**Building Rule-Based OLTP Systems Using Oracle RDF**

Philip Cannata, Nigel Jacobs

### An RDF-OLTP Approach to EmpDept

Recently we watched a presentation on [the most innovative database you’ve never heard of](https://www.youtube.com/watch?v=hicQvxdKvnc). This was a presentation on [Datomic](https://www.youtube.com/watch?v=9TYfcyvSpEQ), which is an [RDF-like database system](https://thornydev.blogspot.com/2012/06/datomic-initial-analysis.html). But the title of this presentation could just as well apply to Oracle’s RDF Knowledge Graph database system. No one has heard of or thinks of using Oracle’s RDF for OLTP applications even though it offers a host of very innovative features. This post will try to explain these features by describing a simple application using the standard EmpDept Logical Model:

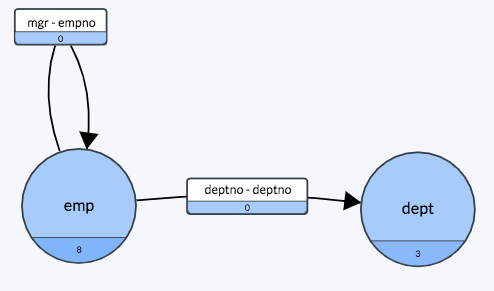


Figure 1 – Standard emp/dept Logical Model

To build an OLTP application using Oracle’s RDF, you can simply build Views over the Logical Model along with three “INSTEAD OF” Triggers (insert, update, and delete) for each View. You don’t need DDL to build the tables because RDF is a schemaless database. This makes it very easy to keep the Logical Model in sync with the application. The traditional steps of:

1. building the Logical Model
2. translating the Logical model to a Data model
3. generating the DDL for the Data Model
4. loading the DDL into the database

are simplified to the single step:

1. build the Logical Model

In other words, to change the database, you just change the Logical Model and the Views of the database. No physical database changes need to be made. This is a huge win, when managing change over steps 1-4 of the traditional approach.

To complete the application, the Views can be used to present Business Objects to a UI through a REST interface. But wait … where is the Business Logic?

In the typical industry-standard architecture (Figure 2):

1. the UI communicates via REST with
2. the Application Layer, which applies business logic to the domain model, which is
3. translated via an ORM to
4. a data model which is read/written via SQL to
5. the RDBMS

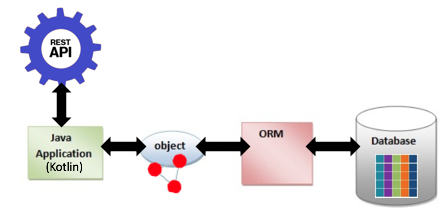


Figure 2- Industry Standard SaaS Architecture Diagram

In the RDF-OLTP approach (Figure 3), this architecture is inverted, to put the relational view above the RDBMS in which are stored the business rules and data:

1. the UI communicates via REST with
2. the thinner Application Layer, which use logical views and SQL to interrogate
3. the DBMS containing
4. RDF data and rules encapsulating business logic

Instead, the Business Logic (and Data Integration Logic if applicable) exists in the database as Rules (Methods) on Views. It’s as though the diagram in Figure 2 has been turned upside down with the Business Rules in the database.

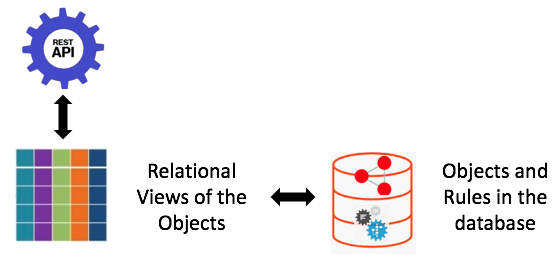


Figure 3 – Oracle RDF SaaS Architecture Diagram

This architecture brings to fruition a 30 year desire to have the “Complexity in the Database”, not in the Application (Program) or User’s head, as depicted in the following figure from this 1988 [Doug Tolbert](https://www.researchgate.net/profile/Doug_Tolbert) [Presentation](file:///Users/../Users/pcannata/Mine/My%20Repositories/turnable/turnable-back-end/docs/Shortcourse%20on%20Next%20Generation%20Database%20Systems.pdf). In the diagram, the size of the boxes depicts where the complexity lives:

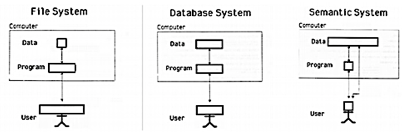


Figure 4 – From left to right, complexity in data Files, Relational Database, and RDF

With RDF-OLTP, Microservices can be completely eliminated or greatly simplified by representing logical views over domain models, streamlining service deployment.

But, how can RDF-OLTP provide multi-tenancy and scalability? Where are the clusters? This is where Oracle’s RDF shines. Multi-tenancy is provided by Pluggable Databases (PDBs) and scalability is provided using partitioning (Models) that are an integral part of Oracle’s RDF implementation. If you are a Cloud SaaS provider, you simply setup a PDB for each customer, build the Views and Triggers in the PDB, and appropriately partition the customer’s data into RDF Models. Each Model shares the Views and Triggers in the PDB.

With this design, the Logical Model and Business Logic change as the Views evolve. Change Management can be simply applied to the Views and their associated Triggers. This even applies to complex changes like those described in Souri’s blogs on [RDF Quads](https://blogs.oracle.com/oraclespatial/modeling-evolving-data-in-graphs%3A-the-power-of-rdf-quads), [Named Triples](https://blogs.oracle.com/oraclespatial/rdf-extending-rdf-to-support-named-triples), and [RDFn](https://blogs.oracle.com/oraclespatial/extending-rdf-to-support-named-triples).

Now for some details. The code for this simple, sample application can be found at <https://github.com/CannataUT/RDF-OLTP.git>.

The application consists of one View (Saas\_Emp\_View), which joins the emp and dept Classes such that:

select rule\_set\_award, empno, ename, job, salary, comm, training, dname, location, award

from SaaS\_EMP\_VIEW order by dname;

yields the following rows:

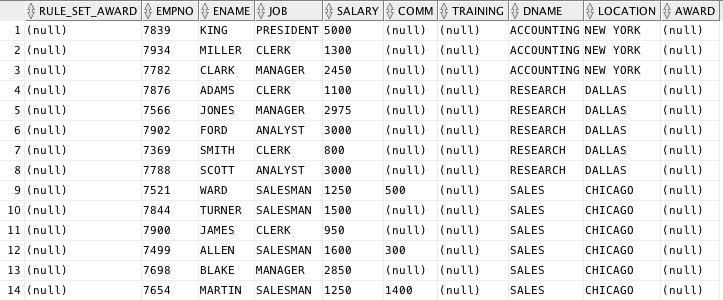


Figure 5 - Title

Updates to the emp and dept attributes can be made through this View. Unlike most other systems, however, the same would be true even if the View was a very complicated join.

The application allows for the emp *training* attributes to be changed to “Yes”. Once all of the *training* attributes for a dept are “Yes”, the application changes the dept *award* attribute to “Eligible”. Here is an example of that change:

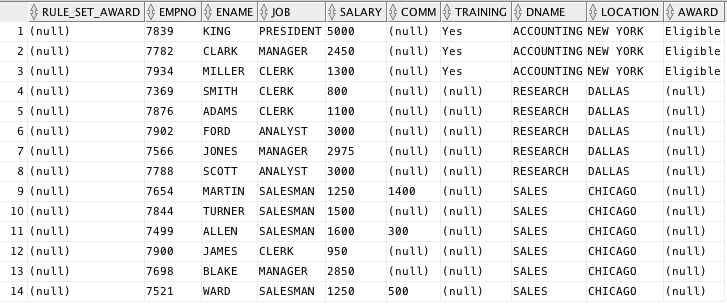


Figure 6 - Title

If there were a View on the dept Class, it would now look like the following:

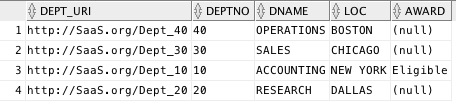


Figure 7 - Title

To make this happen, the UI sends an update to the REST server to set the *training* attribute to “Yes”, the REST API code updates the *training* attribute and then calls the *rule\_set\_award* Rule (Method) on an associated View something like the following:

update Saas\_Emp\_Rules set rule\_set\_award = ‘Run’

When this update is run, the *rule\_set\_award* code checks to see if all of the *training* attributes for a dept are “Yes”, and, if they are, sets the *award* attribute to “Eligible”.

The *rule\_set\_award* code is written using SPARQL. Oracle’s implementation of RDF allows for the SPARQL code to be contained in the update trigger of the View. The act of updating *rule\_set\_award* on the View causes the code to run.

The SPARQL code looks like the following:

**UPSERT** { ?dept\_uri saas:award ?eligible . }

WHERE

{ # Get the dept for each emp.

?emp\_uri a saas:Emp .

?emp\_uri saas:empDeptEVA ?dept\_uri .

?dept\_uri saas:deptno ?deptno .

**MINUS** # Remove depts that have at lease one emp who is not trained.

{ SELECT ?dept\_uri

WHERE

{

?emp\_uri a saas:Emp .

?emp\_uri saas:empDeptEVA ?dept\_uri .

OPTIONAL { ?emp\_uri saas:training ?training. }

FILTER(?training = "")

}

}

# Construct the value to be inserted into the dept’s award attribute.

**BIND**(CONCAT("Department ", ?deptno, " is eligible for an award") AS ?eligible)

}

In this version of the code, we set the award attribute to “Department 10 is eligible for an award” instead of just “Eligible”. This was done to give an example of the **BIND** function, which allows a value to be assigned to a variable. This powerful feature of SPARQL allows the **BIND** function to invoke other functions. For example, IF (see [here](https://semaku.com/post/using-coalesce-and-if-in-sparql-for-nested/) for details), or in the case above CONCAT, which can this be used for value mapping, e.g., mapping DB values to UI formats. A function we have proposed -**UPSERT**, would allow a value to be inserted if that value does not already exist, or updated if the value does exist. This proposed unique feature of Oracle’s RDF would be extremely useful for building OLTP SPARQL. It is much less complex/error prone as compared to the DELETE, INSERT, WHERE combination of standard SPARQL as found in our sample code. In our humble opinion, GRAPHQL needs these features in order to compete with RDF/SPARQL.

One very important aspect of Oracle’s RDF implementation is that the Views and the RDF database are tightly integrated into the same system. This allows the Views and Triggers to contain SPARQL code. This would not be possible if Oracle’s RDF implementation only provided a [Graph Server and Client.](https://www.oracle.com/database/technologies/spatialandgraph/property-graph-features/graph-server-and-client/graph-server-and-client-downloads.html)

Oracle’s Property Graph does not have this tight coupling, nor does the Serverless Autonomous Database. This could potentially represent a big flaw in Oracle’s database strategy. The same can be said of all other vendor’s Graph implementations. RDF-like systems might be a big part of the future. [Datomic’s Ions](https://docs.datomic.com/cloud/ions/ions.html) provides just such a tight coupling, allowing business logic to be executed with the database. Amazon / AWS / DynamoDB appear to be aligning with Hickey and Datomic even though these types of database systems are not of the same “Enterprise” class as Oracle.

Oracle has a gem in its RDF implementation, “which you’ve probably never heard/thought of” using for an OLTP application.

### A Traditional Microservice Approach to EmpDept (MS-RDBMS)

[Helidon](https://helidon.io/), from Oracle, is a set of open-source libraries for developing microservices. It supports the Eclipse MicroProfile standard, and provides a plethora of many support services e.g. health, metrics, logging, configuration, db access, REST, fault-tolerance and more. Oracle is now embracing microservice architectures both internally and in it’s SAAS products.

We will explore a more ‘traditional’ microservice implementation of EmpDept, written in Java, based on Helidon as a microservice container, with the widely-used Oracle ADF framework as an ORM, communicating via SQL and JDBC with an Oracle relational data store. The implementation architecture is described in Figure 2.

In the sections below, we will describe the architecture and implementation details of each layer, with a final section describing findings, and contrasting this approach with the RDF-OLTP approach. The code for this application can be found at <https://github.com/CannataUT/MS-RDBMS>

#### *MS-RDBMS: REST Service Layer*

Helidon acts as the mediator between REST clients and REST APIs, with route configuration, authentication and security, It handles JSON parsing and serialization.

curl -X GET <http://localhost:8080/Emp>

→ All employees as a list of Employees as JSON maps

curl -X GET [http://localhost:8080/Dept/30](http://localhost:8080/Dept/20)

→ Sales department as a JSON map

curl -X GET [http://localhost:8080/Emp/name=Nigel%20Jacobs](http://localhost:8080/EmpByName/NigelJacobs)

→ Employee with name “Nigel Jacobs” as a JSON map

curl -X PUT <http://localhost:8080/Emp/name=Nigel%20Jacobs> ...

→ Create / Update employee with name “Nigel Jacobs”

#### *MS-RDBMS:* Business Semantics Layer

This layer manages complex business semantics that can’t be expressed in the more limited mappings of the ORM. It also handles mapping of values between the format of the REST API and the ORM layer, using domain objects (POJOs). It needs to understand the REST layer structure, and the ADF structure: it can dynamically retrieve ADF access modules based on their full path. This is pure Java code: highly expressive but very change intolerant.

EmployeeRepository:

Employee find(Integer employeeNumber) // retrieve an employee from the db

void save(Employee employee) // save an employee to the db

#### *MS-RDBMS:* Object-Relation Mapping (ORM) Layer

The objects and mappings between the presentation-layer domain model and the data model of the Oracle DBMS are often represented in Oracle’s business applications using Oracle’s declarative [ADF](https://docs.oracle.com/middleware/1212/adf/ADFCG/intro.htm#ADFCG111) framework. The mappings are build at design time via Oracle’s Jdev UI tool, generating XML definition files and Java stubbed classes. allowing for queries and updates across the mappings, though they are built declaratively via the Jdev UI-based tool. The mappings are ultimately translated to standard SQL over JDBC at runtime, The more complex business semantics must be handled within this layer by Java code within the stubbed classes, or supporting classes. Since the ADF mappings are stored as XML files within the Java code (within the JAR files), there is a complex coupling between the business layer and database schema, which is not amenable to change (XML files in jar files, Java code, UI-based tools).

The language and model of the ADF mappings present yet another formalism for programmers to learn with orders of magnitude of complications in change management.

Entity Objects: e.g. EmployeeEO, DepartmentEO

- XML files with Java stubs

- attributes (columns)

- keys

- filter definitions

- inter-entity links

View Objects e.g. EmployeeVO, DepartmentVO, EmpDeptVO

- XML files with Java stubs

- SQL queries

- entity refs.

Java code from a class responsible for mapping cached data loaded from DB, to compute the award status of a department (see above for details):

*// For given employee and her department, calculate the department award*  
*// based on the employees training and the departments existing award value*  
for (Employee emp : mapEmpNoToEmployee.values()) {  
 Department dept = emp.getDepartment();  
 boolean training = emp.hasTraining();  
 *// dept has award until at least one employee without training*  
boolean award = (dept.hasAward() == null) || dept.hasAward();  
 dept.setAward(award && training);  
}

#### *MS-RDBMS:* Database Layer

Oracle has been served well, for many years, by the simplicity and power of its relational DBMS. But this base technology is now showing it’s age, and NoSQL, document and Graph databases have recently become popular. For this layer, we will stick with the most popular form for Oracle’s internal applications: SQL over JDBC interrogating relational table, utilizing a Java JDBC client, relational tables, standard SQL for query expression, DDL for schema creation, and DML for initial test data population.

SELECT \*

FROM EMP, DEPT

WHERE EMP.DEPT\_ID = DEPT.ID;

CREATE TABLE EMP

(EMPNO NUMBER(7) NOT NULL,

CONSTRAINT emp\_pkey PRIMARY KEY (empno),

ENAME VARCHAR2(10),

JOB VARCHAR2(9),

MGR NUMBER(4),

HIREDATE DATE,

SAL NUMBER(7, 2),

COMM NUMBER(7, 2),

DEPTNO NUMBER(2));

### *MS-RDBMS: Findings and Comparison with RDF Approach*

#### *1. Loose-Coupling Creates Code Complexity*

The multiple layers each with it’s own language and formalism, makes it difficult to implement complex behaviors. The code fragment listed above to compute a department’s award can not be expressed within the DB as in the RDF approach, nor can it be expressed within the standard ADF structure. We had to build a custom a cache structure (Java map) to compute awards during load / find API calls. We had to keep it as close to the base ORM as possible to avoid the risk and complexity of passing caches between layers.

#### *2. Loose-Coupling Requires More Network Data Traffic*

The business logic and ORM layers are responsible for presenting a fully realized graph of domain objects (POJOs) to the presentation layer on loads i.e. when an Employee is loaded it contains a link to its Department with its computed award. To do this, we had to load ALL employee data for the department to compute the award, when loading just a single employee, increasing the data traffic between the DB server and client layers. Also, the in-memory dynamic caches (Java maps) that must be coded to realize the POJO links adds to code complexity in the ORM layer.

#### *3. Additional Formalisms and Development Tools / Processes Makes Change Harder*

This hybrid approach is fairly typical for current application stacks. It uses many disparate tools and formalisms during the development process: Jdev UI, XML, MVN jar packaging, SQL development, DDL development, POJOs for domain models etc. Simple changes to the data models / schema become much slower to integrate and test, adding to system fragility. In contrast, the simplicity of the RDF data model results in a system that is much more tolerant of change.

#### *4. Monolithic DBMS Architecture Not Well Suited For Flexibility of Microservices*

We chose a microservices (MS) framework as a host for this approach, as it is becoming increasingly popular in application stacks with it’s supposed ease of provisioning and scaling. However, our approach is typical of the first generation of application stacks: a set of microservices that share a traditional relational DBMS backbone. In this case, the DBMS with it’s increased rigidity of schema change and service provisioning will be the weakest link.

#### *5. Scalability*

#### We did not address scalability in this application because database partitioning and microservice clustering are very complex topics, which are beyond the scope of this post. However, as noted in the first section, partitioning is essentially free with Oracle’s RDF Models, and clustering is not needed.