Set Me Up

How to Think in Sets

Brian Hansen
brian@tf3604.com
@tf3604

PASS
SQLSATURDAY
DENVER | SEP 15 2018

Brian Hansen

brian@tf3604.com

@tf3604.com

children international

children.org

- 20 Years working with SQL Server
  - Started as **developer**, still *trying* to keep up
  - Administration going back to 6.5
  - Fascinated with SQL internals

www.tf3604.com/sets

Agenda

- Why sets?
- Anti-patterns and solutions
- Set-based constructs

Why Sets?

- Math: set theory (Cantor, 1874)
  - Rigorous proofs of set operations
  - Relational model / relational algebra (Codd, 1970)
  - Very stable, still basis for most RDBMS engines
- SQL Server internal operators are optimized for sets
  - However, most code still operates row-by-row
  - Some newer operations run in “batch” mode

RBAR

Can be external or internal

Test Harness (SQL)

```sql
declare @loopNbr int = 0;
while @loopNbr < 5
begin
    declare @TestStartTime datetime2 = sysdatetime();
    -- Execute test
    -- ...
    declare @TestEndTime datetime2 = sysdatetime();

    insert dbo.ExecutionResult (TestName, StartTime, EndTime)
    values (N'Test Name', @TestStartTime, @TestEndTime);

    select @loopNbr += 1;
end
```

Test Harness (SQL) – Results

```sql
with MostRecentTestRuns as
(
     select top 5 xr.ID, xr.TestName, xr.StartTime, xr.EndTime,
        datediff(millisecond, xr.StartTime, xr.EndTime) RunTimeMs
     from dbo.ExecutionResult xr
     where xr.TestName = N'Test Name'
     order by xr.StartTime desc
), Middle3Runs as
(
     select xr.ID, xr.TestName, xr.StartTime, xr.EndTime, xr.RunTimeMs
     from MostRecentTestRuns xr
     order by xr.RunTimeMs
     offset 1 row fetch next 3 rows only
)
select ID, TestName, StartTime, EndTime, RunTimeMs,
     (select avg(RunTimeMs) from Middle3Runs) AvgRunTimeMs
from Middle3Runs;
```

Test Harness (C#)

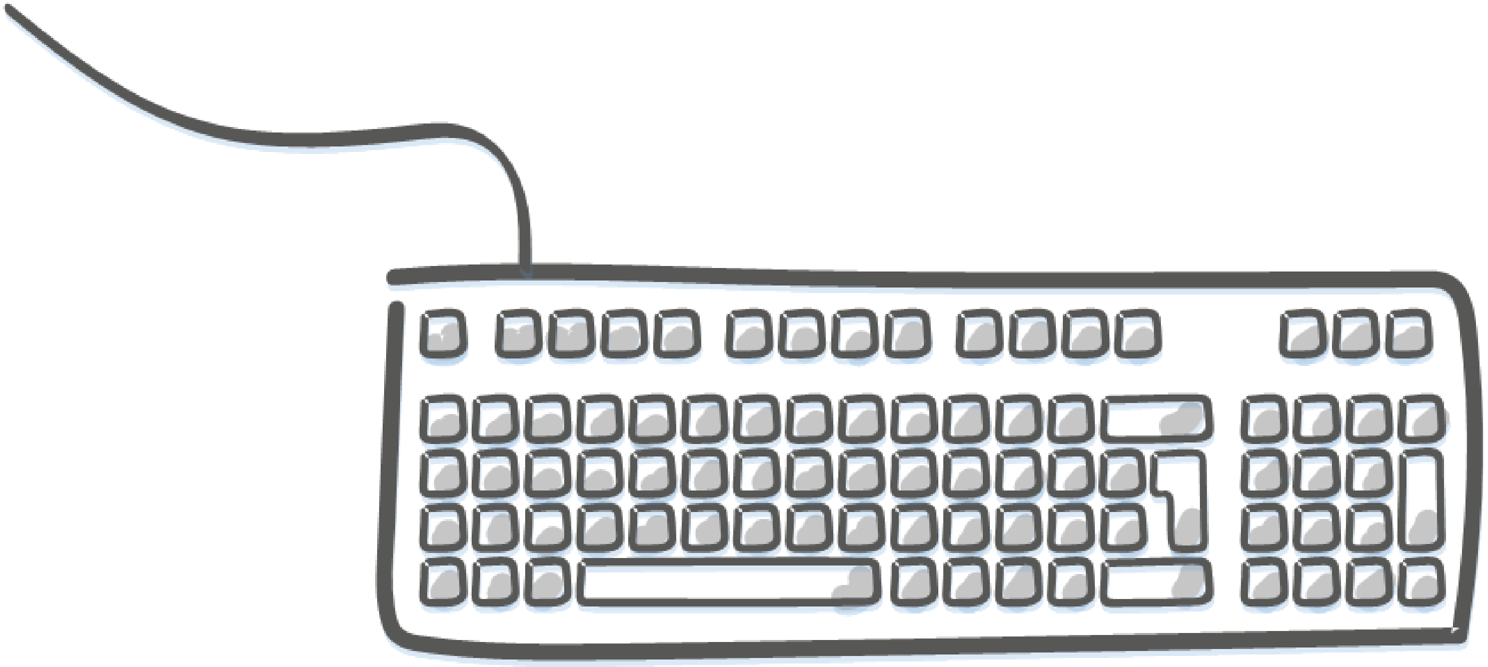
```csharp
List<TimeSpan> executionTimes = new List<TimeSpan>();
for (int executionCounter = 0; executionCounter < 5; executionCounter++)
{
    Stopwatch clock = Stopwatch.StartNew();
    // Execute test
    // ...
    clock.Stop();
    executionTimes.Add(clock.Elapsed);
}
executionTimes.RemoveMinAndMaxValues();
double averageTimeInMilliseconds = executionTimes.Average(t => t.TotalMilliseconds);
```

Cursors and Loops

- Cursors – heavyweight objects
  - Many infrequently used features enabled by default
  - If necessary, declare as `fast_forward read_only`
- WHILE loops
  - More lightweight
  - However, still tend be slow (compared to procedural languages)

Demo

Cursors and Loops



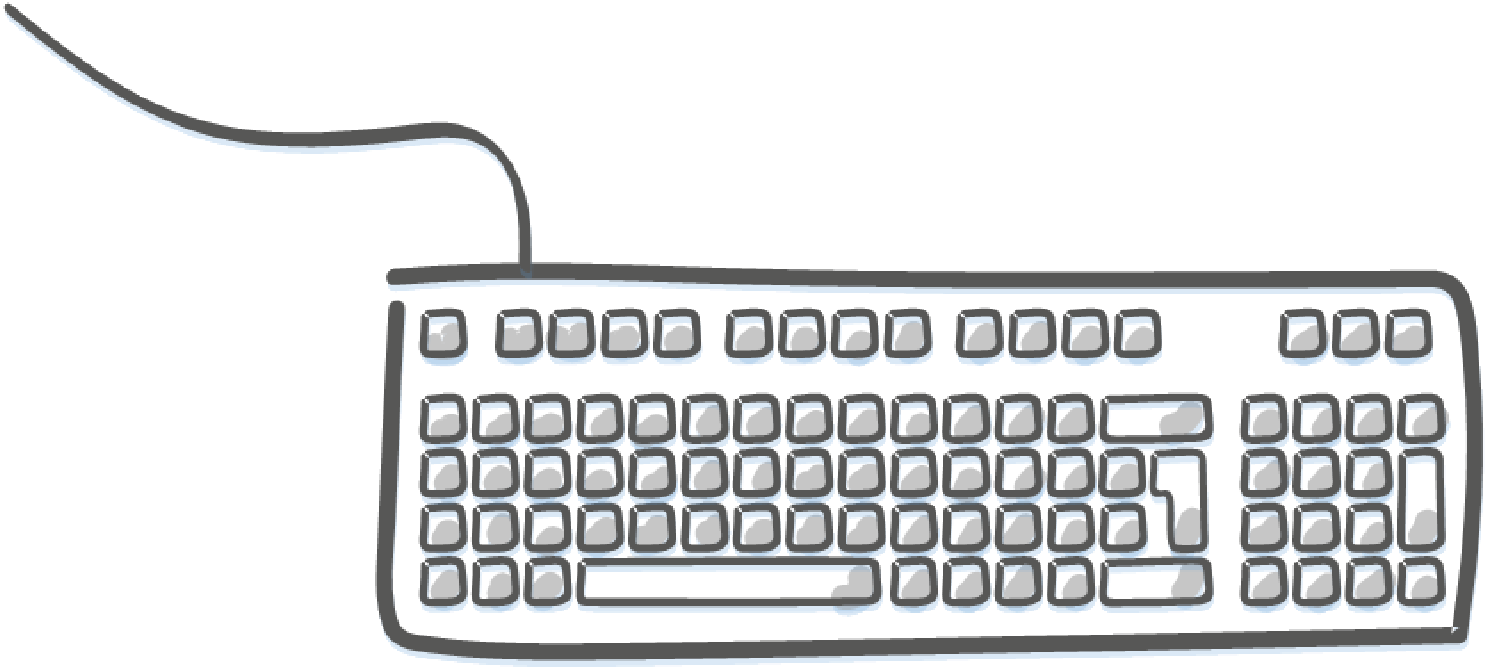
Subqueries

```sql
select oh.OrderId,
    oh.OrderDate,
    oh.CustomerId,
(
    select top 1 od.ProductId
    from dbo.OrderDetail od
    where od.OrderId = oh.OrderId
    order by od.OrderDetailId
) Line1ProductId
from dbo.OrderHeader oh;
```

```sql
select oh.OrderId,
    oh.OrderDate,
    oh.CustomerId
from dbo.OrderHeader oh
where
(
    select top 1 od.ProductId
    from dbo.OrderDetail od
    where od.OrderId = oh.OrderId
    order by od.OrderDetailId
) = 4926;
```

Demo

Subqueries



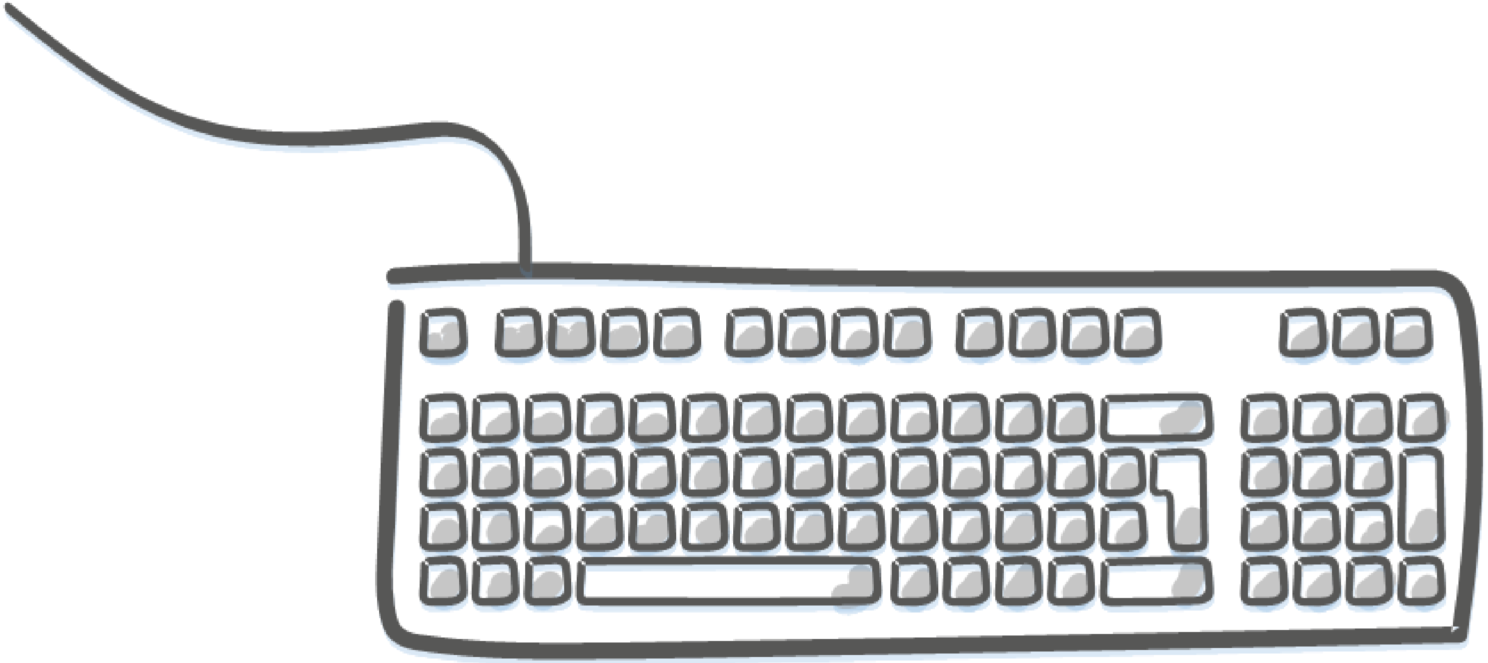
User-Defined Functions (UDFs)

- Scalar
  - Returns single value of any data type
  - Call as `select dbo.ScalarFunc(param1, param2)`
- Multi-Statement Table-Valued *
  - Returns table variable populated by function code
  - Call as `select * from dbo.TableValuedFunc (param1, param2)`
- Inline Table-Valued: single select statement

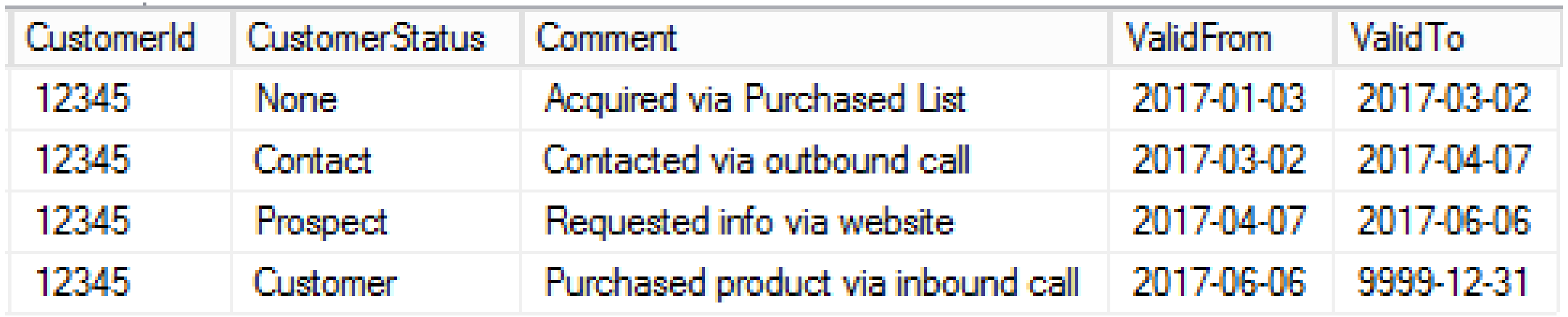
* Improved performance in SQL 2017 under certain conditions (“adaptive join processing”)

Demo

User-Defined Functions



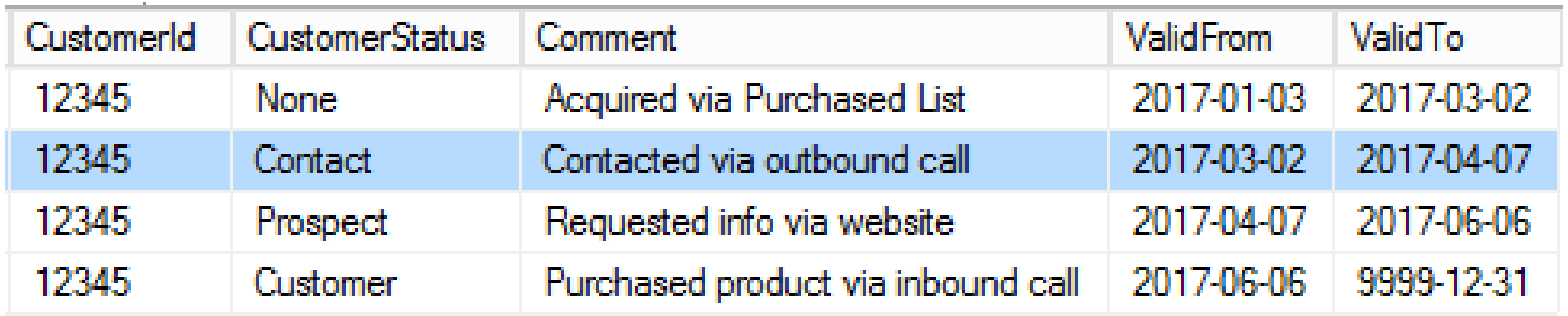
Triangle Joins



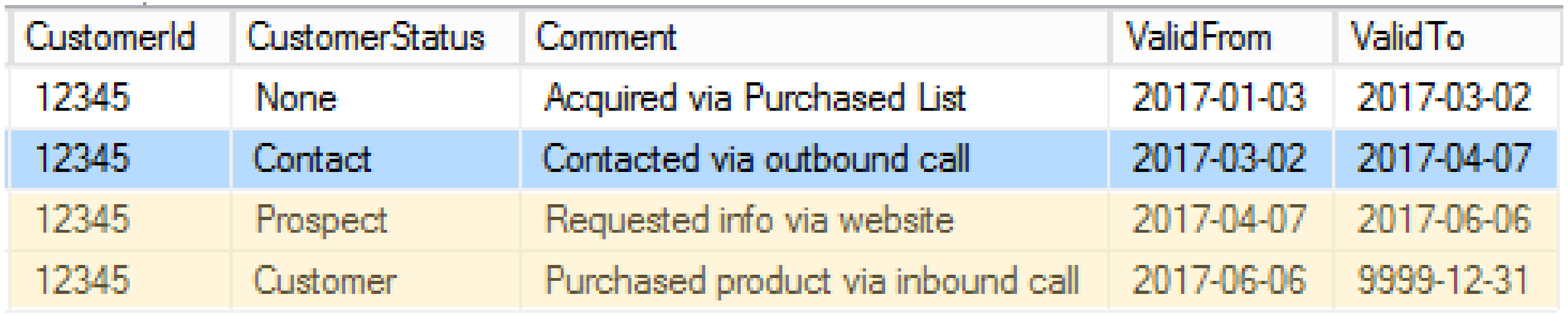
CustomerId	CustomerStatus	Comment	ValidFrom	ValidTo
12345	None	Acquired via Purchased List	2017-01-03	2017-03-02
12345	Contact	Contacted via outbound call	2017-03-02	2017-04-07
12345	Prospect	Requested info via website	2017-04-07	2017-06-06
12345	Customer	Purchased product via inbound call	2017-06-06	9999-12-31

```
select *
from dbo.PersonDim pd
where pd.CustomerStatus = 'Contact'
and
(
    select top 1 pnext.CustomerStatus
    from dbo.PersonDim pnext
    where pnext.CustomerId = pd.CustomerId
    and pnext.ValidFrom > pd.ValidFrom
    order by pnext.ValidFrom
) = 'Prospect';
```

Triangle Joins



CustomerId	CustomerStatus	Comment	ValidFrom	ValidTo
12345	None	Acquired via Purchased List	2017-01-03	2017-03-02
12345	Contact	Contacted via outbound call	2017-03-02	2017-04-07
12345	Prospect	Requested info via website	2017-04-07	2017-06-06
12345	Customer	Purchased product via inbound call	2017-06-06	9999-12-31



CustomerId	CustomerStatus	Comment	ValidFrom	ValidTo
12345	None	Acquired via Purchased List	2017-01-03	2017-03-02
12345	Contact	Contacted via outbound call	2017-03-02	2017-04-07
12345	Prospect	Requested info via website	2017-04-07	2017-06-06
12345	Customer	Purchased product via inbound call	2017-06-06	9999-12-31

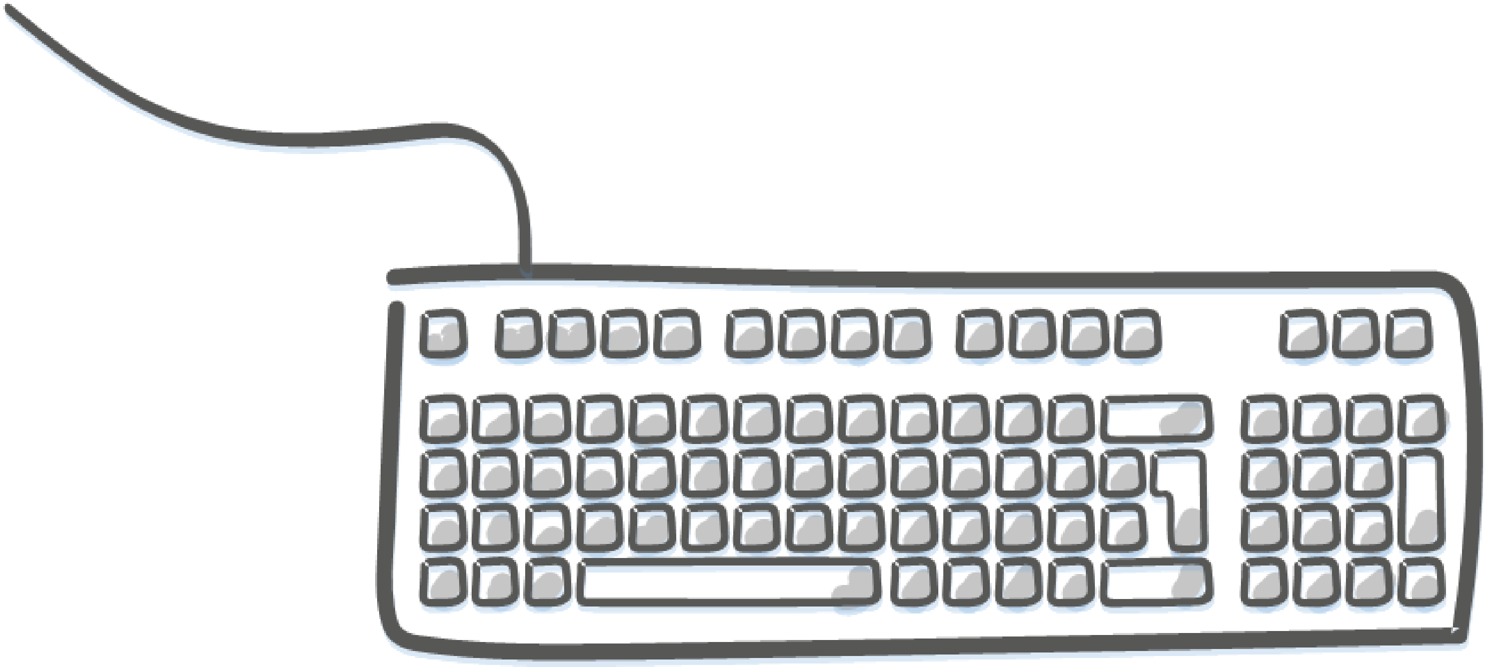
Windowing Functions

- ROW_NUMBER, RANK
- SUM, AVG, …
- LEAD, LAG

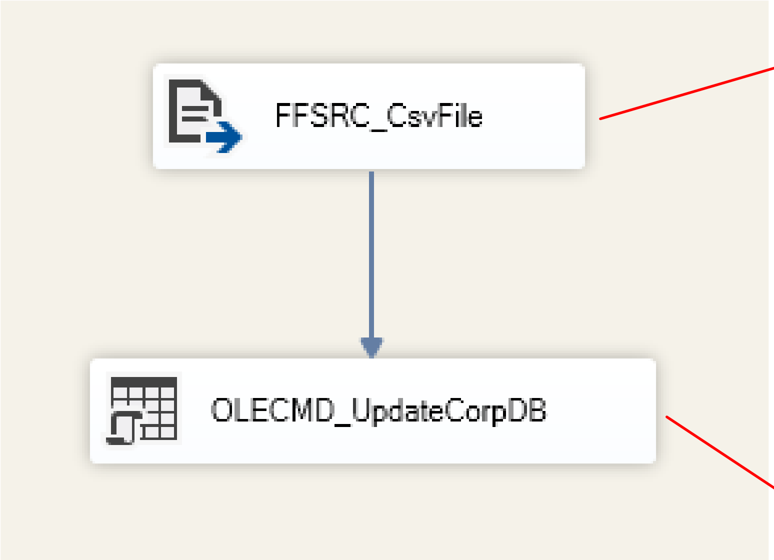
- OVER (partition by tbl.PartitionColumn order by tbl.SortColumn rows …)

Demo

Running Aggregations



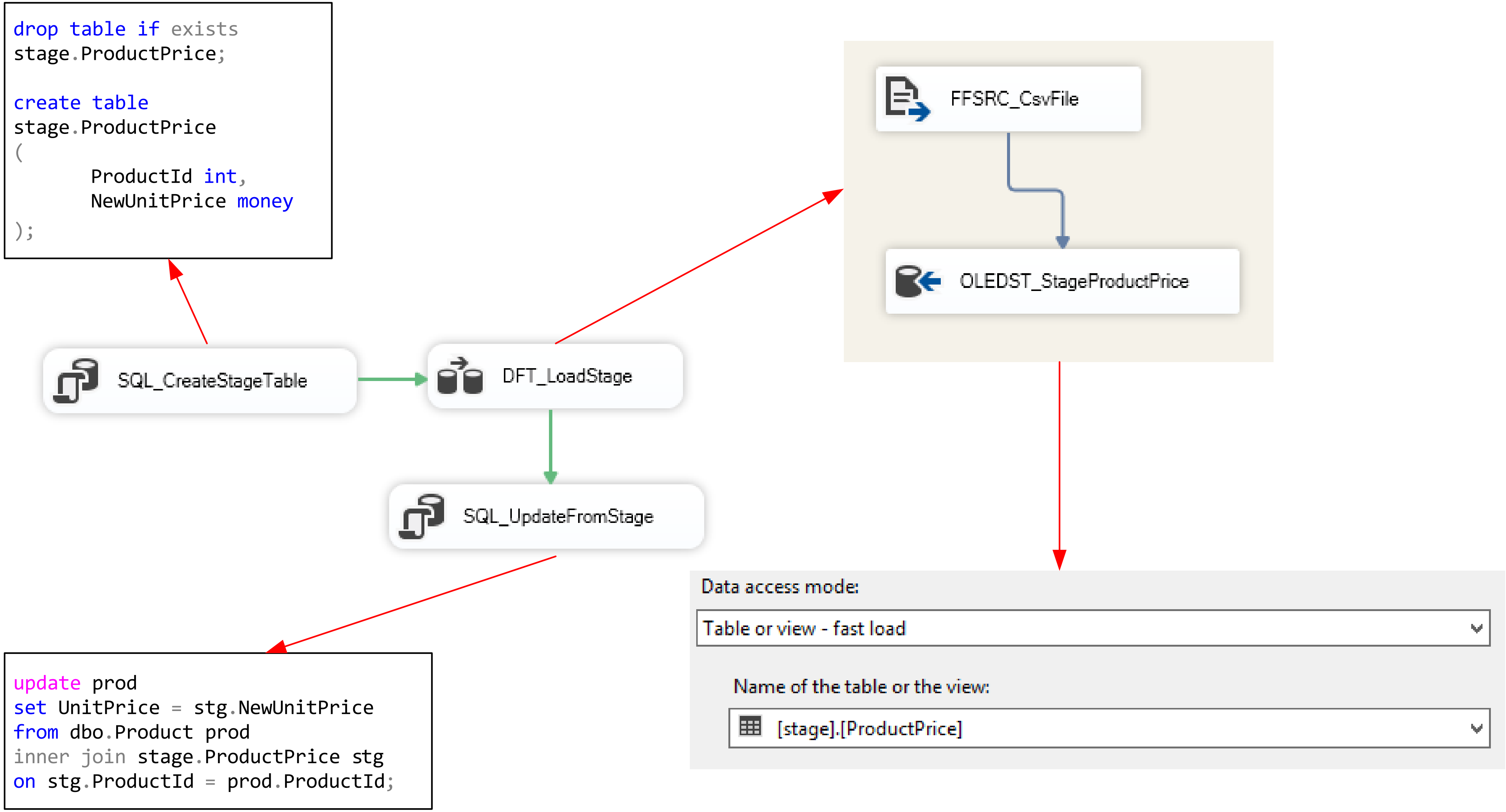
SSIS: Command Component



```
ProductId,NewUnitPrice
1,49.980000
2,2.260000
3,97.720000
4,68.820000
5,64.050000
6,30.610000
```

```sql
update dbo.Product
set UnitPrice = ?
where ProductId = ?;
```

SSIS: Staging Table



```
drop table if exists
stage.ProductPrice;

create table
stage.ProductPrice
(
        ProductId int,
        NewUnitPrice money
);
```

```
update prod
set UnitPrice = stg.NewUnitPrice
from dbo.Product prod
inner join stage.ProductPrice stg
on stg.ProductId = prod.ProductId;
```

Data access mode:

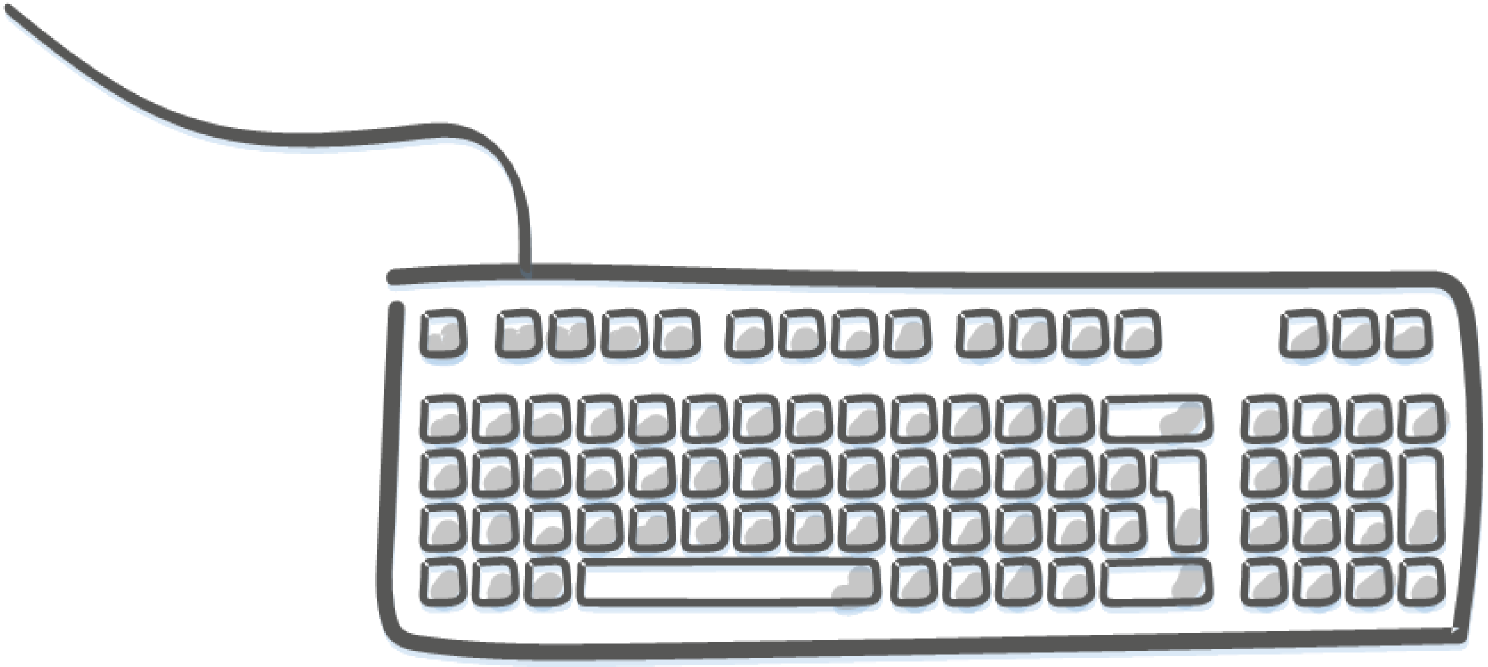
Table or view - fast load

Name of the table or the view:

[stage].[ProductPrice]

Demo

SSIS



C#: Singleton Inserts

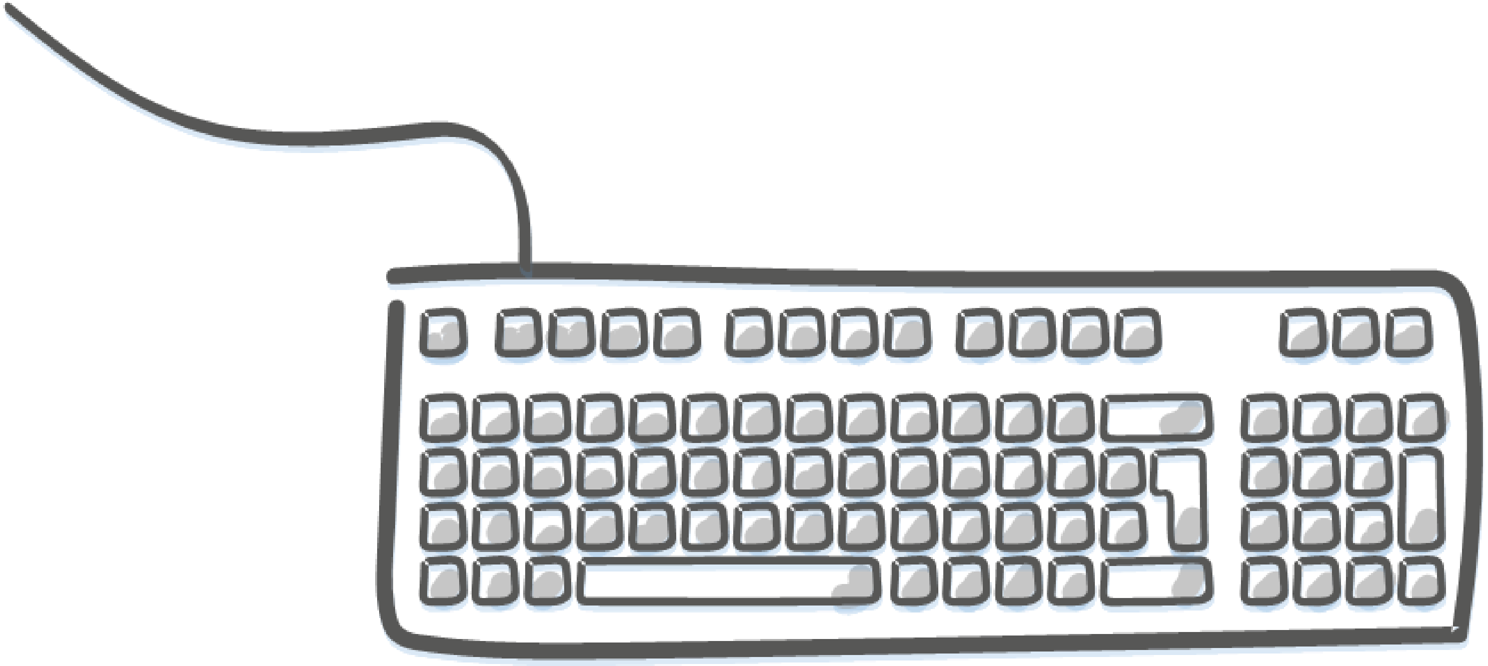
```
sql = "insert stage.DataFile (FilePath, LastWriteTime) values (@FilePath,
@LastWriteTime);";

foreach (FileInfo file in _files)
{
    using (SqlCommand command = new SqlCommand(sql, _connection))
    {
        SqlParameter filePathParameter = new SqlParameter("FilePath", file.FullName);
        command.Parameters.Add(filePathParameter);

        SqlParameter writeTimeParameter =
              new SqlParameter("LastWriteTime", file.LastWriteTime);
        writeTimeParameter.SqlDbType = SqlDbType.DateTime2;
        command.Parameters.Add(writeTimeParameter);

        command.ExecuteNonQuery();
    }
}
```

C#: Bulk Insert

```
using (SqlBulkCopy bulkCopy = new SqlBulkCopy(
    connection,
    SqlBulkCopyOptions.TableLock | SqlBulkCopyOptions.UseInternalTransaction,
    null))
{
    bulkCopy.BulkCopyTimeout = 300;
    bulkCopy.ColumnMappings.Clear();
    bulkCopy.ColumnMappings.Add("FilePath", "FilePath");
    bulkCopy.ColumnMappings.Add("LastWriteTime", "LastWriteTime");
    bulkCopy.DestinationTableName = "stage.DataFile";
    using (DataTable fileTable = CreateFileListDataTable())
    {
        bulkCopy.WriteToServer(fileTable);
    }
}
```

Demo

.NET Code



Thinking in Sets: A 90° Shift

- Think about columns first, then rows
- Use CTEs to help break down processing steps
- Use CASE statements to handle IF … THEN logic
- UDFs are nice for encapsulation …
    - But they can devolve into non-set processing
    - Except for table-valued functions
    - So SQL can involved repeated code

Case Study: Preferred Payment Method

- Legacy Windows app – Customer screen
- Customers have various products they may subscribe to; may have different payment methods
- Customer screen displays a “preferred” payment method
- Developers created scalar user-defined function
- Called once each time the form gets opened

```
create function dbo.fnGetPaymentPreference
(@CustomerId int) returns nvarchar(50)
as …
```

Case Study: Preferred Payment Method

- My task: daily sync of the preferred payment method for ~4 million customers to another system

```
select c.CustomerID,
   dbo.fnGetPaymentPreference
   (c.CustomerID) PreferredPaymentMethod
from dbo.Customer c;
```

- (0.74 ms per customer)
- Runs for 48 min 47 sec.

Case Study: Preferred Payment Method

- Re-write as set-based SQL
- UDF consists of five separate SQL statements to populate variables

Case Study: Preferred Payment Method

```
SELECT @PaymentCount1 = COUNT(Q1.ID)
FROM
    (SELECT MAX(sub.ID) AS ID
    FROM dbo.Subscription sub
    INNER JOIN dbo.PaymentType pt
        ON pt.ID = sub.PaymentTypeId
    WHERE sub.CustomerId = @CustomerId
    AND sub.Status = 'Active'
    AND pt.type = 'Credit Card'
    GROUP BY sub.PaymentTypeID, sub.ccLastFour) AS
Q1
```

Case Study: Preferred Payment Method

```sql
SELECT @PaymentCount2 = COUNT(Q2.ID)
FROM
   (SELECT MAX(sub.ID) AS ID
   FROM dbo.Subscription sub
   INNER JOIN MMS.dbo.PaymentType pt
      ON pt.ID = sub.PaymentTypeId
   WHERE sub.CustomerId = @CustomerId
   AND sub.Status = 'Active'
   AND pt.type <> 'Credit Card'
   GROUP BY sub.PaymentTypeID) AS Q2
```

Case Study: Preferred Payment Method

```
SELECT @PaymentCount3 =
CASE WHEN (@PaymentCount1 IS NULL)
   AND (@PaymentCount2 IS NULL) THEN 0
  WHEN (@PaymentCount1 IS NULL)
      THEN @PaymentCount2
  WHEN (@PaymentCount2 IS NULL)
      THEN @PaymentCount1
  ELSE @PaymentCount1 + @PaymentCount2
END
```

Case Study: Preferred Payment Method

```
SELECT @TotalPaymentCount =
  ISNULL(@CCPaymentCount, 0) +
  ISNULL(@NonCCPaymentCount, 0);
```

Case Study: Preferred Payment Method

```
SELECT @PaymentType = MAX(CASE
      WHEN pt.type = 'Credit Card' THEN 'Credit
Card'
      ELSE pt.name
   END)
FROM dbo.Subscription sub
INNER JOIN dbo.PaymentType pt
   ON pt.ID = sub.PaymentTypeID
WHERE sub.CustomerId = @CustomerId
AND so.Status = 'Active'
GROUP BY sub.CustomerId
```

Case Study: Preferred Payment Method

```
SELECT @PaymentMethod =
CASE WHEN @PaymentCount3 IS NULL THEN
'None'
     WHEN @PaymentCount3 = 0 THEN 'None'
     WHEN @PaymentCount3 = 1 THEN
@PaymentType
     ELSE 'Multiple'
END

RETURN @PaymentMethod
```

Case Study: Preferred Payment Method

```
with CCPaymentCount as
(
    select Q1.CustomerId, COUNT(Q1.ID) Cnt
    FROM
        (SELECT sub.CustomerId, MAX(sub.ID) AS ID
        FROM dbo.Subscription sub
        INNER JOIN dbo.PaymentType pt
            ON pt.ID = sub.PaymentTypeID
        WHERE ~~sub.CustomerId = @CustomerId~~
        ~~AND~~ sub.Status = 'Active'
        AND pt.type = 'Credit Card'
        GROUP BY sub.CustomerId, sub.PaymentTypeID, sub.ccLastFour)
AS Q1
    GROUP BY Q1.CustomerId
)
```

Case Study: Preferred Payment Method

```
, NonCCPaymentCount as
(
    SELECT Q2.CustomerId, COUNT(Q2.ID) Cnt
    FROM
        (SELECT sub.CustomerId, MAX(sub.ID) AS ID
        FROM dbo.Subscription so
        INNER JOIN dbo.PaymentType pt
            ON pt.ID = sub.PaymentTypeID
        WHERE ~~sub.CustomerId = @CustomerId~~
        ~~AND~~ sub.Status = 'Active’
        AND pt.type <> 'Credit Card'
        GROUP BY sub.CustomerId, sub.PaymentTypeID) AS Q2
    GROUP BY Q2.CustomerId
)
```

Case Study: Preferred Payment Method

```
, TotalPaymentCount as
(
    select coalesce(p1.CustomerId, p2.CustomerId) CustomerId,
        isnull(p1.Cnt,  0) + isnull(p2.Cnt, 0) Cnt
    from CCPaymentCount ccCount
    full outer join NonCCPaymentCount nonCcCount
        on nonCcCount.CustomerId = ccCount.CustomerId
)
```

Case Study: Preferred Payment Method

```
, PaymentType as
(
    select sub.CustomerId, MAX(CASE
        WHEN pt.type = 'Credit Card' THEN 'Credit Card'
        ELSE pt.name
        END) TypeName
    FROM dbo.Subscription so
    INNER JOIN dbo.PaymentType pt
        ON pt.ID = sub.PaymentTypeID
    WHERE ~~sub.CustomerId = @CustomerId~~
    ~~AND~~ so.Status = 'Active'
    GROUP BY sub.CustomerId
)
```

Case Study: Preferred Payment Method

```
, FinalResult as
(
    select pc.CustomerId,
        case when pc.Cnt = 1 then pt.TypeName
        else 'Multiple’
        end PaymentType
    from TotalPaymentCount pc
    inner join PaymentType pt
    on pt.CustomerId = pc.CustomerId
)
```

Case Study: Preferred Payment Method

```sql
select c.CustomerId,
    isnull(fr.PaymentType, 'None') PaymentType
from dbo.Customer c
left join FinalResult fr
on c.CustomerId = fr.CustomerId;
```

Case Study: Preferred Payment Type

- Still requires 3 passes through the data, so definitely room for improvements on that front
- However … this rewrite now runs in about 3 seconds (about a 1000x improvement)
- Performance tuning is not always about squeezing every bit out of the query …
- It’s about “good enough”

So if sets are good, really big sets are better, right?

- Transaction log impacts
  - Long-running transactions and clearing the log
  - Log growth
  - Log space reservation
  - What if DB is restored to point in the middle of the operation?
- Splitting up sets is a bit of an art

Other Stuff

- In-Memory OLTP changes things
  - aka Hekaton, new in SQL 2014
  - If natively compiled
  - Loops with data access perform well
  - Beware of limitations

Key Take-Aways

- Cursors are usually inefficient
  - If necessary, declare as `fast_forward read_only`
  - Still necessary for lots of admin functionality
  - Pre-2012, still best way to do running totals, etc.
- Triangle joins are evil

Key Take-Aways

- Avoid most UDFs
  - Scalar and multi-statement TVFs with data access tend to perform poorly
  - CLR with data access tends to perform poorly
- Inline TVFs generally optimize well and tend to perform nicely

Key Take-Aways

- Embrace `row_number()`: It is much more useful than just for counting rows
- Embrace windowing functions
- Embrace `apply`
  - Easy way to improve many scalar UDFs
- May need to split up very large sets

Thank You

This presentation and supporting materials can be found at www.tf3604.com/sets.

- Slide deck
- Scripts
- Sample database

brian@tf3604.com • @tf3604