Rapid Adoption of Cloud Data Warehouse Technology Using Datometry Hyper-Q

L. Antova, D. Bryant, T. Cao, M. Duller, M. A. Soliman, F. M. Waas

Datometry Inc.
300 Brannan St #610, San Francisco, CA 94107, U.S.A.
www.datometry.com

ABSTRACT

The database industry is about to undergo a fundamental transformation of unprecedented magnitude as enterprises start trading their well-established database stacks for cloud-native database technology in order to take advantage of the economics cloud service providers promise. Industry experts and analysts expect 2017 to become the watershed moment in this transformation as cloud-native databases finally reached critical mass and maturity. Enterprises eager to move to the cloud face a significant dilemma: moving the content of their databases to the cloud is relatively easy. However making existing applications work with new database platforms is an enormously costly undertaking that calls for rewriting and adjusting of 100's if not 1,000's of applications.

Datometry has developed a next generation virtualization technology that lets existing applications run natively on new database systems. Using Datometry's platform, enterprises can move rapidly to the cloud and innovate and create competitive advantage as a matter of months instead of years. In this paper, we present Datometry Hyper-Q, a new type of virtualization platform that implements this vision. We describe technology and use cases and demonstrate effectiveness and performance of this approach.

1. INTRODUCTION

Over the course of the next decade, the database market, a $40 billion industry, is about to face fundamental disruption. Cloud-native databases such as Microsoft SQL Data Warehouse [22], Amazon Redshift [18], Google BigQuery [7], and others promise to be functionally equivalent to their counterparts on premises, yet provide enterprises with unprecedented flexibility and elasticity. All while being highly cost-effective: Instead of considerable up-front expenses in the form of hardware and software license costs, cloud-native databases offer a pay-as-you-go model that reduces databases effectively to a set of APIs. They let users query or manipulate data freely, yet shield them from the burden and headaches of having to maintain and operate their own database.

The prospect of this new paradigm of data management is extremely powerful and well-received by IT departments across all industries [2]. However, it comes with significant adoption challenges. Through applications that depend on the specific databases they were originally written for, database technology has over time established some of the strongest vendor lock-in in all of IT. Moving to a cloud-native database requires adjusting or even rewriting of existing applications and migrations may take up to several years, cost millions, and are heavily fraught with risk.

CIOs and IT leaders find themselves increasingly in a conundrum weighing the benefits of moving to a cloud-native database against the substantial switching cost. The situation is further complicated by a myriad of parameters and configurations to choose from when moving to the cloud: different systems offer different characteristics and represent technology at different levels of maturity.

In this paper, we present *Adaptive Data Virtualization (ADV)*, based on a concept originally developed to bridge data silos in real-time data processing [13], and develop a platform approach that solves the problem of adopting cloud-native databases effectively and at a fraction of the cost of other approaches. ADV lets applications, originally developed for a specific database on-premises, run natively in the cloud—without requiring modifications to the application. The key principle of ADV is the intercepting of queries as they are emitted by an application and subsequent on-the-fly translation of those queries from the language used by the application (SQL-A) into the language provided by the cloud-native database (SQL-B), see Figure 1. The translation must be aware of semantic differences between the systems and compensate for missing functionality in many cases.

The overwhelmingly positive receptions by customers and technology partners alike underlines the benefits of ADV over a conventional migration approach along three dimensions:

1. *Short time to value:* ADV can be deployed instantly and requires little or no build-out and implementation. This eliminates the time for adjusting, rewriting or otherwise making existing applications compatible with the new target platform.

2. *Low cost:* Much of the cost in migration projects is incurred by the manual translation of SQL embedded in

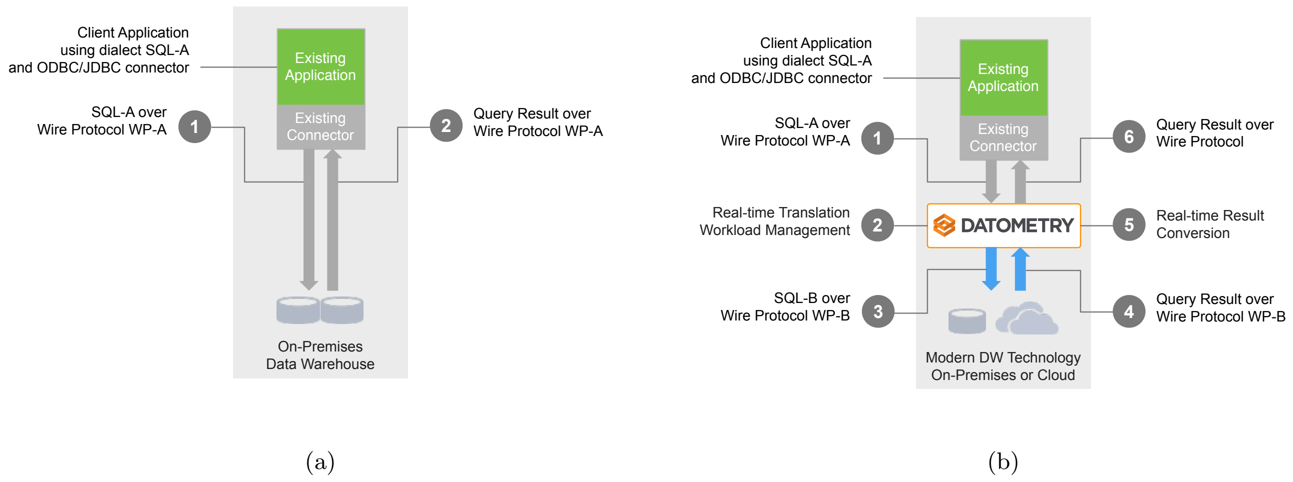

Client Application
using dialect SQL-A
and ODBC/JDBC connector
Existing Application
Existing Connector
1 SQL-A over Wire Protocol WP-A
2 Query Result over Wire Protocol WP-A
On-Premises Data Warehouse
(a)
Client Application
using dialect SQL-A
and ODBC/JDBC connector
Existing Application
Existing Connector
1 SQL-A over Wire Protocol WP-A
2 Real-time Translation Workload Management
3 SQL-B over Wire Protocol WP-B
DATOMETRY
6 Query Result over Wire Protocol
5 Real-time Result Conversion
4 Query Result over Wire Protocol WP-B
Modern DW Technology On-Premises or Cloud
(b)

Figure 1: Application/database communication before re-platforming (a) and after (b). Applications remain unchanged, continue to use query language SQL-A.

applications and subsequent iterations of testing the rewrites. Being a software solution, ADV eliminates both manual interaction and time-consuming and expensive test phases entirely.

3. *Reduced risk:* Since ADV leaves existing applications unchanged, new database technology may not only be adopted rapidly, but also without having to commit significant resources or time. This reduces the risk to the enterprise substantially as projects can be fully tested in advance and, in the unlikely event that results do not meet expectations, switching back to the original stack is instant and inexpensive.

In this paper we present use cases and technical underpinnings of ADV and discuss the trade-offs consumers of the technology need to be aware of when making decisions about when to deploy it.

Roadmap. The remainder of the paper is organized as follows. Section 2 reviews the state of the art of database re-platforming today and outlines standard procedures. In Section 3 we develop a catalog of desiderata or requirements for ADV as a general concept. Section 4 provides a detailed overview of the technology and relevant implementation choices. The major use cases that we identified in practice are illustrated in Section 5 and complemented with performance analysis in Section 6. We conclude the paper with a review of related work in Section 7.

2. CONVENTIONAL DATA WAREHOUSE MIGRATION

The migrating or re-platforming of a database and its dependent application is a highly involved process around which an entire industry has been established. However, practitioners who do not have first-hand experience in this subject matter may not be familiar with the complexities involved. For the reader's benefit we detail the intricacies of what is considered state of the art.

Here, we consider data warehouses as they are the most prevalent and among the costliest databases to deal with. Conversely they offer the biggest benefits when moving to the cloud. Also, we focus on the practical case of moving relational data warehouses which has evolved as the industry's most pressing use case.

For the purpose of the exposition, we assume the decision to move to the new database has been made, i.e., necessary investigations into performance, availability etc. are complete. These are significant tasks in and by themselves, however, they are not in the scope of this paper.

Migration projects have multiple stages. Some of these can be executed in parallel, others require serialization, for the purpose of this paper, we will discuss only technical challenges and skip project management challenges even though they are vital to the success of a re-platforming project.

2.1 Discovery

In order to compile a realistic project plan, a full inventory of all technologies used, i.e., all applications connected to the database, is needed. This activity needs to look at every single component and determine among other things:

1. *Inventory.* Application type and technology for every single client needs to be evaluated including Business Intelligence (BI) and reporting, but also ad-hoc uses. This includes proprietary and 3rd party clients, including embedded use cases where seemingly innocuous business applications such as Microsoft Excel emit queries.

2. *Connectors.* Drivers and libraries used by every connecting component need to be cataloged to understand if they can be replaced. This step often discovers surprising incompatibilities with respect to the new target system and may require custom builds of connectors.

3. *SQL language.* A gap analysis determines what features, specifically complex features, are being used. The understanding of what features may not be available on the downstream system, e.g., recursive query, as well as proprietary features that predate standardization, e.g., pre-ISO rank implementations, drives the design and implementation of equivalent queries and expressions based only on primitives of the new database system.

The discovery phase is a function of the size and complexity of the ecosystem. This phase of a migration typically takes up to 3-6 months, or more.

2.2 Database Content Transfer

The most prominent task though frequently neither the most challenging nor the costliest is the transfer of the content of the original database to the cloud-native database.

2.2.1 Schema Conversion

Transferring the schema from one system to another needs to replicate all relevant structures in the schema definition language of the new system. While most databases by now offer equivalent structures, there are certain structural features that may not be transferable due to limitations in the new system, e.g., Global Temporary Tables, Stored Procedures.

In contrast to logical design features, physical database design including indexes or other physical structures do not necessarily need to be transferred or require separate treatment once the database is transferred. Frequently, physical design choices are not portable or do not even apply to the new database system and can therefore be ignored. A more recent trend is to increasingly eliminate physical design choices altogether, see e.g., Snowflake [14], XtremeData [12].

Treating schema conversion as a separate discipline has severe shortcomings: certain features that cannot be transcribed directly require a workaround that all applications need be made aware of, i.e., simply transcribing the schema in isolation is bound to generate functionally incomplete results. See Section 2.3 for details.

2.2.2 Data Transfer

Once a schema is transferred and established, data needs to be transferred in bulk or incrementally. More than a semantic problem, this is typically a logistical issue: for large amounts of data, most cloud service providers offer bulk transfer through off-line means such as disk arrays shipped via courier services or similar. With today's standard bandwidth available between data centers and the widespread network of cloud data centers, data transfer expect for very large data warehouses rarely constitutes a problem though.

2.3 Application Migration

Adjusting applications to work with the new database is a multi-faceted undertaking. On a technical level drivers, connectivity etc. need be established. Then, SQL be it embedded or in stand-alone scripts needs to be rewritten, and, closely related to the previous, enclosing application logic may need to be adjusted accordingly. To demonstrate some of the challenges when translating queries from one SQL dialect to another, consider the following example:

Example 1 Consider the following query written in Teradata SQL dialect:

```
SEL
   PRODUCT_NAME,
   SALES AS SALES_BASE,
   SALES_BASE + 100 AS SALES_OFFSET
FROM PRODUCT
QUALIFY
   10 < SUM(SALES)
   OVER (PARTITION BY STORE)
ORDER BY STORE, PRODUCT_NAME
WHERE CHARS(PRODUCT_NAME) > 4;
```

There are several vendor-specific constructs which make the query non-portable. For example, SEL is a shortcut for SELECT keyword. QUALIFY filters the output based on the result of a window operation, similar to the HAVING clause for GROUP BY clause. Moreover, the order of the various clauses is not following the standard: ORDER BY precedes WHERE which is not accepted by most other systems. Built-in functions and operators, such as computing the length of a string or date arithmetic, frequently differ in syntax across vendors. Another widely used construct not typically available in other systems is the ability to reference named expressions in the same query block. For example, the definition of SALES_OFFSET uses the expression SALES_BASE defined in the same block. □

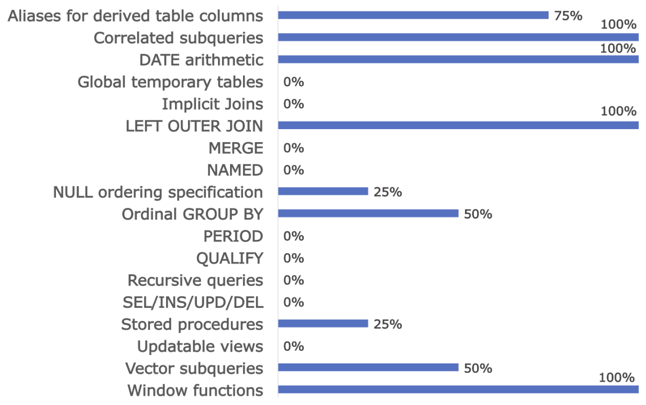


Figure 2: Support for select Teradata features across major cloud databases

Figure 2 shows a selection of query features supported by Teradata and frequently used in analytics workloads, contrasted with the percentage of leading cloud databases supporting them as of the time of writing. While this list is not meant to be complete or representative of Teradata workloads, the shown features include the ones that we have encountered while working with customers to re-platform their applications from Teradata to cloud databases. Some of these features are non-standard SQL extensions introduced by Teradata. For example, QUALIFY is a non-standard clause that combines window functions with predicates. Similarly, implicit joins refers to the ability to implicitly join tables not explicitly referenced in the FROM clause.

Others are standard SQL features. However, they are either not implemented at all, or only partially supported by cloud databases. Those include the ability to specify column names in a derived table alias or the MERGE operation, which inserts and/or updates records in the same statement. While query rewriting may address some of these feature gaps, there are many cases where it may not be possible to do so. Some features need extensive infrastructure on the database side to be functional. We discuss how our solution addresses these different scenarios in Section 4.

In our experience, we can distinguish three categories of difficulty when it comes to rewriting SQL terms and expressions:

1. *The good.* These are fairly simple syntactic rewrites. For example, keywords often differ between systems, there are slight variations between standard built-in function names, date formats may need adjusting, etc. The required changes are rather straightforward and often highly localized; many can be even addressed with textual substitution using regular expressions.

2. *The bad.* The next category requires a complete understanding of the structure of a query including proper name resolution and type derivation. Examples for this category include Teradata's QUALIFY clause, NAMED terms, or the use of ordinals instead of column names in GROUP BY or ORDER BY clauses. While a rewrite is often possible, it may involve restructuring the query in a non-local manner, which is difficult and error-prone, specifically in the case of complex queries. In some cases, the new database system lacks critical functionality, e.g., control flow, which may require pushing parts of the queries to the application layer.

3. *The ugly.* The last category are differences of a subtle nature such as different ordering of NULLs in ORDER BY expressions. They require slightly more complex rewrites, however, they pose a particular difficulty as they may go undetected during the migration project: the original SQL code compiles and executes on the new system and may produce even appropriate results in select cases. However, correctness has been compromised and leads to subtle defects that are hard to spot and correct.

The rewritten SQL expression needs to be placed into the application which may lead to further complications: if a single SQL statement leads to a rewrite consisting of multiple statements for the next system, the control flow of the application needs to be adjusted accordingly.

Application migration outweighs most other processes by far in time and cost.

2.4 Discussion

The previous sections touched on a number of technical issues and are intended to give readers an overall sense of the complexities of migrations. For a complete migration many more aspects including process elements, policies, and business priorities need to be taken into account. For our purposes we can safely omit these and focus on the technical elements. Nevertheless, it appears a number of misconceptions from a technical point of view exist in this. We feel two deserve being called out explicitly.

2.4.1 Migration Paradox

We found many practitioners specifically on the database engineering side maintaining the view that transferring a database was probably not too much of a hassle —and consider the cost associated with a full migration project as grossly exaggerated. Paradoxically, the most difficult task about database migrations is actually not the migrating of the database, i.e., the content transfer, but the adjusting of applications. In fact, the transfer of the content of the database is supported by a variety of tools that make the transfer of database content appear simple *per se.* However, the manual adaptation of applications may require rewriting thousands of lines of SQL, significant portions of the code in which SQL scripts are embedded in etc. Once the magnitude of the changes needed and the risk associated with them is understood, the price tag for a migration appears justified, even when it runs into the millions.

2.4.2 Assumed Independence

Closely related to the "Migration Paradox" is the second most common misconception which assumes independence between transferring content and the adjusting of applications. However, transcribing the schema and/or data cannot be dealt with independently from the migration of applications. Consider the case of missing support of data types, for example the PERIOD type, which describes start and stop of a time range. Few database systems support it. A simple translation would be breaking it into two separate fields for the two components. However, all queries that reference the original PERIOD field need to be rewritten to deal explicitly with the start and stop fields. We used the PERIOD type for the sake of simplicity. However, more complex constructs such as Global Temporary Tables are effectively impossible to translate. As a consequence, migration tools frequently error out in these situations and leave users with an incomplete translation.

3. DESIDERATA FOR DATABASE VIRTUALIZATION

Database Virtualization is a highly complex affair. In the following we lay out requirements such a solution needs to address to be successful. In particular, we organize the desiderata in three groups: (1) functionality, (2) operational aspects around integrating Database Virtualization in an existing IT environment, and (3) managing and supporting of a Database Virtualization solution in a production environment.

We refer to the original database system as DB-A and its language as SQL-A, to the replacement database as DB-B and SQL-B accordingly.

3.1 Functionality

Requirements in this category refer primarily to establishing equivalent behavior between original and emulated database.

- Statements in SQL-A need to be translated into zero, one, or more terms using SQL-B that produce an equivalent response. Differences that need to be reconciled typically include alternate keywords or simple grammatical differences such as TOP as opposed to LIMIT. In some cases, the original statement may be eliminated altogether.

- Besides queries, DDL commands to create, alter or drop objects need to be supported, e.g., management and layout of tables, views, as well as more advanced concepts. DDL that does not have immediately corresponding equivalents may require more elaborate workarounds, e.g., tables originally defined with set semantics may get represented as tables with unique primary keys in systems that do not natively support set semantics.

- Basic data types such as ODBC types are translated straightforwardly. Compound or abstract data types, e.g., PERIOD type, are represented as combination of basic data types. More complex types, including user-defined types, may be serialized as a string representation. Additional metadata may need to be stored to capture the full semantics of the complex data type.

- Support for native wire protocols is needed to ensure the original applications do not need to be modified or altered. This solves the large scale problem of having to replace tens if not hundreds of drivers and connectors across the enterprise. It also guarantees support for applications that are used infrequently and that are not known to the IT staff as actual clients; this is a surprisingly common case in practice.

- Representation and translation of procedural constructs such as calling of functions or stored procedures is required in order to support a large number of established workloads. In the context of this work, we focus on call semantics only and assume DB-B does have appropriate support for representing functions. Research on emulation of procedures and functions is outside the scope of this paper. See also Section 8.

The above list is akin to a task list for migrating manually between databases: the underlying gaps in expressivity need to be addressed in workarounds and implementations. We did not list correctness as we consider it a basic, non-negotiable requirement.

3.2 Operational Aspects

For ADV to meet standards of modern IT, a number of operational properties must be established. These pertain to questions around efficiency, safe operations and integration with existing adjacent systems.

- Latency and throughput may not be impacted significantly by ADV's being in the data path. Most virtualization technologies are of small but measurable overhead, e.g., in compute virtualization 5-8% overhead are typically. It stands to reason that ADV should be of similar impact.

- Optimality is desirable, i.e., queries translated as a result of ADV may not perform substantially slower or in any other way suboptimal when compared to a hand-crafted translation. We note that this property is hard to define and even harder to measure but we list it here for completeness as it does come up as a customers' concern in practice.

- As ADV is dependent on shape and structure of queries and results, strict limits for resource consumption are desirable in order for administrators to be able to safely operate such a system in demanding customer application scenarios.

- Integration with the existing IT ecosystem is required: this includes authentication mechanisms and systems such as Kerberos, Active Directory, or LDAP; it also includes integration with monitoring infrastructure, deployment frameworks, load-balancers and mechanisms for fault-tolerance and high-availability.

- Scalability to satisfy concurrency and throughput requirements is necessary. A successful design needs to provide the ability to scale horizontally—by replicating instances of the ADV solution across independent machines—as well as vertically in being able to leverage additional CPU cores transparently.

This second category of desiderata puts emphasis on integration and some of the points above are clearly dependent on DB-B, the database that is chosen as a replacement.

3.3 Manageability and Supportability

The last category pertains to day-to-day operations of such a system and reflects troubleshooting and diagnosing as well as quality assurance elements.

- While there are typically no guarantees in sophisticated industrial software projects, means to ascertain correctness, including testing hooks or side-by-side tests of different backend systems, can alleviate concerns around correctness. In many cases a rigorously executed test regimen at the vendor site might suffice.

- Elaborate mechanisms for tracing that provide transparency and visibility into the system at runtime can cut short troubleshooting sessions and simplify the pinpointing and diagnosing of software defects. In particular, the ability to quickly discern which component is at fault—application, ADV, or backend database—accelerates defect analysis greatly.

In summary, the desiderata above are not meant as a strict requirements analysis but rather guidelines for the development. In working with customers, we have come to understand that depending on the system and application scenarios at hand, some of these are negotiable, while others may be hard requirements. That means, in many ways the development of ADV needs to be prepared to view them as strict and completely fulfill the requirement or offer a high degree of approximation. See also Section 8 for further discussion.

4. HYPER-Q PLATFORM

In this section, we give a technical deep dive into the architecture and capabilities of Hyper-Q, the world's first ADV solution designed to fulfill the desiderata outlined in Section 3. Hyper-Q automates database functionality mismatch discovery as well as schema and data transfer across databases. However, we focus in this section on the application re-platforming aspect, which we believe outweighs in time and cost other database migration problems.

We describe the architecture of Hyper-Q in Section 4.1. We then show how Hyper-Q framework allows instant application migration across databases through powerful query rewriting (Section 4.2) and feature augmentation (Section 4.3) capabilities.

4.1 Hyper-Q Architecture

Hyper-Q intercepts the network traffic between applications and databases and translates the language and communication protocol on-the-fly to allow application to seamlessly run on a completely different database. Figure 3 shows Hyper-Q architecture. Applications are typically built on top of database libraries including ODBC/JDBC drivers as well as native database vendor libraries such as libpq in Postgres or CLIv2 in Teradata. These libraries are needed to abstract

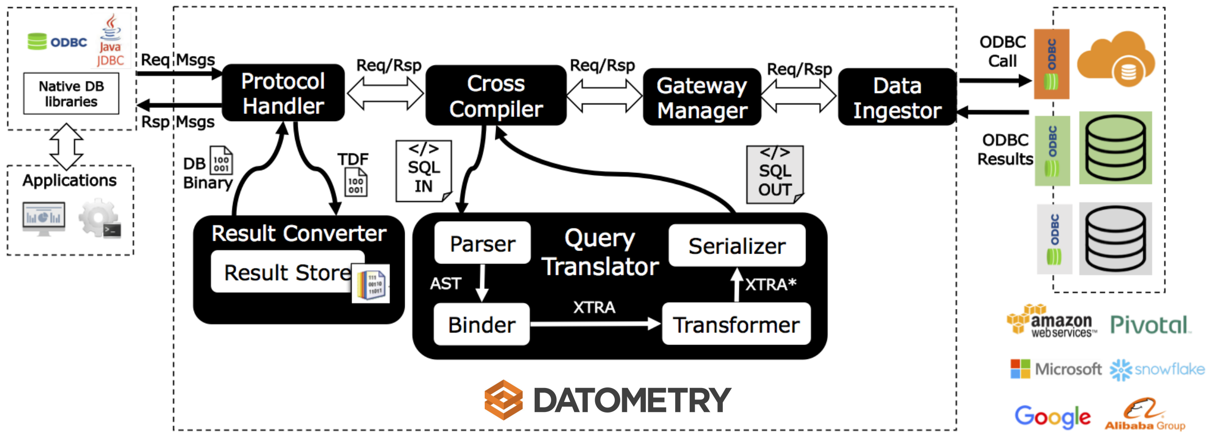


Figure 3: Architecture of Hyper-Q

the details of database communication with the application by providing simple APIs to submit query requests and consume responses.

Hyper-Q intercepts the incoming traffic from the application and rewrites it to match the expectation of a completely different on-premises or in-the-cloud database. When an application request is received, it gets mapped in real time to equivalent request(s) that the new database can comprehend and process. After processing is done, Hyper-Q also does the reverse translation, where database query response is converted into the same binary format of the original database. This is crucial to allow applications to work as expected by providing query results that are bit-identical to the original database. In the next subsections, we describe the details of query and result translation inside Hyper-Q.

4.1.1 *Protocol Handler*

Network traffic between application and database includes an exchange of network messages designed according to the original database specifications. Different constructs need to be implemented to emulate the native communication protocol of the original database. This includes authentication handshake required to establish secure connection between the application and the database, network message types and binary formats, as well as representation of different query elements, data types and query responses. Hyper-Q fully implements these details providing application with the same look-and-feel of its original database even though the application effectively runs on a different target database system.

The Protocol Handler component of Hyper-Q is responsible of intercepting the network message flow submitted by the application, extracting important pieces of information on-the-fly, e.g., application credentials or payloads of application requests, and passing this information down to Hyper-Q engine for further processing. When a response is ready to be sent back to the application, the Protocol Handler component packages the response into the binary message format expected by the application. This completely abstracts the application-database communication details from the rest of Hyper-Q components.

The Protocol Handler component provides native support of communication protocols of different flavors, including JDBC/ODBC protocols, as well as native database communication libraries. For historical reasons, widely different protocol implementations may be introduced at different points of time by the original database to deal with new client types. In many cases, database clients become non-functional with the slightest difference in behavior of the database server. Emulating the protocol by producing identical network traffic of the original database is crucial to use clients functionalities. With Hyper-Q, the application does not need to change at all in order to work with a new database. The message exchange with the new database becomes bit-identical to the message exchange with the original database.

4.1.2 *Algebrizer*

The statements submitted by an application are expressed according to the query language of the original database. Even when an application is built using standard JDBC/ODBC APIs, the SQL text submitted through these APIs is specified according to the syntax and feature set of the original database.

The original database language (or SQL dialect) could be widely different from the query language expected by the new database. The queries embedded in the application logic would thus be mostly broken if executed without changes on a new database. Query rewriting is necessary to port applications written for the original database to the new target database.

The Algebrizer component in Hyper-Q is a system-specific plugin implemented according to the language specifications of the original database. It includes a rule-based parser that implements the full query surface of the original database and a universal language-agnostic query representation called eXtended Relational Algebra (XTRA) used to capture different query constructs. Query algebrization is performed place in two phases: (1) parsing incoming request into an Abstract Syntax Tree (AST) capturing high-level query structure, and (2) binding the generated AST into XTRA expression, where more involved operations such as Metadata lookup and query normalization are performed.

4.1.3 *Transformer*

XTRA is a powerful representation that lends itself to performing several query transformations transparently for query correctness and performance purposes. The Transformer component is the driver responsible for triggering different transformation rules under given pre-conditions. The transformations are plug-able components that could be shared across different databases and application requests. For example, the original database may allow direct comparison of two data types by implicitly converting one to the other. The new database may not have the same capability, and would thus require the conversion to be spelled-out explicitly in the query request. In this case, a transformation that converts a comparison of two data types into an equivalent comparison after explicitly transforming one of the types to the other needs to be applied.

Transformations could also be used to improve the performance of generated queries. For example, if the target database incurs a large overhead in executing single-row DML requests, a transformation that groups a large number of contiguous single-row DML statements into one large statement could be applied.

Controlling which transformation are triggered against a given query request is done by maintaining a map for each target database system associating different XTRA operators with their corresponding transformations. Given an algebrized XTRA expression that includes these operators, the Transformer automatically triggers the relevant transformations to generate a new XTRA expression after applying transformations logic. Transformations could be cascaded, where the output of one transformation represents a valid input to another transformation. The Transformer takes care of running all relevant transformations repeatedly until reaching a fixed point, where no further modifications to the XTRA expression via transformation is possible.

4.1.4 Serializer

Different database systems assume different SQL dialects. This means that we typically need to generate different SQL syntax depending on the type of target database system the application is re-platformed for. This is implemented using the Serializer component of Hyper-Q. Each target database has its own Serializer implementation. These different serializers share a common interface: the input is an XTRA expression, and the output is the serialized SQL statement of that XTRA. Serialization takes place by walking through the XTRA expression, generating a SQL block for each operator and then formatting the generated blocks according to the specific keywords and query constructs of the target database.

4.1.5 Data Ingestor

The Data Ingestor component is an abstraction of ODBC APIs that allows Hyper-Q to communicate with different types of target database systems in an abstract fashion. The APIs provide means to submit different kinds of requests to the target database for execution, ranging from simple queries/DML requests to multi-statement requests, parameterized queries, and stored procedure calls.

The results of these requests are retrieved by Data Ingestor on demand in one or more batches depending on the result size. Result batches are packaged according to Hyper-Q binary data representation, called Tabular Data Format (TDF), which is designed to be an extensible binary format that is able handle arbitrarily nested data. By relying on the TDF representation, the Data Ingestor handles data retrieval in a number of non-straightforward scenarios including handling very wide rows and extremely large result sets.

4.1.6 Result Converter

The query results in TDF representation cannot be consumed directly by the application, since the application expects query results to be formatted in a specific way, as defined by the binary representation of the original database. Hyper-Q needs to do on-the-fly result conversion from TDF into the binary representation that application expects. The Result Converter is the driver for such operation.

TDF packets are unwrapped by Result Converter to extract result rows and convert them into the binary format of the original database. This conversion operation happens in parallel by starting a number of processes where each process handles the conversion of a subset of the result rows. If the original database allows streaming query results to the application, the converted results are streamed directly to Protocol Handler which packages them into network messages to be sent back to the application.

Alternatively, the original database may not allow streaming the results. For example, some databases require that the total number of results is sent to the application first before sending any actual result. In this case, the Result Converter needs to buffer all result rows until they are fully consumed from the target database. When the result size is very large, the buffered results may not fit in memory. In this case, the Result Converter spills the buffered results into disk and maintains the set of generated spill files until result consumption is done. At this point, the converted results are sent to Protocol Handler to be packaged into messages returned back to the application.

4.2 Query Rewriting

In this section, we walk through the architecture of Hyper-Q to illustrate its powerful query rewriting capabilities. We use the following simple example for illustration.

Example 2 Consider the following query:

```
SEL *
FROM SALES
WHERE (AMOUNT, AMOUNT * 0.85) >
ANY (SEL GROSS, NET
     FROM SALES_HISTORY);
```

The query uses a quantified subquery construct that is not natively supported by many target database systems. The construct assumes the following semantics of vector comparison: $(AMOUNT, AMOUNT * .85) > (GROSS, NET) \iff (AMOUNT > GROSS) \vee (AMOUNT = GROSS \wedge AMOUNT * .85 > NET)$. That is, the query finds sales with amounts exceeding any gross sales amount in sales history such that ties are broken using net values.

The generated XTRA expression after the binding phase is given in Figure 4, which is a normalized relational algebra expression that captures the input query. XTRA builds on an uniform algebraic model,

```
+-select
    |-get (SALES)
    +-subq_quantified (GT, ANY)
        |-remap columns: (GROSS, NET)
        |   +-get (SALES_HISTORY)
    +-list
        |-ident (SALES.AMOUNT)
        +-arith (op:*)
            |-ident (SALES.AMOUNT)
            +-const (0.85)
```

Figure 4: Generated XTRA for Example 2

where the output of a given operator depends on operator's inputs as well as operator's type. Different constructs in the query body are captured using different XTRA operators with well-defined semantics. For example, subq_quantified operator captures quantification with (GT, ANY) over a subquery. The inputs to this operator are a relational expression that emits rows with two columns (SEL GROSS, NET FROM SALES_HISTORY)), we well as a comparison expression ((AMOUNT, AMOUNT * .85) > . . .), which is captured using a list of scalar operators capturing vector comparison semantics. The output of subq_quantified is a boolean value that reflects the comparison final result. □

For some target databases, the generated XTRA in Figure 4 cannot be used directly to generate SQL in its current form. The reason is that such target databases do not understand vector comparison, and would thus raise a syntax error. In this case, the Transformer needs to modify the generated XTRA to match the capabilities of the target database system. This can be done by a transformation triggered before serializing the generated XTRA. The transformation detects patterns where subq_quantified operator uses vector comparison. The transformation replaces such quantified subquery pattern with a semantically equivalent existential correlated subquery.

Figure 5 shows the new transformed XTRA, which can now be serialized into SQL that is compatible with the target database as given by the following query:

```
SELECT *
FROM SALES S1
WHERE
EXISTS
(
  SELECT 1
  FROM SALES_HISTORY S2
  WHERE (S1.AMOUNT  > S2.GROSS)
  OR (S1.AMOUNT = S2.GROSS
      AND S1.AMOUNT * .85> S2.NET)
);
```

In practice, the quantified subquery pattern in Example 2 could be embedded in far more complex queries. Hyper-Q is a principled framework that can handle more complex query rewriting cases via transformations.

4.3 Feature Augmentation

The features used by queries in client application may be completely missing in the target database. Sophisticated data warehouse features (e.g., stored procedures, recursive

```
+-select
  |-get (SALES 'S1')
  +-subq_exists
      +-select
          |-remap consts: (1)
          |  +-get (SALES_HISTORY 'S2')
          +-boolean (op:OR)
              |-comp (cmp:>)
              |  |-ident (S1.AMOUNT)
              |  +-ident (S2.GROSS)
              +-boolean (op:AND)
                  |-comp (cmp:=)
                  |  |-ident (S1.AMOUNT)
                  |  +-ident (S2.GROSS)
                  +-comp (cmp:>)
                      |-arith (op:*)
                      | |-ident (S1.AMOUNT)
                      | +-const (0.85)
                      +-ident (S2.NET)
```

Figure 5: Transformed XTRA for Example 2

queries and updatable views) that are missing in target databases are still supported in Hyper-Q by emulation. In order to support these features, streamlined query rewriting is not a viable option. Hyper-Q breaks down these sophisticated features into smaller units such that running these units in combination gives the application exactly the same behavior of the main feature. We describe how emulation of missing features is enabled by Hyper-Q using recursive queries as an example.

Example 3 Given an employee relation EMP(EMPNO,MGRNO), consider the following recursive query which computes all employees reporting either directly or indirectly to emp10:

```
WITH RECURSIVE REPORTS (EMPNO,MGRNO) AS
(
      SELECT EMPNO,MGRNO
      FROM EMP
      WHERE MGRNO = 10
   UNION ALL
      SELECT EMP.EMPNO,EMP.MGRNO
      FROM EMP,REPORTS
      WHERE REPORTS.EMPNO = EMP.MGRNO
)
SELECT EMPNO FROM REPORTS ORDER BY EMPNO;
```

Figure 6 shows hierarchical employee-manager sample data for the EMP relation. □

Recursive queries are introduced in the SQL standard as a way to handle queries over hierarchical model data. By introducing a common table expression that includes self references (e.g., REPORTS.EMPNO inside the definition of REPORTS), the database system is required to run query recursion until reaching a fixed point, where further recursion does not introduce any new results.

When query recursion is not natively supported by the target database, there is no direct translation that could be utilized to generate a single SQL request equivalent to a recursive query. However, a deeper analysis shows that the computation of recursive queries relies on simpler constructs that might be readily available in the target database. In

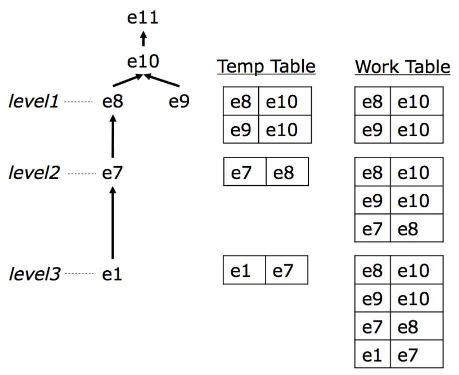


Figure 6: Emulating recursive query in Example 3

particular, using a sequence of temporary table operations, recursion can be fully emulated.

Hyper-Q emulates query recursion using two temporary tables: TempTable, which stores the results of current recursive call, and WorkTable, which stores the full results of previous recursive calls. At each recursive step, the contents of TempTable are appended to WorkTable. Recursion ends when TempTable contains no results.

Figure 6 shows the sequence of temporary table operation performed by Hyper-Q to emulate the execution of the previous recursive query:

1. Initialize both WorkTable and TempTable with the results of seed expression, which is the first UNION ALL input: {(e8, e10), (e9, e10)}.

2. Execute recursive expression by joining EMP with TempTable. Append the results {(e7, e8)} to WorkTable which becomes now {(e8, e10), (e9, e10), (e7, e8)}. TempTable= {(e7, e8)}. Recursion needs to continue.

3. Execute recursive expression by joining EMP with TempTable. Append the results {(e1, e7)} to WorkTable which becomes now {(e8, e10), (e9, e10), (e7, e8), (e1, e7)}. TempTable= {(e1, e7)}. Recursion needs to continue.

4. Execute recursive expression by joining EMP with TempTable. The result is empty and so recursion stops. Drop TempTable.

5. Substitute references of REPORTS with WorkTable in main query, and execute modified query SELECT EMPNO FROM WorkTable ORDER BY EMPNO;.

6. Return results of main query to client application. Drop WorkTable.

In the previous example, Hyper-Q generates multiple query requests, and maintaines the state of recursion by inspecting the results of these requests. The behavior from application stand point remains the same, where the results of the recursive query are obtained with exactly the same behavior of the original database system.

5. VIRTUALIZATION USE CASES

During our engagement with customers and prospects, we have identified several major approaches to moving to the cloud, which we detail next. Those are visually depicted in Figure 7.

5.1 Complete drop-in replace

In the past months we have seen multiple cases where enterprises across different verticals have undertaken campaigns to reduce their data center footprint and vendor licensing fees and move their operations to the cloud. Due to the difficulties outlined in Sections 1 and 2, most of those enterprises have been reluctant to pursue such a project as it often meant a costly multi-month and high-cost investment, which often exceeds the cost savings of the cloud, and poses high risks to the business. A small fraction of those customers have also tried looking positively at the challenge: a rewrite of the applications to enable portability across databases also means those applications can be modernized or otherwise optimized to keep up with the ever-changing technologies. However, even those optimistic customers have more often than not given up on the “application modernization” idea after they realize the magnitude of the migration efforts involved in simply porting the application without any change.

ADV enables instantaneous deployment of existing applications on the cloud, as shown in Figure 7 (a). Modernization of applications can happen gradually, if the enterprise wants to invest in this. What is more, developers now have the choice what query language they want to use for their new applications: if they feel more comfortable with the old query syntax, they can keep using that, or they can switch to the language of the new database, while keeping the guarantee that their application will not only work, but also be portable in the future.

5.2 Disaster recovery

In several cases we have encountered customers who are happy with their existing on-premise database and want to keep it, but want to add a cloud database to the mix for disaster recovery purposes, as shown in Figure 7 (b). The cloud database in those scenarios is often not fully utilized, thus incurring significantly lower costs compared to maintaining a second on-premise installation. This scenario poses significant challenges to the enterprise, as the same applications now need to run on two potentially different database installations. While multiple database vendors are now coming up with cloud offerings of their solutions, the latter are not always fully compatible with the on-premise distribution and many features may have been disabled for performance reasons. So the only remaining solution used to be maintaining two different versions of the application for the different database vendors. In addition to the pure migration challenges outlined before, this has additional drawbacks as any modification to the application (for example bug fixes or new feature development) now needs to be applied in two separate codebases. This obviously is highly undesirable.

ADV easily enables disaster recovery deployments across different stacks as it does not require the original application to be changed in order to have it execute on a different platform. Application evolution can happen naturally on the original code and does not need to be reimplemented for the secondary database.

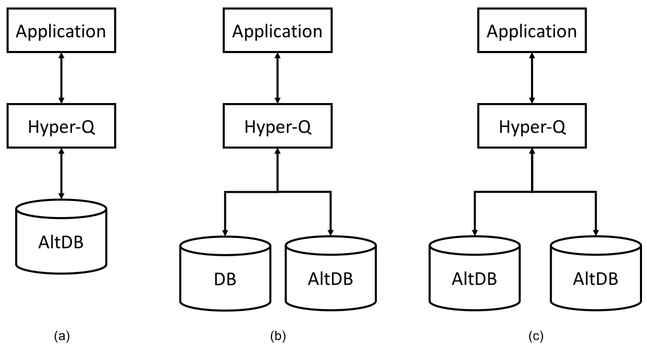


Figure 7: Database Virtualization Use Cases. (a) Drop-in Replace, (b) Disaster Recovery, (c) Scaling Out

5.3 Scaling out applications

When moving from an on-premise to a cloud-based data warehouse, customers do not want to experience performance regressions. In one case, a customer's throughput requirements could not be met even when they used the largest available instance provided by the cloud database.This could be partially attributed to the fact that the existing on-premise solution ran on dedicated hardware, and the customer's workload was highly optimized for that database.

A common solution for such case is to maintain multiple replicas of the data warehouse and load balance queries across them, as shown in Figure 7 (c). The ADV solution on top can then automatically route the queries to the different replicas, without sacrificing consistency, and without requiring changes to the application logic. We are currently working on extending Hyper-Q to handle this scenario.

6. EXPERIMENTS

In this section we provide performance evaluation of our framework for both synthetic and customer workloads, and show that Hyper-Q introduces negligible overhead to the execution of analytical queries in a cloud database.

6.1 TPC-H

For this experiment we ran the 22 queries of the TPC-H benchmark[1] on a 1TB of data stored in one of the leading cloud databases. The cloud database was provisioned as a 2 node cluster, where each node had 32 virtual cores, 244GiB of main memory and 2.56TB of SSD storage. We used Teradata's bteq client to submit queries to Hyper-Q, and we cleared the database cache before every execution.

For each query we measured the following times:

- Query translation time: Time spent by Hyper-Q to translate the query from the original SQL dialect to the SQL dialect used by the cloud database, including parsing, binding, backend-specific transformations and emitting the final query into the target language
- Execution time: Time taken by the cloud database to execute the query and return the results
- Result transformation time: Time taken by Hyper-Q to transform the results to Teradata's native format

[1]http://www.tpc.org/tpch/

Figure 9(a) shows that the total overhead introduced by Hyper-Q with respect to the end-to-end execution time is less than 2%, with around 0.5% for query translation, and around 1% for result transformation. The break down by query is depicted in Figure 8. While the time for query translation is negligible for all queries, there is query 11, which shows a relatively larger overhead for result transformation. This can be attributed to the fact that this query returns larger result sets. While most analytical queries typically return smaller aggregated results, we are currently working on improving our data processing pipeline, including building an ODBC bridge to avoid redundant marshalling of data types for the cases where the original application uses ODBC as well.

6.2 Stress Test

This experiment mimics a real-world application scenario for a Fortune 10 customer that used Hyper-Q to re-platform applications from Teradata to a leading cloud data warehouse. A customer application starts ten simultaneous sessions to connect to the underlying database. Each session continuously sends queries through Hyper-Q. The cloud data warehouse was configured to use the most powerful available specifications, and was manually tuned to achieve the highest possible performance for the workload. The test lasted for 17.5 hours, submitting a total of 967,506 queries capturing customer's peak load for generating analytical reports. The submitted queries had a rich set of features, including many of the features in Figure 2 such as QUALIFYclause, named expressions, implicit joins and vector subqueries.

To mimic that scenario, we used a similar setup to the one used in section 6.1. However, instead of using a single Teradata's bteq client, we used ten clients, each repeatedly sends TPC-H queries through Hyper-Q to execute on the cloud data warehouse. We let the experiment run for several hours and then collected the aggregated query translation time, execution time and result transformation time.

Figure 9(b) shows that in our stress test scenario, Hyper-Q overhead is a tiny fraction with respect to end-to-end execution time. In particular, the total overhead of Hyper-Q is between 0.1% and 0.2% of the total execution time. This is due to the fact that while the query execution time increases substantially with the level of query concurrency, Hyper-Q only introduces a small constant overhead per query.

7. RELATED WORK

The problem of database migration is as old as databases themselves. Naturally, a number of approaches, systems and strategies have been developed over the past decades.

The prevalent technique is manual migration often executed by third party consultants. Any manual migration is inherently non-scalable, i.e., the number of person-years required is proportionate to the number of applications and lines of SQL code.

7.1 Database Migration Utilities

A variety of proprietary tools have been developed in this space to aid in the otherwise manual process [3,5,8]. These tools address primarily the content transfer and as such address the lesser of the actual challenges. Several of these systems also provide general SQL rewrite capabilities as needed to rewrite certain schema features like views as mechanisms to rewrite application logic. They are advertised as aiding

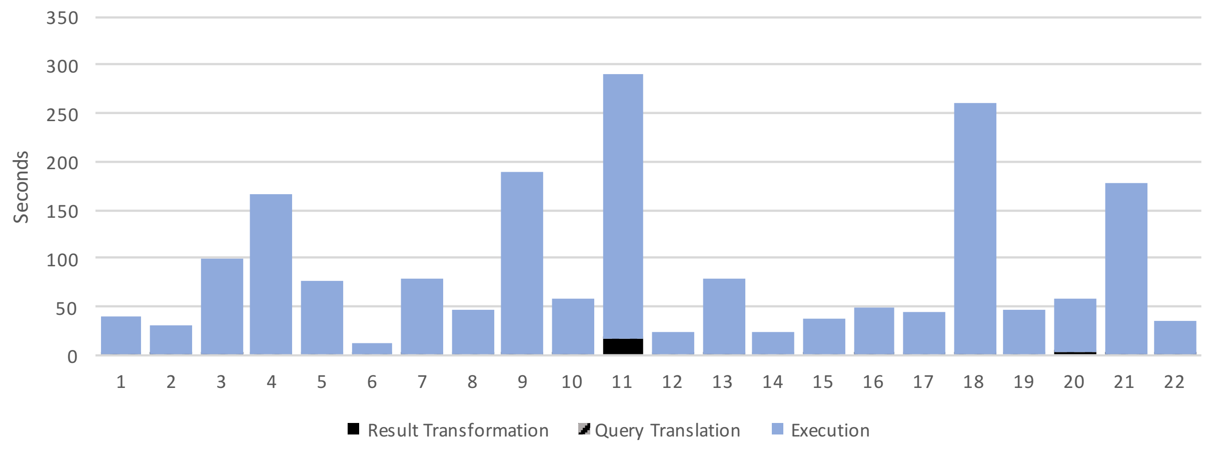


Figure 8: Break down of running time (in sec) for the queries of TPC-H by individual query

in manual migrations but do not claim to be actually full automations.

7.2 Remote Query Extensions

Instead of addressing the challenges of migrations, a class of database or database-like systems exists that effectively reduces the need for migrations by making other databases accessible through the original database. That way, a new database can be integrated as a subordinate database. This kind of technology has long existed in most commercial databases [9, 10] and most recently in Teradata Query Grid [11]. In all cases, the old database remains the main access point. New databases are only added but will not replace the old database. These should be viewed as a tactical move by database vendors to preserve their footprint. The main drawback of these systems is that they still require the old database to remain functional indefinitely.

7.3 Query Federation

Similar to the previous category, query federation systems address the problem of integrating new data sources and has seen a lot of interest in the research community since the mid 1980s [16, 17, 19, 20, 23], where the focus is on source selection and pushing computation as close to the source as possible. Commercial implementations are few and far between [4, 15, 21]. However, they cannot close the gap between applications and federation system in that a migration *to* the federation system is necessary. As a result, federation systems have yet to establish a credible footprint in the industry.

7.4 Database Virtualization

Adopting elements of a competitor's feature set has a long tradition in the database industry. Several commercial database systems feature compatibility modes to accommodate language features of competitors. Examples include EnterpriseDB which mimics Oracle [6], or ANTS, a now defunct company that enabled IBM DB2 to mimic Sybase [1]. These approaches are limited to language features only and while they mitigate some of the rewriting effort do not eliminate the problem at all. Most notably, the application still needs to be converted and use the new drivers and connectivity libraries. These approaches do generally not go far enough and still do not break the non-scalable nature of a manual migration.

[13] lays the foundation to our current approach. In this paper, we transfer some of the ideas presented there to cloud-native databases. In particular, we implemented a complete front-end emulation for Teradata, one of the most prevalent data warehousing systems across Fortune 500 companies.

8. SUMMARY

We presented, Datometry Hyper-Q a complete and production-ready implementation of ADV to address what has become one of the biggest obstacles in adoption of new data management technology: how to preserve longstanding investments in application development when moving to the cloud and replacing conventional databases with cloud-native systems.

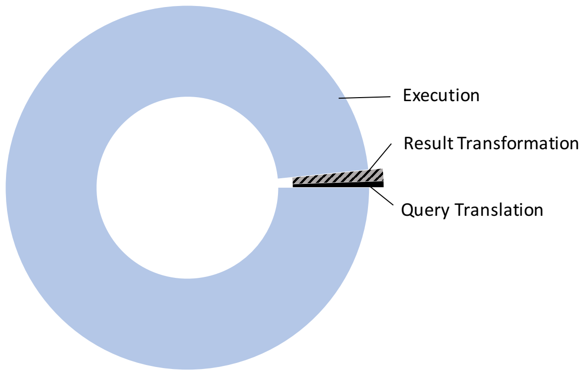
ADV is a holistic and powerful vision built on the key insight that intercepting the communication between application and database leads to a fundamentally different and much more efficient approach compared to the conventional re-platforming.

8.1 Discussion

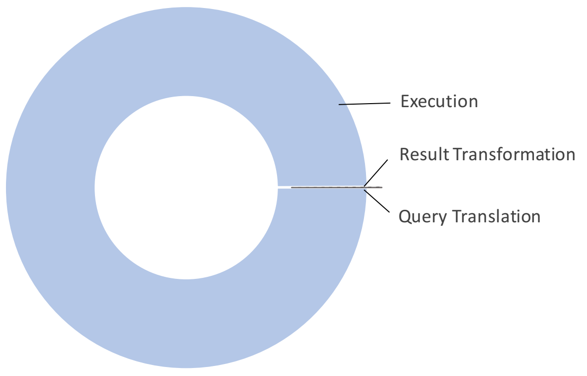
Using a select number of examples, we demonstrated the viability of this approach both from a functional as well as operational point of view. Due to space constraints, we could survey only a small fraction of the functionality implemented in Datometry Hyper-Q. The system is in use at a significant number of Fortune 500 companies already.

It should be pointed out that the interplay between Hyper-Q and the replacement database system is critical for the success. By selecting a specific replacement system, customers make a deliberate choice that not only affects them from a customer relationship perspective but needs to take functional, performance, and scalability requirements into account. Hyper-Q is a highly effective selection tool that lets customers trial the new database system at virtually no cost and risk.

While situations are conceivable where the feature gap between systems is so wide that closing it using ADV is no



(a) Aggregated elapsed time for single sequential run



(b) Aggregated elapsed time for concurrent stress test

Figure 9: Overhead of Hyper-Q below 2% and 0.3% of total query response time respectively.

longer cost-effective, we have not experienced this in practice to date.

8.2 Market Dynamics

The feasibility and realization of an ADV concept in the form of Datometry Hyper-Q has seen enormous acceleration through cloud-shift [2], the ubiquitous industry trend of consolidating data management into public or private clouds. Within the next years, this trend is expected to accelerate even further transforming the database market, one of the most coveted and established pillars of the overall IT market completely.

We have chosen data warehousing as a starting point and first candidate based on market demand and availability of cloud-based database systems with appropriate capabilities. Going forward we plan to broaden the scope to include also transactional systems as we see customers increasingly inquire about these systems already.

8.3 Future Work

Supported by the overwhelmingly positive reception by customers and prospects, we believe ADV has much larger application. Specifically, ADV is not only means to facilitate re-platforming but will become an integral part of the IT stack. Becoming a universal connector between applications and databases can break open data silos, make data freely usable across the enterprise, and accomplish all this without the need to restructure or rewrite applications. With a broad agenda to add functionality, support for additional and emerging technologies ADV should be seen as a *de facto* insurance policy for the enterprise and its data strategy.

9. REFERENCES

[1] (2008) Emulate Sybase in DB2 using ANTs Software. [Online]. Available: https://ibm.co/2eMvpUL
[2] (2016) By 2020 Cloud Shift Will Affect More Than 1 Trillion in IT Spending, Press Release. Gartner. [Online]. Available: http://www.gartner.com/newsroom/id/3384720
[3] (2017) AWS Database Migration Service. [Online]. Available: https://aws.amazon.com/dms/
[4] (2017) Data Federation in Composite Software. [Online]. Available: http://bit.ly/2eMF79V
[5] (2017) DBBest Technologies. [Online]. Available: https://www.dbbest.com/
[6] (2017) EnterpriseDB. [Online]. Available: https://www.enterprisedb.com/
[7] (2017) Google BigQuery. [Online]. Available: https://cloud.google.com/bigquery/
[8] (2017) Inspirer. [Online]. Available: http://www.ispirer.com/
[9] (2017) Linked Servers in Microsoft SQL Server. [Online]. Available: http://bit.ly/2xDy9vJ
[10] (2017) Oracle Distributed Queries. [Online]. Available: http://bit.ly/2vS57pM
[11] (2017) Teradata QueryGrid. [Online]. Available: http://bit.ly/2wuwuZ4
[12] (2017) XtremeData. [Online]. Available: https://www.xtremedata.com/
[13] L. Antova *et al.*, "Datometry Hyper-Q: Bridging the Gap Between Real-Time and Historical Analytics," in *SIGMOD*, 2016.
[14] B. Dageville *et al.*, "The Snowflake Elastic Data Warehouse," in *SIGMOD*, 2016.
[15] Denodo. [Online]. Available: http://www.denodo.com/
[16] A. Deshpande and J. M. Hellerstein, "Decoupled query optimization for federated database systems," in *ICDE*, 2002.
[17] A. J. Elmore *et al.*, "A Demonstration of the BigDAWG Polystore System," *PVLDB*, 2015.
[18] A. Gupta *et al.*, "Amazon Redshift and the Case for Simpler Data Warehouses," in *SIGMOD*, 2015.
[19] D. Heimbigner and D. McLeod, "A Federated Architecture for Information Management," *ACM Transactions on Information Systems*, vol. 3, pp. 253–278, 1985.
[20] S. Hwang *et al.*, "The MYRIAD federated database prototype," in *SIGMOD*, 1994.
[21] H. Kache *et al.*, "POP/FED: progressive query optimization for federated queries in DB2," in *VLDB*, 2006.
[22] S. Shankar *et al.*, "Query Optimization in Microsoft SQL server PDW," in *SIGMOD*, 2012.
[23] A. Simitsis *et al.*, "Optimizing analytic data flows for multiple execution engines," in *SIGMOD*, 2012.