### TDV Caching Use Case Study

#### Caching Best Practices

<table>
<thead>
<tr>
<th>Project Name</th>
<th>TIBCO Data Virtualization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Release</td>
<td>V1.5</td>
</tr>
<tr>
<td>Date</td>
<td>January 12, 2018</td>
</tr>
<tr>
<td>Primary Author</td>
<td>Matthew Lee</td>
</tr>
<tr>
<td>Document Owner</td>
<td>TIBCO Software</td>
</tr>
<tr>
<td>Purpose</td>
<td>TDV Caching Use Case Study Caching Best Practices</td>
</tr>
</tbody>
</table>

**TIBCO Software** empowers executives, developers, and business users with Fast Data solutions that make the right data available in real time for faster answers, better decisions, and smarter action. Over the past 15 years, thousands of businesses across the globe have relied on TIBCO technology to integrate their applications and ecosystems, analyze their data, and create real-time solutions. Learn how TIBCO turns data—big or small—into differentiation at www.tibco.com.
### Revision History

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Author</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1.0</td>
<td>February 11, 2015</td>
<td>Matthew Lee</td>
<td>Initial Draft</td>
</tr>
<tr>
<td>V1.1</td>
<td>July 2, 2015</td>
<td>Dale Taylor</td>
<td>Second draft with expanded use cases and discussions</td>
</tr>
<tr>
<td>V1.2</td>
<td>July 10, 2015</td>
<td>Matthew Lee</td>
<td>Third draft with initial internal revisions</td>
</tr>
<tr>
<td>V1.3</td>
<td>April 6, 2016</td>
<td>Matthew Lee</td>
<td>Initial Release</td>
</tr>
<tr>
<td>V1.5</td>
<td>January 12, 2018</td>
<td>Venky Madireddi</td>
<td>Applied TIBCO branding</td>
</tr>
</tbody>
</table>

### Distribution

This document has been distributed to:

<table>
<thead>
<tr>
<th>Name</th>
<th>Title</th>
<th>Company</th>
<th>Date of Issue</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Related Documents

This document requires the following approvals. Signed approval forms are filed in the project files.

<table>
<thead>
<tr>
<th>Document</th>
<th>Date</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDV Administration Guide</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDV User Guide</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Copyright Notice

COPYRIGHT© 2018 TIBCO Software Inc. This document is unpublished and the foregoing notice is affixed to protect TIBCO Software Inc. in the event of inadvertent publication. All rights reserved. No part of this document may be reproduced in any form, including photocopying or transmission electronically to any computer, without prior written consent of TIBCO Software Inc. The information contained in this document is confidential and proprietary to TIBCO Software Inc. and may not be used or disclosed except as expressly authorized in writing by TIBCO Software Inc. Copyright protection includes material generated from our software programs displayed on the screen, such as icons, screen displays, and the like.

Trademarks

All brand and product names are trademarks or registered trademarks of their respective holders and are hereby acknowledged. Technologies described herein are either covered by existing patents or patent applications are in progress.

Confidentiality

The information in this document is subject to change without notice. This document contains information that is confidential and proprietary to TIBCO Software Inc. and its affiliates and may not be copied, published, or disclosed to others, or used for any purposes other than review, without written authorization of an officer of TIBCO Software Inc. Submission of this document does not represent a commitment to implement any portion of this specification in the products of the submitters.

Content Warranty

The information in this document is subject to change without notice. THIS DOCUMENT IS PROVIDED "AS IS" AND TIBCO MAKES NO WARRANTY, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO ALL WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. TIBCO Software Inc. shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

Export

This document and related technical data, are subject to U.S. export control laws, including without limitation the U.S. Export Administration Act and its associated regulations, and may be subject to export or import regulations of other countries. You agree not to export or re-export this document in any form in violation of the applicable export or import laws of the United States or any foreign jurisdiction.

For more information, please contact:

TIBCO Software Inc.
3303 Hillview Avenue
Palo Alto, CA 94304
USA
# Table of Contents

## 1 INTRODUCTION

1.1 Purpose ................................................................. 5  
1.2 Audience ............................................................... 5  

## 2 TDV Caching Functionality

2.1 Overview and key features ........................................ 6  
2.2 How TIBCO Professional Services Regards Caching .......................... 7  

## 3 Common Uses of TDV Caching

3.1 Caching Derived Data ............................................... 8  
3.2 Caching the Results of Expensive Queries .......................... 9  
3.3 Caching Slowly Changing Data ..................................... 9  
3.4 TDV (and caching) as source for ETL processes ................. 10  

## 4 Caching to Mitigate Data Source Challenges

4.1 Caching Large Data Sets to Offload Data Sources .................. 11  
4.2 Supplement EDW Staged Data with Live Data ..................... 11  
4.3 Conceal / Address Data Source Latency ........................... 12  
4.4 Conceal Outage Windows / Insulate Production Systems from Load 12  
4.5 Temporary Staging For Geographically Constrained Data .......... 12  
4.6 Add Push Down Support .............................................. 13  

## 5 Some Cautionary Examples

5.1 Caching to Avoid Query Optimization ................................ 14  
5.2 ETL Workflow Replacement .......................................... 15  
5.3 Cache as a Failover Source .......................................... 15  
5.4 Materialized EDW Replacement ...................................... 16  
5.5 Cached Data Set Cannot be Restored by a Full Refresh .......... 16  
5.6 Constrained Cache Refresh Windows ................................ 17  
5.7 Large and/or Frequently Updated Data Sets ........................ 17  

## 6 Conclusion .............................................................. 18  

**Table of Figures**

No table of figures entries found.

**Table of Tables**

No table of figures entries found.
1 INTRODUCTION

1.1 Purpose
Caching is one of the most commonly used features of TIBCO Data Virtualization (TDV) platform. Unfortunately, many customers find themselves unsure of the best way to leverage the caching functionality of TDV and often seek guidance on when and how to best use caching. This document presents and discusses a number of uses for the TDV caching mechanism. Our goals are to assist the reader with understanding the various uses of caching and to give the reader some guidance in assessing the viability of their own potential uses of TDV caching.

On the latter goal, we should note that we have specifically avoided labeling any of the examples in this paper as “good” or “bad” use cases. Instead we try in each case to assess its fit with both the design intent of TDV caching, and with the broader design intent of TDV itself. We also discuss the potential challenges in developing or managing such a solution. But even a use case that stretches the intended use of the platform, or which presents significant implementation or management challenges, may in fact be the most acceptable available approach. By building the necessary management infrastructure (e.g., intelligent cache management policies) and being clear on the risks inherent in a given design and the steps required to mitigate those risks (e.g., reducing timing dependencies on interrelated cache refreshes, or enlarging the maintenance window to accommodate more expensive refresh operations), even a less than ideal design can be successful.

Note that none of the use case examples we discuss in this document are theoretical. They are all derived from first-hand experience of our consultants, working alongside TIBCO TDV clients, solving real world problems. Although we have been careful to mask details that would identify specific clients, the patterns we discuss here are based on reality.

1.2 Audience
This document is intended to provide guidance for the following users:

- TDV Developers
- TDV Architects and Designers
2 TDV Caching Functionality

2.1 Overview and key features

Caching enables TDV to store the result set returned after executing a view or procedure so it can be reused to satisfy similar client requests. The TDV developer can specify when, where, and how often result sets are cached.

Clients are sometimes confused by our use of the term “cache”, assuming that the mechanism works like a memory/disk caching capability. The feature in fact functions similarly to materialized views in database design.

Result set caching may be applied to any view or procedure created in TDV Studio. By configuring a cache for a resource, the TDV developer instructs the server where to cache the result set and how often the cache is to be updated. The cached result set may be stored in a local file system, a local PostgreSQL database or a supported external database specified by the developer. If the resource being cached is a procedure (including scripts, java components, and web services), multiple result sets will be cached, each corresponding to a unique set of input parameters.

During query planning, TDV’s query engine (QE) recognizes that a particular resource has been cached, and it modifies the query plan to retrieve data from the cached result set instead of the underlying source data. This can eliminate potentially costly requests to the underlying data source.

It is worth reiterating that any view or data service resource at any level may be cached, and that resource may still be used to build other higher-level resources. This granular approach to result set caching allows the TDV developer to truly take advantage of data encapsulation, and balance the volatility of the data with the currency needs of the consumer.

The caching policy for a resource is established at design time and allows the data designer to dictate storage location, storage mode, update strategy and update timing as follows:

- Storage Location: A result set cache may be stored in a file or a database. Files work very well for small and relatively static result sets in development phase. For production use, databases serve as primary storage for larger and more volatile result sets.

If a TDV instance is participating in a cluster, using a database for the cache storage allows a single result set cache to be shared by all nodes in the cluster.

Please refer to the TIBCO Data Virtualization Administration Guide for a complete list of supported cache data sources.

- Storage Mode: When defining caching for TDV views, developers have the option of selecting multi-table cache storage in addition to single table cache storage. When using multi-table caching TDV uses several physical tables to store multiple cache snapshots for a single cached resource. Having one table to store each snapshot improves performance by allowing data to be deleted very quickly and enabling more efficient use of the source resource’s indexes at query execution time. Multi-table caching works very well for caches that contain substantial amounts of data to be retained for a long time, whereas the single table mode is simpler to manage. Multi-table caching also removes conflicts between reading cached data while new data is being loaded into a new snapshot, allowing TDV to automatically manage indexes on the cache.

- Update Strategy: A cache can be updated by performing a complete refresh or by incrementally modifying the cache with new changes. The complete refresh strategy is the most straightforward approach, and it can be automatically applied to any cached resource. An incremental refresh strategy enables the TDV developer to
provide business logic by which to perform required updates. A resource can be cached individually or assigned to a cache policy where the cache refresh or other settings are managed at the policy level. Cache policies allow resources to be organized into groups so that cache schedules and other settings can apply to the group, rather than to individual resources.

- Update Method: TDV can be configured to leverage native load functions in the cache target to load and refresh the cache. This native cache method improves cache load performance by leveraging bulk load mechanisms and/or bypassing TDV and loading data directly into the cache tables from the source tables. In addition to native cache, TDV uses multiple threads to load the cache in parallel.

- Update Timing: A cache may be updated periodically based on the clock, or it may be updated opportunistically when the cached data exceeds a certain age. It is also possible to refresh a cache manually only, which is very useful for caching reference tables that seldom change.

In addition to the multiple result set caching options described above, TDV also supports Incremental Caching, Procedural Caching and Table Caching.

For more information on incremental caching, please refer to the TIBCO Data Virtualization User Guide.

### 2.2 How TIBCO Professional Services Regards Caching

- Customers are sometimes surprised by how conservative TIBCO Professional Services consultants sometimes can be when choosing to implement a cache. Usage of caching in various TDV implementations is evaluated on a case-by-case basis. It may be useful to explain how we see caching and its capabilities, in the overall context of the TDV platform capabilities.

- A Cache is a snapshot or a subset of source data or derivative of source data, similar to regular TDV views being subsets or derivatives of source data.

- Caching is not persistent in the same way that source data is persistent. Operational data store or data warehouse holds its information until someone explicitly decides to delete it. TDV caches are transient in nature and are intended to be refreshed periodically, to keep in sync with the source data.

- A key value proposition of data virtualization is providing efficient real-time access to data, where it lives and as it is. Caching could be leveraged to augment TDV capabilities but it is generally most effective when used judiciously – staging result sets from complex and expensive queries that are frequently run; solving availability problems for some data sources; providing temporary relief for overloaded source systems.

- Caches must be managed. TDV has excellent out of the box cache management functionality (e.g., definable cache refresh policies), so in most cases the administrative overhead of cache is low. But it is another “moving part” to be aware of, and therefore introduces some measure of complexity into the design and management of the TDV solution.

A well-designed and carefully targeted cache can make a significant difference in system performance or reliability. We have seen many examples in which caching dramatically reduced runtimes of critical queries and made the difference in producing a successful TDV solution. We have seen other examples where caching was the key TDV capability that overcame persistent performance or access problems in source data systems.
3 Common Uses of TDV Caching

3.1 Caching Derived Data

Most database platforms rely heavily on indexes to improve query performance. When a join requires derived, or manufactured data, indexes are generally not available which limits a developer’s options for improving query performance.

As an example, suppose we have the following tables and relationships:

```
T_CUSTOMER T_PARTY_GROUP
------------- -------------
CUSTOMER_ID (PK) PARTY_GROUP_ID (PK)
CUSTOMER_NAME PARTY_GROUP_NAME
CUSTOMER_CLASS ...
CUSTOMER_ADDR_ID ...
```

And suppose a relationship exists between the two tables such that we can join as so:

```
T_CUSTOMER INNER JOIN T_PARTY_GROUP ON 'A000' || T_CUSTOMER.CUSTOMER_CLASS = PARTY_GROUP.PARTY_GROUP_ID
```

Performing a join in this way will invalidate any index on the CUSTOMER table’s CUSTOMER_ID column when joining the two tables, and performance will suffer.

This performance challenge can be addressed in a number of ways:

If the database platform supports functional indexes and TDV developers are able to modify the source tables, a functional index matching the structure of the derived data can be added to improve performance without the use of a cache.

This approach may not be possible in all cases. Not all database platforms support functional indexes and many organizations do not allow TDV developers to make modifications to tables in production databases.

Caching can be used to address this challenge when use of a functional index is not possible.

As a second alternative, a wrapper view can be created that adds the derived column used for the join to the source table. This view can then be cached and the derived column can be indexed to improve performance of the join.

Caching the wrapper view requires materialization of at least one source table, which introduces data currency concerns and may require significant cache storage if the source table is large. Also use of a cache will also require a federated join if the cache is not stored on the same database as the source tables.

A third option involves creating an additional view that bridges the two tables in TDV and the caching and indexing the view. This view can then be used to join the original source tables and the cache together performantly.

Once we define this view, we would cache using multi-table caching so that TDV will automatically drop and recreate physical indexes when the data is loaded. Additionally, the TDV query engine will take advantage of the indexes on the manufactured cached view to improve performance of the join.
This approach limits the size of the data set in the cache to only the records needed to perform the join, which reduces the amount of cache storage required. As in the previous approach, using a cache introduces some data concurrency concerns and may add a federated join.

For the above example tables, we would define our bridge view as:

```
CUSTOMER_PARTY_GROUP_BRIDGE
-----
SELECT
  CUSTOMER_ID,
  'A000' || T_CUSTOMER.CUSTOMER_CLASS AS PARTY_GROUP_ID
FROM
  T_CUSTOMER
```

We would also define 2 virtual indexes: CUSTOMER_ID, and PARTY_GROUP_ID.

This is an interesting use case that applies caching in a non-obvious way, to solve a common performance challenge. It uses native TDV caching features, does not require additional user code outside of the definition of the cache object, supporting views and virtual indexes, and doesn’t impose unusual cache maintenance requirements.

### 3.2 Caching the Results of Expensive Queries

“Expensive” queries are generally one or both of two things: derived from or creating large data sets (e.g., several million row result set from a several billion row table); and expensive query constructions (e.g., complex joins, aggregations or transformations).

Some users of TDV may assume that they should use caching whenever they are dealing with large result sets. The reasoning is that queries that return a large number of rows will be slow, and thus they should be “cached”. This is sometimes the case, but most design situations argue against this usage profile. More on that later in this paper.

The case for caching is more easily made when a frequently requested result set is the product of expensive joins, aggregations and transformations. It is a TDV best practice to cache a high level view that materializes and persists the result set from such operations. Since these result sets are typically smaller in volume – often much smaller – than the underlying data set that from which they are derived, there are attendant improvements in utilization of disk space and network capacity.

This usage pattern can be particularly beneficial when the cache object serves to offload a heavily loaded data source. It reduces the load on the source, and by preparing an oft-used and expensively derived result set, improves the efficiency of the overall system.

This is an example of what we would call a “sweet spot” use case for caching. It makes use of both the core competency of the caching facility (materializing and persisting result sets) and of TDV (data abstraction). If the cached object maps to frequently executed user queries, the amount of system resources saved by avoiding expensive and predictable database operations can be huge.

### 3.3 Caching Slowly Changing Data

As with any of the examples described in this document, the decision to use caching is also a decision to trade off data freshness/currency for availability or efficiency. A cached object is a snapshot in time. Live data will by definition be more current than the cache. So, caching is most effective when it is derived from data which either changes less frequently, or whose currency is relatively unimportant to the user. For example, a common application of TDV is as a
data services layer for reporting and business intelligence clients. Many of our clients, particularly those in Financial Services, employ TDV to facilitate monthly and quarterly reporting. These systems typically implement a large update of data on a monthly basis. Once that update is completed, the data tends to be static until the next month’s update. Furthermore, the queries that are run tend to be similar – aggregating result sets in similar ways (e.g., results by region or by client grouping) or requesting similar subsets (e.g., results for the Northeast region).

These conditions make caching – particularly caching the results of expensive queries, as previously described – a very useful feature for improving the efficiency and responsiveness of the overall system.

### 3.4 TDV (and caching) as source for ETL processes

Many people view data virtualization as a viable replacement for ETL technology, but TDV platforms are generally not very good at full scale ETL. TDV, ETL and persistent-store DBMS technologies (RDBMS, Big Data, etc.) are complementary foundational platforms, which are best used together to build solid enterprise information management architectures. In fact, we often participate in solution development that includes TDV working alongside other data technologies, including live (e.g., transactional) data systems and enterprise data warehouses. One example of a complementary pattern is the use of TDV as a data source for ETL processes. ETL platforms are optimized for processing of large scale and complex data movement and transformation, while TDV is optimized for presenting disparate data sources as a homogeneous, single virtual “database”. To the extent that we can simplify the ETL process’ access to the various data sources in play within a client’s data environment, TDV actually adds value to the ETL process.

One interesting case to this concept also illustrates the ability of TDV and caching to solve problems that are not entirely consistent with its design intent, but work well nonetheless. The client wished to apply TDV to facilitate ETL processes toward building their enterprise data warehouse. A small number of datasets (less than 5) were not available from traditional relational sources and were exposed as SOAP or REST/JSON services. For those sources, we leveraged TDV to convert the data to relational form, and exposed the relational form for consumption by the ETL batch process. Caching was used to persist the relational form due to lengthy execution times to convert from JSON to relational.

This use case applied caching as part of a directed process flow, which is at first glance inconsistent with its intended use as a means of persisting result sets for user queries. But it actually is a successful use of caching. The lengthy execution times for data extraction through JSON services meant that the extraction process took up a lot of wall clock time, making it impractical to couple the extraction directly to the “TL” part of the ETL processes. Further, TDV uses native platform capabilities to cast these sources into relational format, which offloaded that expensive task from the ETL platform. The combination of cache persistence and simple, SQL based relational casting allowed TDV to add significant value to an ETL process.
4 Caching to Mitigate Data Source Challenges

There are a number of use cases that apply caching, as part of the TDV technology stack, to add value to large or challenging data sources.

4.1 Caching Large Data Sets to Offload Data Sources

A familiar use case for caching is to offload data sources that are overloaded, unreliable or not consistently available. While caching can be used to solve this type of challenge, this usage does present its own challenges and caveats. For one, these cached views are often defined without the filters, joins, and aggregations one would typically find in higher-level views. They are therefore likely to return large data sets, which may require larger allocation of greater data capacity for the cache storage platform. It is still the best practice to create the cached object with as much refinement and filtering as you can, though we understand this will not always be possible. For example, there is no reason to cache all of a 36-month fact table if you know that your user queries will only address the current year’s data. Also, if this data will be joined with other data for most or many queries, you might consider performing those joins as part of the cache view definition.

In other instances, clients want to cache large data sets simply because they’re large. We tend to recommend against moving a data set to cache simply because the data set is large if (i.e., the data source is not overloaded and there are no other performance/capacity issues such as network congestion). If the use of a cache results only in moving processing load from one available database (the source) to another (TDV-managed cache database), all that has been accomplished is to transfer workload. Actually, this usage adds to the overall workload, since the cache object must be created and then periodically refreshed.

If the source data from which the cache is derived is frequently updated and must be kept in sync in the cache, the bar is set even higher in justifying a cache solution due to the management overhead incurred in maintaining the cache. TDV caches can be maintained via either full refreshes (essentially dropping the cache table’s result set and recreating it in its entirety) or incremental refreshes (a more surgical approach that captures only changes in the underlying data but requires a greater implementation effort). Developers should always consider that the processing cost of a full refresh increases with the volume of source data. For particularly large tables, it may be difficult to fit the refresh into your available maintenance window. Incremental updates can take less (perhaps much less) time, but require the data to be structured in such a way as to enable TDV to determine what has changed. If the underlying tables are update/append only (i.e., no row deletions) and contain ordered primary keys it may be possible to take advantage of incremental caching to reduce the amount of data that has to be pulled in order to refresh the cache. If deletes occur on the table, extra work is necessary to identify deleted rows.

It is important to keep in mind that while a view that returns a large result set may in fact take a long time to execute; when you configure a view to be cached, you’re simply storing a copy of that large data set somewhere else. The cache storage location may not be able to return this large data set any faster than the original data source. Thus it’s usually more efficient to apply caching in such a manner that you choose views that have a lot of processing (joins, filters, aggregations) that take time and also reduce the number of rows which will result in the maximum performance gain by minimizing cache refresh time as well as minimizing view execution time.

4.2 Supplement EDW Staged Data with Live Data

Through the use of TDV’s federation engine, clients are able to conceal ETL latency periods by federating data loaded from an EDW and the live production systems. However, many use cases such as BI reporting and analytics may generate a higher volume of requests than the production system can absorb without impacting critical business
processes. Caching is frequently used to temporarily stage a snapshot of production data until records can be transferred to the EDW. Requests for data from the production system are resolved against the snapshot of data staged in the cache, which protects the production system from excess load while still allowing users to access a more recent data than available in the EDW.

Depending on how quickly production data changes and how current the snapshot of production data must remain, it may be necessary to configure the cache to refresh several times during the day or to configure the cache as an incremental cache.

### 4.3 Conceal / Address Data Source Latency

Caching can be used to address data source fetch latency in some cases by staging a snapshot of data on a different platform. Issues such as network latency, poor query response time due to excess server load and poorly structured data, which may impact performance of TDV generated queries can potentially be addressed with a cache.

Caching does not guarantee improved performance in all cases. When developing a cache to improve query performance it is frequently necessary to investigate options such as modifying the structure of the cache to improve the generated query, adjusting TDV’ capabilities or settings to enable push down, adding indexes at the database level or performing database-tuning activities such as refreshing statistics on the cache table.

Be aware that cache platform limitations such as network latency, platform capability limitations that may prevent TDV from pushing operations to the data source and hardware limitations are factors that cannot be addressed within TDV.

### 4.4 Conceal Outage Windows / Insulate Production Systems from Load

TDV caching can be used to fulfill requests for production data using a staged copy of the data when the source system is unavailable due to a scheduled period of unavailability such as a maintenance window or scheduled backup. This is accomplished by enabling a preconfigured cache on the views impacted by the outage and triggering a cache refresh operation prior to the outage period, which allows TDV to transparently redirect requests to the staged copy of the data for the duration of the outage. Once the outage period passes the TDV caches can be disabled to allow TDV to service requests from the original source system.

Note that this use case only covers predictable source system outage periods when sufficient time can be allocated to execute a cache refresh operation.

### 4.5 Temporary Staging For Geographically Constrained Data

Data privacy and retention laws of some countries may prohibit an organization from replicating or staging data generated within the country outside of its geographic borders. For example, Swiss banking privacy laws limit what financial data can be accessed or copied from outside of the country.

A company’s data consistency policies may prohibit persisting data outside of a central data store, which may be geographically distant from users that need to make use of the data.

Accessing data that is geographically distant from the client application typically will incur significant fetch latencies due to longer data transmission times. In many cases this latency can be a significant performance impediment that can mean the difference between a successful or failed implementation.

TDV caching can be used to temporarily stage frequently accessed data closer to the requesting users while still complying with laws or other requirements that prevent data from being permanently persisted outside of a particular location. The careful use of TDV cache expiration policies can ensure that cached data does not persist for longer than
a prescribed length of time while also insulating users against fetch latencies when accessing frequently used data that is stored in distant locations.

### 4.6 Add Push Down Support
Caching can be used to stage a snapshot of data onto a platform that allows TDV to push down operations such as joins and filters that could not be pushed to the original source system due to platform limitations. Common use cases involve caching data sourced from data source that do not enable push down at all, such as flat files or web services or platforms that cannot have certain operations pushed to them either due to a lack of support on the source system or incompatible case sensitivity or trailing space settings on the TDV server and source system.
5 Some Cautionary Examples

In this section we will discuss scenarios where the use of caching introduces risks or complexities that may argue against its application. It is important to emphasize that, as we said at the top of this document, these are not necessarily “bad” uses of the mechanism. It may well be that TDV cache is the best available solution you have for these problem sets. But even then, it is important to understand the challenges these uses bring with them, so that you can properly plan and allow for them.

5.1 Caching to Avoid Query Optimization

We have occasionally worked with clients who develop a profound fondness for TDV caching – to the point where they cache a large percentage of the data that is modeled through the data virtualization layer. But caching is more suited as a scalpel than a sledgehammer. Judiciously applied, it can be a great boon to the efficiency and reliability of the data environment.

But overuse of the mechanism can have the opposite effect. In TDV, a cached object is the result of a periodically executed query or procedure which stages a copy or derivation of source data. Its very nature requires extra steps to be periodically taken to maintain it (the cache refresh). And because we’re making a copy of the data, we now have to concern ourselves with data freshness (how current is the information) and synch (is it current with other critical data with which it will be joined). So a basic truth of caching is that it makes the solution more complex to manage. The decision to use caching should in part be about weighing the cost and risk of that complexity against the operational advantage gained by the caching. For a few well-placed caches that dramatically improve runtime performance, it’s an easy win. But if you find yourself caching a third or more of your virtual database objects and having to write custom procedures to manage the synchronization and updating of those objects, the cost may well be greater than the benefit.

This situation typically arises due to one or more of several situations:

- The project’s designers and developers are repurposed ETL development resources. TDV, with its focus on real time access of data “where it lives”, is unfamiliar to them. Caching, with its ETL-like movement and copying of data, is more comfortable for them.

- Tight project delivery timelines lead the team to look for quick short-term solutions to performance or throughput issues. Lacking either the time or motivation to optimize critical real time queries, the team instead casts the result sets for these non-performing queries into cache.

- One or more critical data sources does not meet performance or availability needs of the project. This may be due to either technical challenges (e.g., underpowered dbms environments) or process/policy hurdles (e.g., DBA’s limiting access to key sources).

One client, facing all three of these factors, ultimately produced a “TDV solution” that actually cached almost a third of the over 900 views in the virtual data model. Furthering the challenge, there were numerous interdependencies among the cached and non-cached objects, which led to a requirement for precise order and timing of cache refreshes, which in turn led the client to construct fairly elaborate procedures (using the Admin API) to directly coordinate the cache refresh operations.

This was, to put it mildly, a lot of moving parts. The sheer weight of the design led to painful management and use issues. Management of the system was complicated due to the need to directly and precisely control the order and timing of caches via scripts. This proved to be impossible due to the sheer volume of caches to manage, so the caches...
(which, as we noted, had a lot of interdependencies) were often not in synch with each other. A single failure along the string of operations could lead to cascading failures down the line. All this led to one of the worst things that can happen with a database environment -- incorrect results -- due to joined data that was current to different points in time. Aged caches joined against refreshed caches, interacting with current data on the source systems.

There is a happy ending to this story. The client invited TIBCO’s professional services team in to evaluate and assist with optimizing the system. Working with the client project team, we helped to reduce the number of cached objects from 300 to 7. We also helped them simplify their virtual data model, reducing the number of views from 900 to 700. The revised system was much more stable and reliable, requiring none of the complex and fragile cache management scripting that was such a management headache in the first solution release. The revised system also made much more appropriate use of the TDV platform. Whereas the original solution design cast TDV in an uncomfortably ETL-like role, the revised design allowed TDV to do what it does best — access the data in real time, where it lives, efficiently and reliably. The small number of remaining caches solved specific, cache-friendly issues by staging the results of expensive and broadly used queries and had their intended effect — dramatically reducing overall system cost while adding an acceptable level of systems management overhead.

The optimization exercise was a huge success, producing a more reliable, manageable and efficient system. It has become a showcase for the use of TDV technology at this large enterprise. The key activities leading to this success were: removing caches that were created out of mere expedience, to avoid the necessary work of query optimization; preserving caches that presented real opportunities to avoid repeated heavy queries whose result sets were widely consumed by users; and taking the time to train and mentor the design and development staff on data virtualization technology and its benefits.

5.2 ETL Workflow Replacement

Customers will sometimes look to TDV as a means of replacing their ETL processes. In this scenario caches are used to retain the calculated results of the workflow.

Caches can be used effectively to temporarily persist relatively small volumes of data generated during execution of a complex workflow for later consumption by an external system or later process. However, the transitory nature of caching means that any calculated results that need to be permanently persisted should be stored externally.

Use cases that involve a large number of expensive operations involving very large volumes of data such as aggregations, complex data derivation operations and data cleanup operations generally cannot be completed quickly enough to deliver results in real time.

Several of our customers have successfully used TDV to supplement ETL platforms precomputing data conversion operations that are more efficiently handled in the TDV platform and caching the results for later use by the ETL platform.

5.3 Cache as a Failover Source

In this case, the design uses caching as a fall back data source if attempts to access the system of record / production system fail. Cache is used to keep a relatively recent version of the data staged as a backup source of data in case the production system goes down or is otherwise unavailable.

This use case is problematic for a number of design, implementation and administrative reasons. The approach is very tricky to implement and even trickier to debug. It requires the use of a custom procedure or row-based security to intelligently route requests either to the live source system or, in the case of a primary system failure, to the cache data. The requirement for additional control / query routing functionality may significantly increase complexity of the
implementation. It can also compromise the ability of the TDV optimizer to push down operations to any data sources, thereby creating potentially large additional processing loads on the TDV instance.

This usage also sets the stage for likely data consistency issues since the user cannot be certain, at the time their query is run, if they are retrieving current data or a relatively old snapshot.

5.4 Materialized EDW Replacement

Use cases to replace an enterprise data warehouse (EDW) for data persistence and its ETL load process require design strategies that are inconsistent with the cache mechanism’s basic operation.

Cache refresh operations always succeed or fail as a single unit of work. Unlike an ETL process, TDV cannot resume a failed cache load operation at the point of failure, a failed refresh must be completely restarted from the beginning each time. This behavior cannot be bypassed without significant custom developed assets, which increase the complexity of a cache implementation and the level of effort required to support the implementation.

A basic design assumption of TDV caching is that a cache object does not have guaranteed persistence – it may be invalidated (made unavailable for use) and refreshed at any time. A cache failure may invalidate the entire dataset stored in a cache, which requires a full cache refresh to resolve. This is an especially significant concern for incremental cache implementation where a full cache refresh may require a significant amount of time to complete and may place too much load on the system. While this risk can be mitigated somewhat through carefully written error handling in custom incremental caching code, not all cache failures can be reliably captured and mitigated.

Cache functionality assumes that the cached dataset can always be restored by a full cache refresh. If this is not possible either because the source system data does not persist for the required retention period or because a full refresh operation would not be possible due to source system access window constraints then caching is likely not an appropriate option.

So, the nature of the caching mechanism poses significant management and maintenance challenges as a persistent store such as a materialized EDW.

5.5 Cached Data Set Cannot be Restored by a Full Refresh

TDV caching is designed for use as a temporary data-staging platform. It should never be used as a data persistence store due to the way cache refresh operation executes.

The cache implementation assumes that the cached dataset can always be restored from the source systems by executing a full cache refresh operation. If a cached result set becomes invalidated, for example due to a cache expiring or an incremental cache refresh failure, TDV will mark the cached result set for deletion and execute a full cache refresh operation the next time a cache refresh is triggered.

Use cases that require that records in a result set be retained for a longer period than they can be restored using a full cache refresh either because the source system does not retain the dataset for the full retention period or the source system’s availability window precludes a full refresh operation should not be implemented as a cache. Long-term external storage of critical data is more appropriate for an EDW or other (intentionally persistent) store.

On the other hand, caching can be used very effectively when a TDV view federates between an EDW and other source systems. It is also possible to use caching as part of an ETL process to transfer data from other source systems to an EDW.
5.6 Constrained Cache Refresh Windows

Use cases with very tightly constrained access windows to source systems needed by TDV caching can carry significant risk with respect to supportability and performance.

Performance may become a concern if one or more cache refresh operations depend upon the constrained data source(s). If the cache refresh operations cannot be efficiently executed in parallel against the source system a potentially significant risk that all of the caches will not refresh within the availability windows may arise.

Restrictive source system availability windows may also present a risk in the event of a cache failure. If a cache failure that necessitates a full cache refresh to recover from occurs during a period that the source system is not available, all triggered cache refresh operations will fail and requests that attempt to access cached data will fail with errors. This may represent a significant risk if the impacted cache(s) have many downstream dependencies or the source system will not be accessible for an extended period.

5.7 Large and/or Frequently Updated Data Sets

As already mentioned, caching can be quite beneficial in storing result sets from complex, expensive queries. It is most useful when data currency is not an issue, either because the data changes slowly or user requirements do not necessitate the freshest data. If data currency is an issue, then we have to consider the tradeoffs in deciding whether to use caching, and in how we manage the caches that we do employ.

For example, consider a view that is configured for caching and contains complex transformations. Also assume that cache refresh cycle takes 3 hours to process 10 million source data rows into a several thousand row cache object. In this example, the business requirement is for users to see data that is current to the last fiscal week. This means that we would need to perform a cache refresh once a week. The resultant cache object reduces average run time on some critical and frequently executed queries from several minutes to several seconds. Given these assumptions, the use of cache may be an excellent design choice.

But let’s now assume the source data volume grows from 10 million to 100 million, driving the cache refresh time to 8 hours. With a weekly refresh requirement, this may still be an acceptable processing window.

But if changing business requirements now dictate a daily refresh, things get more complicated. With an 8 hour processing window for full refreshes of the cache, we are probably cutting it too close to guarantee the availability of the cache object during normal working hours. We might want to consider using incremental caching in order to reduce the amount of effort required for refreshes. But the feasibility of incremental caching will depend on how the data is updated. Changes are insert only and the table has a primary key? Good. Changes may include some updates? Gets more complicated; let’s hope the table has a timestamp column or some other way to identify changed rows. Changes may also include deletes? This is going to make it pretty difficult to make an incremental caching strategy work consistently.

If the incremental refresh fails for whatever reason, TDV will likely need to perform a full refresh to ensure the cache is brought to an acceptable state. This will very likely overrun the processing window and cause an extended cache outage while the full refresh operation is executed. If the live data is accessible, it may be preferable to disable caches during business hours to allow TDV simply query the source to allow users to retrieve their data. While the queries may run slower, they will still run.
6 Conclusion

In this document, we have reviewed several uses of TIBCO Data Virtualization’s caching functionality and we have explored various factors that should be considered when deciding whether to implement a cache.

All this is not meant to dissuade developers from using caching outright. Rather, we want to ensure that you consider the tradeoffs in risk, design complexity and maintenance effort on one hand, and runtime efficiency and performance on the other.