

# Database Tuning for SDB (/MySQL)

This is a work in progress / discussion starter. It is not - currently, at least - intended to provide a comprehensive view of tuning databases. Instead, this is presented as a set of observations and techniques to inform implementing sites on aspects they may wish to follow.

## Introduction

Search the web, and you'll find a number of resources comparing the use of SPARQL on various triples stores - e.g. BigOWLIM, Virtuoso, Jena TDB and even Jena SDB - using standard tests such as the Berlin SPARQL Benchmark. These usually show SDB (on MySQL) as being the slowest, and even the Jena project states that SDB is "deprecated", suggesting to use TDB (this may be more of a reflection of committer activity and interest than really technical capability).

However, this may belie what is happening under surface - for instance, whilst Virtuoso is often shown as being about twice as fast for querying (and a lot faster at loading) than TDB, it actually has a Quad table layout with hashed columns much like SDB at it's core. So what makes it so much faster than SDB, or even TDB faster than SDB?

One advantage that "dedicated" triple stores (or rather, engines that support quads / triples explicitly), may be optimised by default for the specific layout and typical queries used. Whereas SDB relies on external, generic SQL stores, which won't - out of the box - be tuned to only have large Quad tables.

## Software and Hardware Used

(At the time of writing) All of the work presented here was conducted on a 2015 MacBook Pro - 2.5Ghz i7, 16GB RAM, exceedingly fast 500GB SSD storage.

MySQL 5.6.26 Community Server, as downloaded from <http://dev.mysql.com/> was used as the SQL back end.

## Dataset

A "realistic" VIVO-ISF triple dataset was loaded, consisting of over 24 million triples (asserted and inferred combined), 140,000 people, and 155,000 publications.

## Out of the Box

Simply using the out of the box configuration of MySQL, performance on this dataset was largely encouraging. With VIVO 1.8.1, the majority of profile pages rendered in under 2 seconds, and the worst case ones - approximately 1500 publications - still took under 5 seconds. It's likely that much of this can be attributed to the performance of the SSD storage being well equipped to support IO bound tasks.

However, even with reduced overheads and better querying in VIVO 1.8.1, there were still some examples, where - on the default settings - pages would take a longer time to render. For example:

*Large profile Co-Author network: 20 seconds*

*Map of Science / Temporal Graph cache warming: 2 minutes*

Compared to VIVO 1.8, these represent reasonable performances - the same co-author network in 1.8 took 10 minutes to fully render the Flash page, and the Map of Science / Temporal Graph visualization models took too long to build to actually measure.

But even so, what actually is going on under the hood?

## In Depth: Co-Author Network Visualization

This is one area that received a couple of benefits in VIVO 1.8.1:

- 1) RDFService was used directly to execute SPARQL, rather than a query on a service backed Dataset proxy (1 query vs many finds)
- 2) Split of the SELECT into CONSTRUCT and SELECT.

## CONSTRUCT vs SELECT

It has - for example, in the context of the list view configurations - been presented that using a CONSTRUCT rather than direct SELECT is an SDB optimisation, where OPTIONAL queries are involved. That may only be partially true. Certainly - and very much so in an out of the box configuration - SDB can struggle with SELECTs that other engines handle more easily.

However, contrary to speculation on support forums for the triple stores, CONSTRUCT statements with large numbers of UNIONS are not particularly bad for alternative triple stores. Testing the VIVO 1.8.1 authorship queries directly against Virtuoso's SPARQL endpoint for a large profile gave results of 1271ms for the CONSTRUCT, and 1396ms for the SELECT. It is still faster to generate the temporary small model from Virtuoso than it is to execute the necessary SELECT directly.

True, there may not be much in it for a triple store that can optimize the SELECT to more aggressively, but it isn't really any worse off. Logically, this shouldn't be surprising, as it should be easier for any triple store to recognise and optimise the use of the same basic graph pattern multiple times in the same query, than it is to determine the selectivity of two BGPs and order their execution efficiently.

So, whilst the introduction of a CONSTRUCTed model in the Co-Author network may be particularly helpful for SDB, it's not specifically an SDB optimisation, and should provide the best, and most predictable performance across the widest range of triple stores.

This leads to the expensive part of the Co-Authorship Network Visualisation being the CONSTRUCT query to generate the temporary model (the SELECT on this in memory model is only a couple of hundred milliseconds).

The SPARQL query that is being executed takes the form:

Query for profile: <http://localhost/individual/author>

```
PREFIX foaf: <>
PREFIX core: <>
PREFIX rdfs: <>
PREFIX local: <http://localhost/>
CONSTRUCT
{
  <http://localhost/individual/author> local:authorLabel ?authorLabel .
  <http://localhost/individual/author> local:authorOf ?document .
  ?document local:publicationDate ?publicationDate .
  ?document local:coAuthor ?coAuthorPerson .
  ?coAuthorPerson rdfs:label ?coAuthorPersonLabel .
}
WHERE
{
  {
    <http://localhost/individual/author> rdf:type foaf:Person ;
    rdfs:label ?authorLabel ;
    core:relatedBy ?authorshipNode .
    ?authorshipNode rdf:type core:Authorship ;
    core:relates ?document .
    ?document rdf:type <http://purl.obolibrary.org/obo/IAO_0000030> ;
    core:relatedBy ?coAuthorshipNode .
    ?coAuthorshipNode rdf:type core:Authorship ;
    core:relates ?coAuthorPerson .
    ?coAuthorPerson rdf:type foaf:Person ;
    rdfs:label ?coAuthorPersonLabel .
  }
  UNION
  {
    <http://localhost/individual/author> rdf:type foaf:Person ;
    rdfs:label ?authorLabel ;
    core:relatedBy ?authorshipNode .
    ?authorshipNode rdf:type core:Authorship ;
    core:relates ?document .
    ?document core:dateTimeValue ?dateTimeValue .
    ?dateTimeValue core:dateTime ?publicationDate .
  }
}
```

The quirk of "local"

Notably, the above query uses a "local" prefix. In VIVO, most CONSTRUCT statements are written to generate models that match the underlying source ontologies, such that the SELECT queries executed on those models could also just as easily run against the source triple store directly.

If you know that you are always going to be executing against a CONSTRUCTed temporary model - and when the CONSTRUCT is usually the best case, you might as well commit to doing it - then there is no need to replicate the original model in the reduced set, when you can easily collapse certain graphs into more descriptive "invented" predicates, reducing the model size, reducing the complexity and increasing the specificity of the following SELECT.

## Going Deeper

Most of the optimisation work that went into the VIVO 1.8.1 release depended on using the YourKit profiler (<http://yourkit.com/>), pointing it at a running server and getting statistics for where code is taking a long time to execute.

One of the nice features of YourKit is that it can profile SQL statements as well as the Java functions - it doesn't go to the level of telling you what is happening inside the SQL engine itself (that would require an SQL profiler in the database server), but it does tell you what queries are being executed, and how long they take.

### YourKit SQL Probes

In order to profile SQL statements, YourKit uses a technology they call "probes". By default, when you integrate YourKit with an enterprise server (e.g. Tomcat), the startup line that loads the agent will have the profiles disabled: "probe\_disable=". By removing this part of the line, it will use the default probe settings, which includes profiling SQL statements when the CPU profiling is enabled.

Initially, it was the profiling of SQL statements that alerted us to potential problems being caused by RDFService being hidden behind a Dataset proxy - when what was a single SPARQL query in the code seemed to cause hundreds of SQL queries to be executed in the back end.

That explosion of queries was actually caused by the execution of SPARQL against the proxy dataset being translated into find(s) that would then be executed as (individually, very efficient) SELECT queries on the RDFService.

By taking the proxy out of the equation, and getting the SPARQL executed directly against the SDB backend, the above CONSTRUCT was now being executed as a single - albeit large! - SQL statement:

```
SELECT
    -- V_1=?authorLabel V_2=?coAuthorshipNode V_3=?document V_4=?
authorshipNode V_5=?coAuthorPerson V_6=?coAuthorPersonLabel
  R_1.lex AS V_1_lex, R_1.datatype AS V_1_datatype, R_1.lang AS V_1_lang, R_1.type AS V_1_type,
  R_2.lex AS V_2_lex, R_2.datatype AS V_2_datatype, R_2.lang AS V_2_lang, R_2.type AS V_2_type,
  R_3.lex AS V_3_lex, R_3.datatype AS V_3_datatype, R_3.lang AS V_3_lang, R_3.type AS V_3_type,
  R_4.lex AS V_4_lex, R_4.datatype AS V_4_datatype, R_4.lang AS V_4_lang, R_4.type AS V_4_type,
  R_5.lex AS V_5_lex, R_5.datatype AS V_5_datatype, R_5.lang AS V_5_lang, R_5.type AS V_5_type,
  R_6.lex AS V_6_lex, R_6.datatype AS V_6_datatype, R_6.lang AS V_6_lang, R_6.type AS V_6_type
FROM
  ( SELECT DISTINCT
    -- ?coAuthorPerson:(Q_9.o=>S_1.X_1) ?document:(Q_5.o=>S_1.X_2) ?
coAuthorshipNode:(Q_7.o=>S_1.X_3) ?authorLabel:(Q_2.o=>S_1.X_4) ?coAuthorPersonLabel:(Q_11.o=>S_1.X_5) ?
authorshipNode:(Q_3.o=>S_1.X_6)
    Q_9.o AS X_1,
    Q_5.o AS X_2,
    Q_7.o AS X_3,
    Q_2.o AS X_4,
    Q_11.o AS X_5,
    Q_3.o AS X_6
  FROM
    Quads AS Q_1
  rdfs:type foaf:Person
  INNER JOIN
    Quads AS Q_2
  rdfs:label ?authorLabel
    ON ( Q_1.s = -4883577620769971978 -- Const: <http://localhost/individual/author>
    AND Q_1.p = -6430697865200335348 -- Const: rdfs:type
    AND Q_1.o = -1118181488561280847 -- Const: foaf:Person
    AND Q_2.s = -4883577620769971978 -- Const: <http://localhost/individual/author>
    AND Q_2.p = 6454844767405606854 -- Const: rdfs:label
    )
  INNER JOIN
    Quads AS Q_3
  core:relatedBy ?authorshipNode
    ON ( Q_3.s = -4883577620769971978 -- Const: <http://localhost/individual/author>
    AND Q_3.p = 7813032771907687750 -- Const: core:relatedBy
    )
  INNER JOIN
    Quads AS Q_4
  Authorship
    ON ( Q_4.p = -6430697865200335348 -- Const: rdfs:type
```

```

AND Q_4.o = -7985466041922445122 -- Const: core:Authorship
AND Q_3.o = Q_4.s -- Join var: ?authorshipNode
)
INNER JOIN
  Quads AS Q_5 -- <urn:x-arq:DefaultGraphNode> ?authorshipNode core:relates ?document
ON ( Q_5.p = -3633326295402292183 -- Const: core:relates
AND Q_3.o = Q_5.s -- Join var: ?authorshipNode
)
INNER JOIN
  Quads AS Q_6 -- <urn:x-arq:DefaultGraphNode> ?document rdfsyn:type <http://purl.
obolibrary.org/obo/IAO_0000030>
ON ( Q_6.p = -6430697865200335348 -- Const: rdfsyn:type
AND Q_6.o = 1885280957395725387 -- Const: <http://purl.obolibrary.org/obo/IAO_0000030>
AND Q_5.o = Q_6.s -- Join var: ?document
)
INNER JOIN
  Quads AS Q_7 -- <urn:x-arq:DefaultGraphNode> ?document core:relatedBy ?
coAuthorshipNode
ON ( Q_7.p = 7813032771907687750 -- Const: core:relatedBy
AND Q_5.o = Q_7.s -- Join var: ?document
)
INNER JOIN
  Quads AS Q_8 -- <urn:x-arq:DefaultGraphNode> ?coAuthorshipNode rdfsyn:type core:
Authorship
ON ( Q_8.p = -6430697865200335348 -- Const: rdfsyn:type
AND Q_8.o = -7985466041922445122 -- Const: core:Authorship
AND Q_7.o = Q_8.s -- Join var: ?coAuthorshipNode
)
INNER JOIN
  Quads AS Q_9 -- <urn:x-arq:DefaultGraphNode> ?coAuthorshipNode core:relates ?
coAuthorPerson
ON ( Q_9.p = -3633326295402292183 -- Const: core:relates
AND Q_7.o = Q_9.s -- Join var: ?coAuthorshipNode
)
INNER JOIN
  Quads AS Q_10 -- <urn:x-arq:DefaultGraphNode> ?coAuthorPerson rdfsyn:type foaf:Person
ON ( Q_10.p = -6430697865200335348 -- Const: rdfsyn:type
AND Q_10.o = -1118181488561280847 -- Const: foaf:Person
AND Q_9.o = Q_10.s -- Join var: ?coAuthorPerson
)
INNER JOIN
  Quads AS Q_11 -- <urn:x-arq:DefaultGraphNode> ?coAuthorPerson rdfs:label ?
coAuthorPersonLabel
ON ( Q_11.p = 6454844767405606854 -- Const: rdfs:label
AND Q_9.o = Q_11.s -- Join var: ?coAuthorPerson
)
) AS S_1 -- ?coAuthorPerson:(Q_9.o=>S_1.X_1) ?document:(Q_5.o=>S_1.X_2) ?
coAuthorshipNode:(Q_7.o=>S_1.X_3) ?authorLabel:(Q_2.o=>S_1.X_4) ?coAuthorPersonLabel:(Q_11.o=>S_1.X_5) ?
authorshipNode:(Q_3.o=>S_1.X_6)
LEFT OUTER JOIN
  Nodes AS R_1 -- Var: ?authorLabel
ON ( S_1.X_4 = R_1.hash )
LEFT OUTER JOIN
  Nodes AS R_2 -- Var: ?coAuthorshipNode
ON ( S_1.X_3 = R_2.hash )
LEFT OUTER JOIN
  Nodes AS R_3 -- Var: ?document
ON ( S_1.X_2 = R_3.hash )
LEFT OUTER JOIN
  Nodes AS R_4 -- Var: ?authorshipNode
ON ( S_1.X_6 = R_4.hash )
LEFT OUTER JOIN
  Nodes AS R_5 -- Var: ?coAuthorPerson
ON ( S_1.X_1 = R_5.hash )
LEFT OUTER JOIN
  Nodes AS R_6 -- Var: ?coAuthorPersonLabel
ON ( S_1.X_5 = R_6.hash )

```

*Note: The hashes in the query above are not necessarily the correct hashes for the actual predicates / objects, but the overall statement is correct.*

**SQL Execution time:** As noted at above, this query was taking - for a particular large profile - 20-ish seconds, and accounting for almost all of the time taken to generate the co-author visualisation.

## Tweaking MySQL

Knowing the exact query that is being executed against the database allows us to actually focus on the database, see how the query is being executed and make configuration changes that help it to support the actual data structures and workload being demanded of it.

The first thing to note is that (executing the SQL directly) the performance of the query is very consistent. Knowing that it returns quite a large amount of data (approx 200,000 rows), and has a significant amount of work to do, this suggests that the caches aren't necessarily being used effectively.

So, lets increase the join, read and innodb buffers. If you've installed MySQL on a Mac as described above, you'll need to create a my.cnf in /etc, otherwise edit the my.cnf that is being used:

```
join_buffer_size = 32M
read_rnd_buffer_size = 32M
innodb_buffer_pool_size = 1536M
innodb_file_per_table = 1
innodb_file_format = barracuda
```

NB: There are also some tweaks to the InnoDB file layout - these are good practice things to have.

Whilst this does not affect the initial execution of the SQL query (20 seconds), it does make subsequent executions for the same profile faster - approximately 14 seconds. It may be that innodb\_buffer\_pool\_size is the main reason for the performance boost - having a sufficiently large buffer pool means that InnoDB can generate in-memory adaptive hash indexes. Hash indexes (which are only available in memory) are far more efficient for the types of joins that SDB is doing, which are all based on hash values anyway.

There are a number of different philosophies for sizing the innodb\_buffer\_pool\_size. On a dedicated server, MySQL recommends 70% of the available memory. If you are hosting Tomcat on the same server, then maybe that should be no more that 70% of the system memory minus the max Tomcat heap size.

Actual requirements depends on data shape and usage patterns - for the SDB layout, after accessing a number of different pages in VIVO and checking the free buffers, a good setting seems to be around 1MB for every 16,000 triples - e.g. 1.5GB for a 25million triple store. If you have the resources, increasing the buffer size doesn't hurt performance - in which case allocating at least 100MB per 1million triples would be a good idea.

The InnoDB buffer pool can be pre-loaded at startup, which would help the initial query performance times - see: <https://dev.mysql.com/doc/refman/5.7/en/innodb-preload-buffer-pool.html>

*It's not known what size buffers should really be used for any given database size, or how well this translates into overall performance, e.g. how much can be cached to the benefit of the entire application. Of course, each individual site would need to look at the resources available on their own systems, and what they can dedicate to a given purpose.*

## Even Deeper Explanation

So far, we've just scratched the surface of MySQL, and put in a basic tweak to the configuration, but we haven't really looked in depth at what the query is actually doing.

Like other database engines, MySQL provides execution plans for queries, so lets go ahead and ask MySQL to give us the execution plan - by adding EXPLAIN to the start of the query.

Doing so, gives the following output:

id	select_type	table	type	possible_keys	key	key_len	ref	rows	extra
1	PRIMARY	<derived2>	ALL					852	
1	PRIMARY	R_1	eq_ref	PRIMARY	PRIMARY	8	S_1.X_4	1	
1	PRIMARY	R_2	eq_ref	PRIMARY	PRIMARY	8	S_1.X_3	1	
1	PRIMARY	R_3	eq_ref	PRIMARY	PRIMARY	8	S_1.X_2	1	
1	PRIMARY	R_4	eq_ref	PRIMARY	PRIMARY	8	S_1.X_6	1	
1	PRIMARY	R_5	eq_ref	PRIMARY	PRIMARY	8	S_1.X_1	1	
1	PRIMARY	R_6	eq_ref	PRIMARY	PRIMARY	8	S_1.X_5	1	
2	DERIVED	Q_1	ref	SubjPredObj,PredObjSubj,ObjSubjPred	SubjPredObj	24	const,const,const	1	Using where; Using index; Using temporary
2	DERIVED	Q_2	ref	SubjPredObj,PredObjSubj	SubjPredObj	16	const,const	1	Using where; Using index
2	DERIVED	Q_3	ref	SubjPredObj,PredObjSubj,ObjSubjPred	SubjPredObj	16	const,const	852	Using where; Using index
2	DERIVED	Q_4	ref	SubjPredObj,PredObjSubj,ObjSubjPred	SubjPredObj	24	Q_3.o,const,const	1	Using where; Using index

2	DERIVED	Q_5	ref	SubjPredObj,PredObjSubj,ObjSubjPred	SubjPredObj	16	Q_3.o,const	1	Using where; Using index
2	DERIVED	Q_6	ref	SubjPredObj,PredObjSubj,ObjSubjPred	SubjPredObj	24	Q_5.o,const,const	1	Using where; Using index
2	DERIVED	Q_7	ref	SubjPredObj,PredObjSubj,ObjSubjPred	SubjPredObj	16	Q_5.o,const	1	Using index
2	DERIVED	Q_8	ref	SubjPredObj,PredObjSubj,ObjSubjPred	SubjPredObj	24	Q_7.o,const,const	1	Using where; Using index; Distinct
2	DERIVED	Q_9	ref	SubjPredObj,PredObjSubj,ObjSubjPred	SubjPredObj	16	Q_7.o,const	1	Using where; Using index; Distinct
2	DERIVED	Q_10	ref	SubjPredObj,PredObjSubj,ObjSubjPred	SubjPredObj	24	Q_9.o,const,const	1	Using where; Using index; Distinct
2	DERIVED	Q_11	ref	SubjPredObj,PredObjSubj	SubjPredObj	16	Q_9.o,const	1	Using index; Distinct

So, what does this tell us about the query? Well, first of all it looks like there is very good index use, so the most obvious thing you look for - index usage - is already covered.

But look carefully at the 8 row - the entry for table Q\_1 - what is that it says at the end? "Using temporary".

Under certain circumstances, MySQL can use a temporary table to answer DISTINCT queries, and when we look back at the original SQL, there is a SELECT DISTINCT on Q\_1. Now, as an experiment, if we remove the DISTINCT, we actually get the same results from the query - but it returns in a little over 6 seconds. SIX SECONDS! This query originally took 20...

But we can't just remove the DISTINCT - because that's part of the SQL generated from SDB, and we would need to go in and modify SDB. Hmm. So what can we do about the temporary table?

It's not clear what is going into the temporary table, and therefore how big it is. But knowing that it is select 4 hashes per row, each hash is 8 bytes, and there are 200,000 rows, that's **at least** 9M. Possibly up to 67M (11 tables, 4 hashes per table, 200,000 rows).

The amount of memory allocated by default to temporary tables is 16M, after which it creates temporary tables on disk. The temporary table above being written to disk is certainly going to kill performance!

There are two settings in my.cnf that need to be changed to alter the available temporary table size:

```
max_heap_table_size=256M
tmp_table_size=256M
```


It's important to note that this is not a recommendation for a temporary table size, we're just testing the hypothesis - we may be able to get away with smaller. Alternatively, your system may need larger.

But for now, we've put in a big number, which should be sufficient for the query above. So how long does the query take?

SIX SECONDS.

From cold. Repeat the query with warmed caches, and it dips under 5 seconds. Even with the DISTINCT present. For a query that took 20 seconds originally.

#### Unused Tweaks

 Reading the documentation on MySQL, there is in 5.6 a feature called "batched key access", which on the surface should help with large joins. However, when tested on this query, it appeared to make no difference - there was no change in either execution plan (it would say *Using join buffer (Batched Key Access)*) or in the execution time.

Even with a large query with OPTIONALS (e.g. the listview-authorInAuthorship select [\[view the Author in Authorship Query as SQL\]](#)), MySQL can't be provoked into using batched key access on a join. It *does* however start doing batched nested loops. This isn't hideous - under 4 seconds on a 1700+ publication profile - but not quite as fast as Virtuoso (2 seconds).

Despite the SDB layout using hashes for keys, and all joins based on equivalence, the architecture of MySQL is quite heavily leveraged on BTREE indexes, rather than hash indexes. MariaDB (a MySQL fork) adds batched hash joins to those offered by MySQL - although when MariaDB was provoked into using them, observed performance was awful.

**MariaDB** uses Aria engine for temporary tables by default, in those cases setting the `aria_pagecache_buffer_size` can make a profound difference. For the authorship query, execution time dropped as low as 2.5 seconds for repeated executions.

## Returning to Map of Science

Without an in depth look at the queries being executed by the Map of Science / Temporal Graph, what difference has the new MySQL settings made?

Originally, "out of the box", the cached model took a little over 2 minutes to build. With the new settings in place - and no other changes - it takes just 25 seconds.

## Conclusion

Despite the bad press, it is evident that you can make a LOT of difference to the performance of SDB by taking time to look at optimisations that can be made to the SQL engine. That's the trade-off of using a general purpose SQL core instead of an engine that already knows what data structures and queries it is trying to optimise for. But there is a lot of scope for improving performance simply by tuning the core, without affecting the application or the SPARQL queries that it is using.