

INMEMORY.NET

Manual

Table of Contents

IRDB / InMemory.Net.....	6
What is InMemory.Net?.....	7
What is an in-memory database?.....	7
InMemory.Net typical import speed	7
InMemory.Net Configuration	8
IRDB Server (irdbServer.exe)	8
IRDB.INI	9
IRDB Import.....	11
Importing into IRDB	12
IRDB Import - Reference Guide.....	13
Specifying a Source	13
ODBC Datasource.....	14
OLEDB Datasource	14
SQL Server Datasource.....	14
Local InmemoryDB Datasource	15
LazyLoad Local InmemoryDB Datasource.....	15
Remote IRDB Datasources	15
DOT NET Datasource.....	16
CSV Datasource	16
ME Datasource.....	18
CLOSE Command.....	19
Localization Settings/ Culture	20
ImportTimeout.....	20
Import	21
IRDB Import - Optimisation.....	23
USEDISK.....	23
USEDISK FLUSH	23
DO PARALLEL END PARALLEL	24
Slurp (IMPORT ALL FROM SOURCE)	24

Execute.....	26
Drop	26
RENAME	27
REORDER.....	27
Try/ Catch.....	27
Exit.....	28
Add a Column (Alter Table).....	28
Add Constant Column Command	28
Embedded SQL.....	29
Create Table/ Insert Into ... VALUES	29
COLUMNIZE Command	30
UNCOLUMNIZED IMPORT MODE.....	30
DECOLUMNIZE Command.....	30
SET PRESERVE DECIMAL.....	31
FLUSH	31
Delete.....	32
Update.....	32
Clustered Index	32
Save	33
IRDB Import - programmable elements.....	34
Declare	34
Parameterizing Imports	34
Command Line Parameters	34
Parameterizing Connection Strings and Tables to Import.....	35
Set	36
Running dynamically generated scripts using ScriptExec.....	37
Running external Programs with CMDExec	37
Running external SQL Scripts with SQLExec	37
Accessing field values in a table.....	37
MoveFirst & MoveNext.....	38
Iterating Through a Recordset - For & While Loops	38
Print.....	39

If	39
ASSERT	39
IRDB Import Functions	40
Picking up new IRDB data	40
IrdLoad	41
Exporting from IRDB	42
CSV Export.....	42
EXPORT to SQL SERVER.....	42
DEBUGGING With BREAKPOINT.....	44
IRDB Query	45
Starting IRDB Query	46
Open Database.....	46
Open Database (Lazy Load).....	46
Connect to Server	47
IRDB QUERY -How to Use	47
Searching for a Database/ Table/ Field.....	48
New Query (CTRL-N)	48
Add/ Remove a Comment (CTRL-plus/ CTRL-minus).....	49
Select a Database	49
Row Limit	50
Execute Query (F5).....	50
DBStats.....	50
IRDB SQL Reference.....	51
SELECT	51
SELECT DISTINCT	52
SELECT *	52
NOCACHE	54
CACHE	54
CASE STATEMENTS.....	54
COUNT.....	54
COUNT (DISTINCT)	55
SUM (DISTINCT)	55

OVER() / OVER (PARTITION BY).....	56
JOINS	56
CROSS JOIN	57
PIVOT.....	57
WHERE	57
GROUP BY.....	58
WITH ROLLUP & GROUPING Function	58
ORDER BY	58
HAVING	59
UNION ALL.....	59
INTERSECT	59
EXCEPT	60
LIMIT	60
Correlated Subqueries	60
DELETE.....	61
UPDATE	61
INSERT	61
CREATE TABLE – DROP TABLE	62
Temp Table Support (#tempTable)	62
DECLARE and SET	63
IF statement	63
WHILE statement	64
TRY CATCH statement.....	64
With statement.....	65
Clustered Index	65
Columnize Statement.....	65
Print Statement.....	65
CREATE VIEW – DROP VIEW	66
Stored Procedures	66
EXEC Dynamic SQL Statement	66
SCALAR User Defined Functions	67
Table Value Functions.....	67

Inline Table Value Functions	68
SPLIT_TABLE Table Value Function	68
Table & Column Information	68
Functions.....	70
Function Types.....	71
<i>Note: FILEEXISTS is available within IRDBImport script only.</i>	72
Function Syntax.....	73
Special Functions	85
Timeshift	85
OPENBAL / CLOSEBAL	85
LastChild / LastPrice / LastPriceWithZeroes	86
IRDB Further Reading.....	88
IRDB with Interactive Reporting	89
Programming Overview	90
Performance Characteristics.....	91

IRDB / InMemory.Net

What is InMemory.Net?

InMemory.Net is a read-only Dot Net (4.0+) based columnar in-memory database designed for use within the Microsoft Dot Net environment. For shortcut purposes we call it IRDB.

What is an in-memory database?

An in-memory database primarily relies on main memory for data storage in contrast to more traditional database management systems that employ a disk storage mechanism. Once a database is loaded, data is “in memory” and ready-to-go, therefore it is able to retrieve and present information far more quickly than previously possible.

Following importation, an in-memory database (*InMemory.Net at least*) can be used as a data warehouse to be refreshed only when an update is required. You do not need to reload from a third-party source each time your pc is reset.

InMemory.Net typical import speed

Import speed into InMemory.Net varies depending on connection type to the source database. Data can be imported via ODBC, OLEDB or Native Dot Net Providers.

In tests, we've found the following to be speeds typical of a Sql Server import:

ODBC link	20,000 records per second
OleDB	40,000 records per second
Native Dot Net Provider	170,000 records per second

InMemory.Net Configuration

The IRDB Server is run as a Windows Service, installed by the InMemory.Net installer.

InMemory.Net is comprised of a number of applications, namely:

IRDB Server	...	<i>In Memory Server application</i>
IRDB Import	...	<i>Import data from another source, in effect, to build a database</i>
IRDB Query	...	<i>run SQL queries/ view data structure, etc. on an IRDB</i>
IRDB Load	...	<i>Load/Reload an IRDB file into memory, ready for data-access</i>

IRDB Server (irdbServer.exe)

IRDB Server is run automatically on installation of InMemory.NET and subsequently on system reboot.

It is possible to stop and start IRDB Server with the following command-line commands

```
net stop "irdb server"  
net start "irdb server"
```

IRDB.INI

irdb.ini contains system configuration details such as the port number for IRDB, password, data directory, whether cache is turned on or not, etc.

It is placed in the install directory for irdbServer. All configuration parameters currently go into the general section.

The main required ones are set by the installer:

data_dir Directory of the data files.
port Port to run the irdbServer on. Default value is 5060
password Password to validate against when clients connect. Default is !TopSecret

Optional parameters include:

nocache Setting this to **true**, will cause subselect inner queries not to be cached. This causes nocache to become the default behaviour.

maxcpus Specify an integer here to limit the maximum number of cores that that IRDB uses to process a query. When running on computers with large number of cores > 16, setting a value of 50% of the number of cores, can help performance.

unload_timer Specify an integer here that causes inactive databases to be unloaded after that amount of time in minutes. This helps reduce memory consumption, when you have large numbers of databases, of which only a % are being used concurrently. Default is 0, which causes inactive databases not to be unloaded.

unload temptables Specify an integer here that causes inactive temp to be released automatically after that amount of time in minutes. The system default is 30 minutes.

log_dir Lets you specify a directory to place the irdb.log irdbServer log file.

Querycache =true ... allows results of queries to be cached and reused as necessary (saving time).

Querycache_timeout =20 can be used in irdb.ini to change the default timeout, e.g. to 20 minutes.

LogAll Setting this to **false** will reduce the amount of data logged on a query. No Timing, sql or other blurb information is logged. This helps performance when running large numbers of queries.

LazyLoad Setting this to **true** will enable the LazyLoad mode. This loads columns on demand, and reduces memory consumption.

lazyload_unload_column_timer Setting this to a number e.g. 15 will cause database columns to be unloaded after 15 minutes of inactivity, when lazyload is set.

windowsauth Setting this to **true**, makes the server run in Windows Authentication mode.

windowsauth_role When windowsauth is set, this specifies a role that is allowed to connect. Default value is Administrators.

encrypt Setting this to **true**, will make the server run in encrypted mode.

encrypt_cert Specifies the file location of the server certificate for encryption. (PFX File)

encrypt_password Specifies the encryption password for the cert specified by encrypt_cert

skiptables Specific a comma separate list of tables / or table patterns not to load. Patterns use sql like syntax. E.g. skiptables=stage% would cause any table starting with stage not to load. Designed to help reduce memory consumption by specifying certain tables not to load.

query_log_size Setting this to a non zero number e.g. 10000 will cause the server to remember the last 10,000 queries for each database. This information can then be retrieved with select * from information_schema.query_log

Note: If you change the irdb.ini file, you will need to restart the service for the changes to take effect.

IRDB Import

Importing into IRDB

IRDB Import is a command line utility used to import data into IRDB (build a database).

IRDBImport.exe takes one main parameter which is the name of the import file to read in e.g. From the command line, type

```
irdbimport northwind.imp
```

This executes an import file called northwind.imp and stores the data as northwind.irdb in the same folder.

Additional parameters can be referenced as variables inside the script.

Note 1: If you would like to run the example imports (supplied with IRDB) based on the Northwind database, ensure you have a Northwind database available.

–e.g. MS Access file “northwind.mdb” can be downloaded from multiple sources.

Next create a data connection to this source.

–e.g. 32bit or 64bit ODBC driver, called “**ir_northwind**” (as used in northwind.imp), pointing to this database.

Note 2: You can edit northwind.imp with a text editor to see what it is doing.

So, in this case, assuming IRDB was installed to d:\irdb, you can run this import file by opening a command prompt, going to the d:\irdb\data folder, then entering

```
..\irdbimport northwind.imp
```

Alternatively, from anywhere

```
d:\irdb\irdbimport d:\irdb\data\northwind.imp
```

This calls on irdbimport.exe to run the .imp file.

Note 3: .imp is the standard extension for IRDB import scripts. It generates a .irdb data file of the same name in the same directory.

Note 4: *irdbImport.exe runs in 64bit mode in a 64bit environment and in 32bit mode in a 32bit environment.*

To connect to a 32bit data-source (e.g. 32bit ODBC) whilst running in a 64bit environment, **irdbImport32.exe** should be used instead.

IRDBImport32 does exactly the same as IRDBImport but will always run in 32 bit mode

To reload the data file on the IrdbServer, use IRDBLoad with the /reload parameter.

IRDB Import - Reference Guide

The import file (.imp) contains a list of commands that imports tables into a database. Below is a description of the various aspects of this imp file...

Specifying a Source

First thing that should be done at the top of the import file (.imp) is to declare the Datasources you want to import-from by using the **DATASOURCE** command.

Note: *You can declare as many datasources as you need in order to create your IRDB file. Datasource Names can be anything you call them –not just the sourcenames given below.*

Sources can include ODBC, OLEDB, DOTNET, CSV or several other sources listed below.

e.g. The following can be used in a single .imp file importation.

```
DATASOURCE ODBC1 = ODBC 'odbc_connection_X'  
DATASOURCE ODBC2 = ODBC 'odbc_connection_Y'  
DATASOURCE SQLSRC1 = SQLSERVER 'connection_string'  
DATASOURCE DNetSrc = DOTNET PROVIDER 'System.Data.SqlClient'  
DATASOURCE localirdb = LOCAL IRDB 'c:\IRDB\DATA\north1.irdb'  
DATASOURCE Remoteirdb = REMOTE IRDB 'IRDB=CONTOSO;PWD=mypass'  
DATASOURCE CSVsrc1 = CSV 'Data Source=<source  
folder>;Delimiter=<delimiter>;Has Quotes=<True/False>;Skip  
ROWS=<number rows to skip>;Has Header=<True/False>;Comment  
Prefix=<comment prefix>;Trim Spaces=<True/False>;Ignore Empty  
Lines=<True/False>;encoding=<encoding>'
```

ODBC Datasource

Syntax:

```
DATASOURCE yourname=ODBC 'connection_string'
```

Connection_string is an ODBC Connection string e.g. dsn=ir_northwind for a system odbc datasource called ir_northind.

Refer to www.connectionstrings.com for a full reference on odbc connection strings

yourname is the name that you use as an alias for the datasource in IMPORT commands.

OLEDB Datasource

Syntax:

```
DATASOURCE yourname=OLEDB 'connection_string'
```

Connection_string is an OLE DB Connection string.

Refer to www.connectionstrings.com for a full reference on oledb connection strings

SQL Server Datasource

Syntax:

```
DATASOURCE yourname=SQLSERVER 'connection_string'
```

Connection_string in this case is the connection string for the Sql Server Dot Net Provider.

e.g. for Standard Authentication

```
DATASOURCE mySQLSource=SQLSERVER 'Data Source=localhost;  
Initial Catalog=myDBName;User Id=myUser; Password=myPwd;'
```

Note: *The Dot Net Provider is the fastest way of connecting to sql server.*

Local InmemoryDB Datasource

Syntax:

```
DATASOURCE yourname=LOCAL INMEMORYDB 'LOCAL_FILE_Name'
```

You can open other locally stored IRDB Data files and import tables or execute sql against them like any other datasource. This loads the full contents on the file into memory.

e.g.

```
DATASOURCE A1=LOCAL INMEMORYDB 'contoso.irdb'
```

This opens the local data contoso.irdb as datasource A1

You can CLOSE the datasource and release the memory used, with the CLOSE command.

LazyLoad Local InmemoryDB Datasource

Syntax:

```
DATASOURCE yourname=LAZYLOAD LOCAL INMEMORYDB  
'LOCAL_FILE_Name'
```

This will allow you to open a local inmemory db, without loading the full contents into memory. Columns are loaded on demand.

e.g.

```
DATASOURCE A1=LAZYLOAD LOCAL INMEMORYDB 'contoso.irdb'
```

Remote IRDB Datasources

Syntax:

```
DATASOURCE yourname=REMOTE INMEMORYDB 'IRDB_CONNECTION_STRING'
```

You can connect to remote or local IRDB servers

e.g.

```
DATASOURCE A1=REMOTE INMEMORYDB  
'irdb=northwind;pwd=MyPassword'
```

This connects to the local IRDB server, and creates a datasource called A1 pointing to the Northwind database on the local server

DOT NET Datasource

A Dot Net Data provider source can be set with the keywords DOTNET PROVIDER after the equals symbol. It takes two parameters.

Param 1 is the dot net name of the class

Param 2 is the connection string for the provider

e.g. DATASOURCE a1=DOTNET PROVIDER 'System.Data.SqlClient' 'Data Source=localhost; Initial Catalog=yourDB; User Id=your_username; Password=your_password;'

The Dot Net provider Class needs to be accessible from IRDBImport/IRDBImport32, so you may need to couple it with the LOAD ASSEMBLY command.

Example 2:

```
LOAD ASSEMBLY 'Npgsql.dll'
```

```
DATASOURCE a1 = DOTNET CONNECTION 'Npgsql.NpgsqlConnection' 'User  
ID=your_user_id;Password=your_password;Host=your_host;Port=5432;Database  
=your_database'
```

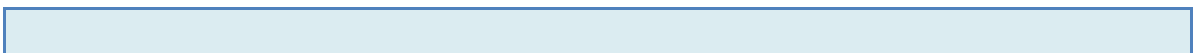
This example loads the Postgres Dot Net driver with the LOAD ASSEMBLY Command, and creates a connection to a Postgres server.

CSV Datasource

CSV files may be imported directly into IRDB without requiring an ODBC or similar connection.

SYNTAX:

```
Data Source=<source folder>; Delimiter=<delimiter>;Has Quotes=<True/False>;  
Skip Rows=<number of rows to skip>;Has Header=<True/False>;  
Comment Prefix=<comment prefix>;Trim Spaces=<True/False>;  
Ignore Empty Lines=<True/False>;encoding=<encoding>
```



CSV Datasource Example 1.0.

```
DATASOURCE a9=CSV 'Data Source = C:\irdb\data\csvtest;  
Delimiter =,; Has Quotes=True; Skip Rows=0;Has Header=True;  
Comment Prefix=#;Trim Spaces= False; Ignore Empty  
Lines=True; encoding=default'
```

By default, the system default text encoding is used. Other supported options for the encoding option are UTF8, ASCII, Unicode, BigEndianUnicode .

To specify a delimiter use the delimiter character e.g. , for a comma delimited csv file. For a tab delimiter use the keyword **tab** .

Note: By default the system will analyse the first 25,000 rows and use that to guess a data type for each column.

If you want explicit control over the datatypes that are imported, you need to create a table using the CREATE TABLE command, and then perform an Uncolumnized Import. Don't forget to Columnize the table when done!

CSV Datasource Example 2.0. *csv import script in full*

```
DATASOURCE a9=CSV 'Data Source=C:\irdb\data\csvtest;  
Delimiter=,; Has Quotes=True; Skip Rows=0; Has Header=True;  
Comment Prefix=#; Trim Spaces=False; Ignore Empty  
Lines=True; encoding=default'  
  
/* Simple CSV Imports, where it autodetects column type */  
import [orders] = a9.[orders.txt]  
import [customers]= a9.[customers.txt]  
import [order details]= a9.[order details.txt]  
import [categories]= a9.[categories.txt]  
// This format of the Import statement will override the  
// auto-detected datatypes in the next step, with the  
// previously defined types  
CREATE TABLE [categories2]  
  ( CategoryID text, CategoryName text, Description text,  
    Picture text )  
import [categories2]= uncolumnized a9.[categories.txt]  
  
// columnize the newly-created table  
COLUMNIZE categories2  
  
/* this example uses Decimal, int, long, date & Strings.
```

```

Other types are SMALLINT,TINYINT */
CREATE TABLE [orders2] ( OrderID DECIMAL ,
    CustomerID STRING, EmployeeID LONG, OrderDate DATE,
    RequiredDate DATE, ShippedDate DATE, ShipVia INT,
    Freight STRING, ShipName STRING, ShipAddress STRING,
    ShipCity STRING, ShipRegion STRING,
    ShipPostalCode STRING, ShipCountry STRING)

IMPORT [orders2] = uncolumized a9.[orders.txt]
COLUMNIZE orders2

SAVE

```

Internally we are using the Lumenworks CSV Parsing library by Sebastian Loriens and extended by Paul Hatcher. This has a MIT license. We have wrapped it with a custom DataReader.

ME Datasource

ME (case unimportant) is a special “built-in” data source that refers to the current in-memory database you are building. ME is not declared at the top of the import file since it always refers to “this current import process”.

ME is used with the IMPORT statement when you want to refer to a table or execute a sql command against a table that you have already imported in this same .imp file import.

mysample.imp 1.0 we will refer to these datasources in subsequent examples.

```

DATASOURCE Source1 = ODBC 'odbc_connection_X'

// Comments can be entered in an impfile like this
DATASOURCE Source2 = ODBC 'odbc_connection_Y'

/* Comments can also begin and end with these characters which
can be useful if they span more than one line */

DATASOURCE Source3 = SQLSERVER 'Provider=SQLNCLI11;
    Server=192.168.1.45;Database=ir_dw;Uid=usr1;Pwd=pwd;'

```

```
DATASOURCE yesterdays_irdb = LOCAL IRDB 'contoso.irdb'  
DATASOURCE A1 = ODBC 'odbc_connection_Z'
```

CLOSE Command

If you want to close a datasource, you can use the close command. It takes one parameter, which is the name of the datasource.

e.g. `CLOSE myDataSourceName`

For LOCAL INMEMORYDB datasources, this will release the memory, that has been used by loading the database. For others it will the database connection(s). We recommend closing LOCAL INMEMORYDB datasources, when they are no longer required in your script, to help release memory.

The datasource name can then also be used in new DATASOURCE commands.

```
Example program importing from two datasources, and using the CLOSE command  
DATASOURCE Source1 = local inmemorydb 'database1.irdb'  
import table1=source1.table1  
  
CLOSE source1  
  
DATASOURCE Source2 = local inmemorydb 'database2.irdb'  
import table2=source2.table2  
  
CLOSE source2
```

Localization Settings/ Culture

Setting Culture allows you to define a Dot Net environment for regionalization purposes (if different from your server's default).

e.g.

```
// The following will parse as DD/MM/YYYY  
set culture 'en-GB'  
import date1 = me.{select safecast ('09/01/2010' as datetime ) as col1}  
print date1.col1
```

```
// Will parse as MM/DD/YYYY  
set culture 'en-US'  
import date2 = me.{select safecast ('09/01/2010' as datetime ) as col1}  
print date2.col1
```

ImportTimeout

By default, the timeout duration for any query to run in IRDBImport is 15 minutes. You can alter this timeout value by use of ImportTimeout.

e.g.

```
// This sets the default timeout to 30 mins  
importtimeout 1800
```

ImportTimeout should be set in the .IMP file anywhere above an Import command

Import

Once a data-source is declared, data can be imported via the IMPORT command.

Syntax:

```
IMPORT table_name_in_IRDB=datasource_name.table_name
```

table_name_in_IRDB is the table the import will store the data as.

Datasource_name is the alias for one datasource.

table_name is the name of the table to read in the source database.

e.g.

```
IMPORT categories=a1.categories
```

This imports the categories table from datasource a1 and stores it as the categories table in IRDB

You can use **Square Brackets** [] around the destination table name to save it as is. You can also use CURLY {} brackets, after the . to read from an arbitrary sql statement,

e.g.

```
IMPORT [Order Details]=a1.{select * from [Order Details]}
```

You can also import data using the sql contained in another file by putting quotes around the filename. It is assumed the .sql file is in the same folder.

e.g.

```
IMPORT sometable=a1.'Name_Of_File_to_read.sql'
```

You can combine different queries from different sources together into one combined table by using UNION ALL

e.g.

```
IMPORT combined_table= A1.table1 UNION ALL a2.table2
```

Multiple UNION ALL operators can also be combined. The number of columns and field datatypes need to match across each table/ query in the Union.

Data Compatibility in UNION ALL

By default, IRDB insists on strict datatype matches between datasets in a Union All. For instance, numeric fieldA (*which is, say, a Double*) in Table1 linking with fieldA (*which is datatype Float*) in Table2 will cause an error as IRDB expects them to have the same datatype. This assumption improves performance, but may be switched off by use of the USETEMP keyword.

e.g.

```
IMPORT combined_table=USETEMP A1.table1 UNION ALL a2.table2
```

This creates a temporary table behind the scenes in IRDB for both tables before creating the final one. Thus more time and overheads are used.

Note: SQL syntax is dependent on the source database syntax rules.
For a list of IRDB-specific SQL Syntax see **IRDB SQL Reference**.

mysample.imp 1.1 *this is a continuation of mysample.imp 1.0 above.*

```
IMPORT decimal_test= source2.decimal_test
IMPORT Products= source2.products
IMPORT large_query = source1.
    {SELECT Orders.OrderID AS int_col,
        Orders.OrderDate AS date_col,
        [Order Details].Quantity AS short_col,
        [Order Details].UnitPrice AS dec_col,
        Orders.ShipName AS text_col,
        Products.Discontinued AS bit_col,
        CDb1([Order Details].[UnitPrice]) AS Dbl_Col,
        CSng([Order Details].[UnitPrice]) AS float_col,
        Categories.Picture AS bitarray_col
    FROM Categories
    INNER JOIN (Products INNER JOIN (Orders INNER JOIN
        [Order Details] ON Orders.OrderID = [Order
        Details].OrderID) ON Products.ProductID = [Order
        Details].ProductID) ON Categories.CategoryID =
        Products.CategoryID; }

import datatype_table= me.{
    select *, CAST_NUM_AS_LONG(int_col) as int64_col,
        CAST_NUM_AS_BYTE(int_col%64) as byte_col
    from large_query }

/* Say I have Customers in two data sources (duplicate tables/
```

```
different set of customers in each). I want to bring them together as one in IRDB... */
```

```
IMPORT Combined_Custs =USETEMP Source1.customers  
                        UNION ALL  
                        Source3.customers
```

```
// Alternatively, we could bring in both, then combine...
```

```
IMPORT Custs1 =Source1.customers  
IMPORT Custs2 =Source3.{SELECT * FROM customers}  
IMPORT Combined_Custs = ME.Custs1 UNION ALL ME.Custs2
```

IRDB Import - Optimisation

USEDISK

By default, IRDBimport.exe takes advantage of RAM that is available to it. However, you may not want to use so much memory, in which case it is possible to *buffer data to disk* while a data extraction is being run by including the keyword USEDISK.

eg.

```
IMPORT table=USEDISK a1.table1
```

This loads data in chunks of 1,000,000 rows, saving to a temp file. Data is then Columnized to a different file containing an InMemoryTable. The table is loaded in its entirety into memory only when finished.

USEDISK FLUSH

USEDISK FLUSH performs the same operation in the same way as USEDISK, except the end-result table is saved to a single (flushed) table that is then consolidated when SAVE is called at the end of the Import Script.

eg.

```
IMPORT table=USEDISK FLUSH a1.table1
```

Note: Tables are not available for Querying, Updates or Deletes with the FLUSH variant.

By using USEDISK FLUSH, it is possible to generate larger extract files using a lot less memory as it only needs to be able to contain one column of information at a time. Extraction times will take about 5-10% longer. Do not use USEDISK FLUSH if this table needs to be referenced in a later part of the import script.

Sufficient storage is required during the import phase for an extra copy of this table (*on the disk in which the IRDB file is being saved*).

The USEDISK functionality can be accessed programmatically using the DiskCombiner class in IRDB.

DO PARALLEL END PARALLEL

If you want to import a series of tables in parallel, to speed up your import process, irdblImport supports a PARALLEL statement. This is best used with multiple import statements. While other statements can be put in the parallel block, ideally each statement should be self-contained, and not depend on the sequence of execution and any other statement in the parallel block.

Syntax:

```
DO PARALLEL
    Statement 1
    .....
    Statement N
END PARALLEL
```

Example of DO PARALLEL Imports 3 tables in Parallel

```
DO PARALLEL
    IMPORT TABLE1 = A1.TABLE1
    IMPORT TABLE2 = A1.TABLE2
    IMPORT TABLE3 = A1.TABLE3
END PARALLEL
```

Slurp (IMPORT ALL FROM SOURCE)

All tables from an underlying datasource can be imported in one go using the SLURP Command. SLURP does not import views. Slurp functions against IRDB, Access & Sql server. Other source DBs are currently not supported by the slurp function.

Syntax:

SLURP <source_name>

This imports all data from all tables in the given datasource.

e.g.

SLURP A1

mysample.imp 1.2

SLURP A1

//other sources can also be added

IMPORT decimal_test= **source2**.decimal_test

Execute

You can use the EXECUTE Command to execute a sql statement against the current IRDB, much the same as with 'ME', storing the results back into the IRDB.

Syntax:

```
EXECUTE table_to_store_results_as = { some_SQL_command in curly bracks }
```

mysample.imp 1.3

```
EXECUTE Combined_Sales = {  
    SELECT 2.0 as currency_factor,  
    [order details].UnitPrice AS unit_price,  
    [Order Details].Quantity, [Order Details].costprice,  
    Orders.CustomerID, Orders.EmployeeID,Orders.ShipVia,  
    [Order Details].ProductID,  
    Products.CategoryID, Products.SupplierID,  
    Orders.OrderID, Orders.OrderDate  
FROM [Order Details]  
INNER JOIN Orders ON [Order Details].OrderID =  
Orders.OrderID  
INNER JOIN Products ON [Order Details].ProductID =  
Products.ProductID  
}
```

Drop

The DROP command is used to drop tables no longer required. This is useful for removing a table in order to reduce the size of the irdb file while importing.

One reason for dropping an imported table might be if the imported table was brought in for the purpose of using data to create another table(s).

Syntax:

```
DROP table_name
```

The table in question will not be saved in the IRDB file.

e.g.

```
DROP [order details]
```

RENAME

The RENAME command allows you give an existing table a new name

Syntax:

```
RENAME [OLD_TABLE_NAME] [NEW_TABLE_NAME]
```

e.g.

```
RENAME Orders Sales
```

REORDER

The REORDER command allows you reorder a table by a different sort order. This can be useful if you want to go through a table record by record.

Syntax:

```
REORDER TABLE [TABLE NAME] BY column1 (optional desc), ...,  
columnN (option desc)
```

e.g.

```
REORDER TABLE orders by ORDERID Desc
```

Try/ Catch

A TRY block can be used in an IRDBImport file in order to run a group of commands together. A CATCH block follows the TRY in order to catch any resultant errors.

e.g.

```
TRY  
    IMPORT mytable1 = a1.mytable1  
    IMPORT mytable2 = a1.mytable2  
END TRY  
  
CATCH  
    CLOSE CONNECTION  
END CATCH
```

In this example, the connection is closed if a problem arises within the TRY section.

Exit

You can have IRDBImport stop execution and exit with a specific code to the OS.

e.g.

```
EXIT {exit_code}
```

This exits IRDBIMPORT and sends an optional code back to the Operating System. By using a negative code, you code indicate to a calling batch file, a specific kind of failure occurred.

e.g.

```
EXIT -1
```

Add a Column (Alter Table)

To add a column to a table in IRDBImport, we use the Alter Table command.

Syntax:

```
ALTER TABLE table_name  
ADD Column column_name = {SQL Expression}
```

The value in the new column is expressed in terms of its relation to other value(s) within the same table –and/or constant(s) –and/or *@variable* substitution (see “Declare” below).

e.g.

```
ALTER TABLE Orders  
ADD Column DoubleID = {Orderid * 2}
```

(Orderid being an existing fieldname in the Orders table)

Add Constant Column Command

This will command will add the same constant value to every table in a datasource.

```
ADD CONSTANT COLUMN column_name=<Expression> TO datasource_name
```

e.g. `ADD CONSTANT COLUMN company_id=1+1 TO ME`

The command will only work with ME or local INMEMORYDB datasources.

Embedded SQL

Embedded SQL can be used anywhere in an IRDB import file for Selecting, Updating, Deleting, etc. purposes. Embedded SQL can also use multiple statements. It is also a good place to define things like views and functions.

To use SQL in an import script file, enclose the query within curly brackets.

e.g. *SELECT*

```
IMPORT [Order Details]=a1.{select * from [Order Details]}
```

e.g. *UPDATE*

```
IMPORT orders = a1.orders  
{update orders set employeeid = 5 where employeeid=6}
```

e.g. *DELETE*

```
IMPORT orders = a1.orders  
{DELETE FROM orders WHERE employeeid = 2}
```

Create Table/ Insert Into ... VALUES

Tables may be created using the Create Table command in an IRDBImport import script.

Tables can be created directly inside the script or inside a SQL block.

INSERT INTO populates the table. While INSERT INTO is both supported inside IRDBImport and an SQL Block, it will run faster in the IRDBImport Script.

The Columnize command is required following an insert.

Create Table and INSERT Example

```
CREATE TABLE Orders ( Orderid int, OrdDate DateTime,  
Customer string, OrdAmt double)  
  
INSERT INTO orders ( orderid,orddate,customer,ordamt )  
VALUES ( 1,'2010-01-01','A0001' , 1000.0 )  
INSERT INTO orders ( orderid,orddate,customer,ordamt )  
VALUES ( 2,'2010-01-02','B0001' , 2000.0 )
```

COLUMNIZE Command

If using the INSERT command to populate a table in IRDB, once all values have been added, the COLUMNIZE command is used in order to compact the table for use with InMemory.NET. Neither SQL Select statements, nor any other database function can be run on a Created table until the table has been Columnized.

Note: *Once a table has been Columnized no further Insertions are allowed, unless you run decolumnize on it*

UNCOLUMNIZED IMPORT MODE

Data imported using the IMPORT command (*as opposed to Create Table above*) is automatically “columnized” (*compactd for use with IRDB*).

If one or more table with the same name is subsequently imported in the same script, the existing data is automatically *uncolumnized* and the new data is appended to the existing IRDB table before again being automatically columnized.

This works well if there are only a few tables involved, however, if importing tens or hundreds of separate tables, this automatic columnizing and uncolumnizing will slow the import down.

UNCOLUMNIZED mode should be considered in this case for these tables as it keeps data in an uncolumnized state until the COLUMNIZE command is issued.

e.g.

```
IMPORT t1 = UNCOLUMNIZED a1.orders
IMPORT t1 = UNCOLUMNIZED a1.orders
COLUMNIZE t1
```

The above is an example irdbImport script that imports the orders table twice from datasource A1 as table t1. The final statement COLUMNIZE will then columnize the data.

DECOLUMNIZE Command

If you want to decolumnize one of your tables, you can use the DECOLUMNIZE command. This will take the table, that has been columnized, into the a compressed format, and expand it back out, so you can insert data into it.

e.g DECOLUMNIZE orders

SET PRESERVE DECIMAL

By default then decimal data is imported it is converted into Double format, to speed up calculations. Double types can execute up to 5 times faster, than Decimal. This can cause small rounding issues when dealing with large amount of records.

IF you want to disable this behaviour you can use the command

```
SET PRESERVE DECIMAL TRUE
```

To reenable Decimal to Double conversion you can then use

```
SET PRESERVE DECIMAL FALSE
```

FLUSH

The Flush Command writes data that is currently “in memory” to a temporary file in order to free-up memory to facilitate further Imports. Once the SAVE command is issued (at the bottom of the Import script), the temporary file contents are automatically stitched onto the rest of the database currently in-memory.

This is useful if a lot of data is being imported and/or RAM is limited.

```
IMPORT categories=a1.categories
IMPORT Customers=a1.customers
IMPORT Employees=a1.Employees
FLUSH
IMPORT [Order Details]=a1.{select * from [Order Details]}
IMPORT orders=a1.orders
IMPORT Products=a1.Products
IMPORT Shippers=a1.Shippers
IMPORT Suppliers=a1.Suppliers
```

```
SAVE
```

You can also FLUSH a single table to disk by passing FLUSH a table name.

e.g. `FLUSH orders`

Note: Data that has been Flushed may not be used/ referred-to in subsequent queries/ procedures within the import script.

Delete

Data rows may be deleted from a table using Delete, inside embedded sql brackets.

e.g.

```
IMPORT orders = a1.orders  
{DELETE FROM orders WHERE employeeid = 2}
```

Note: The Delete command does not work with data that has been created using the CREATE TABLE command if that table has not yet been Columnized.

Update

Field values may be changed in IRDBImport using the Update command.

e.g.

```
IMPORT orders = a1.orders  
{update orders set employeeid = 5 where employeeid=6}
```

Note: The Update command does not work with data that has been created using the CREATE TABLE command if that table has not yet been Columnized.

Clustered Index

For even more speed, it is possible to create a clustered index on one field in a table. The table will be sorted by that index, the index will be used on Range Queries or queries with multiple equalities or potentially on joins on that key, where a filter on a joined table, can generate a filter on the first. To use a clustered index in a query, it needs to be the first table, in the SELECT list.

Syntax:

```
CREATE CLUSTERED INDEX ON table_name ( field_name )
```

e.g. CREATE CLUSTERED INDEX ON [combined_sales] (OrderDate)

Save

The last command in an irdbimport .imp file should be SAVE without any parameters. This saves the results. This is required if you would like to save any data. It does not take any parameters

Syntax:

SAVE

By default the file will be saved using the stem of the IMP file with an .IRDB extension

You can override the name used, by using SAVE AS

e.g. SAVE AS 'mycustomfile.irdb'

IRDB Import - programmable elements

Declare

You can declare variables for use inside the `IrdbImport` script, which can be used in expressions and other commands.

Variables begin with `@` and then have an alphabetic character, followed by an alphanumeric character.

There are 7 supported types. `STRING`, `DATETIME`, `LONG`, `INT`, `DOUBLE`, `DECIMAL` & `UNIQUEIDENTIFIER` (Guids)

Syntax:

```
DECLARE @variablename as datatype
```

Note, a numeric character cannot be used immediately after the `@` symbol, but can be used in any position subsequent to that.

e.g.

```
DECLARE @cvar1 as STRING
```

```
DECLARE @nvar1 as INT
```

Parameterizing Imports

Any command line parameter is available to an import script as a special variable called `@param1` ,... `@paramN` of type `STRING`.

Command Line Parameters

Values can be passed to an import script from the command line when it is being called.

`@param1` is the value corresponding to the first word following the `IRDBIMPORT` command (generally the name of the import file), `@param2` is the second word (separated by a space).

mysample2.imp 1.0 ...From the command prompt

```
C:\IRDB\Data> ..\irdbimport.exe mysample2.imp "companyX"
```

For this import, within the mysample2.imp file,

```
@param1 = "mysample2.imp"  
@param2 = "companyX"
```

Note: When importing from an external source, a variable Declaration and Set command must be performed at the beginning of each import query where it is used.

See mysample2.imp 1.1 below.

Parameterizing Connection Strings and Tables to Import

The connection string in the Datasource command can accept regular expressions as input. Therefore an expression can be built using variables.

Pre-existing variables can also be used in the import command by putting an extra @ in front of the variable. So if the variable is called @tablename use @@tablename

mysample2.imp 1.01 ...Take a script called param_example.imp with the following contents:

```
datasource a1 = odbc 'dsn=' + @param2  
IMPORT [@@param3] = a1.[@@param3]  
IMPORT [table_@@param3] = a1.{select * from [@@param3] }  
save
```

When run with

```
..\irdbimport32 param_example.imp ir_northwind orders
```

will import the orders table from the 32 bit odbc datasource called "ir_northwind" and save it as orders and table_orders.

Set

You can use the SET command to set the value of a variable equal to an expression.

Syntax:

```
SET @variablename = expression
```

e.g.

```
set @var1 = 5+5
```

```
set @var1 = 5+(20*3)
```

```
set @var2 = @var1+50
```

```
// Set a variable from result of an SQL statement
```

```
declare @count as int
```

```
set @count = {select sum(1) from orders }
```

An important feature of variables is that they can be used inside the SQL of IMPORT statements. When the import statement uses an IRDB type datasource, a single Declare and Set can be made at the beginning of the .imp file, otherwise a variable to be used in a query from an external source must be declared and set within the IMPORT brackets as in the example below.

mysample2.imp 1.1

```
DATASOURCE Source1 = ODBC 'odbc_connection_X'
IMPORT orders = source1.orders

/* Say we want to create another order table with a
specific subset of items only... */
IMPORT orders_some_only = ME.
{
  DECLARE @cSuppl AS STRING
  SET @cSuppl = "Smiths Sweets Inc."

  SELECT *
  FROM ORDERS AS o1
  WHERE o1.Supplier = @cSuppl
        AND o1.Office = @param1
/* Notice we did not have to declare @param1 since
it is a system function, in this case created when we
called the 'mysample2.imp' import script */
}
```

Running dynamically generated scripts using ScriptExec

It is possible to dynamically generate complete programs and execute them using the scriptexec statement. The following example imports the Orders table from the ir_northwind odbc data source. The generated code, should be in the IRDBImport language syntax.

mysample2.imp 1.01

```
DATASOURCE a1 = odbc 'dsn=ir_northwind'  
DECLARE @tableName as string  
SET @tableName = 'Orders'  
ScriptExec 'import [' + @tableName + ']' = a1.[' + @tableName + ']'  
SAVE
```

Running external Programs with CMDExec

You can run an external program using the CMDExec Command. It takes two string parameters. The first is the full path to the external program to run. And the second are the command line Parameters for that external program.

CmdExec.imp Use CMDExec to run an external IMP Script

```
CMDEXEC 'c:\irdb\irdbimport' 'c:\irdb\data\externalscript.imp'
```

Running external SQL Scripts with SQLExec

You can run an external sql script SQLExec Command. It takes one string parameters. This is the location of the external SQL Script to run.

SqlExec.imp Use SQLExec to run an external SQL Script

```
SQLEXEC 'myquery.sql'
```

Accessing field values in a table

You can access a field in the current record, by doing using the tablename.fieldname syntax. Each table has a current record pointer. Tablename.eof will tell you if you are at the EOF or not.

e.g. `orders.orderid` will give you the current row value of Orderid in the Orders table

You can access a field in the nth record, by doing using the tablename.fieldname(n) syntax. e.g. `orders.orderid(10)` will give you the value of the 10th row value of Orderid in the Orders table

You can also change the values in existing tables using this notation and the SET command as well. The underlying table needs to be uncolumized for this to work

```
SET orders.orderid = 20000
SET orders.orderid(10) = 20000
```

MoveFirst & MoveNext

The MOVEFIRST command can be used to point the table record pointer at the first record in a table. E.g. `MOVEFIRST Orders`

The MOVENEXT command can be used to move the table record point to the next record in a table. E.g. `MOVENEXT Orders`

Iterating Through a Recordset - For & While Loops

In IRDBImport, a table may be iterated-through using a loop such as FOR or WHILE. e.g.

WHILE Loop

```
WHILE orders.eof = false
    print orders.orderid + ' ' + orders.employeeid
    movenext orders
LOOP
```

FOR Loop

```
declare @i as int
FOR @i =1 to 10
    ALTER TABLE Orders
    ADD Column NewOrderID = {Orderid * @i}
NEXT
```

Print

Output from an expression may be printed to the console using the Print command. This is useful for debugging purposes.

Syntax:

```
PRINT expression
```

e.g.

```
Print 'Hello World'
```

```
Print @var2
```

If

You can use the IF Command to do control flow in your IRDBIMPORT script.

AND, OR, NOT and parenthesis are supported in the logic.

ELSE is optional in the IF statement

Syntax:

```
IF expression1 = expression2 THEN
```

```
    statementList
```

```
{ELSE
```

```
    statementList}
```

```
END IF
```

e.g.

```
IF @var1=10 then
```

```
    PRINT 'HELLO WORLD'
```

```
END IF
```

ASSERT

The assert command takes one parameter, which is a logic expression similar to an IF

statement. If it evaluates false, it will halt program execution on that line. This is helpful for script validation, to stop execution if there is a logic error.

e.g. `ASSERT 2+2=4` will continue execution

but `ASSERT 2+2=3` will halt the program.

IRDB Import Functions

All functions are listed in this manual under “Functions”.

Picking up new IRDB data

If an IRDB database is loaded when an import of the same database is run and saved, queries from the existing database will not automatically reflect the newly-imported dataset (the data has been imported to the `irdb` file, but not loaded into memory).

To refresh this data, one possible method would be to stop and start the server itself

ie.

```
net stop "irdb server"  
net start "irdb server"
```

One issue here would be if you had more than one IRDB database running, each one would be stopped and restarted.

Also, after an IRDB Server start, databases are not automatically loaded into memory. This is done (automatically) following the first query –which is why you might notice a delay the first time a query is run on a newly-opened database.

So, a better way to refresh an IRDB dataset would be to explicitly load it using the `IRDBLoad` command:

IrdbLoad

The IRDBLoad command takes one parameter which is the irdb connection string of the database you want to load.

Syntax:

```
IRDBLoad
    database=<irdb_filename>
    [;host=<hostname>]
    [;username=<username>]
    [;pwd=<password>]
    [{/reload} {/unload}]
```

After the irdbLoad command is issued there is a pause of 30 seconds or so before a period (".") is displayed every few seconds while it polls the server to ensure it has finished loading.

/reload parameter forces a reload.

/unload unloads the named datafile.

e.g.

```
IRDBLOAD database=mysample;username=default;pwd=!TopSecret
/reload
```

Exporting from IRDB

Exporting from IRDB is performed as part of an IRDBIMPORT script. Tables need to be imported within the script prior to exporting.

CSV Export

Save a table to a CSV file in an IRDBIMPORT script (*either at import-stage –or from another script with your existing IRDB file as the Datasource*) in the following manner.

Syntax:

```
SAVE TABLE <tableName> TO CSV <filename>
```

Tablename and Filename should be enclosed in single quotes. Filename may include the path (otherwise saved to the same folder as the source) and should include the file extension, as required. If you leave out CSV, it will save in a raw InMemoryDB table format.

```
mysample3.imp 1.0
```

```
DATASOURCE Source1 = ODBC 'odbc_connection_X'  
IMPORT orders = source1.orders
```

```
SAVE TABLE 'orders' TO CSV 'myorders.csv'  
SAVE
```

```
/* Note, the second SAVE above saves the new inmemory.net  
database. Since the export has already been performed by  
this time, it is not necessary to include this save if  
you need to save the CSV file(s) only at this time. */  
}
```

EXPORT to SQL SERVER

Data can be exported from an InMemory Database format to SQL Server using the EXPORT command.

The base syntax, that appends to an existing table, or creates the table is
EXPORT {destination data source}.[Destination Table Name] from {source data source}.[Source Table Name]

```
EXPORT a1.orders FROM me.orders
```

You can add WITH TRUNCATE, to force it to truncate the existing table, if it is existing
e.g. EXPORT with TRUNCATE a1.orders FROM me.orders

To help prevent accidental deletion of data, to use WITH TRUNCATE, you need to run the command ALLOW TRUNCATE ON the datasource e.g.
allow truncate on a1

You can add WITH DROP, to force it to DROP the existing table, if it is existing
e.g. EXPORT WITH DROP a1.orders FROM me.orders

To help prevent accidental deletion of data, to use WITH DROP, you need to run the command ALLOW DROP ON the datasource e.g.
ALLOW DROP ON a1

Example Export Script

```
DATASOURCE db1=local irdb 'c:\irdb\data\northwind.irdb'  
  
DATASOURCE a1=SQLSERVER 'Data Source=localhost;Initial  
Catalog=ir_export_test;User Id=ir;Password=mypassword;'  
  
slurp db1  
  
allow drop on [a1]  
export with drop a1.orders from me.orders  
export with drop a1.[order details] from me.[order  
details]  
export with drop a1.[customer] from me.[customers]  
export with drop a1.[employees] from me.[employees]  
export with drop a1.[Products] from me.[products]  
export with drop a1.[shippers] from me.[shippers]  
export with drop a1.[suppliers] from me.[suppliers]
```

DEBUGGING With BREAKPOINT

You can add a BREAKPOINT command anywhere in your script, where you want to pause execution and starting debugging the irdblImport script.

irdblImport will ignore the BREAKPOINT command, but you can debug using the command line program

```
irdbTest IMPDEBUG myscript.imp
```

This will run your imp script in debug. Your IMP script will execute as normal until it hits a breakpoint. If it hits a breakpoint, it will show a debugging prompt labelled D>

The debugging prompt supports the following options

C+Enter	Continue Execution
S+Enter	Step one statement
V+Enter	Show the current values of all variables
QUIT+Enter	Exit the Debugging Session
IMP Command + Enter	Execute a one line statement in IMP Script.
! + SQL Command + Enter	Execute a one line SQL Statement. Outputs the result as a CSV

If you want execute a sql statement and not see CSV results, put it in curly brackets. You can also use the regular Print statement within the debug prompt.

IRDB Query

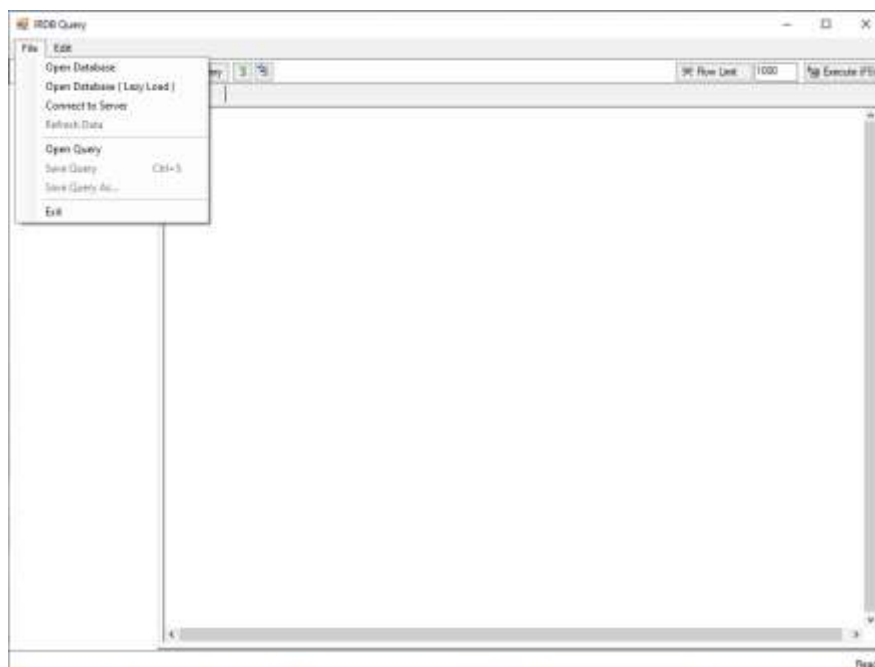
Starting IRDB Query

IRDB Query can execute IRDB SQL syntax against an IRDB datasource, displaying the results in a grid. It has a tree structure that allows you to view databases, tables and field names.

To run, execute IRDB_Query.exe in the IRDB folder (doubleclick, or via command prompt).

IRDB Query can connect to a local or remote IRDB datasource.

There are a few ways to open a database in IRDB Query...



Open Database

Choose a particular .IRDB file, thus opening that database alone.

Open Database (Lazy Load)

Open an .IRDB file, but the table contents are brought into memory only as needed/ selected. Thus, it may prove a quicker/ less consuming way to open a large database on a system with limited resources.

Connect to Server

View any database available to a particular IRDB Server.

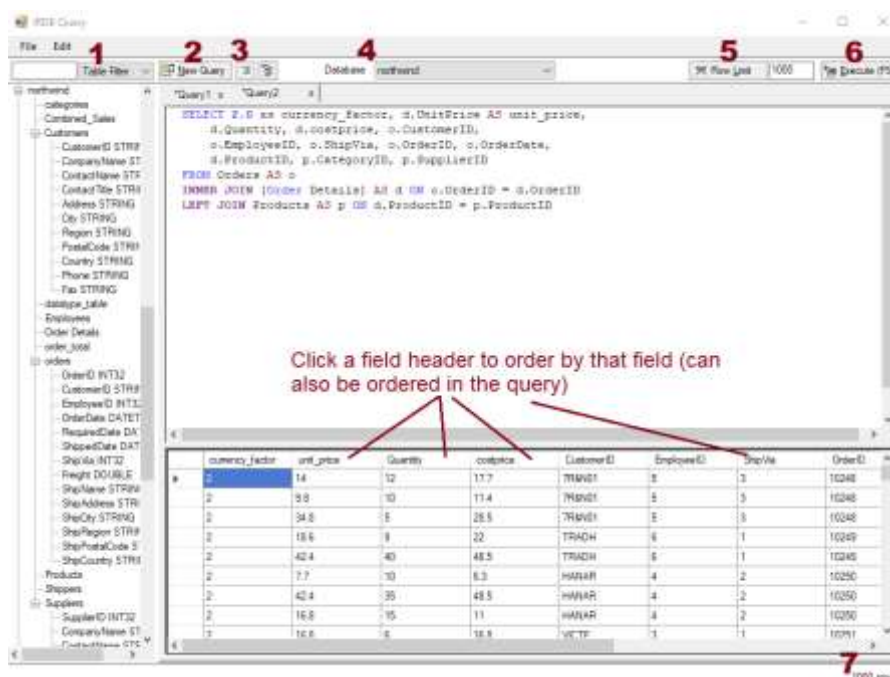
By selecting Connect to Server, you can enter the servername (including path/ ip address if necessary) and any other required security connection settings.



Note: By default, IRDB works from port 5060 (as recorded in IRDB.INI). Within IRDBQuery it is possible to use an alternative port by placing the following in the Server Name box.

HOST:Port_number

IRDB QUERY -How to Use



After connecting to a Database (see previous section), choose a Database from the left hand column and type your query in the main editbox. If you open a single Database, there is no need to choose a database here.

Alternatively, right click a table name to auto-create a query, listing all fields in that table.

Clicking on a Database name, where available in the left hand column, expands the table view for this Database. Similarly, clicking a Table name, expands to field view.

Individual fields can be added to the Query window by double-clicking the fieldname, if needed.

Searching for a Database/ Table/ Field

It is possible to filter on a name of a Database/ Table/ Field if you know only part of the name -or want to guess at it. (**1** in above image)

In the box next to Table Filter, entering eg. "ord" (without the quotes) will display only tables with this text in the table name. [*You may need to expand database name to view tables*]

Similarly, database or fieldnames can be filtered by choosing the correct filter-type next to the textbox.

New Query (CTRL-N)

If there is no other query on-screen, type a query in the large Query Window in the IRDB Query main screen.


To add new query tab, click New Query (**2**).


You can also Open a saved query by selecting "Open Query" from the File menu.

Another way to create a new query is to rightclick a tablename and choose to select fields for this table. If the existing query tab is being used, it will place this new query containing all fields for the selected table, in a new tab.

Add/ Remove a Comment (CTRL-plus/ CTRL-minus)

Comment-out a section, by placing `/*` at the beginning and `*/` at the end of the comment-area. This can be done “automatically” in the following ways:

Highlight text in the Query Window and click  (3) to comment-out this section.

Similarly, press  to remove a selected comment.

If no text is highlighted, clicking ‘Add Comment’/ CTRL-<plus key> will comment-out the current line.

If no text is highlighted, clicking ‘Remove Comment’/ CTRL-<minus key> will remove a comment from the current line.

Select a Database

The easiest way to select a database to run a query on, is to click the Database name in the lefthand margin, then add a new query. The most-recently clicked Database will be assumed to be the one required.

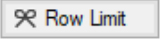
To change a selected database, when a user is Connecting to a Server, select the database from the dropdown at top/middle of the screen (4).

If you have Opened a database, rather than ‘Connected’ to the server, you will need to open the alternative database you wish to switch to. Your existing query/ queries will remain and will instead use the new Database, once run.

It is also possible to have more than one existence of IRDB Query in operation at once. You might find this useful for multi-database operations -although not essential, as the “Database” dropdown can be used across multiple databases when connecting to the IRDB server (as opposed to “opening” a database).

When connected to the server, each tab can reference a different database on the server.

Row Limit

By default, queries in IRDB Query will return 1000 rows of data only. This can be disabled by pressing  (5)

Alternatively, adjust the 1000 row limit in the textbox next to this button, to return a different number of records.

Warning: Disable the row limit at your peril... returning a very large number of rows can take a very long time. You might want to use COUNT() to return the number, rather than the actual rows.

Execute Query (F5)

To run a query, click  or press F5.

DBStats

If connected to an irdbServer, if you enter DBStats, it will give you some statistical information on the different databases that are on the server. It will show Database Name, Status (NOT_LOADING / LOADING / LOADED), Estimated Memory Usage, Time Loaded, Time Created, Disk Size when loaded, Latest File Created & Latest File Size

This can be useful to manage memory usage, and manage the load statuses of Databases to reduce memory consumption in the server.

IRDB SQL Reference

Whether running queries in IRDB Query, in an IRDB Import script (against an IRDB source) or anywhere else, IRDB supports the following SQL commands.

It is not in the remit of this document to go into too much detail on the following, since IRDB commands (where available) adopt common SQL syntax. Here we list the commonly-used SQL commands and conventions that are available in IRDB and describe its syntax in a general sense –or in more detail if different from the norm.

In general your SQL script can also contain multiple SQL commands. The result returned to IRDB Query or any client will be the last SELECT command.

SELECT

SELECT is the cornerstone of any SQL query. IRDB supports the regular SQL select syntax.

ie.

```
SELECT [NOCACHE] [TOP N ]
       select_list
[ INTO new_table ]
[ FROM table_source [AS alias1] ]
{[ INNER|LEFT JOIN|RIGHT JOIN|FULL JOIN|CROSS JOIN table2 (as
a2) on alias1.ColX = a2.ColY ]
} [ ,...n ]
[ WHERE search_condition ]
[ GROUP BY group_by_expression [WITH ROLLUP] ]
[ HAVING search_condition ]
[ ORDER BY {order_expression} [ ,...n ] ]
[ LIMIT n ]
```

The UNION ALL operator can be used between queries to combine or compare results into one result set.

SELECT Clause supports selecting from fields in the From Clause, doing expression using + / - * and () and other functions. It also supports SUM, MAX, MIN , AVG, COUNT,COUNT (DISTINCT some-field) aggregate functions. Aggregate functions can also be combined. CASE Statements with multiple WHEN Clause are also supported

Note1 : If you do use an Alias for a field or table, the keyword "AS" is required.

Note2: SELECT INTO works in IRDB Query, creating the named table in the in-memory database. However, since there is no way to SAVE from within IRDB Query, once you close the IRDB Query window the created table will be lost.

SELECT DISTINCT

SELECT DISTINCT is supported, but it is limited internally by the *group by* cardinality limits.

To make a distinct selection from a query that uses GROUP BY, you will need to add a new outer select.

e.g.

```
SELECT DISTINCT fieldX
FROM (      SELECT fielda, fieldb, fieldX
            FROM table1
            GROUP BY fieldb
        ) AS inner_query
```

SELECT *

*Select ** or *SELECT alias.** are also supported, but if you have columns with non supported characters, they will not be validated, and will cause the query to misbehave.

IRDB is designed for doing aggregative queries quickly. Non Aggregative queries will run and will also continue to execute in full if you select a large number of records from them. In our testing, tables with millions of records is not a problem, but if you pass a query that performs a *SELECT ** from a table with hundreds of million records, internally it goes through the process of creating the table from scratch and columnizing it, thereby possibly running out of memory.

SELECT Example1: All orders whose Freight is greater than 50

```
SELECT *
FROM Orders
```

```
WHERE FREIGHT > 50
```

SELECT Example2: Order totals for customerID = 'Frank' where any one item has a quantity of 40 or more, order by employee name. Show 100 rows at most.

```
SELECT d1.OrderID,  
       MAX(e1.FirstName + ' ' + e1.LastName) AS EmpName,  
       MAX(OrderDate) AS OrderDate,  
       SUM(d1.UnitPrice) AS 'Price',  
       SUM(d1.quantity) as QTY  
FROM Orders AS h1  
INNER JOIN [Order Details] as d1 ON d1.OrderID = h1.OrderID  
LEFT JOIN Employees AS e1 ON e1.EmployeeID = h1.EmployeeID  
WHERE CustomerID = 'FRANK'  
GROUP BY d1.OrderID  
HAVING QTY >= 40  
ORDER BY MAX(e1.FirstName + ' ' + e1.LastName)  
LIMIT 100
```

The screenshot shows a SQL query window with the following SQL code:

```
SELECT d1.OrderID, MAX(e1.FirstName + ' ' + e1.LastName) AS EmpName,  
       MAX(OrderDate) AS OrderDate,  
       SUM(d1.UnitPrice) AS 'Price',  
       SUM(d1.quantity) as QTY  
FROM Orders AS h1  
INNER JOIN [Order Details] as d1 ON d1.OrderID = h1.OrderID  
LEFT JOIN Employees AS e1 ON e1.EmployeeID = h1.EmployeeID  
WHERE CustomerID = 'FRANK'  
GROUP BY d1.OrderID  
HAVING QTY >= 40  
ORDER BY MAX(e1.FirstName + ' ' + e1.LastName)
```

The results table shows the following data:

OrderID	EmpName	OrderDate	Price	QTY
10623	Laure Calahan	07/02/2013	64.95	94
10488	Laure Calahan	27/09/2012	56	50
10267	Margaret Peacock	29/01/2012	73.1000000000	136
10670	Margaret Peacock	16/03/2013	57.75	192
10337	Margaret Peacock	24/04/2012	95.8999999999	137
10342	Margaret Peacock	30/04/2012	59.5999999999	160
10529	Michael Soyama	04/05/2013	30.75	124
10859	Nancy Davolio	29/07/2013	45.2	105
11012	Nancy Davolio	08/10/2013	64.7	146
10717	Nancy Davolio	24/04/2013	53.45	72
10396	Nancy Davolio	27/06/2012	52.2	121
10653	Nancy Davolio	02/03/2013	51.45	50
10675	Steven Buchanan	19/03/2013	69.3	70

Note: *Order By* requires the relevant field/ expression as an argument rather than a column number.

NOCACHE

As a performance-booster, IRDB creates temp tables when it feels sub-query-returns might be reused (e.g. for nested JOINS or where dynamic dates are used).

The creation of these temp-files can be switched off by the inclusion of the NOCACHE keyword.

The creation of these temp-files can be switched off by the inclusion of nocache=true in the irdb.ini file.

CACHE

Including the CACHE keyword in a select statement, overrides this statement by using cache, and forces the subquery to be explicitly cached in a temp table.

CASE STATEMENTS

```
CASE WHEN logic_expression THEN SomeExpression  
WHEN logic_expression2 THEN SomeExpression2  
ELSE SomeOtherExpression END
```

COUNT

Count(*field_name* | * | 1) counts the number of non null data rows in the selection.

e.g.

```
SELECT COUNT (customerid)  
FROM Customers  
WHERE city = 'London'
```

COUNT (DISTINCT)

Count Distinct returns the number of distinct values within a table.

e.g.

```
SELECT COUNT (DISTINCT CompanyName) AS NoCompanies  
FROM Customers
```

Expressions may also be used in a COUNT (DISTINCT).

e.g.

```
SELECT COUNT ( DISTINCT shipcity + shipregion)  
FROM Orders
```

IRDB also allows a number of COUNT(DISTINCT)s to be included within a single SELECT, each with its own WHERE clause.

e.g.

```
SELECT COUNT (DISTINCT CompanyName WHERE city = 'Madrid') AS MADRID,  
COUNT (DISTINCT CompanyName WHERE city = 'London') AS LONDON  
FROM Customers
```

Returns *(from the current Northwind database)*

MADRID	LONDON
3	6

You can also call this function with a list of database fields e.g COUNT (DISTINCT X,Y,Z), and it will count them.

COUNT DISTINCT in IRDB also supports a **WHERE** clause e.g. SELECT COUNT (DISTINCT x,y,z WHERE orderid = 10248) FROM Orders

SUM (DISTINCT)

When you give it one parameter, it will perform a unique sum on that column. If there are duplicate values they are only summed once.

With, multiple parameters (which have to be database fields), it will find the unique values,

and sum the first column.

OVER() / OVER (PARTITION BY)

OVER (PARTITION BY ORDER BY....) allows values to be partitioned based on the list of expressions following PARTITION BY, and ordered by a list of expressions in the ORDER BY Clause. It can be used with the RANK() and DENSE_RANK(),SUM,MIN,MAX,ROW_NUMBER() functions. ROWS UNBOUNDED PRECEDING at the end allows for CUMULATIVE Calculations.

e.g. *(the following can be run with the included Northwind database to view the results)*

```
SELECT EMPLOYEEID, CUSTOMERID,  
    /* number of customers per employee */ SUM(1) AS NoOfCustsPerEmployee,  
    /* Total number of Custs per employee -note, same total in each row within Customerid */  
    SUM ( SUM(1) )  
        OVER ( PARTITION BY EMPLOYEEID ORDER BY CUSTOMERID ) AS  
NoOfAllCustsPerEmployee,  
    /* Cumulative sum of Customerid within Employeeid */  
    SUM ( SUM(1) )  
        OVER ( PARTITION BY employeeid ORDER BY customerid ROWS UNBOUNDED PRECEDING )  
    AS CumulativeSumOfCustsPerEmp  
FROM ORDERS  
GROUP BY EMPLOYEEID,CUSTOMERID
```

JOINS

1-1 or 1-Many inner/left joins/right and full joins can be supported. Your query will execute best if you have the table with the highest cardinality first, joining against the tables with decreasing granularity. You can join as many levels deep from the first table that is driving it.

Currently IRDB supports **INNER** and **LEFT** and **CROSS , RIGHT JOINS** and **Full Joins**

All joins are driven by the first table which is assumed to be the main fact table. It supports joining on up-to 50 dimensions. Queries should be designed without join loops and joined in such a way that the joins all flow from the first table or higher level tables. Expressions involving fields from the same table are now supported. You can also put constants expressions on one side and fields from the joined table on the other.

The first clause in the join list should include explicitly include one field from the parent table and one from the child.

Left Join

e.g.

```
SELECT *  
FROM table_a AS a  
LEFT JOIN table_b AS b ON a.fieldID = b.fieldID  
INNER JOIN table_c AS c ON a.fieldID = c.AnotherID
```

CROSS JOIN

Cross joins can be created by having tables and joins separated by a comma, or by the words CROSS JOIN.

e.g. `SELECT *`
`FROM a,b,c`

You may be able to speed up cross join queries by moving related parts of the where clause into B,C as subqueries in this case.

e.g. `SELECT *`
`FROM a,b, (SELECT value1, value2 FROM c) AS cx`
`WHERE a.id = b.id AND b.value = cx.value1`

PIVOT

You can use a pivot operator as part of a join as well. The pivot operator allows you generate columns dynamically based on the list in the FOR subclause.

Example. This sums freight by customer & year, with a column for each employeeid in 1..9

```
select pivotTable.* from ( select customerid ,year(orderdate) as  
orderYear,freight,employeeid from orders ) as sourceTable  
pivot ( sum( freight) for employeeid in  
([1],[2],[3],[4],[5],[6],[7],[8] , [9]))  
as pivotTable
```

WHERE

WHERE clause supports = , > , < , >= , <= predicates and IN (value1,value2...,valueN) , & LIKE. it also supports AND, OR and NOT, using Parentheses. Expressions can be used and also the DAY, Month and Year functions. IRDB is also optimized to support expressions like DBFIELD in (SUBQUERY) . The EXISTS Clause is also supported.

Strict String equality executes and returns results quickly if = or IN is used, but will run less quickly if other comparison predicates are used. String comparisons are not case sensitive. You can also use a subquery with one parameter with IN.

Null values may be checked in the following manner: *expression IS NULL* or *expression = NULL*.

GROUP BY

GROUP BY supports grouping by multiple elements.

Alias.Field Style or with Expressions that only use one dimension, or Expressions with several dimensions from the same table.

e.g. *DAY (Alias.SomeDateField)*.

Group by will run faster if the cardinality of the group dimensions multiplied together are less than 500,000. This will use a Parallelizable algorithm using arrays. Cardinality greater than this will currently use a different parallel algorithm using a dictionary, which is slower

WITH ROLLUP & GROUPING Function

You can add WITH ROLLUP after the GROUP BY clause in an aggregate query, it will take the GROUP BY elements, one by one from the right and remove them, and run the query, with that group by clause, (nulling out the unused GROUP by columns), and append those results to the Result Set. The GROUPING Function is also supported in the SELECT statement. It takes a column as a parameter. It returns 0 if the column is in the GROUP BY, and 1 if it is not e.g.

```
select customerid, employeeid, sum(1) as NoOfOrders
, grouping(customerid), grouping(employeeid) from orders
group by customerid,Employeeid
with rollup
```

This will return the No Of Orders by Customer, Employee, then the no of orders by Customer, and then at the bottom, just the total no of orders. The last two columns will display 0,0 for the first part, 0,1 for the customer subtotals and 1,1 for the last row.

This clause is useful to add higher level subtotals to an aggregate query. Currently we don't support GROUPING SETS.

ORDER BY

ORDER BY can use any amount of fields/ columns

Table names, Column Names and Aliases are case insensitive and may contain an alphabetic character followed by alphanumerics -unless you put square brackets around the token, and

then any character can be used.

Names and aliases with a space also need to be contained within square brackets.

e.g.

```
SELECT OrderID, productid AS [1pID], supplID + ' ' + empID AS SuppEmp
FROM [ORDER DETAILS] AS details
ORDER BY productid, supplID + ' ' + empID
```

is all good, assuming the fieldnames are valid.

Note: *Order By* requires the relevant field/ expression as an argument rather than a column number.

HAVING

HAVING is supported by IRDB. The ALIAS from the select part of the query is used along with the keyword Having - or else an aggregative expression.

e.g.

```
SELECT item, sum(qty) AS quantity_sum FROM sometable HAVING quantity_sum>100
```

OR

```
SELECT item, sum(qty) AS quantity_sum FROM sometable HAVING sum(qty)>100
```

UNION ALL

Multiple tables with the same field structure may be combined using UNION ALL.

Note UNION will also work. It will perform a union all, and then a SELECT DISTINCT on the results.

e.g. SELECT * FROM table1 **UNION ALL** SELECT * FROM table2

INTERSECT

Returns the common rows between two SELECT statements.

INTERSECT Example

TABLE1 has one field with the following data a,b,c,d,e,f,g

TABLE2 contains c,d,f,x,z,

```
SELECT * FROM TABLE1 INTERSECT SELECT * FROM TABLE2
```

Results in the following

c,d,f

EXCEPT

Returns rows that *are* in an initial SELECT statement, but not in a secondary query.

EXCEPT Example

TABLE1 has one field with the following data a,b,c,d,e,f,g

TABLE2 contains c,d,f,x,z,

```
SELECT * FROM TABLE1 EXCEPT SELECT * FROM TABLE2
```

Results in the following

a,b,e,g

LIMIT

You can limit the number of results returned by IRDB, by adding a limit clause to the end of a query. It can take two forms:

```
SELECT * FROM sometable LIMIT 10 returns the first 10 records
```

```
SELECT * FROM sometable LIMIT 10,20 ignores the first 10 records, returning the next 20.
```

The alternative TOP N syntax will also work, e.g. Select top 100 * from sometable.

Correlated Subqueries

IRDB supports basic correlated subqueries. The correlated query needs to be a field in the SELECT Clause, and return at most one row per correlated query. It must join on one parent

table in the parent SELECT, using the same rules as the regular / inner join. Currently Correlated queries are not supported in the Where clause.

e.g

```
select orderid ,  
(select sum(1) from [order Details] od where od.orderid = orders.orderid )  
as noOfItems  
from orders
```

DELETE

Delete from sometable { additional joins }
{whereClause as above}

There is delete support at the moment. If you are querying an IRDBServer and issue Delete commands, it is deleted in the current in-memory version only. The changes are not preserved to disk.

UPDATE

Update in IRDB works similar to other databases like SQL Server. If you are querying an IRDBServer and issue Update commands, it is updated in the current in-memory version only. The changes are not preserved to disk. E.g.

```
UPDATE sometable SET columnA=100 WHERE columnA is null
```

or

```
update [order details]  
    inner join orders as orders  
        on [order details].orderid=orders.orderid  
set unitprice=unitprice*1.0
```

preserved to disk.

INSERT

INSERT in IRDB works similar to other databases like SQL Server. If you are querying an IRDBServer and issue INSERT commands, it is updated in the current in-memory version only. The changes are not preserved to disk.

Also table needs to be decolumnized, when INSERTing data into it. If the table is not decolumnized, it will be decolumnized, so the INSERT statement will execute. This could

cause concurrency issues if another process is running SELECT at the same time, as SELECT requires a table be columnized.

Two versions of INSERT are supported.

```
INSERT INTO mytable ( column1,column2 , ... columnN) VALUES ( value1,value2 ,... ,valueN)
```

e.g. `INSERT INTO orders (orderid) VALUES (12000)`

```
INSERT INTO mytable ( column1,column2 , ... columnN) {SELECT Statement}
```

e.g. `INSERT INTO orders (orderid ,customerid) SELECT orderid , customerid FROM orders`

CREATE TABLE – DROP TABLE

CREATE TABLE is also supported, and follows the standard SQL Syntax. You can also set default values for the columns as well. NOT NULL is not supported in the column definition.

```
CREATE TABLE tablename ( column1 sql_datatype {DEFAULT expression} ,...,  
columnN sql_datatype {DEFAULT (expression)} )
```

e.g.

```
CREATE TABLE orders01 ( orderid bigInt , customer varchar )
```

```
CREATE TABLE orders02 ( orderid bigInt , customer varchar default ( 22) )
```

Table orders02 in the examples will have a default value of 22 for the customer column.

DROP TABLE is also supported, and follows the standard SQL Syntax.

e.g. `DROP TABLE Orders02` will remove the table Orders02 in the above example

Temp Table Support (#tempTable)

IRDB SQL also supports temporary tables. They can be created with a SELECT .. INTO command, or a CREATE TABLE command. The name of the temporary table must start with a hash(#) character. e.g. #orders. The temporary tables are then automatically dropped when the current SQL script finishes. While IRDB-SQL doesn't support Table Variables, the temp tables, in IRDB-SQL have a similar behaviour. Inside a stored procedure, or table value function, their life time, will exist within that object. At the end of your script or SP, they will also be automatically dropped, after use.

Temp Table Example. This will display 830 when run against Northwind

```
CREATE TABLE #temp ( orderid bigInt , customerid varchar )  
insert into #temp ( orderid, customerid ) select orderid,customerid  
from orders  
select sum(1) from #temp
```

DECLARE and SET

IRDB SQL allows you to declare variables, and use them in batch SQL commands, to help program complicated sql scripts or reports. The syntax for Declare and SET is similar to IRDBImport. Variable labels need to start with a @.

```
DECLARE @today as DATETIME
set @today = '2018-01-01'

select @today
```

Currently supported datatypes for DECLARE are as follows.

DOUBLE,FLOAT - Double precision number

STRING,CHAR,VARCHAR,NVARCHAR,NCHAR,TEXT,NTEXT - String / Text Type

LONG,BIGINT – 64 bit integer

DATE,DATETIME – Datetime type

BIT – Boolean type

DECIMAL – Decimal Type

INT,INTEGER – 32 bit integer

UNIQUEIDENTIFIER – Guid Type

IF statement

IRDB SQL supports the standard SQL for if statements.

```
IF Boolean_condition
  statement
(ELSE
  statement )
```

e.g.

```
if 1=1
    select 1
or
if 1=1
    select 1
else
    select 0
```

If you need multiple statement within, then you can use BEGIN statements END in either the True or False part of the logic, e.g. will return 300

```
if 1=1
begin
    declare @var1 as bigint
    set @var1=100
    select @var1+200
end
```


WHILE statement

IRDB SQL supports the standard SQL for WHILE statements. While the Boolean condition is true, the statement(s) will be executed in the body of the while statement. **BREAK** can be used to exit the WHILE loop, or **CONTINUE**, to move execution back to top of the block.

```
WHILE Boolean_condition  
    statement
```

OR

```
WHILE Boolean_condition  
BEGIN  
    statement_block  
END
```

While Example. This will display 55

```
DECLARE @counter as bigint  
declare @total as bigint  
set @total =0  
set @counter =0  
while @counter<=10  
BEGIN  
    set @total=@total+@counter  
    set @counter=@counter+1  
END  
select @total
```

TRY CATCH statement

You can use a TRY CATCH block to catch any errors within a block of SQL code. The statement, takes two blocks of code. It will try and execute the statements within the TRY block. If a SQL error occurs, it will then execute the statement within the CATCH Block. It has the following syntax. A try catch block would be useful if you want to track errors within a script or stored procedure.

```
BEGIN TRY  
    statement_block  
END TRY  
BEGIN CATCH  
    statement_block  
END CATCH
```

Try Catch Example This will return An Error Occurred because of division by zero

```

BEGIN TRY
    select 1/0
    select 'Executed Ok'
END TRY
BEGIN CATCH
    select 'An Error Occurred'
END CATCH

```

With statement

IRDB SQL supports the standard WITH statement. WITH Statements allow you to break more complicated SQL logic into more easily handled chunks. They are consumed by the next SELECT statement. They act like temporarily views that are used by the next SQL Select statement.

E.g.

```

with MyCustomTable as ( select * from orders )
with MyCustomTable2 as ( select * from customers )
select sum(1) from MyCustomTable
inner join MyCustomTable2
on MyCustomTable.customerid = MyCustomTable2.customerid

```

You can also do recursive WITH statements. However you need to add the keyword RECURSIVE before the recursive SELECT. The following examples counts from 5 to 0.

E.g.

```

with RecursiveExample as (select 5 as level
                        union all
                        recursive select level-1 as level
                        from RecursiveExample
                        where level > 0 )
select * from RecursiveExample

```

Clustered Index

IRDB-SQL supports the same clustered index command as irdblImport.

Columnize Statement

IRDB-SQL supports the same Columnize statement as irdblImport.

Print Statement

IRDB-SQL supports a **print** statement, that takes an IRDB-SQL expression, and returns the output to the console. It is designed to help debug SQL scripts. If you are running within a gui environment like IRDB_Query you may not see the results.

```
print 2+2
```

CREATE VIEW – DROP VIEW

IRDB-SQL used CREATE VIEW for creating views, and DROP View for dropping them. Views act as virtual tables within subsequent SQL operations. ALTER View is not currently supported. Adding nocache after the first SELECT will cause the view to be evaluated on demand, otherwise it will follow the default caching behaviour, which is to materialize the subquery. An example follows.

```
create view MyView as
    select * from Orders union all select * from orders

select sum(1) from myView
```

Stored Procedures

IRDB-SQL supports stored procedures. Stored procedures are created with the **CREATE PROCEDURE** Command, and dropped with **DROP PROCEDURE**. ALTER Procedure is not currently supported. Stored Procedures can also accept Parameters. Stored Procedures are executed with the **EXEC** or **EXECUTE** command. Stored procedures can accept no parameters, or as many parameters as you want. They return the result of the last executed SQL statement. You can use #temptables within a stored procedure, where the scope is limited to the procedure. **RETURN** can be used to return from a SP.

Example 1 Declaring a SP with 1 parameter, and the two calling conventions with EXEC

```
CREATE PROCEDURE myProc @param1 integer as
    BEGIN
    SELECT * FROM ORDERS
        where orderid in (@param1 )
    END
```

```
EXEC myProc 10248
or EXEC myProc @param1=10248
```

Example 2 showing setting a default value to a parameter

```
CREATE PROCEDURE myProc3 @param1 integer = 10250 as
    BEGIN
    SELECT * FROM ORDERS
        where orderid in (@param1 )
    END
```

```
EXEC myProc3
```

EXEC Dynamic SQL Statement

You can execute dynamic sql with the EXEC / EXECUTE statement as well. It can take one parameter that evaluates to a string in parameters.

```
declare @tableName as varchar
set @tableName = 'Orders'
execute ( 'select * from [' + @tableName +']' )
```

You can also execute dynamic sql with the built in **sp_executesql** store procedure. This can take multiple parameters. The first parameter is the dynamic sql statement. The next N are variable definitions for the dynamic sql, and the next N are the values for variables. This is more secure than the former method, because you explicitly bind your variables. E.g.

```
exec sp_executesql 'select * from orders where orderid=@orderid' ,
'@orderid int' , 10248
```

SCALAR User Defined Functions

You can use CREATE Function to define a scalar user defined function. These functions can be used in SET and DECLARE expressions, and can take constants and variables as parameters within SELECT or other SQL statements. However when operating within other SQL statements, they cannot accept column values as input. They are evaluated at the start of the SQL Statement, and then have a constant value for the rest of the life time of the sql statement.

Example 1. This squares the parameter passed as input.

```
CREATE FUNCTION udf_squared (@nr int)
RETURNS long
as begin
    declare @result as long
    set @result = @nr*@nr

    return @result
end

select udf_squared(1000)
```

Functions are dropped with DROP FUNCTION. Alter Function is not currently supported.

Table Value Functions

These allow you to call parameterized functions within the FROM clause of a SELECT statement. The function returns a table in a specified format, and the results then work like a table, with the specified alias. Table value functions can take zero parameters or as many as you like. In the generic case, you can have 0-N input parameters. You give the output table a label, and a table definition. The following example shows how to declare a table value function, that takes one parameter, and outputs a table with two columns.

@outputTable works as virtual table where the output goes.

```
create function tvf_example ( @param1 integer )
returns @outputTable table ( col1 int ,col2 int )
as BEGIN
    insert into [@outputTable] select @param1,2
end

select * from tvf_example(10248) as xxx
```

Inline Table Value Functions

You can use a simplified version of a table value function with one sql statement. You also don't need to declare the output table format. The following example shows how to declare an inline table value function with one parameter, and how to call it from a SELECT statement.

```
CREATE FUNCTION getOrder ( @orderid int )
RETURNS TABLE
as RETURN (
    select * from orders where orderid in (@orderid )
)
select * from getOrder(10248) as OrderInfo
```

SPLIT_TABLE Table Value Function

SPLIT_TABLE is a special built in table value function. It takes 3 parameters. The first parameter is a SELECT statement. The second parameter is a column in the SELECT statement and the third parameter, is a character to parse for. The function is designed to take the column specified by the second parameter, and generate a new table splitting the data one per row, by the third parameter. This function is designed to help parse information within a column into separate rows.

Example. This produces a result set with 3 rows. A in row 1, B in Row 2, C in row 3.

```
select * from split_table ( (select 'a,b,c' as coll ) , coll , ',' )
as xyz
```

Table & Column Information

Information on the currently-connected IRDB database may be retried querying Information_Schema.tables

e.g. using the Northwind database

```
SELECT * FROM Information_Schema.tables
```

Result:

Table_Catalog	Table_Schema	Table_Name	Table_Type
irdb	dbo	categories	BASE TABLE
irdb	dbo	Combined_Sales	BASE TABLE
irdb	dbo	Customers	BASE TABLE
irdb	dbo	datatype_table	BASE TABLE
irdb	dbo	Employees	BASE TABLE
irdb	dbo	Order Details	BASE TABLE
irdb	dbo	orders	BASE TABLE

irdb	dbo	order_total	BASE TABLE
irdb	dbo	Products	BASE TABLE
irdb	dbo	Shippers	BASE TABLE
irdb	dbo	Suppliers	BASE TABLE

For more detailed information on tables, query Information_Schema.columns, e.g.

```
SELECT * FROM Information_Schema.columns
WHERE table_name = 'categories'
```

Results:

Table_Catalog	Table_Schema	Table_Name	Column_Name	Ordinal_Position	Data_Type
irdb	dbo	categories	CategoryID	1	Int32
irdb	dbo	categories	CategoryName	2	String
irdb	dbo	categories	Description	3	String
irdb	dbo	categories	Picture	4	Byte Array

Other Information_Schema Tables.

information_schema.updated Tells you when the current database was last loaded, updated and memory usage

information_schema.table_usage Tells you memory usage of each table

information_schema.column_usage Tells you memory usage of each column

information_schema.query_log Gives a list and stats on recently run queries. By default, the query_log is turned off. You can enable it by running a sql statement **SET querylog = 1000** , and it will record the last 1000 queries. You can disable it again by running **SET querylog = 0**

information_schema.query_log_delta Gives you the queries since the last time you queried query_log or querylog_delta. This is useful if you have a tool, that is querying the querylog incrementally.

Functions

Function Types

Below is a breakdown of the type of functions available in IRDB SQL.

String Functions

CAST | CAST_STR_AS_INT | CAST_STR_AS_DECIMAL | CHAR | CHARINDEX | COALESCE | CONCAT | CSTR | ENDSWITH | INSERT | ISNULL | ISNULLEEMPTY | LEFT | LEN | LCASE | LTRIM | REMOVE | REPLACE | REPLICATE | REVERSE | RIGHT | RTRIM | SUBSTRING | STARTSWITH | TRIM | UCASE

Date Functions

CDATE | DATEADD | DATEDIFF | DATEDIFFMILLISECOND | DATE() | DATEPARSE | DATEPART | DATESERIAL | DAY | DAYOFWEEK | GETDATE() | MONTH | MONTHNAME | TRUNC | YEAR

Math

ABS | CAST_NUM_AS_BYTE | CAST_NUM_AS_DECIMAL | CAST_NUM_AS_DOUBLE | CAST_NUM_AS_INT | CAST_NUM_AS_LONG | CAST_NUM_AS_SHORT | CAST_NUM_AS_SINGLE | CEILING | FLOOR | LOG | MAX | MAXLIST | MIN | MINLIST | POWER | RAND | ROUND | SAFECAST | SIGN | SQRT

LOGIC

AND | OR

Trigonometric

ASIN | ACOS | ATAN | ATAN2 | COS | COSH | SIN | SINH | TAN | TANH

Aggregate Functions

MIN | MAX | COUNT | AVG | SUM | COUNT (DISTINCT()) | SUM (DISTINCT()) | MINLIST | MAXLIST | GROUP_CONCAT

Statistical Functions

DENSE_RANK | RANK | STDEV | STDEVP | VAR | VARP | MODE | MEDIAN | PERCENTILE_DISC | PERCENTILE_CONT

Other Functions

COLUMN_EXISTS

Functions that are available in IRDBImport Expressions

These functions are available in IRDBImport expressions only. They are a subset of the IRDB SQL functions.

Functions with 1 Parameters

DAY | MONTH | YEAR | ABS | FLOOR | LEN | LCASE | UCASE | SIGN | LTRIM | RTRIM | TRIM | WEEKDAY | QUARTER | HOUR | MINUTE | SECOND | WEEK | DAYOFWEEK | DAYOFYEAR | EXP | SQRT | LOG | SIN | COS | TAN | SINH | COSH | TANH | ASIN | ACOS | ATAN | CHAR | ASCII | TRUNC | REVERSE | FILEEXISTS

Functions with 2 Parameters

LEFT | RIGHT | CONCAT | FORMAT | DATEPART | TRUNCATE | CONTAINS | SUBSTRING | CHARINDEX | REMOVE | LOG | POW | MAX | MIN | ROUND | STARTSWITH | ENDSWITH | DATEDIFFMILLISECOND | DATEDIFFTICK | ATAN2

Functions with 3 Parameters

DATESERIAL | DATEADD | DATEDIFF | REPLACE | SUBSTRING | CHARINDEX | INSERT | REMOVE

Note: FILEEXISTS is available within IRDBImport script only.

Function Syntax

ABS (SOME_NUMBER)

returns some_number if positive -or the positive part of some_number if negative

AND (SOME_BOOL_EXP1 , SOME_BOOL_EXP2)

returns true if SOME_BOOL_EXP1 and SOME_BOOL_EXP2 are both true. Otherwise false.

ASCII (SOME_CHAR)

returns the ASCII code for a given character – or the first character in case of a character array (string)

ASIN (NUMBER)

returns the principal value of the arc sine of *NUMBER*

ACOS (NUMBER)

returns the ACOS of *NUMBER*

ATAN (NUMBER)

returns the ATAN of *NUMBER*

ATAN2 (NUMBER1, NUMBER2)

returns the ATAN2 of *NUMBER1, NUMBER2*

CAST (SOME_EXPRESSION AS BIGINT

| BIT

| CHAR

| VARCHAR
| NVARCHAR
| NCHAR
| TEXT
| NTEXT
| DATE
| DATETIME
| DECIMAL
| DOUBLE
| FLOAT
| UNIQUEIDENTIFIER
| INTEGER
| VARCHAR
| REAL
| SMALLINT
| TINYINT)

converts some_expression from its original format into the chosen format.

See also: *SAFECAST()*

CAST_NUM_AS_BYTE (SOME_NUMBER)

returns a byte corresponding to some_number

CAST_NUM_AS_DECIMAL (SOME_NUMBER)

returns a decimal corresponding to some_number

CAST_NUM_AS_DOUBLE (SOME_NUMBER)

returns a double corresponding to some_number

CAST_NUM_AS_INT (SOME_NUMBER)

returns an int corresponding to some_number

CAST_STR_AS_INT (SOME_STRING)

returns an int corresponding to some_string

CAST_NUM_AS_LONG (SOME_NUMBER)
returns a long corresponding to some_number

CAST_NUM_AS_SHORT (SOME_NUMBER)
returns a short corresponding to some_number

CAST_NUM_AS_SINGLE (SOME_NUMBER)
returns a single corresponding to some_number

CAST_STR_AS_DECIMAL (SOME_NUMBER)
returns a decimal corresponding to some_string

CAST_STR_AS_DOUBLE (SOME_STRING)
returns a double corresponding to some_string

CAST_STR_AS_LONG (SOME_STRING)
returns a long corresponding to some_string

CDATE (SOME_STRING)
converts some_string as a date.

CEILING (SOME_NUM)
Returns the highest integer equal or after SOME_NUM. E.g ceiling(2.5) returns 3

CHAR (NUMBER)
returns a char from an ASCII code

CHARINDEX (SOME_STRING, INNER_STRING)
returns the numeric position of the first occurrence of *INNER_STRING* in *SOME_STRING*. If not found, return zero.

CHARINDEX (SOMESTRING, INNERSTRING, SEEKSTARTPOS)

returns the position (1-based) of the first occurrence of *innerString* (starting from seekStartPos position) in *someString*. If not found, return zero.

COALESCE (VAL1, VAL2... VALN)

returns the first non null value in the list otherwise valn.

COLUMNEXISTS (COLUMN_NAME,VALUE_IF_EXISTS,VALUE_IF_NO_EXISTS)

returns VALUE_IF_EXISTS, if COLUMN_NAME is present in the FROM Clause of the current query, otherwise return VALUE_IF_NO_EXISTS.

CONCAT (Exp1, Exp2, ... ExpN)

returns a string containing the string result of concatenating Exp1, Exp2, ... , ExpN. must be an expression or field that results in a string or number. You can have 2 or more parameters. Since Feb 2020, parameters that evaluate to NULL, are treated as an empty string. You can also use & to do string concatenation. E.g. Select 'A' & 'B'

COS (NUMBER)

returns the cosine of *NUMBER*

COSH (NUMBER)

returns the hyperbolic Cos of *NUMBER*

CSTR (SOME_EXPRESSION)

converts some_expression to a string.

DATE ()

Returns today's date without a time part.

DATEADD (INTERVALSTRING, SOME_INTEGER, SOME_DATE)

This adds some_integer to some_date corresponding to the interval intervalString. Valid intervals are d,dd,w,ww,m,mm,q,y,yy,yyyy,h,hh,n,s e.g dateadd ('m',1,'2010-01-01')

DATEDIFF (INTERVALSTRING, DATE1, DATE2)

returns an integer corresponding to the difference according to intervalString between date1 and date2. Valid intervals are d,dd,w,ww,m,mm,q,y,yy,yyyy,h,hh,n,s

DATEDIFFMILLISECOND (END_DATE, START_DATE)

Date Diff returning elapsed milliseconds

DATEDIFFTICK (END_DATE, START_DATE) – Date Diff returning elapsed ticks

DATEPARSE (STRING1, FORMAT_INTEGER)

returns a date corresponding to STRING1 parsed using the format specified using FORMAT_INTEGER. Currently a value of 101 is supported for FORMAT_INTEGER for MM/DD/YYYY formats. This is useful if you want to parse US Style dates, anywhere in the world.

DATEPART (DATEPART, SOME_DATE)

returns a value corresponding to the datepart requested from the given date. e.g. DATEPART (h, getdate()) returns the current hour of the current datetime. Valid dateparts are d,dd,m,mm,q,y,yy,yyyy,h,hh,n,s

DATESERIAL (YEARINT, MONTHINT, DAYINT)

returns a date corresponding to the 3 Year, Month and Day parameters.

DAY (SOME_DATE)

returns the Day in the month of the some_date parameter

DAYOFWEEK (SOME_DATE)

returns an integer of the day of week. 1..7

DENSE_RANK()

As with RANK(), DENSE_RANK() quantifies the ranking-position of a field-value within a given Select Partition. The difference is, if there are, for example, four values ranked as number 1, then the following DENSE_RANK()-assigned value will be 2 (ie. doesn't skip ranking values when there are a number of items with the same rank.)

See also: RANK(), OVER() / OVER (PARTITION BY)

As described, Dense_Rank does not skip an order number value if 2 or more (e.g.) employees have the same ranking. i.e. Emp abc and xyz are both ranked 1... next highest-ranked employee will be 2

```
SELECT employeeid, customerid, Orderid,  
       DENSE_RANK() OVER (PARTITION BY employeeid ORDER BY customerid ) AS OrderRank  
FROM orders  
ORDER BY employeeid, customerid, Orderid
```

ENDSWITH (STRING, SUBSTRING)

returns True if *STRING* ends with *SUBSTRING*.

FILEEXISTS(STRING)

**** available in IRDBIMPORT script only. ****

returns True if the file described in the string exists.

FLOOR (SOME_NUMBER)

returns the integer part of some_number. Rounds downwards.

FORMAT (SOME_DATE|SOME_NUMBER, FORMAT_STRING)

This returns a string from a date or number using the format_string syntax. In the case of a date, the format_string should be structured to indicate the year, month, day, etc. with their initial.

e.g.

```
SELECT FORMAT (GETDATE(), 'dd-yyyy-mm') ..might return something like 29-2015-12
```

```
SELECT FORMAT (5459.4, '##,##0.00') ...returns    5,459,40
```

GETDATE ()

returns the current date and time.

GROUP_CONCAT (expressionToConcatenate {SEPARATOR expression})

Aggregate function that concatenates expressionToConcatenate. Each expression will be separated by expression. The aggregate function will sort expressionToConcatenate in ascending order. The separator part is optional

E.g. This produces a comma separated list of order id for each employeeid

```
select employeeid, group_concat( orderid separator ',' ) from orders
group by employeeid
```

IIF (SOMECONDITION, VALUEIFTRUE, VALUEIFFALSE)

returns valueIfTrue if someCondition is true otherwise valueIfFalse

INSERT (BASE_STRING, START_POS, STR_TO_INSERT)

insert STR_TO_INSERT into BASE_STRING at START_POS

ISNULL (SOME_VALUE, ALTERNATE_VALUE)

returns ALTERNATE_VALUE if SOME_VALUE is null. Otherwise returns SOME_VALUE.

ISNULLOREMPTY(AFIELD)

returns True (1) if a field is null or if its (string) length is zero. Otherwise returns False (0). Currently only works with Strings.

LEFT (SOME_STRING, SOME_NUMBER)

returns the left most some_number chars of some_string

LEN (SOME_STRING)

returns the length of some_string as an integer.

LCASE (SOME_STRING)

returns some_string in lower_case.

LOG (NUMBER)

returns the natural logarithm of a NUMBER

LOG (NUMBER, BASE)

returns the logarithm of a *NUMBER* for the given *BASE*.

LTRIM (SOME_STRING)

removes leading whitespace from some_string

Max (NUMBER1, NUMBER2)

returns the greater of *NUMBER1*, *NUMBER2*.

MAXLIST (VAL1, VAL2... VALN)

returns the max value of val1,..valn. It supports an arbitrary number of parameters.

Min (NUMBER1, NUMBER2)

returns the lower of *NUMBER1*, *NUMBER2*.

MINLIST (PARAM1,PARAM2,PARAM ,.. PARAMN)

returns the minimum value from list of params PARAM1 through PARAMN

MONTH (SOME_DATE)

returns the month no of the some_date parameter

MONTHNAME (SOME_DATE)

returns the full month name of the some_date parameter

OR (SOME_BOOL_EXP1 , SOME_BOOL_EXP2)

returns true if either SOME_BOOL_EXP1 or SOME_BOOL_EXP2 are true. Otherwise false.

POW (NUMBER, POWER)

returns *NUMBER* raised to *POWER*.

RAND()

returns a random double that will vary from 0 to 1.

RANK()

Returns the ranked-position of a field-value in terms of its quantitative presence within a secondary partition. If, for example, there are two values ranked number 1, then the next field-value with the most numerous returns will have a ranking of 3.

See also: DENSE_RANK(), OVER() / OVER (PARTITION BY)

The following examples for use with the Northwind database...

/* Orders ranked (by Orderid) within Customer */

```
SELECT customerid, Orderid,  
       RANK() OVER (PARTITION BY customerid ORDER BY Orderid ) AS OrderRank  
FROM orders  
ORDER BY customerid, Orderid
```

/* Orders ranked (by Orderid) within Customer, per employee

e.g. Order abc and xyz are both ranked 1... next order will be 3*/

```
SELECT employeeid, customerid, Orderid,  
       RANK() OVER (PARTITION BY employeeid ORDER BY customerid ) AS OrderRank  
FROM orders  
ORDER BY employeeid, customerid, Orderid
```

REMOVE (SOME_STRING, START_POS)

removes all characters in *SOME_STRING*, beginning with position *START_POS*.

REMOVE (SOME_STRING, START_POS, LENGTH)

removes characters in *SOME_STRING* from *START_POS* delimited by *LENGTH*.

REPLACE (STRING1, STRING2, STRING3)

returns a string which replaces all occurrences of *STRING2* in *STRING1* with *STRING3*

REPLICATE (STRING1, INTEGER1)

returns a string which repeats *STRING1*, *INTEGER1* number of times

REVERSE (SOME_STRING)

reverse character positions of the given string –eg. ‘abcd’ becomes ‘dcba’

RIGHT (SOME_STRING, SOME_NUMBER)

returns the rightmost *SOME_NUMBER* chars of *SOME_STRING*

ROUND (NUMBER, DECIMAL_PLACES)

rounds a given *NUMBER* to the *DECIMAL_PLACES* limit.

RTRIM (SOME_STRING)

removes trailing whitespace from *some_string*

SAFECAST()

Works the same as CAST(), except SAFECAST() does not produce an error if the it proves impossible to convert a value to a chosen format. If an error occurs a NULL value will be returned.

e.g. The following returns null (integer)

```
SELECT SAFECAST('1a' AS INT) AS XYZ
```

See also: CAST()

SIGN (SOME_NUMBER)

returns the sign of the number. 1 if > 0. 0 if = 0 , -1 if < 0

SQRT (NUMBER)

returns the square root for the given *NUMBER*.

SIN (NUMBER)

returns the Sine of *NUMBER*

SINH (NUMBER)

returns the Hyperbolic Sine of *NUMBER*

SUBSTRING (SOME_STRING, START_POS)

returns the string value of *SOME_STRING*, beginning in position *START_POS*

STARTSWITH (STRING, SUBSTRING)

returns True if a *STRING* starts with *SUBSTRING*.

STDEV | STDDEV (NUMERIC FIELDNAME)

returns the standard deviation for all values in the selected field

STDEV_P | STDDEV_P (NUMERIC FIELDNAME)

returns the population standard deviation for all values in the selected field

TAN (NUMBER)

returns the TAN of *NUMBER*

TANH (NUMBER)

returns the hyperbolic TAN of *NUMBER*

TRIM (SOME_STRING)

removes leading and trailing whitespace from *some_string*

TRUNC (SOME_DATE)

removes the time part from a Date

UCASE (SOME_STRING)

returns *some_string* in upper_case

VAR | VARIANCE (NUMERIC FIELDNAME)

returns the variance for all values in the selected field

VARP|VARIANCE_P (NUMERIC FIELDNAME)

returns the population variance for all values in the selected field

YEAR (SOME_DATE)

returns the year of the some_date parameter

Special Functions

There are a series of special functions in IRDB. They can produce special type of results, and are designed to make writing complex SQL easier.

Timeshift

SUM (TIMESHIFT (aggExp , columnToTransform, transformationToMake))

This function operates within a SUM expression, and allows you to make a SUM where the date dimension specified by columnToTransform is transformed to transformationToMake, and the resulting SUM made. This is useful in BI tools, where you want to produce comparative results, e.g. TY vs LY

Example: this produces a some of This Year vs Last Year for 2015 on Orders

```
select  year( orderdate) ,
        sum(1) as total_TY ,
        sum( timeshift ( 1, orderdate, dateadd ( y , 1 , orderdate) ) ) as
Total_LY
from orders
where orderdate>='2015-01-01' and orderdate<='2015-12-31'
group by year(orderdate)
```

OPENBAL / CLOSEBAL

SUM (OPENBAL (aggExp , timeColumn))

SUM (CLOSEBAL (aggExp , timeColumn))

These functions work in conjunction with SUM. They are designed to help calculate opening and closing balances resp. The query will take the timeColumn, and use that to analyze the where clause (if any) to get a cutoff date. The expression aggExp for values before that date will be aggregated to give the opening balance. The closing balance adds the value within the time period as well, to give the balance at the end of the period.

If timeColumn or an expression involving it, or a table joined by it, is involved in the grouping clause, then OPENBAL and CLOSEBAL, will perform a cumulative sum along the time dimension as well.

Example 1: This will sum up orders for 2015 in number_during_period. openingBalance will

contain the number of orders before 2015, and closingBalance the number of orders over all time up to the end of 2015.

```
select employeeid , sum(1) as number_during_period,
       sum( openbal(1, orderdate)) as openingBalance,
       sum ( closebal(1,orderdate)) as closingBalance from orders
where orderdate between '2015-01-01' and '2015-12-31'
group by employeeid
```

SUM (OPENBALWITHZEROES (aggExp , timeColumn))

SUM (CLOSEBALWITHZEROES (aggExp , timeColumn))

OPENBALWITHZEROES and CLOSEBALWITHZEROES are two other versions. These functions operate as the previous two functions, but these versions, will add additional rows to the rowset, if there are missing entries derived from timeColumn in the resultset.

These functions are designed to make reporting on general ledger data easier.

LastChild / LastPrice / LastPriceWithZeroes

SUM (LastChild (aggExp, childDim))

This takes the parent SQL, and add the childDim to the grouping Clause, and returns the last entry ordered by childDim, and does an aggregation on aggExp. The resultset will not be expanded by this grouping level. It will only show the last entry, from this additional query. It doesn't add additional rows to the resultset.

```
select customerid,      year(orderdate) as theyear,
       month(orderdate) as theMonth,
       sum(lastchild(freight, orderdate )) as [lastchild]
from orders
group by customerid, year (orderdate) , month (orderdate)
order by customerid, year (orderdate) , month (orderdate)
```

SUM (LastPrice (aggExp, childDim))

This takes the parent SQL, and adds the childDim to the grouping Clause, and returns the last entry ordered by childDim, and does an aggregation on aggExp. It can handle cases where the grouping clause involves expressions of ChildDim, or child tables expressions derived from it. ChildDim needs to be a database field. It will populate an entry in the resultset, once it comes across an entry in the child data, and will add subsequent rows derived from ChildDim, propagating the LastPrice entry, even if there is no underlying data within the group. Only values of ChildDim that satisfy the where clause are considered. It uses the distinct values of childDim, that satisfy the where clause, for the pivot basis. If an entry of childDim doesn't appear at least once in the resultset, it will not add additional entries for that value.

```
select customerid,      year(orderdate) as theyear,
       month(orderdate) as theMonth,
       sum(lastprice(freight, orderdate )) as [LastPrice]
```

```
from orders
group by customerid, year (orderdate) , month (orderdate)
order by customerid, year (orderdate) , month (orderdate)
```

SUM (LastPriceWithZeroes (aggExp, childDim))

This is similar to the above functions, but it will add additional entries to the output, so that all grouping rows, will be pivoted against all values of childDim, and its joined dimension table, that a where clause with the childDim part only. It will also examine data, from before the minimal value of childDim, that satisfies the where clause. In the example below it would also add entries for months 1 to 11 to the output, because there is prior data in 2014, and there are orders in 2015 for months 1 to 11 for other customers. This adds entry for childDim for values that are in the underlying data, but don't satisfy the where clause.

```
select customerid,          year (orderdate) as theyear,
month (orderdate) as theMonth,
          sum (lastpricewithzeroes (freight, orderdate )) as
[LastPriceWithZeroes]
from orders
where customerid='DRACD' and year (orderdate) = 2015
group by customerid, year (orderdate) , month (orderdate)
order by customerid, year (orderdate) , month (orderdate)
```

These functions are ideal for working with inventory snapshot data, where you have a record that stores inventory say at specific internals (childDim). This will then help you show the last entry. Another example if you want to see the last price something was sold

Note on NULL VALUES:

In irldb, null values are in general supported.

In IRDBImport only the string related functions will be able to handle null string parameters in general.

IRDB Further Reading

IRDB with Interactive Reporting

The latest build of **Interactive Reporting** supports IRDB server. Interactive Reporting treats IRDB databases as any other database. The 32 bit COM Client is used to communicate with the IRDB Server. A mapping connection can be made from IR 3.3 or higher by selecting the IRDB as the database and using the following connection string

irdb=mydatabase;pwd=mypassword

for a local server running on the standard irdb port. The password must match the password in the irdb.ini config file

irdb=mydatabase;host=myServer;port=5060;user=myUser;pwd=mypassword

is the longer syntax for connecting to a remote server named myServer in this example.

Programming Overview

IRDB can be run by adding the IRDB library as a reference to your own Dot Net Projects directly in Visual Studio 2010 or higher. IRDB requires Dot Net 4.0, but will also work higher versions of Dot Net. It can be run in process or by connecting using the IPClient to the IRDB Server. IRDB also offers a COM Client that can be used to communicate with the IRDB library or the IRDB Server. **Please see the Programming IRDB Reference for more information.**

Performance Characteristics

We tested IRDB against a wide variety of computers and datasizes. Optimisation enhancements are introduced to IRDB regularly in IRDB updates, so it's difficult to quote facts and figures for each release, but as an example, the following were valid observations as of April 2015.

On a i5 type machine with 2 cores (4 hyperthreaded) with 8GB of ram, we ran tests on fact tables of about a million records. When running a particular SQL syntax for the first time there is an overhead of about 15ms. After that IR Type queries with 5 metrics were then executing in about 15-20 ms. Loading and saving time was under 1 second for a file that took up 27 Megs on disk and about 40 megs in memory. With about 10 million records execution varied from about 400ms to 700 ms depending on complexity.

On a i7 type machine with 4 cores (8 hyperthreaded) with 32 GB of ram, we ran test on a fact table of about 400 million rows . It took up about 11 Gigs on Disk and used about 14 Gigs in Ram. When aggregating across all data, queries took under 1s for 1 metric, and added about 0.5s per additional metric. Beyond a certain point, there was a doubling in execution time after the fourth or fifth metric, presumably because the cpu cache was saturated. However we had response times of 3-4s with about 5 metrics and about 10s with about 10-11 metrics. Aggregating across all data. When filtered to a smaller part of the range, queries ran much faster, a lot of the time in about 0.5s because it only needed to filter the data. These queries also involved inner joins. It took about 2 mins to Load the database from Disk. Saving took about 3 minutes. The loading and saving look like they could be improved, because the process was CPU bound on 1 cpu thread. Importing the data took about 60 mins, but the import process was saturating the Network connection of 100Mbs, achieving about 70k rows per second

IRDB currently has the following Limitations:

When using IRDB, with DotNet 4.0 there is a 2GB limit on arrays and objects. This means you could expect up to 500 million possibilities if your database column has > 65,536 unique values, or 1 billion if your column has <65536 unique values. Also the amount of unique values within a column will be limited to 2GB.

When using Dot Net 4.5, you can have arrays and objects with more than 2GB. However you need to explicitly enable GCLargeObjects in the App.Config. The limit then would be having 2.17 billion rows of data in your fact table.

IRDB works with the open source cross-platform software, Mono, both on Windows and Non-Windows platforms.

We also provide versions for IRDB for Dot Net Core 2.1 and Dot Net Core 3.1 which work on Windows and Non-Windows platforms.