# Reladomo: Internal Cache Structure

March 2010

# Agenda

- Cache structure
- Cache configuration and general behavior
- Cache invalidation mechanisms
- Transactional behavior
- 3-tier caches
- Operation resolution
- Relationship resolution
- Object cache structure

# Cache Structure

- Protected by a high performance multi-reader single writer lock
- It's a map of Operation -> CachedQuery
- Uses soft references
- It also stores results of deep fetches: it's therefore not a good idea to clear the cache randomly
- Each class is assigned a query cache and an object cache
- Object cache always guarantees uniqueness based on primary key: the same PK is guaranteed to be the same piece of memory
- Object cache is a collection of indices
- Query cache remembers the results of queries that the application has run

# Cache Configuration and General Behavior

- Weak references are used with forEachWithCursor and newly inserted objects
- Empty at start time
- Populated with whatever queries/objects the application performs
- Can only answer queries that hit the query cache or the object cache exactly
- Ignores non-unique indices
- Uses soft and weak references
- Loads everything at the start, unless a "loadOperationProvider" is supplied
- Can answer all queries from the cache (unless in a transaction)
- With a loadOperationProvider, it can be in a fake-full-cache mode.
- Pretends the cache has everything in it.
- Can use both unique and non-unique indices
- Uses regular (hard) references: nothing will be GC'ed
- It's just a partial cache that does not answer any user queries of any kind.

# Cache Configuration and General Behavior (Continued)

- Used entirely for uniqueing and relationship lookup
- Partial cache
- Full cache
- None cache

# Cache Invalidation Mechanisms

- Clears the query cache
- Marks all the partial cache entries as "dirty". Dirty entries cannot be used to answer a query.
- Fairly granular: Insert/update/delete events are broadcast for any interested listeners.
- Programmatically initiated: Finder.clearQueryCache()
- Notification
- Time based expiration: if a query or object has been in the cache longer than the expiration time, it's not trusted
- On a per query level, the cache can be bypassed via findOneBypassCache or setting bypassCache on the list object
- E.g. ProductFinder.description().startsWith("s") => CachedQuery remembers Product class counter and Product.description attribute update counter
- Each CachedQuery object keep a list of per-class and per-attribute update counter at the time the query ran

# Cache Invalidation Mechanisms (Continued)

- When inserts/deletes happen, the class update counter is incremented
- Updates to particular attributes update the attribute update counter
- Cached query is only considered valid if its update counters are current.
- Objects collected via the GC (partial/none cache) : can only happen if no other references exist
- Common invalidations mechanisms:
- Query cache update counters
- Object cache

# Transactional Behavior

- The query cache is empty when the transaction starts
- This query cache is not shared with non-transactional queries or other transactions
- Result: queries prior to the transaction are not trusted. Queries within the transaction are trusted within the limits of update counter expiration
- Unless optimistic locking has been requested for an object. In that case, the cache is trusted, but update/delete statements have extra clauses to ensure the state hasn't changed since the application retrieved the object originally
- The database has to know that the object is in a transaction to provide correct ACID behavior
- No object is returned from the object cache without a read from the database
- It's best to do the reading inside the transaction, otherwise the object is refreshed upon access
- When a transaction updates an object, the committed version is kept separate

# Transactional Behavior (Continued)

- Non-transactional threads don't see the transactional (changed) state
- Two transactions can't write to the same object simultaneously
- When an object is inserted in a transaction, it's not added to the main cache for the class
- Instead, it's added to a per-transaction delta cache. Ditto for delete
- The delta cache takes precedence over the main cache for that transaction
- The transaction has a query cache for all classes
- Each object knows if it's participating in a transaction (shared or exclusive)
- Object cache keeps delta insert/delete indices

# 3-tier caches

- By default it's a partial cache and no configuration is required
- The default can be overriden in the runtime configuration
- The client tries to answer queries from its cache first before hitting the middle tier
- The server cache has to be configured
- The server can chose to answer the client's queries from its cache when appropriate
- The client starts the transaction and creates a proxy transaction on the server side
- The server is holding onto the actual transactional database connection
- The cache behaves as if the client was directly connected to the database
- The client has it's own local cache
- The server also has a cache
- 3-tier transactional behavior

# Operation Resolution

- Query cache only looks for exact matches. It will not returned expired CachedQueries
- A partial cache can only answer queries that map onto its unique indices and have a complete hit.
- A full cache will answer all queries, so long as no transaction is underway
- If no index is found, we give up in a partial cache scenario, or we get the entire contents of the cache in a full cache setup
- Operation has 3 methods: applyOperationToFullCache(), applyOperationToPartialCache(),
- One of the first two methods is called by the portal
- Operation then finds the most selective index to start with and does an index lookup.
- Example: Cache has 3 indices:
- Index 1 attributes: a
- Index 2 attributes: a,b

# Operation Resolution (Continued)

- Index 3 attributes: c
- Query is a = 1 & b = 2 & c = 3.
- If Index 2 is more selective than Index 3, we do index lookup for (a = 1, b = 2), then filter the results for c = 3
- General flow: hit the query cache, then the object cache, then the server
- Object cache query resolution
- It then filters the results based on the rest of the operation using applyOperation(List)
- Relationships used in operations are typically resolved through auto-generated indices
- All current index implementations are hash based: can only resolve "=" and "in"

# Relationship Resolution

```
private static final Extractor[] fororder = new
 Extractor[]
{
    OrderItemFinder.orderId()};

    …

    _portal = OrderFinder.getMithraObjectPortal();

    _result = (Order)
 _portal.getAsOneFromCache(_data, fororder );
```

- For queries that map to unique indices, the query cache is only used for negative (non-existent) hits
- A one-to-one or many-to-one relationships uses a fast path lookup on the cache directly
- A fast path lookup creates no garbage
- The code is essentially doing an index lookup

# Relationship Resolution (Continued)

- If the fast path fails to produce a result, we then create an operation and do a normal loopkup
- A one-to-many or many-to-many relationship creates a list and operation and resolves it normally
- During a deep fetch, the query cache is pre-populated with the operations and results that map the objects in the list to their related objects
- Therefore, a x-to-many relationship usually just hits the query cache

# Object Cache Structure

- Hashing Strategy
- An index is a searchable set (not a map!!!)
- A cache is not a map. It's a collection of indices
- Entry objects are Weak or Soft referenced. An entry can also be marked as dirty
- Weak references are used with forEachWithCursor and new inserts
- FullUniqueIndex: similar to a Trove THashSet, but is searchable
- PartialPrimaryKeyIndex: similar in structure to a HashMap (entry objects)
- PartialWeakUniqueIndex: used for partial cache indices other than the primary key
- NonUniqueIdentityIndex: only used with full caches. It's a compact searchable set that returns a list
- FullSemiUniqueDatedIndex: holds onto the data objects, not the (wrapper) business objects

# Object Cache Structure (Continued)

- PartialSemiUniqueDatedIndex: holds weak references to the data objects
- NonUniqueIndex: full cache only. Holds onto the data objects and returns a list
- DatedObjectIndex: holds onto the business objects using soft or weak references
- Core concepts:
- Non-Dated indices:
- Dated indices:

# FullUniqueIndex

- Hashing Strategy: usually created from a list of Reladomo attributes (ExtractorBasedHashingStrategy)
- Collision resolution is simpler than trove (quadratic probing)
- Unlike a JDK set (which has no get method)
- Search method by the same object class: getFromData
- Don't use the get() methods, as they are specialized for single attribute searches
- remove and contains work as you would expect
- Special feature: can search by a different class using the get(object, Extractor[]) method
- Generally the only class from the Reladomo cache package that's useful outside
- Used in multi-threaded loader for matching
- Can be used in application code for matching as well
- Structurally very similar to a Trove THashSet
- However, it's searchable

# SemiUniqueDatedIndex

- The business object is potentially instantiated if it didn't exist before
- It's an unusual index for the dated data
- It simultaneously holds two hash structures (one fully dated and unique, the other not
- An earlier implementation was using composition of two sets and it wasn't working well
- A dated cache first finds the data and then the business objects for that data
- The business object has the uniqueness guarantee, not the data object

# SemiUniqueDatedIndex Code

```
public class PartialSemiUniqueDatedIndex implements
 SemiUniqueDatedIndex
{
    private ExtractorBasedHashStrategy
 hashStrategy;
    private ExtractorBasedHashStrategy
 semiUniqueHashStrategy;
    private SemiUniqueEntry[] nonDatedTable;
    private SingleEntry[] table;
}

private static class SingleEntry extends
 WeakReference
    implements SemiUniqueEntry
{
    private int pkHash;
    private SingleEntry pkNext;
    private int semiUniqueHash;
```

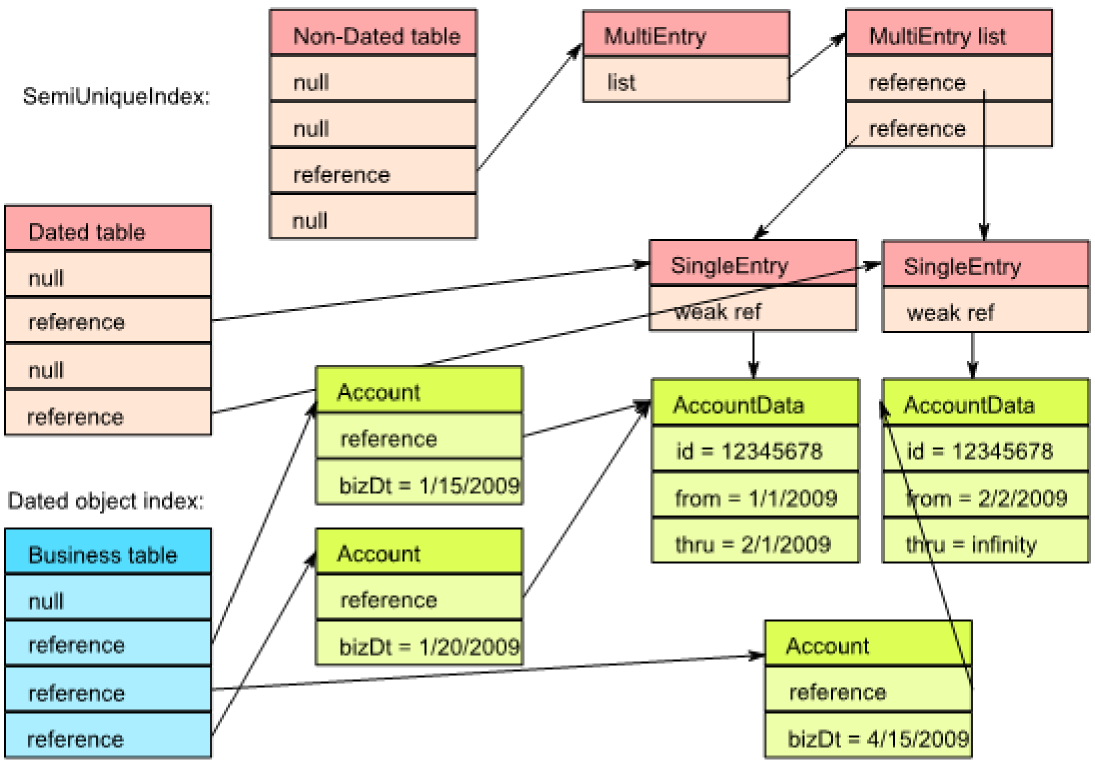
# SemiUniqueDatedIndex Code (Continued)

```
    private SemiUniqueEntry semiUniqueNext;
}

private interface SemiUniqueEntry extends
 SemiUniqueObject
{
    ...
}

private static class MultiEntry implements
 SemiUniqueEntry
{
    private int semiUniqueHash;
    private SingleEntry[] list;
    private int size;
    private SemiUniqueEntry semiUniqueNext;
}
```

# SemiUniqueDatedIndex Instance Diagram



SemiUniqueIndex:

Non-Dated table
null
null
reference
null

MultiEntry
list

MultiEntry list
reference
reference

Dated table
null
reference
null
reference

SingleEntry
weak ref

SingleEntry
weak ref

Account
reference
bizDt = 1/15/2009

AccountData
id = 12345678
from = 1/1/2009
thru = 2/1/2009

AccountData
id = 12345678
from = 2/2/2009
thru = infinity

Dated object index:

Business table
null
reference
reference
reference

Account
reference
bizDt = 1/20/2009

Account
reference
bizDt = 4/15/2009