Dynamic Materialized View Selection during Near Real Time Semantic ETL Process

Author1
Institute for Clarity in Documentation
1932 Wallamaloo Lane
Wallamaloo, New Zealand
trovato@corporation.com

Author2
Institute for Clarity in Documentation
P.O. Box 1212
Dublin, Ohio 43017-6221
webmaster@marysville-ohio.com

Abstract

Nowadays, the big data era augments the variety and complexity of data sources. They become extremely heterogeneous in their structure, with considerable variety even for substantially similar entities. Small, medium and large companies need to integrate, fuse and link their data in order to achieve a high decision making. To satisfy these requirements, active data warehouses is one of the promising technology that has to be intelligently used. It aims at decreasing the time it takes to make decisions and try to attain zero latency. This is a critical task regardless of their workloads, and completely needs of tuning by exhaustive selection of optimized structures such as: indexes and materialized views. The emergence of the semantic RDF data has led to the creation of large semantic data warehouses. Improving their performance necessitates the selection of optimized data management structures. In this paper, we first propose to dynamically suggest a set of materialized views based on a workload of SPARQL queries. Secondly, an algorithms for orchestrating the ETL flows considering materialized views selection is given. Finally, our finding is validated through a case study and an intensive experimentations using the LUBM benchmark and deployed in Oracle RDF Semantic Graph 12c.

Keywords

RDF, SPARQL, Materialized view selection, Near real-time data warehousing, Semantic ETL Process.

1. Introduction

2. Related Work

In the following, we discuss those works that are most related to our main contribution, i.e., RDF materialized view selection and near real time ETL process.

2.1 RDF Materialized View

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2017 ACM. ISBN 123-4567-24-567/08/06...$15.00
DOI: 10.475/123_4

Few works have explored the selection of materialized views of semantic databases. They aim to facilitate and provide efficient processing of RDF queries. Neumann and al. [29, 30] have proposed an RDF-3X engine system which implements a solution for storing and indexing RDF triples. It includes a query optimizer for choosing a join orders using a cost based model. x-RDF-3X [31] is an extension added to the system for online updates and transactions. The system is considered one of the most efficient RDF data management platforms. The drawback of this approach is the creation of redundant indexes. In fact, in case of large data processing, the size of indexes will increase required storage and impact response time. Castillo and al. [10, 9] have proposed an indexing approach (RDFMatView) for RDF Data using materialized SPARQL queries. This work is based on [11] method for classical databases. The authors selects views from triple table for a given SPARQL workload. They discover patterns of shared triples to be used as indexes to improve response times. Then they consider a set of queries as an initial search space and extend it by analyzing each load request and clearing all combinations of related triple patterns of a certain length. They add them to this search space called the candidate index space. Goasdoue and al. [13, 14] examines materialized views based on a cost model and a set of user defined queries. They have proposed an algorithm to identify a set of candidate views for materialization that take in account entailment implicit triples and supporting query rewriting. Mbaiossoum and al. [26] have defined two approaches to select materialized views. The first one hides the implementation aspects by selecting views at the ontological level using a rule-based approach. The second approach selects views at the logical level based on a cost model which considers the diverse storage layouts that can be used.

Analysis of state of the art has shown that all approaches consider a static selection of materialized view which contradicts the dynamic nature of the web. Indeed, any change to the workload should be reflected to the view selection as well. Moreover, there are no work that have used the RDF materialized view results. Indeed, [1] seems to be the only paper which deals with materialized view result of an RDF analytical query. In fact, it does not provide any detail on how to select an effective set of RDF views to materialize and it is applicable for the subsequent queries and not to an arbitrary set of queries. Unlike to our work, none of these attempts aims at using materialized view selection in a real scenarios where the client needs to process the queries even without access to the database. Our contribution demon-
strates how RDF materialized views can be selected dynamically for active large data warehouses satisfying RDF Web data changes.

A comparison including existing works and our proposal is given in Table 1 based on five criteria: (1) the nature of the workload queries (NQW), (2) the data structure used to manipulate input queries (DS), (3) the reordering of queries depending on requirement selection (RQ) (4) Type of Optimization structure used (OS) (5) The use of Materialized view results (UMVR).

### 2.2 Near real time ETL process

The use of semantic web technology in data warehouses and related areas has been widely addressed in recent years by the research community. For example in: (i) data warehouse conceptual design [34, 27, 3, 17]: to derive a Multidimensional (MD) schema, this aim to hide the heterogeneity by the identification of functional dependencies and MD identifiers and then annotate the findings in a reference ontology; (ii) ETL conceptual design [38, 36, 28, 34, 5]: using ontology concepts and reasoners capabilities to identify transformations needed to automate ETL process; and (iii) On-Line Analytical Processing (OLAP) [32, 2, 41, 33]: to discover and query external data most often on the web resulting in exploratory OLAP.

Most of the research related to ETL process deals with the construction of a conceptual design that identifies the data sources and describes the corresponding data transformations needed to map these sources to the target Semantic DW concepts. This is at large an open problem, the proposed approaches aim to facilitate and to automate the design and construction of the ETL part. However, we noticed that no work have been proposed to deal with the real time execution of Semantic ETL process.

In the following, we summarize a brief overview of the works that dealt with classical near real time ETL process. Different solutions have been proposed to enhance the speed and accuracy of the existing data. Some of them have worked on materialized views refreshment [42, 16]. Other one proposed to eliminate duplicated data using equality and similarity features [35]. [19] have used a queue network where an Active Data Staging Area (ADSA) is built between the source system and data warehouse. The ETL transformations are done by choosing the right topology and communication methods. Change Data Capture have been addressed to deal with heterogeneous data source integration [6]. Authors in [40] dealt with the ETL workflow optimization that includes intelligent control module. It automatically allocates job which allow content enrichment using the INLJ (index nested loop join) on the table. Authors in [7] focus on the speed arrival of stream data and the Input/Output access to the disk. They proposed some algorithms that manages disk and memory access and uses MESH join operations in transformation phase that is faster compared to other joins. In the same perspective, [37] have proposed an algorithm that uses Semi-Streaming Index Join (SSIJ) during join operations in dynamic nature, it manages the memory and the incoming stream. Other algorithms were proposed for specific tasks including the resumption from failure and the incremental loading of the warehouse [21, 22, 25] have proposed a solution for data partitioning using quality metrics that makes a tradeoff between various objectives (performance, fault tolerance, reliability, recoverability, freshness). To the best of our knowledge, there are no works that have dealt with Dynamic MV selection during Semantic ETL process.

### 3. PRELIMINARIES

In this section, we present some concepts to facilitate the understanding of our approach that considers RDF data and SPARQL queries to generate the view candidates.

#### 3.1 RDF and SPARQL

RDF² is a graph-based model accepted as the W3C standard for Semantic Web applications. An RDF graph is a set of triples of the form $<s,o,p>$. A triple states that its subject $s$ has the property $p$, and the value of that property is the object $o$. They are used to describe resources, and property assertions. The RDF standard provides a set of built-in classes and properties, with predefined name-spaces. For example rdf:type specifies the classes to which a resource belongs. An example of RDF graph is depicted in Fig. 1.

![Figure 1: An example of RDF Graph for a portion of LUBM ontology](image)

A finite set of RDF triples is an RDF Graph in which every triple describes a directed edge labeled with $p$ from the node labeled with $s$ to the node labeled with $o$. Subjects $s$ can be URIs or blank nodes, properties $p$ are URIs, while objects $o$ can be URIs, blank nodes, or literals (i.e., values). From a database perspective, blank nodes can be seen as existential variables in the data. RDF triples referring to the same blank node may be joined to construct complex results.

RDF Schema (RDFS³) enriches the descriptions in RDF graphs, by declaring semantic constraints between the classes and properties used in the graph. Such constraints can define subClass, subProperty relations, domain and range of properties. The RDFS constraints are interpreted under the open-world assumption (OWA).

SPARQL³ is used to express RDF queries and views, we consider the basic graph pattern queries of sparql, represented by a set of conjunctive queries (atoms, variables or constants). Blocks of a SPARQL query are triple patterns that can contain variables. A basic graph pattern is made of a set of triple patterns. Outcome, an RDF view is a conjunctive query over the triple table $T: <s,o,p>$. Fig. 2 illustrates an example of sparql query from 14 queries of LUBM bench and looks for Students taking Courses at the Faculty.

#### 3.2 Materialized view selection

1. [https://www.w3.org/RDF/](https://www.w3.org/RDF/)
2. [http://www.w3.org/TR/rdf-schema/](http://www.w3.org/TR/rdf-schema/)
3. [www.w3.org/TR/rdf-sparql-query/](www.w3.org/TR/rdf-sparql-query/) , 2008
In this section, we present some concepts to facilitate the understanding of our approach.

Materialized views (MV) are one of the most popular optimization structures used in several databases deployment platforms: centralized [15], distributed [12], Cloud [18]. Historically, the Views Selection Problem (VSP) has been formalized as follows [15]: given a set of most frequently used queries \( Q = \{ q_1, q_2, ..., q_n \} \), where each query \( q_i \) has an access frequency \( f_i \) \( (1 \leq i \leq n) \) and a set of resource constraints \( CS = \{ C_1, ..., C_k \} \). The VSP consists in selecting a set of \( MV \) that minimizes one or more objectives, possibly subject to one or more constraints.

Materialized views for RDF Data consists on defining a system capable of choosing the most suitable views to materialize, in order to minimize the query response time for a specific SPARQL query workload, taking into account the view maintenance cost and storage space constraints.

Rewriting of queries. Let \( q_i \) be an RDF query of a given workload \( Q \) and \( V = \{ v_1, v_2, ..., v_n \} \) be a set of RDF views. A rewriting of \( q_i \) using \( V \) is a conjunctive query equivalent to \( q_i \), referencing to these views. The query rewriting is done by a query optimizer that uses a cost-based selection method to generate a best rewriting for a given query.

Candidate View. VSP relies on candidate view sets: Let \( Q \) be a set of RDF queries. A candidate view set for \( Q \) is defined by a set of RDF views \( V \) and a set of rewritings \( R \) where for each sparql query \( q \in Q \), there exists exactly one rewriting \( r \in R \) using the views \( V \).

RDF VSP formalization. VSP for RDF data can be formalized based on an existing proposal for selecting views to materialize in a relational setting [8], adapted to the particularities of the RDF model. For a given sparql query workload \( Q \), and a cost estimation function \( C \), the VSP for RDF data consists in finding a candidate view set \( (V, R) \) for \( Q \) such that, for any other candidate view set \( (V', R') \) for \( Q \): \( C(V, R) \leq C(V', R') \).

### 4. DYNAMIC MV DURING SEMANTIC ETL PROCESS

This section gives a comprehensive, scalable solution to the above problems. It presents a complete system, designed and implemented from scratch specifically for the management and selection of RDF data. The first section describes the different steps followed to achieve our goal. The second section depicts the \( DMV \) selection approach proposed. The third one presents the algorithms used during semantic ETL process.

#### 4.1 Motivating example

We adopt the example from the popular Lehigh University Benchmark\(^4\)(LUBM) to illustrate the idea and our motivations. Let us discuss the need for and advantage of dynamic materialized view selection during the semantic ETL process.

Given a semantic data warehouse deployed on a triple table integrating a set of RDF data sources. DW refreshment is done by means of a real time ETL process where transformations need to be performed continuously as update RDF instances in the warehouse. ETL transformation operations are commonly performed using ETL operators such as : extract, merge, join, aggregate, etc. Some of these ETL operators require the selection of DW instances to perform the transformation operations (join and aggregation). During the deployment phase, this scenario is implemented by translating the ETL operators into sparql queries having an RDF graphs, and executed over the DW triple table. Fig. 3 illustrates the described example.

Observing the same example from perspective of the solution shown in Fig. 3, parts c) and d). The join operation would not be performed since the result of it’s corresponding sparql query is materialized in the warehouse. The join operation cost is avoided. Although, materialization of the views generates a cost for refreshment and maintenance operations. This latter is insubstantial compared to the join and aggregation operations cost. In addition, the selection is carried out dynamically where the most relevant views are materialized, and ordered with a certain priority depending on their usage.

Here, we are motivated to decrease the join and aggregation operation cost by considering the fact that the RDF query results is already materialized and able to be selected as simple sparql graph operation (selection, projection). Therefore, we first materialize the relevant sparql queries and then execute the real time ETL process. Motivated by these observations, we introduce in the next section our proposal for active ETL process.

#### 4.2 Methodology description

Our methodology is focused on three major areas: (1) RDF data management; (2) Dynamic materialized view selection;

\(^4\)http://swat.cse.lehigh.edu/projects/lubm/
Near real time ETL process. It is mainly based on a very simple principle: new transformation operations that are applied on extracted data sources and require instances from data warehouse are done in memory using available dynamic materialized views. It is obvious and undeniable that this process is done faster since costly transformation operations (aggregation and join) are avoided.

The solution starts by integrating RDF data sources that are manipulated and interrogated using sparql queries. By means of an ETL process, the data are extracted from sources and transformed in order to be loaded in the warehouse. Three possible cases are identified: (i) offline transformations: data from the warehouse are not required, the transformation operations are applied in memory and data are loaded in the warehouse; (ii) Online standard transformations: data from the warehouse are required and are not available from materialized view, a disk access is required; (iii) Stream fast transformations: data from the warehouse are required and are available from selected materialized view. A set of aggregation and join operations are avoided.

We now describe in more detail the different components of the solution illustrated in Fig. 4.

Buffer of Inputs. Receives extracted instances from sources and send them to the Stream Buffer in case that ETL transformations requires data from DW. Otherwise, it merges instances and load them in the warehouse.

Buffer of Streams. Dealt with instances that requires retrieving one or more instances (Triple blocks) from DW, which means need to access to Materialized view cache or disk access, they are sent to memory cache for transformation.

Dynamic MV Cache. Allocated memory for MV selected. This cache contains a set of MV dynamically selected during ETL process. Views which are no longer used are evicted from memory.

4.3 Dynamic Materialized Views Selection for SPARQL Queries

In this section, we propose a solution that evaluates SPARQL queries in order to dynamically selects MV. We formally introduce all necessary concepts for this idea and describes the different components of the proposed solution which are: (1) Identifying mappings between SPARQL queries, (2) capturing of interaction among queries, (3) generation of views candidate and (4) query scheduling. Fig. 5 describes the different steps.

4.3.1 Identify mappings between SPARQL queries

This step is performed by identifying the different mappings between queries. It receives from users a set of SPARQL queries (workload Q) in a queue to be processed by a DBMS. It has to take into account similarity and equivalence of queries. The aim being to reduce the number of queries to be processed and then materialized. Let’s consider the following queries Q1 and Q2 that retrieves student information’s from a university LUBM data-set.

Listing Q1: Select all graduate students from Harvard university taking a course in either physics or mathematics.

```sql
SELECT ?student, ?course, ?university
```
WHERE {
  ?student rdf:type ub:Student .
  ?course rdf:type ub:Course .
  ?university rdf:type ub:University .
  {{ ?student ub:takesCourse ub:PhysicsCourse}
   ?student rdf:type :Student .
   WHERE { 
    SELECT ?student, ?course, ?university
    a course.
    ?course rdf:type ub:Course .
    ?university rdf:type ub:University .
  } } }

Listing Q2: Select all graduate students of university taking a course.

SELECT ?student, ?course, ?university
WHERE {
  ?student rdf:type :Student .
  ?student ub:takesCourse ?course .
  ?student ub:memberOf ?university
}

Figure 6: Matching sub-graphs of SPARQL queries

Our notion of mapping is based on the SPARQL Standard which is a pattern-matching query language. SPARQL allows the definition of graph pattern matching which produces a solution sequence, where each solution has a set of bindings of variables to RDF terms. The result of a query is a solution sequence, corresponding to the ways in which the query’s graph pattern matches the data. There may be zero, one or multiple solutions to a query. We provide here some necessary definitions, based on the W3C recommendation\(^5\).

**Definition 1.** Query containment and equivalence: A SPARQL query \(q_1\) is said to be contained in a query \(q_2\), denoted by \(q_1 \subseteq q_2\), if any set of triples generated from \(q_1\) is a subset of those generated from \(q_2\), i.e., matching queries \(q_1\) to \(q_2\) returns respectively two sub-graphs \(g_1\) and \(g_2\), where \(g_1\) is a projection of the variables of \(g_2\) into \(g_1\). Two queries are equivalent, \(q_1 \equiv q_2\), if \(q_1 \subseteq q_2\) and \(q_2 \subseteq q_1\).

**Definition 2.** Solution mapping is a mapping from a set of variables \(V\) to a set of RDF terms \(T\). \(\mu\) is a partial function \(\mu : V \rightarrow T\). The domain of \(\mu\) \(\text{dom}(\mu)\), is the subset of \(V\) where \(\mu\) is defined.

**Definition 3.** Solution sequence is a list of solutions, possibly unordered. It’s produced by a graph pattern matching where each solution has a set of bindings of variables to RDF terms. SPARQL provides a means of combining graph patterns so that one of several alternative graph patterns may match.

We start by considering variables from the SPARQL queries (workload \(Q\)) in order to identify query containment. Such strategy requires finding mappings between elements which means determine which pattern is contained in another query pattern. For that, we propose to use Algorithm 1 based on the provided definitions. Essentially, we find all mappings between each query and all patterns by enumerating all possible cases. Note that, for a given query, there potentially are many possible solution sequences. Afterward, we analyze which of them are most frequently used and we selects relevant ones.

**Algorithm 1** Searching of all mappings between queries using query containment.

**Input:** \(Q\): Query workload.
**Output:** \(M\): List of query mappings, \(Q'\): Query workload.

1: for Each \(q_i \in Q\) do
2: for Each \(q_j \in Q - \{q_i\}\) do
3: DISTINCT(\(q_i\)) / card(distinct(\(q_i\)))=1 (solution sequence of queries);
4: if \(D = \mu(\mu(q_a)) = q_3\) (containment of queries);
5: \(M_q := M_q \cup \{q_i\}\);
6: end if
7: if \(M_q \neq \emptyset\) then
8: \(Q' = Q' \cup \{q_j\}\);
9: end if
10: else
11: end for
13: end for
14: \(Q := Q - \{q_i\}\)
15: end for

4.3.2 Capturing of interaction among queries

From this step on, our dynamic materialized view selection method will be based on SLEMAS’s approach proposed in the context of relational databases [8]. Generally, interactions between queries are captured using acyclic graph [23]. The execution plan is then derived. Due to the strong interaction between queries, their plans may be merged to generate a graph structure called, unified query plan (UQP) [8].

SPARQL queries, by their nature, are represented by a graph [24]. This may facilitates the fusion of their individual query plans giving raise to a SPARQL Unified Query Plan. In fact, the execution plan of a SPARQL query is defined by an algebraic tree where the intermediates nodes represent join and aggregation operations while the leaf nodes represent the triples \(T <s,p,o>\).
Similarly to [8], we use the hypergraph data structure which have showed it's efficiency in Electronic Design Automation (EDA) [20]. The hypergraph vertices and edges are represented respectively by the join nodes and queries. Fig. 7 describe an example of query interactions applied to the 14 LUBM queries. The hyper-graph is created and partitioned according to the existing interactions among queries, using the HMETIS EDA tool\(^6\). Each partition is transformed to an acyclic directed graph. The result obtained is a UQP having as leaf nodes the triples \(<s,o,p>\) participating in the workload. The root node represent the final query results and the intermediate nodes are the SPARQL algebra operators (such as join, filter..) and solution modifiers (such as projection, distinct, limit, or order by).

![Figure 7: An Example of query interactions using the 14 SPARQL Queries of LUBM Benchmark.](image)

### 4.3.3 Materialized Views candidate

The use of materialized SPARQL views allow to speed up the execution of queries. During query processing, the system may decide which of the existing views to use. This task is performed by finding mappings between queries and generating the global query plan (UQP). In this perspective, a set of candidate views is derived building our search space.

Note that all nodes of the global plan are candidate for materialization which may represent a huge number. To decide which views from the candidate set are the most effective for the given workload, the system needs to evaluate all views and their influences on query processing. This decision should be made based on a cost model. The latter is expected to reduce the execution time by the usage of the appropriate views from the candidates ones. It takes into account the benefit of the nodes and their constraints related to their storage and maintenance. To do so, we define a cost function that returns a value between 0 and 1, that estimates the query processing.

We model the overall cost for the approach as follows:

- \(c(t) = c(s) \times c(p) \times c(o)\).
- \(\text{Cost}_{\text{W}O}(q_i, \Phi)\): the processing cost of the query \(q_i\) with out view(s).
- \(\text{Cost}_{\text{W}V}(q_i, V_j)\): the query processing cost of query \(q_i\) using the materialized view \(V_j\).
- \(\text{Cost}_{\text{Mat}}(V_j)\): the maintenance cost of the view \(V_j\).
- \(\text{Size}(V_j)\): the cost needed to store the view \(V_j\).

\(\text{Cost}\) represents the weight of view \(V_j\) and \(\text{Size}\) is the cost needed to store the view \(V_j\). The system needs to evaluate all views and their influences on query processing. This decision should be made based on a cost model. The latter is expected to reduce the execution time by the usage of the appropriate views from the candidates ones. It takes into account the benefit of the nodes and their constraints related to their storage and maintenance. To do so, we define a cost function that returns a value between 0 and 1, that estimates the query processing.

### 4.3.4 Query re-ordering

In this section, an ordering of queries based on [8] is done in order to avoid view dropping. All materialized views should optimize the maximum of queries before theirs dropping. The following steps are followed: (1) Identification of all nodes having maximal benefit; (2) An ordering is applied based on their benefit; (3) The benefit identified is propagated to the queries of those nodes. At this stage, each query will have a weight representing the sum of the benefit of its nodes. (4) The ordering of queries is done based on these weight. Afterwards, based on the cost model proposed above, we apply [8] algorithm to decide on materializing or dematerializing views.

### 4.4 Stream Processing ETL Algorithm

The algorithm consists of three phases, namely: (1) the Pre-processing phase which deal with transformations that do not require data warehouse instances; (2) Stream phase that apply transformations requiring DW instances, available as materialized views; and (3) Offline phases which manages transformations requiring DW instances, available only from disk. The refresh of the views selection is also performed in this phase. Fig. 8 describes an example of an ETL flow generated using the proposed ETL algorithm involving the different ETL processing phases.

We consider as starting point some RDF graphs defining data sources and target data warehouse. The main goal is then to define an appropriate set of rules, determining how the flow of ETL operations from a source to a target nodes can be executed. Essentially, each rule is responsible for executing an operator in the ETL process. The finally obtained graph is the DW RDF graph resulting from the integration of data sources. Our ETL process uses 10 generic conceptual ETL operators defined in [38]. We have overload them to consider the characteristics of RDF graphs. We redefined their signatures in order to satisfy our requirement, which is manipulating the RDF data structure. The data sources and target data-warehouse are represented by the RDF graph \(G\), where the nodes \(N\), edges \(E\) and labels \(L\) represents respectively classes, instances and data properties, object properties and DL constructors. The signature of each the ETL operator is defined as follow:

- \(\text{Retrieve}(G, n_j, L_j)\): retrieves a node \(n_j\) having an edge labeled by \(L_j\) of \(G\).
Algorithm 2 Pre-processing ETL Algorithm

**Input:** $S$: Local sources (SPARQL having Graph structure), $DW$: schema + instances (RDF Graph structure)

**Output:** $DW$ as graph

1: $c := 0$
2: for Each $S$, do
3: read stream triples;
4: $gS := $ExtractGraph$(S)$;
5: Put $gS$ in InputBuffer;
6: for Each graph $g \in$ InputBuffer do
7: if data does not need $DW$ instances then
8: Merge operations and output in $G_{ETL}$;
9: Clean and Store instances in DW;
10: else
11: if (END_OF_STREAM is true) then
12: Move subgraph to StreamBuffer zone;
13: Increment counter $c$ with number of stream sub-graph;
14: else
15: Go To Stream ETL Algorithm;
16: end if
17: end if
18: end for
19: Remove graph $g$ from cache InputBuffer;
20: end for

- **Extract**($G, n_j, CS$): extracts, from $G$, the node $n_j$ satisfying constraint $CS$.
- **Convert**($G, G_T, n_i, n_j$): converts the format of the node $n_i \in G$ to the format of the target node $n_j \in G_T$. The conversion operation is applied at instance level.
- **Filter**($G, n_i, CS$): is applied on the nodes $n_i$ and allows only the part satisfying constraint $CS$. Filter can also be applied on instance level based on defined axioms.
- **Merge**($G, n_j, I_j$): adds instances $I_j$ as nodes $n_j$ in same graph $G$.
- **Union**($G, G_T, n_i, n_j, E_j$): links nodes that belongs to different graphs. It adds edge $E_j$ that link the node $n_i \in G$ to the node $n_j \in G_T$ in the target graph $G_T$.
- **Join**($G, G_T, n_i, n_j, E_j$): joins instances whose corresponding nodes are $n_i \in G$ and $n_j \in G_T$. They are linked by an object property defined by an edge $E_j$.
- **Store**($G_T, n_j, I_j$): associates instances $I_j$ to the nodes $n_j \in G_T$ added to the target graph $G_T$.
- **DD**($G_T, CS$): starts by sorting the graph $G_T$ based on constraint $CS$ and detects nodes associated to duplicated instances.
- **Aggregate**($G_T, n_i, Op$): aggregates instances represented by the nodes $n_i$.

4.4.1 Pre-processing phase

During this phase, the algorithm starts by extracting triples nodes in the format of RDF graph (instances) from the RDF data sources participating in the integration process. These data are moved to the Input Buffer for a pre-processing. In case of extracted instances does not require transformations making use of meta-data or data from data warehouse, the instances are moved to Memory cache zone for transformations and loading. Otherwise, instances are moved to Stream Buffer for online processing. The ETL operators concerned by this processing are: retrieve, extract, convert, merge, filter, Delete Duplicate (DD), Store. The algorithm 2 presents the different steps applied during this phase.

4.4.2 Streaming phase (Online)

In the Online phase, the algorithm processes the existing RDF sub-graphs (instances) in the Stream Buffer. It waits for a minimum of instances threshold to accumulate (until to receives the message END_OF_STREAM). Then, it applies the transformation operations that requires data from the warehouse. Indeed, the algorithm checks the availability of the instances precomputed in views already materialized; ie join and aggregation operations precomputed through the materialized view dynamically selected. Otherwise, it moves to offline phase where required instances will be extracted from the disk and then computed. The ETL operators concerned by this processing are: Union, Join, Aggregate, Store. The algorithm 3 presents the different steps applied during this phase.

Algorithm 3 Stream ETL Algorithm

**Input:** Stream Buffer zone, $DW$ (schema and instances), number of stream sub-graph, value_limit: max memory size of stream buffer.

**Output:** $DW$ as graph

1: $c$=number of stream sub-graph;
2: if $c$ reach value_limit then
3: Send End_of_stream =false;
4: else
5: for Each graph $g_i$ in StreamBuffer do
6: check the availability of instances required in MV;
7: if Materialized view available then
8: Apply ETL transformations (join, aggregate, Union);
9: Output the $G_{ETL}$ after apply ETL operator;
10: Clean and Store instances in DW;
11: Remove graph g from cache StreamBuffer;
12: else
13: Move subgraphs to in-Memory zone;
14: Go to Offline Phase Algorithm;
15: end if
16: end for
17: end if

4.4.3 Offline phase

In this phase, the algorithm deal with existing RDF sub-graphs in Memory Cache. Transformations involve expensive joins between the newly arrived instances and some warehouse data or metadata. So, it selects instances required from data warehouse disk. Then, it applies the trans-
formation operations. At the end, the materialized view selection algorithm is executed in order to refresh the existing views and selects and/or evicts new ones. The algorithm 4 presents the different steps applied during this phase.

Algorithm 4 Offline ETL Algorithm

Input: in-Memory zone, $\mathcal{DW}$ (schema and instances).
Output: $\mathcal{DW}$ as graph

1: for Each graph $\mathcal{g}_i$ in in-MemoryCache do
2: Read required blocks from disk
3: Apply ETL transformations (join, aggregate, Union)
4: Output the $G_{ETL}$ after apply ETL operator;
5: Clean and Store instances in DW;
6: Remove graph $\mathcal{g}$ from cache in-MemoryCache;
7: end for
8: Refresh Materialized view selection (selects/evicts);

5. CASE STUDY

Modern business applications are running in dynamic data environments where the data are growing and changing constantly. In such environments, active RDF Data Warehousing has become a new trend for decision making where RDF data are updated as frequently as possible, due to the high demands of users for fresh data. To demonstrate the feasibility of our proposal, we present in the following a case study based on the university domain. We consider a research center designed by the higher education Ministry, to collect various searches data such as: scientific publications, PhD theses in progress, ongoing masters, research projects, etc. In this perspective, the research center integrates all existing data sources by mean of an active ETL process to feed a central data warehouse with a high level of freshness. The data sources concerned by this process are the universities, research laboratories as well as the semantic knowledge bases available on the web containing such kind of data, for example: dblp, yago, Wikipedia, etc.

The goal of the ETL process is to populate the target DW schema, represented by an Integrating Ontology (IO). The IO is defined using our previous work [5] from the global ontology, LUBM schema in our context. Let assume the IO is defined using our previous work [5] from the global schema, represented by an Integrating Ontology (IO). The example: dblp, yago, Wikipedia, etc.

bases available on the web containing such kind of data, for ties, research laboratories as well as the semantic knowledge The data sources concerned by this process are the universi-

feed a central data warehouse with a high level of freshness.

PhD theses in progress, ongoing masters, research projects,
collect various searches data such as: scientific publications,
scientific publications, PhD theses in progress, ongoing masters, research projects,
extalents of users for fresh data. To demonstrate the feas-

bility of our proposal, we present in the following a case study based on the university domain. We consider a research center designed by the higher education Ministry, to collect various searches data such as: scientific publications, PhD theses in progress, ongoing masters, research projects, etc. In this perspective, the research center integrates all existing data sources by mean of an active ETL process to feed a central data warehouse with a high level of freshness. The data sources concerned by this process are the universities, research laboratories as well as the semantic knowledge bases available on the web containing such kind of data, for example: dblp, yago, Wikipedia, etc.

The goal of the ETL process is to populate the target DW schema, represented by an Integrating Ontology (IO). The IO is defined using our previous work [5] from the global ontology, LUBM schema in our context. Let assume the existing of four Oracle SDB ($S_1$, $S_2$, $S_3$ and $S_4$) created using LUBM benchmark schema and populated locally using YAGO7 KB.

YAGO was automatically built from Wikipedia, GeoNames, and WordNet, and contains nearly 10 million entities and events, as well as 80 million facts representing general world knowledge. Currently, yago has achieved a precision of 95%. Compatible with RDF, it allows generating data in different formats: Turtle, N-Triple, Literals, etc. YAGO contains knowledge about the real world including entities (e.g., university, people, cities, countries, etc.) and facts about these entities. Note that, data from such knowledge bases are usually freely accessible over SPARQL endpoints. Besides, oracle delivers RDF Semantic Graph features as part of Oracle Spatial and Graph. With native support for RDF and OWL standards for representing semantic data, with SPARQL for query language. Oracle has defined the fragment OWLPrime, a subclass of DLs, that limits the expressive power of the DL formalism in order to ensure decidable query answering. This fragment offers the following constructors: rdfs:domain, rdfs:range, rdfs:subclassOf, rdfs:subPropertyOf, owl:equivalentClass, owl:equivalentProperty, owl:sameAs, owl:inverseOf, owl:TransitiveProperty, owl:SymmetricProperty, owl:FunctionalProperty, owl:InverseFunctionalProperty.

The ETL process is defined at the conceptual level based on the generic operators defined above. Thereby, the logical design phase requires the translation of the DW schema into logical model and translation of ETL process into logical workflow. The latter consists in translating conceptual ETL operators into SPARQL queries. Each query result is represented through an RDF graph. The following example translates the Filter operator into an RDF graph:

\[
\text{Select} \ ?\text{instance} \ ?P \\
\text{where} \\
\{\text{GRAPH} :?G \\
(?\text{instance} \ \text{rdf:type} \ \text{name-space:Class} \ . \\
\text{name-space:P} \ ?P \ . \\
\text{FILTER} \ (?P \ \text{op} \ \text{value\_condition})\}\\n\]

6. EXPERIMENTS

We first conduct an experimental study to analyze the effectiveness of the optimization approach proposed. Then, we evaluate the loading and query performance in comparison with our previous work [4].

6.1 Environment

Platform and data layout. We used Oracle Semantic Database 12c release 2 as the database back-end. For efficiency, we implemented a Btree indexing triples and sparql query hints. The storage of data was done on triple table, using a distinct integer for each distinct URI or literal value. In order to enhance the efficiency of frequently queries, we have indexed the triple table on s, p, o, and all two and three column combinations.

Moreover, some PL/SQL APIs were invoked after the integration of each data source (load of instances). The API SEM_PERF.GATHER_STATS allow to collect statistics for data sources and the API SEM_APIS.ANALYZE_MODEL for DW models in the semantic network graph. The memory SGA and PGA are also increased to 2GB.

Data and queries. Our experiments is based on YAGO KB; version 3.0.2, having an architecture classified on themes. Each theme is a set of facts. A fact is an RDF triple: $<s,p,o>$. YAGO has defined the context relation between individuals [39] which we used to extract the set of themes related to our context study (university domain). The resulting contextual YAGO KB contains around $5.9 \times 10^9$ triples. From this set, we have generated five data-sets, representing data sources (SDB), equivalent to our previous work [4] in order to make a comparison. The five SDBs and the DW schema have been deployed using Oracle DBMS. Oracle offers different format for data loading such as: RDF/XML, N-TRIPLES, N-QUADS, TriG and Turtle. We choose N-Triple format (.nt) to load instances using Oracle SQL*Loader.

The LUBM benchmark is an ontology for the university domain. It was used to generate the schemas of SDBs and DW. It offers fourteen extensional queries representing a variety of properties. As a measurement of query complexity, table 2 describe the number of self-joins on triple table for
Table 2: Number of Joins in the Queries of the LUBM Benchmark.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>14</td>
<td>0</td>
</tr>
</tbody>
</table>

6.2 Experimental Results

In this section, we report the major findings from our experimental study. We present results on the following experiments: (i) a validation of the dynamic materialized view selection method; (ii) an analysis of the performance of the ETL process; and (iii) a comparison between our proposal and the previous work [4].

Validation of the dynamic materialized view selection method. In order to evaluate the efficiency of the materialized view selection method proposed in section 4.3, we first check the scalability of the approach. We created larger RDF graphs such that the size of materialized views would be multiplied by a factor of 1 to 5, with respect to the different steps explained above. The corresponding materialization time is shown in Fig. 9, demonstrating linear scale-up w.r.t. the data size.

Figure 9: Materialization time vs. triple size.

Moreover, we have run the fourteen queries of the LUBM bench on the DW before and after the execution of the materialized view selection algorithm. Fig. 10 describe the runtime of some queries over raw data and views, for different size of the DW. It shows the performance of queries over materialized views that becomes more evident with the growth in the volume of data, due to the dynamic selection of the appropriate views. Moreover, evaluating queries using views is on average 4 times faster than raw data. This finding can be explained by the availability of partial results from materialized views.

Analysis of the performance of the ETL process. In this experiment, we evaluate the ETL algorithm which considers the different possible cases namely: preprocessing, streaming and offline. For each phase, we evaluate the response time of the ETL process with and without memory cache. Note that a memory cache is allocated for the preprocessing phase (buffer cache) and for the stream phase (Stream cache and materialized view cache). Fig. 11 depicts the results obtained. The dynamic selection of materialized views greatly improve the response time of ETL process. The use of the cache as well. Indeed, the recently accessed streaming data and materialized views selected are cached in the memory respectively in Stream buffer and materialized view cache. The use of offline phase shows less performance caused by an extra I/O operations from disk-based access.

Figure 11: Query response time before and after materialized view selection (with cache and without cache).

Comparison between our proposal and the previous work [4]. Our experiments demonstrate the feasibility of our approach, based on the standard RDF. We showed a scalable performance during materialized view creation and integration of data sources. Fig. 12 demonstrates a comparison between the two approaches on the basis of time(s) performance of instances loaded in DW per concept. It clearly indicates that our proposal based on dynamic materialized view selection outperforms the [4] approach.

On the other hand, we have studied the ETL Algorithm and we were interested on the time complexity. The algorithm were implemented based on graph theory, where nodes represents concepts and edges roles definitions. We have examined the number of iterations of our algorithm to generate the ETL graph (semantic DW populated) and we have compared it with the state of the art [4]. The algorithms are based on concepts searches (Tbox for intentional mappings) and not instances (Abox for extentional mappings). The time complexity is O(n) for both algorithms, where n is the number of involved nodes (which means concepts). These depend on the resolving of constraints defined

Figure 10: Execution time(s) of some LUBM queries over raw data and views

http://www.cytoscape.org/
Figure 12: Evaluation time(s) of instances loaded in SDW per concept.

Figure 13: Complexity of the ETL Algorithm

7. CONCLUSION

8. REFERENCES


