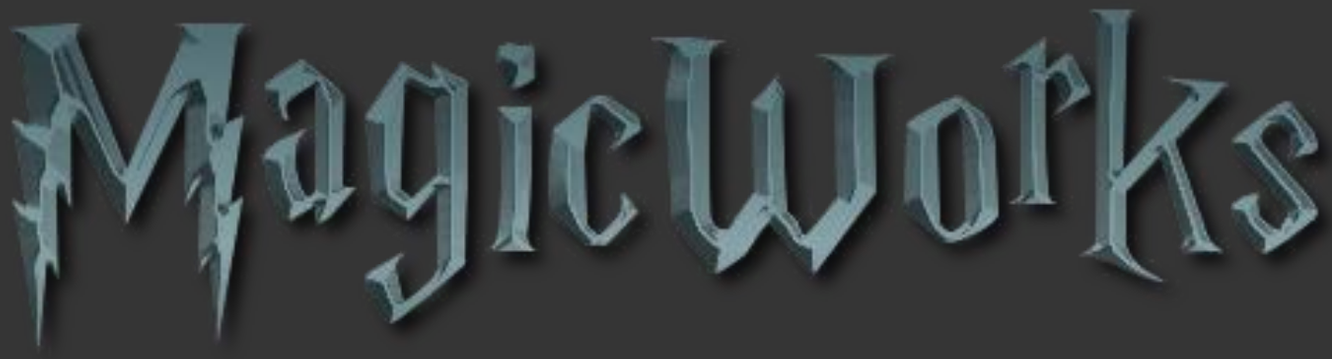
Big Data Vs Pokemon

Analyst Woes



The MagicWorks™ team have their Warehouse loaded but their analysts are having trouble – queries aren't as fast as they hoped!

What affects Query Performance? How can we optimise their workload?

Agenda

Scaling out

Understanding query execution

MPP Operators

Data Movement Operations

Best practices

Scaling out

Scaling out

Massively Parallel Processing (MPP)

Multiple nodes with dedicated CPU, memory, storage

Handles and hides query complexity

Good for scalability

Data is spread (distributed) across servers

Introduces overheads

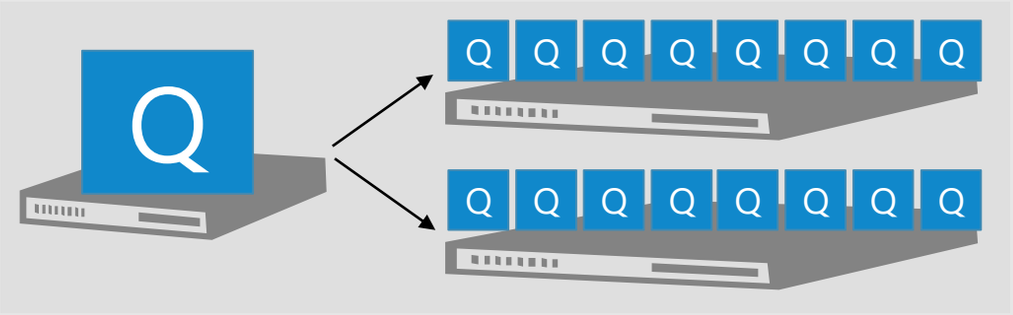
Data is not all in the same place!

Don't know where specific values are located

May need to move the data then process it

SMP Query Execution

MPP Query Execution



Query Execution

Simple example

```
SELECT   COUNT_BIG(*)
FROM     dbo.[FactInternetSales]
```

```
SELECT   SUM(*)
FROM     dbo.[FactInternetSales]
```

Control

Compute

```
SELECT   COUNT_BIG(*)
FROM     dbo.[FactInternetSales]
```

```
SELECT   COUNT_BIG(*)
FROM     dbo.[FactInternetSales]
```

```
SELECT   COUNT_BIG(*)
FROM     dbo.[FactInternetSales]
```

```
SELECT   COUNT_BIG(*)
FROM     dbo.[FactInternetSales]
```

Query Execution

0. Batch (batch, sp)

1. Query submitted

Submit Time

2. Parsed

3. Optimized – MEMO generated, plan chosen, check some perms

Start Time

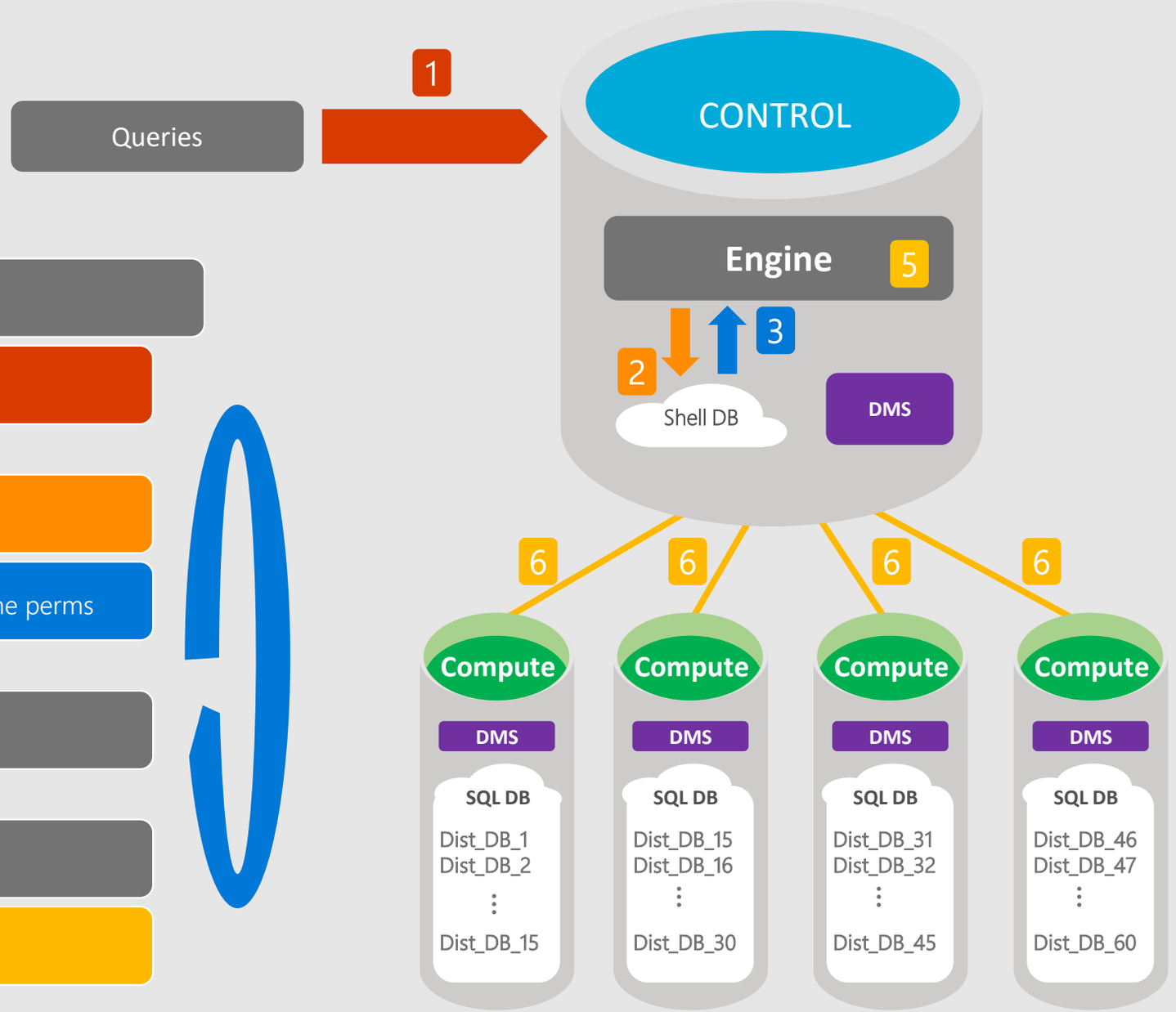
4. Object locks acquired

End Compile Time

5. System locks acquired – wait for concurrency slot

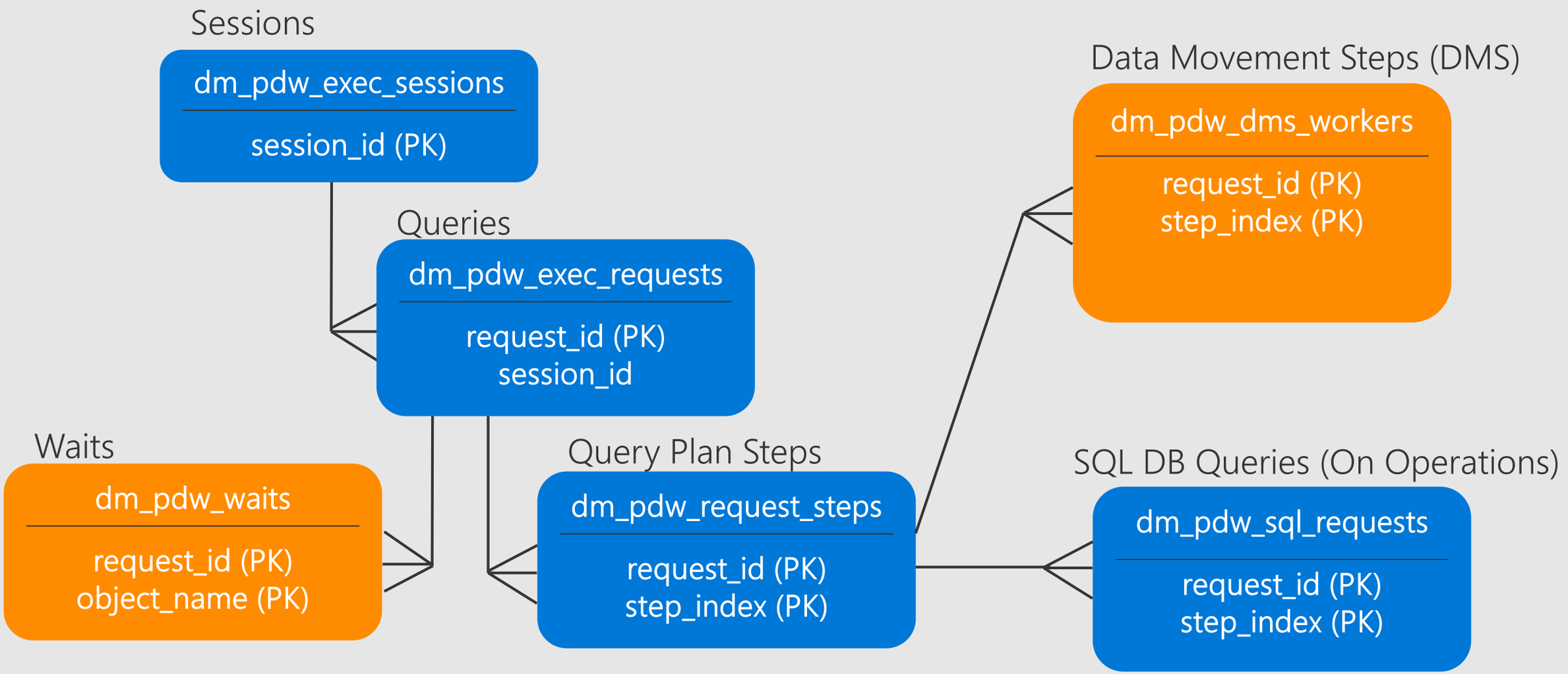
6. Executed

End Time



Execution DMVs

VIEW DATABASE STATE permission
DMVs are in UTC time zone
Last 10,000 queries



EXPLAIN (MPP PLAN)

```
<?xml version="1.0" encoding="utf-8"?>
<dsql_query number_nodes="1" number_distributions="60" number_distributions_per_node="60">
  <sql>SELECT  COUNT_BIG(*) FROM    dbo.[FactInternetSales]</sql>
  <dsql_operations total_cost="0.00192" total_number_operations="4">
    <dsql_operation operation_type="ON">
      <location permanent="false" distribution="Control" />
      <sql_operations>
        <sql_operation type="statement">CREATE TABLE [tempdb].[QTables].[QTable_7cb3c9d5271e41bc9a28e583eeb2bd4c] ([col] BIGINT ) WITH(DATA_COMPRESSION=PAGE);
        </sql_operation>
      </sql_operations>
    </dsql_operation>
    <dsql_operation operation_type="PARTITION_MOVE">
      <operation_cost cost="0.00192" accumulative_cost="0.00192" average_rowsize="8" output_rows="1" />
      <location distribution="AllDistributions" />
      <source_statement>SELECT [T1_1].[col] AS [col]
FROM   (SELECT   COUNT_BIG(CAST ((0) AS INT)) AS [col]
        FROM     (SELECT 0 AS [col]
                  FROM   [JRJDW].[dbo].[FactInternetSales] AS T3_1) AS T2_1
        GROUP BY [T2_1].[col]) AS T1_1</source_statement>
      <destination>Control</destination>
      <destination_table>[QTable_7cb3c9d5271e41bc9a28e583eeb2bd4c]</destination_table>
    </dsql_operation>
    <dsql_operation operation_type="RETURN">
      <location distribution="Control" />
      <select>SELECT [T1_1].[col] AS [col]
FROM   (SELECT ISNULL([T2_1].[col], CONVERT (BIGINT, 0, 0)) AS [col]
        FROM   (SELECT SUM([T3_1].[col]) AS [col]
                FROM   [tempdb].[QTables].[QTable_7cb3c9d5271e41bc9a28e583eeb2bd4c] AS T3_1) AS T2_1) AS T1_1</select>
    </dsql_operation>
    <dsql_operation operation_type="ON">
      <location permanent="false" distribution="Control" />
      <sql_operations>
        <sql_operation type="statement">DROP TABLE [tempdb].[QTables].[QTable_7cb3c9d5271e41bc9a28e583eeb2bd4c]</sql_operation>
      </sql_operations>
    </dsql_operation>
  </dsql_operations>
</dsql_query>
```

Labeling your code

- Labels give visibility to your queries in sys.dm_pdw_exec_requests
- It will help you especially when you are running multiple sp with many queries in it.
- Add this at the end of your queries

```
OPTION (LABEL='sp_UpdateProducts : Step 01 : CTAS dim_Products_stg');
```

```
select *
from sys.dm_pdw_exec_requests
where [label] like 'Exercise%'
```

request_id	session_id	status	submit_time	start_time	end_compile_time	end_time	total_elapsed_time	label	error_id	database_id	command
QID335098	SID66869	Completed	2017-04-21 11:32:56.150	2017-04-21 11:32:56.230	2017-04-21 11:32:56.230	2017-04-21 11:32:57.197	1046	Exercise1 | Fast | Count One Date (1)	NULL	5	SELECT COUNT_BIG(*) FR
QID335102	SID66870	Completed	2017-04-21 11:33:03.447	2017-04-21 11:33:03.837	2017-04-21 11:33:03.837	2017-04-21 11:33:09.917	6468	Exercise2 | Fast | TPCH Query 5 (1)	NULL	5	select n_name, sum(l_e
QID335106	SID66871	Completed	2017-04-21 11:33:14.573	2017-04-21 11:33:14.963	2017-04-21 11:33:14.963	2017-04-21 11:33:18.167	3593	Exercise3 | Fast | TPCH Query 5 (1)	NULL	5	select n_name, sum(l_e

Demo: Performance DMVs & EXPLAIN

Lab 005: Performance DMVs & EXPLAIN

Understanding Query Plans

Understanding Distributed SQL (DSQL) Plan

Fundamental step types

- OnOperation: T-SQL statement running on control or compute nodes
- DMS: Data movement service (a process that runs on nodes)
- Return: query executing on control or compute nodes and returning data

Each step type runs SQL

- For DMS step, part of the execution time may be waiting for SQL

How to get a DSQL plan?

- EXPLAIN
- sys.dm_pdw_exec_requests (limited to 4000 characters)

Data Movement Types for a Query

DMS Operation	Description
ShuffleMoveOperation	Distribution → Hash algorithm → New distribution Changing the distribution column in preparation for join.
PartitionMoveOperation	Distribution → Control Node Aggregations - count(*) is count on nodes, sum of count
BroadcastMoveOperation	Distribution → Copy to all distributions Changes distributed table to replicated table for join.
TrimMoveOperation	Replicated table → Hash algorithm → Distribution When a replicated table needs to become distributed. Needed for outer joins.
MoveOperation	Control Node → Copy to all distributions Data moved from Control Node back to Compute Nodes resulting in a replicated table for further processing.
RoundRobinMoveOperation HadoopRoundRobinMoveOperation	Source → Round robin algorithm → Distribution Redistributes data to Round Robin Table.

Example: Most Optimal Plan

No aggregations

Distribution compatible join

Just a return operation "pass-through"

```
<?xml version="1.0" encoding="utf-8"?>
<dsql_query number_nodes="4" number_distributions="60" number_distributions_per_node="15">
  <sql>select
    top 10 *
from
    lgrc.orders,
    lgrc.lineitem
where
    l_orderkey  =  o_orderkey
    and o_orderdate BETWEEN '1997-01-01' AND '1997-12-31'
order by
    o_orderdate, l_shipdate</sql>
  <dsql_operations total_cost="0" total_number_operations="1">
    <dsql_operation operation_type="RETURN">
      <location distribution="AllDistributions" />
      <select>...</select>
    </dsql_operation>
  </dsql_operations>
</dsql_query>
```

Query

1 step

Optimal Plans tend to have fewer steps

Example: Less Optimal Plan

```
<dsql_operations total_cost="3376410.23822858" total_number_operations="9"> 9 steps
  1 <dsql_operation operation_type="RND_ID">...</dsql_operation>
  2 <dsql_operation operation_type="ON">...</dsql_operation>
  3 <dsql_operation operation_type="SHUFFLE_MOVE">
      <operation_cost cost="3275932.83610284" accumulative_cost="3275932.83610284"
      <source_statement>...</source_statement>
      <destination_table>[TEMP_ID_287]</destination_table>
      <shuffle_columns>l_orderkey;</shuffle_columns>
    </dsql_operation>
  4 <dsql_operation operation_type="RND_ID">...</dsql_operation>
  5 <dsql_operation operation_type="ON">...</dsql_operation>
  6 <dsql_operation operation_type="SHUFFLE_MOVE">
      <operation_cost cost="100477.402125744" accumulative_cost="3376410.23822858"
      <source_statement>...</source_statement>
      <destination_table>[TEMP_ID_288]</destination_table>
      <shuffle_columns>o_orderkey;</shuffle_columns>
    </dsql_operation>
  7 <dsql_operation operation_type="RETURN">
      <location distribution="AllDistributions" />
      <select>...</select>
    </dsql_operation>
  8 <dsql_operation operation_type="ON">...</dsql_operation>
  9 <dsql_operation operation_type="ON">...</dsql_operation>
</dsql_operations>
```

Important steps

- 2 shuffles
- 1 return

Overhead steps

- Rnd_Id
- Creates/Drops

Same query as last example except now both underlying tables are round robin

Questions to ask when looking at DSQL plan

What is taking the longest?

- What is this step doing?
- Does it make sense? (This takes some time to learn.)

What objects is the query using?

- Tables
  - Statistics?
  - Distribution? Hash or Round Robin?
  - How many rows? Skew?
  - Partitioned table?
- External Tables
  - No push down, consider loading
- Views
  - What's under the view?
  - Overloaded with joins?

The plan behind the plan

Within each DSQL step, SQL has its own plan

- Use DBCC PDW_SHOWEXECUTIONPLAN (distribution_id, spid) to return plans
- Can only return plans when query is running on node
- Returns the estimated plan from underlying SQL server plan cache
- One plan per distribution, distributions can have different plans
- If execution time varies widely on one distribution, look at skew or SQL plan
- This is if you need it, but not usually needed

Demo: Optimising Queries

Designing Queries

Top Tips

Avoid using SELECT * in your queries

- It's a COLUMNSTORE folks ☺

Join on distribution keys

- Equi-join (=)
- Use the same data type on both columns

Avoid returning massive result sets to the client

- Use WHERE clause to filter
- Use searchable filters to improve performance
- Export with CETAS if you need to

Joining HASH tables

Store_Sales HASH([ProductKey])
[ProductKey] **INT** NULL

Web_Sales HASH([ProductKey])
[ProductKey] **INT** NULL

Joining HASH tables

Store_Sales HASH([ProductKey])
[ProductKey] **INT** NULL

Web_Sales HASH([ProductKey])
[ProductKey] **BIGINT** NULL

Data movement

Incompatible join

Incompatible aggregation

Forced movement

Query semantics

Aggregation - compatible

Resolved completely on each compute node

No Data Movement when

1. Hash distribution key is contained in the group by keys
2. Count distinct on distribution key

Examples of aggregation compatibility

```
-- FactOnlineSales distributed on ProductKey
SELECT        COUNT_BIG(*)
FROM          [cso].[FactOnlineSales]
GROUP BY      [ProductKey]
;
SELECT  COUNT_BIG(DISTINCT ([ProductKey]))
FROM    [cso].[FactOnlineSales]
;
```

Aggregation - Incompatible

Partially aggregated on each node

Shuffle move co-locates rows with same group by key

1. Table is round robin distributed
2. Hash distribution key is not contained in group by keys
3. Count distinct on non-distribution key or on round robin table

Examples of aggregation incompatibility

```
-- FactOnlineSales distributed on ProductKey
SELECT        COUNT_BIG(*)
FROM          [cso].[FactOnlineSales]
GROUP BY      [StoreKey]
;
SELECT  COUNT_BIG(DISTINCT [DateKey])
FROM    [cso].[FactOnlineSales]
;
```

Forced data movement

You can move:

From hash to round_robin and vice versa

From hash (a) to hash (b)

From hash to replicated and vice versa*

From round_robin to replicated and vice versa*

* APS only today

Re-distribution example

```
--EXPLAIN
CREATE TABLE [tmp].[DimEmployee]
WITH (DISTRIBUTION = Hash(EmployeeKey))
AS
SELECT *
FROM   [cso].[DimEmployee]
OPTION (LABEL = 'CTAS : Redistribution')
;
```

Query Syntax

Causes of data movement:

Expressions on the distribution key

Additional causes of data movement:

OVER()

COUNT(DISTINCT [col])

…these can be optimised…

COUNT DISTINCT examples

```
--EXPLAIN
SELECT  COUNT_BIG(DISTINCT [DateKey])
FROM    [cso].[FactOnlineSales]
OPTION (LABEL = 'COUNT DISTINCT incompatible dist key')
;
--EXPLAIN
SELECT  COUNT_BIG(DISTINCT ([ProductKey]))
FROM    [cso].[FactOnlineSales]
OPTION (LABEL = 'COUNT DISTINCT compatible dist key')
;
```

OVER() examples

```
--EXPLAIN
SELECT  SUM([SalesAmount]) OVER(PARTITION BY [DateKey])
FROM    [cso].[FactOnlineSales]
OPTION (LABEL = 'OVER() incompatible dist key')
;
--EXPLAIN
SELECT  SUM([SalesAmount]) OVER(ORDER BY [ProductKey])
FROM    [cso].[FactOnlineSales]
OPTION (LABEL = 'OVER() incompatible no partition key')
;
--EXPLAIN
SELECT  SUM([SalesAmount]) OVER(PARTITION BY [ProductKey])
FROM    [cso].[FactOnlineSales]
OPTION (LABEL = 'OVER() compatible dist key')
;
```

Demo: Minimizing Data Movement

Lab 006: Minimizing Data Movement

Improving performance

Common query issues

Statistics!!!

Distribution compatibility

Skew at query time or in source tables

Overusing round robin tables

Overusing smallrc

Non-deterministic functions

- WHERE Date = Getdate() -– materialize the column beforehand

Poor quality CCI

- Not periodically maintaining your indexes

Optimizing with Statistics – A MUST

Statistics are NOT automatic

- Cost Based Query Optimizer needs statistics
- Create statistics for all columns used in JOINs, GROUP BY, WHERE

Multicolumn statistics can also help performance

- multi-column joins

You can use filtered statistics

- partitioned tables is a good example

Performance degradation

- Sudden performance degradation
- Slow over time degradation

Impact on

- Data movements
- Distributed plan queries

Performance Impact of Skew

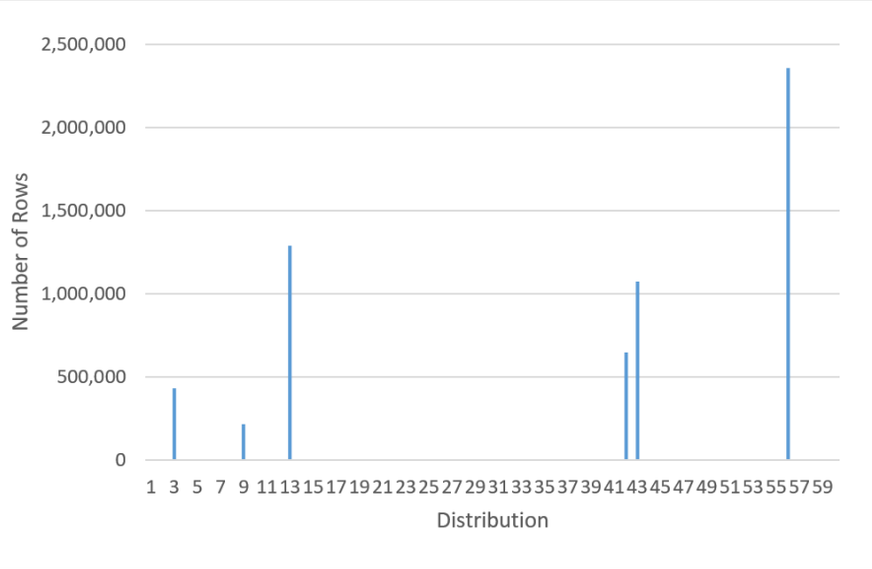
Some distributions finish very quickly, others take disproportionately longer

Use CTAS to change distribution key or distribution type

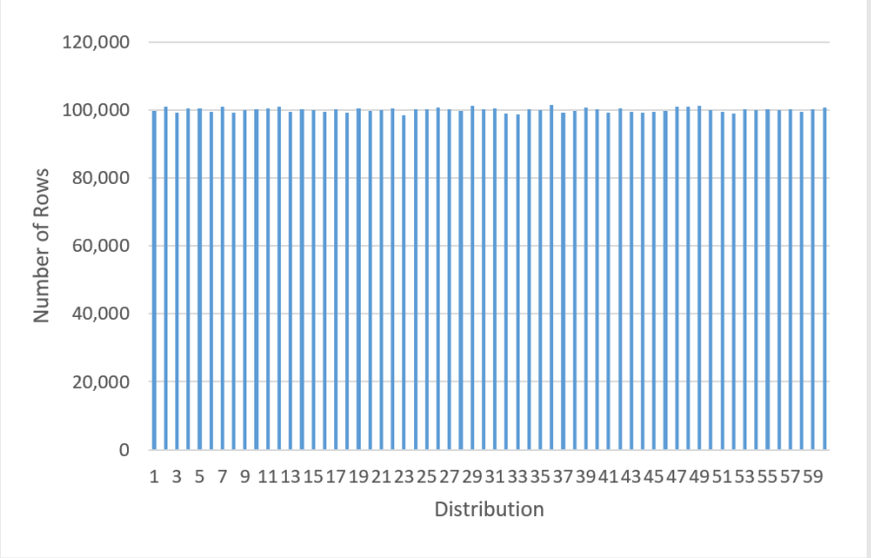
DBCC PDWSHOWSPACEUSED(<Table_Name>)

Generally 7-10% skew is the max acceptable

Skewed Data



Evenly Distributed Data



Concurrency Slots

Concurrency slots used depends on DWU and Resource Class

Allocation and consumption of concurrency slots

	DW100	DW200	DW300	DW400	DW500	DW600	DW1000	DW1200	DW1500	DW2000	DW3000	DW6000
Allocation												
Max Concurrent Queries	32	32	32	32	32	32	32	32	32	32	32	32
Max Concurrency Slots	4	8	12	16	20	24	40	48	60	80	120	240
Slot Consumption												
smallrc	1	1	1	1	1	1	1	1	1	1	1	1
mediumrc	1	2	2	4	4	4	8	8	8	16	16	32
largerc	2	4	4	8	8	8	16	16	16	32	32	64
xlargerc	4	8	8	16	16	16	32	32	32	64	64	128

Demo: Activity Monitor

Summary

Best Practices

Create, update and maintain statistics

- Statistics must be manually created and maintained
- Create “sampled” statistics on all columns in join, group by and where
- Add multi-column statistics where join on multiple columns or predicates

Hash distribute large tables

- Selecting the right distribution columns will minimize data movement

Use resources classes thoughtfully

- Balance need for memory with need for concurrency
- Not all queries benefit

Load large external tables rather than query

- All data is brought back, no push down

Best Practices (cont)

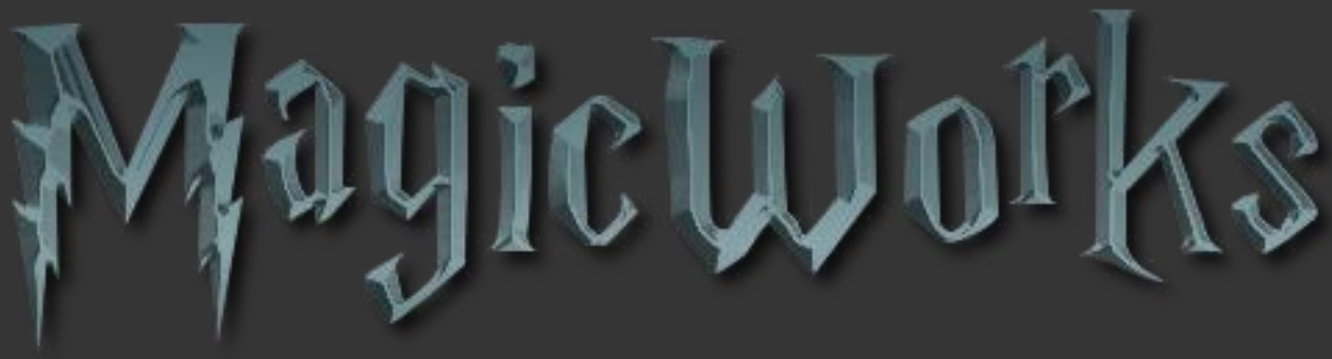
Use query labels in your code

- Helps troubleshooting
- Easy monitoring

Learn how to read a DSQL query plan

- Minimize data movement operations
  - Distribution & aggregation compatible
- Minimize size of data movement
- Use higher resource class for memory intensive queries

Recommendations



Redesign reporting tables with desired workload in mind!

Use Labels & DMVs to track inefficient queries

Monitor & Improve over Time