Datometry Hyper-Q: Bridging the Gap Between Real-Time and Historical Analytics

Lyublena Antova, Rhonda Baldwin, Derrick Bryant, Tuan Cao, Michael Duller,
John Eshleman, Zhongxian Gu, Entong Shen, Mohamed A. Soliman, F. Michael Waas
Datometry Inc.
795 Folsom St #100, San Francisco, CA 94107, U.S.A.
www.datometry.com
⟨first.last⟩@datometry.com

ABSTRACT

Wall Street's trading engines are complex database applications written for time series databases like kdb+ that uses the query language Q to perform real-time analysis. Extending the models to include other data sources, e.g., historic data, is critical for backtesting and compliance. However, Q applications cannot run directly on SQL databases. Therefore, financial institutions face the dilemma of either maintaining two separate application stacks, one written in Q and the other in SQL, which means increased IT cost and increased risk, or migrating all Q applications to SQL, which results in losing the inherent competitive advantage on Q real-time processing. Neither solution is desirable as both alternatives are costly, disruptive, and suboptimal.

In this paper we present Hyper-Q, a data virtualization platform that overcomes the chasm. Hyper-Q enables Q applications to run natively on PostgreSQL-compatible databases by translating queries and results on the fly. We outline the basic concepts, detail specific difficulties, and demonstrate the viability of the approach with a case study.

1. INTRODUCTION

The financial services industry has long been a pioneer in data processing. Real-time databases and special purpose query languages have fundamentally transformed trading desks in the past twenty years. Today's trading engines are elaborate database applications that implement complex mathematical models in the form of queries. The advent of Big Data technology over the past years presents a significant opportunity not only to integrate new data sources but also to extend existing models to incorporate substantial amounts of historical data.

Unfortunately, the mismatch between data models and query languages makes this kind of data integration a difficult undertaking and institutions so far faced the choice of either (1) committing to one specific solution as a compromise and migrate their entire application ecosystem to that technology or (2) maintaining multiple copies of the application stack, one for each technology, e.g., one for in-memory, one for SQL databases, one for archival NoSQL systems. Either choice is associated with high implementation cost. Either choice may also result in suboptimal operational performance or, in the case of multiple stacks, high maintenance cost of an inherently error-prone process. The underlying problem has been referred to as the Holy Grail of historical data processing [5] by practitioners in this space: a unified solution will have significant financial implications as it allows trading applications to take advantage of additional data and thus outperform competitors, but also avoids the penalty of high maintenance cost.

In this paper we present a unique solution that avoids the aforementioned compromise but rather keeps the application layer intact while enabling the use of real-time and archival database systems side by side. Building on the basic concept of Adaptive Data Virtualization (ADV) [1], we present Hyper-Q, an actual implementation of a platform that abstracts the backend database and translates queries and data exchange on the fly. The use-case we present in this paper shows how applications originally written for real-time systems using the programming language Q can seamlessly access SQL databases, as Hyper-Q provides a full-featured Q interface on top of an MPP database or a NoSQL system.

During the development of Hyper-Q we had to master a number of hard challenges including the mapping of different data types, reconciliation of different semantics around operators and functions, as well as operational challenges like emulation of management or authentication mechanisms.

Note that at this stage of our project we address the challenge of *accessing* data from native applications using emulated APIs. We rely on the assumption that all relevant data is loaded into the underlying systems independently. In practice, we found transferring the data to be much less of a problem. We consider adding tools that perform data movement and the mapping of schemas in the future; we expect that development to be greatly simplified by Hyper-Q's capabilities.

The remainder of this paper is organized as follows: In Section 2 we provide background on the technologies we leverage in our work. Section 3 gives an architectural deep-dive of Hyper-Q. Section 4 is a discussion of implementation details. We present a case-study on an actual deployment of Hyper-Q in Section 5. Our experimental evaluation is in Section 6. We discuss related work in Section 7. Finally, Section 8 summarizes the paper with concluding remarks.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA
© 2016 ACM. ISBN 978-1-4503-3531-7/16/06…$15.00
DOI: http://dx.doi.org/10.1145/2882903.2903739

2. BACKGROUND

In this section, we briefly survey background needed for the developments in the rest of this paper. First, in Section 2.1, we present the approaches adopted by the industry for supporting real-time and historical analytics. Then, we give an overview in Section 2.2 on the kdb+ system and its Q language, which provide the main motivating application of our framework.

2.1 Real-time and Historical Analytics

In this section, we give background on the building blocks and problems associated with integrating real-time and historical data analytics in financial services.

Financial services utilize specialized environments and proprietary algorithms to perform stock trading at large scale. Stock trading is a highly competitive business. Entities that have the fastest access to information and the most sophisticated trading algorithms are at an advantage.

Market data is the time-series data produced by an entity like a stock exchange. For specification and samples, we refer the reader to the New York Stock Exchange Trades and Quotes (NYSE's TAQ) dataset [17]. Market data is used in real-time to make time-sensitive decisions about trading equities and is also used for historical data applications like predictive modeling, e.g., projecting pricing trends, calculating market risk and backtesting of trading strategies.

Real-time databases have held a strong presence in capital markets for two main reasons; low-latency query processing and query language expressiveness. Query response times must be sub-second and systems achieve this by utilizing columnar data organization, in-memory query processing, and avoiding the typical RDBMs bottlenecks such as locking and synchronization overheads. Expressiveness and terseness of the query language to perform time-series analytics was also a major advantage over early RDBMs systems.

Recently, retention and analysis of data beyond short horizons have become increasingly important and a competitive differentiator within the financial services industry. Organizations are confronted with finding a way of continuing to do the analysis their business run on today within the tight SLAs as well as extend that analysis to include larger time windows of data.

The approaches that are typically adopted to run both real-time and historical analytics in financial services are the following: (1) scaling real-time databases, (2) migration to a scale-out analytical database, or (3) using a hybrid of real-time and analytical databases. We highlight the main limitations of these different approaches.

Scaling Real-time Databases. Adding more real-time database servers could allow analytical financial services to scale to some extent. However, this approach can be prohibitively expensive because of multiple reasons:

1. Cost: The hardware costs associated with expanding real-time systems can easily exceed the cost of conventional RDBMS technology on commodity hardware.

2. Compute Power: Most real-time databases are single machine and sometimes even single CPU. The slowdown of Moore's law against ever increasing data volumes means that the compute power in these systems could not catch up with workloads as data volumes increase.

3. Tools: Real-time database systems are built for a very different use case than analytics on historical data. They often lack the compliance, management utilities, and tools required by enterprises when storing massive volumes of historical data, which RDBMS vendors have taken 20-30 years to perfect.

Migration to Analytical Databases. Analytical databases, such as Massively Parallel Processing (MPP) databases, are effective at scaling to very large data sizes. Migrating real-time database applications/workloads to use an analytical database can address the historical data challenge.

However, latency overhead in analytical databases, especially for short-running queries, is typically larger than the traditional RDBMs overhead. This is mainly attributed to the large system scale and the need to optimize and dispatch query fragments over a cluster of machines. Consequently, analytical databases cannot typically meet the real-time SLAs that financial applications needs. Furthermore, migrating applications and workloads that are fine-tuned over years from real-time database to analytical database is an overwhelming undertaking. This process is known to be error-prone and has no clear quality guarantees. In some settings, database queries could be automatically generated by libraries, packages, web services, etc. This makes changing the query language very difficult.

A Hybrid of Real-Time and Analytical Databases. Building a hybrid system that deploys both real-time and historical database comes with its own problems. Most organizations find it straightforward to handle the data schema and loading of data between the two systems. Beyond that they quickly run into many challenges that must be overcome:

1. Training costs and challenge of finding qualified personnel who are adept at writing financial analytic algorithms in both the real-time database language, typically vector oriented, as well as in the historical database language, typically set oriented.

2. Once analytics are written for both systems, how does an organization vet and prove that they give the same or compatible results in both places. What configuration management practices must they employ to ensure that they stay compatible over time as minor modifications must be made?

3. Having to write analytics in both places slows down the pace of innovation and ability for the organization to continue to innovate and remain competitive.

An alternative solution would leverage investments in existing applications by keeping the real-time database applications, while allowing a choice of runtime execution environment. In this scenario, the application layer is kept intact, while the underlying data processing layer is virtualized such that the same query language can be used to query both real-time and analytical databases. This paper advocates for the viability and value of this novel solution.

2.2 kdb+

This section gives a high-level overview of the kdb+ system and is primarily meant to highlight the challenges involved in translating queries written in the query language Q into

a relational language, such as SQL. A more complete documentation can be found at [14].

kdb+ is a columnar database specifically designed for real-time analytics. Its premier application area is in-memory query processing although it can operate over on-disk data. The system lacks many of the features found in classical relational database systems such as ACID transactions. Like other special purpose database systems, kdb+ accomplishes isolation through serialization, i.e., the main server loop executes a single request at a time. Concurrent requests are queued to be executed serially. Atomicity, consistency and durability are the application's responsibility, if desired. For historical reasons, kdb+ had no need for access control or sophisticated security mechanisms. Similarly, kdb+ does not provide built in support for data replication. Disaster recovery or high-availability are accomplished through external tooling.

kdb+ is queried using Q, a highly domain-specific query language. Q is distinguished by its terse syntax. It is rich in idioms that are specifically tailored to the financial analytics use case. The highly compressed syntax is often lauded for eliminating room for error: a single line of Q may correspond to an entire page of SQL, the correctness of which is usually much harder to ascertain.

Q pre-dates most of the OLAP extensions found in recent SQL standard which initially gave it a unique competitive advantage over relational database languages. More recently, the SQL standard has caught up and provides sufficient language constructs to implement equivalent functionality. Unlike relational databases, Q is not based on a tabular calculus. Rather, Q is a list processing language that supports, besides scalar data types, several compound types such as dictionaries or tables, which are made up of lists. Lists are, by definition, ordered, which in turn greatly facilitates time series analysis in Q.

As a recent addition to the language, Q features several SQL-like constructs even though their semantics often diverge from relational dialects in surprising ways. For example, UPDATE operation in Q simply replaces columns in the query output instead of changing any persisted state.

To illustrate the expressiveness of Q consider the following example:

Example 1 A standard point-in-time query to get the prevailing quote as of each trade:

```
aj[`Symbol`Time;
   select Price from trades
   where Date=SOMEDATE, Symbol in SYMLIST;
   select Symbol, Time, Bid, Ask from quotes
   where Date=SOMEDATE]
```

This query is one of the most commonly used queries by financial market analysts [9]. It can be used to measure the difference between the price at the time users decide to buy and the price paid at actual execution, i.e. the fill message reported by the broker. The *as-of-join* (aj) is a unique built-in function in Q which natively supports time-series queries. In this query, *Symbol* and *Time* are the columns to join on, *trades* is a reference table and *quotes* is a table to be looked up.

The performance of this query largely depends on the size of the *quotes* table. If the data is small enough so that the underlying database has one partition per date, this *as-of-join* achieves very good performance. Instead of reading the entire Symbol, Time, Bid, and Ask columns into memory to perform the lookup, it can search through the memory map of the *quotes* table. However, if the *quotes* table is large, and there are multiple partitions per date, all partitions need to be read into memory to perform the lookup since rows with the same symbol could occur in multiple partitions. To work around this, Q programmers often need to manually rewrite the above query to do *as-of-join* on each partition and then aggregate the result from each partition. The rewrite is very complex and requires deep knowledge of the structure of the underlying database [9].

kdb+ does not have a query optimizer. A query is executed in reverse order of its components. Q programmers have to determine join orders explicitly in their queries and have to be aware of how individual operations such as a lookup-join are executed. Mistakes in crafting the query may have severe consequence for the performance of a query or even crash the server process due to out-of-memory conditions.

In many ways, performance is achieved by exposing to users how data is actually stored so that users can take advantages of this while constructing queries. For example, Q allows marking a list as having all occurrences of a value occurring in a sequential block. This allows the creation of a lookup table from each distinct value to its first occurrence, then all occurrences of that value can be found in one sequential read. As a consequence, Q programmers often need to understand how underlying data is structured in order to write optimal queries [7].

Due to considerable differences in query language and data model, building a virtualized system allowing Q application to run on top of a SQL database involves multiple challenges:

- Q applications communicate with kdb+ using specific wire protocol which is usually very different from the wire protocol of the underlying SQL database. While SQL databases typically implement protocols that stream individual rows, Q uses an object-based protocol that communicates a query results as a single message. In order to run Q applications seamlessly on an SQL database, we need to convert the packets in Q wire format to the underlying database wire format and vice versa. This conversion includes the transformation of data types and values as well as the pivoting of database rows into the object-based format that is to be presented to the application.

- While SQL is based on set/bag semantics, in which order of rows in a table is not defined, Q is based on *ordered-list* semantics, in which ordering is the first class citizen for all complex data structures, such as tables and dictionaries. In particular, each Q table has an implicit order column. Providing implicit ordering using SQL requires database schema changes and imposes challenges on query generation.

- Q is *column-oriented*, i.e., it stores tables as columns and can apply operations to entire column. In contrast to Q, SQL is not column-oriented. Even in columnar SQL databases, the support is mainly implemented in the underlying storage and execution engines. Therefore, query constructs that express column-wise operations cannot be easily specified in SQL.

- Q uses a *two-valued logic* in contrast to SQL's *three-valued logic*. Operations on null values have very dif-

ferent semantics in Q and in SQL. For example, two nulls compare as equal in Q, while the result is undefined/unknown in SQL. Imposing these semantics on SQL queries requires careful composition of query constructs to maintain correctness of the results.

- Unlike SQL which is statically typed, Q is dynamically typed where the type of a variable is determined by its value. Consider an expression x+y, if x and y are not statically defined then their types are determined by the values assigned to them at runtime. In particular, if x and y are assigned scalar values then x and y have scalar type; if x and y are assigned as lists then x and y have list type. Translating a dynamic-typed language to a static-typed language requires significant amount of runtime support. If not done efficiently, type inference may add a considerable overhead to the query latency.

- Q expressions are evaluated strictly *right-to-left* with no operator precedence, reflecting the implementation of the underlying execution engine. This is considerably different from expression evaluation using SQL.

Q programmers are often unwilling to switch to SQL and its verboseness. A single line of Q code may be semantically equivalent to a large snippet of SQL. It can be challenging to reason about the correctness of such transformation by eyeballing. Manual migration of complex applications quickly becomes infeasible. Additional challenges arise from operational aspects such as using advanced authentication mechanisms (e.g., Kerberos).

3. HYPER-Q PLATFORM

Hyper-Q bridges the gap between real time and historical data analytics by virtualizing the data processing layer. Using Hyper-Q, applications and workloads written in the Q language can run unchanged while using a PostgreSQL (PG) compatible database for running data analytics.

Q to SQL translation is only one of the problems that need to be addressed to enable such communication. Other problems include (i) network communication, where queries and results need to be parsed, transformed and packaged according to the wire protocols of the two end systems, and (ii) state management, where a variable may be defined and reused across multiple queries.

The choice of translating Q queries into PG-compatible SQL was driven by customer requirements. Many analytical database systems are based on some version of PG. Examples include Greenplum [20], HAWQ [3], Vertica [2], Aster Data [24] and Redshift [8]. Many of these systems maintain, to a large extent, PG-compatible query language, query clients and communication protocols. This makes supporting PG query language (which closely follows SQL standard) and PG communication protocol a plausible path to enable many customers to run Q applications through Hyper-Q.

Figure 1 shows a high-level architectural diagram of the Hyper-Q platform illustrating the interaction between a Q application and a PG database.

Query Life Cycle. A connection request is sent from a Q application to Hyper-Q. The connection message is encoded according to the **Q**-**I**nter **P**rocess **C**ommunication (QIPC) wire protocol. Once authenticated, the Q application uses

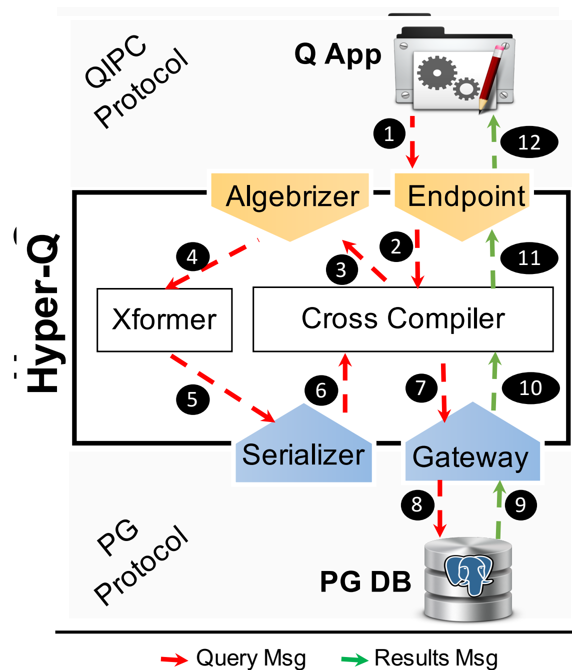

Figure 1: The Hyper-Q platform

the Hyper-Q connection to send Q query messages to Hyper-Q (Section 3.1). The Algebrizer component parses the incoming Q query and transforms it into an extensible relational algebra expression (Section 3.2). The Xformer component then modifies the algebraic expression by applying a series of transformations to guarantee correct query semantics and optimize query performance (Section 3.3). The end algebraic expression is then translated into one or more SQL query messages encoded using the PG v3 wire protocol. The SQL queries are sent to PG database for processing. Query results are translated back into the QIPC data format before getting sent to the Q application. These operations are managed by the Cross Compiler (XC) component (Section 3.4). System-specific plugins are used for handling network communication, parsing messages exchanged between the Q application and the database, as well as generating query messages.

Hyper-Q virtualizes access to different databases by adopting a plugin-based architecture and using version-aware system components. For example, the Algebrizer component triggers parsing rules based on the type and version of the database system that client application is designed to work with. Similarly, the Xformer component triggers transformations based on the type and version of the backend database system. This flexibility allows Hyper-Q to support Q applications that were designed to run on top of specific versions of the kdb+ system, as well as PG-based database systems that have deviated in functionality or semantics from the core PG database. Non-PG database systems can be supported by adding a plugin to the platform that enables query and result transformation as well as network protocol support for the desired database system. In the next sections we describe the components of the Hyper-Q platform that enable Q clients to work with a PG-compatible databases.

3.1 Network Communication

kdb+ uses TCP/IP for inter-process communication. The QIPC wire protocol describes message format, process handshake, and data compression. Messages can be of different types including connection open/close and synchronous/asynchronous calls. PG uses its own message-based protocol for communication between applications and servers. The PG wire protocol is supported over TCP/IP as well as Unix domain sockets.

The two previous protocols are widely different in terms of message format and process handshakes. The network packets transmitted from Q applications cannot be directly parsed by PG servers. To enable such communication, Hyper-Q acts as a bridge between the two protocols. Hyper-Q takes over kdb+ server by listening to incoming messages on the port used by the original kdb+ server. Q applications run unchanged while, under the hood, their network packets are routed to Hyper-Q instead of kdb+.

An incoming message to Hyper-Q includes a Q query. The Endpoint component in Figure 1 is a kdb+-specific plugin we implemented for handling the communication between Q application and Hyper-Q. The Endpoint component parses the incoming message, extracts the query text and passes it to the Algebrizer component (cf. Section 3.2) for subsequent processing.

Hyper-Q transforms incoming Q queries into semantically equivalent SQL queries, compatible with the backend PG database. The Gateway component in Figure 1 is a PG-specific plugin we implemented for handling the communication between Hyper-Q and PG database.

The Gateway component packs a SQL query into a PG formatted message and transmits it to PG database for processing. After query execution is done, the query results are transmitted back from PG server to Hyper-Q. Hyper-Q extracts the row sets from result messages and packs them into messages with the same format that a Q application expects (i.e., using the QIPC protocol). The formatted messages are sent to the Endpoint component, which in turn forwards the results back to the Q application.

Some of the previous operations could be performed using a database driver (e.g., ODBC/JDBC driver). However, integrating a third party driver in our data pipeline adds further complexity and comes with performance overhead. Processing network traffic natively is key for high throughput in Hyper-Q.

3.2 Algebrizer

The Algebrizer component translates a Q query into an eXTended Relational Algebra (XTRA) expression. XTRA is the internal query representation in Hyper-Q. It uses a general and extensible algebra to capture the semantics of query constructs, and make the generation of SQL queries a systematic and principled operation.

The Algebrizer operates in two steps. In the first step, the Q query text is parsed into a Q abstract syntax tree AST. In the second step, the AST is bound to an XTRA tree by resolving all variable references through metadata lookup and translating Q operators to semantically equivalent XTRA tree nodes. We describe the parsing and the binding steps using the following example:

> **Example 2** Consider the following Q query, which computes the so called *as-of-join* (`aj`) of two tables `trades` and `quotes`. For each record in `trades`, `aj` returns a matching value of `Symbol` column in `quotes`. A match occurs based on an implicit range condition on the `Time` column. If no match is found, a null value is returned.
>
> ```
> aj[`Symbol`Time; trades; quotes]
> ```

The algebrization result of Example 2 is shown in Figure 2. We discuss how algebrization is done in the next sections.

3.2.1 Parsing

The parser converts Q query text to an AST. In contrast to traditional relational query languages like SQL, the data model of Q allows for stand-alone table, scalar, list, or dictionary queries to be expressed in the query language. Variables are dynamically typed based on the values they are bound to. Moreover, the query syntax does not restrict the type of the result. To illustrate, consider the following Q language examples:

```
x:  1
x:  1 2 3
x:  select from trades
```

The first statement assigns a scalar value 1 to variable `x`. The second statement redefines `x` to be a list, while the third statement redefines `x` again to be a table expression. The type of `x` depends on the value it is bound to. Global variables are stored in kdb+ server's memory, and they can be redefined and used from different Q query clients.

The Q query `x+y` could be interpreted as arithmetic addition of two scalars or a pairwise addition of list elements. It could also raise an error if `x` and `y` are two lists of different length.

The previous query semantics are different from SQL, where a query clause restricts the type of expressions that can appear in some context. For example, the SQL `FROM` clause restricts the following expression to be a table expression.

Dynamic typing in Q can yield a complicated parser design since we need to inspect a large number of possible parse trees. In Hyper-Q, determining a variable type may require a round trip to the PG database for metadata lookup. For example, when a Q query refers a variable `x`, the parser may need to know if `x` is a table in the PG database. Due to these considerations, we have decided to design a lightweight parser in Hyper-Q whose only role is to create an abstract representation of the query in memory, and delegate the type inference and verification to the binder component (cf. Section 3.2.2).

The parser constructs an AST, consisting of the following main elements:

- literals: constant expressions such as integers (e.g., `1i`) and symbols (e.g., `` `GOOG``).
- variables: expressions that reference a named entity (e.g., `trades`).
- monadic and dyadic operators: operations on one or two arguments, respectively.
- join operators: different types of Q joins such as the `aj` operator in Example 2.

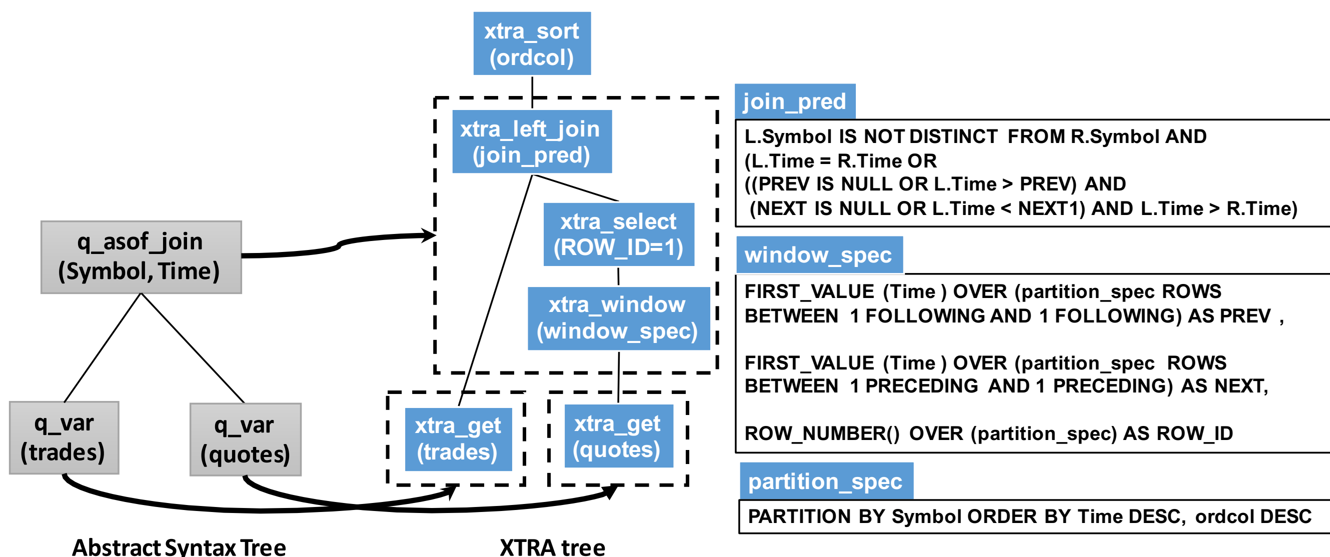

Figure 2: Algebrization of query in Example 2

- variable assignments: expressions of the form `var:expression`.

The AST for Example 2 is shown in Figure 2. The parser does not attempt to decide on the types of the variable references `trades`, and `quotes` since in the general case, they may be expressions of any type.

3.2.2 Binding

After parsing is done, the binder performs semantic analysis of the resulting AST and binds it to XTRA, the internal query representation in Hyper-Q. Binding Q queries into XTRA expressions is based on the fact that, although Q and SQL data models are different, the vast majority of Q operators can be mapped to corresponding (but sometimes more complicated) relational algebra expressions. Hyper-Q provides an extensible framework to build and compose such mapping rules to achieve the largest possible coverage of the Q language. The framework also allows using more sophisticated methods, such as UDFs in SQL databases, to capture the Q language constructs that cannot be directly mapped to relational algebra expressions.

Binding is a recursive operation that is done in a bottom-up fashion, where for each Q operator in the AST, the binder processes the operator's inputs, derives and checks input properties, and then maps the operator to its corresponding XTRA representation in the following way:

- Literals get bound to scalar const operators `xtra_const`, where the Q type of the literal is mapped to Hyper-Q's type system. For example, int types get mapped to equivalent integer types, symbol type gets mapped to varchar, whereas string literals get mapped to text constants.

- Variable references are resolved by looking up their definition through the metadata interface. Table variable references get bound to a relational get operator `xtra_get`. For example, in Figure 2, `q_var(trades)` is bound to `xtra_get(trades)`.

- When binding each operator, operator's properties are derived in the resulting XTRA tree. For relational operators, derived properties include the output columns with their names and types, keys, and order. For scalar operators, derived properties include the output type and whether the expression has side effects.

- After binding the inputs of an operator, the binder first checks whether the inputs are valid for the given operator by accessing their properties derived at the previous step. For example, the `aj` operator expects its two inputs to be table expressions and the right input to be a keyed table. Also, the join columns must be included in the output columns of input operators. If property checking succeeds, the operator is bound to its XTRA representation. In Figure 2, the `aj` operator is bound to a left outer join operator that computes a window function on its right input. The results need to be ordered at the end to conform with Q ordered lists model.

3.2.3 Metadata lookup

The binder resolves variable references by looking up associated metadata in the metadata store. In the basic case, where Q variables are mapped to PG tables, this corresponds to executing a query against PG catalog to retrieve various properties of the searched object. For tables, the retrieved metadata include columns, keys and sort order, while for functions, the retrieved metadata include function arguments and return type. Q also allows the definition of in-memory variables. A computation result can be stored to a variable that gets referenced in subsequent queries. Consider the following example:

Example 3 Consider the following Q function, which returns max price of `trades` matching a given symbol `Sym`:

```
f:
{ [Sym]
  dt:  select Price from trades where Symbol=Sym;
```

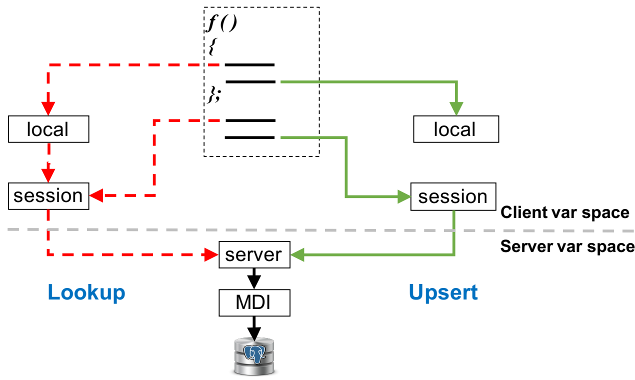

Figure 3: Hierarchy of variable scopes in Hyper-Q.

```
   :select max Price from dt;
};
f[`GOOG];
```

Example 3 defines a function `f`, which assigns a computed table to variable `dt`, while applying a filter to `Symbol` column using the function argument `Sym`. The function returns the maximum `Price` in the computed table `dt`. The last statement in Example 3 calls `f` while passing the symbol (`` `GOOG ``) as an argument.

Q distinguishes between two main types of variables: (i) local variables defined in function bodies like `dt` in Example 3, and (ii) global (server) variables like the function `f` itself. Local variables are only visible in the scope where they are defined, whereas global variables are visible to all Q query clients connected to the same kdb+ server.

Local variables shadow global variables with the same name. In Example 3, after the program is executed, the function `f` becomes accessible by any client connected to the server. If `f` is invoked later in the same session, there is no guarantee that the function definition would still be the same, since it may have been overwritten in the meantime by another query client.

Hyper-Q needs to maintain the aforementioned behavior for Q applications. The backend PG database is used to store and materialize global server variables in publicly accessible schemas. Shadowing of global variables by local variables with the same names is implemented using a hierarchy of variable scopes, as depicted in Figure 3. The hierarchy has three variable scopes:

1. Local scope stores local function variables.

2. Session scope stores variables defined within session.

3. Server scope stores global variables.

Figure 3 shows a query session with a function `f` and two statements outside `f`. The first statement in `f` looks up a variable. The lookup has to be performed in the *local* scope first. If the variable is not locally defined, the lookup operation follows the scopes hierarchy. The first statement outside `f` also looks up a variable. In this case, lookup is directed to the *session* scope, since we are now outside the function `f`. The bottom-most scope corresponds to retrieving variable definition, such as tables and functions, through PG MetaData Interface (MDI).

Figure 3 also shows how variable upsert (definition/redefinition) operation takes place. The second statement in `f` upserts a variable (e.g., through variable assignment). This upsert call can only be executed in the *local* scope since, according to Q semantics, local upsert calls never get promoted to higher scopes. The second statement outside `f` is making another upsert call. In this case, the call is directed to *session* scope. Session variables are promoted to global (server) variables after upsert call is processed. This is done as part of the session scope destruction.

3.3 Xformer

The Xformer component is responsible for applying transformations to the XTRA expression before serializing it into a SQL query. Transformations are used in Hyper-Q for three main purposes:

- **Correctness.** Data models and query languages in kdb+ and PG systems are widely different with implicit assumptions on each side. For example, null values in Q assume 2-valued logic, while in SQL, null values assume 3-valued logic. To bridge this gap in semantics between the two languages, a transformation is used to replace strict equalities in XTRA expressions with `Is Not Distinct From` predicate, which provides the needed 2-valued logic for null values when serializing the outgoing SQL query.

- **Performance.** The XTRA expression holds relational and scalar properties that are used to optimize the serialized SQL. For example, each node in the XTRA tree is annotated with all columns it can produce. The requested columns at each node may be however a small subset of the available columns. A transformation that prunes the columns of each XTRA node, to keep only the needed columns, is used to avoid bloating the serialized SQL with unnecessary columns, which may negatively impact query performance.

- **Transparency.** Ordering is a first-class citizen in Q. However, this is not the case in SQL. To maintain Q ordering semantics in the serialized SQL query, ordering criteria may need to be automatically added to the SQL query constructs generated by Hyper-Q. This operation is implemented via a transformation. Each relational XTRA operator can declare an implicit order column and an order preservation property that indicates whether the XTRA operator can preserve the order in its output or not. This property is used by the Xformer to ignore ordering in some cases. For example, consider a nested query in which the outer query performs a scalar aggregation on the result of the inner query. In this case, the Xformer can remove the ordering requirement on the inner query. The Xformer may also generate implicit order columns by injecting window functions in the transformed expressions.

3.4 Cross Compiler

The XC component is responsible for driving the translation of incoming queries written in the Q language into PG SQL, as well as the reverse translation of query results produced by PG database into the result format expected

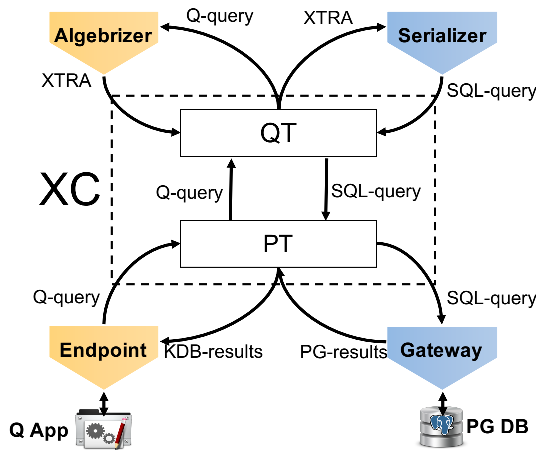

Figure 4: XC architecture

by Q application. Figure 4 gives an overview of XC architecture:

- **Protocol Translator (PT).** This layer is the DB protocol handler of Hyper-Q. PT is responsible for cross translation of messages sent to and received from the two end-point systems (Q application and PG database). Parsing DB protocol messages to extract queries as well as creating messages that hold queries and query results are handled by this layer.

- **Query Translator (QT).** This layer is the query language handler of Hyper-Q. QT is responsible for driving the translation of Q queries into XTRA, the internal query representation of Hyper-Q, serializing XTRA expressions into equivalent SQL statements, and communicating the generated SQL statements to PT to be sent to the PG database for execution. QT communicates with the Algebrizer and Serializer components to perform these tasks.

The interface between PT and QT is as simple as sending out a Q query from PT, and receiving back an equivalent SQL query from QT.

The design of XC abstracts the implementation details of PT and QT into two separate processes. Each translator process is designed as a Finite State Machine (FSM) that maintains translator internal state while providing a mechanism for code re-entrance. This is particularly important because operations performed by the translators may entail heavy processing, such as serializing large SQL statements or executing PG queries. FSMs allow firing asynchronous events that kick-off backend processing, as well as defining function callbacks that get automatically triggered when events occur. For example, when the results of a PG query are ready for translation, an FSM callback is automatically triggered to process the results and generate the required translation.

4. DISCUSSION

In this section, we discuss some of the implementation details of Hyper-Q platform.

```
Schema
54 00 00 00 30 00 02 63 - 31 00 00 00 54 35 00 01
T (row description) 2 (columns) `c1` (column)
00 00 00 17 00 04 FF FF - FF FF 00 00 63 32 00 00
 column metadata                  `c2` (column)
00 54 35 00 02 00 00 00 - 17 00 04 FF FF FF FF 00
 column metadata
00
Contents
44 00 00 00 10 00 02 00 - 00 00 00 01 31 00 00 00 01 31
D (data row)    2 (columns)    `1` (value)    `1` (value)
44 00 00 00 10 00 02 00 - 00 00 00 01 32 00 00 00 01 32
D (data row)    2 (columns)    `2` (value)    `2` (value)
EOM
43 00 00 00 0D 53 45 4C - 45 43 54 20 32 00
C (command complete) `SELECT`    `2` (rows)
```

(a) PG v3

```
01 02 00 00 39 00 00 00 - 62 00 63 0B 00 02 00 00
 response  message length   table result   dict   symbol vector  2
00 63 31 00 63 32 00 00 - 00 02 00 00 00 06 00 02
  `c1`   `c2`   list          2 (length)  int vector  2
00 00 00 01 00 00 00 02 - 00 00 00 06 00 02 00 00
(length)  `1`        `2`        int vector   2 (length)
00 01 00 00 00 02 00 00 - 00
  `1`        `2`
```

(b) QIPC

Figure 5: Representation of {(1, 1), (2, 2)} result set

4.1 Erlang

We chose Erlang as the programming language to build Hyper-Q. The decision to use Erlang was motivated by the requirement of building an efficient and highly available virtualization system. Erlang is a programming environment that combines powerful abstractions of parallel primitives and is natively able to manipulate network traffic, while providing high availability and massive data throughput. Building on the Erlang parallelism and networking libraries greatly improves our productivity while building Hyper-Q.

4.2 Database Systems Protocols

To communicate with both Q applications and PG databases, Hyper-Q uses several methods to extract information from network messages, package information into messages, and implement process handshake and message flow needed to establish connections with both ends.

When establishing a connection using QIPC specifications, a client sends Hyper-Q a null-terminated ASCII string "username:passwordN" where N is a single byte denoting client version. If Hyper-Q accepts the credentials, it sends back a single byte response. Otherwise, it closes the connection immediately. After the connection is established, the client sends queries in the form of raw text. Hyper-Q sends back query result messages encoding both result type and contents as defined in [22].

When communicating using PG v3 protocol, Hyper-Q exchanges different types of messages with PG database to handle start-up, query, function call, copy data, and connection shutdown requests. An authentication server is used during connection start-up to support authentication mechanisms such as clear text password, MD5, and Kerberos. A PG v3 message starts with a single byte denoting message type, followed by four bytes for message length. The remainder of the message body is reserved for storing contents [18, 19].

A key point to enable two-way communication between a Q application and PG database is handling the different representations of queries and results in the two protocols. PG v3 protocol allows streaming of query results. An initial message describes the schema of the results. Each row in the results is then transmitted in the following messages. At the end, an end-of-content message is sent. On the other hand, QIPC forms the result set in a column-oriented fashion and sends it as a single message back to the client.

Figure 5 shows the raw byte representation of a tabular result set, with two columns `c1` and `c2` and two rows $\{(1, 1), (2, 2)\}$, using both QIPC and PG v3 protocols. To send query results back to Q application, Hyper-Q buffers the query result messages received from the PG database until an end-of-content message is received. The results are then extracted from the messages, and a corresponding QIPC message is formed and sent back to the Q application.

The incompatibility of result set formats between QIPC (column-oriented) and PG v3 (row-oriented) poses a challenge when transmitting large data sets: Hyper-Q needs to buffer the entire result set before it transmits the corresponding QIPC messages. A possible solution is materializing the result set in the PG database (as a temporary table), and then extracting column by column to form the QIPC messages. This problem does not exist when the two end systems use compatible formats, since Hyper-Q can stream the results directly after translation.

4.3 Eager Materialization

This section discusses the need to implement eager materialization of intermediate results during query cross-compilation in Hyper-Q. We use Example 3 for illustration. The function `f` gets interpreted only when it is invoked, e.g., when issuing the query ``f[`GOOG]``. When algebrizing the definition of `f` in Hyper-Q, we store the function definition as plain text in the current variable scope (cf. Section 3.2.3). When `f` is invoked, the textual definition is retrieved from the current variable scope and it gets algebrized.

The first statement of `f` assigns a computed table to an in-memory variable `dt`. Before algebrizing the rest of the function's body, the definition and metadata of `dt` must be stored in the current variable scope, so that the following statements that refer to `dt` can be successfully algebrized. In general, a Q variable assignment statement may need to be physically executed before algebrizing the following statements. The reason is that a variable assignment in Q could have side effects (e.g., modifying other tables). In Hyper-Q, materialization of Q variables into PG objects may need to be done in situ to maintain a behavior consistent with the behavior of Q applications with kdb+.

The previous semantics trigger the need to implement eager materialization of Q variables into PG objects during query translation. In some cases, only logical materialization (e.g., using PG views, or maintaining the variable definition for scalar variables in Hyper-Q's variable store) is sufficient. In other cases, physical materialization (e.g., using temporary PG tables) is necessary for correctness. To illustrate, Hyper-Q generates the following SQL queries when translating the Q query ``f[`GOOG];`` in Example 3 using the physical materialization approach:

```
CREATE TEMPORARY TABLE HQ_TEMP_1 AS
SELECT ordcol, Price FROM trades
WHERE Symbol IS NOT DISTINCT FROM `GOOG`::varchar
ORDER BY ordcol;

SELECT `1`::int AS ordcol, MAX(Price)
FROM HQ_TEMP_1 ORDER BY ordcol;
```

Note that in the presence of variable assignment multiple Q statements may be folded into a single SQL statement, where each variable reference is replaced by its definition.

5. CASE STUDY

The broad vision behind Hyper-Q is to provide full functional compatibility between kdb+ and PG. Naturally, due to the technical challenges associated with an automated solution, there has been some skepticism about the feasibility of our approach.

We describe our experience delivering on this vision and providing real business value in deploying Hyper-Q at one of our early adopters - a large Wall Street investment bank. This customer had a pressing need to migrate some of their workloads into a PG-compatible MPP database (while still keeping the real-time analytics on kdb+). We started this engagement by collecting representative query workloads, which helped us shape our backlog and attack the features in order of importance. In order to achieve a tight feedback loop, we provided the customers with monthly release drops, gathered their feedback and prioritized features for the upcoming releases.

The following summarizes our lessons learned throughout the course of deploying Hyper-Q at the customer:

- The customer was easily able to perform the schema mapping and data movement part of the migration into their MPP database. This reaffirmed our belief that while most automated solutions focus on this step, the real challenge is elsewhere.

- The backup and disaster recovery solution, as well as increased concurrency provided by the target MPP system was critical to the customer's IT department. This further validated our hypothesis that different data processing environments match customer requirements better, and providing a seamless integration between these environments is important to success.

- Since Hyper-Q is a man-in-the-middle technology, one of the initial challenges when operationalizing Hyper-Q was around security. We worked closely with the customer to resolve Kerberos authentication issues and provided them with a solution.

- As we implemented features from the customer workload, we needed a way to ensure the exact same behavior to the application as before. For this purpose we built a side-by-side testing framework, which can be used for internal testing of features, and also used by the customers in their staging environments to ensure correctness of operation.

- In certain cases Hyper-Q improves on the user experience provided by kdb+. For example, error messages in Hyper-Q are more verbose and informative than those provided by kdb+, which can increase productivity especially for system administrators and analysts new to Q and kdb+.

One of the most important insights from this engagement was that while Q as a whole is rather large, customer queries use only a fraction of the language. As a result, we have achieved a sufficiently large coverage of the kdb+ functional surface to make Hyper-Q useful in practical customer use cases, including in workloads with a large variety of Q operators, local and global variables, and unrolling a large class of Q user-defined functions without the need to create user-defined functions in PG. The latter is important as the data analyst using Hyper-Q may not have sufficient rights to create objects at the backend system.

Since Hyper-Q is still under active development, it has certain limitations, which can be broadly summarized in the following categories:

1. Missing common language features, which have a corresponding SQL representation. Naturally, we will close this gap as we move forward.

2. Missing features for which PG does not provide the same built-in capabilities as kdb+, for example kdb+ data types that do not exist in PG. We have not seen many of these in current customers' workloads, but it is possible that they become important in the future. We plan on attacking these limitations through PG's extensibility mechanism, that is, providing customers with a "toolbox" of PG's user-defined types and functions to achieve the desired behavior. Since Q includes complex query language constructs such while-loops and recursion, just-in-time compilation to SQL stored procedure may be required in certain cases.

In addition, there are also certain cases where Hyper-Q enhances the kdb+ experience without breaking application code in areas like configurable concurrency and improved error logging. Despite all roadblocks, we have demonstrated that an automated solution for integrating real-time and historical analytics systems is possible. We have built a system, which solves real customer use cases for a large set of application scenarios. In the future, we plan to continue working closely with customers to prioritize our development according to their use cases.

6. EXPERIMENTS

In this section, we present our experimental evaluation of Hyper-Q. Since Hyper-Q sits between the application and data processing layers, one obvious question that arises is what is the overhead introduced by Hyper-Q?

We focus on evaluating the efficiency of query translation in Hyper-Q. Evaluating end-to-end query performance is outside the scope of this paper, since it mainly depends on the relative performance and capabilities of kdb+ and the analytical databases, which is orthogonal to this work. The goal of our evaluation is to demonstrate that the overhead introduced by Hyper-Q is typically minimal. We also illustrate the feasibility of adding new features in Hyper-Q to cover the language surface of Q.

All experiments are conducted on an *Analytical Workload* driven from customer use-cases. The workload is representative of actual production settings and consists of 25 queries that involve three or more wide tables (e.g., tables with more than 500 columns), joins, and various kinds of analytical aggregate functions. We use Greenplum [20, 23], a massively parallel database system, as the backend database. All experiments are done on a dual core machine with 3GHz Intel Core i7 processor and 16 GB RAM.

Query translation goes through the following stages: algebrization of Q queries to XTRA, optimization by applying XTRA transformations, and finally serialization of XTRA expressions to SQL queries. During algebrization, Hyper-Q needs to lookup metadata (e.g., table definitions) in the PG database catalog to bind parse trees into XTRA expressions. Typically, metadata do not have frequent updates. Hyper-Q provides a configurable metadata caching mechanism with configurable invalidation policies and cache expiration time. Our experiments are conducted with metadata caching enabled.

Figure 6 shows the total time consumed by query translation for the Analytical Workload. On average, the time consumed is around 0.5% of the total query execution time. The maximum query translation time is 4% of the query execution time. Queries # 10, 18, 19, and 20 involve more tables to join compared to other queries. Hence, it takes longer time to algebrize these queries, lookup the required metadata, and serialize them into final SQL queries.

Figure 7 shows the split of translation time across different stages. The optimization and serialization stages consume most of the time in the shown results. This is because the processing done in these stages for analytical queries typically involves multi-table joins and aggregate functions that generate XTRA expressions resulting in multi-level subqueries. During optimization, multiple transformations need to be invoked to prepare the XTRA expressions for SQL serialization (cf. Section 3.3). For example, the generated columns in these expressions need to be processed to prune the unused ones before serializing the final query.

The extensibility of Hyper-Q framework allows building support for the surface of the Q language at a consistent speed. While building the framework has been a challenging task, the payoffs from having an extensible framework are substantial. Typically, supporting a new Q query type in Hyper-Q requires adding parsing routines, AST nodes and query transformations. The Hyper-Q framework handles all the details of establishing the data pipeline, representing new query constructs in the XTRA world, firing up the relevant transformations and exchanging messages in the right formats with the two endpoint systems. By building on the powerful infrastructure provided by Hyper-Q, we have been able to support many customer feature requests out-of-the-box, and provide short turnaround times for missing features. Driven by customer requirements, we expect reaching parity with the bulk of the Q language surface in the very near future.

Hyper-Q machine-generated queries go through extensive correctness and quality assurance checks. The equivalent manually migrated queries pose a much higher risk of bugs and semantic inconsistencies. Adding a new feature in Hyper-Q starts with finding a safe translation, and then, based on customer feedback, adding further optimizations.

7. RELATED WORK

In [1] we have proposed the concept of *Adaptive Data Virtualization* (ADV) and identified the primary requirements for the success of an ADV platform. This paper presents Hyper-Q, the first implementation of an ADV platform. Data virtualization has initially been proposed 30

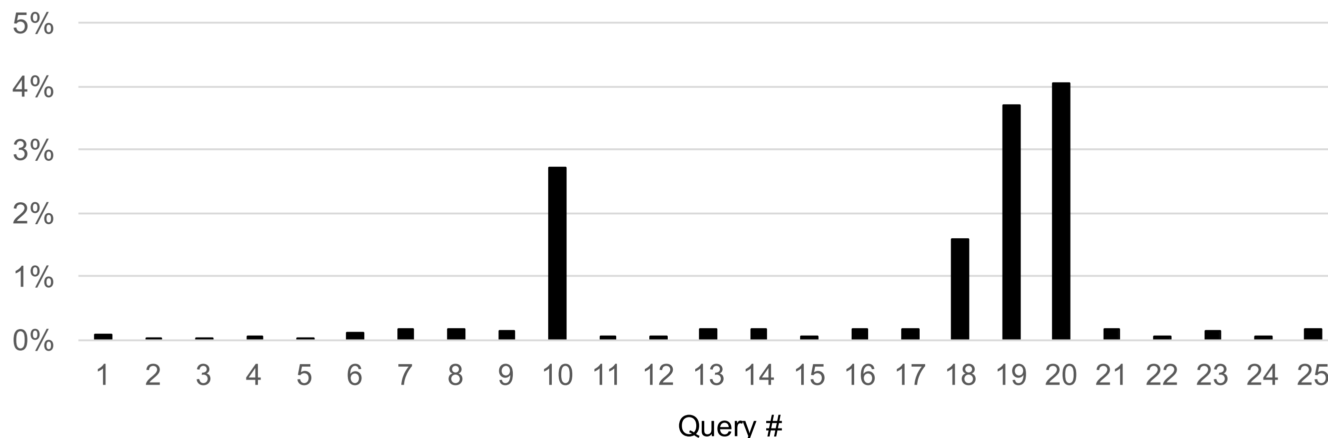

Figure 6: Efficiency of query translation

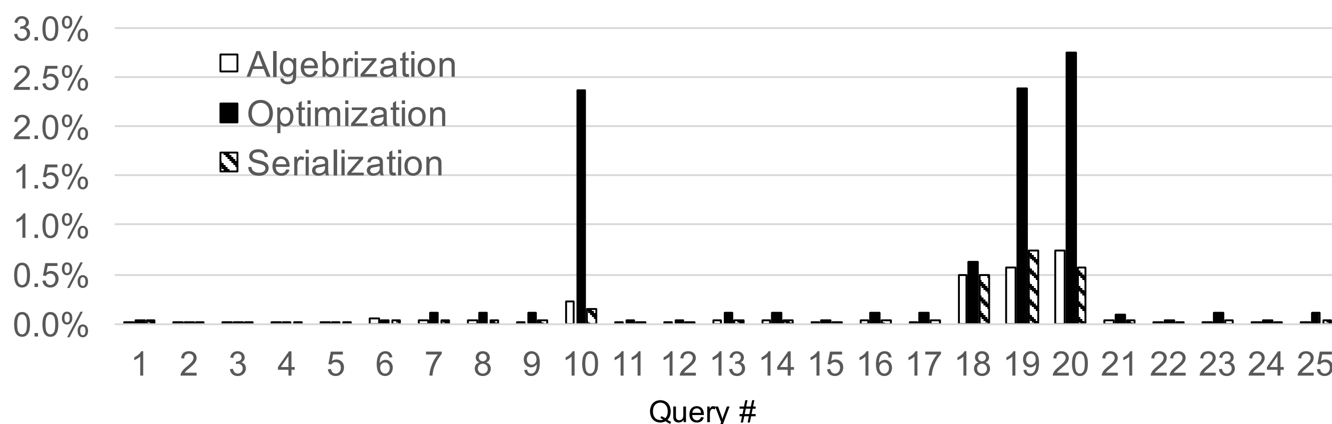

Figure 7: Query translation stages

years ago [10]. Recently it has become an alternative to ETL for business intelligence applications [21] with a number of vendors like, e.g., Informatica [13], Denodo [4], and IBM [11]. Adaptive Data Virtualization extends traditional Data Virtualization to also include applications. In addition to data and the machines it is hosted on, applications also remain unmodified and are transparently adapted by the platform.

The trading algorithms of the financial industry are written for time-series databases like KDB [25]. They focus on low-latency processing of large volumes of real time trade ticks. To meet the timing requirements, the query processing engines manage and process data directly in main memory and in columnar format. Their capacity is restricted by main memory and scalability constraints, limiting the data that can be analyzed and processed to a couple of days. Massively parallel processing database systems (MPPs) like, e.g., SQL Server PDW [16], Amazon Redshift [8], Greenplum [20], and IBM dashDB [12], on the other hand, scale well in data size and compute capacity. Their shared-nothing architecture and feature set suits Big Data analytics workloads well, enabling processing of very large amounts of data. However, they lack the ability to process complex queries on time-series data in real time.

Real-time warehousing solutions like MemSQL [15] combine one or more of the aforementioned technologies (data virtualization or integration of transactional processing in one system to avoid ETL, in-memory processing for better performance) to provide fast analytics. They cannot close the gap between real time processing and Big Data analytics, though, leaving the financial trading applications to the specialized systems built for them.

[6] raises the concern that vendors become “locked in“ to outdated costly technologies, despite various options to use faster and lower cost execution engines. [6] introduced Musketeer, a proof-of-concept workflow manager to map workflows expressed in high-level query languages to various query engine executions. In particular, high-level query language operators are translated into an intermediate representation from which it generates jobs for query execution

engines. However, the intermediate representation is limited to SQL-like operators and depends on user-defined functions to translate the remaining high-level operators.

By tackling the integration of existing applications with different kinds of data management system, adaptive data virtualization enables businesses to leverage the advantages each system provides and combine them seamlessly in their existing applications.

8. SUMMARY

Adaptive Data Virtualization is the broad vision of decoupling applications from the underlying database systems in a way that is completely transparent and non-intrusive. In this paper, we presented Hyper-Q, an initial implementation of this vision. Hyper-Q enables applications written for one specific database to run on a wide variety of alternative database systems – fully transparently and without requiring changes to the application. Effectively liberating enterprises from vendor lock-in, Hyper-Q provides businesses with unprecedented flexibility to adopt database technology and modernize their data infrastructure.

The financial use case of bridging real-time and analytics has proven a great starting point for our development as it required us to address differences between systems on all conceivable levels. As a result, our framework matured substantially over the past 12 months; great care has been taken to delineate platform and plugins cleanly. The resulting system clearly demonstrates the feasibility of our vision. Our initial experience in implementing this platform has been very encouraging and additional plugins for other languages are currently under development.

While Hyper-Q as it stands today is not a data integration or a federated system, it can be used as a building block in this context. For example, in a federated system which delegates parts of queries to different systems, Hyper-Q can be used to translate the query to the language of the corresponding backend system, and convert the results back into the format expected by the federated system.

9. REFERENCES

[1] L. Antova *et al.*, "An Integrated Architecture for Real-Time and Historical Analytics in Financial Services," in *BIRTE*, 2015.

[2] C. Bear, A. Lamb, and N. Tran, "The Vertica Database: SQL RDBMS for Managing Big Data," in *Proceedings of the 2012 Workshop on Management of Big Data Systems*, 2012.

[3] L. Chang *et al.*, "HAWQ: A Massively Parallel Processing SQL Engine in Hadoop," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, 2014.

[4] Denodo. [Online]. Available: http://www.denodo.com/

[5] S. Garland, "Big Data Analytics: Tackling the Historical Data Challenge," *Wired Magazine, Innovation Insights*, 2014.

[6] I. Gog *et al.*, "Musketeer: all for one, one for all in data processing systems," in *Proceedings of EuroSys*, 2015.

[7] C. Gorman. [Online]. Available: http://www.firstderivatives.com/downloads/q_for_Gods_Mar_2013.pdf

[8] A. Gupta *et al.*, "Amazon Redshift and the Case for Simpler Data Warehouses," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015.

[9] J. Hanna. [Online]. Available: http://www.firstderivatives.com/downloads/q_for_Gods_Oct_2012.pdf

[10] D. Heimbigner and D. McLeod, "A Federated Architecture for Information Management," *ACM Transactions on Information Systems*, vol. 3, pp. 253–278, 1985.

[11] IBM. [Online]. Available: http://www.ibm.com/

[12] IBM. IBM dashDB. [Online]. Available: http://www-01.ibm.com/software/data/dashdb/

[13] Informatica. [Online]. Available: https//www.informatica.com/

[14] kdb+. [Online]. Available: http://kx.com/software.php

[15] MemSQL. [Online]. Available: http://www.memsql.com/

[16] Microsoft. Microsoft Analytics Platform System. [Online]. Available: http://www.microsoft.com/en-us/server-cloud/products/analytics-platform-system/

[17] NYSE Market Data. [Online]. Available: http://www.nyxdata.com/Data-Products

[18] PGv3 Message Formats. [Online]. Available: http://www.postgresql.org/docs/9.2/static/protocol-message-formats.html

[19] PGv3 Protocol Flow. [Online]. Available: http://www.postgresql.org/docs/9.2/static/protocol-flow.html

[20] Pivotal Software. Pivotal Greenplum Database. [Online]. Available: http://greenplum.org/

[21] L. J. Pullokkaran, "Analysis of Data Virtualization and Enterprise Data Standardization in Business Intelligence," Ph.D. dissertation, Massachusetts Institute of Technology, 2013.

[22] QIPC. [Online]. Available: http://code.kx.com/wiki/Reference/ipcprotocol

[23] M. A. Soliman *et al.*, "Orca: A modular query optimizer architecture for big data," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, 2014.

[24] Teradata Aster Big Data Analytics. [Online]. Available: http://www.teradata.com/Teradata-Aster/

[25] A. Whitney and D. Shasha, "Lots o' Ticks: Real-Time High Performance Time Series Queries on Billions of Trades and Quotes," in *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, 2001.