

## Why Systems should architect for distributed caching upfront

Is it possible to design applications in such a way as to build-in the ability to take advantage of performance enhancements such as caching upfront?

In order to fully leverage the benefits of caching in the architecture, applications may have to include in their design data access methodology that is cache aware. Many systems may plan to eventually leverage an application cache to help them scale, reduce network load, or increase application performance at some point in the future when the load on the system requires this in order to meet desired service levels. Applications designed not to do this upfront may need to have code rewritten to maximize cache coherency and to fully realize the benefits they desire from using an application level cache. (And as most systems designers are aware, the cost of fixing a problem in a system rises exponentially as an application moves from design, to code, to test, and then to deployment).

The key issues to consider in order to maximize the performance benefits of distributed cache include query rewrites, anticipatory data fetches, and O-R mapping changes. Database query rewrites help applications leverage cache in several ways, including the need to write database access queries in two parts in order to:

**a) Maximize the amount of data that is extracted from cache**

Often two queries will be similar but not identical and the results sets they return have significant overlap. In that case, each result set would be stored as a separate object and would look like two separate queries. If each row of a result set where it's own query result set then a large percentage of the query results would come from cache. (See Figures 1 and 2).

**b) Enforce row level security on results that are extracted from cache**

In systems that depend on DBMS row level security features to limit which rows users with different security privileges see, two users who issue identical queries will see different results sets. In this case, the system needs a way divide a results set into pieces so that users only retrieve from cache data they are authorized to see. This is significantly easier if each row of a result set where it's own query result set.

**c) Allow retrieval of near real-time and reference data at the same time**

In systems where some of the data constantly changes (and some of the rows may move in and out of a result set based on these changing conditions), then two

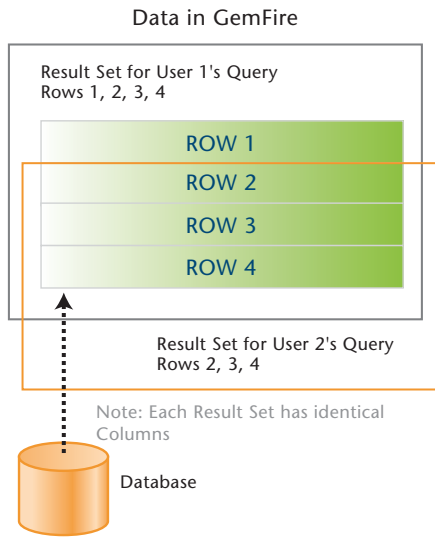


Figure 1: Overlapping Query Results Sets

identical queries may return slightly different results sets. If large amounts of data are in cache, and if each row of a result set where it's own query result, it is easier to retrieve the correct data from cache.

Query rewrites involve splitting the query into two parts. The first part is to fetch the primary key(s) from the database using the initial query's conditional clauses. This affectively returns a list of pointers to the rows that may be in cache (see Figure 3). The second part is to iterate over the list of

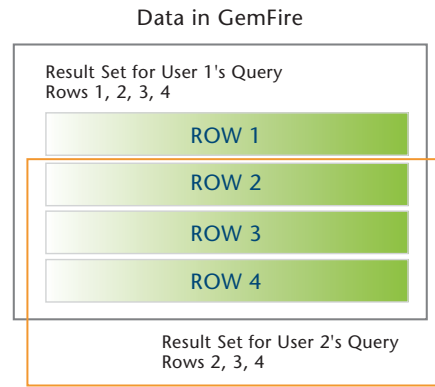


Figure 2: Overlapping Query Results with Each Row Stored as Separate Objects

keys, using the key in the conditional statement of the query so that only that row would be fetched from the database. If the row is in cache then it will be retrieved from the cache and not the database. This technique enables applications to retrieve data more from cache where the data sets from a query are not identical or where database row level security must be enforced. There are other variations on this technique that can be used if the data should be retrieved in chunks (see Figure 4).

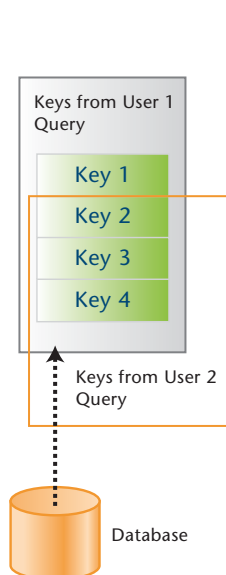


Figure 3: Step 1 of Query Rewrite

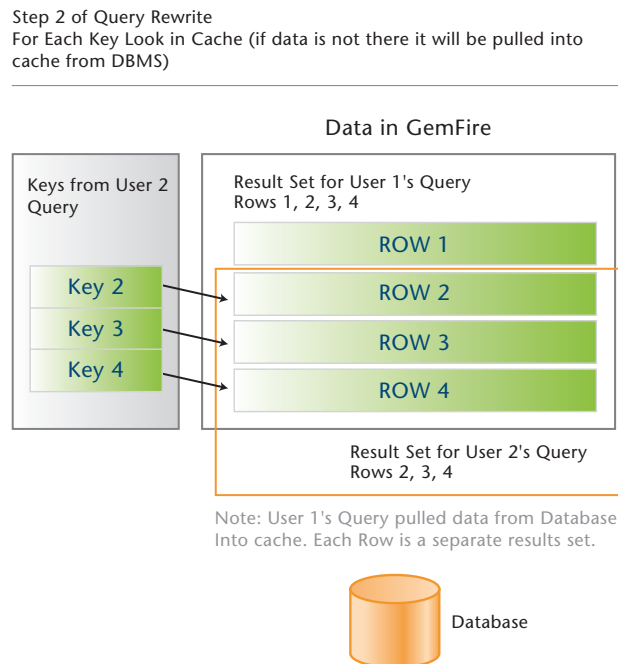


Figure 4: Step 2

Some applications may be able to predict which data a user may want based on the current data request. For example, if a user is fetching geographic based data, an application may predict that a user may want to look at data near the current location. Or, if a user is looking at data from a specific day, the application may predict that the user may want to look at the data before and after this day. All of this can be accomplished by having a fetch through the cache start threads to run additional queries that would cause the cache to be populated with the appropriate data, so that when the application later goes to look for the related data it will already be in cache. Catching this sort of work early in the application design can minimize design and code changes at a later point in time, simplifying the work and reducing cost.

Many applications use object-relational mapping (O-R mapping) to store Java objects in a relational database. Techniques for doing this either have negative impacts on performance since they tend to result in a large number of joins, or they tend to complicate the Java object models and related code by breaking relations among objects to avoid the overhead of many complex relational joins. When objects only need to persist for the life of a user session, or for some other non-permanent timeframe, then the user of distributed caching can result in several orders of magnitude increase in performance. Even if the objects do need to be persisted to a relational database, they may have a number of intermediate states that do not need to persist, that lend themselves to caching. Examining what needs long term persistence and what does not, is best accomplished early in the application lifecycle to avoid the costs associated with inspecting and changing large amounts of code.

## CONCLUSIONS:

The answer to the opening question on the possibility of designing applications early to take advantage of more robust caching is yes! By raising this question in the design phase or even early in the implementation process, the benefits can be significant. The only requirement is stopping to take time to consider the overall benefits and why this is important. The key gains are greater cache coherency which equates to better performance of the application especially in a stress environment and the potential increase in scalability of the application and overall architecture. The other potential advantages include the ability to leverage row level security, reductions in network load and increases in application robustness.

For more information on this subject please contact the author, Michael Lazar who can be reached at [michael.lazar@gemstone.com](mailto:michael.lazar@gemstone.com) or at 301-664-8494.

### Corporate Headquarters:

1260 NW Waterhouse Ave., Suite 200 Beaverton, OR 97006 | Phone: 503.533.3000 | Fax: 503.629.8556 | [info@gemstone.com](mailto:info@gemstone.com) | [www.gemstone.com](http://www.gemstone.com)

### Regional Sales Offices:

New York | 90 Park Avenue 17th Floor New York, NY 10016 | Phone: 212.786.7328

Washington D.C. | 3 Bethesda Metro Center Suite 778 Bethesda, MD 20814 | Phone: 301.664.8494

Santa Clara | 2880 Lakeside Drive Suite 331 Santa Clara, CA 95054 | Phone: 408.496.0242

