This is an attempt to equip the developers who want to use Access as a front-end client to any RDBMS (Relational DataBase Management System) backend (ie, SQL Server, DB/2, Oracle, MySQL, PostgreSQL) with the right questions to explore and ask during design and development. As we cannot cover every special case or all the nuisances that each developers must deal with, it is hoped that,by reading over this document, you will be better equipped to find the needed answers for your specific case. The article assumes that you are familiar with developing an Access application, understand the fundamentals of data types used in Jet and VBA and know basic SQL and is familiar with Data Access Objects (DAO) library and/or ActiveX Data Objects (ADO) library, but are otherwise new to working with ODBC data sources. The objective is to provide you with a set of questions about design and development and know how to ask them.

The first thing that you should be familiar with is potential issues that can crop up when developing against a RDBMS. This is, in no way, a complete list, but hopefully broad and high-level enough to give you an idea of where to start in investigating or anticipating any problems they that you may encounter in development.

A note about terminology: In Access 2007 and future, the engine that runs Access has been renamed from Jet to ACE, though ACE is fully backward compatible with Jet used in version 2003 and earlier. In the rest of document, it is safe to substitute 'Jet' with 'ACE'.

An overview of ODBC

Open DataBase Connectivity (ODBC) is essentially a set of specifications that dictate how to form and communicate a request between two different data sources without requiring special knowledge of either product's native calls. In order to provide those functionality, all function calls into ODBC driver have to conform to the expected conventions and use certain data types. It could be thought of as a translator between different database vendors. However, when we think about how Access can import and export into several formats including Excel spreadsheet, comma-separated values (CSV) files, other ISAM engines such as dBase or Paradox, even though this is not Jet’s native language, but it is able to do because it has the library to translate such formats into Jet’s native language. With ODBC, however, the principle is slightly different. Instead of translating from one specific end to another specific end, we are trying to provide a translation that any sources and destinations can use. An analogy could be thought of having two people who do not speak same language, suppose one spoke French and other spoke Spanish. They both do know some English and thus agree to use English to communicate and this can succeed as long all three languages can convey the concepts accurately. Of course no language is exactly like other and may express same thing differently that there simply is no good description for the exact same concept in other language, and even if French and Spanish shared the same concept, if English didn’t have that expression, then it is lost in translation.

A request between the ODBC client (which in our case would be our Access front-end application) and the ODBC-compliant RDBMS can take this general form:
Jet - ODBC manager - Vendor's ODBC Driver - Vendor's database

Regardless how vendors have implemented their ODBC drivers, vendors will have made some assumptions to satisfy requirements for ODBC compliance. Understanding the assumptions used will help us accurately diagnose what otherwise be a head-scratching deviant behavior. All in all, we are most likely performing translation twice as French man and Spanish woman were doing via English. Take data type for example. Because we're not merely looking at case of translating one data type into other but rather translating one data type to a ODBC data type to final data type. To help illustrate, we’ll consider the two possible scenarios relating to the data types disparity.

Scenario A

Oracle’s `CHAR` is mapped to ODBC’s `SQL_CHAR` datatype, which can either map to Jet’s `Text` (if less than 255 character in length) or to Jet's `Memo` (if more than 255 character in length). It is quite rare to use `CHAR` in Jet as it is not available via Access UI so passing it unhandled would give us a strings that are usually padded with spaces to fill the remaining space, making it a fixed length string. Furthermore, the users will be surprised and annoyed when they notice that a bound textbox contains padded spaces where they must delete the padded spaces to add more letters.

Scenario B

Another example is MySQL's `MEDIUMINT`, which is mapped to ODBCs `SQL_INTEGER` which ends up in Jet as a `Long Integer`. MySQL's three-bytes `MEDIUMINT`'s range is not same as Jet's `Long Integer`/ODBC's `SQL_INTEGER` which are four bytes and overflow could crop up as we reach numbers not supported by `MEDIUMINT`.

ODBC works reasonably well in most general usage, but do count on finding some gotchas, and not necessarily just with the data types but also with SQL syntaxes, object definitions and so forth, as the document will discuss further. The ODBC API defines a set of specifications including data types, syntaxes and functions. Therefore, over the time, you may want to familiarize yourself with the ODBC specifications, because this is the specification that the developers of any given ODBC driver wrote to as well being the same specification Access team will use in generating ODBC calls.

ODBC Driver

The first step in any development involving a ODBC data source is to have a driver. The only exception to this rule is that if the backend is a SQL Server, we most likely already have the driver to SQL Server loaded in Windows by default. For any other sources, you may have to visit their respective site and download their driver. You also will have to come up with a plan to distribute the same driver with your final application for the users to be able to access the same ODBC data source.

It is important to be aware there can be more than one driver, and not necessarily merely different versions. Oracle is an example; there exists a driver for Oracle made by Microsoft and another made by Oracle themselves. Thus, it behooves you to ensure that you do not miss out on other available drivers and learn about what features they offer over their counterparts.

Let me take a moment to point out something else- OLE DB is a separate and newer specifications that's more broad than ODBC and provide for capabilities of interfacing with other data sources that are not necessarily RDBMS. This is worth mentioning because some sources may have their OLE DB driver that we can use instead of ODBC driver. The maturity of driver is also a factor. For instance, SQL Server's drivers is based on OLE DB, and will certainly perform better than if we used ODBC driver for same source but on the other hand, MySQL's ODBC driver is more mature than its OLE DB driver (at least at time of writing), and thus will perform better.

The next thing is to familiarize with what your ODBC driver is capable of. In a connection string, there is a parameter for "Options", which is usually a long integer. Alternatively, vendors may provide a list of options to be added to the connection string. You should look at the ODBC driver's manual to ensure you have all options you need turned on. A good example would be to ensure that a certain data type is correctly mapped to Access's data type. In my case, I ensured that Big Integer (64-bit) that were used as a primary key were turned off because Access does not support this. Passing this may cause Access to show those rows as `#Deleted`. In cases where `BIGINT` isn't a primary key, it usually is converted to text which also has the implications that it can't be manipulated as an integer such as performing arithmetic upon it on Access's side.

Another example where you may want to examine drivers' capabilities is its supports for Multiple statements and Active statements. Drivers capable of supporting such will be helpful in minimizing connections, and even in case where this is not possible, help you as the developer understand the limitation and design your application accordingly.

Different types of Connections and Data Access Technologies

As illustrated earlier in the overview, there are several layers that we may have to pass through between Access and our backend. It's just not enough to say "I'm accessing my data using ODBC." as it only describe what set of specifications we will be using, but not how we will get the information we need to communicate with the server, and what library will we be using.

Data Source Names (DSN) is one way of providing all the information we may require for connecting to the server. We also can opt to use DSN-less connection, which basically mean we provide all information in code. The performance difference may be minimal, but the administration will definitely differ. When we use DSN, it can be either stored as a plain text file or inside Windows' Registry hive. The information is stored plaintext, including the username and password. With DSN, We just need to use the string "`ODBC;DSN=<MyDSN>`" whenever we need to fill a connection string. However, when time comes to distribute the application, you may need to distribute the DSN especially if the objects in your application refer to the DSN in its `Connect` property. DSN-less connections circumvents the potential implementation issues by providing all information without performing a lookup in the registry or locating a file.

Regardless of whether you use DSN or not, there are security ramifications that has to be considered, which will be covered in the Security section. It also should be noted that you do not need to fill in all required parameters, in which case, ODBC driver will by default pop up a dialog prompting for the missing parameters.

Also, when we successfully connect to any data source, Access will cache the connection and re-use. This is very useful. For instance, it is possible to create objects such as passthrough queries with incomplete connection strings and allow Access to re-use the existing connection if one is already exists when the object is executed. At a minimum, three parameters must be specified to help Access identify the connection and thus associate with the cached connection; the driver name, the server and the database name.

Another layer that could get involved in this mix is either ADO or DAO, which can act as a high-level interface to the Jet or in cases where we bypass Jet, to ODBC or OLEDB layer, so we do not have to concern ourselves with tedious plumbing tasks. Each library has their strength and any developer would do well to use both. The general statement has been "Use DAO for Jet, use ADO for anything else", though I should qualify the statement to be: "Use DAO for Jet's objects such as linked tables and saved queries, and ADO for anything else." To explain why, if we tried to set a recordset variable based on a form's recordset and the form is bound to a table or linked table, a DAO recordset is returned, signifying that Jet uses DAO by default. Also, DAO is heavily optimized for interaction with Jet while ADO's design consideration was to be able to work with any data source. Of course, we can always set form's recordset to ADO recordset (given that we've followed the requirements for doing so) and still benefit from bound forms but this usually requires us to set it in code and not via Access UI.

Cursors and Recordset Types

Though a full and detailed discussion of cursors and recordset types is beyond the scope and probably merits its own FAQ, a brief touching upon is necessary.

When developing a pure Access solution, it is possible to go about making queries and recordset willy-nilly without much thought and thus consequences about precisely what those recordsets are. Thus, when we move to a ODBC backend, it is a frequent occurance that we find what worked fine and dandy was either abysmal or defective in the new environment. This is because in that context, the ADO's cursors and/or DAO's recordset types are much more important than they were under the original context.

To help illustrate why, suppose we wanted to get a recordset with this SQL statement-

```
SELECT *
FROM aTable;
```

If we used ADO's static cursors or DAO's snapshot recordset, what would happen is that a full table scan would be performed. On the other hand, if that query was performed via ADO's server-side keyset cursors or DAO's dynaset recordset, it will only ask for the key columns and perform a "lazy evaluation"; that is it fetches the rows only when it's actually requested to do (e.g. by issuing a `MoveNext` method upon the Recordset object for instance). See the section on Using Jet Intelligently for further explanation.

Even though the static cursor or snapshot recordset may perform a full table scan with the request, it may make more sense for say, combobox's rowsource or a report's recordsource because we do not need updatability in that context and this can be actually cheaper than having a keyset cursor or dynaset recordset. Another good reason is once the snapshot/static cursor has fully fetched the recordset, there is no further communication with the server, maximizing the availability and concurrency.

To further complicate the situation, though snapshot recordset may be preferred for most read-only operations, if we have amount of data so large that it takes long time to access all rows, it could be the case that dynaset recordset or keyset cursor may be actually faster and better for concurrency. It should highlight the importance of doing tests and verifying this is the appropriate choice for this specific query operation we want to perform.

Numbers of Connection

For most parts, we need not concern ourselves with how Jet handles connection, as it does an admirable job. What should call our attention is our own code that creates recordsets, execute SQL statement, setting a form's recordsource runtime or just about anything where we need to fetch or modify the data.

Jet does try to pool all queries to the server into one connection whenever possible, but if the driver cannot support multiple statements over single connection, Jet will open another connection to submit a statement. Therefore, if you have a form that has a combobox, Jet will send two queries, one for form's recordsource and another for combobox's rowsource. In case of lack of support for multiple statement, Jet will need two connections.

Even if your driver supports multiple statements, Jet may find it necessary to open a second connection if the driver cannot have more than one active statement. Active statement is when you fetch a big recordset and need to wait a bit for the full set to come over the wire. So back to that form with a combobox, if Jet find that the form's recordsource will take a while to be fetched, it will go ahead and open second connection to populate the combobox in order to allow for 'instantaneous' loading of a form from a user's POV.

Wherever possible, keep numbers of connection to a minimum. As mentioned before, Jet will try to pool all statements into one connection wherever possible, but Jet cannot help you out if you issue a query using a separate ADO `Connection`, executing SQL statement in code, and/or using DAO `ODBCDirect` workspace. One common mistake crops up in various forums is that records has been locked or there's a write conflict even though there's only one user using the application. This can be triggered by attempting to perform an operation in separate connection in VBA upon same table that is bound to a already open form. From the database's perspective, this is two separate requests; it doesn't even know if the requests were made by same user or application. All it knows that one request has been met, and it can't complete the second request until the first request is done. Thus the user get a lock or write conflict error.

It is also helpful to verify that your driver is configured to support multiple & active statement; it may come with capability but is not enabled by default.

Locking & Transactions

Various vendors have their own implementations of locking mechanisms which can factor in the development. As shown in the previous section, a careless block of code that creates two different connections trying to access same block of records could potentially lead to a deadlock. However, it need not be just the same record but a table and so forth. It is your responsibility to verify that your application design, especially with bound form, do not inadvertently create unnecessary lock contentions and thus harm the concurrency of the overall project.

It should be also noted that within Access environment, Jet/DAO has no control over how the locking are performed on the server side, so the settings for locking will not apply. In that case, the defaults used by server usually applies. In case where you require a specific locking behavior, consider using a stored procedure or at least either pass-through query or command in native dialect. The SQL may look something like this:

```
SELECT *
FROM aTable
LOCK FOR UPDATE;
```

Note: Consult your own backend's documentation for specific syntax used to provide hints for locking.

Even better, consider performing the work in a transaction instead, which usually deal with the concurrency issues more efficiently than locks. Jet is capable of handling transaction with ODBC source, provided that the backend supports transactions as well, so you have choice of doing it via Jet or directly upon the backend whether as a stored procedure or a ADO command. Unfortunately, even wrapping it in a transaction does not render it immune to deadlocks though it helps solves several of concurrency issues. It should be also be noted that even though Jet, and possibly the backend as well, is capable of nesting transactions, ODBC does not support nesting of transactions.

ADO also provides some control by setting the `LockType` property but in this case, the control depends on whether the provider can support such features and it still is your responsibility to ensure that all ADO commands use the appropriate locks. ADO/OLE DB may be capable of nested transaction but only if the provider supports this feature.

Networking

Depending on the specific backend used, and the scope of use, networking, especially network security may be a hurdle to overcome. This is especially true if we need to allow remote connections, have several networks within one company or a tight network policy. We may also need to verify that every users' endpoint has a clear path to the server and firewalls are configured to permit the traffic through. This may require consultation with the network administrator.

Data types and SQL Standard, ANSI-92 Option

As mentioned earlier, there are no vendors that follows the SQL standard exactly. This is partly because the standard was open to interpretation, and partly because vendors have their business plans to pursue. This is apparent when comparing different syntaxes along vendors to accomplish the same thing and differing definitions of certain data types. For those interested, someone is working on compiling a comparison chart of SQL standard and major RDBMS's implementation, linked in the external links section.

One commonly cited 'gotcha' is how Jet and SQL Server interpret the `BIT` field. In Jet, `Yes/No` is essentially a `BIT` data type, with no null allowed. Presented to ODBC, it is interpreted as a `BIT` and SQL Server receives it as such. No problem so far. But SQL Server can allow null values for `BIT` data type and pass it back as null. ODBC will faithfully translate that back to Jet. In case where a null `BIT` is passed to Jet, it could potentially be interpreted as a table lock, which would be an unpleasant situation to be in.

Another common pitfall is how several RDBMS provide supports for `UNSIGNED` option to be specified for numeric data types, something that Jet does not support. Therefore, if we defined a key as `UNSIGNED INT`, the size is still 4 bytes and usable by Access but Access will interpret it as signed `Long Integer`. Thus when we reach 2,147,483,648th record, trouble could appear, perhaps by overflow error or representing the record as `#Deleted`.

It helps to review the different data types that are supported by Jet (and also by VBA which differs slightly), and the source of choice and how they get mapped. As a general rule, the closer we stick to SQL standard, the better the chance that our application won't be hit by weird bugs due to the disparity. Some vendors may provide their own custom data types and it *may* be possible to pass it along to Jet, but when it can be done using SQL standard data types, we'll be sure to find our lives easier for it.

One option the developers have available is to set the Access database file to use ANSI 92 standard instead of Jet's usual SQL. For anyone upsizing their already existing application, that option is more likely to entail more unnecessary work, but for those starting from scratch, it can help provide some degree of consistency and allow copying and pasting the SQL from the source's development studio to the Access's query builder, for example, and it'd still work whether it's a passthrough or not.

Jet and ODBC; how to use Jet intelligently
Note: All SQL statements in the section has been tested with Access 2003 -> SQL Native Client/SQL Server 2005 and MyODBC 5.1/MySQL 5.1

Whenever we use a linked table as our source for queries or recordsource or such, we are involving Jet in the operation. As shown before, Jet is quite conservative and intelligent in data management, but it's just a software that will obey our whims, even if our whims are not ideal. Thus we need to be careful that we do not write queries that forces Jet to perform a full table scan and local evaluation when it should have been passed off to the server.

One easy way to force unnecessary evaluations is to use Jet expressions and/or VBA functions:

```
SELECT *
FROM aTable
WHERE Not IsNull(TextValue)
  AND Year(SomeDate) = '2009';
```

Note: in SQL Server, there exists a function for `ISNULL()`, but it actually act more like Access's `Nz()` function so even if the query was sent to the SQL Server unparsed, it may not return the expected results!

Neither ODBC or the server has no knowledge of VBA's `IsNull()` function and though both understand Year function the expression is evaluated on Jet's side anyway, so Jet tries to complete the request by first collecting all data with the SQL Statement:

```
SELECT id, TextValue, SomeDate
FROM aTable;
```

Bringing over both key column and the column to be evaluated upon, and doing the evaluation locally, tossing out the records. This is waste of bandwidth and could hurt concurrency as well.

One possible way to fix this is to use standard SQL:

```
SELECT *
FROM aTable
WHERE TextValue IS NULL
  AND SomeDate BETWEEN '2009-01-01'
                 AND '2009-12-31';
```

As mentioned earlier, `Year()` function is understood by both ODBC and the backend, but when it's used in a expression such as:

```
Year(SomeDate) = '1992'
```

It's then forced upon Jet to evaluate it locally because `Year()` returns a numeric data type which is being compared with a string data type. This can be fixed by changing it into:

```
Year(SomeDate) = 1992
```

and will thus be passed back to the backend for evaluation server-side.

As shown above, ODBC provides some support for certain functions. For example:

```
SELECT *
FROM aTable
WHERE WeekDay(SomeDate) = 1
   OR Len(TextValue)>=5;
```

Will show up on the backend log as this:

```
WHERE (({fn dayofweek( SomeDate )}= 1 )
   OR ({fn length( TextValue )}>= 5 ))
```

It may be a good rule of thumb to say that when in doubt, avoid using non-SQL functions, especially for WHERE clause. It is possible to use Jet's native functions as an expression in part of SELECT without any ill effects upon the data retrieval:

```
SELECT Iif(Len(TextValue), TextValue, "Blank")
FROM aTable
WHERE SomeDate BETWEEN #1/1/2009#
                   AND #12/31/2009#;
```

This will then turn up in the backend's log as this:

```
SELECT TextValue
FROM aTable
WHERE SomeDate BETWEEN '2009-01-01'
                   AND '2009-12-31';
```

with the `Iif()` evaluated on client's side. We can also elect to require processing to be done on the server's side by doing a pass through query or using ODBC-supported functions:

```
SELECT CASE
   WHEN TextValue IS NULL THEN 'Blank'
   WHEN TextValue = '' THEN 'Blank'
   ELSE TextValue END
FROM aTable
WHERE SomeDate BETWEEN '2009-01-01'
                   AND '2009-12-31';
```

You have the luxury of choosing which side you want to process certain evaluations and if used properly, can gain lot in scaling up the application with managing the processing loads all around.

The above discussion also applies to other part of SQL statements such as using expressions in `ORDER BY` clause, and it is your responsibility to verify that you are writing efficient queries that does not force unwanted evaluations upon the wrong side.

Another way we can help Jet is to be selective of our data requests, by adding a `WHERE` clauses for all bound forms. This is especially invaluable when we have so many records. We, being humans, can not possibly browse million rows, let alone manage them in any meaningful sense. For that reason, it's always good to have a clause to keep the dataset at a certain size that is manageable for our end users' workflow. One common way to provide full access to the whole table is to provide only X most recent entries, with a button to access the 'archive'.

The difference can be seen in how Jet sends the statement to the source when it needs a set of keys for record navigating. For a query or recordsource using Dynaset-type recordset without a WHERE clause:

```
SELECT id
FROM aTable;
```

When the form is opened, Jet will then use the previously fetched keys to get the full row. Assuming we're talking a single form displaying only one record, the next statement will be:

```
SELECT id, somedata
FROM aTable
WHERE id = 1;
```

Adding the WHERE clause will most assuredly reduce the expense of first and necessary request for the keys.

Using continuous form or datasheet view will require Jet to fill more rows in one go; usually it fills as many as needed to paint the screen or a fixed amount which is cached. So for such form, the request may look like this:

```
SELECT id, somedata
FROM aTable
WHERE
  id = 1 OR id = 2 OR id = 3 OR id = 4 OR id = 5 OR
  id = 5 OR id = 5 OR id = 5 OR id = 5 OR id = 5;
```

Assuming Jet needs only five records to paint the entire screen, it fetches that many while filling the unused remaining parameters with the last key.

If the form happens to include a combobox, a listbox or any other object that has its own rowsource (or in case of subforms, recordsource), those counts as a separate query. It is possible to find that a given combobox performed acceptably in unspilt database now performs poorly when it's split or migrated to an ODBC backend. Other than obvious answer of creating index upon the source table, a viable option is to keep your lookup tables, especially if it's not updated frequently, local to the application. This also help in reducing network traffic and where updates are needed, several nice folks has written and shared front-end updaters, of two possible are linked below, that can be used to keep the lookup tables current as well solving the more general problem of maintaining the front-end.

The whitepaper, linked in external links at end of the article, covers those issue in greater depth, and this is my opinion that this is a mandatory reading for any developers who want to use Access as a front-end to any RDBMS source.

Bound forms' recordsource and updateability

Bound forms deserves a special mention because they enforce additional rules on what we can use as a recordsource. It must be understood that the rules for updateability is same whether the source is a local Jet table or ODBC tables, and literature abound discussing those rules. Allen Browne's article is a good starter, linked below. We will concentrate on the special cases where we require use of backend's SQL extensions or other functionalities not known to Jet and/or ODBC.

One distinct advantage ADO have over DAO, including ODBCDirect, is it enables us to pass a SQL string in the backend's native dialect and return a recordset that is fully updateable and thus can be bound forms, under certain constraints. The article discussing about binding an ADO recordset to a form is linked below. This is useful in passing certain parameters such as locking or indexing hints that is only understood by the backend as well as any other expressions provided they do not break the same set of rules affecting updateability.

Another good reason to use ADO recordset is the ability to disconnect the recordset by using this VBA code fragment:

```
With rs
   .ActiveConnection = <Some connection object>
   .Open "SELECT * FROM aTable;"
   .ActiveConnection = Nothing
End With

Me.Recordset = rs
```

You can then control how and when the updates to the form will be actually sent back to the backend, which is useful when you need to be able to do batch processing and/or perform integrity checks across multiple recordsets. All edits would be then committed by a call to UpdateBatch method.

In cases where passthrough query is used, some has solved by dynamically unbinding the form just in the time. One example would be to bind the form to read-only recordset and providing a command button that would then blank out the form's recordsource while preserving the data already loaded in the current record and opening up the form for editing. This enables us to capitalize the convenience of bound form doing all the navigation work for us while keeping full control over what we can view and manipulate.

Another technique we have available to us is to make use of local temporary table which can be employed in conjunction with a stored procedure or non-updateable query and binding the form to the temporary table. This provide us similar benefits as using ADO's disconnected recordsets but can be used upon otherwise nonupdateable query such as a UNION'd query where you want to to retain the ability to edit without requiring another roundtrip.

Keep an eye on SQL log when developing

As most RDBMS provide tracing functionality, you almost certainly will want to take advantage of this as you develop. This will provide you feedback on whether your queries and recorsource are well-formed and reasonable. What you place in recordsource could be very different from what shows up in the log, but as a rule, we want to ensure that Jet is just fetching keys then few records, just enough to paint the screen without any redundancy or forcing a local evaluation and measuring whether the selected recordset type/cursor type is appropriate for this operation.

Another option is to use Jet's builtin tracing function, which can be toggled on via registry setting, but I would prefer to use `DBEngine.SetOptions` so the changes aren't permanent. It can then generate both SQL trace and ODBC trace files. However, be aware that the SQL generated by those files may not be precisely same as the SQL that the backend ultimately receives from its ODBC driver. For that reason, I usually prefer to watch backend's tracing logs.

This is quite important because when we move into complex forms with subforms and several controls (especially comboboxes and listboxes with their rowsources), with the ultimate effect that multiple queries are required to support one "record". Thus, the logs can help you to know the "price" of the operation being performed and help you make a informed decision whether the price is justified.

Another important use for this is to help us make decisions on which columns will need indexing and how to create an effective index that Jet can make use of in maintaining both your server's and your application's performance.

Error Handling with ODBC errors

When manipulating ODBC data, it is possible that it will be necessary to use a different process for handling errors. In a pure Access solution, one only had to look at `Err` object's number and description to identify the cause of error and move on. With ODBC errors, we have to look at different sources, depending on the exact circumstances.

Generally, it's easier to handle errors with connections, commands and recordsets created by yourself via code using the Errors collection of either DAO or ADO. Microsoft is kind enough to provide numerous articles of how to handle such errors for login and objects created via code as listed in external links section. However, bound form may be a bit more problematic. The current behavior is that form's `OnError` event can capture the generic "ODBC Error--Call failed" error but does not report the specific errors as provided by the specific backend, making it difficult to handle error contingent on what specific error is caused by the recent operation.

One possible workaround is to create a pass-through query or ADO command that request for the recent errors from the server in its native dialect.

Security

We must realize there more than one security mechanism at play here. At most, there can be four;

- Host/Network Access Authentications & Permissions
- Windows Filesystem Permissions
- RDBMS Security Model
- Access Security

There are several ways to implement security for specific needs, and that can involve one, two or all mechanisms.

The thing is that each mechanism, at least out of the box, is ignorant of other mechanism, and will not in general interact with other mechanisms. It only cares that its own set of credentials is satisfied, regardless if the other mechanism sets of credential were. That is a important point here.

Here's one example of common implementations: Some may not even bother with Access security at all, deigning to put the Access file in a folder managed by Windows permissions so workers who do not need to use that database can't open the folder and subsequently cannot get to the .mdb at all, while workers who need it can use it. This implementation is sufficient for where we trust the workers to be honest with their data processing and there is no need for giving different permission upon same table to different groups of worker.

This should hint that the more complex interaction we have with a database, so will be the security.

Some vendors, notably SQL Server and Oracle have their product integrate Windows' logins with its own security mechanism, so for that context, #1 and #3 in the list can be combined. But merely logging in Windows does not give you the golden ticket to the tables in SQL Server or Oracle. You still have to connect to the source, and it will check your credentials at time of connecting.

In case of Access connecting to RDBMS back-end, Access can allow us to save the password in the connection string. This can be a good or bad thing, depending on our specific security needs. If we never ever want a dialog from the RDMBS to pop up asking user for the password, saving password is certainly one way. In such cases, the password is stored as part of Connection string, which is stored in `Connect` property of any `TableDef` or in `Connect` column of hidden system object `MSysObject`.

Whenever we do not want the password for connecting to RDBMS to be saved, we can do so in two ways, or combine both:

- Writing the DSN, omitting the password and asking the user to fill it in
- Implement your own login windows and pass it to the connection string

In SQL Server's and Oracle's case of using Windows login, you don't actually need to log in but this should still prevent the information about Windows login being stored inside Access.

It also goes without saying that it would be futile exercise to attempt to connect to the host containing the database if the host and the network is not configured to permit such connections and this also can be used as a security mechanism to restrict connections to only a certain subnets or only within LAN and not from outside.

Where does Access security comes in the picture? Nowhere. It's not even relevant because at this point, we're still talking about logging to the RDBMS and accessing the data and thus still in context of RDBMS security mechanism as well the operating systems & networking systems granting us the access to the RDBMS.

Does it mean that there's no point of using Access's security? Hardly. ULS (at least if we are using .mdb files) still can be usable as a permissions manager to specific Access object; you could deny all permissions to the tables itself and give users permission to the forms only thus ensuring that they don't edit the table directly. This is something that Windows and RDBMS security cannot stop because once you've already logged in and connected, you are now past their gate. To put this bluntly- they don't know and don't care about the manner of how data is modified as long their *own* credential is satisfied. This means if users can open the tables after successfully logging in, they have the capability to enter data outside of any validation you may have had set up inside forms. Thus, Access (and as well as any front-end clients) must manage their own object and ensure that users only go through the proper channels without getting cute and using shift bypass or linking to the table or whatever they come up with circumventing the security.

But that's not the only approach we can take, as this require us implement all (or at least just RDMBS's and Access's) security mechanism and can be labor intensive for little gain. If company can't trust their employees, then they probably have bigger problems than properly securing it. One possible "lazy" implementation could be to require a timestamp and userstamp for any and all data modification, allowing us to stay within RDBMS's security mechanism, and if one user goes rogue, it's on the trail and we can rollback the changes and fire the jerk.

So in a nutshell, there are basically three generic ways to secure the whole package:

1. Leave Access unsecured but require user to type in a password when opening a form that requires data from ODBC. Without any coding on your part, they will get a dialog from the ODBC driver itself, not from the Access. Alternatively you could use a custom form to take in the username and password and pass it to the connection string.

2. Use Access's User-Level Security and use same credentials for backend so Jet will automatically carry over the credentials to the backend. You will need to check the registry setting to ensure that Jet is configured to try its credentials first. Keep in mind that if the credential fails and isn't handled, Jet will then display a dialog from ODBC driver.

3. Use Access's User-Level Security to secure the database objects such as local queries, forms and modules but do not use the same credentials for the backend. This either means the user will need to type in two separate logins.

Alternatively, you could only prompt for login to Access itself and automatically log into backend using a stored credential within Access. However, this carry with the risk that the credential may be mined from the file by a malicious user and then will be used to connect to backend. Therefore you will need to take extra precautions to prevent this from happening.

Ultimately, the security mechanism you design will depend on what your project requirement are. Hopefully the above discussion has helped in providing some ideas of what are possible.

In any cases, remember that any connections used in Access will be cached and re-used. There is a security ramification: Even if you have it setup that passwords are not saved to the tables' Connect property and closed the database, but not Access itself, the connection is still sitting there and can be re-used. This is easily fixed by ensuring that Access application quit when the last form closes.

A final word of caution: The act of securing is quite insidious; it's far easier to fool ourselves into thinking it secured than it is to actually secure it!

Special Note on Security in Access 2007

As ULS has been deprecated and removed from ACCDB/ACCDE file, the above discussion involving ULS is not so relevant to Access 2007. Thus developers basically have three choices in that regard: 1) Continue to use MDB/MDE, which Access 2007 still supports, including using ULS, 2) Accept the limitations and work within it, or 3) implement a DIY solution such as vPPC. Regardless of the choice made, data itself still can be effectively secured with a proper implementation of the backend's security, and compiling into ACCDE and MDE usually take care of managing "access" to various Access objects (with some caveats, of course).

External Links

Refer to included ODBC External Links.rtf or the post at Utter Access or Access World Forums.

Contributors *(listed in alphabetical order)*
ace
boblarson
datAdrenaline
DougY
LPurvis
strive4peace2009
The_Doc_Man

As always, any feedbacks or request for clarifications or correction, minor or major, are welcome.