JhuIngestDbAnalysis

Overview

Only Applied to DSpace pre-1.4

The conclusions of the following analysis have been applied in the 1.4.2 of DSpace. Big performance improvements are obtained using Postgresql Vacuum /Analyze after a big batch import.

Over the course of the AIHT project, we noticed that as a DSpace repository grew to very large sizes ingestion time increased dramatically, resulting in poor performance during the batch import of large numbers of objects. I was given the task to analyze DSpace's SQL usage, in particular to find locations where SQL queries were inappropriately slow. Because of the large, distributed scope of the DSpace project, reconstruction and direct analysis of SQL queries being made by the repository is a slow and inexact science. A single object ingestion alone invokes around three hundred queries. On the other hand, *because* DSpace makes such heavy use of the database during batch ingestion, profiling and amortized analysis is a useful strategy for learning which kinds of SQL queries need special attention.

Methodology

To profile DSpace during object ingestion, I leveraged the opensource projects p6spy and sql-profiler (both available: www.p6spy.com).

p6spy is a JDBC driver wrapper which transparently sits between the client application (DSpace) and the actual PostgreSQL JDBC driver. Without requiring any code modifications, p6spy allows for the logging, timing, and tracing of database queries dispatched by the client application, relying on the implementing driver to fulfill the actual query. Logs can be written to disk, or sent over a network in realtime via Log4J.

sql-profiler is a utility which receives p6spy logging information in real time over a network, allowing for interactive viewing of database activity. Also included is the ability to profile over query structure, independent of the actual parameter arguments used. All query statistics presented in this report are derived from sql-profiler.

In order for gathered statistics to be of any value, profiling had to be performed on a test repository which exhibited the same sorts of slowdown seen during the AIHT project. Therefore, I replicated and ingested a handful of simple objects a large number of times-on the order of forty five thousand-until profiling statistics became fairly pronounced. At this point, smallish batch runs of object ingestions with p6spy logging to sql-profiler became a highly useful window into DSpace ingest performance problems.

Observations

Initial statistics gathered clearly indicated that a large amount of PostgreSQL time was spent evaluating a small number of query types. In particular, over 80% of aggregate time was spent evaluating just 7 query types for single object ingests as performed by org.dspace.app.itemimport.ItemImport.

20.97 %	SELECT handle FROM Handle WHERE resource_type_id = 3 AND resource_id = 1
11.47 %	DELETE FROM ItemsByAuthor? where item_id = 42968
10.64 %	DELETE FROM ItemsByTitle? where item_id = 42968
.48%	DELETE FROM ItemsByDateAccessioned? where item_id = 42968
10.39 %	DELETE FROM Communities2Item? where item_id = 42968
9.97%	DELETE FROM ItemsByDate? where item_id = 42968
6.59%	

Analysis

While the high evaluation time for these queries certainly contributes to ingest performance issues, analysis of p6spy logs generated by a single object ingest reveal that these queries are often made repeatedly and unnecessarily.

The first query type, which performs an object handle lookup, is evaluated 3 times with identical parameters for the ingest of just a single object. Dispatch of this query is localized to one place within the DSpace source: the method org.dspace.handle.HandleManager.findHandle(HandleManager.java:298). When you consider that handles, once created, are immutable, this would seem a ripe setting for some sort of transparent caching.

The next five query types, all deletions, also happen a number of times per object ingest, even when there is no entry to be removed. All five queries are dispatched as part of the method org.dspace.browse.Browse.itemRemoved(Browse.java:368). The issue here is that itemRemoved is indiscriminately called by org.dspace.browse.Browse.itemChanged(Browse.java:394) as part of the browse index update process, regardless of what entries really need removal. Though not as easy an optimization to implement as the aforementioned caching, ingestion might benefit from more cleverness in tracking object changes and the database writes they require. This issue the DSpace developers are aware of, judging from code comments within itemChanged().

Implementing these two optimizations would positively benefit DSpace ingest performance, but as it turns out a dramatic improvement comes from the most obvious optimization, and further renders strategies such as caching and enhanced write tracking unnecessary. The DSpace database schema does not include appropriate indexing for the most expensive SQL calls as revealed by profiling. So, a clear improvement comes from adding the following indices:

For*SELECT handle FROM Handle WHERE resource_type_id = 3 AND resource_id = 1

```
CREATE INDEX handle_resource_id_and_type_idx ON handle( resource_id, resource_type_id );
```

For*DELETE FROM (TABLE) WHERE item_id = 42968

```
CREATE INDEX ItemsByAuthor_item_id_idx ON ItemsByAuthor( item_id );
CREATE INDEX ItemsByTitle_item_id_idx ON ItemsByTitle( item_id );
CREATE INDEX ItemsByDateAccessioned_item_id_idx ON ItemsByDateAccessioned( item_id );
CREATE INDEX Communities2Item_item_id_idx ON Communities2Item( item_id );
CREATE INDEX ItemsByDate_item_id_idx ON ItemsByDate( item_id );
```

For*SELECT community. FROM community, community2item WHERE community2item.community_id=community.community_id AND community2item.item_id = 42968 *

```
CREATE INDEX Collection2Item_item_id_idx ON Collection2Item( item_id );
CREATE INDEX Community2Collection_collection_id_idx ON Community2Collection( collection_id );
```

Benchmarking of Performance Improvement

Method:

A batch import was created by replicating a simple object 50 times, where each object consisted of a small dublin core and bitstream (~2K). A trial run then consisted of two successive imports via org.dspace.app.itemimport.ItemImport of all 50 items into a DSpace repository numbering around 45,000 items. Three independent trial runs were completed with statistics gathered using p6spy. A "vacuum analyze" was performed on the postgresql database before each set of trial runs. The test machine was a Macintosh Dual 2 GHz PowerPC G5 with 2GB DDR SDRAM.

Vanilla DSpace 1.2.1 trials:

Mean Aggregate Database Time - 153,328 ms
Std Dev - 212 ms

Most expensive SQL queries:
(numbers indicate % of aggregate time spent evaluating queries of this form)

14.58 %	SELECT handle FROM Handle WHERE resource_type_id = 3 AND resource_id = 1
14.24 %	DELETE FROM ItemsByAuthor WHERE item_id = 43713
12.93 %	DELETE FROM ItemsByDateAccessioned WHERE item_id = 43713
12.78 %	DELETE FROM ItemsByTitle WHERE item_id = 43713
12.70 %	DELETE FROM Communities2Item WHERE item_id = 43713
12.10 %	DELETE FROM ItemsByDate WHERE item_id = 43713
7.22%	SELECT community.* FROM community, community2item WHERE community2item.community_id = community.community_id AND community2item.item_id = 43713
3.54%	SELECT collection.* FROM collection, collection2item WHERE collection2item.collection_id = collection.collection_id AND collection2item.item_id = 43713
1.98%	SELECT 1

Optimized DSpace trials:

Mean Aggregate Database Time - 15,644 ms
Std Dev - 247 ms

Most expensive SQL queries:
(numbers indicate % of aggregate time spent evaluating queries of this form)

18.69 %	SELECT 1

10.12%	UPDATE resourcepolicy SET epersongroup_id = ?, resource_id = ?, eperson_id = ?, end_date = ?, action_id = ?, start_date = ?, resource_type_id = ? WHERE policy_id = ?
7.03%	UPDATE dcvalue SET text_value = ?, dc_type_id = ?, place = ?, text_lang = ?, source_id = ?, item_id = ? WHERE dc_value_id = ?
6.88%	INSERT INTO resourcepolicy (epersongroup_id, resource_id, eperson_id, end_date, action_id, start_date, policy_id, resource_type_id) VALUES (?,?,?,?,?,?,?,?,?)
6.02%	SELECT * FROM dctyperegistry WHERE element LIKE 'title' AND qualifier IS NULL
4.37%	INSERT INTO dcvalue (text_value, dc_type_id, place, text_lang, source_id, dc_value_id, item_id) VALUES (?,?,?,?,?,?,?)
3.17%	SELECT * FROM history WHERE checksum = 'e41f764054b69f69b26443bb4fe3d685'
3.05%	UPDATE bitstream SET deleted = ?, user_format_description = ?, checksum_algorithm = ?, checksum = ?, store_number = ?, bitstream_format_id = ?, description = ?, internal_id = ?, size = ?, source = ?, name = ?, sequence_id = ? WHERE bitstream_id = ?
2.46%	INSERT INTO history (creation_date,history_id,checksum) VALUES (?,?,?)
2.13%	

Current behavior of previously expensive queries:

1.58%	SELECT handle FROM Handle WHERE resource_type_id = 3 AND resource_id = 1
1.32%	DELETE FROM ItemsByDateAccessioned WHERE item_id = 44113
1.32%	DELETE FROM Communities2Item WHERE item_id = 44113
1.28%	DELETE FROM ItemsByAuthor WHERE item_id = 44113
1.26%	DELETE FROM ItemsByTitle WHERE item_id = 44113
<1%	DELETE FROM ItemsByDate WHERE item_id = 44113
2.12%	SELECT community.* FROM community, community2item WHERE community2item.community_id = community.community_id AND community2item.item_id = 44113

Conclusions

As benchmarking reveals, the improvement simply by adding these indices in a repository of 45 thousand items is about 9.8-fold. This speedup will clearly vary with the size of the repository, having little effect in small cases but impacting heavily on large-scale deployments. It is unlikely that further indexing would benefit DSpace ingest times; the most expensive call is now "SELECT 1", which is used extensively to validate connections and typically returns in under a millisecond. Extracting further speedup from DSpace ingestion thus would require the much more tedious task of refactoring the codebase to be more conservative with regard to query dispatch.