

Datastore caching

Datastore level caching

This document tries to outline a datastore level caching for features, that should allow to increase the performance of both map rendering, WFS serving and analysis operations.

At the moment this document is just a "brain dump" for the things that have been sitting on the back of my mind during the week, feel free to comment, expand and fix.

Requirements

1. Transparent caching by means of wrapping a base datastore
2. Spatial oriented caching, that is, have the spatial extent of features being the main driver in the keep/purge decisions
3. Serving up to date data by using datastore change listeners as appropriate
4. In memory caching, possibly with disk overflow
5. Size limited cache, possibly with global and per feature type limits
6. Eventually reuse potential in cases where the cached information is not a Feature, but has nevertheless a spatial dimension (think renderer optimized geometries and resolved styles caching, or any other kind of pre-processed spatial information).
7. The cache should have some kind of trashing control, reverting to pure streaming in cases where the hit ratio for a particular feature type becomes consistently so low that the cache becomes simply an overhead without any justification.

Preliminary design considerations

Read only vs write thru

The most important client of such a class would be read-only access, so this will be developed first.

Please note anyways that write thru caching is important to operations such as interactive editing, where both rendering and writing are performed in parallel, and all cases where data has a tendency to be quite volatile (in which write thru would allow to keep the cache intact instead of purging significant parts of it).

Access patterns

The usual way to access a datastore specifies a Query object in which a selection of properties and a filter are specified in order to limit the amount of data gathered.

The cache should leverage Query in a couple of ways:

- by allowing to drive the cache based on the query, that is, caching query results, not simply full features.
- by recognizing query "specialization", that is, that some queries are really a subset of other queries, thus possibly answering the queries out of the cache;

As already said, we want to drive the cache on the spatial dimension. This means every Query coming in will be interpreted as the conjunction of a spatial filter and a secondary, non spatial or eventually complex one. This also means every query that cannot be split like this will be interpreted like a conjunction of an empty spatial filter and the actual query (normalization).

The cache will try to keep into memory boxes of spatial data of various feature types and queries.

Granularity

All cache systems are designed as some kind of map from access keys to cached data. Trying to cache every single feature alone will result in significant overhead, so what we do want to cache is really the full set of features that fall inside a certain area, whose size and shape are subject of separate considerations.

What's important here is that whatever choice is made, a block of cached data is big enough to avoid high overhead (say, 10-100 features at least) and not too big to make the cache trash too easily (so it should probably be a matter of user configuration, or subject to an educated guess based on total memory allocated to caching and feature size).

Cache providers reuse

Open source and commercial cache providers are already available, with most of the features we need already set up, and with nice extras such as disk overflow, clustering, distributed transaction awareness.

So, it would be nice if the cache subsystem would be able to handle the Query and spatial dimensions, thus generating the cache keys and block contents, and let the available providers do the heavy lifting involved in cache maintenance.

Since different production environment demand for different kind of caches, a pluggable mechanism that allows to switch the cache provider would be nice to have. We could borrow the one used by Hibernate, for example (see http://www.hibernate.org/hib_docs/reference/en/html/performance.html#performance-cache and http://www.hibernate.org/hib_docs/v3/api/org/hibernate/cache/Cache.html).

Disk and clustering issues

Disk and clustering cache both requires some way to Serialize objects. At the time of writing, Feature is not Serializable. We need to change this, it's just a matter of extending the Serializable interface and move the responsibility of using Serializable attributes to the library user (all basic attributes are already serializable anyways, JTS geometries included).

Spatial cache design

The spatial cache level should take features coming in from the datastore, assign them into keyed "blocks" of features, and then give them to the chosen cache provider.

The cache could be based on a quadtree design (see <http://www.jump-project.org/docs/jts/1.7/api/com/vividsolutions/jts/index/quadtree/Quadtree.html>), but with the variant that each square should contain at least a given amount of features before being split into 4 sub-squares, in order to avoid excessive block granularity.

The center point of the quadtree could be the center of the feature type bounds, if known, or the center of the SRS validity area, or else assumed arbitrarily as (0,0).

The actual size and shape of the boxes is known only after the full answer to the Query has been gathered, so the cache system must be prepared to throw away blocks during construction already, and keep in memory only part of it.

If a query comes in that can be served by the cache, the part of the cache that can serve it must be locked, and a datastore query should be derived to serve parts of the query that cannot be matched in memory (case of query with a bbox larger than the areas kept in memory). This means that possibly multiple queries may be sent to the datastore, or a single query with multiple or-ed bboxes).

Some features will obviously overlap between boxes. Given the way the qtree approach works, a reference to them

will be held from each block overlapped (again, this is a reason to have lower block granularity in order to reduce duplications).

An alternative approach could be to split overlapping features, and to keep enough informations to put them back. In the general case this may prove to be overly complex, but datasets such as shorelines or isolines could benefit from splitting so much as making the difference between an effective cache and a useless one.

Reading concurrency is an issue since the cache blocks are populated during the read. So, what do we do if two reading sessions hit the same feature type and same or compatible query?

The two could go just happily over the blocks in the cache, but must synchronize over then the blocks that they may populate during the reading operation (and also consider the blocks could be thrown away because of lack of memory space). The two concurrent queries could be non overlapping, partially, or totally overlapping...