

SQL Server Execution Plan Primer

DevSpace Conference
11 October 2019

Allison Benneth

Allison@sqltran.org

@sqltran

www.sqltran.org

DevSpace would like to thank our sponsors



Agenda

- ▶ Why do we care about execution plans?
- ▶ What are the inputs to the optimizer?
- ▶ How does the optimizer generate a plan?
- ▶ What types of plans are there?
- ▶ What operators do we see in execution plans?
- ▶ What are some useful ways to execute a plan?

Why Care about Execution Plans?

- ▶ SQL is a declarative language
 - ▶ We are telling the server **WHAT** we want, not how to answer the question
- ▶ The execution plan tells us **HOW** SQL Server is resolving the query
- ▶ Can be very useful to identify performance issues

Why Care about Execution Plans?

- ▶ Execution plans provide front-line insight into decisions made by the optimizer
 - ▶ Order in which tables are accessed
 - ▶ What indexes are used
 - ▶ How much data is expected
 - ▶ “Hidden” internal operations

Why Optimize?

```
select per.FirstName,  
       per.LastName,  
       soh.ShipDate,  
       soh.TerritoryID  
from Person.Person per  
join Sales.Customer cust on per.BusinessEntityID = cust.CustomerID  
join Sales.SalesOrderHeader soh on cust.CustomerID = soh.CustomerID  
join Sales.SalesOrderDetail sod on sod.SalesOrderID = soh.SalesOrderID  
join Production.Product prod on sod.ProductID = prod.ProductID  
where prod.Name = 'Mountain Bottle Cage'  
and soh.ShipDate >= '2014-04-01'  
and soh.TerritoryID in (7, 8);
```

Inputs to Optimization

- ▶ The query text
- ▶ Physical specs of system (memory, cores, etc.)
- ▶ SET options in effect
- ▶ Cardinality estimates
- ▶ DB properties of referenced objects (data types, nullability, check constraints, foreign keys, uniqueness, etc.)
- ▶ Plan cache (optimizer bypass)

Items that are NOT optimizer inputs

- ▶ Has the data already been loaded into memory?
 - ▶ Cold cache is assumed
- ▶ Type of I/O subsystem
 - ▶ Spinning disk vs. SSD

Cardinality Estimation

- ▶ How many rows will this part of the query generate?
- ▶ SQL Server will always generate an estimate
- ▶ May be based on statistics or just a guess (heuristics)
- ▶ Two primary versions of estimator
 - ▶ SQL Server 7
 - ▶ Server Server 2014
 - ▶ (But each later version of SQL has its own CE)
- ▶ Version used based on compatibility level, DB settings, trace flags, query hints

Statistics

- ▶ SomeTable has 1,000,000 rows
- ▶ There is an index on SomeColumn
- ▶ How many rows will the query generate?

```
select ID, SomeColumn, Description  
from dbo.SomeTable  
where SomeColumn = 123456;
```

Selectivity

- ▶ It depends on how selective SomeColumn is
- ▶ Maybe every row has 123456
 - ▶ Low selectivity
- ▶ Or maybe every row is unique
 - ▶ High selectivity
- ▶ Or somewhere in-between

Selectivity

- ▶ A high-level measure of selectivity is “density”

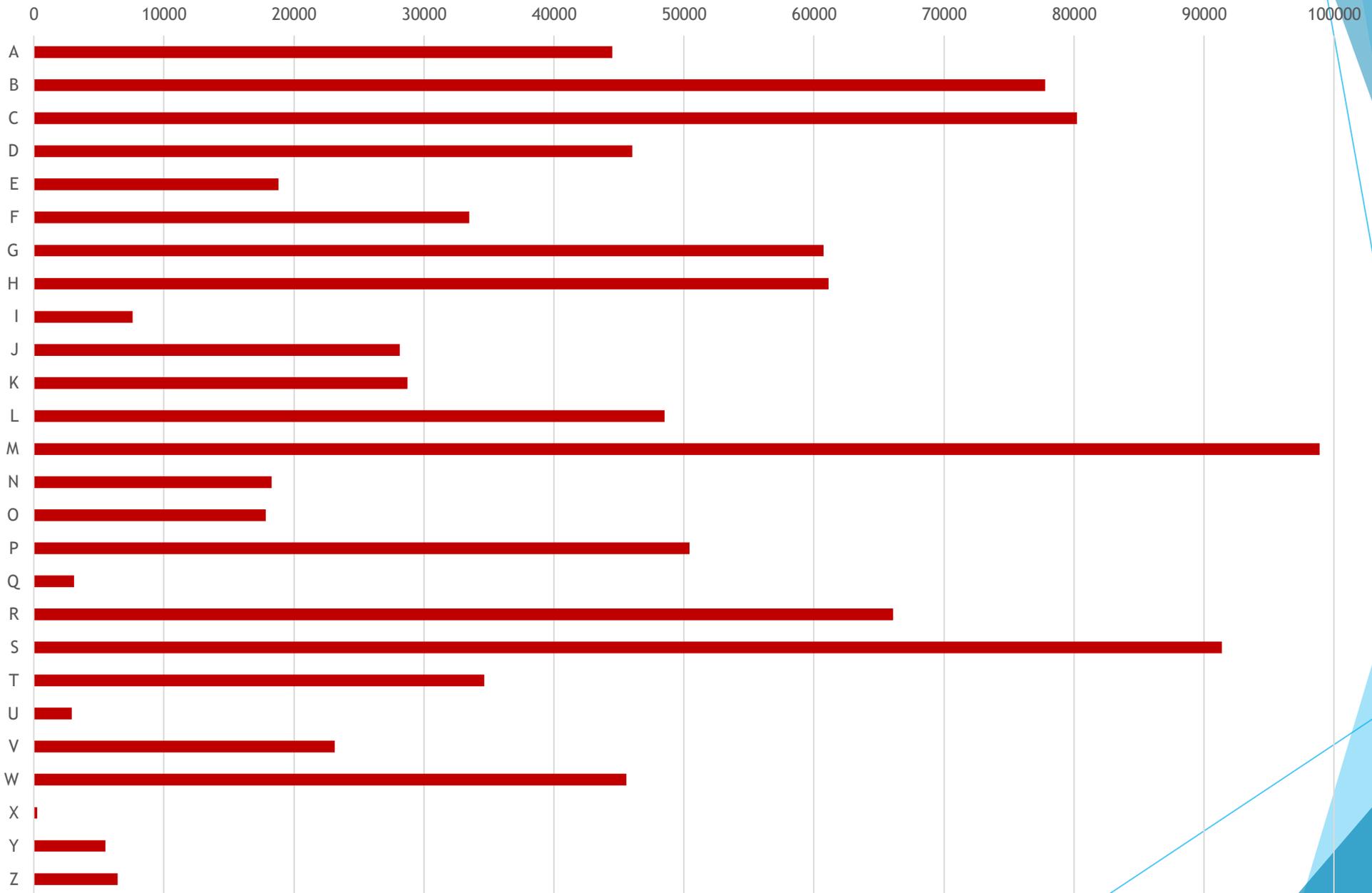
$$\frac{1}{\text{Number of distinct values}}$$

- ▶ If every row is 123456
 - ▶ Density = 1
- ▶ If every row is unique
 - ▶ Density = 0.000001 (1/1,000,000)

Let's get more specific

```
select c.ID, c.FirstName, c.LastName, c.State
from dbo.Customer c
where c.LastName like 'B%';
```

```
select c.ID, c.FirstName, c.LastName, c.State
from dbo.Customer c
where c.LastName like 'Q%';
```



An equality query

```
select c.ID, c.FirstName, c.LastName, c.State  
from dbo.Customer c  
where c.LastName = 'Baker';
```

Name	Updated	Rows	Rows Sampled	Steps
idx_PersonSample_LastName	Feb 1 2019 10:25AM	1000000	1000000	200

All density	Average Length	Columns
4.892512E-06	13.34736	LastName
1E-06	17.34736	LastName, ID

RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
Ayala	5736	572	1711	3.352426
Baker	4287	1193	1198	3.578464
Bames	5963	597	1655	3.603021
Barrett	1633	327	204	8.004902

► Baker = 1193 rows

RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
Ayala	5736	572	1711	3.352426
Baker	4287	1193	1198	3.578464
Barnes	5963	597	1655	3.603021
Barnett	1633	327	204	8.004902

▶ How about this one?

```
select c.ID, c.FirstName, c.LastName, c.State  
from dbo.Customer c  
where c.LastName = 'Baldwin';
```

- ▶ Between Baker & Barnes: average key has 3.603 rows
- ▶ Estimate is 3.603 rows (actual is 210 rows)
- ▶ But 'Banjo' will also be estimated as 3.603 rows (actual = 1)

Statistics: Key Points

- ▶ Based on contents of the index at **some past time**
- ▶ Maximum of 200 steps
- ▶ Becomes a key input to the cardinality estimator
- ▶ Update frequency based on how many rows in the table have been modified
 - ▶ Through SQL 2014: 20% of rows
 - ▶ After SQL 2016: Default is more aggressive updating
 - ▶ DBA jobs to update ([Ola Hallengren maintenance solution](#))

Types of Execution Plans

- ▶ Text
- ▶ XML
- ▶ Graphical

```
select c.State, sum(od.Quantity * od.UnitPrice) as
    OrderAmount
from dbo.OrderHeader oh
join dbo.OrderDetail od on od.OrderId = oh.OrderId
join dbo.Customer c on c.CustomerID =
oh.CustomerId
where od.ProductId >= 760 and od.ProductId <= 792
group by c.State;
```

Types of Execution Plans - Text (Deprecated)

```
set showplan_text on; (less detail)
```

```
set showplan_all on; (more detail)
```

StmtText

```
-----  
|--Hash Match(Aggregate, HASH:([c].[State]), RESIDUAL:([CorpDB].[dbo].[Customer].[State] as [c].[St  
|--Merge Join(Inner Join, MERGE:([c].[CustomerId])=([oh].[CustomerId]), RESIDUAL:([CorpDB].[dbo  
|--Clustered Index Scan(OBJECT:([CorpDB].[dbo].[Customer].[PK__Customer__A4AE64B8B53AF27A]  
|--Sort(ORDER BY:([oh].[CustomerId] ASC))  
|--Merge Join(Inner Join, MERGE:([oh].[OrderId])=([od].[OrderId]), RESIDUAL:([CorpDB]  
|--Clustered Index Scan(OBJECT:([CorpDB].[dbo].[OrderHeader].[PK__OrderHea__C390  
|--Sort(ORDER BY:([od].[OrderId] ASC))  
|--Compute Scalar(DEFINE:([Expr1004]=CONVERT_IMPLICIT(money, [CorpDB].[dbo].  
|--Clustered Index Scan(OBJECT:([CorpDB].[dbo].[OrderDetail].[PK__Orde
```

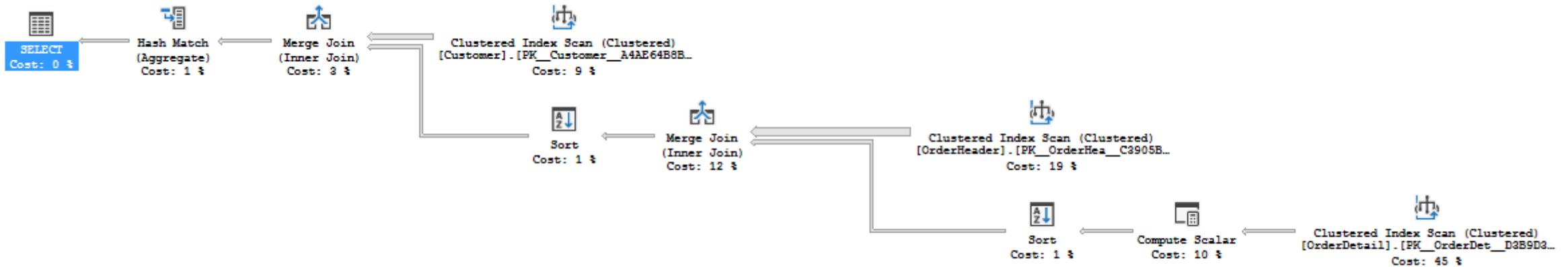
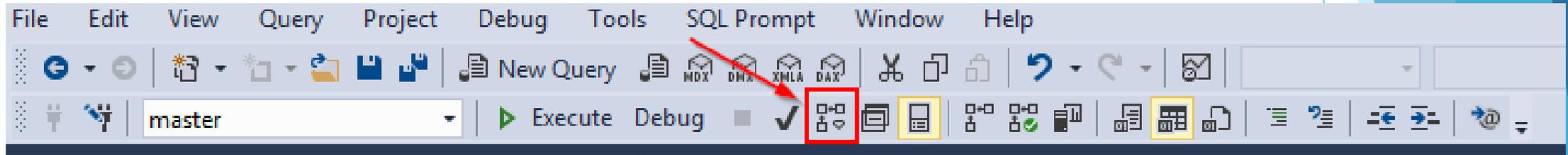
Types of Execution Plans - XML

```
set showplan_xml on;
```

```
<ShowPlanXML xmlns="http://schemas.microsoft.com/sqlserver/2004/07/showplan" Version="1.5" Build
StatementText="select c.State, sum(od.Quantity * od.UnitPrice) OrderAmount&#xd;&#xa;from dbo
oh.OrderId&#xd;&#xa;join dbo.Customer c on c.CustomerID = oh.CustomerId&#xd;&#xa;where od.Pr
c.State&#xd;&#xa;option (maxdop 1)" StatementId="1" StatementCompId="1" StatementType="SELEC
StatementEstRows="50.6341" SecurityPolicyApplied="false" StatementOptmLevel="FULL" QueryHash
CardinalityEstimationModelVersion="130"><StatementSetOptions QUOTED_IDENTIFIER="true" ARITHA
ANSI_WARNINGS="true" NUMERIC_ROUNDABORT="false"></StatementSetOptions><QueryPlan NonParallelf
CompileMemory="592"><MissingIndexes><MissingIndexGroup Impact="44.243"><MissingIndex Database
Usage="INEQUALITY"><Column Name="[ProductId]" ColumnId="3"></Column></ColumnGroup><ColumnGrou
Name="[Quantity]" ColumnId="4"></Column><Column Name="[UnitPrice]" ColumnId="5"></Column></C
SerialRequiredMemory="2048" SerialDesiredMemory="2576"></MemoryGrantInfo><OptimizerHardwareD
EstimatedPagesCached="209699" EstimatedAvailableDegreeOfParallelism="1" MaxCompileMemory="676
Match" LogicalOp="Aggregate" EstimateRows="50.6341" EstimateIO="0" EstimateCPU="0.033272" Avg
EstimateRebinds="0" EstimateRewinds="0" EstimatedExecutionMode="Row"><OutputList><ColumnRefer
Column="State"></ColumnReference><ColumnReference Column="Expr1003"></ColumnReference></Output
MemoryFractions><Hash><DefinedValues><DefinedValue><ColumnReference Column="Expr1003"></Column
Distinct="0" AggType="SUM"><ScalarOperator><Identifier><ColumnReference Column="Expr1004"></C
DefinedValue></DefinedValues><HashKeysBuild><ColumnReference Database="[CorpDB]" Schema="[dbo]
HashKeysBuild><BuildResidual><ScalarOperator ScalarString="[CorpDB].[dbo].[Customer].[State]
CompareOp="IS"><ScalarOperator><Identifier><ColumnReference Database="[CorpDB]" Schema="[dbo]
Identifier></ScalarOperator><ScalarOperator><Identifier><ColumnReference Database="[CorpDB]"
ColumnReference></Identifier></ScalarOperator></Compare></ScalarOperator></BuildResidual><Rel
```

Types of Execution Plans - Graphical

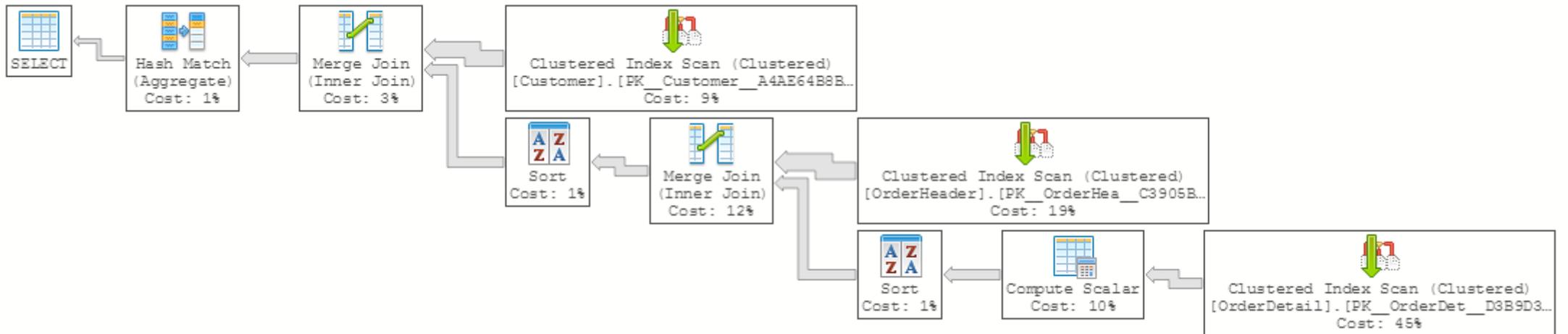
► Display Estimated Execution Plan (Ctrl-L)



Types of Execution Plans - Graphical

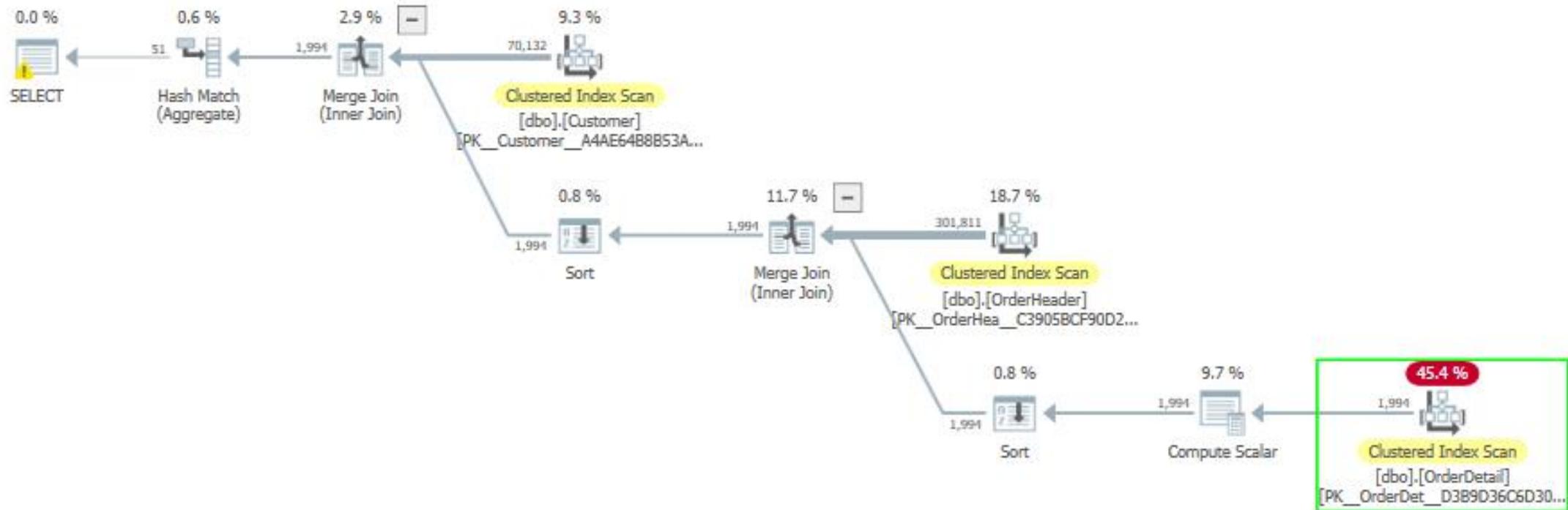
► Azure Data Studio

► Run Cancel Disconnect Change Connection | master  



Types of Execution Plans - Graphical

- ▶ Alternate way to view graphical plans ([SentryOne Plan Explorer](#))

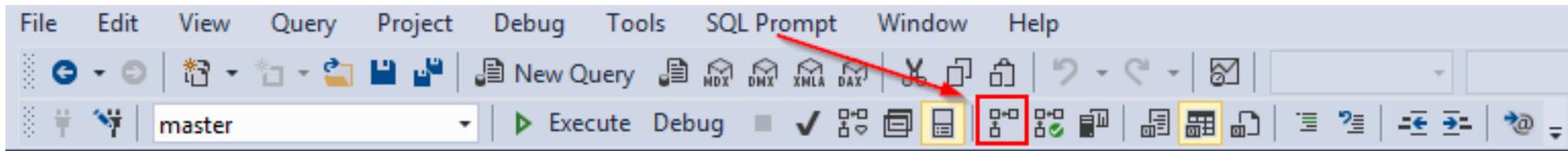


Types of Execution Plans - Estimated vs Actual

- ▶ Estimated execution plans
 - ▶ Query is not executed
 - ▶ Best guess of plan that would actually be used
 - ▶ In some cases cannot be generated
- ▶ Actual execution plans
 - ▶ Query is executed
 - ▶ Some chance it may differ from estimated plan
 - ▶ Includes runtime statistics (actual rows)

Actual Execution Plans

- ▶ Actual plan - text
 - `set statistics profile on;`
- ▶ Actual plan - XML
 - `set statistics xml on;`
- ▶ Actual plan - Graphical
 - ▶ Include Actual Execution Plan (Ctrl-M)



Two Types of Tables

▶ Heaps

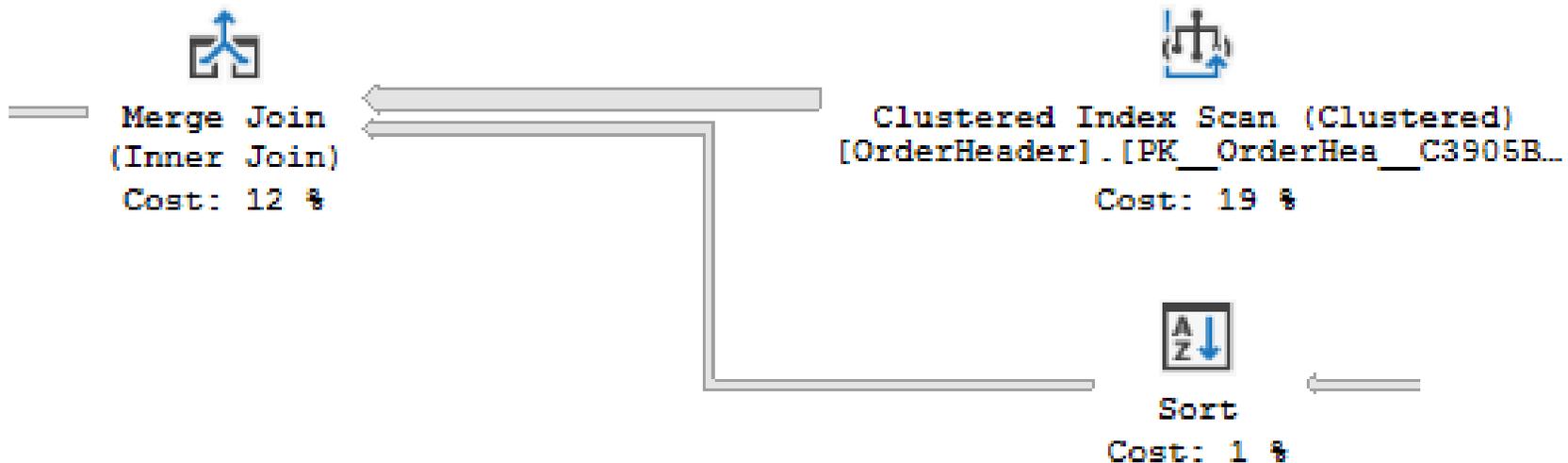
- ▶ Not organized in any particular way
- ▶ No index structure on top of data
- ▶ Can still have nonclustered indexes

▶ Clustered Index

- ▶ Data is stored in key order
- ▶ Has a B-tree structure on top of the data
- ▶ Can also have nonclustered indexes

The Execution Plan

- ▶ Consist of operators and connectors
- ▶ Connector (flow of data)
 - ▶ Width indicates number of rows
- ▶ Plans are frequently read right-to-left, top-to-bottom



Operators

- ▶ About 70 operators possible; most are infrequently seen
- ▶ Responsible to respond for a request for the next row
- ▶ Common operators
 - ▶ Data Access (scans, seeks, lookups)
 - ▶ Joins (merge, nested loops, hash)
 - ▶ Other (sorts, aggregations, spools, etc.)

[Full list of operators](#)

Operators - Data Access

- ▶ Scan - Read entire contents of object
- ▶ Does not necessarily return all rows read
- ▶ May result from non-SARGable predicates
- ▶ Myth: scans are evil



Clustered Index Scan



Index Scan



Table Scan

Operators - Data Access

- ▶ Seek - Uses index structure to find key values
 - ▶ Can be a point lookup or involve a partial scan
- ▶ Cannot seek into a heap
- ▶ Myth: seeks are always good



Clustered Index Seek



Index Seek

Scans vs. Seeks

- ▶ SQL will tend to favor scans if the number of rows expected is large enough that cost for a (sequential) scan is less than the cost of random I/O for seeks
 - ▶ “Tipping point”
- ▶ Cardinality errors can cause the “wrong” access type to be used

Operators - Data Access

- ▶ Lookup - Retrieve additional columns from table
 - ▶ Used when non-clustered index does not have all the columns needed to resolve query (not covering)
- ▶ Useful when number of lookups is small



Key Lookup



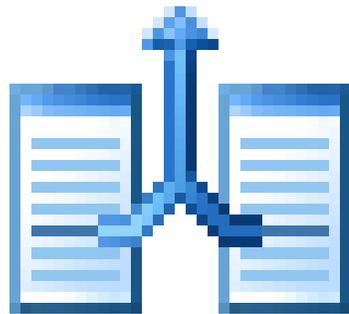
RID Lookup

Operators - Joins

- ▶ Three main join algorithms
 - ▶ Merge Join
 - ▶ Nested Loop Join
 - ▶ Hash Join
- ▶ (Also adaptive join, hybrid nested loop and hash)

Operators - Merge Join

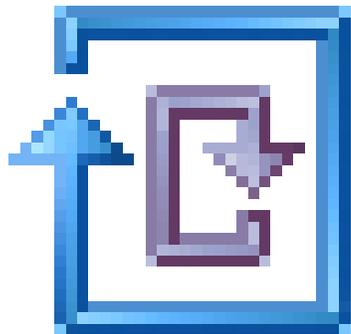
- ▶ Requires both tables to be sorted on join columns
 - ▶ May introduce intermediate sort operation
 - ▶ But sorts are expensive
- ▶ Useful when data is already naturally sorted by join columns



[Bert Wagner video with animation of merge join](#)

Operators - Nested Loop Join

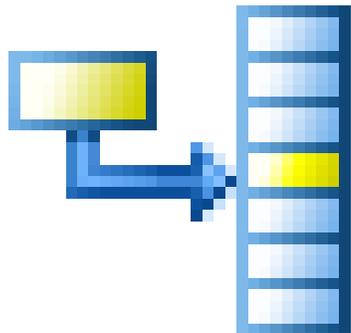
- ▶ Compare each row in top input with each row in bottom input
- ▶ Bottom input may be static or may change depending on value of top row
- ▶ Useful when top input is small and bottom input is efficient to search



[Bert Wagner video with animation of loop join](#)

Operators - Hash Join

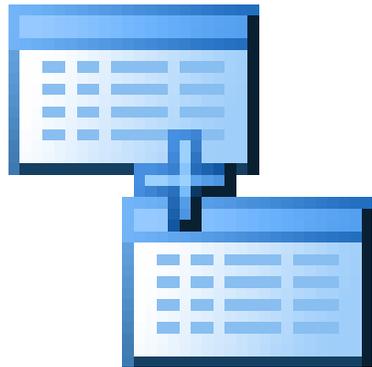
- ▶ Each top row is hashed by join columns and bucketized
- ▶ Each bottom is hashed, looked up in hash table
- ▶ Useful when both inputs are large and unsorted



[Bert Wagner video with animation of loop join](#)

Operators - Set Operations

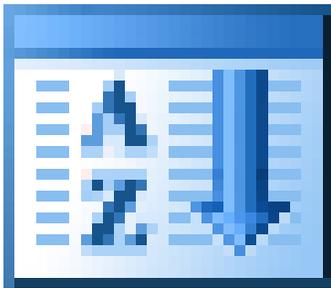
- ▶ Concatenation (UNION ALL)
- ▶ Other operations (UNION, INTERSECT, EXCEPT) handled by combinations of operators



Concatenation

Operators - Sort

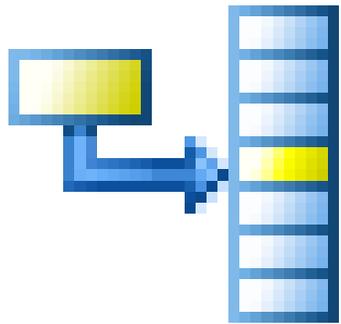
- ▶ Tends to be a very expensive operation
- ▶ Highly dependent on cardinality estimate
 - ▶ Drive memory grant
- ▶ Watch for spills to tempdb
- ▶ Is the sort really needed?



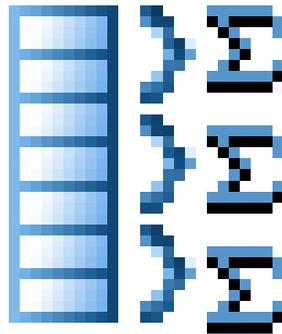
Sort

Operators - Aggregation

- ▶ Calculate SUM, COUNT, AVG, MIN, MAX, etc.
- ▶ Hash aggregate builds hash table to find common rows (based on grouping columns)
- ▶ Stream aggregate input must be sorted, watches for changes in grouping columns



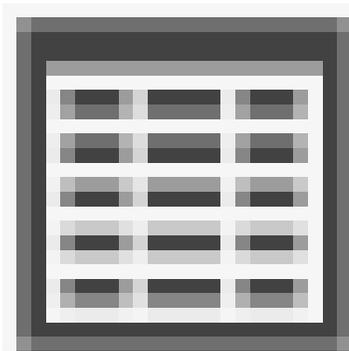
Hash Aggregate



Stream Aggregate

Operators - SELECT

- ▶ (Or INSERT, DELETE, UPDATE, MERGE)
- ▶ Left-most pseudo-operator
- ▶ Contains properties of the execution plan as a whole



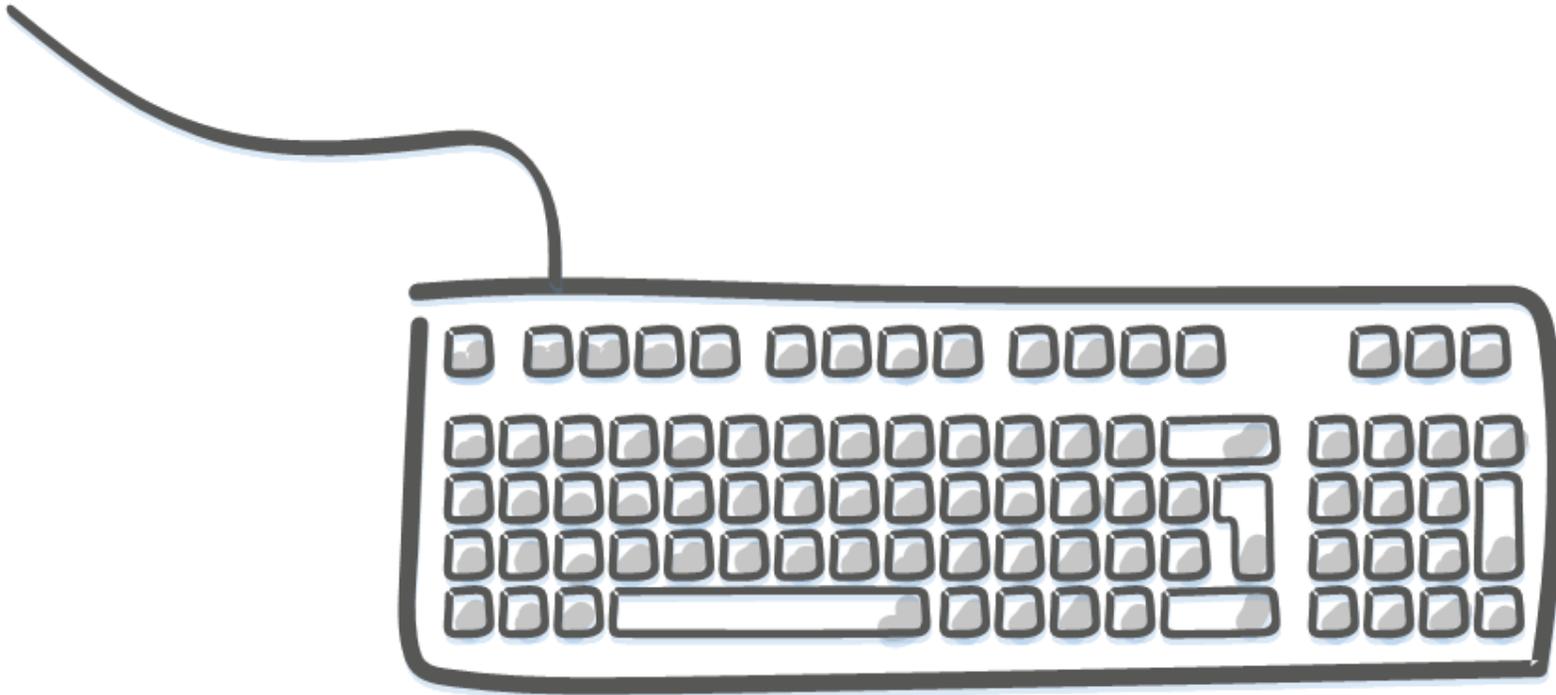
SELECT

Operators

- ▶ And many, many more operators
 - ▶ Various Insert, Update, Delete, Merge operators
 - ▶ Clustered idx, non-clustered idx, heap
 - ▶ Compute Scalar, Constant Scan
 - ▶ Spools (Eager vs. Lazy)
 - ▶ Parallelism
 - ▶ Distribute Streams, Repartition Streams, Gather Streams
 - ▶ Etc.

Demo

- ▶ Execution problem pain points



Resources

- ▶ Grant Fritchey, SQL Server Execution Plans, 3rd Edition ([free download](#))
- ▶ AdventureWorks2014 ([download](#))

Thank You

- ▶ This presentation and supporting materials can be found at www.sqltran.org/executionplans
 - ▶ Slide deck
 - ▶ Scripts

- ▶ allison@sqltran.org • [@sqltran](https://twitter.com/sqltran)