

Get Your Optimizer to Give up All Its Secrets

Brian Hansen
brian@tf3604.com
@tf3604



*PASS

SQLSATURDAY

LOS ANGELES | JUN 13 2020

Our Sponsors



Local User Groups

LA Data Platform

3rd Wednesday of every month

<https://ladataplatform.pass.org/>

Los Angeles User Group

3rd Thursday of each odd month

<https://sqlla.pass.org/>

San Diego User Group

1st and 3rd Thursday of every month

<https://www.meetup.com/sdsqllug/>

<https://www.meetup.com/sdsqllbig/>

Orange County User Group

2nd Thursday of every month

<https://bigpass.pass.org/>

Los Angeles - Korean

Every other Tuesday

<https://sqlangeles.pass.org/>



PASS Summit 2020 - Virtual

Pre-Conference
Sessions

\$349

\$399 after June 19

Monday or Tuesday Pre-
Conference Sessions ?

REGISTER NOW

All-in-One
Bundle

\$899

\$999 after June 19

Monday and Tuesday Pre-
Conference Sessions ?

3-Day Conference ?

12 Month On Demand Access ?

Community Events

Ability to add Sessions
Download ?

REGISTER NOW

Full 3-Day
Virtual Summit

\$499

\$599 after June 19

3-Day Conference ?

12 Month On Demand Access ?

Community Events

Ability to add Sessions
Download ?

REGISTER NOW



PASS Summit 2020 – Discount Code

- LA User Group: LGDISBEKB
- LA Data Platform: LGDISEI2J
- South Florida SQL Server User Group:
LGDISSC75



Welcome to SQL Saturday

- Enjoy this day of learning
- Be sure to visit and thank the sponsors
- Be sure to thank the organizer and volunteers
- Take time to NETWORK with others. That's what this is really all about!
- Act professionally and treat others with respect (like this was a work environment)



About This Session

- What this session is not
 - An end-to-end optimizer session
 - A performance tuning session



- Goals of this session
- Additional understanding of SQL Server internals
- Deeper understanding: write better queries!
- Provide additional skills for performance tuning



Agenda

- Background:
 - Logical processing order
 - Physical processing considerations
- Executing a query:
 - Parse, bind, transform, optimize, execute
- Heuristics, transformation rules, parse trees, memos
- Limitations & DMVs



This Is Only the Foundation

- The materials we are covering here will only skim the surface of what is possible.
- Understanding optimizer internals takes time and study.
- Many features you run across have minimal available information out there.
- Don't get frustrated ... just keep on diving in (if this is interesting to you)!



Logical Processing Order

- Defines the sequence in which SQL elements are logically processed
- Forms the starting basis for parsing the submitted query
- Usually discussed from the perspective of a SELECT query; similar for UPDATE / DELETE / INSERT / MERGE
- Declarative vs procedural programming
 - “What” vs “How”



Logical Processing Order

- FROM
- ON
- JOIN / APPLY
- PIVOT / UNPIVOT
- WHERE
- GROUP BY
- WITH CUBE / ROLLUP
- HAVING
- SELECT
- DISTINCT
- ORDER BY
- TOP
- OFFSET ... FETCH

For more details, see [this](#) and subsequent articles from Itzik Ben-Gan



```
select top 5 od.ProductId, sum(od.Quantity) - 20 ExcessOrders
from dbo.OrderHeader oh
inner join dbo.OrderDetail od on oh.OrderId = od.OrderId
inner join dbo.Customer cust on oh.CustomerId = cust.CustomerID
where cust.State = 'CA'
group by od.ProductId
having sum(od.Quantity) >= 20
order by od.ProductId;
```

Table	Columns	Rows
OrderHeader	3	301,811
OrderDetail	5	603,133
Customer	6	70,132



```
select top 5 od.ProductId, sum(od.Quantity) - 20 ExcessOrders
from dbo.OrderHeader oh
inner join dbo.OrderDetail od on oh.OrderId = od.OrderId
inner join dbo.Customer cust on oh.CustomerId = cust.CustomerID
where cust.State = 'CA'
group by od.ProductId
having sum(od.Quantity) >= 20
order by od.ProductId;
```

Step 1: FROM

- OrderHeader joined to OrderDetail
- Perform Cartesian join
- Result is 182,032,173,863 rows / 8 columns
- This is result table R1



```
select top 5 od.ProductId, sum(od.Quantity) - 20 ExcessOrders
from dbo.OrderHeader oh
inner join dbo.OrderDetail od on oh.OrderId = od.OrderId
inner join dbo.Customer cust on oh.CustomerId = cust.CustomerID
where cust.State = 'CA'
group by od.ProductId
having sum(od.Quantity) >= 20
order by od.ProductId;
```

Step 2: FROM

- R1 joined to Customer (Cartesian join)
- Result is 12,766,280,417,359,916 rows / 14 columns
- This is result table R2



```
select top 5 od.ProductId, sum(od.Quantity) - 20 ExcessOrders
from dbo.OrderHeader oh
inner join dbo.OrderDetail od on oh.OrderId = od.OrderId
inner join dbo.Customer cust on oh.CustomerId = cust.CustomerID
where cust.State = 'CA'
group by od.ProductId
having sum(od.Quantity) >= 20
order by od.ProductId;
```

Step 2: ON

- Find rows in R2 where OrderId = OrderId
- Result is 42,298,923,556 rows / 14 columns
- This is result table R3




```
select top 5 od.ProductId, sum(od.Quantity) - 20 ExcessOrders
from dbo.OrderHeader oh
inner join dbo.OrderDetail od on oh.OrderId = od.OrderId
inner join dbo.Customer cust on oh.CustomerId = cust.CustomerID
where cust.State = 'CA'
group by od.ProductId
having sum(od.Quantity) >= 20
order by od.ProductId;
```

Step 4: ON

- Find rows in R3 where CustomerId = CustomerId
- Result is 603,133 rows / 14 columns
- This is result table R4



```
select top 5 od.ProductId, sum(od.Quantity) - 20 ExcessOrders
from dbo.OrderHeader oh
inner join dbo.OrderDetail od on oh.OrderId = od.OrderId
inner join dbo.Customer cust on oh.CustomerId = cust.CustomerID
where cust.State = 'CA'
group by od.ProductId
having sum(od.Quantity) >= 20
order by od.ProductId;
```

Step 5: WHERE

- Find rows in R4 where State = 'TN'
- Result is 96,317 rows / 14 columns
- This is result table R5



```
select top 5 od.ProductId, sum(od.Quantity) - 20 ExcessOrders
from dbo.OrderHeader oh
inner join dbo.OrderDetail od on oh.OrderId = od.OrderId
inner join dbo.Customer cust on oh.CustomerId = cust.CustomerID
where cust.State = 'CA'
group by od.ProductId
having sum(od.Quantity) >= 20
order by od.ProductId;
```

Step 6: GROUP BY

- Arrange rows into groups by ProductId
- Within each group compute SUM(Quantity)
- Result is 7,514 rows / 2 columns
- This is result table R6 (ProductId, SUM(Quantity))
 - Only these 2 columns are available in downstream steps



```
select top 5 od.ProductId, sum(od.Quantity) - 20 ExcessOrders
from dbo.OrderHeader oh
inner join dbo.OrderDetail od on oh.OrderId = od.OrderId
inner join dbo.Customer cust on oh.CustomerId = cust.CustomerID
where cust.State = 'CA'
group by od.ProductId
having sum(od.Quantity) >= 20
order by od.ProductId;
```

Step 7: HAVING

- Find rows in R6 where SUM(Quantity) >= 20
- Result is 3,492 rows / 2 columns
- This is result table R7



```
select top 5 od.ProductId, sum(od.Quantity) - 20 ExcessOrders
from dbo.OrderHeader oh
inner join dbo.OrderDetail od on oh.OrderId = od.OrderId
inner join dbo.Customer cust on oh.CustomerId = cust.CustomerID
where cust.State = 'CA'
group by od.ProductId
having sum(od.Quantity) >= 20
order by od.ProductId;
```

Step 8: SELECT

- Evaluate expressions in the select list
 - ProductId → ProductId
 - SUM(Quantity) – 20 → ExcessOrders
- Result is 3,492 rows / 2 columns
- This is result table R8



```
select top 5 od.ProductId, sum(od.Quantity) - 20 ExcessOrders
from dbo.OrderHeader oh
inner join dbo.OrderDetail od on oh.OrderId = od.OrderId
inner join dbo.Customer cust on oh.CustomerId = cust.CustomerID
where cust.State = 'CA'
group by od.ProductId
having sum(od.Quantity) >= 20
order by od.ProductId;
```

Step 9: ORDER BY

- Sort R8 by ProductId
- Result is 3,492 rows / 2 columns
- This is result table R9



```
select top 5 od.ProductId, sum(od.Quantity) - 20 ExcessOrders
from dbo.OrderHeader oh
inner join dbo.OrderDetail od on oh.OrderId = od.OrderId
inner join dbo.Customer cust on oh.CustomerId = cust.CustomerID
where cust.State = 'CA'
group by od.ProductId
having sum(od.Quantity) >= 20
order by od.ProductId;
```

Step 10: TOP

- Keep the first 5 rows in R9
 - Remaining rows get discarded
- Result is 5 rows / 2 columns
- This is result table R10
- Logical processing is complete



How SQL Server Sees the Query

```
from dbo.OrderHeader oh
inner join dbo.OrderDetail od
inner join dbo.Customer cust
on oh.OrderId = od.OrderId
on oh.CustomerId = cust.CustomerID
where cust.State = 'CA'
group by od.ProductId
having sum(od.Quantity) >= 20
select od.ProductId, sum(od.Quantity) - 20 ExcessOrders
order by od.ProductId
top 5;
```



Logical Operators

- Get
- Join
 - \bowtie inner
 - $\bowtie\bowtie\bowtie$ outer
 - \times Cartesian
 - \ltimes semi*
 - \triangleright anti-semi*
- Apply
- Set Operators
 - \cup union
 - \cap intersection
 - \setminus except
 - σ Select (SQL: where)
 - π Project (SQL: select)
 - Γ Aggregate









Join Type Comparison

Join Type	Condition	Left rows	Right rows
Inner	For each predicate match, output left row + right row	0+	0+
Left outer	Same as inner, but if no predicate match, output left row + NULL placeholders for right row	1+	0+
Full outer	Same as left, but if no predicate match in right, output NULL placeholders for left table + right row	1+	1+
Cartesian / cross (m × n)	Match each row in left with each row in right (no concept of predicate)	n	m
Left semi	Output left row once if predicate match	0 or 1	0
Left anti-semi	Output left row once if no predicate match	0 or 1	0









Physical Operators: Logical “get”

- Scan  
- Seek  
- Lookups  
- Heap vs clustered index vs non-clustered index
- Ordered vs unordered
- Forward vs backward



Physical Operators: Other

- Join
 - Merge 
 - Nested loops 
 - Hash 
- Aggregate
 - Stream aggregate 
 - Hash aggregate 
- Select
 - Filter 

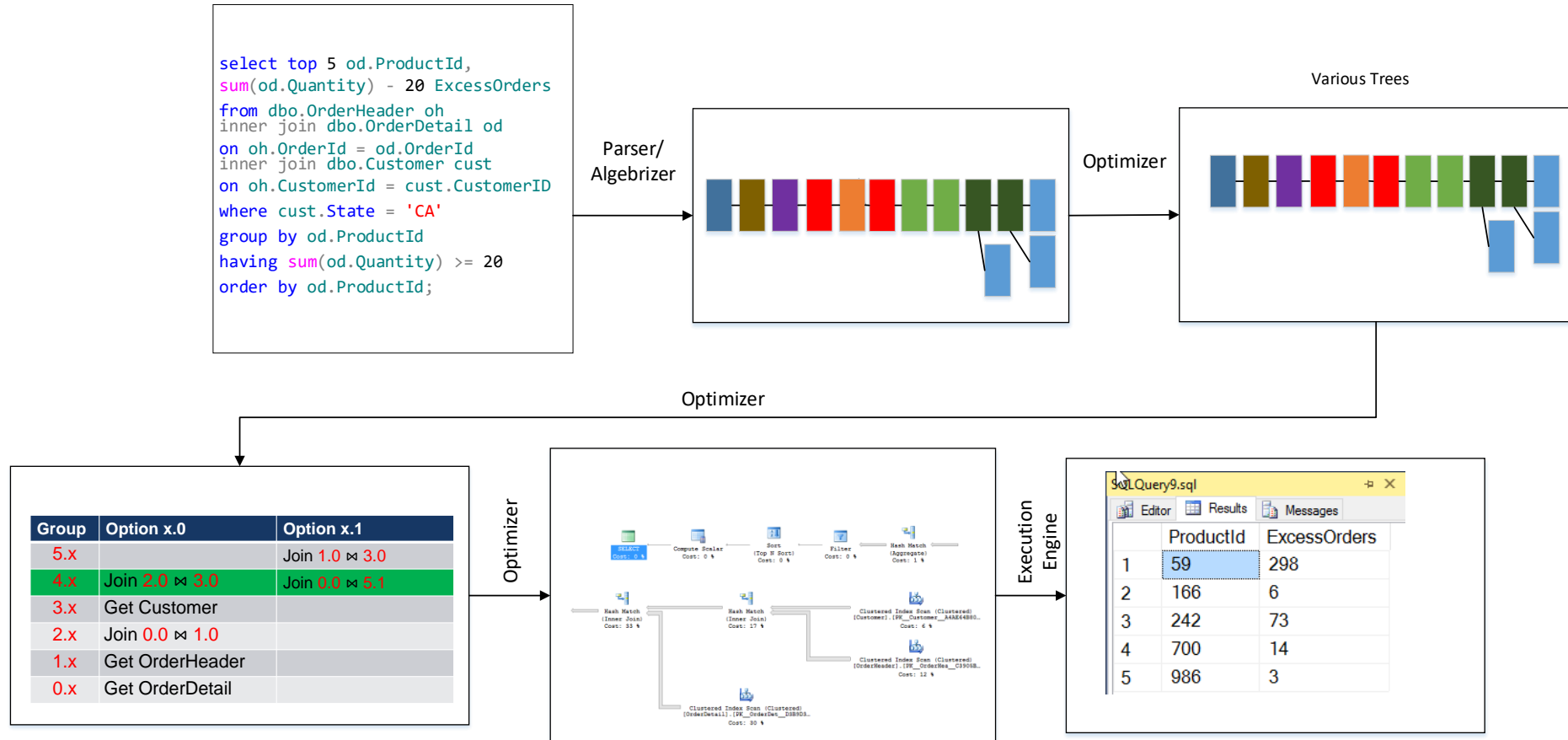


Process of Executing a Query

- Parsing and binding
- Optimization
- Execution



Process of Executing a Query - Graphical



Parsing and Binding (1 of 3)

2

- Algebrizer (the “normalizer” in SQL 2000)
 - Parser: validate syntactical correctness
 - Build initial parse tree
 - Identify constants

```
selekt col1  
form objA  
wear col2 = 1  
orderby col3
```



Parsing and Binding (2 of 3)

- Expand views

```
select c.CustomerID, c.FirstName, c.LastName, oh.OrderDate
from CorpDB.dbo.ImportantCustomers c
inner join CorpDB.dbo.OrderHeader oh on oh.CustomerId = c.CustomerId
where c.LastName = 'Hansen';
```

```
select c.CustomerID, c.FirstName, c.LastName, oh.OrderDate
from
(
    select c.CustomerId, c.FirstName, c.LastName, c.State
    from CorpDB.dbo.Customer c
    where c.State = 'MO' )
) c
inner join CorpDB.dbo.OrderHeader oh on oh.CustomerId = c.CustomerId
where c.LastName = 'Hansen';
```



Parsing and Binding (3 of 3)

- Binding

- Metadata discovery / name resolution / permissions
- Data type resolution (i.e., UNION)

```
select 1 union all select 'Some text';
```

Conversion failed when converting the varchar value 'Some text' to data type int.

- Aggregate binding

```
select LastName, CustomerID, count(*) Nbr from Customer group  
by LastName;
```

Column 'Customer.CustomerID' is invalid in the select list because it is not contained in either an aggregate function or the GROUP BY clause.



Parse Trees*

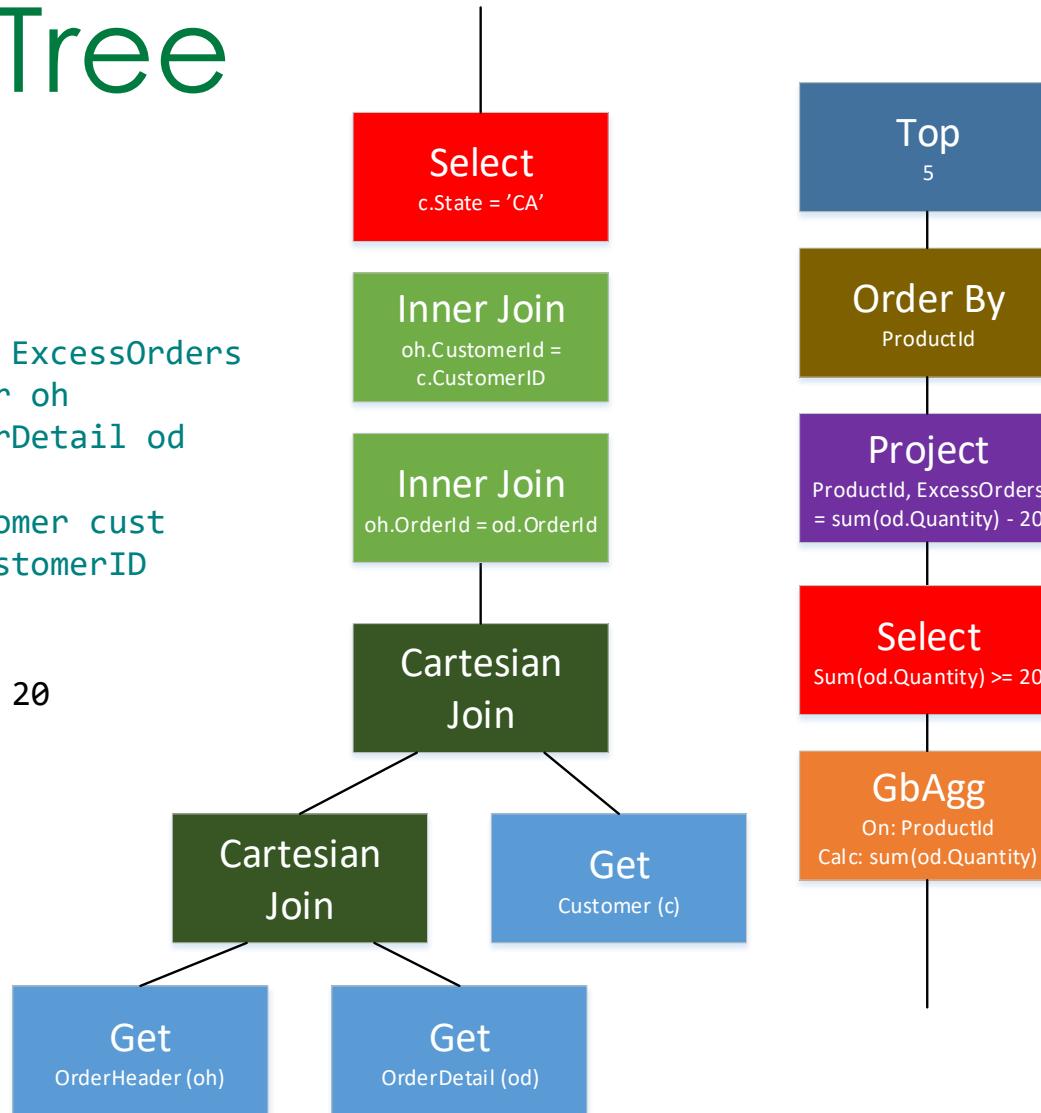
- Internal representation of query operation
- Nodes may be logical or physical operators
 - 0 to infinity inputs, 1 output
- SQL Server will output parse trees at various phases of optimization
 - A variety of trace flags will trigger output

* Or query trees, or relational trees

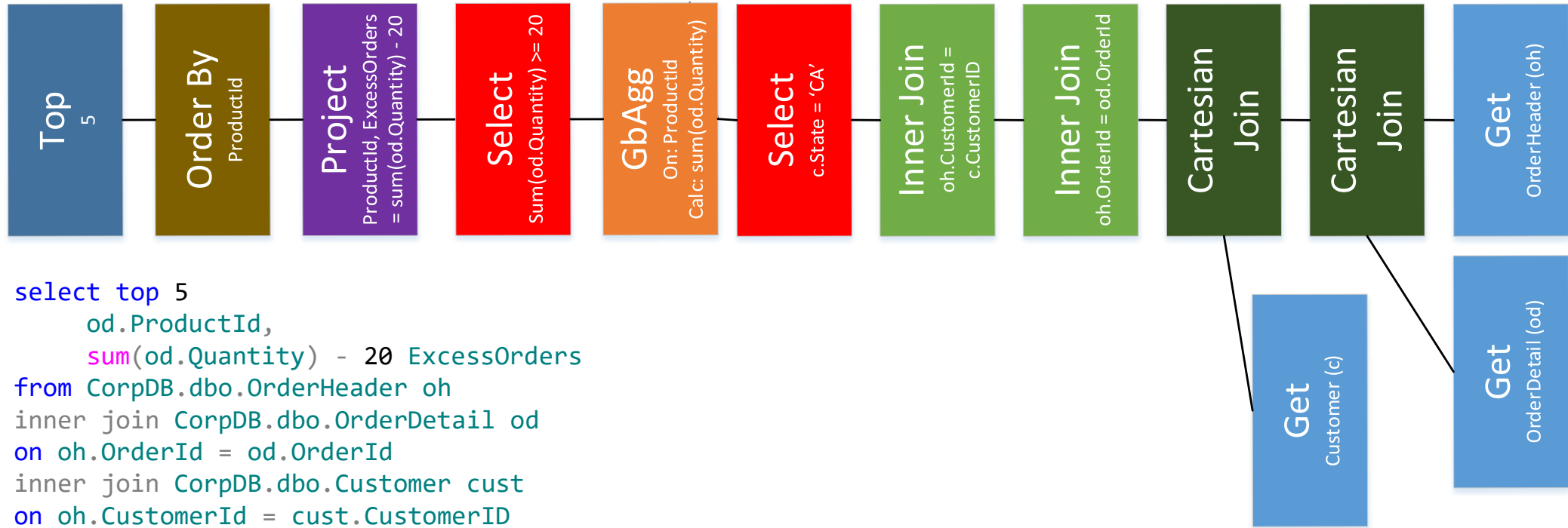


Example Parse Tree

```
select top 5
  od.ProductId,
  sum(od.Quantity) - 20 ExcessOrders
from CorpDB.dbo.OrderHeader oh
inner join CorpDB.dbo.OrderDetail od
on oh.OrderId = od.OrderId
inner join CorpDB.dbo.Customer cust
on oh.CustomerId = cust.CustomerID
where cust.State = 'MO'
group by od.ProductId
having sum(od.Quantity) >= 20
order by od.ProductId;
```



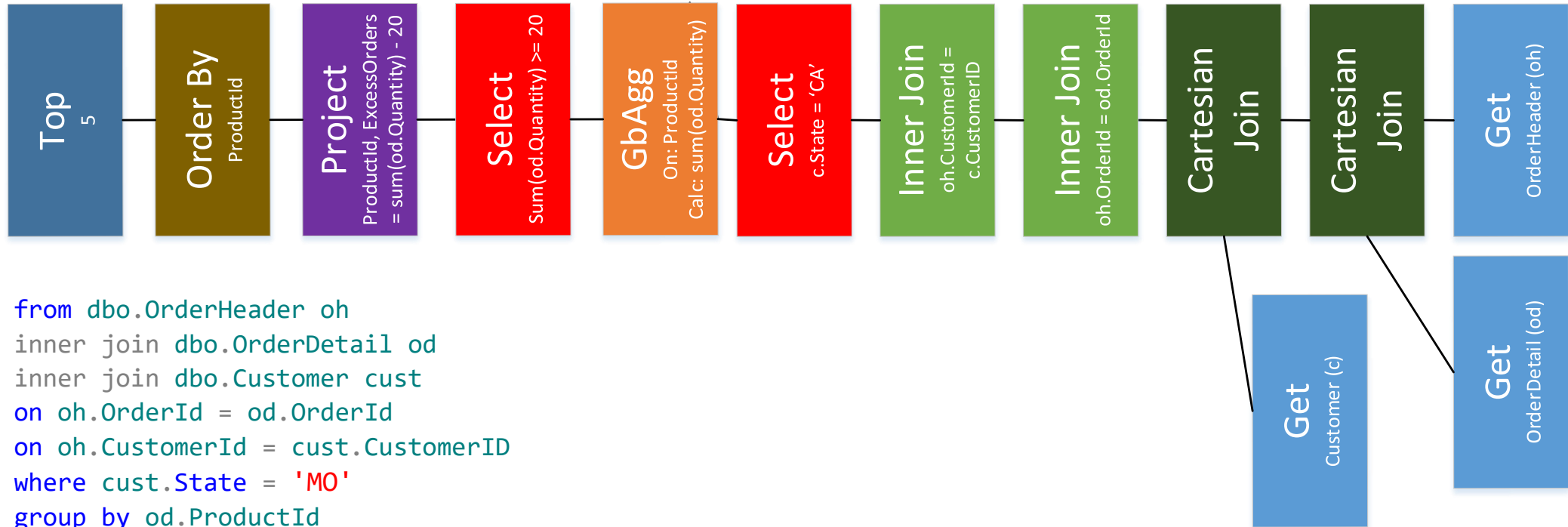
Example Parse Tree (Horizontal)



```
select top 5
    od.ProductId,
    sum(od.Quantity) - 20 ExcessOrders
from CorpDB.dbo.OrderHeader oh
inner join CorpDB.dbo.OrderDetail od
on oh.OrderId = od.OrderId
inner join CorpDB.dbo.Customer cust
on oh.CustomerId = cust.CustomerID
where cust.State = 'MO'
group by od.ProductId
having sum(od.Quantity) >= 20
order by od.ProductId;
```



Query Tree



```
from dbo.OrderHeader oh
inner join dbo.OrderDetail od
inner join dbo.Customer cust
on oh.OrderId = od.OrderId
on oh.CustomerId = cust.CustomerID
where cust.State = 'MO'
group by od.ProductId
having sum(od.Quantity) >= 20
select od.ProductId,
       sum(od.Quantity) - 20 ExcessOrders
order by od.ProductId
top 5;
```



Logical Plans

- Similar to physical execution plans
- Multiple logical plans generated during query optimization
- Have no physical properties, such as
 - Indexes
 - Row counts
 - Keys
- Logical operators only



Showing Query Trees

Trace Flag	Result
3604	Output extra information to “Messages” tab in SSMS
8605	Show initial parse tree (converted)
8606	Show transformed parse trees (input, simplified, join-collapsed, normalized)
8607	Show output tree



Optimization (1 of 2)

- Simplification (heuristic rewrites, not cost-based)
 - Standardize queries, remove redundancies
 - Subqueries to joins
 - Predicate pushdown
 - Foreign key table removal
 - Contradiction detection
 - Aggregates on unique keys
 - Convert outer join to inner
 - Retrieve statistics; do cardinality estimation
 - Create / update auto stats
 - SQL Server 7 vs 2014+ CE engine
 - Other physical properties (keys, nullability, constraints)
- Trivial plan
 - Only one possible way to execute query

8

3



Optimization (2 of 2)

- Search phases 0 through 2
 - Search 0: “Transaction Processing”
 - Simple, basic tests; internal cost threshold
 - Search 1: “Quick Plan”
 - More rules, parallel exploration; internal cost threshold
 - Search 2: “Full Optimization”
 - Full set of rules; usually exits on timeout
 - Extensive use of heuristics to prune search space
- Construct execution plan
- Plan caching (query text hash, set options)



Search Space

- “Every possible execution plan that achieves the directive of a given query”
- Can be an enormous number of plans
- Consider:

```
select ...  
from a join b on ... join c on ... join d on ...
```
- Assume a, b, c, d are tables with clustered index & 3 non-clustered indexes each



```
select ...
from a join b on ... join c on ... join d on ...
```

Physical access methods (per table)

Unordered clustered index scan	1
Unordered nonclustered index scan (covering)	3
Ordered clustered index scan	1
Ordered nonclustered index scan (covering)	3
Nonclustered seek + ordered partial scan + lookup	3
Unordered nonclustered index scan + lookup	3
Clustered index seek + ordered partial scan	1
Nonclustered index seek + ordered partial scan (covering)	3
Indexed views	0
Index intersection*	54
Total	72

6 combinations of 2 indexes; 1 join per pair = 6 joins; 3 join methods each = 18
 6 combinations of 3 indexes; 2 joins per triplet = 12 joins; 3 join methods each = 36; total = 54



```
select ...
from a join b on ... join c on ... join d on ...
```

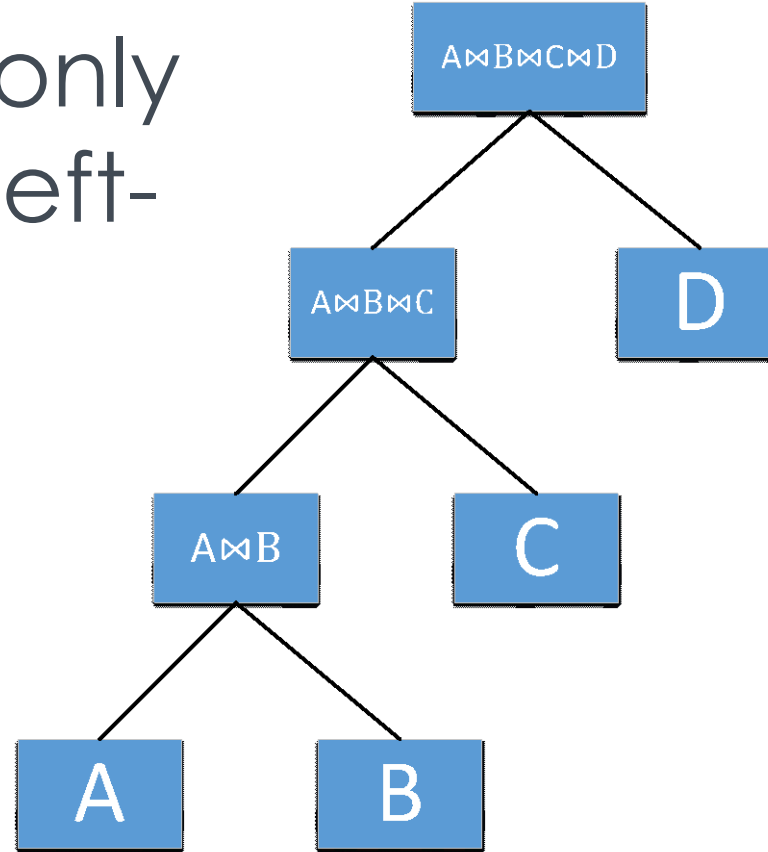
Logical Join Orders: 24 Total (or are there more?)

a⋈b⋈c⋈d	c⋈a⋈b⋈d
a⋈b⋈d⋈c	c⋈a⋈d⋈b
a⋈c⋈b⋈d	c⋈b⋈a⋈d
a⋈c⋈d⋈b	c⋈b⋈d⋈a
a⋈d⋈c⋈b	c⋈d⋈a⋈b
a⋈d⋈b⋈c	c⋈d⋈b⋈a
b⋈a⋈c⋈d	d⋈a⋈b⋈c
b⋈a⋈d⋈c	d⋈a⋈c⋈b
b⋈c⋈a⋈d	d⋈b⋈a⋈c
b⋈c⋈d⋈a	d⋈b⋈c⋈a
b⋈d⋈a⋈c	d⋈c⋈a⋈b
b⋈d⋈c⋈a	d⋈c⋈b⋈a



Join Order Considerations

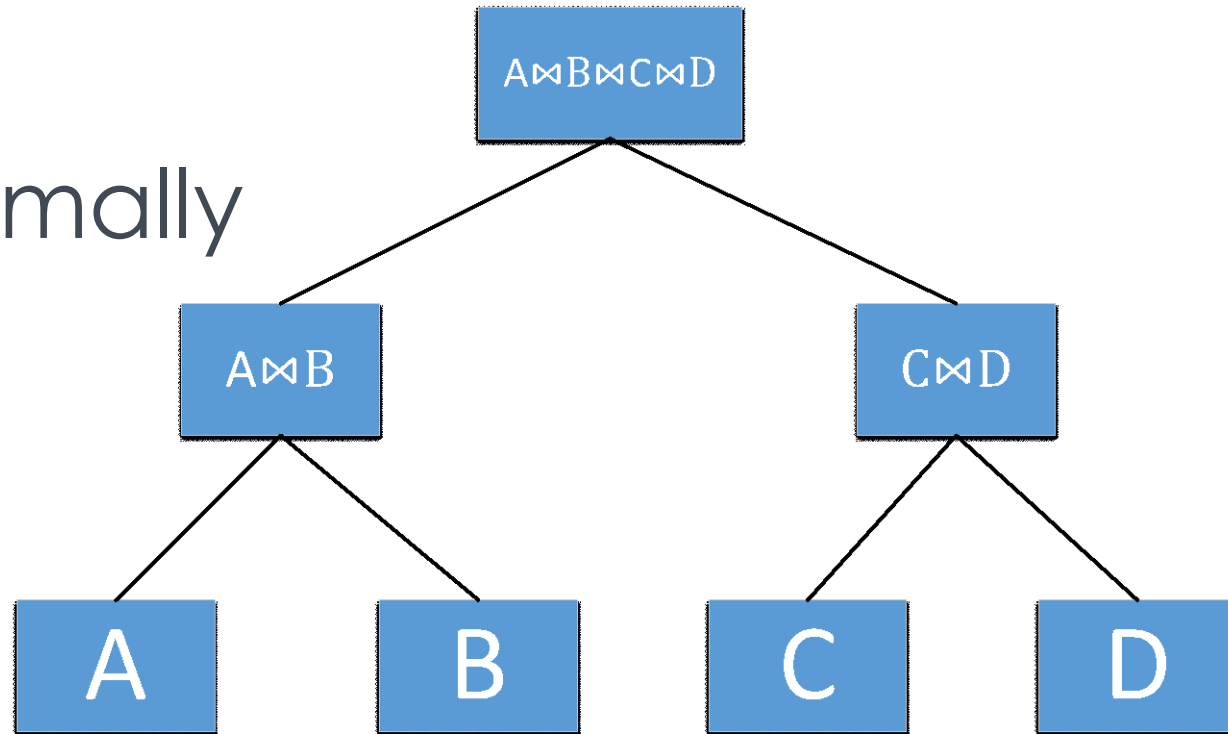
- So far we've only considered "left-deep" trees
- $n!$



Join Order Considerations, continued

- There are also “bushy” trees
- $(2n-2)!/(n-1)!$
- Optimizer normally does not consider these

6



```
select ...  
from a join b on ... join c on ... join d on ...
```

- 72 possible physical data access methods
- 120 possible logical join orders
- 3 physical joins possible per logical join
 - May require intermediate sort operation
- = 25,920 possible plans
- Much larger for more complex queries
- Optimizer uses heuristics to limit search space



sys.dm_exec_query_optimizer_info



- Documented. Sort of.
- Three columns:
 - *counter*: Name of the observation
 - *occurrence*: Number of times observation was recorded
 - *value*: Average per occurrence
- Collect before and after images of this view on a quiet system



Smart Optimization



<http://imgs.xkcd.com/comics/efficiency.png>



Can the Optimizer Dig a Bit Deeper?

- Trace flag 8780:
 - Considerably more attempts in Search 2
 - Very often still won't come up with a different (or better) plan

counter	occurrence	value
elapsed time	1	0.05799999999...
final cost	1	4.070147361373...
hints	1	1
maximum DOP	1	0
optimizations	1	1
search 0	1	1
search 0 tasks	1	1883
search 0 time	1	0.01799999999...
search 1 tasks	1	3103
search 1 time	1	0.015000000000...
tables	1	12.99999999994...
tasks	1	4986
timeout	1	1

counter	occurrence	value
elapsed time	1	7.193999999999...
final cost	1	1.542904861038...
gain stage 0 to st...	1	0.459126414372...
gain stage 1 to st...	1	0.299136825438...
hints	1	1
maximum DOP	1	0
optimizations	1	1
search 0 tasks	1	1883
search 0 time	1	0.01699999999...
search 1 tasks	1	8250
search 1 time	1	0.075000000000...
search 2	1	1
search 2 tasks	1	598191
search 2 time	1	7.045000000000...
tables	1	13.000000000005...
tasks	1	608324.0000000...



An Interesting Metric: Gain

- Indicates improvement from phase to phase
 - Search 0 to 1 gain
 - Search 1 to 2 gain
 - Value that is ≥ 0 and < 1
 - 0 indicates no improvement
 - Approaching 1 indicates significant improvement

- Definition:

$$Gain_{S0 \text{ to } S1} = \frac{MinCost(S0) - MinCost(S1)}{MinCost(S0)}$$



Heuristics and Transformations

- Heuristics
 - Rules that can eliminate entire branches of the search space
- Transformations
 - Find equivalent operations to get same output
 - Rule-based
 - DBCC SHOWONRULES
 - DBCC RULEON / RULEOFF
 - Four types
 - Simplification, exploration, implementation, property enforcement



Transformations: Exploration

- Start from a logical operation (may be a sub-branch of the full query): the pattern
- Find equivalent logical operations: the substitute
- Examples:
 - Join commutativity: $A \bowtie B \rightarrow B \bowtie A$
 - Join associativity: $(A \bowtie B) \bowtie C \rightarrow A \bowtie (B \bowtie C)$
 - Aggregate before join



Transformations: Implementation

- Start from a logical operation
- Find equivalent physical operation
- Example:
 - $A \bowtie B \rightarrow A$ (nested loops join) B
 - $A \bowtie B \rightarrow A$ (merge join) B
 - $A \bowtie B \rightarrow A$ (hash join) B
- Obtain costing on physical operations
- Can prune expensive branches from tree



Transformations: Property Enforcement

- Properties associated with parse tree nodes
 - Uniqueness, type, nullability, sort order
 - Constraints on column values
- Transformation rules may cause certain properties to be enforced
 - Example: sort order for a merge join



sys.dm_exec_query_transformation_stats

10

- One row per transformation rule
 - “Promise_Total” – Estimate of how useful might the rule be for this query
 - “Built_Substitute” – Number of times the rule generated an alternate tree
 - “Succeeded” – Number of times the rule was incorporated into search space
- Collect before and after images of this view on a quiet system



Factors Considered by the Optimizer

- Memory grants
- Costing
 - Cold cache
 - Sequential vs random I/O
 - But not the nature of the I/O subsystem
 - CPU costs, core count, available memory
 - Cardinality estimator
 - What do cost units really mean?



Memo Structure

- Used to explore different alternatives to a portion of the query tree
- Can think of it as a matrix
 - Rows (groups) represent substitutes – each entry is logically equivalent
 - Columns represent application of a transformation rule
- Each entry is hashed to prevent duplication
- Physical substitutes are costed



Example Memo

```
select *  
from CorpDB.dbo.OrderDetail od  
inner join CorpDB.dbo.OrderHeader oh on od.OrderId = oh.OrderId  
inner join CorpDB.dbo.Customer c on c.CustomerID = oh.CustomerId;
```

Group	Option 0
4	Join 2 ⋈ 3
3	Get Customer
2	Join 0 ⋈ 1
1	Get OrderHeader
0	Get OrderDetail



Example Memo

- Apply join associativity:
 - $(OD \bowtie OH) \bowtie C \rightarrow OD \bowtie (OH \bowtie C)$

Group	Option x.0	Option x.1
5.x		Join 1.0 \bowtie 3.0
4.x	Join 2.0 \bowtie 3.0	Join 0.0 \bowtie 5.1
3.x	Get Customer	
2.x	Join 0.0 \bowtie 1.0	
1.x	Get OrderHeader	
0.x	Get OrderDetail	



Example Memo

12

- Apply join commutativity:
 - $(OD \bowtie OH) \bowtie C \rightarrow (OH \bowtie OD) \bowtie C$

Group	Option x.0	Option x.1	Option x.2
5.x		Join 1.0 \bowtie 3.0	
4.x	Join 2.0 \bowtie 3.0	Join 0.0 \bowtie 5.1	Join 2.2 \bowtie 3.0
3.x	Get Customer		
2.x	Join 0.0 \bowtie 1.0		Join 1.0 \bowtie 0.0
1.x	Get OrderHeader		
0.x	Get OrderDetail		



The Optimizer Is Exceptionally Complex

- It has to deal with many things we've not discussed
 - DML (updates, deletes, inserts, merges; output clause)
 - Halloween protection
 - Triggers
 - Index updates
 - Constraint management
 - Wide vs. narrow updates
 - Data warehouse optimization
 - Columnstore, full-text, spatial, xml, filtered indexes and sparse columns
 - Window functions, partitioned tables, Hekaton, Stretch DB, other new features
 - Row vs. batch mode
 - And much more



Conclusions

- SQL is a declarative language
 - In theory, it shouldn't matter how SQL is written
 - We are effectively giving SQL Server a set of requirements and asking it to write a program for us
 - In practice, it does matter because no optimizer is perfect
 - It will give us correct results
 - In the real world, efficiency matters
- Writing “better” queries
 - Sometimes we need to “out-smart” the optimizer



Appendix: Trace Flags

TF	Meaning
2363	Show statistics used by optimizer (SQL 2014+ CE) and <i>lots</i> of other info
2372	Show memory usage at each phase
2373	Show memory usage for rules and properties
3604	Output to client (“Messages” tab)
7352	Show final query tree (post-optimization rewrites)
8605	Show initial parse tree (converted)
8606	Show transformed parse trees (input, simplified, join-collapsed, normalized)
8607	Show output tree
8608	Show initial memo structure



Appendix: Trace Flags, continued

TF	Meaning
8609	Show task and operation type counts
8612	Add cardinality info to tree
8615	Show final memo structure
8619	Show applied rules (SQL 2012+)
8620	Show applied rules and memo arguments (SQL 2012+)
8621	Show applied rules and resulting tree (SQL 2012+)
8649	Force parallel plan
8666	Add debugging info to query plan (in the “F4” properties)
8675	Show optimization search phases and times
8757	Disable trivial plan generation



Appendix: Trace Flags, continued

TF	Meaning
8780	Give query processor more “time” to optimize query
9130	Show pushed predicate
9204	Show statistics used by optimizer (fully loaded) (SQL 7 CE only)
9292	Show statistics used by optimizer (header only) (SQL 7 CE only)

* And many more ...



Appendix: Commands

- DBCC TRACEON / TRACEOFF
- DBCC RULEON / RULEOFF
- DBCC SHOWONRULES
- DBCC SHOWOFFRULES
- option (recompile, querytraceon #####, queryruleoff 'xxx')
- sys.dm_exec_query_optimizer_info
- sys.dm_exec_query_transformation_stats



Appendix: History of SQL's Optimizer

- Volcano Optimizer (April 1993) ([PDF](#))
 - Goetz Graefe, William J. McKenna
 - Based on Graefe's earlier Exodus Optimizer
- Cascades Framework (1995) ([PDF](#))
 - Goetz Graefe
 - Refinement of the Volcano Optimizer
 - Basis for rewritten optimizer in SQL Server 7.0
 - Major innovation: the memo structure



References

- Benjamin Nevarez ([Blog](#))
 - [Inside the SQL Server Query Optimizer](#)
- Paul White
 - [Page Free Space blog](#) (especially [this](#) series)
 - [SQL Performance blog](#)
- Conor Cunningham ([Blog](#))
 - *Microsoft SQL Server 2012 Internals* (Kalen Delaney, editor), chapter 11
 - [SQLBits Session](#)



Thank You

- This presentation and supporting materials can be found at www.tf3604.com/optimizer.
 - Slide deck
 - Scripts
 - Sample database
 - SQL Server Query Tree Viewer binaries & source

brian@tf3604.com • @tf3604

