# Graywulf System Documentation

**László Dobos**  
Eötvös Loránd University  
The Johns Hopkins University

2013

# Table of contents

[Graywulf System Documentation 1](#_Toc373417924)

[Introduction 4](#_Toc373417925)

[Graywulf concepts 5](#_Toc373417926)

[Basic operations by users 6](#_Toc373417927)

[Schema Browser 6](#_Toc373417928)

[Query editor 7](#_Toc373417929)

[MyDB 7](#_Toc373417930)

[Export and import data 7](#_Toc373417931)

[Plot tables 8](#_Toc373417932)

[Share data 8](#_Toc373417933)

[Basic operations by administrators 8](#_Toc373417934)

[Creating a cluster 8](#_Toc373417935)

[An overview of Graywulf modules 8](#_Toc373417936)

[Graywulf Registry 10](#_Toc373417937)

[Cluster group 10](#_Toc373417938)

[Federation group 11](#_Toc373417939)

[Layout group 12](#_Toc373417940)

[Jobs group 13](#_Toc373417941)

[Security group 14](#_Toc373417942)

[Operations by the registry 14](#_Toc373417943)

[Discover 14](#_Toc373417944)

[Deploy 15](#_Toc373417945)

[Diagnose 15](#_Toc373417946)

[Schema and metadata 15](#_Toc373417947)

[Metadata parser 15](#_Toc373417948)

[SQL Parser 16](#_Toc373417949)

[Data formats 17](#_Toc373417950)

[IO building blocks 17](#_Toc373417951)

[Activities and jobs 18](#_Toc373417952)

[Workflows 18](#_Toc373417953)

[Job definitions 19](#_Toc373417954)

[Queues 19](#_Toc373417955)

[Job instances 20](#_Toc373417956)

[Scheduler 20](#_Toc373417957)

[Schedule startup 20](#_Toc373417958)

[Job execution 20](#_Toc373417959)

[Stopping the scheduler 21](#_Toc373417960)

[System monitoring 21](#_Toc373417961)

[Job binaries 21](#_Toc373417962)

[Task delegation via Remote Service 21](#_Toc373417963)

[Logging 22](#_Toc373417964)

[System setup 22](#_Toc373417965)

[Hardware configuration 23](#_Toc373417966)

[Platform configuration 23](#_Toc373417967)

[Hardware configuration 23](#_Toc373417968)

[Windows configuration 23](#_Toc373417969)

[Database Server configuration 23](#_Toc373417970)

[Setting up a Windows account for the services. 23](#_Toc373417971)

[Developing Databases for Graywulf 24](#_Toc373417972)

[Credits 25](#_Toc373417973)

[References 25](#_Toc373417974)

[Copyright 25](#_Toc373417975)

# Introduction

Graywulf is a framework and API to implement distributed database systems over a cluster of Microsoft SQL Servers. It aims to support many distributed database scenarios: system registry, database allocation, query partitioning, distributed joins, scheduling etc.

Graywulf is written in C#, uses SQL Server for the registry and data operations but can read remote data sources, such as data files, MySQL, Postgres etc.

Besides the API, Graywulf also provides a web-based user interface for end users. The user interface allows users to submit queries via a batch system and store query results and their own uploaded data in a private database called MyDB. With the help of this user interface Graywulf is not only a database federation toolkit, but can also be used to publish data to end-users of data warehouses.

# Graywulf concepts

Graywulf was designed to be a platform for distributed database applications with the following goals.

* Simplify database cluster management via a simple web-based management console.
* Provide an API to build data loading pipelines.
* Provide an API to implement data management jobs.
* Provide (primarily read only) access to a federation of databases co-located on a high performance database server cluster via simple queries.
* Provide a simple, yet powerful web-based user interface that allows users to build there scientific data processing workflows from raw data analysis to press quality plots.

**Cluster:** A collection of machines that are managed together and work together in a controlled way. Clusters usually consist of a head node, or controller, that stores the system registry and runs the job scheduler, as well as worker nodes which run SQL Server instances containing the data. Worker nodes might also run delegated jobs, such as data import and export tasks.

**Federation:** A collection of databases that belong to the same application, field of research, etc.

**Slice:** Databases that are two large to fit on a single server might be sliced into multiple smaller databases. Slicing most of the time happens along the primary key of the largest fact table, carefully observing foreign key constraints. While Graywulf has had the concept of slices since the beginning, sliced databases are not implemented yet.

**Dataset:** In standard TSQL, tables and function are identified by their three part names: database.schema.objectname. In case of linked servers, one can use four part names to identify tables. In Graywulf the syntax is slightly extended to add a so called dataset name in the following form: dataset:schema.objectname. Datasets are distinguished from databases for a series of reasons. First of all, a dataset is most of a time a collection of databases (a set of mirrors, a set of slices, etc.) and not a single database. In a typical federation, different catalogs of data are distinguished by their dataset name.

**MyDB:** Downloading query results after each execution is inefficient for many reasons. To avoid all these problems, users get a scratch database called MyDB for their own purposes at registration where all query results are written by default. Users then can either further process these tables, or download them in various formats.

**Jobs:** All long-running operations are implemented in Graywulf as jobs. Jobs are stored in a queue until execution, when they are picked up by a scheduler that executes them. The job system is designed to be highly extensible and federations can easily implement their own jobs or inherit from the generic Graywulf jobs. System maintenance tasks are also jobs, often recurring ones.

**Controller:** Contoller is the head node of the cluster. It holds the central databases for the registry, job persistence store and logs. The controller also runs the scheduler.

**Partitioned query:** If a database is mirrored to multiple servers, a single query can be partitioned into smaller pieces that run in parallel on multiple machines and results are gathered upon completion. Graywulf implements a very simple model of query partitioning. When a partitioning clause is added to the table that comes first after the FROM keyword in a SELECT query, statistics about the partitioning key are gathered, partition bounds are determined and finally the query is executed in parallel on all machines where the necessary data is available.

**Sliced query:** If a database is slices, i.e. it spans multiple servers due to limited space on individual servers queries must be executed on all slices and results must be gathered upon completion. This query type is not yet implemented.

**Distributed join:** Users may write queries that can reference databases throughout the cluster, even databases that are not available on the same server. Certain queries can reference MyDB tables and even tables on remote data sources (servers not managed by Graywulf). When query references tables from a single database only, it can be always executed on a single server but once we allow multiple databases, it is likely to happen that the required data are not available co-located. Joins between tables residing on separate servers are called distributed joins. Graywulf already supports distributes queries, but the implementation is very simple. A database server is chosen with the most data readily available and all other tables are copied to that particular machine. Simple logic is used to determine a subset of columns and rows of remote tables that are absolutely necessary to execute a join.

**Vertically partitioned table:** Wide tables usually take up too much space to be mirrored to multiple servers. Also, most columns are only accessed by a few percent of the queries while most queries use a few percent of the columns. It might be worth to split the table into to and store the less frequently used columns on a separate, slower sever with larger storage. Vertically partitioned tables could be even implemented transparently, not requiring the users to explicitly do a join between the two table parts. Vertically partitioned tables are not yet implemented in Graywulf. This concept is frequently called lazy join too.

**Metadata:** While database columns, function parameters, variables have only a name and a data type in SQL, most physical quantities in science carry additional information about a piece of data than a value and a name. Graywulf extends the database schema with metadata that can store a physical unit, a human readable description and a software readable content identifier for each column and parameter.

# Basic operations by users

The main objective of Graywulf is to provide a development platform for distributed and federated databases. Users can access database federations by writing queries. Graywulf provides a web-based user interface to interact with the system consisting of a schema browser, a query editor, a set of tools to manage myDB, import and export features, a plotting tool and data sharing tools.

## Schema Browser

Queries can only be written in the knowledge of the underlying schema, thus an efficient schema browser is essential. As Graywulf is designed for scientific applications, detailed descriptions on the contents of the tables and columns are also necessary. The Graywulf schema browser is based on the same schema library classes that are used internally for query processing, and can benefit from schema caching, etc.

The schema browser already knows a lot but there’s quite a few features missing. Searching the schema would be useful (though hard to implement, because it would require collecting and indexing the schema of all datasets and maintaining this index). The user interface could also be a bit more intuitive. Meta data already supports adding sample code to object descriptions. No support for enums and user defined types yet. Also, if we add enums, we could support named constants, a feature missing from SQL Server. The schema browser uses AJAX and does not report exceptions, which makes detecting bugs hard.

## Query editor

The Graywulf query editor is based on an open source javascript code editor component that supports syntax highlighting.

The query editor uses codemirror for code editing. It supports all major browsers nicely and I’ve written an asp.net component that wraps it nicely, even when AJAX is used. Codemirror could be wired up with the schema to auto-complete SQL and identifiers. It would be great to add buttons to hide the menu bar from the top of the page to give more space for the editor on smaller screens. Displaying quick results might be buggy. No button to download quick results immediately, like in casjobs.

## MyDB

Query results and uploaded data appear in MyDB. Users have the ability to drop and rename MyDB tables. Future plans include complete schema management tools where database objects could be freely edited, indices created, metadata added, tables duplicated, etc.

I think the MyDB GUI is already much nicer than in casjobs. There’s no way to share tables yet. Also, a schema editor would be great, when tables could be modified, indices created (no DDL queries are supported yet), etc. I have plans to add features like generating identity columns with a single click, etc.

## Export and import data

Data tables can be directly imported into MyDB tables. The web user interface provides a form for uploading data files. File types can be automatically detected form file extensions, or set manually. The data import framework can automatically detect data columns (in case of formats without schema description), add identity columns and append metadata. As files are uploaded via the browser, import operations are done synchronously and as coordinated by the web server. Future plans include allowing uploading zip archives that may contain multiple data files. Instead of uploading files via the browser, fetching files from HTTP, FTP, etc. will be available. The currently supported data format is CSV but the format framework is designed to be extensible.

Data from MyDB can be exported into various formats. Data exports are scheduled as jobs and coordinated by the controller machine. Data export operations are executed by the worker nodes of the cluster and not the web server.

There’s a data format connector for CSV implemented and that’s all. I’m working on FITS and Deoyani has made some progress with VOTable for SkyQuery. I want to add tabular ASCII (IDL output), XML datasets and SQL Server native bulk insert, maybe HDF5, depending on needs. It’s relatively easy to add new formats but testing can take a significant time. Gzipping individual files works but there’s no way to download multiple tables in a zip archive yet.

## Plot tables

Future plans include adding a plotting tool to the user interface that will enable users to create print quality plots. Either a tool based on gnuplot or python’s matplotlib will be implemented.

We have a gnuplot-based tool that was successfully wired up to graywulf by someone but it doesn’t integrate into the job system yet. Gnuplot is easy to use but lacks many features. Also, creating nice plots can be challenging because default settings aren’t the best. Tamás is suggesting using matplotlib. Python runs on windows natively but not on IronPython, thus we need to come up with a solution to restrict python access to plotting only. We definitely don’t want to expose file names, connection strings, etc. to python script writers, which might be challenging.

## Share data

There are no means of sharing data tables among users in the current versions.

# Basic operations by administrators

Setting up Graywulf on a cluster of servers is not a trivial task, but does not require software developer skills. Most basic setup steps can be done with command-line tools and once the admin web site is running, it can be used to manage the cluster.

## Creating a cluster

Graywulf

Creating a federation

Creating a database definition

Creating physical databases

Loading data

Publishing data

# An overview of Graywulf modules

Graywulf consist of the following modules.

**Registry:** Stores system information about the status of the cluster in a central SQL database. Hardware configuration is kept down to the RAID volume level to support optimized allocation of databases. Every physical database is reflected in the registry down to the file level to support database mirroring, moving etc. The registry also contains a complete user database.

**Schema:** This module is a collection of classes to reflect the schema of SQL databases with additional (custom implemented) metadata. The schema manager class supports caching of schemas to minimize schema read request toward the database servers. Schema connectors are implemented for SqlServer, MySQL and Postgre, but the framework is extensible. The schema browser web interface is built on top of the schema module.

**SQL parser:** The system consists of a SQL language parser automatically generated from a grammar at compile time. The parser builds a parsing tree from the tokens of a SQL statement that can be later used to analyze the query. Many parser tree node types are extended manually to support the analysis of the query. The SQL name resolver identifies every reference to tables, columns and function calls in the parsing tree generated by the SQL parser, and associates them with the underlying database schema. The name resolver can identify unbound names (identifiers that do not exist in the database) and collect the set of tables and columns that are necessary to execute a query.

**Format:** The format module is an API to implement data file format connectors that allow reading files as if they were data tables, and serialize data tables into various data file formats.

**IO:** The IO module implements data import and export operations. It is based on the Format module but extends its functionality. It can do resilient file copies between servers, but also run bulk inserts and bulk copies from remote servers.

**Activities:** Graywulf implements a set of activities that function as building blocks of workflows that in turn implement jobs that can, for example, execute complex, long running queries, or data export operations, etc. An activity, for example, can export a data table into a file or execute a query on a remote server and store the results in a local cache table.

**Scheduler:** Long-running operations in a distributed database environment are implemented as jobs. In Graywulf, every job is a workflow (built on top of .Net Workflow Foundation) of multiple steps that may contain asynchronous operations and parallel loops. The Graywulf Scheduler can queue and execute such jobs reliably.

**Remote Service:** Task delegation is essential in multi-machine environments. The Graywulf Remote Service module is a lightweight implementation that allows activating and executing user code on any machine of a cluster from the scheduler.

**Logging:** All operations are logged into a central SQL Server database. Logging can be done at such granularity that log record can be used for debugging.

**Web interfaces:** Graywulf provides a web interface for system administration, a user interface and a .Net single sign on interface. The admin web site allows managing the system registry directly. The user interface allows users to browse the database schemas, write and execute queries, track query execution, get information about their MyDBs, import and export data and access help. The authentication service is a central web site for user registration and authentication that can be shared among .Net web sites and services running under the same domain name.

# Graywulf Registry

The Graywulf registry is a complete description of the logical and physical configuration of the database cluster. The registry is a hierarchically organized collection of entities. Entities are also organized into for main groups: Cluster, Federation, Layout, Jobs and Security.

The Graywulf registry is stored in an SQL Server database that is used heavily because it is shared by all Graywulf system components. Communication among the components is basically done via the registry, and this is why it is important to store the registry on the same machine where the scheduler runs, to minimize network latency. The registry is never accessed directly by simple SQL queries but via a large set of classes. Every entity is implemented as a separate class and factory classes exist to support searches.

Entities are organized into a hierarchy and identified by their unique IDs. Because IDs are hard to be used in config files, besides unique IDs, entities also have names. Names are unique to a given level of the hierarchy only, but can be combined into unique names, called fully qualified names, for example Graywulf.VOServices.SkyQuery.SDSSDR7. The hierarchy of entities can be navigated by either using the Parent property of an entity or loading and iterating all children of a node.

Entities can also reference each other, making the hierarchy essentially a graph. Classes exist to handle lazy loading the referenced entity, making navigating the graph relatively easy.

Implementing new entities is simple as they all derive from the same class. Persistence and loading of entities from the database is handled automatically with a set of custom routines. The custom implementation of a persistence tool was necessary as none of the available tools (entity framework, NHibernate) could provide the required speed flexibility and simplicity. Adding new entities to the system, however, cannot be done dynamically and requires rebuilding, reinstalling and restarting the entire system.

The Graywulf registry, or subtrees of the hierarchy can be saved into XML and reloaded from it. This allows scripting system configuration.

The implementation of the registry is more-or-less complete. Certain features are missing here or there, like automatic discovery of machine capabilities and configuration sanity test functions.

## Cluster group

The cluster group contains entities that reflect hardware components. A cluster consists of servers organized into server roles. Each server can have multiple disk volumes and database server instances. The cluster group contains the following entities.

**Cluster:** The root entity of the registry hierarchy is a single cluster. A single Graywulf installation might control multiple clusters, but each cluster is administered separately, so in the administration web site, only a single cluster will appear and no new cluster can be created. A cluster is a collection of database server machines, databases allocated, as well as application running on them.

**Machine Role:** Machine role is a collection of machines that serve the same purpose in a server cluster. By default, the ‘Controller’ role is used to store the Graywulf registry and execute the job scheduler, whereas the ‘Node’ role is created for database server worked nodes that execute queries, data import and export operations, etc.

**Machine:** A single physical machine of the cluster.

**Disk Volume:** Represents a physical disk volume, usually a big RAID volume, that can be used to store databases. Graywulf can use this information to optimize database allocations for the hardware. To support data movement, data volumes must be shared on the local network and share UNC path is stored for each disk volume.

**Server Version:** Defined on the machine role level and identifies SQL Server versions. Graywulf supports multiple SQL Server versions running side by side, and each server instance is associated with a server version.

**Server Instance:** Defined on the machine level and represents a SQL Server instance running on that particular machine.

## Federation group

The Federation group contains entities that describe the logical configuration of the system. A single cluster can serve multiple purposes, for example, used for two different fields of research. Therefore, database federations and services are combined into domains. Federations are collections of database definitions, which are essentially collections of databases with the same schema, but possibly different data. The Federation group contains the following entities.

**Cluster:** Represents the entire cluster. It is the root of the entire registry and is the same node as in the Cluster node, see above.

**Domain:** A domain a collection of database federations (applications) belonging to the same field of interest. Federations belonging to the same cluster can share certain settings, for example users can register into a domain, instead of individual federations, and share their identity across the services provided within a domain.

**Federation:** A federation is a loose collection of databases definitions that can be accessed in parallel to combine the information contained in them. A federation of databases is usually access by one or more applications are services.

**Database Definition:** A database definition is an abstract representation of a set of physical databases with *identical schema*. A database definition can refer to a set of mirrored databases, or various versions of the same database. A database definition can also be a set of sliced databases, databases that are two large to be stored on a single server and have to be slices (partitioned) and distributed over multiple nodes. It is very important that all databases within a database definition must have exactly the same schema. If the schemas differ, multiple database definitions have to be created. Database definitions can be created under a federation, or directly under the cluster. The latter ones are databases that are shared by all application of the cluster, for example, temporary storage.

**Database Version:** Database versions can be defined to distinguish different variants of the same database. It is not intended to do actual database versioning, but to differentiate between different states of operation. For example, during a data loading process, a ‘hot’ database can serve queries while a ‘warm’ database replicates the same data for redundancy and a ‘cold’ version is used for merging in new data. Once loading is completed, database versions are rotated. Another use case of database versions is when a ‘hot’ version serves queries while a ‘mini’ version of the same database contains a subset of the data only and is used to gather statistics before query execution.

**File Group Definition:** Each database definition has multiple file groups. Data tables will be associated with these file groups when the databases are allocated. Storage requirements for each file group have to be set manually and Graywulf later can use this information to generate database files automatically across multiple disk volumes.

**Slice:** Large databases that do not fit on a single server can be sliced up (partitioned) to span multiple servers in the future. This functionality is not yet implemented. A database definition always has at least one slice called ‘FULL’, meaning that databases are monolithic.

**Partition:** SQL Server supports partitioning of data tables. In Graywulf, partitions for each slice can be defined and the partitioning functions for each physical database are generated automatically. Not fully implemented, and not used with monolithic databases.

**Deployment Package:** Not yet implemented.

**Remote Database:** Remote databases are databases that are not controlled by the Graywulf system but can be referenced in queries. Data on remote servers will be cached before query execution. Graywulf implements some basic logic to pre-filter data on remote servers and fetch only those parts of remote tables that are absolutely necessary to execute a query. Currently MSSQL, MySQL and PostgreSQL are the supported system.

## Layout group

Federations consist of database definitions. Database definition are only abstract representations of physical databases as they can represent sets of actual databases, for example, a set of mirrored databases, databases with different versions of data or databases that are sliced (partitioned) over a collection of servers. The layout group, consequently, consists of the description of the mapping of the logical system configuration (Federation group) to the actual physical configuration of the system (databases existing on the cluster nodes). The layout group contains the following entities.

**Cluster:** Same as described above.

**Domain:** Same as described above.

**Federation:** Same as described above.

**Database Definition:** Same as described above.

**Database Instance:** While database definitions are only abstract representations of a collection of databases with identical schema, database instances refer to actual databases residing on the cluster nodes. A database instance is always an ascendant of a database definition in the configuration hierarchy, but also tagged with a database version and associated with a slice.

**File Group Instance:** File group of a physical database, associated with a database instance and partition. Partitioned databases are not yet fully implemented.

**File Instance:** Refers to a physical file on one of the cluster servers. This information is used when databases are allocated, moved or mirrored. Database files are associated with the disk volumes in the cluster configuration, consequently their local and UNC path can always be figured out easily.

## Jobs group

The jobs group contains the entities describing job queues, job types and actual jobs. Queues can be defined on the cluster or federation level and associated with machines. Certain jobs are defined on the cluster level, such as regular maintenance jobs, but most jobs are defined by the federations, such as different types of query jobs.

**Job Definition:** Jobs in Graywulf are implemented as binaries and they have to be registered in the system before they can be scheduled. A job definition is a reference to a class in a DLL implementing the job. Job definitions are either created on the cluster level (system jobs for maintenance) or on the federation level (application specific jobs). Jobs have input and output parameters and checkpoints. Checkpoints can be used to track job execution.

**Job Instance:** Job instances are actual parameterized jobs that are executed by the scheduler. Job instances are created when users interact with the system and send queries for execution, download tables, etc. Jobs can be enqueued for batch execution or scheduled for a given time.

**Queue Definition:** Job definitions are ordered to queue definitions, so not any job can be enqueued in just any queue. This distinction of jobs is important because certain jobs should run with higher priority.

**Queue Instance:** Job instances are actual queues, associated with a server of the cluster. Jobs that require many machines are usually queued on the controller but this doesn’t mean the actual work will be done by the controller. Maintenance jobs can be queued on the worker nodes.

I added checkpoints to jobs to emulate how the original SkyQuery data loader worked. Checkpoints, however, aren’t appropriate for parallel jobs, so it would be better to come up with something else to give users a clue about the progress of their jobs.

The scheduler always runs on the controller, so jobs shouldn’t do any hard computations, they must delegate everything to the worker nodes. The concept of having queues associated with machines comes from the requirement of scheduling maintenance jobs on the individual machines. The scheduler can only pick up a given number of jobs from a single queue at the same time (outstanding parallel jobs), whereas any number of jobs in separate queues can run in parallel.

## Security group

**Security group:** The security group contains entities that identify users and user groups. Users and groups can be defined either on the cluster level (this option is for administration purposes) or on the domain level. As a consequence, users registered to a domain may share their identity among the different federations (application) in the same domain.

**User Group:** User groups are collections of users having the same role in the system. Users can participate in any number of groups.

**User:** Users are identities with passwords. The web interface includes a central web site that has forms for user registration and authentication. Users of a domain can be shared among federations and web-based authentication is done centrally for all web sites associated with the same Graywulf domain.

Right now there’s only one user group defined for the cluster that contains the users who can access the system management tools. Additionally, there’s a user group automatically defined for each domain with which self-registered users are associated. Additional user groups could be created, for example, for sharing tables etc.

Users are authenticated by a central web site that shares the machine key with the other sites requiring authentication. This way a single-sign-on scenario is implemented but it only works with .Net form-based authentication.

There’s quite a few things stored in the Graywulf registry associated with users. For example, each user can have a set of MyDBs, though the query interface exposes a single MyDB right now. We could use this feature for MyScratch. If we decide to use Keystone, I think the best would be to replicate the keystone database in Graywulf, because storing some user-related info right in the Graywulf registry is essential for fast operation.

# Operations by the registry

Registry entities reflect the physical and software configuration of a real server cluster. To help keeping the description of the system consistent with the actual configuration, the registry provides a set of functions to synchronize the registry with reality.

## Discover

The discovery function of the registry can query hardware and software components and compare the settings with the values in the database. If the database differs from the system, functions exist to resolve these conflicts by updating the registry.

Discovery of databases is implemented but hardware cannot be queried automatically.

## Deploy

When new databases are added to the system, they have to be registered into the database first, and Graywulf can create, mirror, etc. the actual databases on the server nodes.

Deployment of databases is implemented. There are plans to add more features, like updating databases by submitting a script that gets executed on each database of the same database definition, etc. Also, the so called deployment packages could be used to install software on the servers. Maybe windows has better system tools to do this, but yet, I can imagine installing new SQL CLR libraries via Graywulf.

## Diagnose

The Graywulf registry provides functions to run sanity tests on the system components. By pinging servers, looking for running processes, trying to open databases, a complete view of the system can be get which simplifies finding system errors significantly.

Some basic diagnostic tools are implemented but a lot more could be added. For regular monitoring tools, like nagios and munin are more useful.

# Schema and metadata

Graywulf features a metadata library that can almost fully reflect the schema of SQL Server databases. The schema browser was written from scratch and does not rely on SQL Server Management Objects (SMO) for many reasons. First of all, the schema library supports additional schema providers to other platforms, such as MySql or Postgres. The schema library supports caching of the schema on the client side which reduces network traffic significantly when the schema is to be reused by different components of the system. Components using the schema library are the SQL name resolves (see SQL Parser section) and the schema browser.

The schema library extends standard SQL database schema with metadata. Metadata is a human/software readable extension to describe the contents of tables, column, parameters etc. The basic metadata entries are a summary, a detailed description that can contain examples, a content identifier field for software clients and a physical unit of the quantities stored in the variable.

Since the Graywulf system parses every SQL query, it can extract detailed information on how output tables and columns are generated from the source tables and columns. This will allow us in the future to extract detailed provenance information from the queries and set metadata on the generated tables automatically.

The user query interface of Graywulf features a schema browser that can display the schema and metadata of all configured datasets in a unified way.

## Metadata parser

Metadata is rather hard to define manually through any user interface. For this reason, Graywulf supports extracting metadata directly from SQL scripts. Similarly to XML comments in C#, metadata can be written directly into CREATE TABLE statements and a command-line tool can be used to extract this information, and save it to the database. Metadata is stored in SQL Server extended properties instead of metadata tables.

SQL Server schema is more or less completely reflected by the schema library. UDTs are missing. Provenance is missing entirely. SQL comments are parsed by a command-line tool right now, but comments could be added by users too, and parser automatically when queries are submitted.

# SQL Parser

SQL Parser is a library to analyze SQL queries prior to execution. It consists of five main modules: a parser generator with a SQL grammar, a SQL parser that is generated with the former but contains many hand-coded extensions, a name resolver, a basic SQL validator and a set of code generators.

The parser generator is a compile-time tool that generates the parser classes from a grammar. As a specialty, grammars are formulated as C# expressions, which enable checking grammar consistency at compile time. The parser generator supports grammar inheritance, a feature that enables writing extensions to another grammar without having to modify the original. The parser generator was written from scratch to generate parser tree nodes exactly in the form a C# SQL parser requires. The generated parser is a simple backtracking one, rules are implemented in the parsing tree node classes. All generated classes are marked partial, so custom implementation to the generated code can be added very easily. The SQL grammar is freely extensible but currently supports a slightly limited version of the SELECT statement only.

The SQL Parser library contains a name resolver which is a very important part of the system. The name resolver can identify nodes in the parsing tree that reference tables, functions calls, columns etc. and matches them with the underlying database schema. As a result, a query executor can figure out lots of things about the query, for example, get the list of tables and columns that have to cached from remote servers before query execution.

The name resolution process runs recursively on subqueries, starting with the innermost query and traversing the hierarchy outwards. 1) Default dataset names and schema names are inserted into table and table-valued function identifiers. 2) Tables and table-valued functions are resolved after the FROM clause. 3) Columns and function calls in the SELECT list, ON criteria, WHERE, HAVING and GROUP BY, etc. clauses are resolved. The last clause to process is the ORDER BY.

The name resolver currently supports the following.

* Resolve table and table valued function references
* Resolve column references in all clauses
* Support multiple occurrences of the same table in the same query (with different aliases)
* Support subqueries
* Support UNION-type multi-part queries
* Can generate CNF and DNF of logical expression
* Can determine the most restrictive where clause that can be applied to a table before joins

The SQL Parser library contains code generators. Code generators visit the parsing tree and convert the nodes back to SQL text. Different generators can be implemented for various flavors of SQL, currently simple code generators exist for MySQL and Postgres, beside SQL Server. Code generators can also generate special queries to fetch parts of tables from remote servers.

The SQL parser is one of the trickiest parts of the system. The grammar is easily extensible, but identifiers have to be resolved in order to be able to select a server for the query. Currently I can only parse SELECTs but there’s a guy working on the rest. The name resolver works for simple selects with fancy subqueries, but no CTEs yet. The current version only supports single statements. Multi-statement queries are tricky because of the distributed execution. Maybe we should restrict partitioned queries and xmatch queries to single statements and let the ordinary queries run as they are. We just need a way to figure out automatically which are ordinary queries and which aren’t. It’s easy with xmatch because there’s a non-standard clause there, but if we add transparent features, like lazy joins of vertically partitioned tables or myscratch it becomes tricky.

# Data formats

The Graywulf Format API contains classes that help implementing new data formats that can be read and written by the framework.

Internally, each data file is read as a table via an IDataReader implementation. Files are written by an IDataWriter, row by row. The IDataWriter takes data rows from an IDataReader. A single file, if the file format supports it, might contain multiple tables, called blocks.

Additional features supported: compressed files, compressed archives of multiple files (not yet complete).

I’m still working on the format API, it’s not really stable yet.

# IO building blocks

Graywulf implements a few building blocks on which applications can rely. These IO building blocks are implemented such a way that they can be executed on a remote machine (see Task delegation). This is important in many cases. For instance, let us consider a file copy between machine A and B, but the copy initiator process (the scheduler) running on machine C. If machine C did a file copy from \\A\share to \\B\share, it would result in doubled network traffic and put high load on machine C. By delegating the copy operation to machine A or B can eliminate this problem. Most of these operations also support async execution and cancellation logic. Cancellation is important in case of long running jobs that might be cancelled manually, or forced to time-out by the scheduler.

**Reliable file copy:** File copies use ESEUTIL (a utility shipped with Exchange Server) for reliability and speed.

**Bulk imports and exports:** Bulk imports can be executed on any machine and can read from any file format supported by the framework. Import and exports can access local and UNC files only, but future plans include adding ability to read and write FTP, dropbox, etc.

**Database mirroring:** Creates multiple mirrors of a database to many servers. It is implemented as a job that runs inside the scheduler. It can do cascaded copies, meaning that once a copy of a database already exists, it can use two source copies to copy to two other servers, and so forth.

File copy works but not cancellable. Should be simple to finish. Bulk imports and exports work, but they use .Net SqlBulkCopy and SqlDataReader which isn’t the most efficient way of moving data around. Database mirroring theoretically works, but needs testing with big databases. I’ve used it to copy the skynodes but experienced heisenbugs (intermittent problems) that I couldn’t track down. The mirror job definitely likes to fail if the servers are in a bad shape (dropping raid volumes, for instance).

# Activities and jobs

In the Graywulf system every job is a workflow, implemented in .Net Workflow Foundation. Workflows consist of activities that are organized into a typical workflow graph which might contain sequences, loops, conditional branches, exception handling, etc. Graywulf job workflows must implement a few properties (user ID, job ID, etc.) but otherwise they are standard activities.

## Workflows

The workflow runtime can execute complex, parallel workflow very efficiently and (kind of) hides parallel programming issues. If multiple branches of workflows share some objects then explicit locking might be necessary. Workflows can be suspended (between activities) and persisted in a database. Persistence and loading from the store is done entirely by the framework.

Graywulf executes the workflows on a single machine (the controller), consequently intensive tasks cannot be run inside the activities, and rather they must be delegated to the worker nodes. This is easy in case of queries, because those all run on the remote servers. Other types of activities, like the ones mentioned at the IO building blocks, must be implemented such a way that processing and data access can be delegated to a worker node. Long running activities must be implemented following an async pattern. There is a special base class for activities doing cancellable asynchronous work. Async workers must support cancellation logic, as jobs must be able to be cancelled manually, or when a time-out period is enforced by the scheduler.

In a distributed database environment single activities might fail for various reasons. For example, a single partition of a partitioned query might fail because of a single server being down for a short time. If this downtime is not detected by the scheduler (see later) the particular branch of the workflow should be able to request a rescheduling from the scheduler. Graywulf calls this retry logic and the implementation consists of a retry activity that can wrap another (usually a sequence) activity and repeat it a predefined number of times in the case of a failure. It’s the responsibility of the activities inside the retry block to request a new worker node from the scheduler.

Workflow foundation supports detailed logging of workflow execution and activities can also produce messages that are routed to the logger component by the framework. This is important, because the order of events cannot me tracked manually, but the framework takes care of it completely. Most log events are generated by the registry and those are automatically routed to the workflow runtime. Every log entry contains a context ID, and a job ID; optionally registry entity IDs can be logged too. For example, a database copy operation might log the IDs of the source and destination databases.

Graywulf predefines a few activities. Beside the aforementioned async base activity and retry activity, there are activities to export and import tables, copy files, etc.

Future plans include supporting dynamic workflows. Dynamic workflows are part of the workflow foundation, but are not yet supported by Graywulf. What they offer is, basically, dynamic composition of arbitrary workflows which will be essential is we want to implement generic distributed joins.

The workflow part is pretty much complete, basic jobs run well. Dynamic jobs are tricky. I’m planning to build them on the fly from the precompiled activities, store them as XAML in the registry and load them when the job is loaded. I wouldn’t go into this unless we get to implementing very generic distributed joins and need dynamic jobs.

## Job definitions

Jobs are activities with a predefined set of input and output parameters. Once workflows are compiled into an assembly (DLL) they will have a unique type name, for example: Jhu.Graywulf.Jobs.Test.TestJob, Jhu.Graywulf.Jobs, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null, and can be registered in Graywulf by this type name as Job Definitions. The DLLs implementing the job have to be made available to both the scheduler and the web admin interface by copying them to a specific directory on the controller machine. Job definitions tell the Graywulf system about the types of jobs and the location of the implementations only, they are not actual jobs.

## Queues

Jobs are scheduled in queues. The scheduler polls every queue for waiting jobs and starts them in a first in first out manner. Jobs can also be scheduled to execute at a given time. Jobs scheduled for a time will be started with priority. Every queue can have a maximum number of outstanding jobs, meaning that jobs from the same queue can run in parallel up to this limit. Jobs in different queues, on the other hand, are always executed in parallel.

Graywulf supports recurring jobs. Once a recurring job finishes, it gets rescheduled for the next execution automatically. This feature is useful for maintenance jobs, such as temporary directory clean ups, etc.

Queues can be defined either on the cluster level (for generic jobs, like database copy) or on the federation level (specific to an application). A queue always belongs to a cluster node, most often to the controller, but sometimes it’s useful to create queues for the individual worker nodes too.

## Job instances

Job instances are actual jobs scheduled in a queue. A job instance has a reference to its job definition and a set of input parameters. For instance, a query job has the following parameters: JobID, UserID and a Query object that contains information such as the query string, the name of the destination table in MyDB, query time-out parameter, etc. A new job is generated when a user submits a query, a file export request, etc.

# Scheduler

The Graywulf Scheduler is responsible of executing long-running jobs. In Graywulf, all jobs are implemented as workflows. Workflows are graphs of sequences of activities that have to be executed in a particular order. Activities can be inside loops and certain loops can be executed in parallel. Besides executing jobs, the scheduler is also responsible for assigning worker nodes to the jobs on request, and following the status of the system components.

The scheduler itself is complex because of the way events are routed between appdomains. Everything has to be handled at the right place and time. Jobs run well already but some common exceptions are not handled on the poller thread. The main missing parts are the regular reloading of the system configuration and active monitoring of the worker nodes. I think these are all high priority, because the scheduler has to be really reliable. What the scheduler can do right now “schedulingwise” is that it accepts a set of database names and it can tell a workflow which server has all those databases, so the workflow can execute all the queries on those. Servers are given to workflows in round robin for mirrored databases. This is still a very simple implementation, generic distributed joins will require much more logic that has to go into the scheduler.

The scheduler runs as a single instance right now but could be easily modified to run on multiple machines in failover mode. The issue is that although two concurrent pollers cannot pick the same job up twice but they can pick up twice as many jobs as the number of allowed outstanding jobs. This is simple to fix, but there might be more problems I haven’t thought about yet.

## Schedule startup

On startup, the Graywulf scheduler loads the system configuration from the Registry. This takes a few seconds, so, for performance reasons, the configuration has to be kept in memory. Because the system configuration might change during system operation, but the administration tools can only write into the registry but cannot communicate with the scheduler directly, it is necessary, to reread the system configuration once in a while from the registry. This feature is not implemented yet.

## Job execution

Once the configuration is read, the scheduler starts polling the registry for new jobs in each queue. Polling is done in every second, so scheduled jobs will suffer at least a few seconds of lag. When a new job is found in a queue it is loaded by the scheduler and passed to the workflow runtime for execution.

Running jobs can be manually cancelled by submitting a cancel request. Cancel requests are written to the registry and the job poller processes them. One a cancel request is received, the scheduler instructs the workflow runtime to start the cancellation of a job. Time-out enforcement works a similar way. Once the scheduler detects that a job has been running for a predefined amount of time, it instructs the scheduler to gracefully cancel the job.

Jobs might throw exception up on system or programming (hopefully only the former) errors. Exceptions are handled by workflow foundation, jobs are terminated gracefully and the details of the exceptions are logged.

## Stopping the scheduler

Stopping the scheduler is not a simple process as long-running jobs have to be drained or paused first. Workflow foundation supports persisting jobs into a database between activities which feature is used by the scheduler. Stopping the scheduler, however, can still take a significant time, if activities run too long.

Graceful stopping is not fully functional yet, it might just hang, especially with persistence. Needs to be tested systematically.

## System monitoring

To be able to schedule queries to live database server nodes only, the scheduler should keep track of the status of the system. This includes regular checking of the network connection, the running state of the database server, memory and disk usage, etc. This feature is not implemented yet.

## Job binaries

Jobs are workflows that are compiled into .Net binaries (DLLs). To allow adding new types of jobs to the system without restarting the scheduler, job DLL references are resolved dynamically. The scheduler can look for binaries in a hierarchy of directories to find the necessary components to execute a job. Even multiple versions of the same job can be executed in parallel.

To conserve memory, and also to support different version of the same job, jobs are executed in different app domains. The scheduler implements some simple heuristics to schedule jobs in the right app domain. Once app domains are not used for a given period of time, they get unloaded.

AppDomain based loading of assemblies is in place but version handling is not complete yet. Also, there are issues here, because DLLs can dynamically load other DLLs and we should be able to find those automatically too. Not a very big problem, but needs testing and quite a bit of knowledge about DLL versioning in .Net.

# Task delegation via Remote Service

The Graywulf scheduler runs on a single machine and executes jobs inside the scheduler process. As a consequence, computations should not be done on the scheduler machine but rather delegated to the worker nodes. SQL query execution is always delegated to the database servers but for generic task Graywulf offers a very simple solution.

Each cluster node runs a simple service that can accept remote request to create a component, parameterize it and execute it. This is called the Graywulf Remote Service module which is basically a simple application server. It is entirely based on the .Net Communication Foundation. Components are implemented ad .Net classes with interface used for communication contracts. Component activation is done by the clients by sending a request to the server with the full name and assembly information of the component. The server finds the assembly and instantiates the component. Graywulf components should support async execution and cancellation.

The remote service is pretty mature already but needs testing, especially the assembly resolver part, that finds the appropriate version of an assembly. Theoretically, it can use new versions of the same components for any subsequent call after the new version is installed (copied to a directory, to keep is simple), but I haven’t really tested is, I always restart the service if I change something. Also, there’s no real deploy scripts to automate copying all the newly compiled stuff to the server nodes. This is what I intended deployment packages in the registry for, but we could also use some existing tool for this.

Right now everything runs under a Graywulf account now, but impersonation of the client would be a great option. It is hard to set up, though, with Active Directory’s Kerberos. I could get it running in my dev environment, but at JHU touching the AD is risky.

We should also look at Windows Process Activation, a relatively new technology. I don’t know if it has the features we need. Hopefully, it’s simple enough and can easily be deployed. I’m not sure if it exists in Windows 2008 R1 though (platform of the GW nodes). The good old Component Services (COM+) is a big no-no.

# Logging

What log contains

How to navigate log

Debugging practices

Logging works nicely but breaks the whole system if the log database is down. Logging is synchronous (if not done from a work flow, that is). Async logging would give much better performance in case of the web interfaces.

The log browser is very immature and navigating the log is not really possible. I use queries to read what I need from the log right now, so a nice browser would help a lot.

# System setup

How to build, install and configure a Graywulf cluster

## Hardware configuration

## Platform configuration

### Hardware configuration

### Windows configuration

### Database Server configuration

#### Configure tempdb

By default, tempdb files are located on the system disk. To move them to the RAID volume, execute:

ALTER DATABASE tempdb

MODIFY FILE (NAME = tempdev, NEWNAME = tempdev\_0, FILENAME = '[path\_to\_data\_volume\_0]\tempdev\_0')

ALTER DATABASE tempdb

MODIFY FILE (NAME = templog, NEWNAME = templog\_0, FILENAME = '[path\_to\_data\_volume\_0]\templog\_0')

Restart the server for the changes to take effect. It is important to move the files first and set the file size later because files are moved to the new location only after a server restart and growing the files residing ont he system volume may eat up all the disk space and crash the system.

In case of multiple volumes, add more files:

ALTER DATABASE tempdb

ADD FILE (NAME = tempdev\_1, FILENAME = '[path\_to\_data\_volume\_1]\tempdev\_1', SIZE = 50GB, FILEGROWTH = 0)

ALTER DATABASE tempdb

ADD LOG FILE (NAME = templog\_1, FILENAME = '[path\_to\_data\_volume\_1]\templog\_1', SIZE = 10GB, FILEGROWTH = 0)

You can verify the settings by executing

exec sp\_helpfile

It is very important to turn off file growth as a runaway query may easily eat up all the disk space on a server.

### Setting up a Windows account for the services.

In order to centrally manage security of the system, a Windows domain account is required. All Graywulf services will run under or impersonate themselves under this domain account. We suggest to name this account MYDOMAIN\Graywulf. By default, the account should only have basic domain user privileges with no remote desktop access.

The following checklist can be used to configure the permission this account must have:

* Member of domain users
* Full access to all data directories on the worker nodes
* Full access to the network shares of all data directories of the worker nodes
* Windows user account added on all SQL Servers
* Member of dbcreator role on all SQL servers on the worker nodes
* Full control access to %windir%\temp on the webserver

Create graywulf domain account, very important to run everything under this account

* grant access to the shared directories, pay attention to mounted volumes!
* DB creator under sql server
* run service under this account
* grant access to %windir%\temp so serializer classes can be generated on the web server

Setting up the Graywulf Registry

Setting up the log

Setting up the database for workflow persistence

- Create an empty database called Graywulf\_Persistence

- Run SqlWorkflowInstanceStoreSchema.sql and SqlWorkflowInstanceStoreLogic.sql located in %windir%\Microsoft.NET\Framework64\v4.0.30319\SQL\en

### Installing the bulk-op server

Open port 5055

C:\Windows\Microsoft.NET\Framework64\v4.0.30319\InstallUtil.exe .\gwrsvr.exe

By default the service is installed as GWRSvr. By specifying the /svcname to the installer, this name can be overridden, for instance, to install a debug version of the service:

C:\Windows\Microsoft.NET\Framework64\v4.0.30319\InstallUtil.exe /svcname=debug .\gwrsvr.exe

### Setting up the Graywulf Cluster Administration Console

How to run the admin console with windows credentials

Creating a Federation

Configuring MyDB location

Configuring existing monolithic databases

Setting up the front-end for a federation

# Developing Databases for Graywulf

# Updating worker nodes

On the GW nodes:

icm gw06, gw07, gw08, gw09, gw10 { net stop GWRSvr\_debug }

foreach ($s in "gw06", "gw07", "gw08", "gw09", "gw10") { cp \* \\$s\data\data0\Graywulf\debug -Force -Recurse }

icm gw06, gw07, gw08, gw09, gw10 { net start GWRSvr\_debug }

# Credits

# References

# Copyright