Qlik Application Performance Optimization Strategies (QAPOS)

QlikPerf.com

Document Author: Jeff R. Robbins

Email: Jeff.Robbins@qlik.com; jr@QlikPerf.com

Date: February 2nd, 2020

Contents

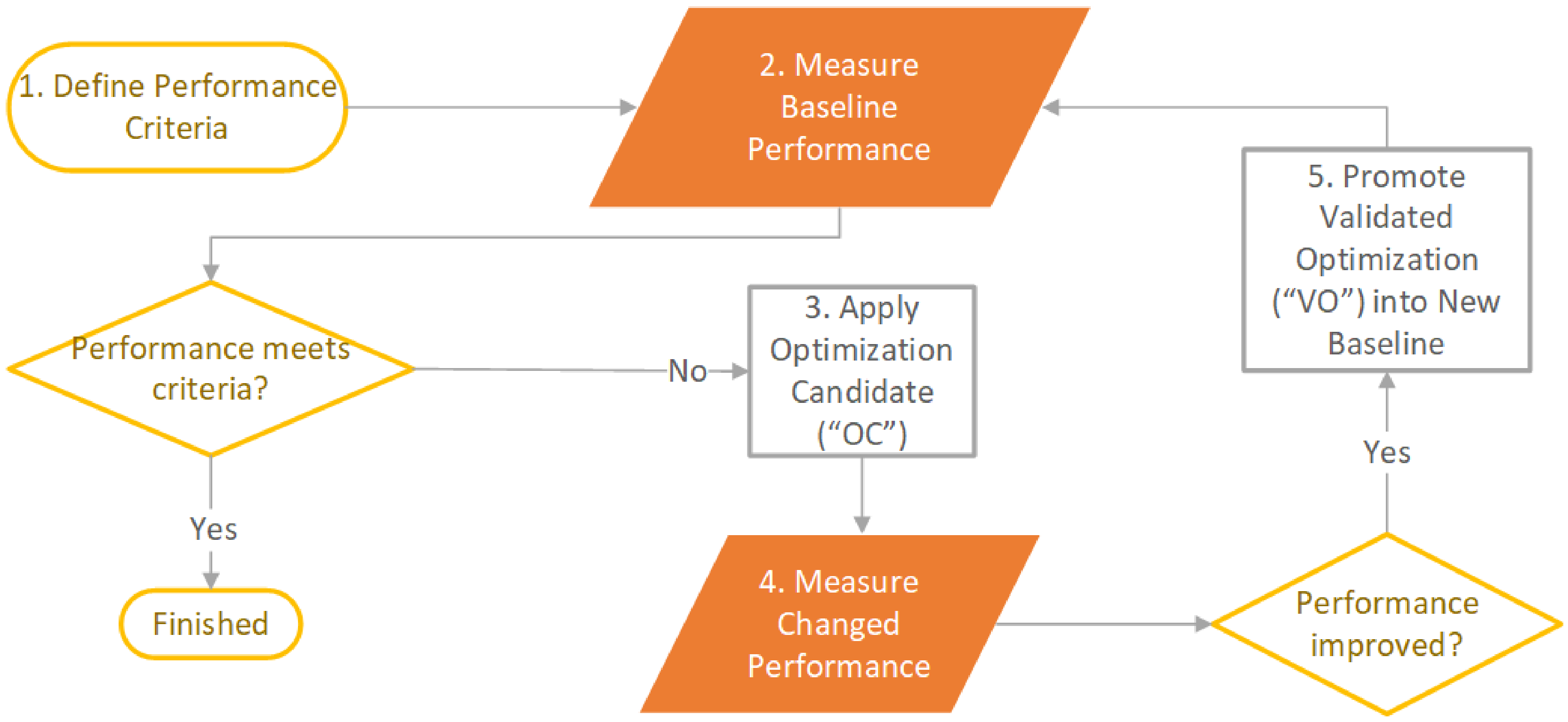
Introduction

Qlik Application Performance Optimization Strategies, abbreviated as **QAPOS**[1]**,** provide a framework by which to identify and implement specific **optimizations** in QlikView and Qlik Sense applications**.** Optimizations are application changes that provide empirically measurable performance improvement, in one or more of the following aspects:

1. Reduced response time
2. Decreased hardware resource consumption
3. Improved system reliability and stability

The Performance Lifecycle

Software application performance measurement and improvement is a cyclical process, as shown in this diagram:


1. Define Performance Criteria
2. Measure Baseline Performance
5. Promote Validated Optimization ("VO") into New Baseline
Performance meets criteria?
No
3. Apply Optimization Candidate ("OC")
Yes
Yes
Finished
4. Measure Changed Performance
Performance improved?

[1] The pronunciation of "QAPOS" is /kāpōs/, identical to that of "capos", plural for a guitar accessory: https://en.wikipedia.org/wiki/Capo

QAPOS facilitate a performance measurement and improvement lifecycle as follows:

1. Define performance criteria.

   Example criteria are: *An average response time of 8 seconds or lower is acceptable, where response time is the time between a user input (click) and the time at which all charts have completed calculating and rendering.*[2] *The 8 second average response times shall be provided with a population of 1000 concurrent users, with each user thinking an average of 1 minute after each system response.*[3]

2. Measure performance with the baseline application(s), work load, and hardware configuration.
3. Apply a proposed application change (**optimization candidate**) in a test environment. The strategies (**QAPOS**) described in this document specify areas in which to identify optimization candidates.
4. Measure performance after the implementation of the proposed change and compare baseline performance (step 2) with this changed performance, as well as with the criteria established in step 1.
   - Simple manual stopwatch testing can provide initial results to justify additional performance testing of the proposed change, in both QlikView and Qlik Sense.
   - In QlikView, object **CalcTimes** can also provide a useful unit performance testing mechanism, as described in Appendix 1. The **PMPT** is another useful QlikView unit performance testing tool, as described in Appendix 2.
   - Comprehensive multi-user system testing can be automated with the Scalability Tools for Qlik Sense and QlikView.
5. If the changed performance is better; the change is a **validated optimization**; promote the **optimization** to the new baseline.
6. Repeat steps 2-5 until acceptable performance (as defined in step 1) is obtained.

The following pages describe each of the 7 **QAPOS** in turn.

[2] There is a very effective argument that percentiles are often more meaningful than averages at this link
[3] The simulation of think times in performance testing is covered several places, including this link.

QAPOS 1: Refactor Objects & Expressions

Here, we discuss two categories of refactoring: **decreasing object calculation time** (which does not require a UI change) and **decreasing object calculation frequency** (which may require a UI change).

A. Decrease Object Calculation Time (with no UI changes)

Qlik objects (such as charts) and the expressions they contain are analogous to SQL aggregation queries. In the same way that there are typically multiple possible SQL queries (each with distinct performance characteristics) to achieve a desired output, there is often more than one way to construct a Qlik object and its constituent expressions to achieve a specific result. As such, a very powerful Qlik performance improvement strategy is object & expression refactoring, with steps as follows:

1. Review the results shown by the object under various selection states.
2. Examine the entire object (including constituent expressions and variables) to understand how it performs calculations against the data model.
3. Conceive of and implement alternate object versions that provide the same output as the original object. In some cases, it may be worthwhile to add a field to the data model to support the creation of a more efficient object. A common example of this concept is replacing a calculated dimension in a chart with a dimension based on a new data model field.
4. Test the performance of the alternate object versions. If the changed performance is better in one of the alternate versions, promote that version into the new baseline.

B. Decrease Object Calculation Frequency (may require a UI changes)

The Qlik calculation engine re-calculates all UI objects in cases where it detects that a user action (for example a selection in a field) is impactful to the data displayed in the UI object. As such, it is useful in many cases to disable calculation of an object at certain points in time. Consider the following steps performed by a user:

1. Click *Open Filters* button to display list boxes for dimension fields.
2. Select values in dimension field A.
3. Select values in dimension field B.
4. Click *Close Filters* button to hide list boxes for dimension fields.
5. View charts.

If all UI objects are re-calculating between each user interaction, there could be a noticeable delay between each click, providing a slow user experience consisting of a sequence of delays. As such, it is often advantageous to disable the calculation of resource-intensive charts until the user indicates (via clicking a *Close Filters* or similar button) that he would like to view the charts. The delay of chart calculation is implemented simply by setting the sheet objects' **calculation** or **show** conditions, such that that objects will not show or calculate if user has not yet clicked *Close Filters*.

QAPOS 2: Reduce Data Model Complexity

Data models with more than 25 tables should be evaluated for consolidation opportunities, as a larger number of tables increases run-time response times due to more frequent table traversals. In short, a data model with more tables is generally less efficient than one with fewer tables.

To improve performance for end users in data models containing more than 25 tables, reduce the number of tables by joining tables with fewer than four fields into the logically related table (unless a many-to-many relationship would result in a multiplication of records and/or invalid data). Further consolidation (and run-time performance improvements) of the data model can often be achieved by creating a common unified fact table, using Qlik techniques such as JOIN or CONCATENATE.

In short, the **general recommendation** is to aim for a data model with characteristics closer to a star schema than a snowflake schema, as the lower number of tables in the star schema results in fewer time-consuming table traversals.

QAPOS 3: Pre-aggregate (Consolidate) Fact Rows

A reduction in fact table record count, with a corresponding decrease in response times, can potentially be accomplished by consolidating fact rows. Consider the following example data set:

Transaction ID	Customer ID	Region	<Metric Field>
1	A	Z	3
2	A	Z	4
3	B	Z	7

This data set could be pre-aggregated at the Region level as follows:

Region	<Metric Field>
Z	14

The basic process here is to store the metric fields in the Qlik fact table at the highest granularity at which the end user needs to view the metrics. The pre-aggregation of the metric fields can be done quite readily in the Qlik data load script with the Group by construct.

The time required for the aggregation of the transaction level data up to the higher level (the region level in this example) is consumed during the Qlik data refresh process (to which the end user is not directly exposed) to reduce application run-time response times (as experienced directly by the end users). While generally it is advantageous to reduce the response times experienced by end users, the trade-off in the time consumption between data refresh and application run-time should be evaluated with the performance requirements and data refresh cycles in mind.

QAPOS 4: Streamline Table Links

Replace Textual Keys with Numeric keys: Numerical keys in the Qlik data model consume less space, and allow for potentially faster run-time performance, than textual keys.

As with the prior strategy (fact row consolidation), one should evaluate the trade-off between run-time response times and data refresh duration; converting textual keys to numeric keys (typically with the Qlik AutoNumber function) will likely provide faster run-time response to end users while potentially increasing cyclic data refresh times.

QAPOS 5: Remove Redundant and Unused Data

Fields in the data model that are not used in any charts, UI objects or variables consume system resources but provide no benefit. Identifying and removing such unused fields can provide performance gains.

Also, flaws in the ETL process may result in duplicated records in the data model. For example, if you know that you have 10 million customers, but the **Customers** table in the Qlik data model has 20 million records, it is likely that you are experiencing record duplication somewhere in the ETL process (possibly in the ETL process prior to Qlik, and possibly within the Qlik load script itself). Ideally, one should analyze the ETL process to understand at which point the duplicate records are introduced, and then repair the issue at the source. If enough time is not available such analysis, then the `distinct` pre-fix of the Qlik **`LOAD`** script command is an expedient way to eliminate duplicate records, at any point within the application build process.

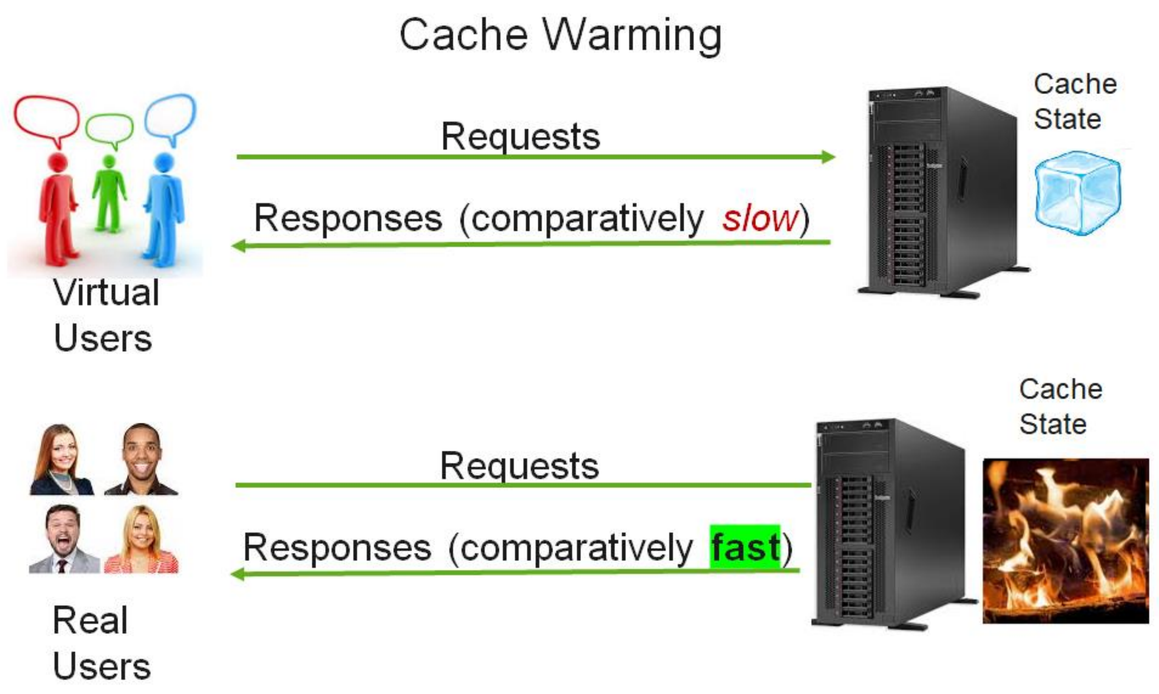
QAPOS 6: Warm the Cache

Cache warming entails the use of an automated mechanism to simulate a user accessing a Qlik application, between the publish time of the app and the time of first access by a real live human user. After Qlik app publication and then cache warming, when a real user subsequently accesses the same Qlik app, calculation results will be pulled from Qlik Server cache, rather than calculated "from scratch". As such, response times provided to the real user will be lower than had the cache not been warmed.

The user simulation mechanism to warm the cache can be any one of the following:

1. Any automated mechanism that can drive a web browser through a set of actions.
2. A Qlik Sense Scalability Tools scenario built with the QSST.
   a. Some materials use the term "pre-caching"; here are links to those materials for Qlik Sense.

One could reasonably argue that cache warming is an operational procedure, rather than a true application optimization, since the application itself need not be changed to implement cache warming. However, we could consider cache warming a mechanism by which we optimize the *state* of the application; we provide better performance to end users by proactively changing the cached state of the deployed application from cold to warm.

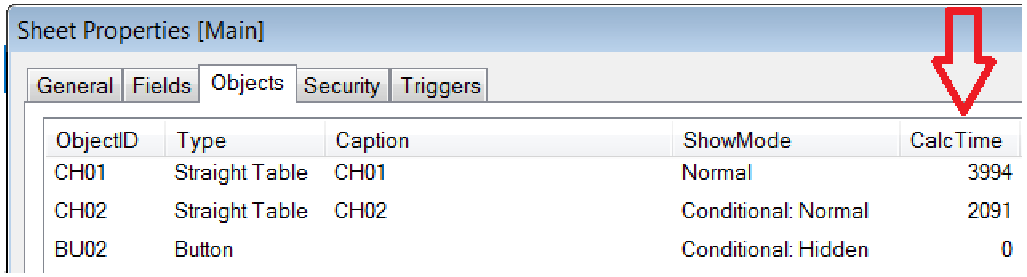


QAPOS 7: Segment the Application

A single logical application can be split across multiple physical QVW or QVF files, with each file loaded "on-demand" in a seamless fashion. This physical division can provide lower response times to end users, as the segmented QVW or QVF active at any point it time is smaller than a single monolithic QVW or QVF, with a smaller set of records upon which chart expressions must aggregate.

1. In software development, ***decomposition*** entails dividing a large piece of software into smaller components, for better performance and easier maintenance. As one common example, Microsoft Outlook consists of over 80 separate files on disk, rather than a single monolithic file. Even though it is composed of many separate files, Outlook is one single application.
2. Qlik application development is quite simply one specific type of software development; we can therefore apply the technique of decomposition to our Qlik projects to enhance the user experience by providing lower response times.
3. A single logical Qlik application can be split ("segmented") across multiple physical files (QVFs or QVWs), with each file loaded when needed in a seamless fashion. This decomposition can improve performance, as the segmented QV files are smaller than a single monolithic QV file, with a smaller set of records upon which chart expressions must aggregate.
    1. Application segmentation can be implemented in 2 different ways:
        a. *a priori,* where application segments are built during a regularly scheduled data refresh process
        b. *a posteriori*, where application segments are built at application run-time based on end user requests
        c. Now, the *a priori* approach is traditionally referred to as "document chaining", and the *a posteriori* approach is often called "ODAG" (on-demand app generation). But note that both approaches are actually chaining (or linking) multiple QVWs into a cohesive navigation path. The only difference between the *a priori* and *a posteriori* approaches is whether the creation of the application segment QVWs is done proactively based on a specification or reactively based on users' run-time requests.
        d. The *a priori* approach ("document chaining") is typically most effective when there are a limited number of ways in which users might slice the data, and we can therefore set up a schedule to create a manageable number of application segments that meet the users' analysis requirements.
        e. The *a posteriori* approach ("ODAG") is a good option when where there are a very high number of ways in which the users might slice the data, and implementing every potential slice with its own pre-built segment would result in an excessively large number of segments, many of which might not be used on a regular basis. With ODAG, the number of segments created is limited to those that users request at dashboard run-time.

Appendix 1: Unit Performance Testing with CalcTime (QlikView Only)



Sheet Properties [Main]

ObjectID	Type	Caption	ShowMode	CalcTime
CH01	Straight Table	CH01	Normal	3994
CH02	Straight Table	CH02	Conditional: Normal	2091
BU02	Button		Conditional: Hidden	0

During QlikView dashboard development, we can use *CalcTimes* to easily test the performance impact of changes. Object-level *CalcTimes* can be viewed in the properties for each sheet. *CalcTime* is the wall clock time in milliseconds between the user action (click) and the time which the object displays the calculation results.

Within sheet properties, the object list can be sorted by *CalcTime*. The highest *CalcTime* represents the response time for the most recently completed action (selection or sheet transition); response time is the time between the user click and the completion of calculation of all objects on the sheet. Due to concurrency, the objects' *CalcTimes* are not additive; rather, the highest *CalcTime* tells us the end-to-end response time. For example, the screen shot immediately above indicates that that the response time for the most recently completed action was 3994 milliseconds, the time at which the last object, CH01, completed calculation.

Assuming an equivalently-sized hardware platform for both QV Desktop and QV Server, then the response time experienced by end users through QlikView Server (for non-cached calculations) will typically be higher than the best-case numbers reflected in the *CalcTimes* in QlikView Desktop, due to network latency and contention for resources across multiple users. Regardless, *CalcTimes* do provide a useful mechanism for comparative unit performance testing of alternate implementation options.

Note that QlikView Server and Qlik Sense Server both provide cross-session caching; calculation results for a particular app and selection state for a given user can be stored and provided very quickly to other users at a later point in time.[4] The benefits obtained by cross-session caching can be simulated in QlikView Desktop by *not* clearing the cache immediately before making a selection for the second time. To simulate an un-cached state and the resulting higher *CalcTimes*, a *ClearCache* button can be used during development. This *ClearCache* button should invoke a macro, defined as follows:

```
sub ClearCache
     ActiveDocument.ClearCache
     msgbox "Cache has been cleared."
end sub
```

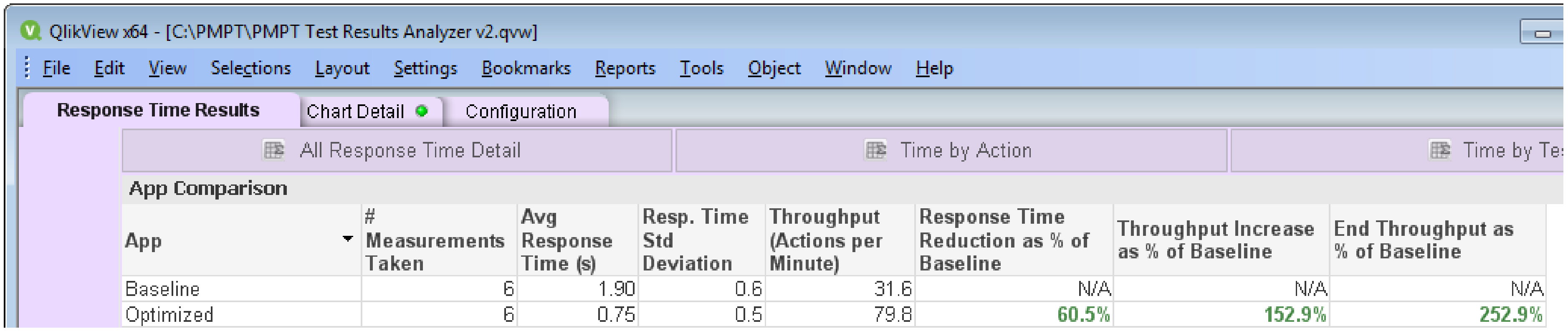
[4] Caching provides the most benefit when there is a common data set being analyzed by multiple users; if section access constrains each user to his own distinct data set, then cross-session caching is not applicable.

Appendix 2: Unit Performance Testing with the PMPT (QlikView Only)

Multi-user system performance tests gather performance data objectively; these tests can be run with the Qlik Sense or QlikView Scalability Tools. However, developers may wish to work on performance optimization prior to the availability of a system test environment.

As such, we provide the supplemental mechanism of the "**Poor Man's Performance Tester**" (PMPT), which can be run and analyzed entirely within QlikView Desktop, with a hardware platform as small as a single developer's laptop. The PMPT is a *unit performance testing* utility, run by the developer prior to system testing.

The PMPT, which can be obtained from Qlik Consulting, provides a higher level of automation than the CalcTime testing described in the prior section, with less setup overheard than the Scalability tools for Qlik Sense or QlikView.


QlikView x64 - [C:\PMPT\PMPT Test Results Analyzer v2.qvw]
File Edit View Selections Layout Settings Bookmarks Reports Tools Object Window Help
Response Time Results
Chart Detail
Configuration
All Response Time Detail
Time by Action
Time by Te

App Comparison

App	# Measurements Taken	Avg Response Time (s)	Resp. Time Std Deviation	Throughput (Actions per Minute)	Response Time Reduction as % of Baseline	Throughput Increase as % of Baseline	End Throughput as % of Baseline
Baseline	6	1.90	0.6	31.6	N/A	N/A	N/A
Optimized	6	0.75	0.5	79.8	60.5%	152.9%	252.9%