Distributed/Heterogeneous Query Processing in Microsoft SQL Server

José A. Blakeley Conor Cunningham Nigel Ellis Balaji Rathakrishnan Ming-Chuan Wu
Microsoft Corporation
One Microsoft Way, Redmond, WA 98052

Abstract

This paper presents an architecture overview of the distributed, heterogeneous query processor (DHQP) in the Microsoft SQL Server database system to enable queries over a large collection of diverse data sources. The paper highlights three salient aspects of the architecture. First, the system introduces well-defined abstractions such as connections, commands, and rowsets that enable sources to plug into the system. These abstractions are formalized by the OLE DB data access interfaces. The generality of OLE DB and its broad industry adoption enables our system to reach a very large collection of diverse data sources ranging from personal productivity tools, to database management systems, to file system data. Second, the DHQP is built-in to the relational optimizer and execution engine of the system. This enables DH queries and updates to benefit from the cost-based algebraic transformations and execution strategies available in the system. Finally, the architecture is inherently extensible to support new data sources as they emerge as well as serves as a key extensibility point for the relational engine to add new features such as full-text search and distributed partitioned views.

1. Introduction

The rate at which data, individual and corporate, is being generated in diverse data sources is currently much higher than our ability to collect it, catalog it, and organize it inside database systems. Personal data is generated and stored in multiple, diverse sources (Word, Excel, mail messages, digital pictures) with no easy mechanism to collect and organize them in a single database. At a corporate level, acquisitions and the continuous creation of business relationships and partnerships forces organizations to create an infrastructure that allows access to mission-critical data stored in distributed and heterogeneous data sources. Modern data intensive applications must adapt to the inherent distribution and heterogeneity of data. Therefore, it is no longer optional for modern database management systems to provide efficient, built-in capabilities to query and update heterogeneous and distributed data. We need data management systems that can provide efficient and flexible access to diverse data sources. This paper presents the architecture of such capability in the Microsoft SQL Server database system. The salient features of this architecture are:

- A provider model based on the OLE DB data access interfaces. OLE DB is a broadly deployed, industry standard API that enables access to a very large set of relational and non-relational data sources [2][3][4][5]. OLE DB includes interfaces that enable the exposure of data source capabilities such as query, indexing, and statistics which permit the query optimizer to decide how much computation can be pushed to the remote data sources vs. executed locally.
- Distributed and heterogeneous query and update capabilities are natively built into the query processor. This means all execution plans are generated from cost-based decisions using a rich set of algebraic transformation rules. This enables the system to decompose the original query into sub-expressions that can be pushed to the participating data sources when their capabilities allow it and it is cost-effective.
- The architecture is inherently extensible in two key dimensions: (a) It supports new data sources as they emerge. It suffices to build an OLE DB provider that exposes the capabilities of the data source and the new provider can be “plugged-in” to the DHQP system. (b) It enables the extension of the capabilities of the server itself. To date, the full-text indexing and distributed partitioned view features of the system have been built as extensions to the DHQP.

Another aspect of the framework (not covered in this paper) is tools support. The Microsoft Foundation Class Library includes a set of templates that facilitate the construction of OLE DB providers [16]. Assuming a system programmer is familiar with the semantics of the OLE DB API, it takes a modest development effort to build a working provider pluggable to the DHQP system using these templates.

The rest of the paper is organized as follows. Section 2 introduces the system architecture and some real world applications of SQL Server DHQP. Section 3 describes the key OLE DB interfaces to enable DHQP. Section 4 presents the distributed

query processing infrastructure including optimization and execution. Section 5 presents related work and Section 6 concludes the paper.

2. Architecture and Scenarios

Figure 1 illustrates the general architecture of the SQL Server distributed/heterogeneous query processing system. In this architecture all data sources, including SQL Server, expose their capabilities and data rowsets through the OLE DB interfaces.

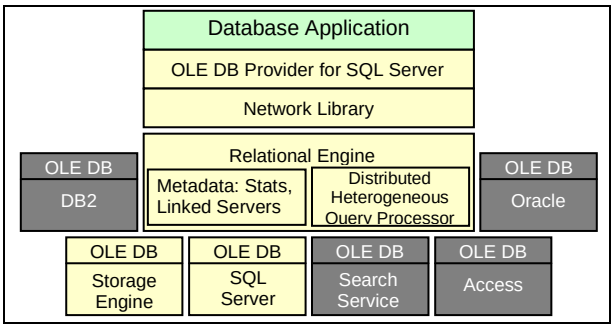


Figure 1 Distributed/Heterogeneous Query Processing Architecture

It is conceivable to consider building a similar system using other widely used data access interface such as ODBC or JDBC as the "plug-in interface" to other data sources. The key benefit of OLE DB over ODBC/ JDBC is that the latter were designed fundamentally as interfaces to SQL data sources whereas OLE DB is factored to expose only the native capabilities of the data source. OLE DB can access both SQL and non-SQL data sources such as ISAM, search engines, personal productivity tools, e-mail systems, and so on. In addition, OLE DB is the interface used by SQL Server to access its local storage engine, thus the code patterns to access data from local and external sources are almost identical. SQL Server uses the *Microsoft Distributed Transaction Coordinator* [15] to ensure atomicity of transactions across data sources. The following subsections show example applications leveraging SQL Server's DHQP.

2.1. Distributed SQL-to-SQL Queries

SQL statements can reference local and remote SQL Server tables and heterogeneous data sources in the FROM clause via "*linked servers*". Linked server names associate a server name with an OLE DB data source. These objects are referenced in SQL statements using the *four-part name* convention: <linked-server>.<catalog>.<schema>.<object>. For example, if a linked server name of DeptSQLSrvr is defined to reference a remote SQL Server machine, the following statement references a table on that server:

```
SELECT *
FROM DeptSQLSrvr.Northwind.dbo.Employees
```

This technique can be used to create and reference a wide range of data sources including other relational sources (e.g., Oracle, IBM DB/2, Microsoft Access) as well as other tabular data sources (Microsoft Excel, text files, or other third-party data sources). SQL Server also supports a technique to reference remote sources in an ad-hoc manner. Details of ad-hoc connections are beyond the scope of this paper.

For each data source accessed as a linked server, an OLE DB provider for that source must be present on the machine running SQL Server. The set of SQL operations that can be used against a specific OLE DB data source depends on the capabilities of the OLE DB provider. For SQL-capable data sources, the DHQP can "push" SQL statements containing joins, restrictions, projections, sorts, and group-by operations to them when it is cost-effective.

2.2. Heterogeneous SQL-to-File System Queries

The Microsoft Search Service supports full-text searches over file system data. Various document formats are supported for inclusion in full-text indexes. There is an OLE DB provider for Microsoft Search which allows SQL Server's DHQP to query documents and files stored in the file system. To enable this capability, users need to setup a full-text catalog/index first by activating the *index service* on the Windows machine and creating a new catalog/index over the directory containing the documents to be full-text indexed.

In the following example, a catalog named DQLiterature is created over a document repository containing various document types, e.g., MS Word, MS PowerPoint, PDF, ZIP files, etc. For all third-party document types, one needs to install necessary *IFilter*s to enable full-text search. The IFilter is an interface for retrieving text and properties out of documents. It provides the foundation for building higher-level applications such as document indexers and application-independent viewers [14]. The following query retrieves all the documents about "parallel database" or "heterogeneous query":

```
SELECT FS.path
FROM OpenRowset('MSIDXS','DQLiterature';'';'',
        'Select Path, Directory, FileName, size,
        Create, Write from SCOPE() where
        CONTAINS(''"Parallel database" OR
        "heterogeneous query"'')') AS FS
```

2.3. Full-text queries on relational data

SQL Server leverages the full-text index and query capabilities of the Microsoft Search Service to provide a full-text search capability over textual data

and documents stored in SQL Server tables. These full-text indexes are stored outside of the database engine and queried using SQL Server DHQP. Figure 2 illustrates the general architecture of this integration. The SQL syntax was extended to enable content-based predicates and the DHQP leverages the index service to derive efficient query plans. With the full-text SQL extension it is possible to find *related* words by searching over word stems. For example, 'runner', 'run', and 'ran' can all be equivalent in full-text searches.

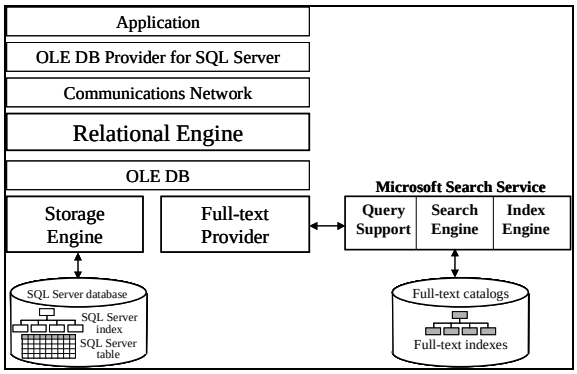


Figure 2 Integration of a full-text component with a relational DBMS via OLE DB

There are two aspects to full-text support: index creation and maintenance, and query support. Indexing support involves creation, update, and administration of full-text catalogs and indexes defined for a table or tables in a database. Query support involves processing of full-text search queries. Given a full-text predicate, the search service determines which entries in the index meet the full-text selection criteria. For each entry that meets the selection criteria, the query component of the search service returns an OLE DB Rowset containing the identity of the row whose columns match the search criteria, and a ranking value. This rowset is used as input to the query being processed by the SQL relational engine just like any other rowset originating from tables or indexes inside the server. The relational engine joins this rowset with the base table on the row identity and along with other predicates in the query evaluates the execution plan that yields the final result set. The types of full-text queries supported include searching for words or phrases, words in close proximity to each other, and inflectional forms of verbs and nouns.

2.4. Heterogeneous SQL-to-Email Queries

Consider a salesman who wants to find all email messages he has received from Seattle customers, including their addresses, within the last two days to which he has not yet replied. This query involves searching the mailbox file containing the salesman's email, as well as a Customers table stored in an Access DBMS to identify customers. Microsoft DHQP plus OLE DB enable the development of an application that will access both information sources and assist the salesman to answer this query, which can be formulated in an extended SQL syntax as follows:

```
SELECT  m1.*, c.Address
FROM MakeTable(Mail, d:\mail\smith.mmf) m1,
      MakeTable(Access,
      d:\access\Enterprise.mdb, Customers) c
WHERE m1.Date >= date(today(), -2) AND
      m1.From = c.Emailaddr AND
      c.City = "Seattle" AND
      NOT EXISTS (SELECT *FROM
          MakeTable(Mail, d:\mail\smith.mmf) m2
          WHERE m1.MsgId = m2.InReplyTo);
```

MakeTable is a table-valued function that transforms the mail file (d:\mail\smith.mmf) into a stream of rows, each representing a message. It also exposes the Customers table from an Access database (d:\access\enterprise.mdb). The function date() takes a date and a number of days as arguments and produces a date.

3. OLE DB: A Component Data Model

OLE DB is designed as a set of Component Object Model (COM) [6] interfaces. This section provides a brief overview of OLE DB, focusing on the pieces most important to building a DHQP. For details about OLE DB, please refer to [2],[3],[4] and [5].

3.1. Common abstractions

OLE DB is built on a small set of common abstractions, along with a set of common extensions that enable access to diverse data sources. By building on a set of common abstractions, OLE DB enables generic components to operate on those abstractions as individual entities without needing to know how the entities were generated or the details of what they represent.

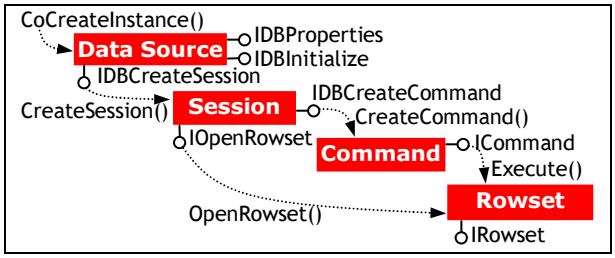


Figure 3 Hierarchy of Data Source, Session, Command, and Rowset Objects

OLE DB defines three object classes common to any data provider; the *Data Source Object*, or DSO, provides a common abstraction for connecting to the data store, the *Session* object provides a transactional

Table 1 languages supported by various OLE DB providers

Type of Data Source	Product	Query Language
Relational	Microsoft SQL Server	Microsoft Transact-SQL
Full-text Indexing	Microsoft Index Server	Index Server Query Language
OLAP	Microsoft OLAP Services	MDX
Email	Microsoft Exchange	SQL with hierarchical query extensions
Directory Services	Microsoft Active Directory	LDAP

scope for multiple concurrent units of work, and the *Rowset* object provides a common abstraction for exposing data in a tabular fashion, as shown in Figure 3. All of the extensions built on OLE DB extend this basic object hierarchy. For example, the *command* object, discussed later, enables query requests on query-capable providers.

3.1.1. Connection abstraction

The OLE DB connection model defines how data providers are located and activated. The *data source object* (DSO) and the *session* object are the basis for the OLE DB connection model. To access a data provider, a consumer must first instantiate a DSO. The DSO exposes the **IDBProperties** interface, which the consumer uses to provide basic authentication information, **IDBInitialize** establishes a connection with the data store, and **IDBCreateSession** creates a session object which supports a simple interface **IOpenRowset** to access rowsets.

DSO supports interfaces used by DHQP to query the *capabilities* of remote sources. These capabilities include the interfaces, rowset properties (e.g., scrollability), transaction properties (e.g., isolation levels), SQL dialects, command operations (e.g., left-outer joins, text search operators), and security options.

3.1.2. The Rowset

A rowset is a unifying abstraction that enables OLE DB data providers to expose data in tabular form. Conceptually, a rowset is a multi-set of rows where each row has zero or more columns of data. Base table providers present their data in the form of rowsets. Query processors present the result of queries in the form of rowsets. This way it is possible to layer components that consume or produce data through the same abstraction. Rowsets are also used to return metadata, such as database schema, supported data type information, extended column information and statistics. For details about common abstractions of OLE DB, please refer to [2].

3.2. Common extensions

In order to be suitable as a primary interface to DHQP, OLE DB defines a set of common extensions on top of the base abstractions. The major extensions include *query support*, *ISAM navigation*, *hierarchical navigation*, *heterogeneous data support*, *statistics*, and *OLAP support* [2]. By building these extensions on the common abstractions, generic consumers can view all data through the common abstractions while special purpose consumers can access the domain-specific functionality through common extensions. In the following subsections, we will briefly describe the DHQP related extensions — query syntax support, ISAM navigation, heterogeneous data support and statistics.

3.2.1. Query syntax support

This extension exposes the ability to issue a textual command to a data store. The command object encapsulates the functions that enable a consumer to invoke the execution of data definition or data manipulation statements such as queries or updates against a relational database. Figure 3 depicts the calling sequence.

Since OLE DB is designed to work with any query-capable provider, it does not mandate a specific syntax or semantics for queries. It is entirely up to the provider to define the language for formulating queries. For example, currently there are OLE DB providers for relational DBMSs, full-text search engines, OLAP engines, directory services, email stores, and spatial stores. Each of these query-capable stores exposes a different query language. Table 1 lists some of the query languages supported by various OLE DB providers.

3.2.2. ISAM navigation

ISAM navigation supports basic indexing functionality. This functionality allows services such as query processors to efficiently access contiguous rows of data within a range of keys. Through this extension, SQL Server DHQP can generate query plans during optimization to leverage these remote capabilities. These indexed solutions will be costed and compared with the costs of other solutions before the optimal plan is chosen.

3.2.3. Heterogeneous data support

If results from different data sources share the same schema, rowsets are well suited. However, for dissimilar results, such as e-mail messages, calendar entries, and spreadsheet data, which may contain different columns, a single rowset becomes a limitation.

Table 2 Interfaces of Data Source objects and DB Session objects

COM Object	Interface	Mandatory?	Purpose
Data Source Object	IDBInitialize	Yes	Initialize and setup connection and security context based on linked server properties
	IDBCreateSession	Yes	Create a DB session object.
	IDBProperties, IDBInfo	Yes	Get information about the capabilities of the provider,.
	IDBInfo	No	Get quoting literal, catalog, name, part, separator, character, and so on.
DB Session Object	IDBSchemaRowset	No	Get metadata about tables, indexes and columns.
	IOpenRowset	Yes	Open a rowset on a table, index or histogram.
	IDBCreateCommand	No	Use to create a command object (query) for providers that support querying.

In order to simplify navigation of rowsets containing heterogeneous rows, OLE DB defines a *row object*. Each row object represents an individual row instance, which can be a member of a rowset or a single result value. Consumers can navigate through a set of rows viewing the common set of columns through the rowset abstraction, and then obtain a row object for a particular row in order to view row-specific columns. In addition, hierarchies of row and rowset objects can be used to model containment relationships common in tree-structured data sources via chaptered rowsets.

3.2.4. Statistics

Another supported extension allows remote sources to pass statistical information (including histograms) from remote sources into the optimizer to generate more accurate cardinality estimates over remote operations. This commonly provides order of magnitude improvements on cardinality estimates similar to what is expected in local queries. Histogram Statistics are exposed through an extension to the **IOpenRowset** interface and can be implemented by any provider. Cardinality information is exposed through the schema rowset **TABLES_INFO**.

As a summary, the major OLE DB extensions to DSOs and Sessions used to support DHQP are listed in Table 2. For a complete list of extended interfaces, see [13].

3.3. Categories of OLE DB providers

The set of distributed query capabilities supported against a linked server depends on the capabilities of the OLE DB provider. Three commonly seen providers are briefly discussed below.

Simple provider: A simple provider is an OLE DB provider which supports only the mandatory OLE DB interfaces of being able to connect and retrieve named rowsets. In this case, DHQP provides all of the querying functionality on top of this base provider.

Query provider: A query provider is an OLE DB Provider which supports the ability to query the data source through the **ICommand** interface. If the query syntax is a proprietary syntax, then DHQP supports only pass-through queries against this provider using the **OpenQuery** function [2]. If the provider supports a standard SQL syntax, then it is considered a SQL provider. In this case, the DHQP can support the full SQL query language against remote tables named through four-part names in SELECT statements. The queries that are remoted depend upon the syntax capability of the provider as reported by its **DBPROP_SQLSUPPORT** remote OLE DB property. This property indicates the level of SQL support in the provider as one of: SQL Minimum, ODBC Core or SQL-92 Entry/Intermediate/Full. The DHQP constructs plans such that the provider's capabilities are fully used while not overshooting its limitations.

Index provider: If the provider supports indexes, then the DHQP can generate plans that use these indexes. Index support requires reporting metadata on the indexes (through **IDBSchemaRowset** interface), ability to open OLE DB rowsets on indexes, the ability to seek (or setting a range) on the index for given key values (using the **IRowsetIndex** interface) and the ability to locate base table rows using bookmark values retrieved from the index (using the **IRowsetLocate** interface).

4. Distributed Query Processing

4.1. Distributed query optimization

In this section, we first introduce the Cascades optimizer framework [9]: how different optimization rules interact with one another in different optimization phases, and how *guidance* and *promise* of each rule help to improve the effectiveness and efficiency of the rule engine. Then, we elaborate on how rules specific to distributed queries work and how well they fit into Microsoft SQL Server's cost-based query optimizer making remote data access as easy as access to local data. Finally, we discuss *distributed partitioned views* which are built on top of the functionality of Microsoft SQL Server's distributed query processing to enable efficient query processing in a federated database system.

4.1.1. Search framework

The SQL Server Query Optimizer is based on the Cascades Framework. While this framework has

been discussed previously [9], this section provides a brief overview of the important design characteristics of this framework with respect to distributed and heterogeneous query processing. More specifically, this section discusses the overall approach to representing query operations, how the space of possible plans is searched, and the mechanisms used to find a plan efficiently. Additionally, some examples of specific rules used in SQL Server are presented to demonstrate how the implementation works.

The Cascades framework is a top-down cost-based optimization technique using a rule engine to enumerate possible alternatives of a query for comparison. Unlike some other optimizers, each operator is represented as a unique node in a query tree. For example, "A JOIN B JOIN C" would be represented as two "joins" and three "get" operations instead of a single node containing all joins. Relational operators are sub-divided into the logical and physical ones. "Join" would be an example logical operation, while "hash join", "loop join", or "merge join" would be corresponding physical operations.

Rules are used to search the space of possible plans by matching one logical query pattern (the before-pattern) and introducing a new query pattern (the after-pattern) as a result. Rules are also subdivided into different categories based on their function. *Simplification Rules* perform heuristic tree rewrites, generally early in the optimization process. In this phase, logical trees are rewritten into simpler logical trees. While some optimizers perform a separate heuristic rewrite phase before optimization, Microsoft SQL Server uses the same rule framework as the optimization phase for heuristic rewrites. *Exploration Rules* enumerate equivalent logical alternatives to a query pattern to be considered in the cost-based optimization process. Join commutation (A Join B ≡ B Join A) is an example exploration rule. *Implementation Rules* generate physical alternatives for a particular logical query tree. The implementation of a logical "group by" into "stream aggregation" or "hash aggregation" is one such rule. Physical implementations are costed and compared to determine the optimal query plan.

The rule engine uses additional structures as part of the framework. A *Memo* is a structure to store equivalent alternatives generated by Exploration and Implementation Rules. Within the Memo, equivalent alternatives are stored in *groups*, and a query tree is represented using connections between groups instead of operators. This design allows for rules that match patterns without comparing whole trees. For example, join commutation can be performed for "Get(A) Join Get(B)" using the same rule as for "Filter(Get(A)) Join Filter(Get(B))". When rules match a pattern during the plan search, an alternative may be generated and inserted into the Memo. If the new alternative already exists in the Memo, nothing is inserted (more importantly, no extra work is required to re-search this portion of the possible query space). If the alternative does not yet exist, it is inserted into the Memo with the top-most operator of the alternative being inserted into the same group where the root of the original pattern resides. This means that the two query trees provide equivalent outputs and may be interchangeable.

Properties are also derived within this framework to further refine the notion of equivalent groups within the Memo. *Group Properties* (which are sometimes called logical properties) represent information about all of the alternatives within a group. For example, the set of output columns from a group is considered to be a group property. Additional group properties include any columns in the output that make up a key, the cardinality estimate for that group, and constraint properties tracking the domain for all active columns in the query. These properties are derived only on logical operators in the query tree. Alternatives within a group should, by definition, have the same logical properties. Physical Plan Properties represent particular physical details about the output of a particular physical plan. Sort is the most well-known physical property. These are computed on physical operators and are useful to determine if a physical implementation delivers the proper physical characteristics to other operators in a query tree. Special rules called *Enforcer Rules* can be used, in some cases, to deliver missing physical properties. For sort, an enforcer can insert a physical sort operation to introduce order when needed. Properties are essential to guide the search of the plan space and deliver exactly what semantics are needed for each query.

Several additional concepts help make the search of all possible plans as efficient as possible. Each operator contains a routine called *Guidance* that enumerates rules that could match it. This avoids unnecessary work to attempt to run rules against an operator when they could never match. Additionally, a *Promise* routine exists on each rule to define how valuable this particular rule could be to identifying an efficient alternative. Commonly used tricks such as pushing filters towards the leaves of a query tree have a high promise, while less common or more expensive rules such as generalized materialized view matching may have a lower Promise.

Costing functions are also used to prune searches that will yield physical alternatives that cost more than existing physical solutions. Finally, rules

are split into different optimization phases consisting of a round of exploration rules followed by implementation rules. Early phases have a restricted set of rules enabled to attempt to find a good plan quickly. If the cost of the best solution found after a phase is acceptable, the solution is returned. Otherwise, additional phases may be run in an attempt to find a better solution. Currently, SQL Server has three possible phases — *transaction processing*, *quick plan* and *full optimization*. These different strategies help provide a good plan as quickly as possible, and search for a better plan as needed.

4.1.2. Rules specific to distributed queries

The Cascades framework provides enough abstraction to allow adding *remote rules* (i.e., those rules specific to distributed queries) as easily as local rules (named "*local rules*"), and both can work seamlessly together. Like local rules, remote rules can be categorized into exploration rules and implementation rules. Examples of remote exploration rules are: parameterization of the remote queries, grouping joins based on locality, splitting and merging selection predicates based on predicate remotability, *etc*. Parameterization enables pushing parameters into the remote sources and opens up a large variety of alternative plans. The rule — "grouping joins based on locality" — reorders the joins into groups of joins based on the locality of the operand tables. Similarly, the rule — "splitting/merging predicates" — splits and collapses predicates based on the remotability of the predicates. The rationale behind grouping the operations based on their locality is to find solutions of pushing the largest possible sub-tree to the remote source (c.f. the implementation rule "*build remote query*" below).

Examples of remote implementation rules are: building SQL statements from trees to run on remote sources, building remote scan/range/fetch, adding spool on top of remote operations, *etc*. The rule — "build remote query" —constructs a SQL statement out of a logical query tree. Namely, it pushes the SQL statement to the remote source and consumes the data once the results are returned. A typical scenario where a sub-tree is pushed to the remote source is that the remote source is a fully capable query processor. Being able to quickly identify a sub-tree based on the locality of the operand tables (c.f. above mentioned exploration rules) reduces the optimization time and leverages the optimization capability of the remote sources. However, although there are rules to generate alternative plans based on locality, it does not necessarily mean they are the only solutions. The optimality of plans is determined by their estimated costs. For example, both customer and supplier in Example 1 are from the same remote source. Figure 4(a) shows the plan of pushing the join of customer and supplier to the remote server, and Figure 4(b) show a plan of joining supplier first to nation before joining to customer.

Example 1

```
SELECT c.c_name, c.c_address, c.c_phone
FROM remote0.tpch10g.dbo.customer c,
        remote0.tpch10g.dbo.supplier s, nation n
WHERE c.c_nationkey = n.n_nationkey AND
        n.n_nationkey = s.s_nationkey
```

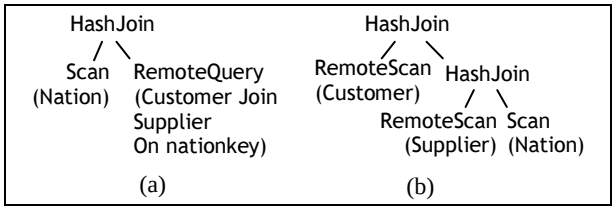


Figure 4 Cost-based optimization

On a 10GB TPCH database, the SQL Server optimizer chooses the plan shown in Figure 4(b), since by joining supplier to nation first will avoid having to send a large intermediate result set of "customer join supplier" over the network. Our optimizer does not simply rely on the heuristics of pushing the largest sub-tree to the remote sources.

The rules — "build remote scan/range/fetch" — implement remote table access via different access paths. "Remote scan" is simply a sequential scan on remote table. "Remote range" accesses a remote table via indexes, and "remote fetch" accesses a remote table via "*bookmark*" (the key). The rule — "spool over remote operation" — implements a spool to store a copy of the remote results for subsequent accesses within the same query context without having to request the data from the remote sources again.

As described in the previous subsection, the phases of optimization start with normalization of the query tree, followed by three phases of exploration and implementation. Whether to continue with the next optimization phase or not depends on the estimated costs of the query. In such ways, the optimizer will not spend too much time on optimizing easy queries, while for complex queries it will spend longer time in order to find the optimal plan. How the distributed query optimization fits into this framework seamlessly will be discussed next.

4.1.3. Seamless integration of local and distributed query optimization

Two of the strengths of the Cascades framework that make distributed query optimization as natural as the local cases are 1) the separation of *logical* and *physical operators*, and 2) the interaction between *group properties* and *rules*.

At the beginning of optimization, both local and distributed queries are algebrized in the same way, i.e., the same logical operator is used no matter the data source is local or remote, except that the remote data sources are tagged with a flag indicating their level of remotability. During the exploration phase, the optimization rules match logical query sub-trees and transform them into their equivalent logical alternatives regardless of the remoteness. Through the distinction between logical and physical operators, most of the exploration rules designed for local query optimization (local rules) are also directly applicable to distributed queries, making the optimizer framework simple and free from duplicates of the same work. For example, the rules — reordering joins, pushing the predicates before joins, removing sub-queries, *etc*. — are all applicable to remote queries. During the implementation phase, the implementation rules will interact with group properties and generate corresponding physical operators. For example, "get(A)" can be implemented as `TableScan(A)` in the local case, or as `RemoteScan(A)` if A is a remote table.

Once a logical operator is implemented as a physical operation, the costing functions are invoked to derive the estimated execution costs of each physical operator. This works for both local and remote operators. All needed here is to define the cost formula for each remote operator. It is obvious that the cost formula of an operation depends on how the operator is implemented. However, in heterogeneous, autonomous environments, it is sometimes impossible to reason about the detailed implementation of the remote operator in addition to other non-deterministic factors, like network latencies.

SQL Server DHQP defines a simple cost model based on the output cardinality of a remote operator. It aims at finding plans with minimal network traffic. With the help of remote *histogram rowsets* via the OLE DB interface, SQL Server optimizer chooses the optimal plan which produces the minimal intermediate result sets, e.g., the plan shown in Figure 4(b). This simple model works well in most of the cases and is easy to extend.

Another important component in the distributed query processing is the *decoder*. The decoder takes a logical query tree as its input and decodes it into an equivalent SQL statement. This is part of the implementation rule — "build remote query". When composing the SQL statement, the decoder responds to different parameter settings of the connection to the OLE DB provider in order to produce a SQL statement which is compliant to the remote system, e.g., the SQL dialect the remote sources support, data collation, *etc*. Additionally, SQL Server DHQP extends OLE DB by defining additional properties that providers can implement to communicate *capabilities* to Microsoft SQL Server. For example, providers can indicate support for nested select statements, parallel table scans, or specific syntactical details about date literals beyond what is defined in SQL. This information is used both in plan selection and by the decoder to more effectively remote queries (especially to non-SQL Server systems).

4.1.4. Extensions to the Cascades Framework to Support Distributed Query

Beyond the addition of specific remote query rules, several modifications and extensions to the Cascades framework are required to better facilitate distributed query processing. For example, sub-queries are usually heuristically transformed into semi-joins in the Simplification phase of optimization for local queries. This avoids duplication of logic for handling semi-joins and sub-queries. However, the un-rolling of sub-queries over remote sources is delayed until the exploration phase to simplify the process of decoding a logical tree back into SQL.

Another framework modification is required to properly generate remote queries in Cascades. If you recall from Section 4.1.1, *groups* in the *Memo* store equivalent alternatives for that portion of the query. However, not all logical alternatives in a specific group may be remotable. This can happen for any number of reasons, but common causes include the use of an abstract operator (such as a semi-join) with no direct SQL corollary, the use of an operator not supported in a previous version of the product, or even cases when the decoder has not yet implemented logic to generate SQL for a specific operator. In these cases, the implementation rule that transforms a logical tree into a remote SQL statement requires special framework logic to pick *any* remotable tree from the same group in the Memo. Since these trees represent logically equivalent alternatives, the SQL generated for them must, by definition, return the same results.

A few practical changes to the framework help improve the performance of specific user operations. It is often beneficial to spool results from a remote source if multiple scans of the data are expected. This is implemented using a special "enforcer" rule in the framework that spools a remote query's results. Additional logic is required to disable spools done for local scenarios, such as Halloween Protection [10].

4.1.5. Application of Distributed Query Processing in Federated Systems

The distributed query processing capabilities of SQL Server were further extended to support efficient processing of queries in a federated database system. A federated database system is a set of loosely cou-

pled database systems all logically forming a single database store. SQL Server announced this technology in February 2000 by publishing the world record TPCC benchmark using a federation of 32 Microsoft SQL Server instances [17].

SQL Servers support for federated systems builds on the following concepts: *constraint property framework*, *static and runtime pruning support*, *partitioned view support*, *algebraic re-writes of query and DML operator trees*, and *delayed schema validation*.

Constraint Property Framework Constraint properties leverage Microsoft SQL Server's existing optimization property framework to support tracking the domain of all scalar expressions. Domain restrictions track possible values for scalar expressions at each point in the query tree. Each relational operation can modify the valid domain for a scalar expression, and this information can be leveraged by the optimizer to make decisions on pruning the search space, cardinality estimation, or constraint validation.

For example, if a integer column CustomerId passes through a filter operator predicate "CustomerId > 50", the optimizer updates the domain property of the CustomerId column from $[-\infty,+\infty]$ to $(50,+\infty]$. (In computer systems, the positive and negative infinities depend on the data types.) This mechanism also supports disjoint ranges by tracking a set of range intervals for each scalar and then performing appropriate interval operations. For example: "CustomerId IN (1, 5) OR CustomerId BETWEEN 50 AND 100" would derive a domain property of $[1,1] \cup [5,5] \cup [50,100]$. Constraint properties can be derived from any scalar expression in the query tree including any constraints defined over columns in the source tables.

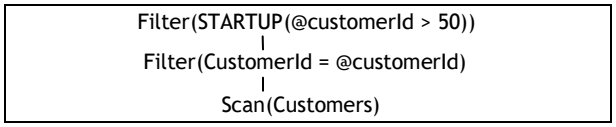
Static and Runtime Pruning Support During optimization, the constraint property framework is leveraged to infer if a plan sub-tree could produce any results. Each Boolean expression can be evaluated at compile time to see if the constraint properties of expressions intersect.

For example, given the column CustomerId with domain $(50,+\infty]$. If there is an expression "CustomerId = 20", we can attempt to intersect the domain of CustomerId with the domain of the constant 20. Since there is no overlap between $[20,20]$ and $(50,+\infty]$, the predicate can be reduced to a constant *false* value. If the predicate appears in a select or join operator, we can reduce the operator to a logical empty table operator using an exploration rule.

This reduction of expressions is only possible at compile time when the domains of the input values are known. In many cases, the domain of a predicate is not known at compile time since most modern SQL applications make use of variables in their queries. In this case, a special filter operator is introduced into the final execution plan — the "*startup filter*".

A startup filter predicate can not contain any references to columns or values in its input tree. This allows the predicate to be evaluated before the sub-tree of the filter has been executed. For example, let's consider the following query and its execution plan with the startup predicate.

```
SELECT *
FROM Customer
WHERE CustomerId = @customerId
```



In this example, the table scan of Customer will only be executed if the @customerId variable contains a value in the domain of CustomerId.

Partitioned Views A *partitioned view* unions horizontally partitioned data from a set of member tables across one or more servers, making the data appear as if from one table or domain. In a local partitioned view, all participating tables and the view reside on the same instance of Microsoft SQL Server. In a *distributed partitioned view*, one or more tables reside on a different (remote) server instance.

Records in the partitioned view are distributed across the member tables, each table representing a single logical partition. The range of values in each member table is enforced by a CHECK constraint on a column designated as the *partitioning column*. Each table must store a disjoint range of partitioned values.

For example, consider the Lineitem table of the TPCH benchmark partitioned into seven tables based on year of the CommitDate column. Each partitioned table is stored at a different server. For example, on one of the server, the partitioned table is defined as follows.

```
-- on server1:
CREATE TABLE lineitem_92 (
        [L_COMMITDATE] [datetime] NOT NULL
        CHECK (L_COMMITDATE >= '1992-1-1' and
        L_COMMITDATE <= '1992-12-31'),
        … -- additional column definitions)
```

Other partitioned tables — lineitem_93 to lineitem_98 — are defined on different servers similarly. Once the member tables are present, a SQL view is created on each member server to union the data from each partition into a single logical table. This allows queries referencing the distributed partitioned view to run on any of the member servers. The system operates as if a copy of the original table

is on each member server, but each server has only a member table and a distributed partitioned view. The location of the data is transparent to the application. The distributed partitioned view is defined as follows.

```
CREATE VIEW v_LINEITEM AS
SELECT * FROM server1.tpch10g.dbo.LINEITEM_92
UNION ALL
SELECT * FROM server2.tpch10g.dbo.LINEITEM_93
UNION ALL
SELECT * FROM server3.tpch10g.dbo.LINEITEM_94
UNION ALL
SELECT * FROM server4.tpch10g.dbo.LINEITEM_95
UNION ALL
SELECT * FROM server5.tpch10g.dbo.LINEITEM_96
UNION ALL
SELECT * FROM server6.tpch10g.dbo.LINEITEM_97
UNION ALL
SELECT * FROM server7.tpch10g.dbo.LINEITEM_98
```

Query Rewrite Support Once the partitioned view is in place, the query optimizer leverages the constraint and dynamic plan pruning support to produce efficient execution plans for the federation of servers. Example 2 illustrates how this all works together.

Example 2

```
SELECT l.*
FROM v_lineitem l, remote0.tpch10G.dbo.customer c,
      remote0.tpch10G.dbo.orders o, nation n
WHERE l.l_orderkey = o.o_orderkey AND
      c_custkey = o_custkey AND
      c.c_nationkey=n.n_nationkey AND
      l.l_commitdate BETWEEN @date1 AND @date2
```

This query pulls data from one or two nodes in the federation. Since the values of the requested Lineitems are not known at compile time, the engine must introduce startup filter expressions to dynamically select the appropriate server for execution at execution time. The execution plan is shown in Figure 5.

Partitioned View DML Support An important part of partitioned view support is the support for DML (insert, update and delete) against the view as if the client were operating over a local table. Microsoft SQL Server leverages support for constraint properties, query re-write and dynamic plan execution to provide support for transparent DML support over partitioned views.

An update over a distributed partitioned view is decomposed into one update against each partition using the check-constraint logic. If the partitioning key is updated, rows may move across partitions. As such, the query optimizer must detect this could occur and produce a plan capable of deleting rows from one server and then inserting them in another server.

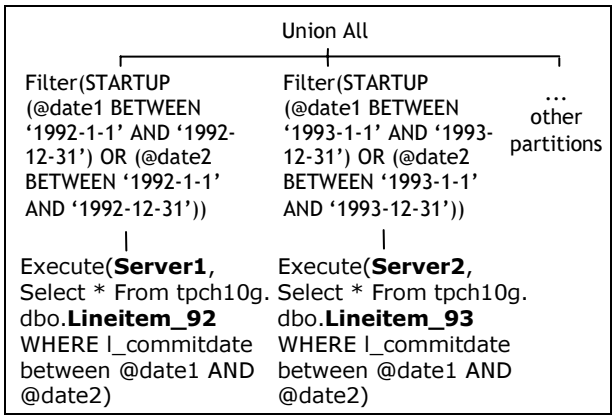


Figure 5 DPV execution plan

Delayed Schema Validation When a plan is prepared for execution, Microsoft SQL Server checks the current schema DDL version numbers of all schema objects participating in the plan. This mechanism allows the server to recompile a plan if a table has changed structure, an index added or dropped since the plan was compiled. In a federated system, the only way to check the schema on remote servers is to perform a query to the remote server node. The cost of checking schema on remote nodes, even nodes which will not be used in the plan execution (due to parameter values) is cost prohibitive in federated systems.

To address this problem, Microsoft SQL Server introduced the concept of "*delayed schema validation*" for distributed servers. When enabled, this option causes the local (source) server to perform optimistic schema checking. Rather than checking all remote schema versions before starting query execution, the query starts execution and then only checks the remote schema of this server which ultimately participates in the query execution. Combining this lazy validation with *partition pruning*, it provides significant performance advantages for applications using distributed partitioned views. If the remote server has changed schema and the server executes a portion of the plan, the plan is discarded from the plan cache and a "schema changed" error is returned to the calling application. If the application resubmits the query, the query is recompiled with the latest schema version and execution continues.

4.2. Execution

A number of techniques improve the execution time of a given distributed query plan. Connections to remote sources are cached to avoid initialization costs for each query. Especially in busy server workloads, this can remove seconds from the average query time. Another simple technique to improve execution time is to batch multiple rows together in

each call to a remote provider. This capability is provided in OLE DB as part of the **IRowset** interface. Providers either implement row batching natively, or the functionality can be simulated using a service component layer provided as part of the OLE DB Service Components framework. Providers over server databases typically can amortize the fixed costs of a network I/O better over 50 or 100 rows. These common techniques are generally supported by most major providers used against Microsoft SQL Server.

Overlapping I/O is also implemented in less obvious places to improve performance. Distributed Partitioned Views commonly store each source table on a different remote source. Therefore, the execution plan must often open several commands. Without special provisions, these may be required to be opened serially since the underlying wire protocols for server providers may not support multiple/asynchronous command requests. This functionality has been added into the upcoming Microsoft SQL Server 2005 release to better facilitate queries over multiple remote sources.

Updates are also optimized to perform better against remote sources. In Microsoft SQL Server, cursor-based updates are represented internally using two scans against a table. The read cursor provides row-ids to a write-cursor which makes the change to the row [8]. This technique has benefits from allowing the use of secondary indexes as the read-cursor to provide rows in a particular order to faster processing of batched, per-index maintenance operations. Microsoft SQL Server has an optimization to perform the updates using a single cursor if the query meets specific restrictions. This "in-place" update allows a single exclusive (X) lock to be taken on the row instead of a shared (S) lock in the read cursor followed by an X lock in the write cursor. For queries requiring no Halloween Protection [10] or sorting, this can be a noticeable performance improvement. In-place updates seamlessly work for remote sources against Microsoft SQL Server. In some cases, these are actually required for the remote source to work properly. If the provider opens the read and write cursors over the remote provider using different connections (and thus different transactions), the S and X locks can actually conflict and return an error. This optimization therefore provides important performance and functional guarantees for some remote sources.

5. Related work

Relevant work in this area includes work on architectures and frameworks to support distributed and heterogeneous query processing, and the optimization techniques for DH queries.

Similar to OLE DB, IBM's Garlic is a middleware system designed to integrate data from different data sources [7][11]. In addition to the exposure of the query capabilities of the data sources, Garlic's wrappers also need to provide cost functions and optimization rules. Microsoft SQL Server DHQP, on the other hand, relieves the responsibility of defining optimization rules and the cost functions from the OLE DB provider implementers. By doing so, it lowers the entry bar for sources to plug into the Microsoft DHQP. In addition, one can implement a minimal subset of OLE DB or one can extend the model to meet the needs of emerging applications. In addition, Microsoft SQL Server DHQP is fully integrated into the Cascades cost-based optimizer framework which enables the generation of efficient query plans over diverse sources.

Regarding to distributed query optimization, SDD-1 [1] from CCA deploys semi-join to efficiently delimit the subset of the database that contains data relevant to the query. IBM's System R* [12] uses dynamic programming for optimizing queries using semi-joins and joins. Mermaid [18] makes use of semi-joins to eliminate unnecessary relations. Others can be found in [19]. All of the above previous work tackles the problem in a distributed homogeneous environment. Microsoft SQL Server DHQP, on the other hand, deals with both homogeneous and heterogeneous distributed queries. The Microsoft SQL Server Cascades optimizer implements dynamic programming and memorization for efficient rule-based exploration and cost-based pruning to find the optimal plan.

6. Concluding remarks

In most organizations the diversity and volume of data grow rapidly. To have efficient and extensible distributed query processing capabilities is essential to any database management systems.

The distributed, heterogeneous query processor in Microsoft SQL Server implements an extensible architecture, based on OLE DB, supporting robust cost-based query optimization against remote data sources. Tight integration with the existing Cascades query optimization framework allows local and remote queries to be treated identically for core optimization decisions while supporting extensions specific to remote queries. Additionally, the query execution framework has been extended to hide many of the latencies seen in this class of distributed application. This framework supports arbitrary third-party providers, and modern development tools enable someone to write a simple provider which can integrate data into the query processor.

The DHQP framework supports a number of additional internal features using the same framework. Full-text content indexing has been implemented by plugging the Microsoft Index Server as an OLE DB provider called from the DHQP, allowing rich text searching over file system and table content. DHQP supports scale-out partitioning through the use of partitioned views over multiple remote sources. This configuration set a TPCC record when it was first introduced. Additional features can be built by third-party vendors who wish to leverage the capabilities of an existing query processor over external data.

References

[1] P. Bernstein, N. Goodman, E. Wong, C. Reeve, J. Rothnie. Query Processing in a System for Distributed Databases (SDD-1). *ACM Transactions on Data Base Systems*, 6:4, Dec 1981.

[2] J.A. Blakeley, M. Pizzo. Enabling Component Databases with OLE DB. In *Component Database Systems*. K. R. Dittrich, A. Geppert (Eds.). Morgan Kaufmann, 2001.

[3] J.A. Blakeley, M. Pizzo. Microsoft Universal Data Access Platform. *SIGMOD Conference* 1998.

[4] J.A. Blakeley. Data Access for the Masses through OLE DB. *SIGMOD Conference* 1996.

[5] J.A. Blakeley. OLE DB: A Component DBMS Architecture. *ICDE* 1996.

[6] D. Box. Essential COM, *Addison Wesley*, 1998.

[7] M. Carey et al. Towards heterogeneous multimedia information systems. *Intl. Workshop on Research Issues in Data Engineering*, March 1995.

[8] C.A. Galindo-Legaria, S. Stefani, F. Waas. Query Processing for SQL Updates. *SIGMOD* 2004.

[9] G. Graefe. The Cascades Framework for Query Optimization. *Data Engineering Bulletin* 18 (3) 1995.

[10] Anecdotes. *IEEE Annals of the History of Computing*. A. Fitzpatrick (Ed). 24(2): 86-93 (2002).

[11] L.M. Haas, D. Kossmann, E.L. Wimmers, J. Yang. Optimizing Queries across Diverse Data Sources. *VLDB* 1997.

[12] G. Lohman, C. Mohan, L. Haas, D. Daniels, B. Lindsay, P. Selinger, P. Wilms. Query Processing in R*. In *Query Processing in Database Systems*. W. Kim, D. Reiner, D. Batory. (Eds.) NY: Springer-Verlag, 1985.

[13] Microsoft. Distributed Queries. *MSDN: Microsoft SQL Server*. http://msdn.microsoft.com/library/en-us/acdata/ac_8_qd_12_9ooj.asp

[14] Microsoft. IFilter. *MSDN: Indexing Service Reference*. http://msdn.microsoft.com/library/en-us/indexsrv/html/ixref_6xid.asp

[15] Microsoft. MSDN: Microsoft Distributed Transaction Coordinator. http://msdn.microsoft.com/library/en-us/cossdk/htm/dtc_toplevel_6vjm.asp

[16] Microsoft. *MSDN: OLE DB Templates*. http://msdn.microsoft.com/library/en-us/vccore/html/vcconOLEDBProgramming.asp .

[17] Transaction Processing Performance Council: http://www.tpc.org/tpcc

[18] C.T. Yu, C.C. Chang, M. Templeton, D. Brill, E. Lund. Query Processing in a Fragmented Relational Distributed System: Mermaid. *IEEE Trans. Software Eng.* 11(8), 1985.

[19] C.T. Yu, W. Meng, Query Processing in Distributed Relational Database Systems, In *Principles of Database Query Processing for Advanced Applications*. Morgan Kaufmann, 1998.