 International Conference on Management of Data - SIGMOD '13, 121. New York, New York, USA : ACM Press, 2013.
4. Aluc, Günes, Tamer M. Ozsu, Khuzaima Daudjee, and Olaf Hartig. Executing Queries over Schemaless RDF Databases. Proceedings - International Conference on Data Engineering 2015-May (2015) : 807-818.

Bibliographie II

5 Chebotko, Artem, Shiyong Lu, and Farshad Fotouhi. Semantics Preserving SPARQL-to-SQL Translation. Data & Knowledge Engineering 68, no. 10 (October 2009) : 973-1000.

6 Goasdoué, François, Zoi Kaoudi, Ioana Manolescu, Jorge-Arnulfo Quiané-Ruiz, and Stamatis Zampetakis. CliqueSquare : Flat Plans for Massively Parallel RDF Queries. In 2015 IEEE 31st International Conference on Data Engineering, 771-782. IEEE, 2015.

7 C. Guttierez, C. Hurtado, A. Mendelzon, Foundations of Semantic Web Databases, PODS 2004

8 Harth, Andreas, Katja Hose, and Ralf Schenkel. Database Techniques for Linked Data Management. In Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, 597-600. ACM, 2012.

9 Huang et al : Scalable SPARQL Querying of Large RDF Graphs. VLDB 2011

Bibliographie III

10 Kaoudi, Zoi, and Ioana Manolescu. RDF in the Clouds : A Survey. The VLDB Journal 24, no. 1 (2015) : 67-91.

11 Andrés Letelier, Jorge Pérez, Reinhard Pichler, Sebastian Skritek, Static analysis and optimization of semantic web queries, ACM Transactions on Database Systems (TODS), v.38 n.4, p.1-45, 2013.

12 H. Naacke, O. Curé, B. Amann, SPARQL query processing with Apache Spark, arXiv preprint arXiv :1604.08903

13 T. Neumann and G. Weikum. RDF-3X : a RISC-style engine for RDF. VLDBJ 19(1), 2010.

14 Özsu, M. Tamer. A Survey of RDF Data Management Systems. Frontiers of Computer Science, 2016, 1-15.

15 Reinhard Pichler, Sebastian Skritek, Containment and equivalence of well-designed SPARQL, Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, June 22-27, 2014, Snowbird, Utah, USA.

Bibliographie IV

16 Schätzle, Alexander, Martin Przyjaciel-Zablocki, Simon Skilevic, and Georg Lausen. S2RDF : RDF Querying with SPARQL on Spark. arXiv Preprint arXiv :1512.07021, 2015. http ://arxiv.org/abs/1512.07021.

17 Schmidt, Michael, Michael Meier, and Georg Lausen. Foundations of SPARQL Query Optimization. In Proceedings of the 13th International Conference on Database Theory - ICDT '10, 4. New York, New York, USA : ACM Press, 2010.

18 Petros Tsialiamanis, Lefteris Sidirourgos, Irini Fundulaki, Vassilis Christophides, Peter Boncz, Heuristics-based query optimisation for SPARQL, Proceedings of the 15th International Conference on Extending Database Technology, March 27-30, 2012, Berlin, Germany

19 Lei Zou, M. Tamer Özsu, Lei Chen, Xuchuan Shen, Ruizhe Huang, Dongyan Zhao, gStore : A Graph-based SPARQL Query Engine, VLDB Journal, 23(4) : 565-590, 2014.

20 Sun et al. Scalable RDF Store Based on HBase and MapReduce. ICACTE 2010

Bibliographie V

21 G. Ladwig et al. CumulusRDF : Linked Data Management on Nested Key-Value Stores. SSWS 2011

22 R. Stein et al : RDF On Cloud Number Nine. NeFoRS 2010

23 Kaoudi, Zoi, and Ioana Manolescu. Cloud-Based RDF Data Management. In Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, 725-729. ACM, 2014.