

Digi-Sense® Data Mover



[Introduction](#)

[Open Source](#)

[Prepare, Transform and Store](#)

[Digi-Sense® Data Mover Features](#)

[Quick Start](#)

[Launch the Panda 🐼](#)

[How Does Data Mover Works](#)

[Schedule](#)

[Job Chain](#)

[Transactions](#)

[Connecting to a Database](#)

[MySQL](#)

[PostgreSQL](#)

[SQLite](#)

[SQL Server](#)

[Globals](#)

[Variables](#)

[Scripting](#)

[Scripting and Context Hooks \(before, after, context\)](#)

[Remote Execution](#)

[Delegate Execution to a Remote Node](#)



Introduction

In computing, a data warehouse (DW or DWH), also known as an enterprise data warehouse (EDW), is a system used for reporting and data analysis and is considered a core component of business intelligence.

DWs are central repositories of integrated data from one or more disparate sources. They store current and historical data in one single place that are used for creating analytical reports for workers throughout the enterprise.

The data stored in the warehouse is uploaded from the operational systems (such as marketing or sales).

The data may pass through an operational data store and may require data cleansing for additional operations to ensure data quality before it is used in the DW for reporting.

Extract, transform, load (ETL) and *extract, load, transform* (ELT) are the two main approaches used to build a data warehouse system.

Moving data from one system to another is a key part of your analytical ecosystem operations. Well-planned, monitored, and managed data movement is essential to effective operation of your overall ecosystem.

Sometimes you only need a one-time copy of data from your enterprise data warehouse (EDW) to your test system. Other times, you must regularly synchronize a dual system environment.

What you need is a solution that automates the data movement process, a solution that also accommodates the many situations found within your comprehensive analytical ecosystem.

And that's exactly what **Digi-Sense® Data Mover** delivers to you and your business all the supporting capabilities you need to easily, effectively—and affordably—meet your data movement demands.



Open Source

Digi-Sense® Data Mover is an Open Source project with a double license:

- `non-commercial license`: free to use for single persons
- `commercial`: require a license

Data Mover is distributed under MIT license for non-commercial use.

If you use it as a tool for your own projects, you can use Data Mover under MIT license.

NON-COMMERCIAL: non-commercial is that no money should be exchanged as part of the transaction of using the materials – regardless of whether the money represents a break-even of marginal cost, reimbursement or profit.

Commercial Use License

If you are a company that sells projects to its customers and need Data Mover, you should ask for a Commercial License.

For Commercial License, please write to angelo.geminiani@ggtechnologies.sm

Sources:

<https://bitbucket.org/digi-sense/gg-progr-datamover/src/master/>

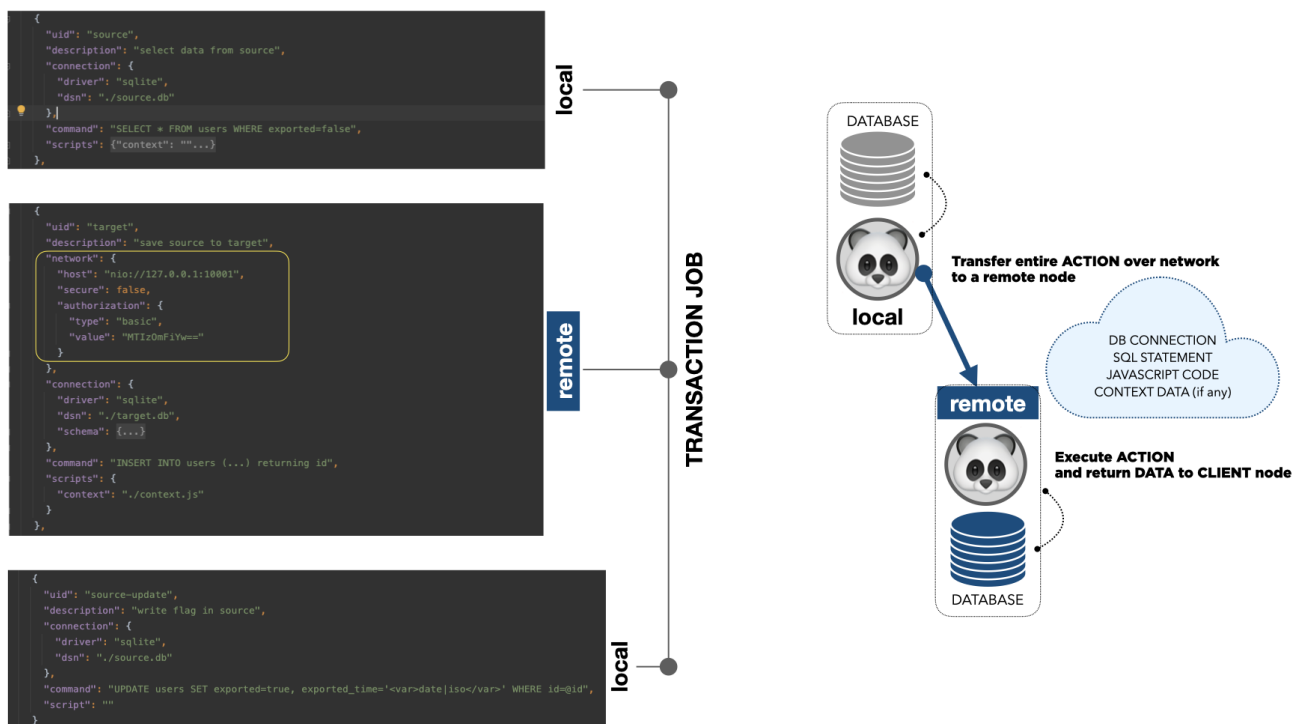


Prepare, Transform and Store

Digi-Sense® Data Mover is a “multi-node” software that allows you to prepare, transform and store data into a new data layer.

With **Digi-Sense® Data Mover** It is possible to extract data from source systems, transform them, and load them into a data mart or warehouse. That’s what is commonly defined as “Data Integration”.

Digi-Sense® Data Mover brings its data integration capabilities to the next level: it works in a “multi-node” environment.



With **Digi-Sense® Data Mover** you can delegate task execution at peripheral nodes if it is required.



Digi-Sense® Data Mover Features

Data Mover features are:

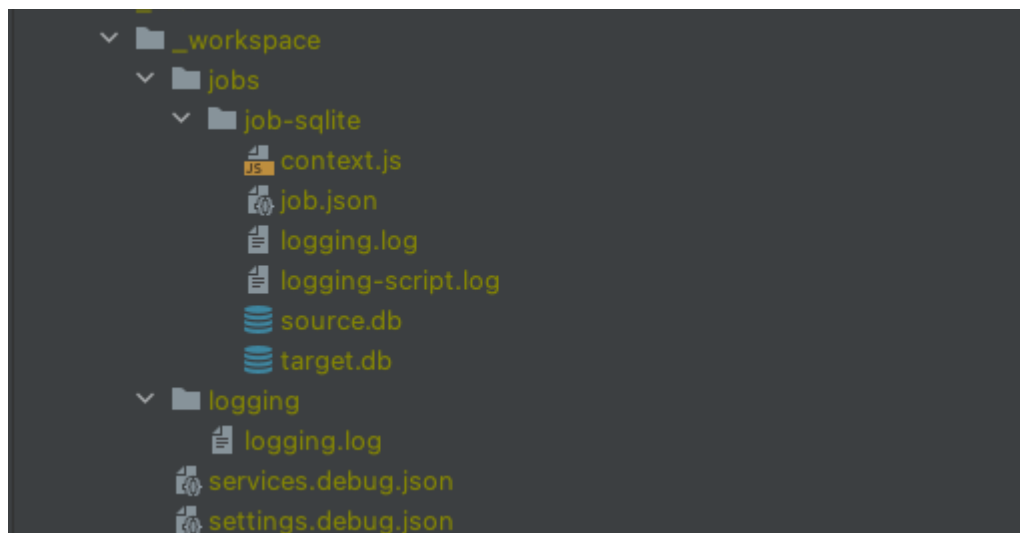
- **Multi database:** SQLite, SQL Server, MySQL/MariaDB, Postgres
- **Auto Migrate Schema:** can migrate/update database schema
- **Scheduler:** jobs can be scheduled and start every x time or just only once a day at the set time
- **Move Data** from Source to Target database: this is Data Mover main feature 🐼
- **Multi Node Architecture:** Data Mover allow to create a network of remote node for edge execution
- **Javascript Native Engine:** With javascript you can customize behavior and transform data
- **Enable HTTP endpoints:**

Quick Start

🐼 Data Mover works on a specific directory named "datamover_workspace" (you can change the default name launching the executable with a parameter).

When Data Mover starts, first of all look for its workspace. If a workspace is not found, Data Mover creates one using the start path a workspace root.

Below is a workspace directory example, named "_workspace".



NOTE: this workspace folder contains also a job named "job-sqlite" with two sample databases, but this is only an example and your databases can be placed anywhere.



The workspace directory contains a "logging" directory and "jobs" directory.
The "jobs" directory should contain subdirectories, one for each job you want to run.

Example:

```
datamover_workspace
|----- jobs
|      |---- job-1
|      |      |--- (job file here...)
|      |---- job-2
|      |      |--- (job file here...)
|      |---- ...
```

TIP: First time consider to start Data Mover in debug mode and look at generated log to ensure all is working fine. If all is working fine you should see a log file in each job directory and also some auto generated json files with your database schema.

Launch the Panda 🐼

Data Mover supports some commands and parameters.

```
# sample batch file to run executable
datamover run -dir_work=./_workspace -m=production
```

COMMANDS:

- run : tell Data Mover you want run the entire program (more commands will come in future)

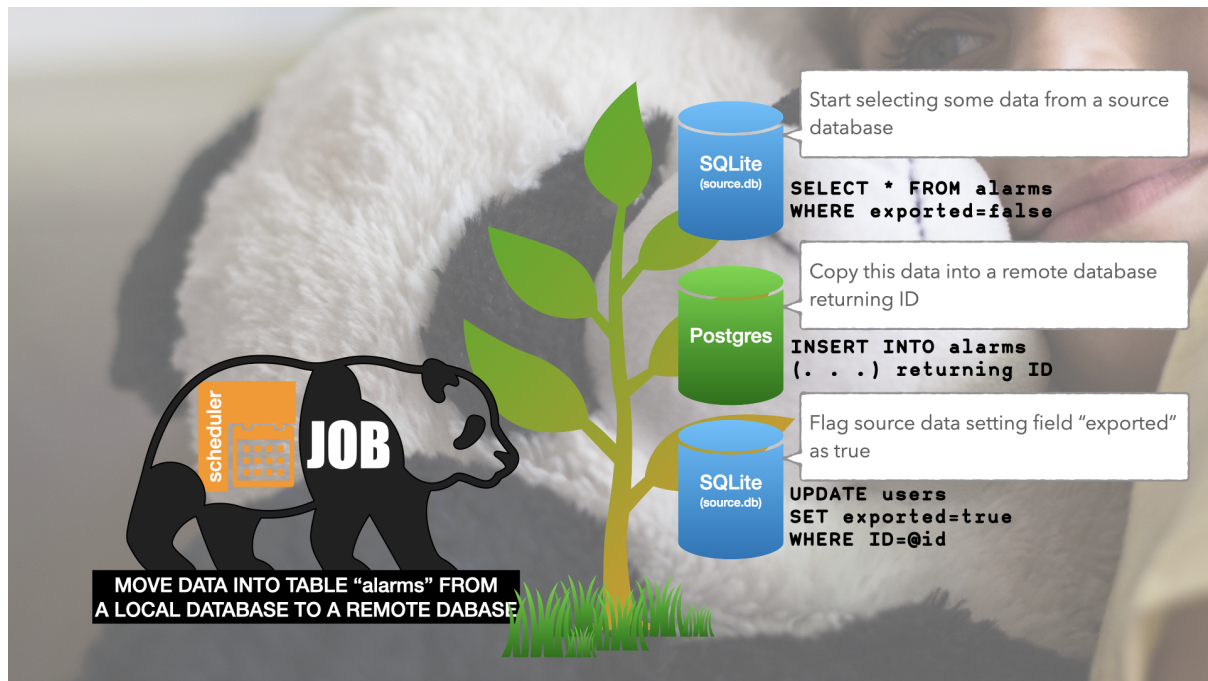
PARAMETERS:

- -dir_work: path of workspace. Can be absolute or relative path
- -m: mode. Can be debug or production. The debug mode is useful if you need a verbose log.

For binary files, please look [here](#).



How Does Data Mover Works



Data Mover hosts "jobs".

A "Job" is basically a JSON file describing what Data Mover should do at a predefined time.

Each "job" contains:

- Schedule: optional data to define WHEN the job must start
- Transaction: an array of "Action" to define WHAT the job must do

So, jobs define WHEN and WHAT about Data Mover.

Let's start analyzing the anatomy of a Data Mover job.

Schedule

Schedule collects some settings to define a timed task. Data Mover has an internal task manager and a scheduler working on a thread safe environment.

```
{
  "schedule": {
    "start_at": "",
  },
}
```



```
  "timeline": "second:3"
}
```

Is quite simple figure how Schedule works:

- **start_at**: optional value representing hour and minute. ex: "10:20", "18:30", etc..
- **timeline**: optional value representing a key-pair "unit:value". ex: "millisecond:100", "second:3", "minute:10", "hour: 24".

Schedule is optional at all.

If you do not specify any value, the job will not be scheduled, but remain a valid job that can be invoked from another job (see below about "Job Chains").

Job Chain

```
{
  "schedule": {
    "start_at": "",
    "timeline": "second:10"
  },
  "next_run": "job-sqlite-users"
}
```

Not all jobs must be scheduled. Sometimes you should prefer to schedule a master job and define different jobs for some other tasks to invoke after the master job.

That's a chain.

"next_run" is the field that tells a job what to do next.



Transactions

```
{
  "transaction": [
    {
      "uid": "source",
      "description": "select data from source",
      "connection": {
        "driver": "sqlite",
        "dsn": "./source.db"
      },
      "command": "SELECT * FROM users WHERE exported=false",
      "script": ""
    },
    {
      "uid": "target",
      "description": "save source to target",
      "connection": {
        "driver": "sqlite",
        "dsn": "./target.db",
        "schema": {}
      },
      "command": "INSERT INTO users (...) returning id",
      "fields_mapping": null,
      "script": ""
    },
    {
      "uid": "source-update",
      "description": "write flag in source",
      "connection": {
        "driver": "sqlite",
        "dsn": "./source.db"
      },
      "command": "UPDATE users SET exported=true, exported_time='<var>date|iso</var>' WHERE id=@id",
      "script": ""
    }
  ]
}
```

A transaction is an array of actions.

This is a sample action:

```
{
  "uid": "source",
  "description": "select data from source",
  "connection": {
    "driver": "sqlite",
    "dsn": "./source.db"
  },
  "command": "SELECT * FROM users WHERE exported=false",
  "script": ""
}
```



This action works on a SQLite db (./source.db) and selects all not exported yet data from the user table.

That's all. But transactions work using actions that interact together creating an execution context, a transaction context.

The context enables actions to share data during transaction execution.

And here comes our first action: this action selects data and keeps them in context for the next action.

Next action will do something new with data in context:

```
{
  "uid": "target",
  "description": "save source to target",
  "connection": {
    "driver": "sqlite",
    "dsn": "./target.db",
    "schema": {}
  },
  "command": "INSERT INTO users (...) returning id",
  "script": ""
}
```

This action, using context data created from the first action, executes an SQL Formula on a target database.

```
INSERT INTO users (...) returning id
```

This SQL Formula is a custom SQL like command with a special statement: (...)

The three-dots-statement 🌿 tells the panda 🐼 of Data Mover to extract all fields and values from the datasource into context and execute an INSERT for each row in the context.

So Data Mover's panda will start moving row after row into the context to the target database executing an INSERT statement for each source row.

Reassuming:

- First we selected some rows from a datasource
- then we started a loop on each row and executed an INSERT into a target database. The insert command was auto-completed from the panda 🐼 of Data Mover that is able to understand a three-dot-statement 🌿.

Now, to complete the transaction, we should mark all source data as exported.



And here comes our third action in Transaction:



```
{
  "uid": "source-update",
  "description": "write flag in source",
  "connection": {
    "driver": "sqlite",
    "dsn": "./source.db"
  },
  "command": "UPDATE users SET exported=true, exported_time='<var>date|iso</var>' WHERE id=@id",
  "scripts": {}
}
```

The third action, using the same context, now tries to execute a new UPDATE command into the source database, setting all fields as "exported".

The "SQL Formula" (a Data Mover special SQL commands) is:

```
UPDATE users SET exported=true, exported_time='<var>date|iso</var>' WHERE id=@id
```

In this statement we have two strange things:

-  `<var>date|iso</var>`: a Special Expression
-  `@id`: a named parameter

Nothing magic, just panda 🐼 style.

Data Mover's panda is also able to interpret some Special Expressions like the one above (that return an ISO-8601 date time).

Otherwise, the `@id` named parameter uses the context to get a value for each loop.

That's all. We just wrote three simple ACTIONS and the panda 🐼 did all the job.

Connecting to a Database

Data Mover officially supports databases:

- MySQL,
- PostgreSQL,
- SQLite,
- SQL Server.



MySQL

```
{
  "driver": "mysql",
  "dsn": "root:root@tcp(127.0.0.1:3306)/test?charset=utf8mb4&parseTime=True&loc=Local"
}
```

PostgreSQL

```
{
  "driver": "postgres",
  "dsn": "host=localhost user=postgres password=Postgres1234 dbname=gorm port=5432 sslmode=disable T"
}
```

SQLite

```
{
  "driver": "sqlite",
  "dsn": "./source.db"
}
```

SQL Server

```
{
  "driver": "sqlserver",
  "dsn": "sqlserver://user:passwordxxx@localhost:9930?database=test"
}
```



Globals

Globals are **special configuration parameters** that are grouped in a single configuration file.

Globals parameters are:

- **Connections:** shared connections to use in jobs
- **Constants:** similar to job variables, but cannot change value and are shared between all jobs. Constants can be used in QUERY, exactly like variables

SAMPLE globals.debug.json file:

```
{
  "constants": {
    "sample_constant": "this is a sample constant"
  },
  "connections": [
    {
      "connections-id": "sqlite-001",
      "driver": "sqlite",
      "dsn": "./data.db",
      "schema": null
    }
  ]
}
```

If you want to use a connection in a job, just set the "connections-id" value into job:

```
{
  "uid": "source-update",
  "description": "write flag in source",
  "connection": {
    "connections-id": "sqlite-001"
  },
  "command": "UPDATE users SET exported=true, exported_time='<var>date|iso</var>' WHERE id=@id",
  "scripts": {}
}
```

Globals are quite useful when you want to centralize connection declarations or have constant values to use into your jobs.

Variables

In JOB's configuration file it is possible to declare "variables".



```
{
  "variables": {
    "vlimit": 1,
    "voffset": 0
  }
}
```

Variables can be used in query or javascript (javascript can also alter variables value or add new variables).

Here is a sample query using variables:

```
SELECT *
FROM alarmlogview LIMIT @vlimit
OFFSET @voffset
```

"vlimit" and "voffset" are two special variables.

Data Mover handle these variables as internal variables and automatically increment the value of voffset at each execution.

Below is a full job.json file containing variables.

```
{
  "schedule": {
    "uid": "3-seconds-run",
    "start_at": "",
    "timeline": "second:3"
  },
  "next_run": "",
  "variables": {
    "vlimit": 1,
    "voffset": 18
  },
  "transaction": [
    {
      "uid": "source",
      "description": "select data from source",
      "network": null,
      "connection": {
        "driver": "mysql",
        "dsn": "root:root@tcp(127.0.0.1:3306)/test?charset=utf8mb4\u0026parseTime=True\u0026loc=Local",
        "schema": null
      },
      "command": "SELECT * FROM alarmlogview LIMIT @vlimit OFFSET @voffset",
      "scripts": null
    }
  ]
}
```



In this case `@vlimit` and `@voffset` are very useful because the table “alarmlogview” may contain a thousand records and this should compromise performance or even system memory when Data Mover work to move data from a database to another.

WARNING: Please, consider using `@vlimit` and `@voffset` at least each time you use a remote data transfer. Moving data over the network may result very slow if you are moving megabytes of data.



Scripting

Data Mover has an internal javascript engine.

Let's take a look at configuration:

```
{
  "uid": "source-update",
  "description": "write flag in source",
  "connection": {
    "driver": "sqlite",
    "dsn": "./source.db"
  },
  "command": "UPDATE users SET exported=true, exported_time='<var>date|iso</var>' WHERE id=@id",
  "scripts": {
    "before": "",
    "context": "./context.js",
    "after": ""
  }
}
```

The "scripts" field contains files to run in predetermined moments.

The javascript engine is native and is written completely in Go. The Go runtime creates a new js engine instance before each Action execution, adding some context data to the runtime environment.

Those context data are accessible to js runtime using some defined variables:

- **\$data**: represent an array of data row that are the result of an SQL command.
- **\$variables**: represent variables defined into settings or programmatically

NOTE: the javascript engine is much more powerful and can send emails, SMS, dispatch messages over https, MQTT, and even more. More documentation will come soon.

SUPPORTED HOOKS:

- **before**: is a script that is launched before a query or SQL command is executed. Use this to prepare or transform data before an INSERT or UPDATE
- **context**: is a script that is launched when "transaction execution context is created"
- **after**: is a script that is launched after a query or SQL command is executed

EXECUTION FLOW

Script execution order is always like:

1. BEFORE



2. CONTEXT

3. AFTER

There is a difference only when DataMover cycle on transaction task.

First action has no context, so the "context" script is executed after the SQL statement.

This is how DataMover invoke scripts:

FIRST ACTION:

1. - before
2. - SQL STATEMENT (retrieve data or execute command)
3. - context
4. - after

OTHER ACTIONS:

1. - before
2. - context
3. - SQL STATEMENT (retrieve data or execute command)
4. - after



Scripting and Context Hooks (before, after, context)

```
(function () {  
  console.log("Calling context.js, LENGTH=", $data.length)  
  for (const item of $data) {  
    item["number"] = item["number"] + 1;  
  }  
  
  // return optionally changed data  
  return {  
    "data": $data,  
    "variables": $variables  
  }  
})();
```

The context hooks can be used to alter data or remove arbitrary rows from the context data.

Scripts are defined into the "scripts" configuration field.

```
{  
  ...  
  "command": "INSERT INTO users (...) returning id",  
  "fields_mapping": null,  
  "scripts": {  
    "context": "./context.js"  
  },  
  ...  
}
```

Context hooks can be used to alter dataset (internal variable `$data`), alter variables (`$variables`) or perform advanced actions like send email, SMS or even HTTP requests and more.



Remote Execution

When you need to access a database that does not export its port to a public network, here comes the Data Mover "remote execution" feature.

Data Mover has the ability to create NODES.

Nodes are Data Mover executable, nothing else. When a Data Mover executable is exposed to a public network (with a public IP), it can open a TCP port and handle remote commands from other Data Mover nodes.

This is a sample Data Mover configuration file (`services.production.json`) that enable this feature:

```
{
  "enabled": true,
  "authorization": {
    "type": "basic",
    "value": "MTIzOmFiYW=="
  },
  "services": [
    {
      "enabled": true,
      "name": "NIO SERVER",
      "protocol": "nio",
      "protocol_configuration": {
        "port": 10001
      }
    }
  ]
}
```

If a Data Mover finds such a configuration file (`services.<MODE>.json`) it configure itself as a Data Mover Node and expose its runtime at the Data Mover Chain.

NOTE: The NIO protocol is an implementation of TCP using auto generated public/private keys for each message.

Data Mover Nodes communicate exchanging certificates before each message.



Delegate Execution to a Remote Node

Let me call CLIENT the node that delegates to another Node the execution of the Action.

Actions must be defined on the client.

An Action contains information for a local connection to a database, an SQL statement and optionally a javascript file. All of those things must be declared on the CLIENT.

Therefore, the SERVER can remain agnostic about the job it will receive.

Here is how we delegate Actions to a remote server:

```
{
  "uid": "target",
  "description": "save source to target",
  "network": {
    "host": "nio://remote.host.com:10001",
    "secure": false,
    "authorization": {
      "type": "basic",
      "value": "MTIzOmFiYw=="
    }
  }
},
...
```

The Action declared into the Job configuration file contains a field named network.

Network field is an object containing:

- **host**: URL or remote node to delegate execution
- **secure**: Enable/Disable encryption (Encryption is more secure, but slow). NIO data are always encoded, packed into proprietary binary format and sent using a specific pattern (so is quite difficult to act as man-in-the-middle or read passing data).
- **authorization**: Authorization mode and token for client authentication on remote node. Only authenticated clients can send commands.

