Database Performance

Using a Remote Backend

How to obtain reasonable performance with a remote database and Access client

Jack D. Leach

jleach@dymeng.com

Access Lunchtime, November 27th, 2018

I N D E X

01

Getting **GROUNDED**

WANs, ODBC and RBAR

Recognizing the **Bottlenecks**

WANs, ODBC and RBAR

Wide Area Networks

Fundamental basis of remote performance issues

We have little to no control over this

Reference Albert Kallal's classic: http://www.kallal.ca/wan/wans.html

Round-trip Communications
10-100x **SLOWER** than LAN!

Every request to the database makes *at least* one round trip

Recognizing the **Bottlenecks**

WANs, ODBC and RBAR

ODBC

Open **D**ata**b**ase **C**onnectivity

Many implementations – ODBC is a standard, not a tool

At its basic level, acts as a translator between SQL dialects

At its fully functional level, acts as a contract per your requests

Different ODBC Drivers may impact performance

Many available – prefer SQL Server Native Client driver for Azure

Recognizing the **Bottlenecks**

WANs, ODBC and RBAR

RBAR

Row **B**y **A**gonizing **R**ow

SQL/SQL Server is a **Set-Based** language/server

SQL operations are optimized to act on sets of data at a time, not single rows

Given certain requests, the database engine may have to drop into "single row" processing mode (i.e. – Cursor-Based processing). This is extremely inefficient!

Move the whole crate at once, not one blueberry at a time!

02

ODBC as a **Translator**

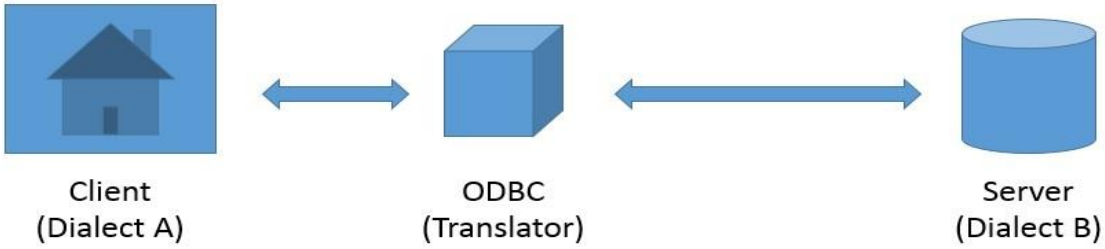
A peek into the black box

ODBC as a **Translator**

Bridging the Gap between SQL Dialects

JET/ACE SQL and T-SQL (SQL Server) have different dialects

ODBC acts as a translator between the dialects



ODBC as a **Translator**

Bridging the Gap between SQL Dialects

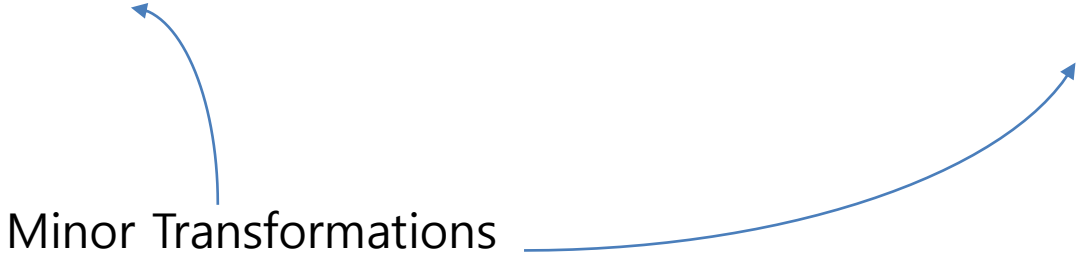
A Simple SQL Statement and Easy ODBC Translation

JET/ACE Dialect

```
SELECT
  ID,
  CustomerName
FROM Customers
WHERE CustomerName = “ACME”
```

T-SQL Dialect

```
SELECT
  ID,
  CustomerName
FROM Customers
WHERE CustomerName = ‘ACME’
```



ODBC as a **Translator**

Bridging the Gap between SQL Dialects

Slightly more complex, but still very easy for ODBC to translate

JET/ACE Dialect

```
SELECT c.ID, c.CompanyName
FROM (dbo_Companies AS c
LEFT JOIN dbo_CompanyLocations AS cl
  ON c.ID = cl.CompanyID)
INNER JOIN dbo_CompanyClients AS cc
  ON c.ID = cc.CompanyID
WHERE cl.CompanyID IS NULL
GROUP BY c.ID, c.CompanyName
HAVING COUNT(*) > 5;
```

T-SQL Dialect

```
SELECT c.ID, c.CompanyName
FROM dbo.Companies AS c
LEFT JOIN dbo.CompanyLocations AS cl
  ON c.ID = cl.CompanyID
INNER JOIN dbo.CompanyClients AS cc
  ON c.ID = cc.CompanyID
WHERE cl.CompanyID IS NULL
GROUP BY c.ID, c.CompanyName
HAVING COUNT(*) > 5;
```

ODBC as a **Translator**

Bridging the Gap between SQL Dialects

Very simple for us, **very difficult for ODBC!**

JET/ACE Dialect

```
SELECT ID, CompanyName
FROM dbo_Companies
WHERE NZ(CompanyName, “”) = “MyComp”;
```

T-SQL Dialect

```
SELECT ID, CompanyName
FROM dbo.Companies
WHERE ?????????
```

ODBC doesn’t understand the NZ() function, and thus can’t convert it

ODBC resolves this by **fetching the entire table** and processing the criteria locally

ODBC as a **Translator**

Bridging the Gap between SQL Dialects

In Contrast: Sargable vs SELECT Transformations

JET/ACE Dialect

```
SELECT ID, Nz(CompanyName, “”)
FROM dbo_Companies
WHERE CompanyName = “MyComp”;
```

T-SQL Dialect

```
SELECT ID, CompanyName
FROM dbo.Companies
WHERE CompanyName = “MyComp”
```

ODBC convert NZ() within the SELECT, but can still return a restricted set

The NZ() from the SELECT statement can be done locally, *after* the results back

ODBC as a **Translator**

Bridging the Gap between SQL Dialects

Invoking RBAR over a WAN

JET/ACE Dialect

```
SELECT ID, CompanyID, Nz(CompanyName, “”)
FROM Companies AS c
INNER JOIN (
  SELECT t.CompanyID, t.ClientName
  FROM CompanyClients AS t
  WHERE Nz(t.ClientName, “”) = “blah”
    AND t.ClientClass = c.Class
) AS cc ON c.PrimaryClient = Nz(cc.ClientName, “”)
```

T-SQL Dialect

Ummmm… what?

No chance of any direct conversion

ODBC’s has to get us results… but how?

ODBC as a **Translator**

Bridging the Gap between SQL Dialects

A Theoretical Worst-Case Scenario

1) Determine that it can't process the join serverside (unrecognized function)
2) Determine that it can't process the subquery for the same reason
3) Determine that the subquery is correlated (relies on outer query)
4) Pull the entire Companies table for local processing
5) Selects the first record, attempts to process the subquery
6) Pulls the subquery table for local resolution
7) Resolves the join to see if the record should be included in the query results
8) Repeat steps 5-7 for each record in the original Companies table

RBAR over a WAN

This can be the difference between a two-second query and a two-hour (or two-day) query

```
SELECT ID, CompanyID, Nz(CompanyName, "")
FROM Companies AS c
INNER JOIN (
  SELECT t.CompanyID, t.ClientName
  FROM CompanyClients AS t
  WHERE Nz(t.ClientName, "") = "blah"
    AND t.ClientClass = c.Class
) AS cc ON c.PrimaryClient = Nz(cc.ClientName, "")
```

ODBC as a **Translator**

Bridging the Gap between SQL Dialects

Takeaways

- ODBC's highest priority is results: performance comes second
- We pay for translation complexity with performance
- JET/ACE and VBA functions should be avoided except in top-level SELECTs
- Normalization has significant impact on efficiency
- Don't try to know it all: ODBC and DB engines are black boxes to most

Remember: WANs, ODBC & RBAR are the three core performance factors

Always consider what you're asking your queries to do!

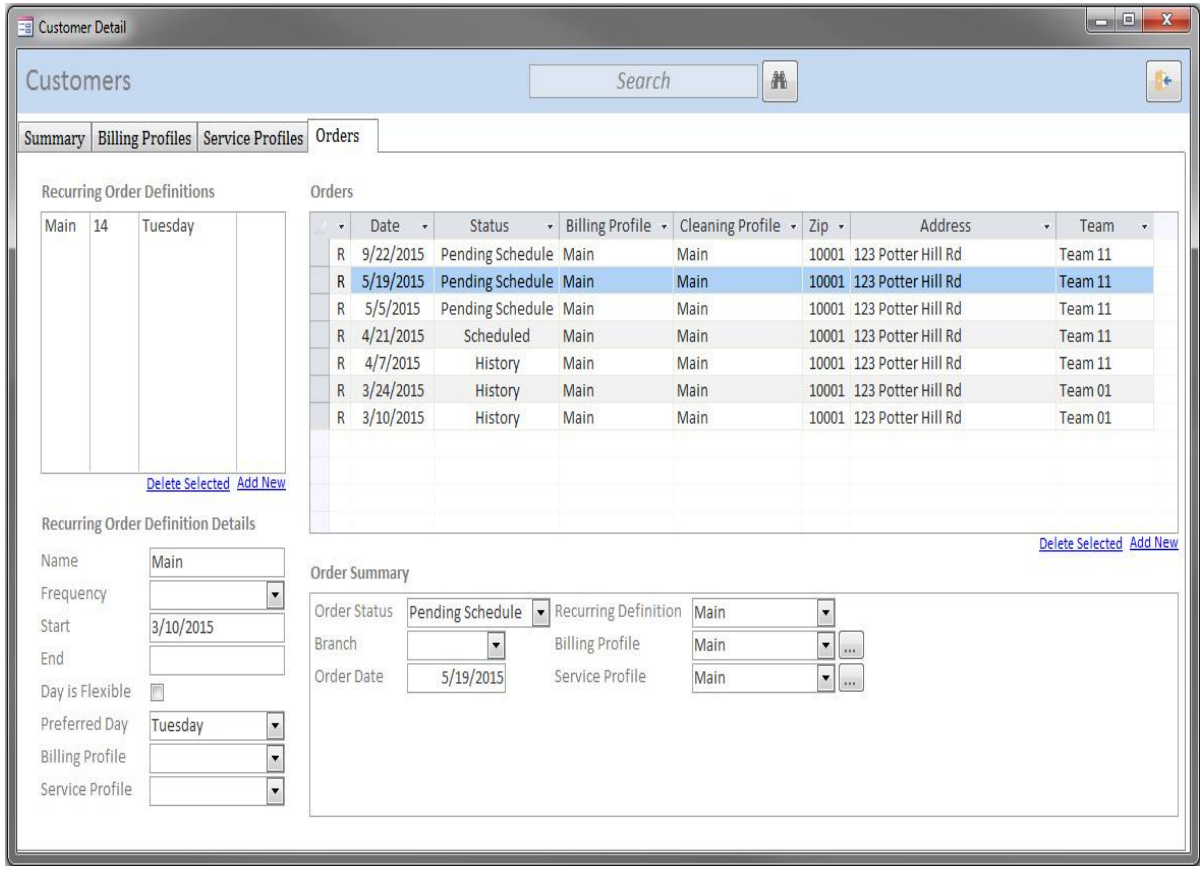
03

Understanding our **Data Usage**

Classifying our information to better serve

Our **Data Usage**

Classifying our Data



List Data

Small sets of read-only data, often used to supply list sources

Primary Data

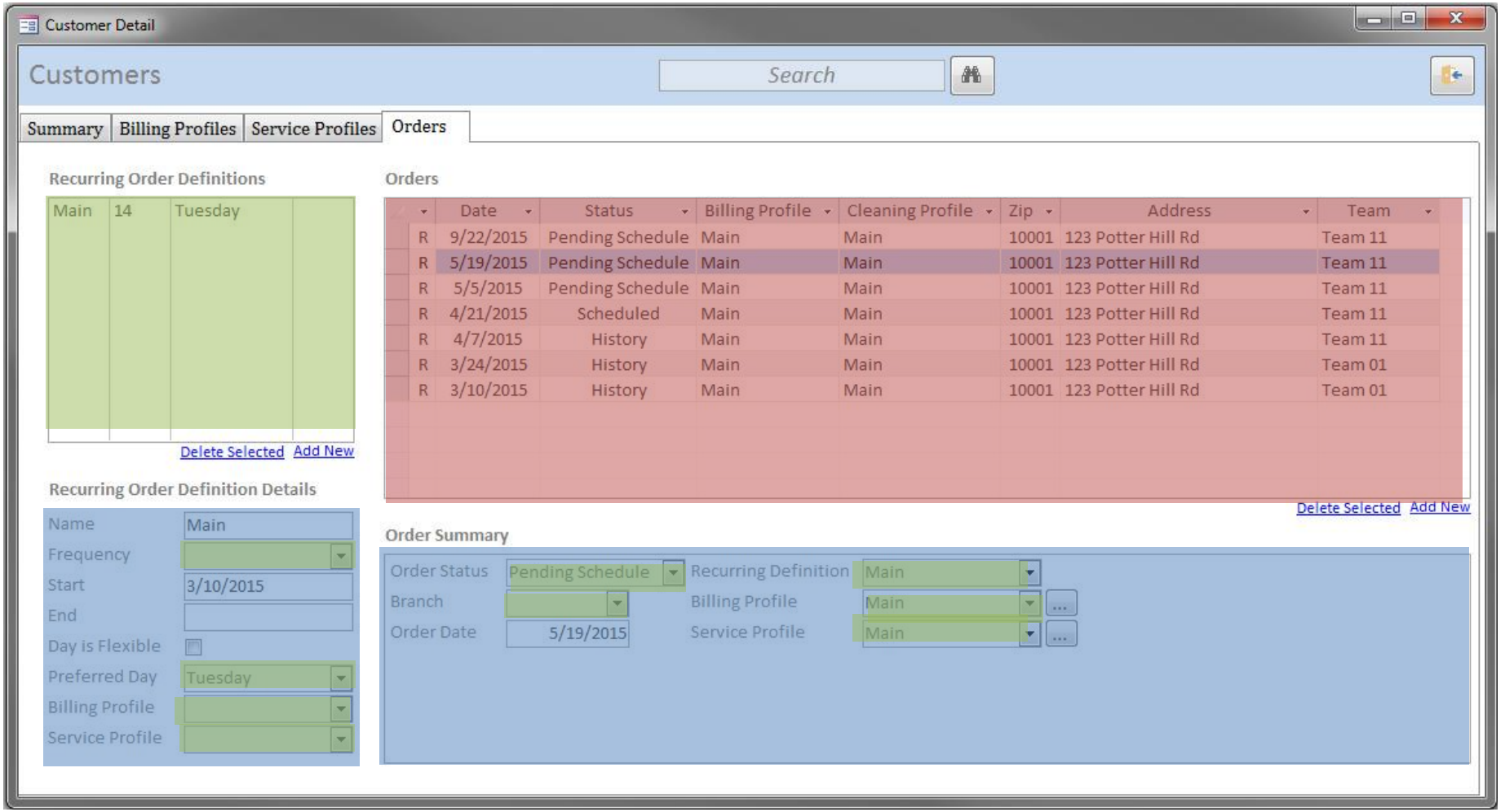
Our usual day to day recordsets. Read-write, two-way binding

Complex Data

Results of "heavier" queries: summaries, reports, etc. Typically read-only

Our **Data Usage**

Classifying our Data

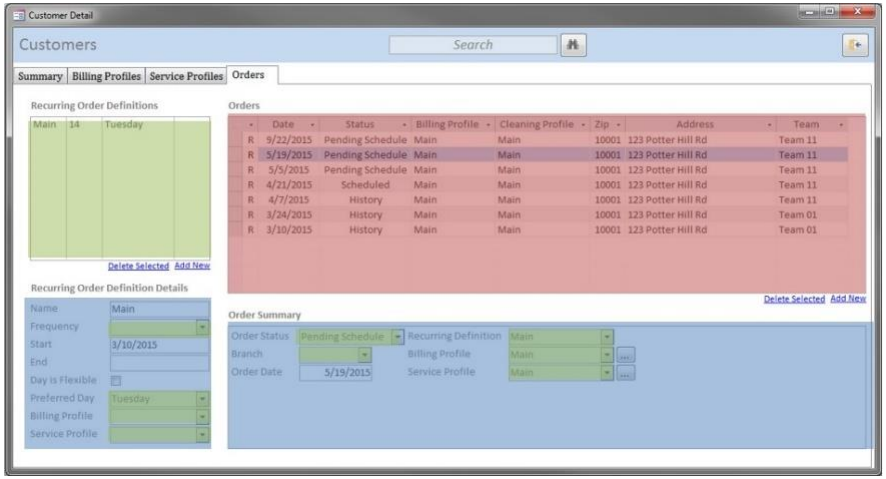


Our **Data Usage**

Classifying our Data

Data Source Requirements

- Recurring Order Defs
- Recurring Order Def Detail
- Frequency Dropdown
- Preferred DOW Dropdown
- Billing Profile List
- Service Profile List
- Orders Summary List
- Order Detail
- Order Status List
- Company Branch List



- About 15 different data source requests
- Figure at least one of these will result in a WAN-RBAR operation
- Figure each data request is a minimum of 1 second
- We can guess **at least 10 seconds** *at best* to open this form (and that's completely disregarding the WAN-RBAR)

Our **Data Usage**

Classifying our Data

Data Classes

- List Data
- Primary Data
- Complex Data

Data State Classification

- Static Data (rarely changes)
- Dynamic Data (changes often)

Considering these two characteristics for each of our required data sources (and bearing in mind our WAN-RBAR knowledge), we can weigh the pros and cons of various techniques to best balance our performance and ease-of-development

Core Techniques

- Caching Data (storing data locally for a session)
- Temp Data (importing for local processing)
- SQL Views/Procedures (letting the server do the work)

04 Working with **the Data**

Down to the nitty-gritty

Caching **Data**

Static Data, Locally Available

Excellent choice for static, list style and/or read-only data

- Cache data to local tables on application startup
- Data transfer "logic" is very simple: no complex queries involved
- Cuts out a large amount of round-trip remote data requests
- Data classes suitable for caching rarely change
- A surprisingly large amount of records can be transferred in short timeframes on startup: 10-300k records can often be loaded in seconds
- Easy to extend: incremental caching, in-app refresh points, optional refresh on a single row, etc.
- Almost every other development platform/paradigm relies on caching

Caching **Data**

Static Data, Locally Available

Important! **Avoid Heterogeneous Queries**

A heterogeneous query is one that requires a comparison from both the remote server and the local data source

One may be tempted to use a frustrated join to update non-existing values locally:

```
INSERT INTO TargetTable
  (ID, OtherField)
SELECT s.ID, s.OtherField
FROM SourceTable AS s
LEFT JOIN TargetTable AS t
  ON s.ID = t.ID
WHERE t.ID IS NULL;
```

A heterogeneous query will cause a RBAR comparison via ODBC to resolve. Use with caution!

Prefer a dual-recordset approach instead: one to read, one to write. Use a VBA loop to read/write. This seems counterintuitive but tends to work better as it avoids the heterogeneous query.

Working with **Primary Data**

Day to Day Recordsets & Detail Edits

Primary Data is your regular detail form editing type of data

Performance Considerations?

Not much to worry about. Follow basic best practices and this mostly falls into place without a lot of extra work on our end

Use a "detached" recordset and select only the required record:

```
Private Sub Form_Load()
  Me.RecordSource = "SELECT * FROM Table WHERE ID = " & Me.OpenArgs
End Sub
```

Prefer to bind combos/listboxes to local data caches where possible
(Access is very liberal about requerying combo sources…)

Maintain ease-of-use for being bound directly to the source

Working with **Complex Data**

Utilizing Views and Stored Procedures for read-only data

Characteristics

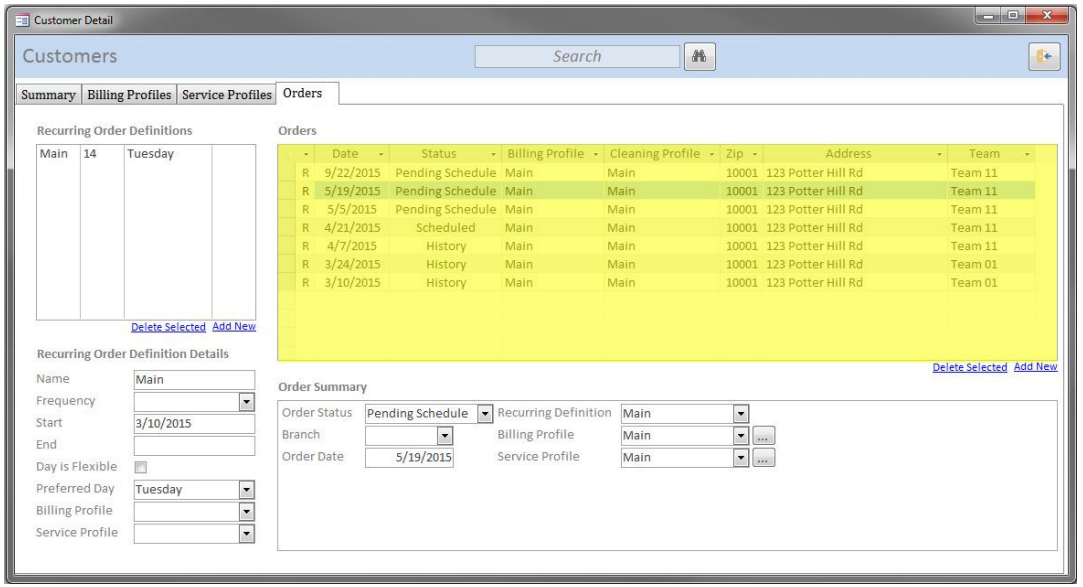
- Almost always read-only
- Multiple tables, numerous joins, extensive criteria

Common Use Cases

Summaries/Dashboards	Reporting
"Friendly" lists	Exporting

Basic Sourcing Concepts:

- Leverage server processing as much as possible
- If further local processing is required, load base data set from server into temp table, then process the rest locally (avoids heterogeneous queries)



Working with **Complex Data**

Utilizing Views and Stored Procedures for read-only data

SQL Server View

A can be considered the SQL Server equivalent to a saved Query in Access

SQL Server offers extensive native performance tuning for Views

Once created, can be linked to and read like any other table from Access
Although, creating writable Views can be a little tricky

Defined by pretty much any valid SELECT statement
But no sort order allowed, generally speaking

View definition itself *cannot be parameterized*

View Creation (SQL Server)

```
CREATE VIEW dbo.MyCoolView AS
    SELECT ThisField, ThatField
    FROM dbo.MyTable
    WHERE X = Y;
```

Usage (Access)

```
SELECT ThisField, ThatField
FROM dbo_MyCoolView
WHERE Z = A
ORDER BY Q DESC;
```

Working with **Complex Data**

Utilizing Views and Stored Procedures for read-only data

Stored Procedures (aka procs, sprocs, stopro, storp, sp, etc.)

- Server based, no equivalent in Access
- Somewhat akin to writing code in VBA (block instructions)
- Can process data and/or return records (or many other things)
- Returning records is optional: can be used to invoke commands only
- Able to accept parameters (unlike Views)

```
CREATE PROCEDURE dbo.MyCoolProc @MyParam INT AS
BEGIN
   SELECT * FROM SomeTable WHERE ID = @MyParam
END;
```

```
CREATE PROCEDURE dbo.MyCoolProc AS
BEGIN
   GRANT EXECUTE ON OBJECT blah TO blah
END;
```

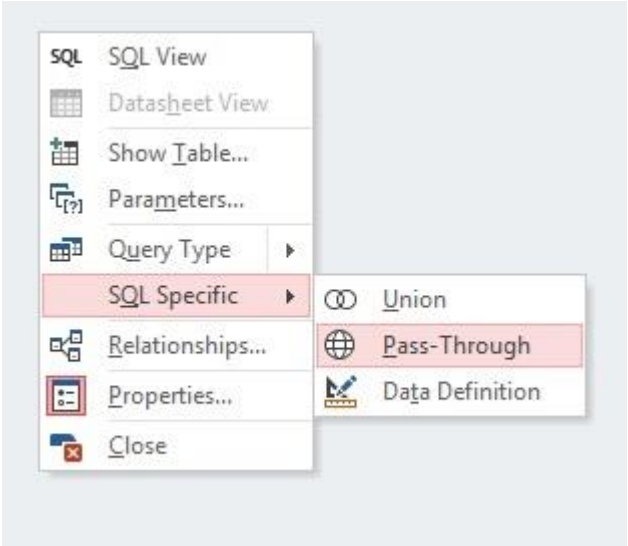
Working with **Complex Data**

Utilizing Views and Stored Procedures for read-only data

Passthrough Queries

Access-specific: change Query Type to Passthrough and set whether it Returns Records via the query properties

Instructs ODBC to ignore the contents of the query and send directly to the server without trying to translate (e.g., the query "passes through" the ODBC driver without intervention)



Must be written in T-SQL, as it received and processed on the server exactly as written

Working with **Complex Data**

Utilizing Views and Stored Procedures for read-only data

Quick Tip for Passthroughs:

Create a "template" passthrough query object. Set the SQL to `SELECT 1;`

Use a VBA function to alter the query as required

Utilize your VBA function throughout your application

```
Public Function GetViaPassthrough(sql As String) As DAO.Recordset
  Dim qd As DAO.QueryDef
  Set qd = CurrentDb.QueryDefs("TemplatePassthrough")
  With qd
    .sql = sql
    Set GetViaPassthrough = .OpenRecordset(dbOpenSnapshot)
  End With
  Set qd = Nothing
End Function

Public Function DoSomething(BranchID As Long)
  Dim rs As DAO.Recordset
  Dim sql As String
  sql = "EXEC dbo.MyCoolProc @BranchID = " & BranchID
  Set rs = GetViaPassthrough(sql)
  '... do stuff
  rs.Close: Set rs = Nothing
End Function
```

05

More **Ideas** & **Considerations**

Moving past the basics

Extra Support with "**Database Logic**"

Utilizing Metadata on the server

Utilize metadata on the server to help improve performance

Example: cartesian query based on numbers or dates table

- Cartesian query using "baseline" dates table
- Query must process all dates in the dates table to procedure its results
- By storing a "datum" reference, we can restrict the number of rows to process
- Such "datum" reference would be driven by our application logic

Persisted vs. **Dynamic** Results

Choosing to store calculated data

Choosing to store low-risk calculated data can improve performance

Example: pricing-in-effect data

- A Pricing table could have prices that were in effect as of given dates
- Querying "current" values would always get the latest price in effect
- However, such queries add complexity
- We could generate a static table of currently effective prices
- Such table could be regenerated via Stored Procedure
- Nightly scheduled task to call the stored procedure
- Application function to refresh on-demand if required
- Now we have access to the current pricing without having to query it

Database **Denormalization**

Breaking normalization to improve analytic performance

Denormalizing our data, we can cut query complexity significantly

- Generally only suitable only for analytic (OLAP) type of data
- Typically a very poor choice for transactional (OLTP) data
- Can reduce our query client complexity significantly
- Can regenerate the OLAP datasets on schedule or on demand

Using **XML** or **JSON**

Loading data to the server quickly

Loading data into the server quickly and efficiently

- SQL Server has built-in support for processing XML
- More recent versions can also similarly process JSON
- Inserting bulk data into the server is often a pain point (e.g., uploading csv)
- By pushing our client-side data into an XML/JSON format and sending to the server, we can process large amounts of data quickly

Stored Procedures & **Bind Parameters**

Improving Execution Plan Performance

Execution Plans are re-used for queries whose text doesn't change

```
SELECT ThisField, ThatField
FROM ThisTable
WHERE ID = 5;

SELECT ThisField, ThatField
FROM ThisTable
WHERE ID = 15;
```

query text changes, separate execution plan created for each call

```
DECLARE @ID INT

SET @ID = 5
SELECT ThisField, ThatField
FROM ThisTable
WHERE ID = @ID;

SET @ID = 15
SELECT ThisField, ThatField
FROM ThisTable
WHERE ID = @ID;
```

query text unchanged: execution plan re-used for each query

Bind Parameters are good habit: more secure than value injection

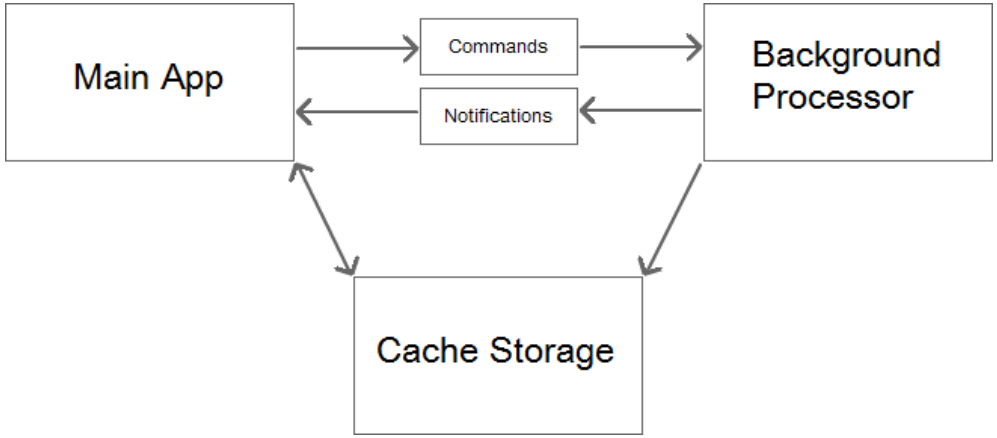
Database **Polling** & **Async** Loading

Letting things happen in the background

Polling the database for changes and loading data in an async manner can dramatically improve perceived performance

Unfortunately, single-threaded nature of Access means we have to be creative

Utilize a sideload data cache for local tables and a separate polling/update utility application to separate the processes



https://dymeng.com/async-processing-in-access/

THANK YOU

Jack D. Leach
jleach@dymeng.com

DYMENG