

**Comparison of Triplestore Architectures in the Oracle RDBMS**  
**Jacob Schrum**  
**(No Partner)**  
**Oracle 10g/11g Enterprise Edition**

# Comparison of Triplestore Architectures in the Oracle RDBMS

By Jacob Schrum  
Final Project for CS386D

## Abstract

Several different database configurations for supporting triplestores are implemented in Oracle and compared to each other in terms of execution speed across several queries. The configurations considered are a simple baseline architecture, a Triple-Triple architecture composed of redundant copies of the triples table with different indexing orders, and a configuration based on bitmap join indexes, which are exclusive to the Oracle RDBMS. Sadly, the results for the alternate architectures show no significant improvement over the baseline architecture, seemingly due to inexplicable behavior of Oracle's internals.

## 1. Introduction

Support for large-scale semantic web applications depends on the use of triplestores for housing massive quantities of RDF data. The appeal of triplestores is in their ability to represent arbitrary relations between entities with a generalized format: the notion of a triple, with which a relationship between a *subject* and an *object* is denoted by some *property*. Such a representation can effectively handle arbitrary ontologies.

However, the generality of this representation results in a single relation, a table of triples, being the focus of any and all operations of the data. For any worthwhile application, the triples table will have to be enormous, which will typically make attempts to retrieve data from it horribly slow. This paper proposes some architectures aimed at speeding up access to a triplestore. But first comes a review of some past work in this area to see what has been done and to explain the inspiration for the architectures proposed later in this paper.

## 2. Previous Approaches

Several complete systems have been designed for the purpose of storing and accessing triplestore data. Such systems include Jena<sup>1</sup>, Sesame (Broekstra and Kampman), Parka (Stoffel et al.) and 3store (Harris and Gibbins). These systems generally use some form of relational database management system (most typically MySQL) for the underlying storage of triples data.

However, simply storing data in a large table of triples is terribly inefficient, so there has been much research into alternate underlying representations that are logically equivalent to a triplestore representation. Some proposed schemes have separated the

---

<sup>1</sup> <http://jena.sourceforge.net/>

triples table out into several separate tables based on a set of properties that given entities tend to have. So for a set of subjects that belong to a common type, each one should also have the same set of properties. However, this is not necessarily true, and unstructured RDF data can often lead to very sparse property tables full of NULLs.

Albadi et al. (2007) proposed a more efficient alternative to this called vertical partitioning. Vertical partitioning is still property-centric, but allows for a more concise representation with no NULL values. For every unique property in the triples table, a separate table of pairs is made. The first column of the table is the subject, and the second is the object. If a given subject/object pair is not in the table for given property, then the corresponding triple is not in the triplestore. Additionally, each of these tables is stored in sorted order on subject and then object, thus allowing for fast merge joins in most cases without the need to sort first.

While vertical partitioning solves the problem of having sparse property tables filled with NULLs, proper use of this schema still requires many UNION operations, and searches across several separate tables. However, the idea of enabling fast merge joins is powerful, and was used soon after by Weiss et al. (2008) to create the Hexastore. The Hexastore is simply a triples table with B+Tree indexes on every possible ordering of the three columns in the triples table. The advantage in this comes from the fact that queries on a single triples table involve many self joins, always on some combination of subject, property and object. Therefore, for any pairing in a self join, there exist indexes for those columns in order, which can then always be exploited to do a fast merge join without sorting first. This configuration seems to be the current state of the art in quickly accessing a triplestore. However, rather than using an existing RDBMS, Weiss et al. chose to implement their Hexastore in Python, which gave them full control over the underlying database system. Such is not the case in commercial database systems.

### 3. Triplestore Configurations

We now present the triplestore architectures under consideration in this paper. The first is a simple baseline configuration against which the other two will be compared. The second is a configuration that tries to benefit from fast merge joins in a way similar to the Hexastore. The last is a new configuration that can only be implemented in Oracle, due to its support for bitmap join indexes.

#### 3.1. Baseline

The baseline architecture features a single table **spoTriples** whose columns are **s** for subject, **p** for property and **o** for object. Each of these columns stores a numeric value that is a key for the **hash** column

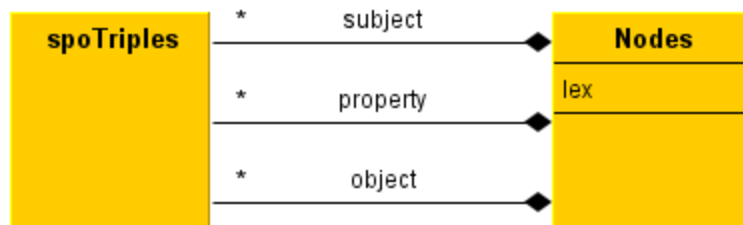


Figure 1: Baseline architecture

in a table named **Nodes**. The **Nodes** table stores string values for each row in a column called **lex** and also tracks other meta-data about the entity.

The primary key of **spoTriples** is a composite key of (s,p,o). This makes it easy to quickly look up rows by their **s** value, but provides no quick way to look up rows by **p** or **o**. The primary key of **Nodes** is **hash**, which is useful since all joins on the **Nodes** table are on this column and one of the columns in **spoTriples**.

### 3.2. Triple-Triple

The Triple-Triple architecture is designed to take advantage of merge joins in a manner similar to the Hexastore. The Hexastore accesses the triples table in sorted order via several secondary indexes. The idea for the Triple-Triple architecture is to have copies of the triples table physically organized on disk in each of the six possible orderings. This should allow for fast merge joins just as with the Hexastore, and it should arguably be faster since the tables are physically organized in each possible ordering (as opposed to just by secondary indexes). In the Triple-Triple architecture (which would perhaps better be thought of as a Hexa-Triple architecture), redundant copies of the triples are stored sorted in every possible order. The table names are **spoTriples**, **sopTriples**, **posTriples**, **psoTriples**, **ospTriples** and **opsTriples**. The

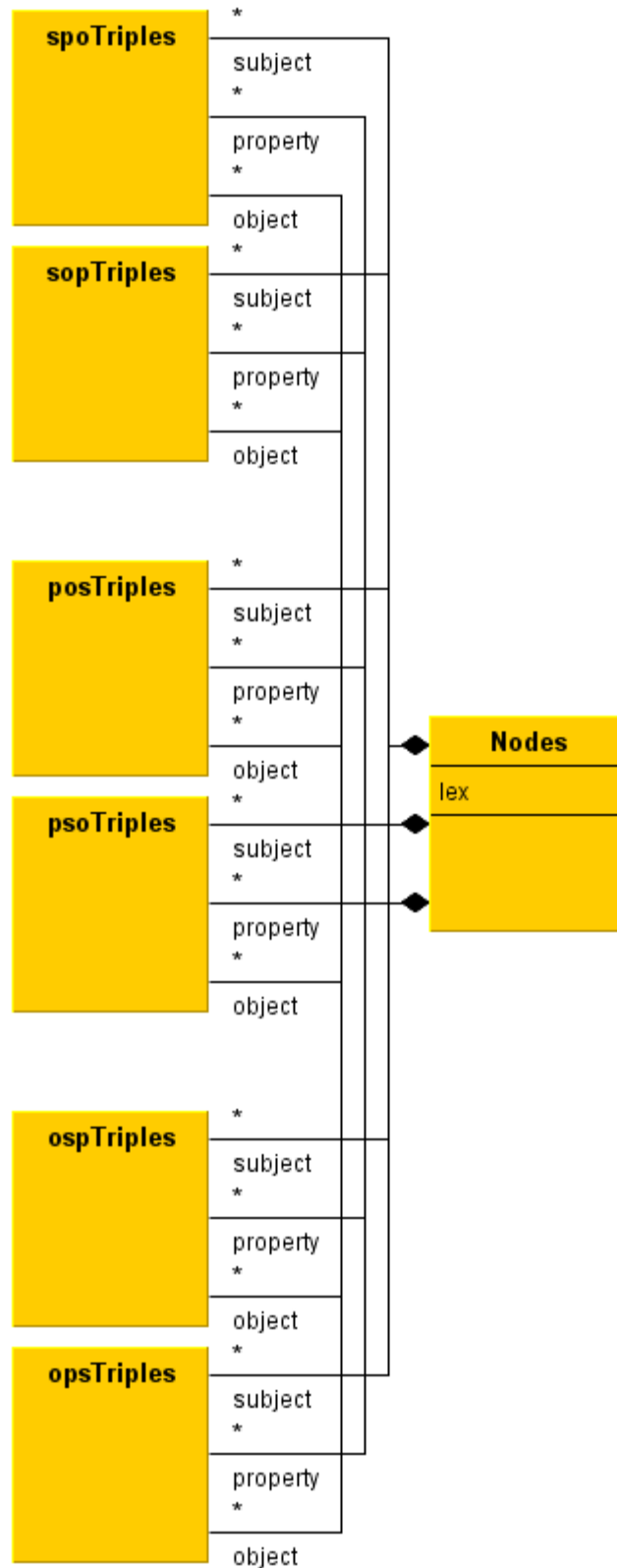


Figure 2: Triple-Triple architecture

corresponding primary keys are **(s,p,o)**, **(s,o,p)**, **(p,s,o)**, **(p,o,s)**, **(o,s,p)** and **(o,p,s)**. In Oracle, tables are not sorted on disk in order of primary key by default. Instead, tables are stored in an unordered heap. In order to store a table on disk in sorted order, the table must be defined as an Index-Organized Table, which results in the table being stored in a B+Tree index structure based on the primary key. This was done for each table in hopes of allowing for fast merge joins.

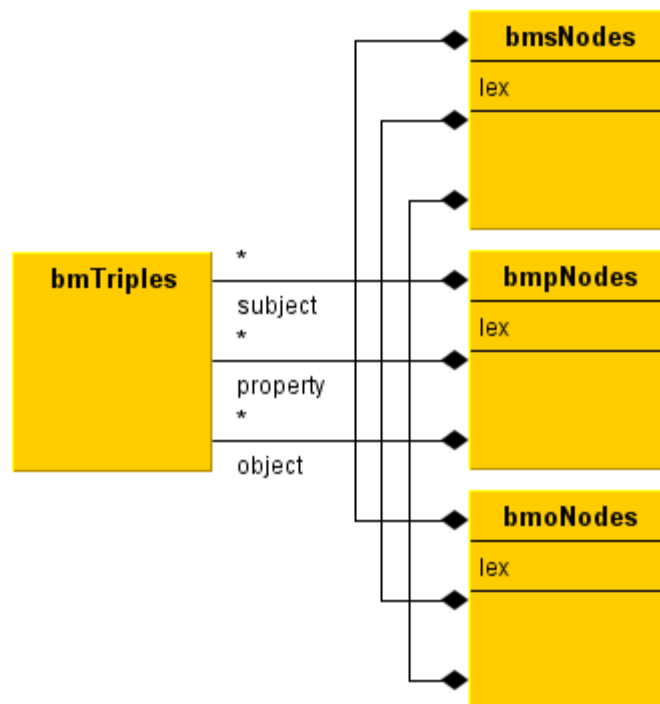
Notice that **spoTriples** is among the tables here. This is the exact same table used in the baseline architecture. Therefore, the baseline architecture is a subset of the Triple-Triple architecture. This means that **spoTriples** is also an Index-Organized Table in the baseline architecture. Each of these tables refers to the same **Nodes** table used in the baseline architecture.

With the introduction of redundant copies, we have made the query writing process more complicated. For any query that can be written in the baseline architecture, there are many equivalent queries in the Triple-Triple architecture. Any instance of one triples table can be replaced with an instance of another triples table without changing the final results. However, what will change is the query plan and the physical sources of the data. Because the purpose of the Triple-Triple architecture is to take advantage of merge joins, tables in queries will be chosen with this under consideration. However, even using this guideline, there will often be more than one choice of tables that is logical for the purposes of using merge joins. Therefore, in all experiments with the Triple-Triple architecture we will be careful to identify which tables were used.

Lastly, it should be kept in mind that since the Triple-Triple architecture uses the same **spoTriples** table as used in the baseline architecture, it can never really be said to perform slower than the baseline. While it is true that a particular choice of tables in the Triple-Triple architecture may perform more slowly than the baseline of only using **spoTriples** repeatedly, it is also true that any query for the baseline architecture is also a query for the Triple-Triple architecture. Therefore, it is always possible for the Triple-Triple architecture to achieve performance at least as good as the baseline.

### 3.3. Bitmapped

The final architecture to be studied takes advantage of Oracle's bitmap indexes, specifically bitmap join indexes. First all triples are stored in a single table named **bmTriples**. The table has a



**Figure 3: Bitmapped architecture**

standard B+Tree index on (**s,p,o**) for its primary key, as in the baseline architecture. Then three bitmap join indexes are created between the triples table and the nodes table. Each of these indexes specializes on the **lex** column of the nodes table for a join between the **hash** column of nodes table and either the **s**, **p** or **o** column of the triples table. This means that for any join between the nodes and triples tables that also involves selection on the **lex** column, the keys of the join are pre-computed, and stored as a bitmap index. Although conventional knowledge indicates that bitmap indexes should only be used on columns with low cardinality, the compression of bitmap indexes should actually make their usage appropriate on the **lex** field, which has unique values (Johnson).

Configuring the tables as described above requires making several bitmap indexes on the nodes table, each with a different join condition. Oracle does not allow this because, despite the different join conditions, each index is considered to be on the **lex** column, so they conflict with each other. In order to implement all of the desired join indexes, there needs to be three separate copies of the nodes table: **bmsNodes** is a copy of the nodes table that is used whenever there is a join between the triples and the nodes on the **s** column, **bmpNodes** is used whenever there is a join on the **p** column, and **bmoNodes** is used for joins on the **o** column. This requires the queries to be rewritten, but the process is completely deterministic and straight forward, unlike in the case of the Triple-Triple architecture.

The expectation with these bitmap join indexes is that queries involving many conditions in the WHERE clause for a specific value of **lex** will be speed up, since all the join results are pre-computed. Since every meaningful query has to join on the nodes table at some point, this should be a large savings.

## 4. Experimental Approach

Now we describe the details of how these three architectures were tested.

### 4.1. Platform

The architectures described above were all tested using the Oracle RDBMS version 10g enterprise edition. Oracle was used because it is the only major RDBMS that supports *persistent* bitmap indexes (though other systems do support the creation of *on-the-fly* bitmap indexes at runtime).

The Oracle system was loaded in Windows Vista 32 bit on a laptop with dual core AMD Turion(tm) 64 X2 Mobile Technology TL-60 processors clocked at 2000 Mhz each. The total available RAM was 2 GB and the available disk space after loading the database tables was around 20 GB.

Query speeds were evaluated using a Java program to execute and time the queries. Query times presented therefore include whatever overhead is introduced by interfacing with Oracle via Java's ODBC drivers. In order to make the timing data as accurate as possible, Oracle system commands were used to flush the cache between queries. The two commands executed before every query were:

```
ALTER SYSTEM FLUSH BUFFER_CACHE;  
ALTER SYSTEM FLUSH SHARED_POOL;
```

For most queries, preliminary results were obtained with a repetition of the exact same query and compared to a result where certain literal values in the query were replaced with random values. The fixed vs. randomized queries usually returned nearly equal results, which implies that clearing the cache in the above proposed manner is sufficient. If it were not sufficient, we would expect repetition of a fixed query to be very fast after the first execution, since the query results would still have been in memory.

## 4.2. Data

The data used to test these triplestore configurations comes from the Berlin SPARQL Benchmark<sup>2</sup> (BSBM). The data here is based on an e-commerce usage case with the following classes in its ontology: **Product**, **ProductType**, **ProductFeature**, **Producer**, **Vendor**, **Offer**, **Person** and **Review**.

Different dataset sizes are used to test the triplestores under different loads. Versions of the BSBM data with approximately 250 thousand (250K) and 1 million (1M) triples were tested using all the queries described below. To assure that all trials were executed with the same amount of disk space available, all architectures for a given dataset were loaded into the system before any trials were run.

## 4.3. Queries

The queries used for benchmarking were originally written in SPARQL. These SPARQL queries had to be translated into appropriate SQL queries for each triplestore architecture. The original SPARQL queries were:

### 4.3.1. Query 1

```
PREFIX bsbm-inst:
  <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/>
PREFIX bsbm: <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?product ?label ?value1
WHERE {
  ?product rdfs:label ?label .
  ?product a bsbm-inst:ProductType31 .
  ?product bsbm:productFeature bsbm-inst:ProductFeature131 .
  ?product bsbm:productFeature bsbm-inst:ProductFeature105 .
  ?product bsbm:productPropertyNumeric1 ?value1
}
ORDER BY ?label
```

---

<sup>2</sup> <http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/spec/index.html>

### 4.3.2. Query 2

```
PREFIX bsbm-inst:
  <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/>
PREFIX bsbm:
  <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX dataFromProducer9:
  <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromProducer9/>

SELECT ?label ?producer ?productFeature ?propertyTextual1
?propertyTextual2 ?propertyTextual3 ?propertyNumeric1 ?propertyNumeric2
?propertyTextual4 ?propertyTextual5 ?propertyNumeric4
WHERE {
  dataFromProducer9:Product444 rdfs:label ?label .
  dataFromProducer9:Product444 rdfs:comment ?comment .
  dataFromProducer9:Product444 bsbm:producer ?p .
  ?p rdfs:label ?producer .
  dataFromProducer9:Product444 bsbm:productFeature ?f .
  ?f rdfs:label ?productFeature .
  dataFromProducer9:Product444
    bsbm:productPropertyTextual1 ?propertyTextual1 .
  dataFromProducer9:Product444
    bsbm:productPropertyTextual2 ?propertyTextual2 .
  dataFromProducer9:Product444
    bsbm:productPropertyTextual3 ?propertyTextual3 .
  dataFromProducer9:Product444
    bsbm:productPropertyNumeric1 ?propertyNumeric1 .
  dataFromProducer9:Product444
    bsbm:productPropertyNumeric2 ?propertyNumeric2
}
```

### 4.3.3. Query 3

```
SELECT ?a ?c
WHERE {
  ?a rdf:type ?b.
  ?b rdf:type ?c.
}
```

### 4.3.4. Query 4

```
SELECT ?productLabel ?reviewTitle ?personName
WHERE {
  ?product bsbm:productFeature bsbm-inst:ProductFeatureX.
  ?review bsbm:reviewFor ?product.
  ?review rev:reviewer ?person.
  ?product rdfs:label ?productLabel.
  ?r dc:title ?reviewTitle.
  ?person foaf:name ?personName
}
```

Where X is appropriately randomized for the dataset.

### 4.3.5. Query 5

```
SELECT ?product ?price ?vendor
WHERE{
    ?pt      rdfs:subClassOf bsbm-inst:ProductTypeX.
    ?p rdf:type ?pt.
    ?o bsbm:product ?p.
    ?o bsbm:vendor ?v .
    ?o bsbm:price ?price .
    ?p rdfs:label ?product.
    ?v rdfs:label ?vendor .
    ?v bsbm:country http://downlode.org/rdf/iso-3166/countries#Y
}
```

Where **X** and **Y** are appropriately randomized for the dataset.

### 4.3.6. Query 6

```
SELECT ?p
WHERE{
    bsbm-inst:ProductTypeX ?p bsbm-inst:ProductTypeY
}
```

Where **X** and **Y** are appropriately randomized for the dataset.

## 4.4. Query Translation

Here are the details of how these queries were translated into SQL for each of the three architectures.

### 4.4.1. Baseline

#### 4.4.1.1. Query 1

Query 1 was translated into the following SQL query:

```
SELECT
    s1.lex as product,
    o1.lex as label,
    o5.lex as value1
FROM
    spoTriples t1, Nodes s1, Nodes p1, Nodes o1,
    spoTriples t2, Nodes p2, Nodes o2,
    spoTriples t3, Nodes p3, Nodes o3,
    spoTriples t4, Nodes p4, Nodes o4,
    spoTriples t5, Nodes p5, Nodes o5
WHERE
    t1.s = s1.hash
    and t1.p = p1.hash
    and t1.o = o1.hash
    and p1.lex = 'http://www.w3.org/2000/01/rdf-schema#label'
    and t1.s = t2.s
```

```

and t2.p = p2.hash
and t2.o = o2.hash
and p2.lex = 'http://www.w3.org/1999/02/22-rdf-syntax-ns#type'
and o2.lex =
  'http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/ProductType31'
and t1.s = t3.s
and t3.p = p3.hash
and t3.o = o3.hash
and p3.lex =
  'http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productFeature'
and o3.lex =
  'http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/ProductFeature131'
and t1.s = t4.s
and t4.p = p4.hash
and t4.o = o4.hash
and p4.lex =
  'http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productFeature'
and o4.lex =
  'http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/ProductFeature105'
and t1.s = t5.s
and t5.p = p5.hash
and t5.o = o5.hash
and p5.lex =
'http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productPropertyNumeric1'
ORDER BY label ASC

```

The query above is the fixed version of the query that is known to return results from the 250K version of the database. When randomizing the query, several of the literal values were changed to select from available string values in the nodes table. Specifically, if a given search string ended in a number, this numeric value was replaced with another number for which a corresponding entry with the same prefix and the new number existed in the **Nodes** table. These numbers are bold in the query above.

Choosing random search strings in this way assures that the question being asked by the query still makes some degree of sense, but it should be noted that for this specific query, randomizing the search strings in this way nearly always results in an empty result set.

#### 4.4.1.2. Query 2

Proper SQL for representing query 2 was never fully developed. Instead, two queries that calculate a portion of the result for the overall query were used. The queries below are the fixed versions of the queries. These queries were randomized using the same method for query 1 above. The bold numbers in the queries indicate the parts that were randomized.

##### 4.4.1.2.1. Query 2a

```

SELECT
  o1.lex as label,
  o2.lex as "comment",
  o4.lex as producer
FROM

```

```

spoTriples t1, Nodes s1, Nodes p1, Nodes o1,
spoTriples t2, Nodes p2, Nodes o2,
spoTriples t3, Nodes p3,
spoTriples t4, Nodes p4, Nodes o4
WHERE
  t1.s = s1.hash
  and t1.p = p1.hash
  and t1.o = o1.hash
  and s1.lex =
'http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromProducer9/Product444'
  and p1.lex = 'http://www.w3.org/2000/01/rdf-schema#label'
  and t1.s = t2.s
  and t2.p = p2.hash
  and t2.o = o2.hash
  and p2.lex = 'http://www.w3.org/2000/01/rdf-schema#comment'
  and t1.s = t3.s
  and t3.p = p3.hash
  and p3.lex =
'http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/producer'
  and t3.o = t4.s
  and t4.p = p4.hash
  and t4.o = o4.hash
  and p4.lex = 'http://www.w3.org/2000/01/rdf-schema#label'

```

This query reliably returns results for both the 250K and 1M databases even when randomized.

#### 4.4.1.2.2. Query 2b

```

SELECT
  o1.lex as label,
  o2.lex as "comment",
  o4.lex as producer,
  o6.lex as productFeature
FROM
  spoTriples t1, Nodes s1, Nodes p1, Nodes o1,
  spoTriples t2, Nodes p2, Nodes o2,
  spoTriples t3, Nodes p3,
  spoTriples t4, Nodes p4, Nodes o4,
  spoTriples t5, Nodes p5,
  spoTriples t6, Nodes p6, Nodes o6
WHERE
  t1.s = s1.hash
  and t1.p = p1.hash
  and t1.o = o1.hash
  and s1.lex =
'http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/dataFromProducer9/Product444'
  and p1.lex = 'http://www.w3.org/2000/01/rdf-schema#label'
  and t1.s = t2.s
  and t2.p = p2.hash
  and t2.o = o2.hash
  and p2.lex = 'http://www.w3.org/2000/01/rdf-schema#comment'
  and t1.s = t3.s
  and t3.p = p3.hash
  and p3.lex =

```

```

'http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/producer'
and t3.o = t4.s
and t4.p = p4.hash
and t4.o = o4.hash
and p4.lex = 'http://www.w3.org/2000/01/rdf-schema#label'
and t5.s = t1.s
and t5.p = p5.hash
and p5.lex =
'http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productFeature'
and t6.s = t5.o
and t6.p = p6.hash
and p6.lex = 'http://www.w3.org/2000/01/rdf-schema#label'
and t6.o = o6.hash

```

This query reliably returns results for both the 250K and 1M databases even when randomized.

#### 4.4.1.3. Query 3

Query 3 is designed to return a very large result set. There are no parts of the query that can reasonably be randomized, so there is only a fixed version

```

SELECT
  s1.lex as subject,
  o2.lex as pred
FROM
  spoTriples t1, Nodes s1, Nodes p1,
  spoTriples t2, Nodes p2, Nodes o2,
WHERE
  t1.s = s1.hash
  and t1.p = p1.hash
  and p1.lex = 'http://www.w3.org/1999/02/22-rdf-syntax-ns#type'
  and t2.s = t1.o
  and t2.p = p2.hash
  and t2.o = o2.hash
  and p2.lex = 'http://www.w3.org/1999/02/22-rdf-syntax-ns#type'

```

#### 4.4.1.4. Query 4

This query uses a randomized **productFeature**, which is indicated by a bold X.

```

SELECT
  o4.lex as product, o5.lex as reviewer, o6.lex as person
FROM
  spoTriples t1, Nodes s1, Nodes p1, Nodes o1,
  spoTriples t2, Nodes s2, Nodes p2,
  spoTriples t3, Nodes p3, Nodes o3,
  spoTriples t4, Nodes p4, Nodes o4,
  spoTriples t5, Nodes p5, Nodes o5,
  spoTriples t6, Nodes p6, Nodes o6
WHERE
  t1.s = s1.hash
  and t1.p = p1.hash
  and t1.o = o1.hash

```

```

and p1.lex =
  'http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/productFeature'
and o1.lex =
  'http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/ProductFeatureX'
and t2.s = s2.hash
and t2.p = p2.hash
and t2.o = t1.s
and p2.lex =
  'http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/reviewFor'
and t3.s = t2.s
and t3.p = p3.hash
and t3.o = o3.hash
and p3.lex = 'http://purl.org/stuff/rev#reviewer'
and t4.s = t1.s
and t4.p = p4.hash
and t4.o = o4.hash
and p4.lex = 'http://www.w3.org/2000/01/rdf-schema#label'
and t5.s = t2.s
and t5.p = p5.hash
and t5.o = o5.hash
and p5.lex = 'http://purl.org/dc/elements/1.1/title'
and t6.s = t3.o
and t6.p = p6.hash
and t6.o = o6.hash
and p6.lex = 'http://xmlns.com/foaf/0.1/name'

```

#### 4.4.1.5. Query 5

This query uses a randomized **productType** indicated by a bold **X**, and a randomized two digit country code indicated by the bold **Y**.

```

SELECT
  o6.lex as product, o7.lex as vendor, o5.lex as price
FROM
  spoTripleS t1, Nodes s1, Nodes p1, Nodes o1,
  spoTripleS t2, Nodes s2, Nodes p2,
  spoTripleS t3, Nodes s3, Nodes p3,
  spoTripleS t4, Nodes p4, Nodes o4,
  spoTripleS t5, Nodes p5, Nodes o5,
  spoTripleS t6, Nodes p6, Nodes o6,
  spoTripleS t7, Nodes p7, Nodes o7,
  spoTripleS t8, Nodes p8, Nodes o8
WHERE
  t1.s = s1.hash
  and t1.p = p1.hash
  and t1.o = o1.hash
  and p1.lex = 'http://www.w3.org/2000/01/rdf-schema#subClassOf'
  and o1.lex =
    'http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/ProductTypeX'
  and t2.s = s2.hash
  and t2.p = p2.hash
  and t2.o = t1.s
  and p2.lex = 'http://www.w3.org/1999/02/22-rdf-syntax-ns#type'
  and t3.s = s3.hash
  and t3.p = p3.hash

```

```

and t3.o = t2.s
and p3.lex =
  'http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/product'
and t4.s = t3.s
and t4.p = p4.hash
and t4.o = o4.hash
and p4.lex =
  'http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/vendor'
and t5.s = t3.s
and t5.p = p5.hash
and t5.o = o5.hash
and p5.lex =
  'http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/price'
and t6.s = t2.s
and t6.p = p6.hash
and t6.o = o6.hash
and p6.lex = 'http://www.w3.org/2000/01/rdf-schema#label'
and t7.s = t4.o
and t7.p = p7.hash
and t7.o = o7.hash
and p7.lex = 'http://www.w3.org/2000/01/rdf-schema#label'
and t8.s = t4.o
and t8.p = p8.hash
and t8.o = o8.hash
and p8.lex =
  'http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/country'
and o8.lex = 'http://downlode.org/rdf/iso-3166/countries#Y'

```

#### 4.4.1.6. Query 6

This query uses two appropriately randomized **productTypes** indicated by a bold **X** and a bold **Y**.

```

SELECT
  p1.lex as pred
FROM
  spoTriples t1, Nodes s1, Nodes p1, Nodes o1
WHERE
  t1.s = s1.hash
  and t1.p = p1.hash
  and t1.o = o1.hash
  and s1.lex =
    'http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/ProductTypeX'
  and o1.lex =
    'http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/ProductTypeY'

```

#### 4.4.2. Triple-Triple

The SQL queries used on the Triple-Triple architecture were identical to those used for the baseline, except that each of the table aliases for **spoTriples** needed to be chosen in a logical way from the set {**spoTriples**, **sopTriples**, **psopTriples**, **posTriples**, **ospTriples**, **opsTriples**}. For each query, a binding between table aliases (**t1**, **t2**, ...) and

table names (**spoTriples**, **sopTriples**, ...) is given along with a visual representation of the join relations in the form of a query multi-graph.

#### 4.4.2.1. Query 1

For query 1, the baseline architecture already provides a logical selection of tables for enabling fast merge joins. Therefore, the Triple-Triple architecture is unnecessary for defining a good set of tables for query 1. However, the Triple-Triple architecture does offer more possibilities. Therefore, the Triple-Triple version of query 1 will use the **sopTriples** table exclusively instead of the **spoTriples** table, which is the case with the baseline architecture. Any mixing of **spoTriples** and **sopTriples** would arguably be just as valid.

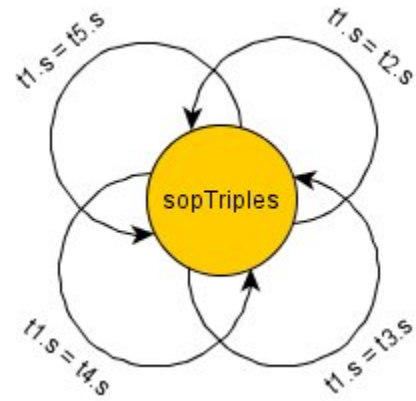


Figure 4: Query graph for query 1 in Triple-Triple architecture

#### 4.4.2.2. Query 2

This query was split into query 2a and 2b, as in the baseline.

##### 4.4.2.2.1. Query 2a

The challenge in dealing with query 2a is that **t3** is involved in a join both on its **s** column and on its **o** column. This presents a challenge in that it is uncertain whether the join on **s** or the join on **o** is more important to the speed of the query.

The following alias/table mapping was used: **t1/spoTriples**, **t2/spoTriples**, **t3/ospTriples**, **t4/spoTriples**. The **spoTriples** table is used for tables that only involve joins on **s**. The usage of the **ospTriples** table for **t3** treats **o** as the most important join column for that table, which is important for the join with **t4**.

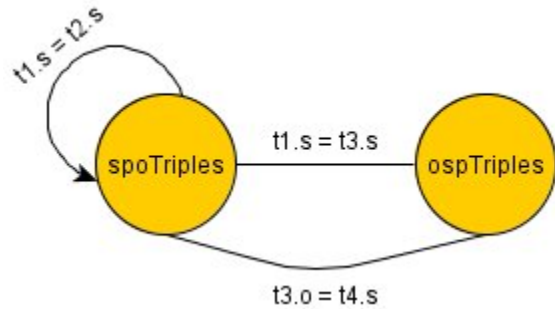


Figure 5: Query graph for query 2a in Triple-Triple architecture.

##### 4.4.2.2.2. Query 2b

Query 2b has problems similar to query 2a in that several table choices are possible. This is because tables **t3** and **t5** are involved in joins on both **s** and **o**. For this experiment, **ospTriples** is used instead of **spoTriples** for both **t3** and **t5**. All other tables use **spoTriples**.

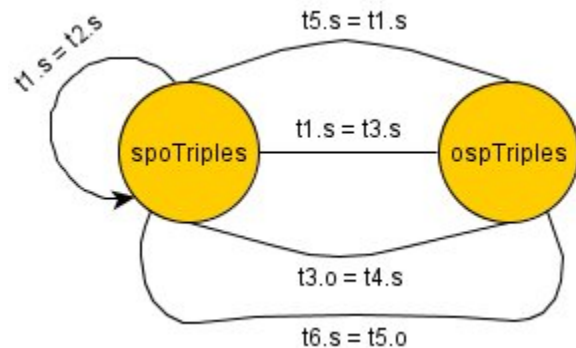


Figure 6: Query graph for query 2b in Triple-Triple architecture.

#### 4.4.2.3. Query 3

The query graph for query 3 is much simpler in comparison to the previous queries. There is only one join on the **s** column of one table and the **o** column of another, so an appropriate Triple-Triple table mapping is straight-forward. The **spoTriples** table is used for **t2**, and the **ospTriples** table is used for **t1**.

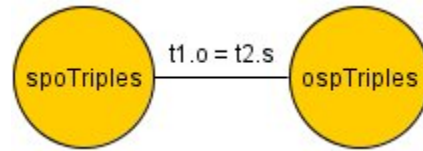


Figure 7: Query graph for query 3 in Triple-Triple architecture

#### 4.4.2.4. Query 4

Query 4 is slightly more complicated, and once again requires a choice to be made between several available options. The most challenging table designation is for **t2**, which joins on both **s** and **o**. However, since it is involved in two joins on **s** and only one on **o**, it is set to use **sopTriples**, as is every other table except for **t3**, which is only involved in a join on **o**, and therefore maps to **ospTriples**.

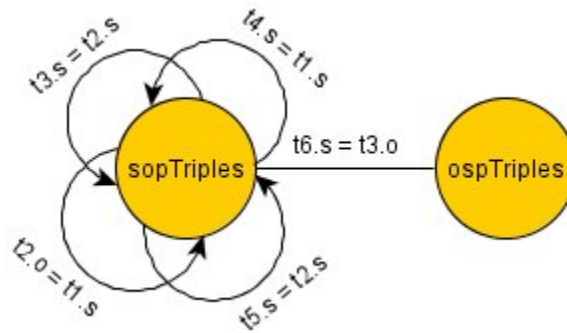


Figure 8: Query graph for query 4 in Triple-Triple architecture.

#### 4.4.2.5. Query 5

In this query the tricky tables are **t2**, **t3** and **t4**, all of which are involved in joins on **s** and **o**. The other tables are only joined on **s** and are therefore mapped to **sopTriples**. The alias **t4** is joined twice on **o** and only once on **s**, so it is mapped to **ospTriples**. The alias **t3** is joined twice on **s** and only once on **o**, so it is mapped to **sopTriples**. The choice of **sopTriples** for **t3** makes it impossible to merge join with **t2** without a sort no matter what **t2** maps to, so that join no longer matters in the choice of **t2**'s table. Of the remaining joins on **t2**, one is on **s** and the other is on **o**, so it's a toss up. We chose to map **t2** to **ospTriples**.

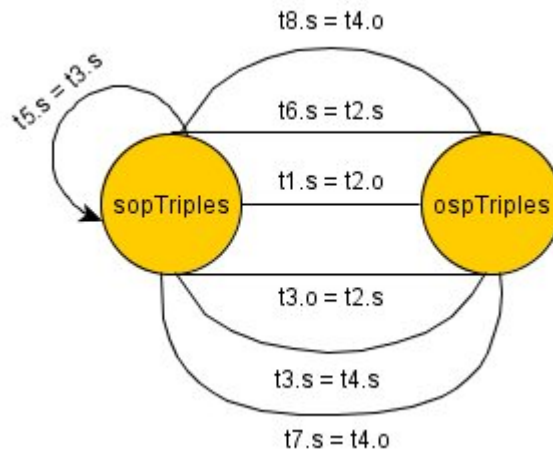


Figure 9: Query graph for query 5 in Triple-Triple architecture.

#### 4.4.2.6. Query 6

Query 6 actually doesn't involve any self joins on the triples table, so no query graph is necessary. The **sopTriples** table was arbitrarily chosen as the table to be used, simply so that this trial would be different from the baseline.

#### 4.4.3. Bitmapped

Translating queries into the bitmapped architecture was easy and straightforward. Starting from any baseline query, the bitmapped equivalent can be created by exchanging the **bmTriples** table for the **spoTriples** table, and by swapping each **Nodes** table with the appropriate table from **bmsNodes**, **bmpNodes** and **bmoNodes**. Which table is correct depends on which column is joined on in the WHERE clause of the SQL statement, though this information is also present in the alias for any given node table. For any given baseline query, a **Nodes** instance with an alias that starts with s (for example, **s1**) should be modified to reference **bmsNodes**. Aliases starting with p should reference **bmpNodes**, and aliases with o reference **bmoNodes**. Clearly, the process of defining queries in the bitmapped architecture is much easier than in the Triple-Triple architecture.

### 5. Results

The results show that the Triple-Triple architecture did not perform better than the baseline architecture, and that the bitmapped scheme did not produce the correct query results due to an error in Oracle.

The error with bitmap join indexes is examined in further detail in the discussion later. Timing results are excluded for the bitmapped architecture because, although times were obtained, these times are meaningless due to the error. For certain queries the bitmapped scheme produced correct results, but when the explain plans for these queries were examined, it was revealed that they were not even making use of the bitmap join indexes. When optimizer hints were used to change this, the queries failed.

The explain plans for the Triple-Triple queries indicated that, despite the tables of the Triple-Triple architecture being sorted, the optimizer always chose hybrid-hash joins over merge joins. Therefore, another set of trials were run for each Triple-Triple query in which merge joins between tables presorted on their join keys were forced using optimizer hints. The exception to this is query 6, which did not involve any self joins on the triples table.

Each query was executed 10 times. Times are reported in seconds. For each query and dataset size, boxplots are shown. Average execution times are summarized in figures 10 and 11. In all figures and tables, **BL** is for the baseline architecture, **TT** is for Triple-Triple and **TTMerge** is for Triple-Triple with merge joins forced.

<b>250K</b>	Q1	Q2a	Q2b	Q3	Q4	Q5	Q6
BL	8.1198	6.8767	6.0637	7.5128	7.8716	7.9859	6.3540
TT	8.9824	8.7969	9.4036	9.5568	8.5208	11.1477	7.9436
TTMerge	10.7045	10.4831	9.8249	9.0746	11.0372	15.8964	X

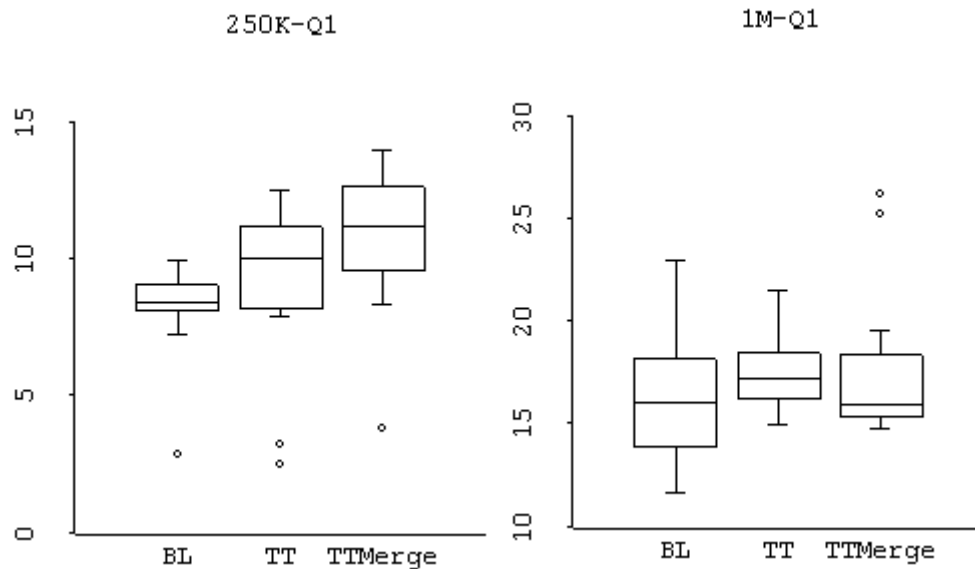
**Figure 10: Average execution time in seconds for 250K dataset.**

1M	Q1	Q2a	Q2b	Q3	Q4	Q5	Q6
BL	16.6966	12.2759	9.1385	13.4705	16.0538	17.5358	10.9217
TT	17.7871	15.9526	17.3831	17.4625	18.1476	31.3029	10.8373
TTMerge	18.1442	56.2659	77.1762	33.5929	87.7299	91.5671	X

**Figure 11: Average execution time in seconds for 1M dataset.**

In some cases, **BL**, **TT** and **TTMerge** are roughly the same, but when they are different, **TTMerge** generally takes longer than **TT**, which takes longer than **BL**. Sometimes the difference between **TT** and **TTMerge** is enormous.

### 5.1. Query 1

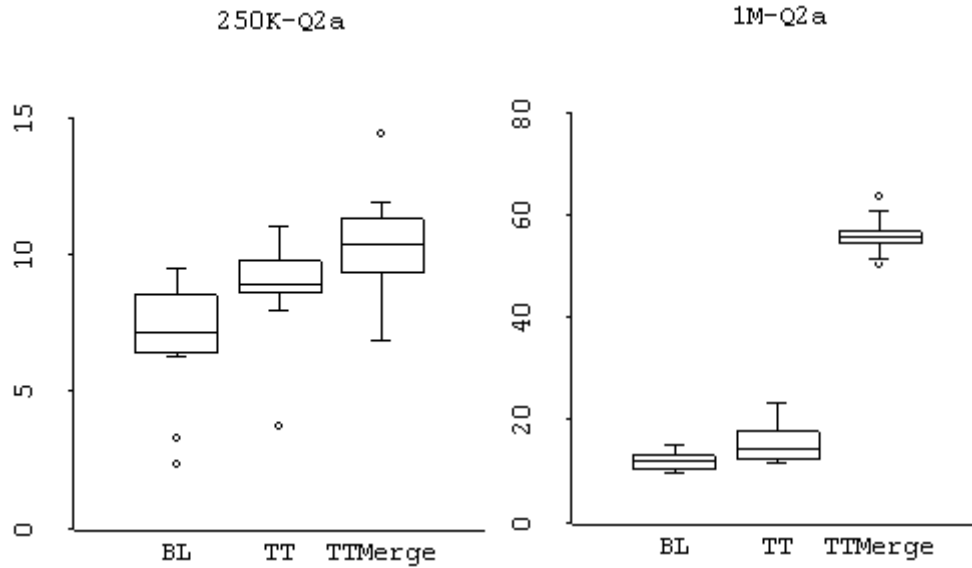


**Figure 12: Boxplots of execution times for query 1.**

There is little difference across architectures in the execution time of query 1. The Triple-Triple architecture is a bit slower on the 250K dataset, but is about the same on the 1M dataset. The differences are not large, and may be due in part to random variations. Recall that query 1 only used copies of the same table for both the baseline and the Triple-Triple architectures, so there is little reason to suspect a significant difference between them.

## 5.2. Query 2

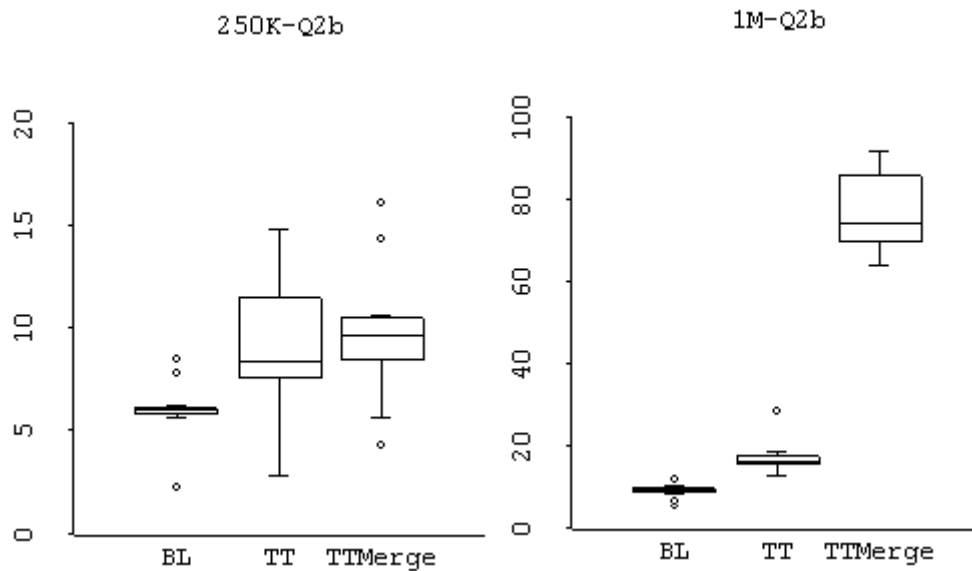
### 5.2.1. Query 2a



**Figure 13: Boxplots of execution times for query 2a.**

The Triple-Triple architecture is a clear loser in these trials, especially when merge joins are forced on the 1M dataset.

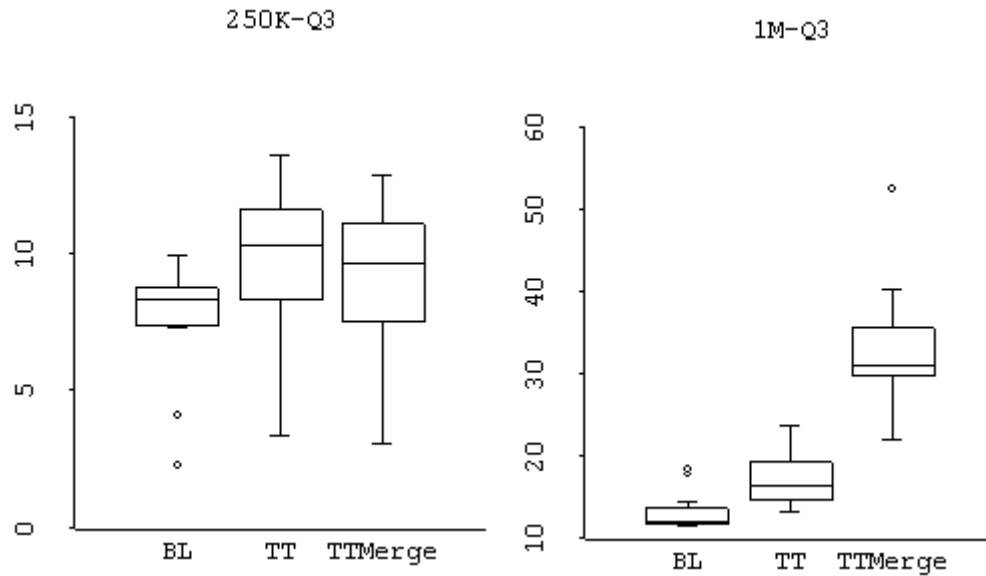
### 5.2.2. Query 2b



**Figure 14: Boxplots of execution times for query 2b.**

The trend for query 2b is essentially the same as for query 2a.

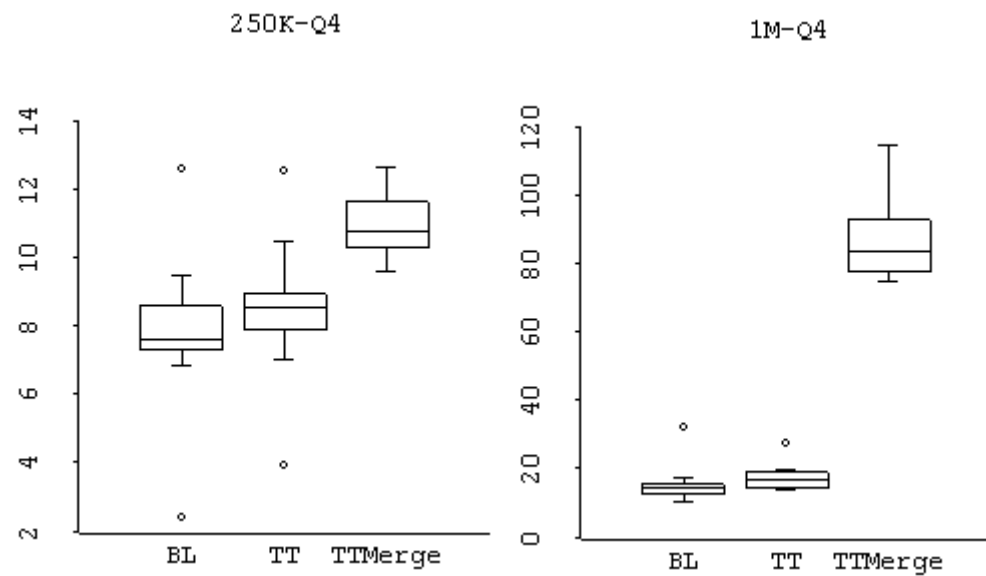
### 5.3. Query 3



**Figure 15: Boxplots of execution times for query 3.**

Query 3 was very simple in that it only had two tables, and the best manner in which to join them for the sake of fast merge joins was clear. However, even this query is slower in the Triple-Triple architecture, especially when the merge joins are forced on the 1M dataset.

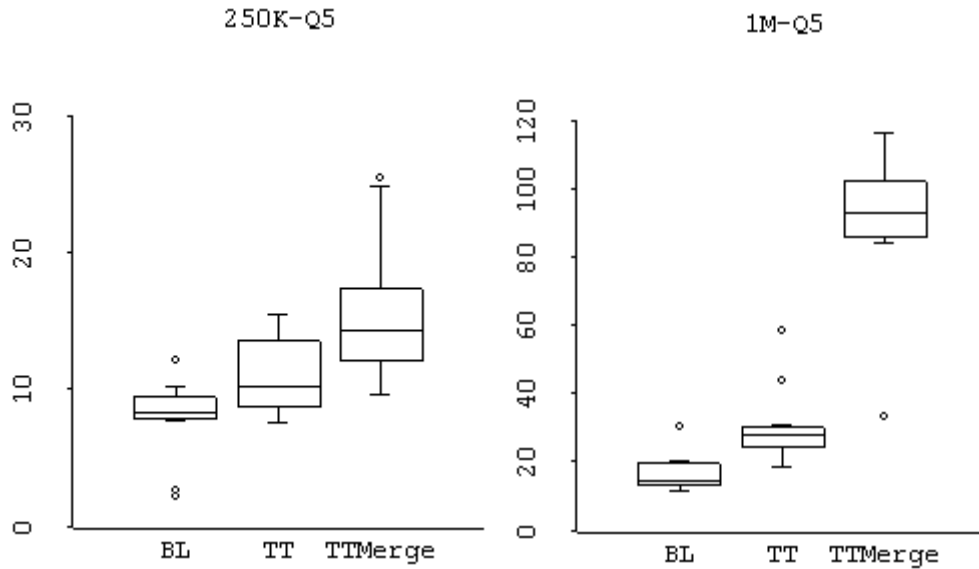
### 5.4. Query 4



**Figure 16: Boxplots of execution times for query 4.**

The trend continues with query 4: the Triple-Triple architecture is worse, especially when merge joins are forced. For this query, even the 250K dataset causes the forced merge joins to slow down.

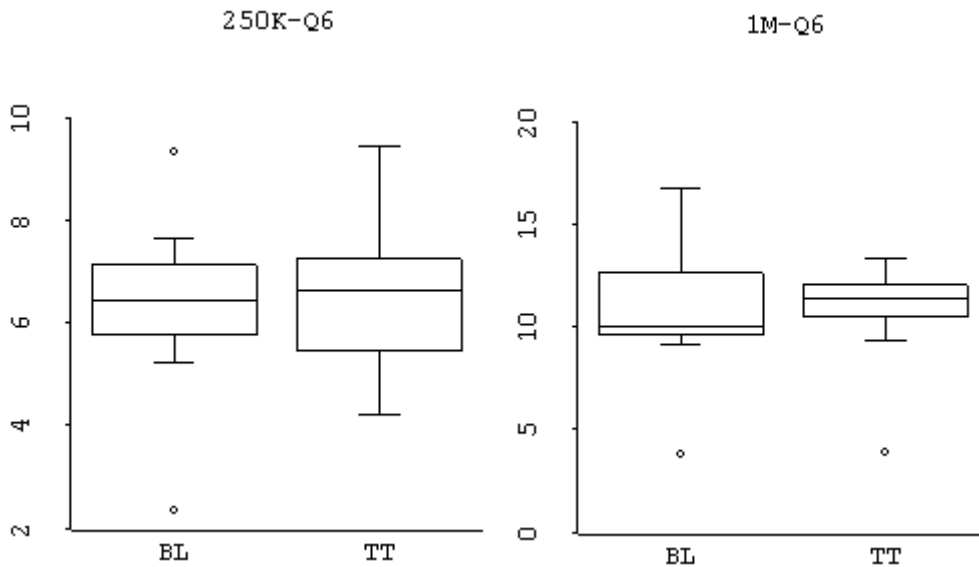
### 5.5. Query 5



**Figure 17: Boxplots of execution times for query 5.**

Query 5 shows a trend similar to that of previous queries.

### 5.6. Query 6



**Figure 18: Boxplots of execution times for query 6.**

Query 6 is very different from the previous queries in that it involves no joins between triples tables. This is why no attempt to force merge joins was made. The baseline and Triple-Triple architectures are essentially the same in this case, but use different tables. As expected, this did not make a large difference. Execution times are nearly the same across architectures.

## 6. Discussion

Neither of the proposed architectures worked as hoped. For each architecture, careful investigation indicates that the lack of success can be linked to unusual behavior on the part of the Oracle RDBMS.

### 6.1 Triple-Triple

First it should be stated that the six redundant copies of the triples table required of the Triple-Triple architecture seems completely unnecessary for our query set. If this query set is taken to be in any way representative of interesting queries, then it should be safe to eliminate several of the copies from the architecture.

The only tables used in the Triple-Triple queries of the experiments above were **spoTriples**, **sopTriples** and **ospTriples**. Of course, this might mean that the queries above did not make the best use of the available tables. Of course, the added difficulty of the query writing process has already been mentioned as a problem with the Triple-Triple architecture.

A bigger problem is why it does not work in the manner expected. The Oracle query optimizer never picked merge joins on its own for the queries presented in this paper. It always opted to use hybrid-hash joins instead. While it is true in general that a hybrid-hash join is quicker than a merge join, this should not be the case when data is already sorted on the join keys. In other words, the primary reason that hybrid-hash join is faster is that the sorting portion of the merge join is slow. The intent of the Triple-Triple architecture was to do merge joins without the sorting.

The results of the **TTMerge** trials show that Oracle was indeed justified in picking hybrid-hash joins over merge joins since the merge joins were actually slower. Looking at the explain plans for these queries, we can see the reason why. Here is the explain plan for query 3 on the 1M dataset:

Id	Operation	Name	Rows	Bytes	TempSpc	Cost (%CPU)	Time
0	SELECT STATEMENT		2992	2600K		29009 (2)	00:05:49
* 1	HASH JOIN		2992	2600K	2040K	29009 (2)	00:05:49
* 2	HASH JOIN		2992	2004K	1448K	23729 (2)	00:04:45
* 3	HASH JOIN		2992	1408K		18477 (2)	00:03:42
* 4	TABLE ACCESS FULL	NODES	1	204		2179 (2)	00:00:27
5	MERGE JOIN		119K	31M		16295 (2)	00:03:16
6	<b>SORT JOIN</b>		<b>25008</b>	<b>5885K</b>	<b>12M</b>	<b>5316 (2)</b>	<b>00:01:04</b>
* 7	HASH JOIN		25008	5885K		4003 (2)	00:00:49
* 8	TABLE ACCESS FULL	NODES	1	204		2179 (2)	00:00:27
9	INDEX FAST FULL SCAN	PKOSPTRIPLES	1000K	35M		1805 (2)	00:00:22
* 10	<b>SORT JOIN</b>		<b>1000K</b>	<b>35M</b>	<b>91M</b>	<b>10979 (2)</b>	<b>00:02:12</b>
11	INDEX FAST FULL SCAN	PKSPOTRIPLES	1000K	35M		1162 (3)	00:00:14
12	TABLE ACCESS FULL	NODES	292K	56M		2176 (1)	00:00:27
13	TABLE ACCESS FULL	NODES	292K	56M		2176 (1)	00:00:27

First realize that the **HASH JOIN** operations present in this plan are performed for joins on the **Nodes** table. The join between **spoTriples** and **ospTriples** is done with a merge join because it has been forced with an optimizer hint. However, despite the fact that these tables are already sorted in the desired fashion, the **SORT JOIN** (red and bold) operator is used on each immediately before the **MERGE JOIN**. According to the Oracle specification, the **SORT JOIN** operator should only be used if the data is not already sorted. So, the reason that **TTMerge** takes longer than **TT** and **BL** is that it is unnecessarily incurring the I/O costs of a sort operation. Why the Oracle optimizer does not recognize that the data is sorted is a mystery.

To be absolutely sure that the data was in fact sorted, a quick and simple test was done. Here is the query plan for selecting all triples from the **spoTriples** table ordered by **s, p, o**:

```
EXPLAIN PLAN FOR
SELECT * FROM spoTriples ORDER BY s,p,o;
```

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		1000K	35M	5222 (1)	00:01:03
1	INDEX FULL SCAN	PKSPOTRIPLES	1000K	35M	5222 (1)	00:01:03

As we would expect, ordering by **s, p, o** does not incur a sort operation, because the data is already sorted in that order. However, if we choose to sort by some other order, the sort operation will appear in the explain plan:

```
EXPLAIN PLAN FOR
SELECT * FROM spoTriples ORDER BY o;
```

Id	Operation	Name	Rows	Bytes	TempSpc	Cost (%CPU)	Time
0	SELECT STATEMENT		1000K	35M		10979 (2)	00:02:12
1	SORT ORDER BY		1000K	35M	91M	10979 (2)	00:02:12
2	INDEX FAST FULL SCAN	PKSPOTRIPLES	1000K	35M		1162 (3)	00:00:14

Here the appearance of the sort operation is both justified and expected. So, although the Oracle optimizer is in theory capable of detecting and using the fact that the tables are sorted to its advantage, it fails to do so. This is why the Triple-Triple architecture fails in Oracle.

## 6.2 Bitmapped

The bitmapped architecture also fails, but for completely different and inexplicable reasons. For some reason, defining bitmap join indexes on the nodes table as proposed in this paper causes Oracle to return incorrect results whenever the bitmap join indexes are used. Specifically, whenever a bitmap join index is part of the query plan, the outcome of the query is an empty result.

This is an exceptional claim, so steps were taken to thoroughly understand and verify this phenomenon. First, it should be stated that some queries on the bitmapped

architecture did return results, notably query 1 and query 3. The explain plans for these queries were examined, and it was revealed that neither plan made use of any bitmap join index. When these query plans were forced to use the bitmap join indexes using optimizer hints, the queries returned empty results like all the others. Likewise, if the query plans for those queries that initially returned empty results were modified so that they were not allowed to use the bitmap join indexes, they would also return results. These successful forced results generally took slightly longer to execute than the baseline architecture.

To further verify that it was the bitmap join indexes that were causing the problem, query 2a was executed to obtain an empty result. Immediately afterwards, the bitmap join indexes involved in the query were dropped, and the query was run again. This time it returned the expected result. The bitmap join index was then recreated using the same SQL statement originally used to create it, and the query was executed again. Once again, it returned an empty result.

As a last resort, I upgraded to Oracle 11g to see if the bug persisted in Oracle's latest release, but the bug was still there.

Since I have no visibility on Oracle's source code, I can offer no explanation as to what causes this bug, but based on what seems to be standard practice among Oracle users (hearsay from blogs and forums), I can offer some speculation as to what might be causing it.

Firstly, I have found no mention of this or any similar bugs, so the triplestore architecture proposed must have some peculiar properties uncommon to most systems that make use of bitmap join indexes. Standard practice with bitmap indexes is to only apply them to columns with low cardinality, which is certainly not the case with the nodes table, whose **lex** values are unique. However, this alone cannot be the cause since even Oracle acknowledges that bitmap indexes can be useful on columns with unique values<sup>3</sup>. Of course, there is a possibility that the uniqueness issue combined with the size of the table is a problem.

Another potential issue in defining bitmap join indexes on the **lex** column of the nodes table is that the values in **lex** are in some cases extremely long strings. I specified the type of the **lex** column as **VARCHAR2(4000)** in order to support the data that needed to be contained in the nodes table. Perhaps the bitmap join index balks at being based on a **VARCHAR2** column with such long values.

Whatever the cause of the error, it is unfortunate that Oracle has allowed it to persist in their RDBMS. Returning incorrect query results under any circumstances is unacceptable.

## 7. Conclusion

Neither of the architectures proposed in this paper worked properly when applied to the Oracle RDBMS. However, in both cases the cause of the problem was strange behavior by Oracle. In theory, the ideas proposed for these architectures are sound, and have the potential to speed up access to triplestore data in the future, if implemented in systems that exploit the architectures properly.

---

<sup>3</sup> [http://www.oracle.com/technology/pub/articles/sharma\\_indexes.html](http://www.oracle.com/technology/pub/articles/sharma_indexes.html)

If the Triple-Triple idea were to be implemented in a system that performed merge joins without unnecessary sorting, the potential for speed up would be at least equal to that of the Hexastore. It should be noted that in the original presentation of the Hexastore, a custom database written in Python was used rather than any commercial RDBMS. It remains to be seen if any RDBMS supporting merge joins knows how to properly take advantage of pre-sorted triples tables, but doing so should, in theory, be very easy.

The use of bitmap join indexes also has great potential to speed up access to triplestores, but only if they work properly. Since Oracle is currently the only RDBMS that supports persistent bitmap indexes, it is either up to them to fix this bug, or to a competitor to incorporate bitmap join indexes into their repertoire of indexing methods.

Until these problems are fixed, those looking to have fast access to triplestores will need to look to alternatives methods.

## References

Daniel J. Albadì, Adam Marcus, Samuel R. Madden and Kate Hollenbach, Scalable Semantic Web Data Management Using Vertical Partitioning, Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB-07) (2007).

Jeen Broekstra and Arjohn Kampman, Sesame: A generic architecture for storing and querying RDF and RDF Schema, Technical report, Administrator Nederland b.v., October 2001, <http://sesame.aidministrator.nl/publications/del10.pdf>.

Theodore Johnson, Performance Measures of Compressed Bitmap Indices, Proceedings of the 25th International Conference on Very Large Data Bases (VLDB-99) (1999).

Stephen Harris and Nicholas Gibbins, 3store: Efficient Bulk RDF Storage, Conference for Practical and Scalable Semantic Systems (PSSS-03) (2003).

Kilian Stoffel, Merwyn Taylor and James Hendler, Efficient management of very large ontologies, Proceedings of American Association for Artificial Intelligence Conference (AAAI-97) (1997).

Cathrin Weiss, Panagiotis Karras and Abraham Bernstein, Hexastore: Sextuple Indexing for Semantic Web Data Management, Proceedings of the 34th International Conference on Very Large Data Bases (VLDB-08) (2008).