Adaptive Real-time Virtualization of Legacy ETL Pipelines in Cloud Data Warehouses

Ehab Abdelhamid[1], Nikos Tsikoudis[1], Michael Duller[1],
Marc Sugiyama[2], Nicholas E. Marino[3], Florian M. Waas[1]
[1]Datometry Inc. [2]Independent Researcher [3]ReadySet
ehab@datometry.com,niko@datometry.com,mduller@datometry.com
sugiyama@acm.org,nick@readyset.io,mike@datometry.com

ABSTRACT

Extract, Transform, and Load (ETL) pipelines are widely used to ingest data into Enterprise Data Warehouse (EDW) systems. These pipelines can be very complex and often tightly coupled to a given EDW, making it challenging to upgrade from a legacy EDW to a Cloud Data Warehouse (CDW). This paper presents a novel solution for a transparent and fully-automated porting of legacy ETL pipelines to CDW environments.

1 INTRODUCTION

An Enterprise Data Warehouse (EDW) ecosystem typically consists of two distinct solution stacks. On the one hand is the Extract, Transform, and Load environment (ETL), responsible for ingesting data from various sources and integrating it into the EDW. On the other hand are the Business Intelligence (BI) and reporting tools, which issue complex analytical queries against the EDW. Each solution stack typically consists of a deeply integrated collection of native client drivers and utilities, third-party tools and applications, and custom-built applications containing many SQL queries. Figure 1 depicts such a typical EDW environment.

Data warehouse replatforming is when an enterprise replaces its legacy backend data warehouse with a new modern one. This task has sparked interest with the emergence of Cloud Data Warehouses (CDW) such as Amazon Redshift [14], Google BigQuery [20], Microsoft Synapse Data Warehouse [23], and Snowflake [11]. Legacy EDW systems are seen by many as expensive and inflexible to handle the ever-growing volumes of data in today's world. CDWs free enterprises from procuring and managing their infrastructure and data warehouse systems, reducing upfront costs and staffing needs. However, replatforming is a lengthy, costly, and risky undertaking due to the high complexity of the EDW environment, as many of the components are vendor-specific and not easily portable to a new data warehouse. Mismatched data types and formats, incompatible semantics of transformation logic, discrepancy in data load/export APIs, and impedance mismatch of bulk loading utilities are only a few challenges to mention in this context.

Datometry introduced the concept of Adaptive Data Virtualization (ADV) to support replatforming of legacy EDW environments to novel cloud data warehousing solutions [6, 7, 26]. In a nutshell, ADV virtualizes access to the database by intercepting the application's communication with the database, translating, and redirecting it to the new CDW. Datometry's Hyper-Q is based on a powerful algebraic framework for query translation [7], that maps incoming SQL queries to a system-agnostic abstraction and applies the necessary transformations to make the query executable on the new system. Additionally, Hyper-Q supports the emulation of unsupported features like stored procedures, macros, and recursive queries [26]. It allows customer tools and applications to work out of the box with minimal changes, reducing the cost and duration of the replatforming project significantly.

Our earlier work focused on reporting queries and presented a solution for replatforming legacy BI environments. This paper focuses on how we extended Hyper-Q to handle ETL pipelines. Ingestion pipelines are typically written using a proprietary scripting language, making them very difficult and expensive to rewrite for CDWs. Some of the unique challenges when replatforming ETL pipelines are as follows:

- There is a large discrepancy between the loaders available in the different CDW environments. These are often vendor-specific and non-portable. Examples include Amazon's AWS S3 cp [5] and Microsoft's AzCopy [21], each of which has its own configuration options and formats.
- Third-party tools often reuse and build on existing data load/export APIs the EDW provides. In this paradigm, tightly coupled software components work hand-in-hand to implement various fine-tuned business use cases.
- ETL tools and applications support the injection of customized SQL into the data pipelines. As a result, there is a large volume of legacy SQL sprinkled around almost all of the business processes.
- Some legacy EDW allow per-tuple error handling and reporting. A tuple that has a data format issue or causes a uniqueness constraint violation gets excluded from the load job and is instead recorded in an error table for later review. This tuple-at-a-time processing clashes with the set-oriented nature of query processing in modern CDW.
- Replatforming the ETL pipelines has to go hand in hand with replatforming the BI environment. Any decision about how data is modeled must be consistent across both sides of the ecosystem since they operate on the same data.

This paper presents a solution for virtualizing legacy ETL pipelines based on the idea of Adaptive Data Virtualization and presents the following novel contributions:

- We describe a novel solution for real-time virtualization of legacy ETL constructs. We show how they can be mapped transparently to the primitives and APIs of CDWs.
- We present techniques for optimizing the client to server data transfer and the transformation part of ETL pipelines.

This work was conducted while M. Sugiyama and N. Marino were with Datometry.

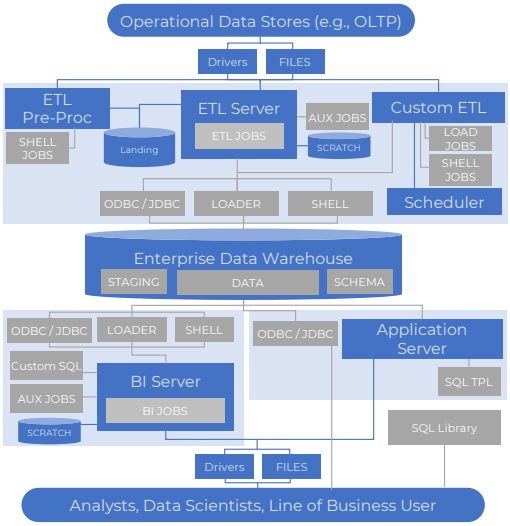


Figure 1: EDW environment: ETL (top) and BI (bottom)

- Our solution includes an integrated component for error handling, which achieves configurable granularity of error reporting without sacrificing performance.
- We present an experimental study using real-world jobs, illustrating the impact of our solution in practical settings.
- We report on a customer case study and the lessons we learned during this experience.

2 BACKGROUND

In this section, we give background on legacy ETL pipelines. ETL jobs in a legacy EDW environment rely on proprietary and/or third-party client tools that interact with the data warehouse server to import/export large data volumes in a scalable manner. Client tools are typically used to define and execute ETL workflows coded using proprietary scripting languages. At a high level, an ETL job script specifies input/output data files, target database tables where data needs to be imported to or exported from, and the transformations that need to be applied to the data before being imported/exported.

Example 2.1. The following ETL script loads data from input file *input.txt* into target table *PROD.CUSTOMER*. The job defines a simple transformation that uses an SQL *insert* command to load data from input fields and apply trimming and formatting logic before data is imported.

```
.logon host/user,pass;
.layout CustLayout;
.field CUST_ID  varchar(5);
.field CUST_NAME  varchar(50);
.field JOIN_DATE  varchar(10);

.begin import  tables PROD.CUSTOMER
  errortables PROD.CUSTOMER_ET PROD.CUSTOMER_UV;

.dml label InsApply;
insert into PROD.CUSTOMER values (
  trim(:CUST_ID), trim(:CUST_NAME),
  cast(:JOIN_DATE as DATE format `YYYY-MM-DD') );

.import infile input.txt
    format vartext `|' layout CustLayout
    apply InsApply;
.end load
```

□

Example 2.1 shows an ETL script to load data from an input file into a target table after applying data transformations. It is written using a proprietary scripting language that is interpreted using a legacy ETL client. During the script execution, the following sequence of operations is performed:

(1) Client requests the EDW server to create error tables to store any data/constraint violations in these tables.
(2) Client starts a number of data-loading sessions connected to the EDW server.
(3) Client extracts data from the input data file, splits it into chunks, and sends it to the EDW server through the parallel data loading sessions. Over the wire, data is formatted according to the format and protocol of the EDW system.
(4) Server loads data into a staging (work) table.
(5) Server executes the job transformation (*insert* statement) to transform and load data from the staging table into the target table.
(6) Client logs off data loading sessions and queries the server to extract any errors recorded in the job error tables.

The legacy ETL import job consists of two main phases:

Data Acquisition. The data is pumped in at high speed from client to server. The server caches the raw data it receives and waits to be instructed by the client on what to do next.

DML Application. The client sends the transformation logic to the server, expressed in SQL, and then it gets applied to the loaded data. Data and constraint violations are recorded in appropriate table structures that can be queried to get the job status.

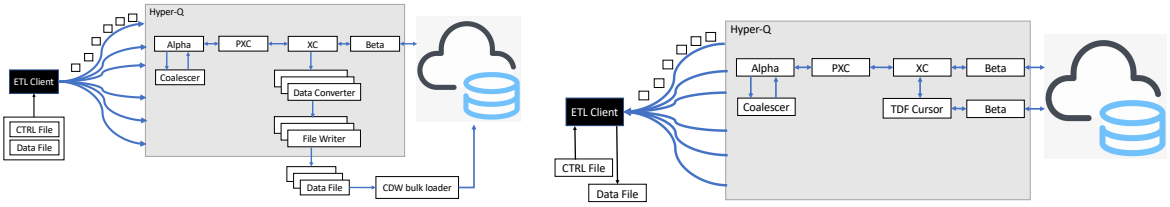
An ETL client follows a specific protocol to communicate with the EDW server. Based on this protocol, the client can formulate certain requests, send them out to the server, and interpret the server's responses. If ETL scripts need to be ported or migrated to a new data warehousing environment, the legacy protocol will be incompatible; legacy client tools will not be functional and will no longer be able to execute the ETL scripts. This means that client scripts and tools must be rewritten and replaced to construct a functional pipeline using the APIs and constructs of the new environment.

There are multiple technical problems involved in this migration process such as porting incompatible features in the new environment, bridging mismatching data types and formats, and rewriting the legacy SQL to maintain the same business logic implemented in the legacy environment. This process can be overwhelming, time-consuming, and error-prone since it typically involves rewriting, testing, and integrating many complex data processing pipelines.

3 SYSTEM ARCHITECTURE

This section describes the ETL pipelines virtualization system we built as part of Hyper-Q [6, 7, 26]. Hyper-Q's ability to follow the same communication protocol of a legacy system is key to enabling virtualization. Hyper-Q listens to connection requests coming from the legacy ETL client and creates a pipeline of connected processes to serve each client session.

Import. Figure 2(a) shows how Hyper-Q virtualizes import jobs. In this case, the legacy client opens a number of data loading sessions to efficiently transmit a large data volume. Then, the client sends a SQL transformation to be applied to data before it gets loaded into the target table. The SQL transformation can be a DML operation to insert/upsert/delete data in the target table (cf. Example 2.1).



(a) Virtualizing import jobs

(b) Virtualizing export jobs

Figure 2: System Architecture

Each client message is captured by a network port listener implemented by the Alpha process, which interacts with a Coalescer process to form complete TCP messages from the raw bytes received over the wire. Each client message is then passed to a Protocol Cross Compiler (PXC) process that decodes the message to extract several pieces of information. For example, a client message that carries a SQL payload requires extracting the SQL statement to be transformed so that it can be executed in the new environment.

When the client sends out a data message, the binary format of the legacy system is used to encode data values in the message. This can be different from how the CDW system interprets and stores data. Hyper-Q handles this mismatch by converting data messages on-the-fly, to create serialized data compatible with the CDW system. We discuss data conversion in Section 4.

The cross-compiler process XC initiates a DataConverter process for each data session. It asynchronously submits each received data chunk and immediately switches back to waiting for the client's next data message. It is crucial to perform data processing in the background while not slowing down data acquisition from the client. We describe data acquisition in Section 5.

In the background, each DataConverter process conducts binary conversion of the data from the format of the legacy system to the CDW system format. After a data chunk is converted, it is sent to the FileWriter process, which receives converted chunks from parallel sessions and serializes them into a disk file. There are multiple FileWriter processes working in parallel, creating a number of disk files from the data chunks received from different sessions. When the file size reaches a threshold, the CDW bulk loader is invoked to copy the local disk file into the cloud store. After data is completely consumed, Hyper-Q initiates an in-the-cloud *COPY* operation to move data to a staging table in the CDW. We discuss integration with CDW interfaces in Section 6.

Afterward, the client starts the application phase, where it sends a DML transformation to be applied to the loaded data. Hyper-Q transforms the incoming DML statement to create an equivalent rewrite compatible with the CDW system. The rewritten DML statement uses the staging table as its source in order to affect data in the target table. The Beta process handles the execution of the transformed statement and the decoding of its returned results. After execution completes, Hyper-Q reports to the client the results of ETL jobs in terms of the number of added/updated/deleted rows in the target table and the number of errors detected in input data or violations of integrity constraints.

Export. Figure 2(b) shows the architecture when virtualizing an export job. In this case, data needs to travel in the reverse direction. The client starts a number of export sessions to receive a large volume of data from the server efficiently. The data to be exported is retrieved by executing a *SELECT* statement in

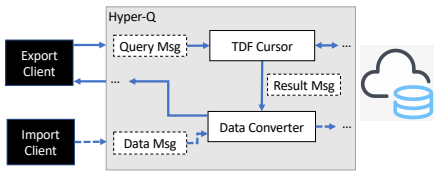


Figure 3: Import/Export Data Conversion

the CDW. Hyper-Q uses a TDFCursor process which allows on-demand retrieval and buffering of result chunks received from the CDW system. TDF (Tabular Data Format) is an internal binary data message representation designed to be an extensible format that can handle arbitrarily large nested data.

Each client export session sends a request number for the chunk order in the full query results. Hyper-Q buffers chunks received by the TDFCursor process in advance and associates each chunk with its order to serve client sessions requesting different chunks. For each request, the data chunk needs to be encoded using the format of the legacy system allowing the client to interpret the returned data correctly and dump it into an exported file.

The ability to follow the legacy system protocol and cross-compile ETL script statements was achieved by extending Hyper-Q's protocol engine. We designed and implemented the DataConverter, FileWriter, and TDFCursor components to ensure high performance, allowing ETL jobs to function as if they were running in the legacy environment. No changes in job scripts, load/export utilities, and configurations are needed. This provides enterprise customers the solution to run legacy ETL pipelines out-of-the-box in foreign CDW environments with minimal migration effort. In order to conform to this ability, Hyper-Q does not use CDW-specific export techniques that do not match the expected data format by the legacy systems.

4 DATA CONVERSION

CDWs use different data representations and formats compared to legacy ones. To bridge discrepancies and make ETL jobs database-agnostic, Hyper-Q intercepts the data communication, converts data formats, and redirects converted data to its destination. Figure 3 shows the data conversion process in Hyper-Q. Data conversion needs to handle both data messages received during import jobs and query messages received during export jobs.

Hyper-Q captures data messages and converts them on-the-fly to create serialized data compatible with the CDW system. These messages are handled by parallel DataConverter processes, one per client session. Depending on the target system and input data, the data conversion process can vary from a simple conversion of binary data formats to a more sophisticated conversion

that includes detecting *null* values, handling empty strings, and escaping special characters. After a message is converted, it is sent to the FileWriter process to be serialized into a disk file.

Query messages include requests to perform staging/error table operations as well as executing SELECT statements as the source of data to be exported in export jobs. As Hyper-Q reports back query results, it needs to convert them into the representation the legacy client expects. Hyper-Q retrieves query results on demand through a TDFCursor process that connects to the CDW. The result is retrieved in batches and packaged according to the TDF format (cf. Section 3). The PXC process then unwraps TDF packets to extract result rows and convert them into the legacy system's format.

Even when the data volume is large, ETL jobs need to perform as if running in the legacy environment with negligible overhead. To achieve this, Hyper-Q does lazy parsing of data messages and parallel conversion of query results. To avoid slowing down data acquisition from the client, when the PXC process receives a data message as part of an import job, it does not parse the data synchronously, i.e., it does not block the client from sending more data messages while previous messages are still being processed. Instead, it pushes data messages to a DataConverter process that performs the conversion in the background asynchronously, while the PXC returns to the client to request the next data message. As a result, a back-pressure mechanism is needed to avoid overwhelming the system with unprocessed data messages.

5 DATA ACQUISITION

Achieving high throughput during the acquisition phase is important, given the large volumes of data involved in ETL. Hyper-Q needs to mimic the performance characteristics of the EDW. For example, an ETL client might use parallel sessions to transmit data to the EDW. Hyper-Q supports this pattern by allowing multiple sessions to run in parallel for the same ETL job, with each session being handled by a separate internal worker. Similarly, loading into the CDW must also be optimized. The performance characteristics of the EDW and CDW often differ, and bridging the gap between the two can be challenging.

To maximize throughput, Hyper-Q processes incoming data using a pipelining model. The first stage reads each chunk of incoming data. ETL clients typically use a synchronous protocol requiring an acknowledgment of one chunk before sending the next. Hyper-Q does minimal processing on the chunk before sending an acknowledgment and handing the chunk to the second stage, the DataConverter. Data conversion on a chunk may take longer than receiving a chunk, so to avoid letting conversion become a performance bottleneck, several chunks are converted concurrently. Converted chunks are ordered and passed to the next stage, the FileWriter. The FileWriter serializes converted data chunks to the disk; performing this task in a separate stage prevents fluctuations in I/O performance from stalling the DataConverter workers.

The FileWriter is tuned to the needs of the CDW; the maximum size of the serialized file is chosen to maximize the load performance into the CDW. It also performs any operations needed to finalize the serialized files, such as applying compression, to prepare for transmission to the CDW. Handling these steps in a separate, concurrent process further insulates the processes from any momentary performance impacts that might occur as these operations occur. Depending on which type of API will be used to upload the data to the CDW, multiple FileWriter may

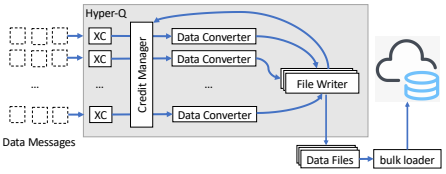


Figure 4: Data Acquisition and back-pressure

run concurrently to parallelize the serialization process further. Finalized files are passed to the final stage of the pipeline, where they are uploaded to the CDW and imported.

A design challenge with any pipeline architecture is avoiding an overload of the system when a later stage is slower than earlier stages. For example, if Hyper-Q reads chunks from ETL clients faster than the FileWriter can write the converted chunks to the staging files, converted chunks can pile up in memory as they wait for the slow FileWriter to catch up.

One possible approach to prevent this would be to synchronize the pipeline. For example, Hyper-Q could wait to acknowledge each incoming data chunk until it's been written to disk. However, this type of synchronization would delay the acknowledgment of the chunk and slow data acquisition from the ETL client. To avoid this slowdown, acknowledgment is sent immediately. An internal watchdog CreditManager process was implemented to provide a lightweight back-pressure mechanism that only kicks in if the DataConverter and/or FileWriter processes fall behind.

Figure 4 illustrates the back-pressure mechanism. When a new ETL job begins, a CreditManager starts with a set number of credits in its pool proportional to the expected number of concurrent sessions and FileWriter processes. When a session process is about to pass another data chunk along for conversion, it first requests a credit. If the credit pool is not empty, one will be immediately returned, and the credit will then be passed along to the DataConverter and then to the FileWriter, where it will be returned to the pool just before the data is written to disk. If the credit pool is empty, the session process requesting the credit will block until a new credit becomes available.

Having a number of credits greater than the number of sessions allows a buffer of several data chunks "in flight" for each session before back-pressure is applied. If we run out of credits, data acquisition will be slowed until more credits become available. With a small credit pool, the back-pressure might slow down the throughput more than necessary. However, with too many credits, too many parallel DataConverter processes may run simultaneously, and Hyper-Q could be overloaded. Good throughput can be maintained with the right balance without overloading the system.

In real-world environments, several ETL acquisitions run concurrently against a single Hyper-Q node. To maximize throughput and avoid overloading the system in such situations, one CreditManager is spawned per Hyper-Q node, with each CreditManager being shared for all concurrent ETL jobs on the node.

6 INTEGRATION WITH CLOUD INTERFACES

Sections 4 and 5 describe how to efficiently generate intermediate files in a format compatible with the cloud interface. Once the data is ready, the next step is to upload the files to a remote storage account. We extended Hyper-Q's interface to support uploading bulk data to the cloud store, e.g., [1, 2], using cloud bulk loaders.

CDWs offer utilities to upload local data files to remote storage accounts [5, 21, 22]. Some tuning may be needed to upload data files efficiently. For example, data compression can improve upload speed if the communication link between the Hyper-Q server and the CDW is slow. It may also be more efficient to upload a directory of files rather than individual files. Furthermore, tuning the intermediate file size is important. A small file size allows more data writing parallelism and fast uploading into the remote storage. On the other hand, a large number of files could impact the efficiency of data copying from the storage account to the CDW staging tables. Hyper-Q exposes these different tuning parameters that the customers can configure according to different ETL job requirements.

The next step is to trigger COPY operations to move uploaded data files into the staging table. Staging tables are needed to apply the transformation logic captured by DML statements in the ETL job. We use a two-step approach to apply these transformations. First, data is serialized and uploaded into the CDW staging table. The staging table is constructed using data types corresponding to what was used by the ETL script. For example, a Unicode character type in the source script could be mapped to the national varchar type in the CDW type system. Hyper-Q transparently performs type mapping and data conversion/serialization across type systems to produce data compatible with the target system. Then, Hyper-Q applies the required transformations by executing the transformed DML statement in the ETL script. The transformation logic is written using the EDW's legacy SQL syntax. Hyper-Q parses and transforms the statement into equivalent syntax compatible with the target CDW.

7 ERROR HANDLING

An important aspect of ETL workflows is the handling of errors. Errors can occur for various reasons and can be broadly classified into two categories:

- *Data Errors*: the input file could have an incorrect number of fields or the wrong data format.
- *Transformation Errors*: applying DML transformation could raise errors such as violation of uniqueness constraints.

In interactive query interfaces, an error during query processing is immediately reported to the user and the remainder of the transaction is aborted. However, errors in ETL jobs do not result in suspending the job. Instead, when an erroneous tuple is encountered, it is skipped, an error is recorded in an error table, and the ETL job proceeds. Later, the error tables are inspected by the user.

Example 7.1. Consider the loading script in Example 2.1 and the data file provided in Figure 5(a). Executing the loading script will result in these errors during the application phase:

- The second and third rows do not contain valid dates in the third field, so casting these values to the DATE type will fail.
- The fourth row violates the uniqueness constraint on the CUST_ID column of the target table.

These errors are recorded in the error tables specified in the loading script, PROD.CUSTOMER_ET and PROD.CUSTOMER_UV. Figure 5(b) and (c) shows the error tables as populated by the legacy data warehouse. The final result of loading is shown in Figure 5(d). It contains all tuples that could successfully be loaded and transformed from the data file. □

Hyper-Q supports error handling by effectively wrapping the application phase in a try-catch block. If an error is encountered during processing, an error tuple is formed and inserted either into the transformation error table for general errors or the uniqueness violation error table if the target table has a uniqueness constraint defined and a violation of that constraint was detected. The CDW might not provide native support for uniqueness constraints. In those cases, Hyper-Q enforces uniqueness through emulation [26].

During the transformation phase, the bulk processing of tuples poses challenges to reporting errors with tuple-level granularity. For example, suppose a tuple violates a uniqueness constraint. In that case, the error will be observed at the level of the chunk containing the faulty tuple rather than at the tuple level. Moreover, all remaining tuples in the chunk will be discarded, even if they do not violate the constraint. However, applications often require that as many tuples as possible are loaded into the target tables, and the error tables record all errors together with the violating tuples so that they can be corrected in a post-processing step.

We devised an adaptive error handling mechanism to satisfy such requirements without sacrificing performance. When an error is encountered, we recursively repeat the application step on smaller data chunks. If transforming a chunk of the input still results in an error, we split the chunk further and proceed recursively. Adaptive error handling stops when the input chunk can no longer be split or when the processing hits a preconfigured limit. The former happens when the chunk contains an individual tuple, in which case we record the tuple in the corresponding error table.

Users can define the following control parameters: *max_errors* and *max_retries*. The first one controls the maximum number of individual errors to record before the retry logic is aborted. The second one controls the maximum number of times an input chunk is split before aborting. For datasets containing multiple errors, using these parameters prevents the adaptive error handling from spending a lot of time finding each error. Instead, it reports ranges of tuples that cannot be transformed correctly.

Figure 6 shows an example error table after loading the data file in Example 7.1 with adaptive error handling enabled and *max_errors=2*. Note that after exhausting the allowed number of individual errors, we recorded an error specifying that the remaining chunk of rows (rows 4-5) include one or more errors but will not be further split.

8 CASE STUDY

In this section, we report on implementing Hyper-Q in a production environment at a large retail organization. They offer a variety of businesses ranging from food retail and wholesale to insurance services and employ over 63,000 employees at over 7,000 locations.

Business locations contribute data (such as sales numbers) daily. These data need to be ingested, cleaned, and pre-processed for reporting. The high number of individual business locations as well as the variety of product categories, result in a complex ETL process, which was built and optimized for an existing on-premises EDW over many years. 127 batch groups are executed under a strict SLA, requiring ETL processing to begin after midnight and be complete by 6 a.m. Each group consists of a number of sequential steps, ranging from custom file preparation to bulk loading of data into the data warehouse and applying transformations inside the data warehouse. Between groups, dependencies control the execution order. These dependencies also effectively limit the degree of parallelism between batch groups and the ability to run them independently.

123|Smith|2012-01-01
456|Brown|xxxxx
789|Brown|yyyyy
123|Jones|2012-12-01
157|Jones|2012-12-01

(a) Data file

PROD.CUSTOMER_ET

SEQNO	ERRCODE	ERRFIELD
2	2666	JOIN_DATE
3	2666	JOIN_DATE

(b) Transformation errors

PROD.CUSTOMER_UV

CUST_ID	CUST_NAME	JOIN_DATE	SEQNO	ERRCODE
123	Jones	12/12/01	4	2794

(c) Uniqueness violations

PROD.CUSTOMER

CUST_ID	CUST_NAME	JOIN_DATE
123	Smith	12/12/01
157	Jones	12/12/01

(b) Successfully loaded tuples

Figure 5: Legacy Error and Target Tables in Example 7.1

ErrorCode	ErrorField	ErrorMessage
3103	JOIN_DATE	DATE conversion failed during DML on PROD.CUSTOMER, row number: 2
3103	JOIN_DATE	DATE conversion failed during DML on PROD.CUSTOMER, row number: 3
9057	NULL	Max number of errors reached during DML on PROD.CUSTOMER, row numbers: (4, 5)

Figure 6: Error Table for Example 7.1 with Adaptive Error Handling

Rewriting the ETL process from scratch would necessitate unpacking the full complexity of the ETL process, from the high-level dependencies down to custom date formats in the input files. A rewrite was deemed infeasible, and they decided to use Hyper-Q to migrate their on-premise EDW to Azure Synapse Analytics. Hyper-Q allowed the migration to proceed without having to rewrite the vast majority of their existing applications, including the ETL processes. Hyper-Q's support for existing bulk load client tools, its ability to leverage modern bulk load technologies on the target data warehouse for optimal throughput, and the preservation of semantics for requests submitted through Hyper-Q all combine to enable a seamless transition to the cloud.

Less than 1% of the queries in ETL jobs had to be rewritten manually. Most manual rewrites are highly localized, i.e., they concern a single construct within a query, and they typically include clauses or data types not supported by the CDW. Rewriting ETL queries is time-consuming, especially during the later stages of the migration process. We use qInsight [4] to identify parts of ETL jobs that need to be rewritten upfront. The lessons we learned during this migration process include establishing a standard process to address query rewrites early on and realizing that several of these rewrites are meaningful and effective for the original EDW as well.

9 EXPERIMENTS

In this section, we present experiments to analyze the overall performance and scalability of Hyper-Q's ETL job virtualization and the individual virtualization phases using real-world jobs that we ran through Hyper-Q on Azure Synapse Analytics.

Performance with Dataset Size. The first experiment is to analyze system performance and scalability with different dataset sizes in number of rows. Each row has data with an average of 500 bytes. The objective of the tested ETL job is to extract, transform and load raw data into a CDW target table. Figure 7 shows that the total job execution time (Y-axis) increases sub-linearly with the increase in dataset size (X-axis). Also, it shows that most overhead is concentrated in the data acquisition phase, in which Hyper-Q needs to do data conversion and serialization. The overhead of the application phase, where DML operations run in the CDW comes next. It is clear that other steps, which include startup and teardown, have minimal overhead that is not affected by the data size. After analyzing the relative increase in processing times of different phases with respect to a baseline of the smallest dataset with a size of 25 million rows,

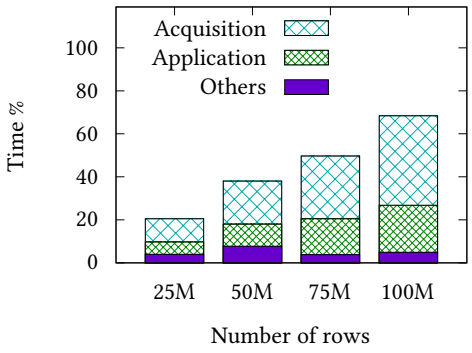


Figure 7: Performance with Different Dataset Sizes

we observe the following. While each phase's processing time increases sub-linearly, the application phase shows a slower increase rate compared to the acquisition phase. For example, for 100 million rows (4X larger), the increase in acquisition phase time is 340%, while for the application phase, it is only 270%. This is due to the bulk processing nature of the DML statements that Hyper-Q generates. This allows the CDW to run DML operations on the whole staging table to affect the target table using a few SQL statements that are optimized by the CDW.

Scalability with Row Width. The second experiment measures how the row width affects the overall performance. Four datasets are used for this experiment. They have the same overall size, while they differ in average row width. One dataset has an average row width of 250 bytes and 100 million rows, while another dataset has 4X average row width and 25% of the number of rows. Figure 8 shows that having larger row width results in better performance. For the acquisition phase, the overhead decreased due to the smaller number of iterations needed for data conversion and serialization within each data chunk received by Hyper-Q.

Scalability with Compute Resources. ETL clients improve data acquisition throughput by using parallel sessions to feed data to the EDW. Using parallel sessions mitigates network and disk latency. With Hyper-Q's approach of acknowledging each chunk of incoming data immediately, we utilize the resources of the machine it resides on regardless of the number of client sessions.

Our experiments show that the acquisition rate is the same when using 2, 4, 8, 12, or 16 parallel sessions. Figure 9 shows

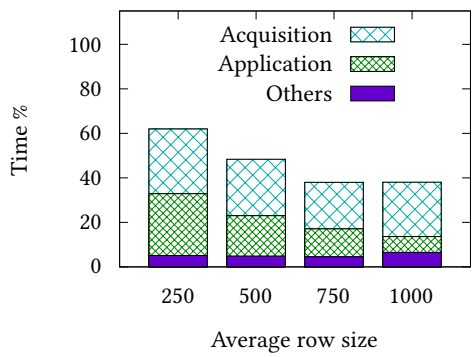


Figure 8: Effect of Row Width on Bulk Load Performance

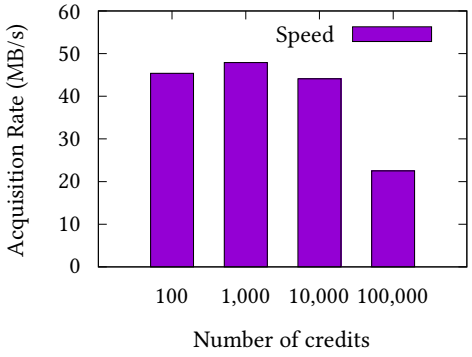


Figure 10: Data Acquisition Scalability with No. Credits

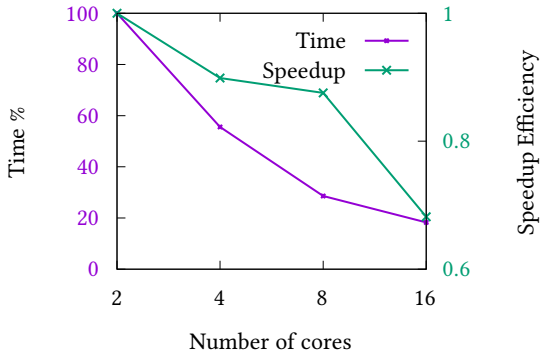


Figure 9: Data Acquisition Scalability with No. CPU Cores

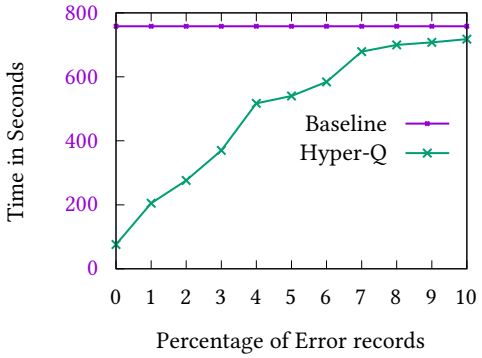


Figure 11: Error Handling Performance

data acquisition scalability when using different numbers of CPU cores on the Hyper-Q machine. The application phase is not measured here since it is not affected by the number of cores. The wall clock time is used in this experiment and is represented as a time % of the 2 cores experiment and is shown on the left Y-axis. The other measure is called the speedup efficiency (S):

$$S = T_s/(T_p * P)$$

where T_s is the wall clock time of a job with the baseline settings (using 2 cores), T_p is the wall clock time using P multiple resources of the baseline setting. Figure 9 shows that Hyper-Q can benefit from the available resources with a good speedup efficiency until 16 cores are used, at which the speedup efficiency degradation is caused by the setup and teardown overhead associated with the acquisition phase, which is done regardless of the number of cores.

Scalability with CreditManager Pool Size. As discussed in Section 5, the CreditManager provides a back-pressure mechanism. Since there is a limit to the number of credits, it is of interest to know more about the performance impact of varying this limit. An experiment was performed wherein 100,000,000 records comprising approximately 97GB of data were loaded through Hyper-Q into a 50-column table. Figure 10 shows that the number of credits did not affect the data acquisition rate up to a certain point. This reflects the easily parallelizable workload involved; there is minimal coordination cost added for each new DataConverter process, and running more of them in parallel does not automatically incur a significant performance penalty. However, eventually, the per-process overhead (i.e., context switching) inevitably begins to dominate the cost of the actual work. This effect becomes apparent when the number of credits is pushed to one hundred thousand and beyond. Finally, in one experimental run not shown in Figure 10 where the number of credits was set to one million, Hyper-Q ran out of memory and crashed before all of the records could be loaded. These results demonstrate that the CreditManager is not just beneficial to performance but is essential for Hyper-Q to handle large ETL jobs.

Error Handling. As discussed in Section 7, error handling is done by Hyper-Q through an adaptive mechanism. The following experiment compares the performance of a baseline system with Hyper-Q as the number of errors increases. The baseline system loads data records using singleton inserts, and when an erroneous tuple is encountered, it is inserted right away into the error log. Alternatively, Hyper-Q does bulk loading and follows the approach discussed in Section 7. In Figure 11, the y-axis shows the elapsed time in seconds, while the x-axis is for the percentage of erroneous records out of the total number of data records. Hyper-Q significantly outperforms the baseline approach when the percentage of errors is none or small. The performance of hyperq is impacted as the number of errors increases, while the baseline system shows consistent performance since it does not need special error handling. There is a steep increase in time comparing no errors to 1% errors since Hyper-Q needs to trigger the error handling mechanism once an error is found, which adds to the overall overhead. Furthermore, the elapsed time increases as the number of errors increases in order for Hyper-Q to look

for the additional errors. A smoother curve is found when finding more errors requires going over most data tuples to identify all erroneous records. Despite such performance degradation, Hyper-Q outperforms the baseline system even with a relatively high percentage of errors (i.e., 10%). Legacy systems have an advantage as these systems perform error handling internally without emulation. Hyper-Q overcomes such overhead by limiting the maximum number of errors to detect.

10 RELATED WORK

Replatforming EDWs to the cloud has been on the agenda of many enterprises in recent years. Cloud data warehouse providers, such as Amazon Redshift [14], Azure Synapse Analytics [24], and BigQuery [20] can free enterprises from having to procure and manage their own infrastructure and data warehouse systems, reducing upfront costs and staffing needs. Another key factor is elasticity, which allows for scaling the system size up and down.

Migrating data across heterogeneous systems is the focus of data migration solutions [8, 9, 16]. They can be used in the same ecosystem as Hyper-Q to help with the initial data migration, but they fall short when it comes to migrating continuously-running legacy ETL pipelines. The pipeline components are full of customization needed for the legacy system. This often requires a manual and expensive rewrite to make the pipelines functional in the new system, which can be as high as 50% of the workload [18, 19]. Repointing ETL through Hyper-Q and leveraging the emulation of missing features and legacy database semantics allows customers to make ETL pipelines portable while avoiding costly rewrites for more than 99% of the workload.

Query federation [12, 15, 17] and schema/data integration solutions [13, 25] primarily focus on source selection problems. The data source that can satisfy query requirements is picked, and input queries are rewritten accordingly. This is often achieved using source wrappers and rewriting rules for identifier and table names under the general assumption that individual sources have a high degree of query language compatibility. In ETL migration projects, legacy system language and semantics could impact a broad scope of interacting system components. The mismatches with the new target systems are often proliferated when data is handed off from one component to another, as in typical export-import-report pipelines.

Data virtualization solutions [3, 10, 27] improve the availability of data integrated from many data sources. However, solutions that rely on *offline* static rewriting are complex to implement in environments where queries are dynamically generated or are data-driven. Hyper-Q's approach of *online* dynamic rewriting allows users to seamlessly migrate the ETL pipelines by simply repointing legacy applications to the virtualization mid-tier.

11 CONCLUSION

We introduced a novel technique for transparent replatforming of legacy ETL pipelines to modern cloud data warehousing environments without requiring code changes in legacy applications. The solution builds on the concepts of adaptive real-time virtualization and allows for the immediate adoption of cloud-native technologies without interrupting daily business operations. We presented a real-world case study discussing the deployment of our system in production settings and an experimental analysis of the scalability and efficiency of our solution.

REFERENCES

[1] 2020. *Amazon S3.* https://aws.amazon.com/s3/

[2] 2020. *Azure Blob Storage.* https://azure.microsoft.com/en-us/services/storage/blobs/

[3] 2020. *Denodo Data Virtualization.* http://www.denodo.com/

[4] Amirhossein Aleyasen, Mark Morcos, Lyublena Antova, Marc Sugiyama, Dmitri Korablev, , et al. 2022. Intelligent Automated Workload Analysis for Database Replatforming. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022.* ACM, 2273–2285. https://doi.org/10.1145/3514221.3526050

[5] Amazon. 2021. *AWS s3 cp.* Retrieved December 2022 from https://docs.aws.amazon.com/cli/latest/reference/s3/cp.html

[6] Lyublena Antova, Rhonda Baldwin, Derrick Bryant, Tuan Cao, Michael Duller, , et al. 2016. Datometry Hyper-Q: Bridging the gap between real-time and historical analytics. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016.* ACM, 1405–1416. https://doi.org/10.1145/2882903.2903739

[7] Lyublena Antova, Derrick Bryant, Tuan Cao, Michael Duller, Mohamed A. Soliman, and Florian M. Waas. 2018. Rapid Adoption of Cloud Data Warehouse Technology Using Datometry Hyper-Q. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018.* ACM, 825–839. https://doi.org/10.1145/3183713.3190652

[8] Attunity/Qlik. 2020. *Qlik Data Integration Products.* https://www.qlik.com

[9] SAP BODS. 2020. *SAP BO Data Services.* https://www.sap.com

[10] CompilerWorks. 2020. *CompilerWorks Transpiler Solution.* http://www.compilerworks.com/

[11] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, , et al. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016.* ACM, 215–226. https://doi.org/10.1145/2882903.2903741

[12] Amol Deshpande and Joseph M. Hellerstein. 2002. Decoupled Query Optimization for Federated Database Systems. In *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002.* IEEE Computer Society, 716–727. https://doi.org/10.1109/ICDE.2002.994788

[13] Aaron J. Elmore, Jennie Duggan, Mike Stonebraker, Magdalena Balazinska, Ugur Çetintemel, Vijay Gadepally, Jeffrey Heer, Bill Howe, Jeremy Kepner, , et al. 2015. A Demonstration of the BigDAWG Polystore System. *Proc. VLDB Endow.* 8, 12 (2015), 1908–1911. https://doi.org/10.14778/2824032.2824098

[14] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon Redshift and the Case for Simpler Data Warehouses. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015.* ACM, 1917–1923. https://doi.org/10.1145/2723372.2742795

[15] San-Yih Hwang, Ee-Peng Lim, H.-R. Yang, S. Musukula, K. Mediratta, M. Ganesh, Dave Clements, J. Stenoien, and Jaideep Srivastava. 1994. The MYRIAD Federated Database Prototype. (1994), 518. https://doi.org/10.1145/191839.191986

[16] Informatica [n.d.]. *Informatica.* https//www.informatica.com/

[17] Holger Kache, Wook-Shin Han, Volker Markl, Vijayshankar Raman, and Stephan Ewen. 2006. POP/FED: Progressive Query Optimization for Federated Queries in DB2. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006.* ACM, 1175–1178. http://dl.acm.org/citation.cfm?id=1164237

[18] Leaplogic. 2022. *LeapLogic.* https://www.leaplogic.io/

[19] Liberatii. 2022. *Liberatii Gateway.* https://www.liberatii.com/

[20] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. 2020. Dremel: A Decade of Interactive SQL Analysis at Web Scale. *Proc. VLDB Endow.* 13, 12 (2020), 3461–3472. https://doi.org/10.14778/3415478.3415568

[21] Microsoft. 2021. *AZCOPY.* Retrieved December 2022 from https://docs.microsoft.com/en-us/azure/storage/common/storage-use-azcopy-v10

[22] Microsoft. 2021. *Bulk copy program.* Retrieved December 2022 from https://docs.microsoft.com/en-us/sql/tools/bcp-utility?view=sql-server-ver15

[23] Microsoft. 2021. *Microsoft Azure Synapse Analytics.* Retrieved December 2022 from https://docs.microsoft.com/en-us/azure/sql-data-warehouse/massively-parallel-processing-mpp-architecture

[24] Srinath Shankar, Rimma V. Nehme, Josep Aguilar-Saborit, Andrew Chung, Mostafa Elhemali, , et al. 2012. Query optimization in microsoft SQL server PDW. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012.* ACM, 767–776. https://doi.org/10.1145/2213836.2213953

[25] Alkis Simitsis, Kevin Wilkinson, Malú Castellanos, and Umeshwar Dayal. 2012. Optimizing analytic data flows for multiple execution engines. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012.* ACM, 829–840. https://doi.org/10.1145/2213836.2213963

[26] Mohamed A. Soliman, Lyublena Antova, Marc Sugiyama, Michael Duller, Amirhossein Aleyasen, Gourab Mitra, Ehab Abdelhamid, Mark Morcos, Michele Gage, Dmitri Korablev, and Florian M. Waas. 2020. A Framework for Emulating Database Operations in Cloud Data Warehouses. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020.* ACM, 1447–1461. https://doi.org/10.1145/3318464.3386128

[27] Tibco. 2020. *Tibco Data Virtualization.* https://www.tibco.com