



Design Considerations for a Scalable Data Fabric



Table of Contents

- 3 Executive Summary
- 4 Distributed Data Access Is a Specific and Poorly Met Need
 - 5 The Goal Is to Build Infrastructure, Not Custom Applications
- 5 The Environment Is Distributed: How Do We Unify Data for Users?
 - 6 Option 1: Use Tools from the Data Warehouse Environment
 - 6 Option 2: Use Hadoop Tools
 - 6 Option 3: Assemble Your Own Tools from Components in the Environment
- 7 The Evolution of Federated Data
 - 8 Loosely Coupled, or Query Federation
 - 8 Tightly Coupled Federation, Also Called “Data Virtualization”
 - 9 Teradata® QueryGrid™: Merging Federated Access with Data Transfer
 - 10 When to Use Different Technologies: Five Tradeoffs
 - 10 Repeatability: How Well-Known Are the Data Requirements?
 - 11 Query Concurrency: How Many Simultaneous Queries Do You Expect?
 - 11 Response Time: What Is the Expected Time to Get Results for the User (or Application)?
 - 11 Data Volume: How Large Are the Datasets You Need to Access?
 - 11 Copying Data: Does Data Need to Be Copied from One Place to Another Before It Can Be Used?
 - 12 When Should You Use These Technologies?

- 12 Nine Key Factors to Evaluate for a Scalable Data Fabric
 - 12 Scalability
 - 13 Efficiency
 - 13 Concurrency
 - 13 Topology of Systems and Connections
 - 14 Data Conversion and Type Management
 - 14 Support for Platform-Specific Capabilities
 - 14 Security
 - 15 Monitoring and Administration
 - 15 These Factors All Contribute to the Overarching Goal, Ease of Use
- 15 Teradata QueryGrid™: Solving the Hard Problems of Scalable Data Use
 - 16 QueryGrid Architecture: How It's Built
 - 16 General Design
 - 17 Parallel Architecture
 - 17 Network Awareness
 - 18 Optimization
 - 18 QueryGrid Link and Connectors
 - 19 QueryGrid Manager
- 19 Conclusion
- 20 About the Author
- 20 About Third Nature
- 20 About Teradata



Executive Summary

Access to data was and is a significant bottleneck in analytics development. In most large organizations, this bottleneck is a function of two related trends. In the first, data is more siloed and distributed than ever before. The environments that were supposed to function as central hubs for data access and processing—first the data warehouse; more recently, big data platforms such as Apache Hadoop™—are commonly supplemented by one or more fit-for-purpose repositories.

In the second trend, organizations want to do more and different things with the data they collect. For example, data warehouse systems are designed to address known or pre-determined questions and needs. Increasingly, however, organizations want to support a range of new analytics use cases, many of which are characterized by open-ended exploration and discovery. Unlike the warehouse, which is a primary destination for derived transactional data, these new use cases require information from systems that provide data on interactions, such as web logs, mobile devices and sensors.

Nor is that all. These new analytics use cases have unpredictable data and processing needs, too. Users often determine the data they need on demand, not as part of a scheduled loading process. Last and most important, whereas the data contained in the data warehouse was traditionally determined by IT, individual analysts define data requirements at the time of use for the new analytics uses.

A general solution for data access is needed. The challenge is that the available technologies—be they custom-built middleware or commercial products—cannot

scale to address this requirement. Most organizations, even the very largest, lack the resources and expertise to build and maintain a custom-coded middleware solution to act as a data fabric that connects systems.

Commercial data integration tools are ill-suited to address this problem, too. Most such tools are optimized for scheduled, batch-oriented operation by IT professionals. Their designs are based on a number of assumptions that are no longer viable, e.g., that data flows are predictable and can be reused; and that the people using them possess coding or low-level technical expertise.

The upshot is that analysts require on-demand access to different sources and types of data. Thanks to the siloing of both data and analytics engines, it is precisely on-demand access that is so difficult to facilitate. For example, an analyst should be able to initiate data access from the platform of her choice—wherever she happened to be working at the time. Ideally, she wouldn't have to use or open up a separate tool to do this: access would, instead, be facilitated by means of a transparent data fabric.

Ideally, this solution would be smart about how it moved data. Instead of moving data in bulk from the platform on which it lives to the environment in which the analyst works, this solution should perform operations where it makes the most sense. In other words, data could be processed in place, with the result that only a small subset of data—the data the analyst actually needed—would be moved. Data could also be moved to the analyst, where the appropriate tools exist to process it. This is critical. At today's large volumes, any viable solution must minimize data movement to only what is necessary. When it does move data between systems or contexts, it must do so as efficiently as possible.

Distributed Data Access Is a Specific and Poorly Met Need

First the data warehouse market and then the Hadoop market made a promise they couldn't keep. They said there would be only one place for data, and that all your data needs would be met there. This is a myth. We have an analytical ecosystem with multiple data stores and processing engines, none of which contains all of the data.

We need a general solution for data access, a need that many people didn't face until recent years. This need isn't met by conventional data integration tools which are designed to move tables and files from one system to another, usually in batch operations. We need a solution that operates on-demand. We need a solution that not only minimizes data movement, particularly at today's large volumes, but one that—when it does move data between systems—is able to do the task efficiently.

Most of the data infrastructure we built assumed people would be passive information consumers. This assumption is built into the foundation of the BI model: it's implicit in the standard of a data warehouse as "a read-only repository," yet the data environment that has evolved over the past twenty years supports much more than passive consumption by reports and dashboards. The use today includes exploration and deeper analysis of data, alongside the fixed dashboards and reports.

Unlike business intelligence, data exploration and analysis have unpredictable needs. We don't know the specific details of how data will be used. Exploration is predictable only at a coarse level, so that we can build repositories to store datasets with less rigorous structure and quality control. An example of this type of system is the data lake. It's used for large-scale collection and exploratory use. It's not used for common, core data, creating a problem for analysts who want to link new data from the lake to the core data in a warehouse or analytics system.

The challenge facing analysts is the lack of easy mechanisms to access data in the different environments. Getting data into Hadoop or Amazon S3 is as easy as copying a file. This ignores the problem of generating that file at a remote data source, transferring it, and ensuring that it is properly formatted and accessible, particularly if it's large. Even a few tens of gigabytes can be problematic for many source systems.

Likewise, getting data from Hadoop or S3 into other platforms is a challenge. It's not hard to access the data alone, but it is hard to fetch, move, load and join that data, more so when the volume is large. When moving data in either direction, the tools for data access are aimed at developers, and to a lesser extent, administrators with technical skills on the platform, but not users.

Our environments are weakest when it comes to support for on-demand data access.

When addressing problems from the new market environment (Hadoop, NoSQL, Apache Spark™ and the like), the data problems are a mirror image of those in the data warehouse environment. The warehouse is the primary destination for core transactional and reference data. A non-relational repository, by contrast, stores a large volume of potentially valuable data. Nevertheless, not all core transactional or reference data is stored in this environment. As a result, the user or developer must get this data from databases or other sources and load it into the big data environment to analyze it.

All of this underscores a critical problem: our environments are weakest when it comes to support for on-demand data access. This is the result of an assumption—that there is only one place where data resides prior to use. While true for core metrics and common data, the assumption is not true when one has to diagnose a never-seen-before problem, analyze information about a new business project, or build an analytic model. These all share one characteristic. They all have unknown data needs.

This on-demand access pattern shares little with the patterns in a data warehouse environment. The patterns in the warehouse are repeatable data flows from known sources to known targets, and repeatable querying from those targets.

On-demand data and processing is promoted in the Hadoop market with statements about schema-on-read, multiple processing engines, and data lakes. The discussion ignores the problems of distributed data that isn't stored on the platform. To solve those problems requires technical developer skills and specific knowledge of underlying platforms.

The Goal Is to Build Infrastructure, Not Custom Applications

Approaching the data access problem by working from the most familiar platform is a mistake because it ignores capabilities of the less known platforms and favors one platform's technology and approach. Tools are invariably tied to the vendor of that platform. We need a solution that works equally well across the environments that analysts and developers use—not just the most familiar platform, whether it is new or old technology.

Data movement in the new analytics environment is not one-way, it's bidirectional. Data can originate from any number of sources. New or aggregated data can be pushed back to those sources. Depending on the work an analyst is doing, they may initiate movement from different systems at different times.

There is no center in the new environment. Every system is a possible source of data, and a possible source of queries to other systems for data. Data access requires a fabric, not a one-way connector or a retrieval mechanism that only works from one location.

The goal of infrastructure is to abstract problems away from the needs of one specific application.

Integrating data for a single project is easy. Doing it for more than one project, which may add new source types or engines, is hard. Now you are building multi-purpose tools rather than assembling components within a single environment. Combine this fact with user requirements that their tools be uniform, consistent, and more reliable than the applications they integrate, and you face an order of magnitude increase in software complexity. This is the problem with custom-building multi-use middleware rather than one-off data integration solutions.

When designing data access tools, the goal is to have a reusable tool, not customized integration that is built into a single application or environment. Custom integration like this means the work will need to be redone as each new application or analytical model is added. The goal of infrastructure is to abstract problems away from the needs of one specific application.

What we want is ease of use for anyone who has to get or move data in an on-demand fashion. We want to make the work of the analyst or developer reusable, no matter which environment they use. The work should look the same to the user no matter whether they are accessing data from a database or Hadoop. This enables portability, allowing IT to change source platforms and relocate data without breaking analytical code that sides on top.

The Environment Is Distributed: How Do We Unify Data for Users?

The first option most architects consider is the primary data platform in the environment. It might be a database, it might be Hadoop, or it might be something else. The starting point is often driven by the idea of a centralized data architecture, for example storing all the data in the EDW or in a data lake built on Hadoop. The natural (and wrongheaded) assumption is that there is only one central data repository.

The analytical ecosystem is inescapably distributed, not centralized. It's not hub and spoke, it's multiple hubs. Different hubs arose for different purposes, each with its own constituent parts and core central platform. For example, the BI hub with a data warehouse at its center, and the customer analysis hub with a product like SAS or SPSS at its center, or a data lake with Hadoop at its center. We need something to efficiently shuttle data between each of these hubs.

The problem lies in connecting disparate systems to move data. This problem is hard to solve because it's seen—or rather, glimpsed—only in piecemeal fashion, one user or project at a time. In this way it is invisible, an assumed cost of IT. In architecture diagrams, the work of integration is not in the boxes but in the arrows that connect them—arrows that weren't supposed to be there because there weren't supposed to be disparate boxes.

If we ignore the “centralize all the things” default position in favor of one that assumes there are always islands of data, then there are a number of tools and technologies for data access based on the platforms in use in those islands. In the first instance, most of these options make the same assumptions: you have a place to put data, you have access to the necessary tools, and you have the platform-specific skills required to use them. While these

assumptions are generally valid for developers, they are not true for users. In the second instance, the tools are not architected with heterogeneity in mind, but with the idea that data moves from somewhere else into the platform that tool is native to.

Option 1: Use Tools from the Data Warehouse Environment



The most common starting point is to use database-centric tools, if only because many analysts and IT people are familiar with them. Setting up connections to remote databases of different types is usually possible, and most databases have some rudimentary connectivity to Hadoop environments.

The analyst's challenge isn't purely access to data, however. Database-oriented tools make it easy to access data, but that data still has to be moved to one platform before it can be linked. Once the various datasets are co-located, analysis can proceed.

Databases have tools to export whole or partial tables locally and over networks. There are bulk loaders to load large datasets quickly. There are two problems with this approach. The first is that data access in this paradigm is batch-oriented, not on demand. The second is that this batch style of data movement is aimed at developers and administrators, not at users.

Moving data in this way involves the use of multiple tools to work with the databases and exported files and presupposes a familiarity with the operating environments at either end. This approach also requires developer- or DBA-level expertise and privileges to accomplish the work—for example allocating space for large files and tables, or invoking fast load processes. If the dataset is too large, a plausible scenario in data exploration use cases, these mechanisms become unwieldy or impossible to use.

ETL tools are almost ubiquitous in data warehouse environments, and many have added capabilities to work with Hadoop, Spark, NoSQL databases, and other platforms. They solve the data movement problem in a developer-friendly way, but they are overkill for users, who just need simple access, join and move functions. They are also not designed for on-demand data access, but for a development environment.

Option 2: Use Hadoop Tools



Once data is located in Hadoop, the job of linking datasets can be made simpler by using SQL rather than using code. SQL-on-Hadoop options make the data easier to integrate and access, abstracting away from a coding model.

A challenge for architects is that each Hadoop vendor has its own preferred SQL alternative. For example Cloudera® wants you to use Apache Impala™, Hortonworks® prefers Apache Hive™, and MapR® wants you to use Apache Drill™. Independent of these are SparkSQL, which is becoming more popular, Presto, which is gaining favor in the enterprise thanks to support from Teradata and adoption by Amazon, and many proprietary SQL-on-Hadoop products.

There's no obvious convergence to one choice or even one model of SQL-on-Hadoop, which makes it difficult to use as a foundation for data access and movement. Even though SQL-on-Hadoop is a necessary component for data access, it isn't sufficient. It helps to link data in Hadoop, but does nothing for data stored elsewhere.

Like databases, Hadoop has a collection of tools for moving data. Tools like Apache Sqoop™ are the equivalent of import and export tools, with the added ability to get data from remote databases. Also like databases, these tools are generally designed for batch movement of data from one system to another. The problem is that they are not designed as end-user tools for quick movement of subsets of data. Like their counterparts in the data warehouse world, they are designed for developers and administrators with technical skills.

Option 3: Assemble Your Own Tools from Components in the Environment



One platform's tools are not an answer to a multi-platform problem. We need to provide a foundation for access across multiple systems that is usable by analysts and developers. Fortunately there are many components across both the database and big data environments that can be used to construct data access tools.

The challenge one faces when building these has to do with the nature of the problem. This is not as simple as connecting several different platforms. That's the base layer. We want a facility that will allow reuse by multiple

parties. Building a multi-user facility starts a long march up the custom-coding mountain because this is much harder than building one-off solutions to specific problems or adding simple point-to-point data movement.

The usual path one follows is to first build basic point-to-point access and movement tools. For a few users, this approach may work, although it becomes difficult to maintain as systems are upgraded, environments change, and the components evolve at different rates. The maintenance challenge of custom integration was hard enough with enterprise systems in the past. The rapid rate of change with open source and the Hadoop market makes it more challenging.

If more than a handful of users and developers use the tools, one discovers the challenges of concurrency, multi-user performance and scalability. Concurrency, parallelism and distributed reliability are deep topics that require esoteric knowledge to do well. Even small things like connection pools and connection initiation times can become big problems with just a few users, particularly with interactive response time. These are problems that have preoccupied commercial software vendors for years, and are core to solving parallel data access problems.

Manually assembling data access tools is an acceptable approach if you are building a prototype or proof-of-concept, but not if your goal is to scale up an organization's data environment to serve different uses and more than a handful of users.

The Evolution of Federated Data

The problem we face is that the tools associated with the old ways of accessing and moving data don't meet new needs well, and the tools aimed at new uses are suboptimal for existing data access patterns. Mixing and matching of these is the right answer, but doing it in an ad-hoc fashion with independent tools is both technically difficult and expensive. What should an IT architect do?

There is no "correct" model. Your analytics needs and constraints dictate what is required. What we want is support for distributed data access, because we can't count on data being centralized before it is used—this is the on-demand access challenge we face today. We also want something that makes distributed data accessible as if it were in one place, which means we want a centralized access model.

Four Architectural Goals for Distributed Data Access

There is rarely a single right answer in architecture, particularly for data integration. There are only tradeoffs between alternatives. Working from architectural principles is a good approach to help keep goals in mind as one evaluates technology tradeoffs.

Centerless (or flat): You can't dictate one platform as the center when any data repository might be a source or a target for data access or transfer, and data flow can't be one way. The underlying topology is a fabric, not a flow, hub or point to point connection. Anything you choose needs to work equally well regardless of which platform the user starts from. To the user it should always appear that they are at the center.

Parallel: Analysts often want to filter or aggregate large datasets and join them with other data. Data access must be parallel-aware to scale well and support these activities. Unfortunately, we are dealing with systems of widely varied sizes. It is easy to overwhelm a small platform with data from a large cluster, so any solution must be aware of the resources available across all of the systems. This means any solution must be aware of resources where data resides so they can be exploited when processing the data and must be parallel-enabled for fast transport.

Abstraction: We need to support analysts in addition to developers. They don't need to see the technical details of the underlying connections, processing and transport. They only need to see the data and have some sense of its location and whether it is feasible to access in place or move it to where it is needed. The value of abstraction is that it gives the users (and developers) portability for their work, regardless of what platforms they use.

Reuse: To enable reuse you can't be locked into one platform. Focus on reuse and portability, because that is where productivity gains come from. Many data access solutions today require rewriting code or queries if they are moved from one place to another. Reuse and portability go hand in hand.

The primary technology on the market that meets these needs is data federation. Federation is designed to access remote data but to make it appear as if the data is local. It abstracts the access problem above platforms by hiding the implementation details. One doesn't know if one is fetching data from a database, Hadoop or NoSQL. Federation products separate the problem of access into a layer that can be managed independently of data storage and working areas. They provide something akin to a "virtual hub" for data.

Data federation technologies can be divided into two categories based on whether they are loosely coupled or tightly coupled systems. A loosely coupled system is like the model of data federation that came out of the database and BI tool markets—they commonly used the term "query federation" so we will use that term to distinguish between loosely and tightly coupled federation.

Loosely Coupled, or Query Federation

The idea of query federation has been around for many years but is often overlooked as a solution to current challenges. In part this is due to legacy infrastructure limitations, like internal network bandwidth and the limited ability of tools to scale. In part this oversight is due to the recent growth of large islands of data and the inability of the vendors to address large-scale needs.

The goal of a federated query is to simplify access to data from multiple databases. The concept is to write one query that accesses two or more different databases to unify the results, as shown in Figure 1. To the user, it appears that one is accessing and joining tables from the same database.

Database vendors have been providing this type of distributed access for a long time. The starting point was database links that let different databases from the same vendor share tables. Then came connectors and gateways that permitted other types of databases to access tables. These approaches typically had limitations, for example only working with one vendor's products, or the inability to access remote and local tables within the same query.

Query federation is also used by some BI tools to make it easier to access multiple data sources without the need to integrate data beforehand. As with databases, these tools suffer from the limitation that a query can only be issued from within that vendor's tool. This means any other tools

are unable to access the data, furthering the data silo problem. Their use is often limited to a developer interface or to use of the data only inside that vendor's database or query tool.

Query federation is not often used, due to the limitations mentioned above. It has been deployed mainly as a point solution for known, repeatedly-used data stored in relational databases, and is usually hidden from analysts, end users or application developers. It's a good solution for a small subset of the problems.

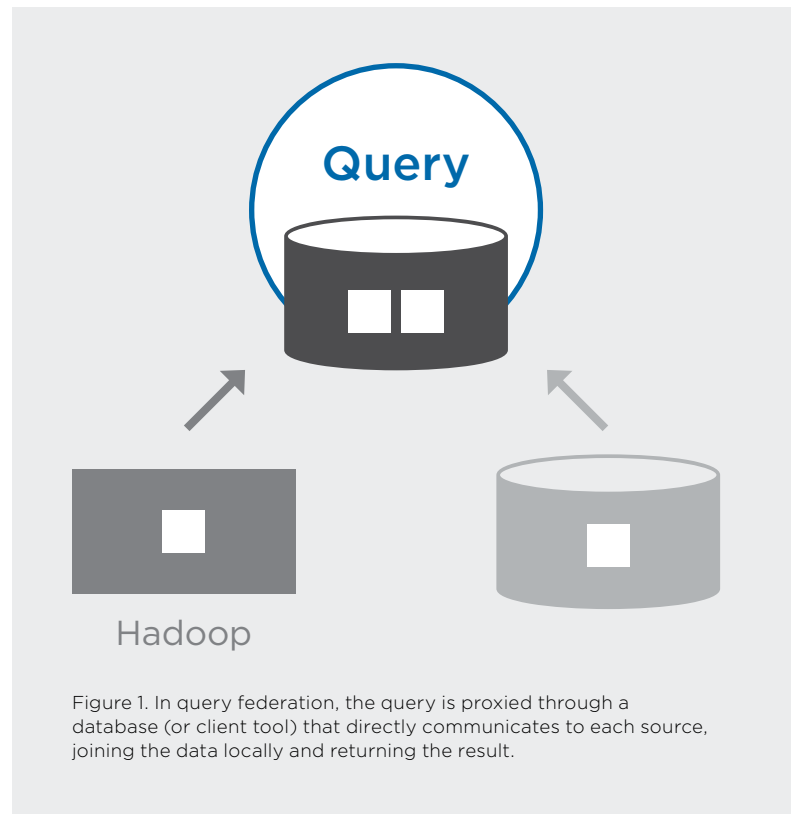


Figure 1. In query federation, the query is proxied through a database (or client tool) that directly communicates to each source, joining the data locally and returning the result.

Tightly Coupled Federation, Also Called "Data Virtualization"

Tightly coupled data federation tools abstract the problem away from any one database or query tool, making queries independent from the underlying databases. They provide connectors to different data sources and permit a single query to join data across them.

In order to resolve distributed joins and operate on the joined set—for example aggregating or filtering the results of a distributed join—they have rudimentary database

capabilities as part of the federation server. The server acts as a place to join data before delivery. To avoid incompatibilities with different levels of SQL support between databases, the SQL syntax is usually limited to a common subset understood by all of the databases.

Tightly coupled federation is often referred to by the product category it spawned, data virtualization (DV). We will use DV to refer to tightly coupled federation, as this is the more popular term for the category of products.

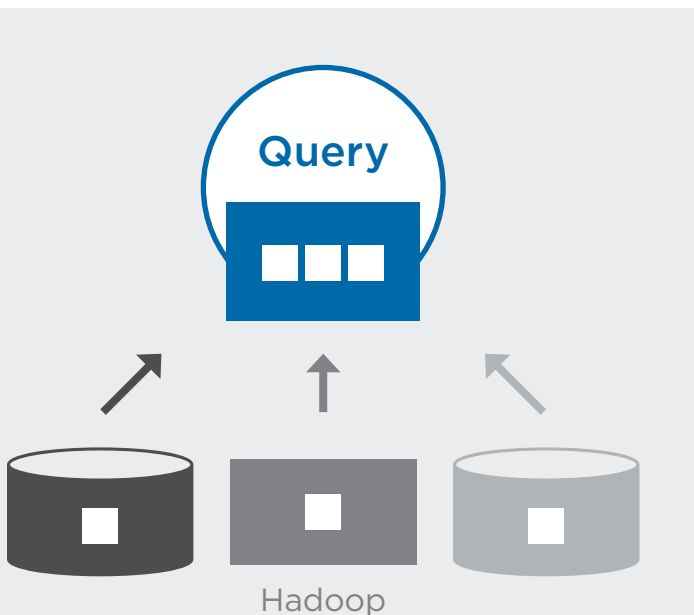


Figure 2. In data virtualization, the DV server processes the query against a view, which is then resolved against remote sources, joining the remote data in the DV server and returning the result.

The evolution that made federation more broadly usable is the idea of using views to hide the underlying details from user queries. A view makes the results of a query look like a table while hiding the details of the distributed query processing in the server.

Combining the idea of database views with query processing, DV creates a collection of virtual tables that appears as a virtual database. Each virtual table is actually a query that joins remote data sources together. The views can be treated like tables and combined with other views or in queries, allowing a virtual database to be built over many sources. Any client tool can access this virtual database via a SQL connector. Users and developers query the views as if they were tables stored in a database, as shown in Figure 2.

Several other technical innovations were added. There is no requirement that the data be delivered as a table. A view is simply a query result set. It's easy to make the DV server appear as a file server and present the same result set as a file. It's just as easy to package the result set in JSON or XML and make it appear as a web service callable by any application.

Likewise, data sources need not be restricted to relational databases. Any type of database, file or web service can be accessed. This allows a DV server to be a sort of universal access layer, proxying any data source type to any query type. A table can become a web service. A web service joined to a file can be turned into a SQL-accessible table.

Data virtualization doesn't come without complexity. While a DV server is a virtual database, it doesn't have the same performance characteristics as a real database because it accesses the data from remote sources over a network. The query optimizer inside a DV server is more complex than in a database. It has to optimize each of the component subqueries that are sent to different data sources as well as the final assembly—it essentially optimizes multiple networked databases.

The DV approach has benefits. A SQL-knowledgeable developer can write complex queries and make the results easily accessible to a large audience as a virtual table. Any user can use SQL to access the table, or use web service or file-access tools. This allows DV to blend different application technologies into a data service layer.

DV has the limitation that one must create and publish data models via the server before data is accessible. Typically, only developers have the ability to define the queries that make a virtual table. This limits use to data access needs that are known in advance, making DV less useful for exploratory data uses typical in data analysis.

Teradata® QueryGrid™: Merging Federated Access with Data Transfer

Teradata® QueryGrid™ takes a step back to take a step forward: back to loosely coupled federation from the model-first approach of data virtualization and forward to an abstracted service for bidirectional data access and transfer that is independent of storage platforms. It can still optimize queries, take advantage of source-specific functions, and do this with the full parallel capability of each platform, unlike the single-threaded nature of data federation alternatives.

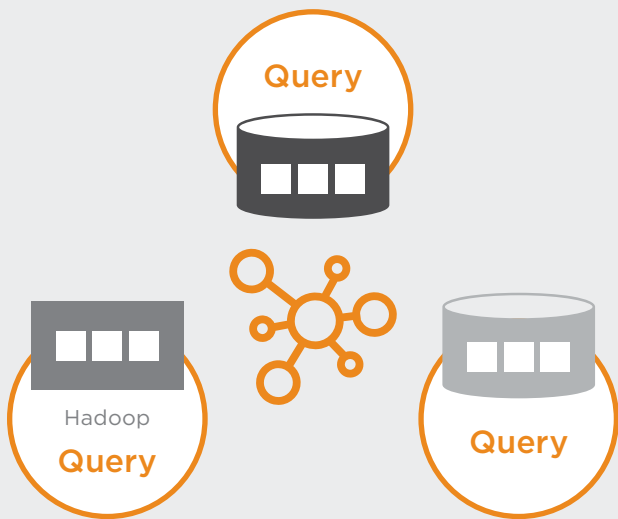


Figure 3. Teradata® QueryGrid™ works by connecting the systems together into a fabric. In QueryGrid the same query can be run from any system. QueryGrid provides user access from anywhere and connects the systems to each other in the background.

The QueryGrid concept orchestrates on-demand data access at scale across multiple platforms in the analytics environment. QueryGrid allows users to initiate queries from any linked platform and access any system it is connected to, combining data from multiple platforms in a single query. It does this without requiring that all the data be copied to one place first. QueryGrid can access and transfer data to and from participating systems, unlike the one-way data flow of data federation, as shown in Figure 3.

QueryGrid is designed using SQL as the common language, but it goes further by allowing a user to take advantage of platform-specific processing capabilities any source. This means QueryGrid can take advantage of the processing power available in multiple parallel engines using the native syntax.

The use of SQL does not limit QueryGrid to database use. QueryGrid integrates with any distribution of Hadoop via Presto, a distributed SQL-on-Hadoop project originally developed by Facebook. With this integration it is possible for QueryGrid to provide parallel access to any data available in Hadoop, from file formats to sharded MySQL to Amazon S3.

Any technology that operates as an infrastructure layer requires security, reliable network transfer, workload management, administration and monitoring. QueryGrid, like DV, is designed to be intelligent middleware with these features. For more details about how QueryGrid is designed to address large-scale parallel and distributed data access, see the section titled “QueryGrid Technical Design.”

When to Use Different Technologies: Five Tradeoffs

Query federation, data virtualization and the parallel data fabric approach taken by QueryGrid are variations of distributed access that don’t require physically copying it can be used. Each approach has a slightly different approach that embodies assumptions about how data is used. This results in design tradeoffs the limit their applicability to some workloads. The following items are key workload attributes that constrain how these technologies can be applied:

Repeatability: How Well-Known Are the Data Requirements?

A high degree of repeatability means the data requirements are well understood. If it’s possible to anticipate data requirements in advance then data can be physically copied in advance to where it is needed, as it is in a data warehouse, or a virtual table can be built in a DV server to access the remote data.

The model-first approach is a strength of DV for repeated use of the same data. A modeling interface within the DV environment is used to construct and publish views, allowing developers to build highly optimized queries and hide this complexity behind a virtual table.

The downside is that views are predefined, making this approach untenable for exploratory work. The need for an in-tool modeling environment and the inflexibility of fixed views is a limitation of DV use for ad-hoc problems, narrowing use to more predictable application and BI workloads.

Loosely coupled federation and QueryGrid are better suited to ad-hoc use, allowing them to be applied when the data requirements are unclear. A user can explore the data through a series of queries and arrive at the data they need. At the end of this process, if the data will be used repeatedly, it is possible to use the query to move data to another location or present it as a view.

Performance Area Covered by Data Virtualization, Query Federation, and Teradata® QueryGrid™

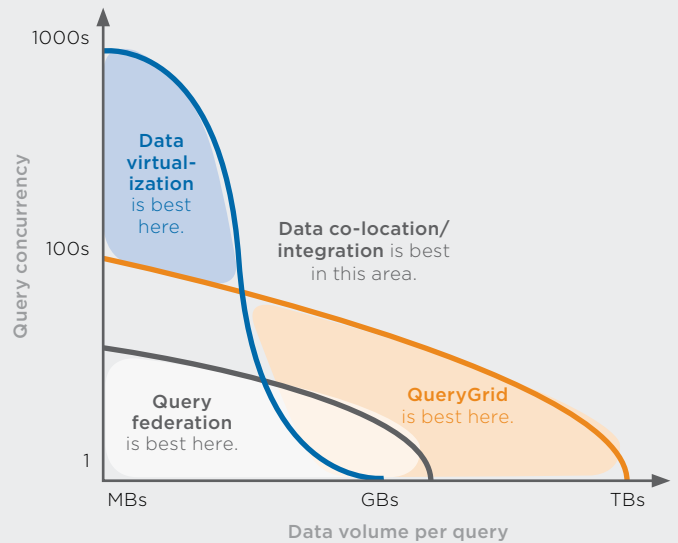


Figure 4. Data virtualization is capable of very high concurrent query loads, but only at relatively small data volume per query. As the data volume grows into gigabytes, performance drops quickly. Loosely coupled query federation, as built into databases and query tools, tends to support far lower user concurrency, but can work with larger data volumes than data virtualization servers permit today. QueryGrid is designed to handle extreme data volumes at the level of user concurrency provided by underlying systems. At small data size and low user count, anything will work so the simplest tool is the best choice. When the user count is high, data virtualization is a better solution. When data size is large, QueryGrid is a better solution.

Query Concurrency: How Many Simultaneous Queries Do You Expect?

Concurrency and repeatability are often linked: many workloads with high concurrency also have high repeatability, for example application workloads or dashboards in BI environments where the same queries are run by many users at the same time.

Data access choices are more limited when there are many simultaneous queries. In general, loosely coupled federation is designed assuming few users, while the tightly coupled DV approach is designed for higher concurrency. QueryGrid falls between the two, limiting concurrency in order to increase parallel throughput.

Just as it's not always possible to move data in advance to where it is needed, sometimes it's not possible to address high concurrency requirements without physically consolidating data in advance, particularly when there are strict response time requirements.

Response Time: What Is the Expected Time to Get Results for the User (or Application)?

Response times are often guaranteed as part of a service level agreement. The on-demand nature of loosely coupled federation and QueryGrid make them less suitable for uses with strict SLAs unless the queries are predictable and can be tuned. A tightly coupled DV server is a better choice when one has guaranteed SLAs because of the approach DV takes to presenting a predefined view. This permits advanced tuning and caching prior to use.

If response time is less strict then the tradeoffs affecting choice will revolve around concurrency and data volume. Figure 4 shows what the relative response time tradeoff between concurrency and data volume looks like for the different approaches.

Data Volume: How Large Are the Datasets You Need to Access?

Federation products are typically designed with a scale-up model to address data volume scaling. Parallelism is limited to what can be done on a single server, and sometimes what can be done within a single process on the server. This is particularly true of loosely coupled federation that is embedded in client tools and databases.

Data virtualization is designed to act as a centralized hub, albeit without physically consolidating the data into one place. This works well for problems that have high

concurrency and response time requirements, provided the data volume is low. It is easy for large queries to overwhelm a DV server because the servers typically do not have a parallel distributed architecture. Accessing a lot of data highlights the central bottleneck of the DV server.

At moderate scale and low concurrency, any choice can be made to work. The challenge is then whether one knows the data requirements in advance or only after looking at the data. The tightly coupled federation model only works with known data requirements. Data volume is usually the biggest constraint mentioned for distributed data access. Accessing and processing a large volume of data generally requires a scale-out parallel architecture.

Copying Data: Does Data Need to Be Copied from One Place to Another Before It Can Be Used?

Most products are designed to facilitate access in read-only fashion. One of the needs we have in an analytical ecosystem with multiple data stores and different processing engines is to link and position data where it will be used.

Copying data can be accomplished (via caching) with federation tools provided the data volume is low. As data volume increases, the ability of these node-serial architectures to keep up is severely limited.

Federation was not designed to move data from one platform to another. It was designed to enable live access to distributed source data. Federation was envisioned as a technology to make remote databases accessible. Some products have the ability to insert the results of a query into the local environment, but the data flow is one way and manually configured for each use.

The federation approach is about publishing the results of a query as a view for others to reuse. The loosely coupled approach is intended for writing queries to fetch results for use. Neither of these is about exploring potentially large volumes of data via query and moving data from a place where it is stored to another where it can be processed. QueryGrid is intended to solve these types of problems.

When Should You Use These Technologies?

The challenge for an architect is that all of these constraints are factors in a workload, so one must prioritize between them, resulting in tradeoffs for any application. For example, querying large amounts of data reduces concurrency, or having a large number of queries places limits on response time guarantees when they are not repeatable.

There are other considerations as well. For example, the variety of data sources accessible via DV is very high, while for loosely coupled federation it can be very low, often limited to one vendor's products. This extends to Hadoop, where access is usually limited to one or two distributions and is often accomplished by querying via Hive, which further constrains query complexity, response time and data availability.

There is rarely a single answer to data access because workloads mix conflicting requirements. It is very often a case of using multiple tools depending on the circumstances, for example mixing DV for repetitious data access with QueryGrid for the ad-hoc and at-scale requirements.

Nine Key Factors to Evaluate for a Scalable Data Fabric

The goals for distributed data access in an analytical ecosystem are clear: access must be on-demand, easy

enough for an analyst to use, and scalable enough to not be the bottleneck. It needs to work across databases and Hadoop, and work for both reading from remote data sources and moving data between remote systems. That's not a simple set of goals.

Whether you are assembling a set of components to build your own solution or evaluating third-party products that purport to address the problem, the general requirements are the same. Here are the top 9 items to evaluate when designing a solution or choosing a product.

Scalability



Scalability is an obvious item, but evaluating it properly is difficult. Data access must be parallel, which in a distributed environment means that a query should take advantage of parallelism available both where the query was initiated and at the source system.

While this might seem obvious, most data access solutions are not fully parallel. They often funnel requests to a single processing thread or they use a single network channel between systems. It is not uncommon to find systems that are designed assuming one side of a connection is single-threaded. For example, many connectors for Hadoop take advantage of parallelism with the Hadoop cluster, but the connector itself narrows to a single process when returning the data.

It is vital that both sides of a data access connection, as well as the transport used by the connection, be parallel. If any part of the connection from initiation through transport to retrieval is not parallel then it will be a bottleneck that severely limits performance and scalability.

The transfer of data between two systems should also be cluster-aware. For example, if a Hadoop cluster has 10 nodes, and a database has 5 nodes, then access from the database should use the appropriate degree of parallelism; perhaps 5 processes on the database nodes, each communicating to 2 of the 10 Hadoop nodes, using a total of 5 network links to move the data. This manages the resources to the level of the limited system, in this case the database. If the cluster size were reversed, the communication would be reversed as well. However, it's never this simple—for example some Hadoop nodes may have splits and some may not, so the software has to determine this at a finer grain of parallelism.

Other questions also come up with parallelism: How much is too much? How do you define it? How do you manage it as load fluctuates? Parallelism is easy when there's one request, but multiple requests have to queue or resources have to be divided between queries. Since cluster sizes can change, this parallelism can't be a fixed configuration, but must be determined as each request arrives.

Performance and scalability come from the architecture, but to get both in a data access framework requires an optimizer that can make decisions about where to do operations, when to reorder them and shift work across platforms, and how to manage parallel operations to avoid local and remote bottlenecks.

These are the types of design criteria that should be tested to ensure the solution scales appropriately for the data volumes and resources available.

Efficiency



A common belief with data today is that resources are cheap, so we can just add more. However, this brute force approach rarely works in real-world situations. When data volumes are large, small amounts of overhead add up and can disproportionately consume processing or transfer time.

Any solution should be smart enough to move only the data that needs to be moved. This is what is entailed, for example, when filtering data or eliminating partitions at the source before transferring it over the network. (This is usually called a "pushdown operation.") Data reduction should always be applied early, which means that the system ought to have an optimizer that can reorder operations in a user request.

There are, however, other important efficiency considerations. For example, resources may not be cheap, depending on where the data lives. If there are excess resources on one side of a transfer between two systems, then operations can be shifted to the resource-rich side of the transfer. This is ideal for work that involves format or code conversions. In some cases, particularly over lower-bandwidth network links, it's worth using extra CPU to encode the data for efficient wireline transfer, or even to compress data prior to transfer and decompress at the other end.

Ideally, the system should be able to detect when this is worthwhile and do it automatically. Due to complexity, most open source tools and custom code do not attempt to optimize operations, instead requiring that a developer manually configure this as part of a data pipeline.

Concurrency



Concurrency is a critical feature of a data warehouse system that is designed to support BI use cases. The good news is that exploratory use cases that join large datasets are not so common that every analyst will be doing them—at least, not at the same time. Even though concurrency doesn't need to be as high as that needed for a BI system, it should be more than one user query at a time. Analysis systems are not single-user systems.

As noted in Figure 2, concurrency varies with dataset size. The larger the query, the lower concurrency can be. When testing a solution, try a mix of large and small data movement tasks and see how they perform as the number of tasks increases.

Topology of Systems and Connections




A user needs the ability to initiate a query from any system that is part of the ecosystem. The limitation of many vendor-supplied data access solutions is their unidirectional and asymmetric approach. They can only initiate queries from their environment and access data from other systems, a common case with database links and federation. This forces analysts to work inside only one environment, even if it's not the best one for their work.

Access should be the same no matter where a query is initiated. If a user is working in Hadoop, then they should be able to initiate their request from that environment. If a user is working on a Teradata Database, then they should be able to initiate their request from that database. For example, an analyst might have machine log events stored in Hadoop and warranty claims stored in a Teradata Database. If they are looking to correlate that data, then they might start in the Hadoop environment and run an analytical model. Later, if they know what data is useful, they can simply query that data from Hadoop via a SQL-based query tool running against the Teradata Database.

The access or transfer of data should also be bidirectional so that an analyst can work with data locally and send it to a remote system, or join data from a remote system with data on the local system. The ideal situation is for the user to always feel as if they are at the center, no matter what platform they are working from and where the data is stored.

Related to this is the quality of the connectors to different platforms. In many cases, they are just JDBC connectors to remote systems despite being one of the most critical components to data transfer performance. You want connectors that operate in parallel, bidirectionally, with the same degree of parallelism as the platforms they run on. They, or their management framework, have to address resource sharing across multiple queries, connection pooling and automatic data conversion.


Data Conversion and Type Management

 One of the problems when dealing with multiple platforms is that they can each store data in slightly different formats. An analyst using data from another platform has to be aware of the differences and convert formats and data types based on the system they are using. This is cumbersome when done manually and often leads to errors; one common example is partial joins that result when certain data types doesn't align.

The ideal situation for a user is to have a common format (and data types when dealing with typed fields) that all of the connected platforms map to. In this way, there is no confusion about the data one is working with because the formats and types all look the same.

Databases use implicit type conversion between varying types of data internally. Data access middleware needs to do the same thing, but across a more varied range of data types. To do this requires implicit type conversion of data from each source system to a common representation so that data can be transparently moved from one platform to another. Doing this automatically prevents users from making obvious mistakes. The ability to override the conversions may be needed for expert users, but this should be the exception rather than the rule.

Support for Platform-Specific Capabilities


 While we put a lot of emphasis on abstracting away from underlying platforms so they all appear alike, the platforms do not have identical capabilities. Some are good at scanning on cheap,

large-scale storage. Others are good for ad-hoc query on large data volumes. Others are good for running complex analytical models.

Analysts need access to underlying capabilities in the platforms. For example, it should be possible to execute commands in the native syntax of a remote system, doing computational work there, and then transferring the results. The local system where the user issued the query may not understand this syntax, but it can still send that work to the remote system for execution. This is called “pass-through” in data virtualization tools.

Likewise, analysts need to be able to take advantage of local system capabilities that may be unavailable on the remote system. For example, they might join remote data with local data and then execute an analytical model on that data using functions only available on the local system. The ability to do useful analysis work can be severely curtailed without this type of pushdown capability.

Security

 Security is another reason the data access solution needs to be abstracted from underlying platforms. You want security to be policy-driven rather than manually configured in point-to-point fashion. For example, it's not useful when a system on one side of a transfer enforces encryption if the system on the other side doesn't or can't do the same. Building this into the data access layer protects data from each point in the system.

If security is tied to the platforms rather than the data access layer, then enforcement of policies is left to individual developers and administrators who configure the software. This has the effect of putting more work on technical staff since they will have to configure connections to each system and deal with the exceptions. An ad-hoc approach leads to gaps in security because one misconfigured platform can permit access to other platforms.

If security is part of the data access software and independent of the underlying platforms, then it's possible to have policy-based security that is not constrained by any one platform. Security should be configured once based on systems, data and teams. This also makes it easier to give users access to data—the access can be controlled by one set of policies rather than requesting access from many different administrators.

Monitoring and Administration



A key factor that is neglected (or under-developed) in most custom-built data access environments is monitoring and administration. In part, it's because the access is built more as data pipelines to supply data to one place or application. Enabling ad-hoc access to data for analysts is a more generic problem, albeit one with more complex administration needs.

There are many layers to this: logging and monitoring of activity on remote systems, monitoring on the local system, tracking information for troubleshooting and tuning, observing and controlling platform and network resources, and throttling activity on local or remote platforms to balance the workload. For a multi-user data access fabric, these are vital or one bad query can bring multiple platforms to a standstill.

These Factors All Contribute to the Overarching Goal, Ease of Use

The biggest challenge for an analyst is the technical complexity when dealing with multiple platforms. To make it easier, the data access solution needs to hide that complexity. Using a single SQL dialect that is available from any connected platform and making all sources appear as tables partly achieves this. The user's work (and their knowledge) is then portable if they move to a different platform.

Other elements like automatic type conversion as data is linked or moved between systems, common security credentials and policies, automatic management of parallelism, and optimizing their queries to minimize data movement all contribute to an easier-to-use environment, which is the real goal here.

Teradata QueryGrid™: Solving the Hard Problems of Scalable Data Use

Teradata QueryGrid™ is designed to make it easy for a user to query and move data between the systems in their analytics environment. It abstracts the complexity of linking data from different types of systems so they see the data without worrying about the underlying platforms. The challenges of linking, accessing and moving data are hidden from view. This allows them to store and process data on the platforms best suited to their needs and to access the data from those platforms when needed.

More importantly, QueryGrid is designed to be as scalable as the platforms it connects to. If a query from one parallel system, like Hadoop, accesses data from another parallel system, like a Teradata Database, QueryGrid will use the full resources available from each system. It is designed to be an independent middleware component for high-speed, large-scale data access that won't become the bottleneck between systems.

QueryGrid is similar in concept to federation, but with a few key differences that put it into a different category. Like federation, QueryGrid uses SQL as the common language for data access. Unlike federation, QueryGrid extends user queries by allowing the use of native functions available on remote systems, enabling one to push processing to data on a remote system and join the output of that process with data on a local system. QueryGrid supports bidirectional data transfer, so local data can be moved to a remote system and vice versa. This is generally not possible with simple federation.

Unlike loosely coupled federation, which typically has point-to-point connections, or DV, which involves a central hub through which the data flows, QueryGrid is a fabric of bidirectional connections. When a system is added to the fabric it becomes accessible to all other systems, regardless of type. Any data on that local system is then usable from any other system, subject to security policies set at both the QueryGrid and the local system level.

QueryGrid is designed to be as scalable as the platforms it connects to.

These differences are due to the design of QueryGrid. Loosely coupled federation is most often a feature that is built into an underlying database or application. It is a technology from the client-server era and embeds a notion of one place from which to initiate a request. Data virtualization operates as a central server through which data must flow from other servers.

Neither of these is an architectural match for a distributed parallel environment or cloud computing. Software architectures to access data must now meet the needs of

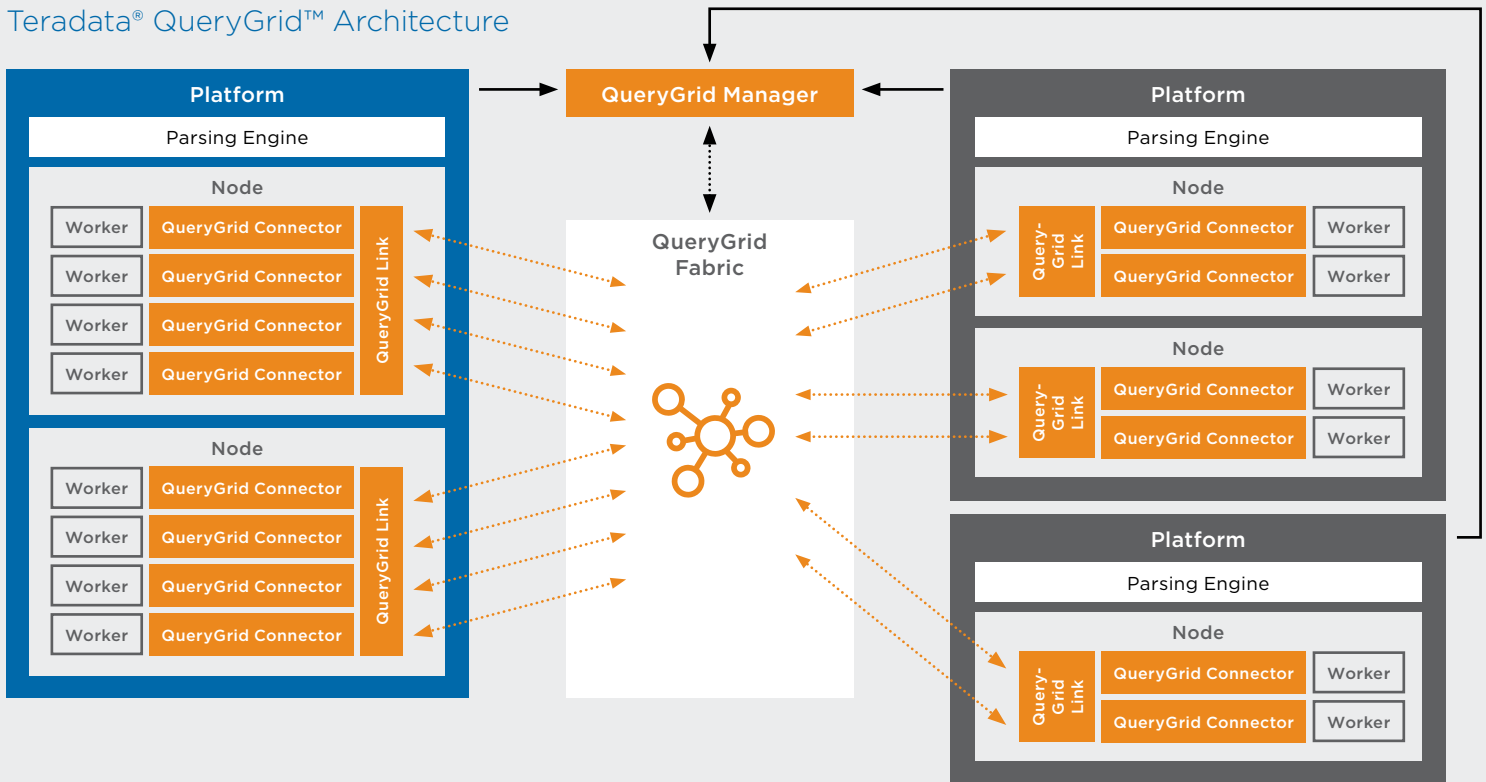


Figure 5. Teradata QueryGrid architecture. Every platform has several components installed: The parsing engine is what receives and parses a query from the user. The QueryGrid connectors provide basic services to each worker in the platform (e.g. security, data conversion, data transfer). The QueryGrid fabric spans all the connectors, enabling one platform to communicate with other platforms and the QueryGrid manager. The manager is responsible for configuring QueryGrid on each platform, collecting metrics from nodes and the network, and monitoring all query and data transfer activity. A fabric is a collection of platforms that can communicate with one another.

large-scale data and compute platforms that may be on separate networks, but hide the underlying complexity from users and developers alike.

The previous sections described design principles and important factors to consider when evaluating solutions for distributed data access. To solve them requires understanding how to engineer solutions to several challenges: scalability in distributed systems, the right level of abstraction for users and making a system that is maintainable without significant developer or administrator involvement.

Before going into more detail, it's worth looking at the high-level QueryGrid architecture to understand how it is designed to avoid the client-server scalability traps of other technologies.

QueryGrid Architecture: How It's Built

This section provides details about the design of QueryGrid to show how it addresses the challenges of parallel data transfer between platforms that are themselves distributed computer clusters. The components in QueryGrid are shown in Figure 5, QueryGrid architecture.

General Design

QueryGrid uses a modular architecture that is designed to provide independence from the underlying platforms. Components that provide shared services implement the same features regardless of platform, so features like authentication, encryption, network transport protocols, data conversion and data transfer are handled in the same way. This guarantees that different types of systems behave in the same way, abstracting the data access and movement problem into a common set of APIs.

QueryGrid creates the concept of a “fabric”—a collection of platforms that can communicate with one another to transfer data. Once a platform is added to a fabric, it can instantly see all other platforms in that fabric. In most federation and custom-integrated environments, this is not the case—the visibility and data flow are unidirectional and must be configured with each pair of platform connections. Point-to-point connection models are unwieldy because each new platform adds an increasing number of pairwise connections to configure and maintain.

The idea of a fabric avoids one of the other problems with most custom data access work, as well as with the server-centric model of federation: forcing a user to work from one primary platform. In the fabric, any platform can access and transfer data from any other platform. The ability to access data is symmetric, subject only to the limitations of the local platform the user is working on.

For example, a user might use Hadoop to join local event data with transactions in a remote Teradata Database, or query from the Teradata Database and join a local table to event data stored remotely in Hadoop. An analyst can work from the environment that makes the most sense for them, not what is technically convenient for administrators who have to copy data between systems first. The data is presented as SQL-accessible tables abstracted above the underlying platforms.

Multiple fabrics are possible in QueryGrid, each managed independently. This allows an administrator to link different systems for different uses, or to create categories of fabric with different resource limits. For example, a fabric for ad-hoc use by analysts might be different than one used to support analytical model execution. This is an important feature for enterprise environments, yet absent from most alternatives.

Parallel Architecture

If your goal is to support people exploring and analyzing data, then you don't want them to wait for hours for data. QueryGrid is designed as a parallel architecture from end to end so that it won't be the bottleneck in data access. There is no primary platform from which queries are run, as there is in client/server-style federation, nor is there a central server through which data must flow.

Parallelism is one of the hardest elements to get right in distributed systems. Most of our existing integration tools were developed assuming that SMP servers communicate across a single network link. In today's environment we are dealing with clusters of nodes. Unlike the server-based world, cluster sizes can vary greatly, meaning that data throughput and network bandwidth use also vary greatly.

Users want their data fast, but not at the cost of overwhelming shared environments, creating difficult design

challenges. QueryGrid must be aware of the units of parallelism available for use in each platform that is attached to a fabric. It's easy to move data from same-sized clusters, but not when moving from a large cluster to a small cluster. Depending on the degree of parallelism, it's possible to overload one cluster with data from another, or saturate the network links between them.

Another challenge in design is the fact that cluster sizes are not static. Resources can be added and removed, sometimes quickly. It is very hard to build software that can introspect the environments it is connected to and change its configuration. Most products and custom solutions are statically configured, so when resources change, an administrator must update the configurations or they will be out of date. QueryGrid is not static.

The volume of data and its distribution also change over time. When combined with queries that filter some data, any approach based on static configuration puts too much of a burden on the user. For example, some solutions to data transfer make the user or developer specify manual data partition management rather than determining that information at runtime.

Network Awareness

Network bandwidth is the scarce resource with this type of work, not server resources. Network bandwidth limits are often overlooked by developers who build custom software to move data back and forth between clusters. With a single user or query, this can be managed. Problems build up when one introduces concurrency. Multiple users or queries will either compete

for resources on the clusters, or they will consume all the available network bandwidth.

Some platforms can read or write data faster than others, so the middleware has to be aware of not just parallelism, but throughput and resource consumption. Resource use is further complicated by cluster placement, for example using a local database to access data stored in Postgres running in a cloud, which implies not a high-bandwidth LAN between the two platforms but (probably) a lower-bandwidth WAN. QueryGrid allows an administrator to set policies that address these problems, for example to set resource limits that are different for different platforms.



Optimization

QueryGrid was designed specifically for the kinds of problems that appear only when one grows into large-scale data environments and cluster-based platforms. In these environments, addressing efficiency of queries and data transfers grows in importance. Small changes in where data is filtered or aggregated can vastly improve response time.

An optimizer separates the logic of data access from its physical execution. When done properly, an optimizer can introspect the metadata available for local and remote datasets to rewrite queries, change the order of operations, and adjust the physical execution of queries and subqueries. In manually configured or static environments, like those in many Hadoop environments and virtualization tools, a developer must make these decisions for each query. Most analysts do not have this level of expertise.

When parsing queries, QueryGrid takes into account many of the elements described earlier. It can push tasks like filtering and aggregation to a remote platform and manage partition elimination so that less data moves across the network. It can push a projection to a remote platform when possible, so only the columns required are returned, rather than an entire row that includes unused columns (important when there are hundreds of millions of rows). Based on resource profiles, it can do implicit data type conversions on local or remote platforms, taking advantage of the platforms with more resources available.

A problem with most of today's big data platforms is that the statistics about datasets are inaccurate or unavailable. Querying a file may return one million rows or 100 billion rows depending on the filters. The only way to know is often to run a query and see what happens. The alternative is to gather stats on the data first, which might take as long as running the query.

QueryGrid is unique in offering a third option: when executing from some platforms, it has an adaptive optimizer that gathers statistics on the data being processed as a query executes. If data is larger or smaller than expected, or is skewing, which means it overloads some workers while leaving others idle, the optimizer can change the execution plan for the running query. This allows a user to query without worrying as much about what might happen.

QueryGrid Link and Connectors

To provide parallel data access, QueryGrid installs local components that communicate with the fabric onto every node in a cluster. These components provide an equivalent interface to every platform so that they all have the same set of services available. Every node has a link that enables communication, transfer and monitoring, as well as a connector for every worker process.

For example, every mapper in a Hadoop cluster running Hive has a connector. This means that there is the same degree of parallelism available to QueryGrid as there is in the platform it is installed on. Many cross-platform data integration solutions use a static configuration to define the units of parallelism, or base the parallelism on partitions in the data. Data distributions can change over time, requiring periodic reconfiguration on the part of administrators or developers. If this is not done then performance degrades. By aligning parallelism dynamically with workers, QueryGrid avoids this flaw.

The connector used by QueryGrid is not a simple JDBC connector, common in many software packages and open source projects for remote communication. It addresses a number of difficulties related to efficiency and performance. Instead of using disk-based mechanisms for data transfer to or from remote systems and local storage, it uses shared memory to transfer the data.

One feature that affects resource efficiency is the internal row conversion API that allows QueryGrid to automatically map data types in local storage to a common representation used on all platforms. By using more efficient representations of data than raw JSON, for example, it's possible to move the same data using far less memory and network bandwidth.

The more important aspect of automatic type conversions is that the user does not need to be aware that types are conformed across systems, or how to do it. Without this feature, simply moving a query that joins data in Hive and Cassandra from Hive to a Teradata Database would require rewriting that query to recast all the data types. This is something users often do not think about, unless they are highly technical. Even then, mistakes are common, particularly with problematic data like DATETIME columns and various Unicode string encodings.



As with a database, it is still possible for the analyst to override type conversions if they want to change how a data element is treated. The connectors and links are designed to support flexibility when working across multiple platforms. The goal is to hide complexity, but provide transparency when desired.

Another example of this transparency is the ability to send commands in the syntax of the remote system from where the user is working. A user on a Teradata Database can send a Cassandra-specific query. It is this type of flexibility in pass-through query processing that makes QueryGrid well-suited to exploratory and analytics uses. Data can be processed in place on the most appropriate remote engine, joined with data elsewhere and brought back to the local platform.

QueryGrid Manager

The QueryGrid Manager runs in a separate environment, independent from the platforms it connects. This enables monitoring and administration of the full ecosystem without concern for what happens when one platform is down. Many do-it-yourself monitoring solutions encounter problems where the monitor and the platform monitored run in the same environment or depend on the same event logging infrastructure. The cloud-native pattern that separates these monitoring functions is uncommon in federation products.

Another cloud-native pattern is providing an always-on system. With many solutions, zero-downtime upgrades are difficult to impossible. QueryGrid has the ability to configure more than one version of the software at one time, install or upgrade software on platforms, and roll the upgrades back.

It facilitates rolling upgrades both within a cluster, as well as across platforms, using this multi-version capability. For example, all the nodes in a Hadoop cluster can use one

version of QueryGrid while a second version is running in the same cluster for testing. The same can be done across platforms in the same fabric. The manager can add and remove platforms in a fabric dynamically without reconfiguring connectivity on all participating platforms, as one must often do with point-to-point platform integration.

In a distributed environment, each platform is like an island, using its own interfaces for administrative tasks. Monitoring is usually handled on a platform-by-platform basis. The problem for a user or administrator is that distributed data access is not tied to one system. A user running a query that joins data across platforms must be able to monitor what is happening on each platform and see a picture of the global state: how much data is being read from each source system? From each worker on that system? What are the performance and resource consumption metrics for each? How much data is moving across the network? How much data was returned?

Monitoring is one of the key features needed to make data exploration and analytics development easier, and to make operations easier. The monitoring environment in QueryGrid is so sophisticated because the same metrics it gathers for management are useful to users who want to know what's happening with their work, developers who want to optimize performance and administrators who want to keep their systems running smoothly.

Conclusion

Data access fabrics that can scale to the extreme data volumes of distributed systems are a new and evolving category. As should be apparent, building parallel-aware software to act as a transparent data interconnect for heterogeneous systems is not a simple task. It isn't as easy as integrating open source components that were designed in isolation for specific tasks. Nor is it as simple as retrofitting a product designed for a different environment.

Building a parallel-aware transparent data interconnect is a daunting but essential task. It's essential because the problem of on-demand data access, while not new, does have new urgency. Like it or not, we're in the midst of a transition away from the traditional, IT-controlled governance regime to a new regime that aims to strike a balance between the business' need for self-service and the organization's need to impose reasonable limits on what the people do with data.

It behooves you to do something to accommodate this transition because analysts expect to be able to do more and different things with data, starting with access to data on-demand. IT is expected to support and promote the efforts of users. It was different in the old model, with little choice but to go to a single, centralized data warehouse to get the data they needed. Ultimately, restriction, not permission, was the default posture.

Now more than ever before, IT needs a scalable, parallel-aware data fabric because the more than one system stores data that analysts need. Users want access to data whenever and wherever they need it. IT needs a solution that strikes a balance between its responsibility to cater to user expectations (for data access, for the freedom to transform, use, and share data) and its duty to enforce a responsible governance regime.

A solution such as Teradata QueryGrid offers the equivalent of a data fabric that facilitates on-demand, self-initiated access for analytical and exploratory uses. QueryGrid makes it possible for organizations to enforce a reasonable governance regime – even in the context of self-service, exploratory analytics.

When deciding whether to assemble your own fabric or to evaluate competing alternatives, it's worth spending the time to understand the deeper challenges and

considerations outlined in this paper. QueryGrid is the first product in this category that is designed specifically to address high-scalability considerations. As such, if you are a Teradata customer, QueryGrid should be on the list of technologies to evaluate. Again, it behooves you to act: to do something to improve the current data environment.

About the Author

Mark Madsen is a research analyst focused on analytics and information management. Mark is an award-winning architect and former CTO whose work with large-scale data infrastructure and analytics has been featured in numerous industry publications. For more information, or to contact Mark, visit ThirdNature.net.

About Third Nature

Third Nature is a research and consulting firm focused on new business practices and emerging technology for analytics and information management. The goal of the company is to help organizations learn how to take advantage of new information-driven management practices and applications, offering consulting, education and research services to support business and IT organizations and technology vendors.

About Teradata

Teradata empowers companies to achieve high-impact business outcomes. With a portfolio of business analytics solutions, architecture consulting, and industry-leading big data and analytics technology, Teradata unleashes the potential of great companies. Visit teradata.com.

10000 Innovation Drive, Dayton, OH 45342 Teradata.com

QueryGrid is a trademark, and Teradata and the Teradata logo are registered trademarks of Teradata Corporation and/or its affiliates in the U.S. and worldwide. Cassandra, Drill, Hive, Impala, Spark, and Sqoop are trademarks, and Apache and Hadoop are registered trademarks of the Apache Software Foundation. Cloudera is a registered trademark of Cloudera. Hortonworks is a registered trademark of Hortonworks. MapR is a registered trademark of MapR Technologies, Inc. Amazon Web Services and AWS are trademarks or registered trademarks of AWS in the U.S. and/or other countries. Microsoft and Azure are registered trademarks of Microsoft Corporation. Teradata continually improves products as new technologies and components become available. Teradata, therefore, reserves the right to change specifications without prior notice. All features, functions, and operations described herein may not be marketed in all parts of the world. Consult your Teradata representative or Teradata.com for more information.

Copyright © 2017 by Third Nature Inc. All Rights Reserved. Produced in U.S.A.

08.17 EB7271

