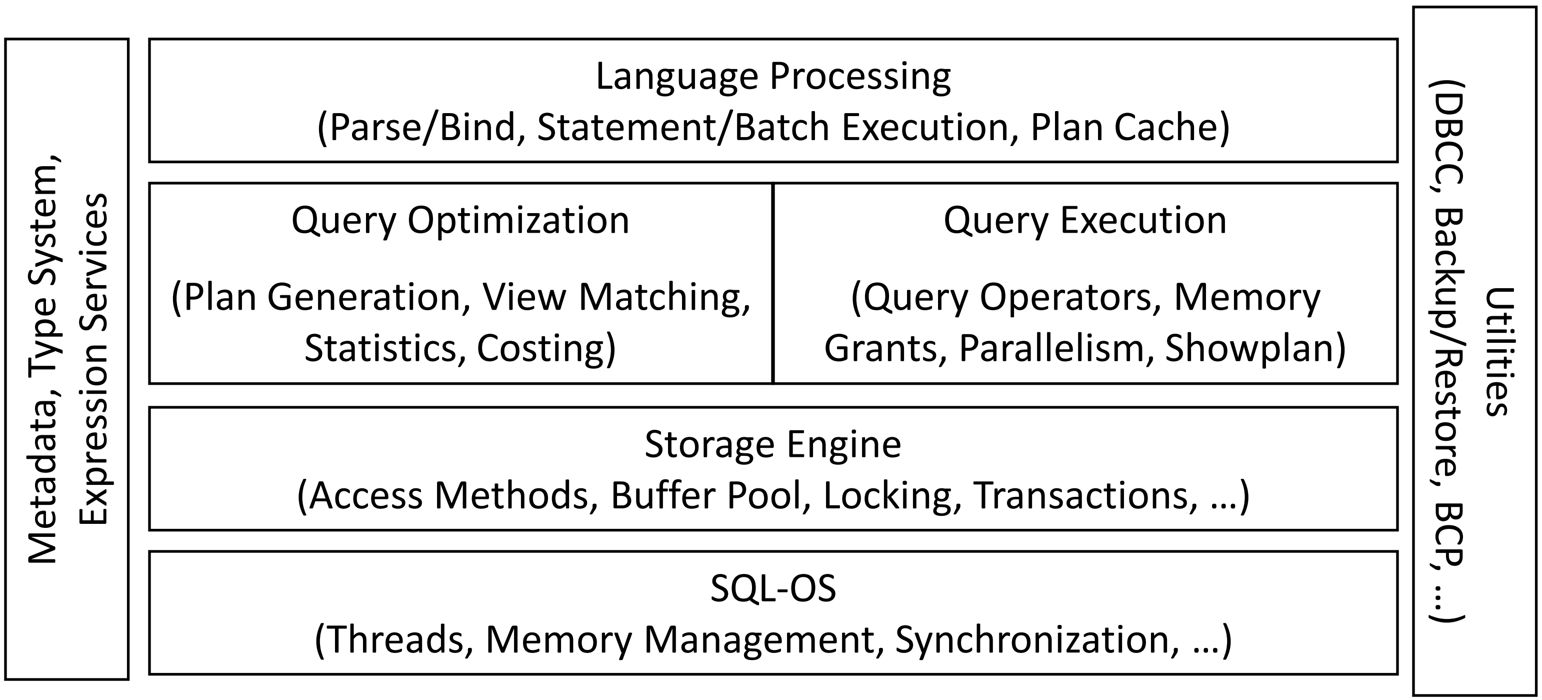
Understanding Query Processing and Query Plans in SQL Server

Craig Freedman
Software Design Engineer
Microsoft SQL Server

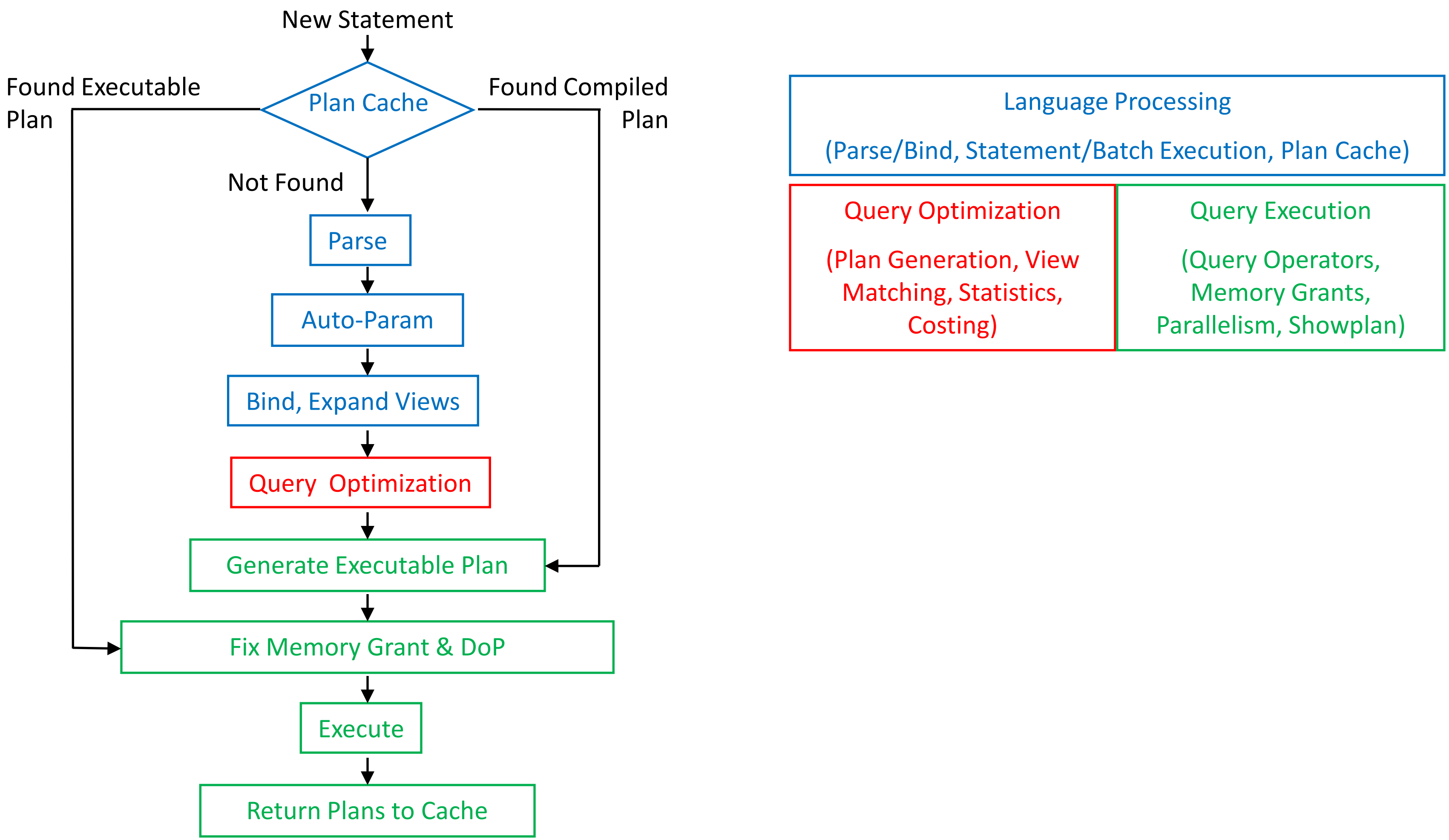
Outline

- SQL Server engine architecture
- Query execution overview
- Showplan
- Common iterators
- Other interesting plans

SQL Server engine high-level architecture



Query processing overview



Query execution overview

- Query plans are iterator trees
- Iterator = basic unit of query plan execution
- Each iterator has 0, 1, 2, or N children
- Core methods implemented by any iterator:
  - Open
  - GetRow
  - Close
- Control flows down the tree
- Data flows (is pulled) up the tree

Types of iterators

- Scan and seek
- Joins
- Aggregates
- Sorts
- Spools
- Parallelism
- Insert, update, and delete
- Top
- Compute scalar
- Filter
- Concatenation
- Sequence

Too many iterators to cover in a single talk!

Properties of iterators

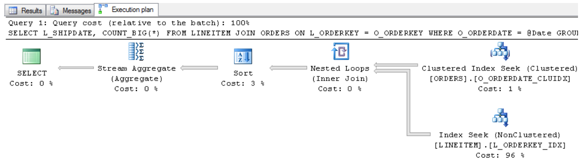
- Memory consuming
  - If usage is proportional to size of input set
- Stop and go?
  - May affect performance for top or fast N queries
  - Defines “phases” for memory grants
- Dynamic
  - Supports special methods for dynamic cursors:
    - Can save and restore position even after caching
    - Support forward and backward scans
    - Support acquiring scroll locks
  - It is not always possible to find a dynamic query plan; if the optimizer cannot find a dynamic plan, it downgrades the cursor type to keyset

Showplan

- Displays the query plan
- Great tool …
  - For understanding what SQL Server is doing
  - For diagnosing many performance issues
- Lots of stats …
  - Estimated and actual row counts
  - Relative iterator costs
- Graphical, text, and XML versions
  - XML is new in SQL Server 2005
  - Text has been deprecated

Graphical vs. text vs. XML plans

- Graphical
  - Nice looking icons with helpful tooltips
  - Quick view of the “big picture” (but sometimes too big!)
  - Easily identify costliest iterators
  - Provides detailed help on each iterator
  - Cannot really be saved in SQL Server 2000; fixed in SQL Server 2005



- Text
  - May be easier to read for big plans
  - Searchable with simple text tools
  - All data visible at once; no pointing/tool tips
  - Easy to save or export into Excel
  - Easy to compare estimated and actual row counts

```
|--Stream Aggregate
     |--Sort
          |--Nested Loops
               |--Clustered Index Seek
               |--Index Seek
```

- XML
  - Basis for graphical plans in SQL Server 2005
  - Most detailed information
  - Harder to read than text or graphical plans
  - Great for automated tools
  - Can be loaded in XML column and searched using XQuery
  - Used by USE PLAN hint and plan guides

```
…
  <RelOp NodeId="0" …>
    <StreamAggregate>
      <RelOp NodeId="1" …>
        <Sort …>
          …
```

Reading plans

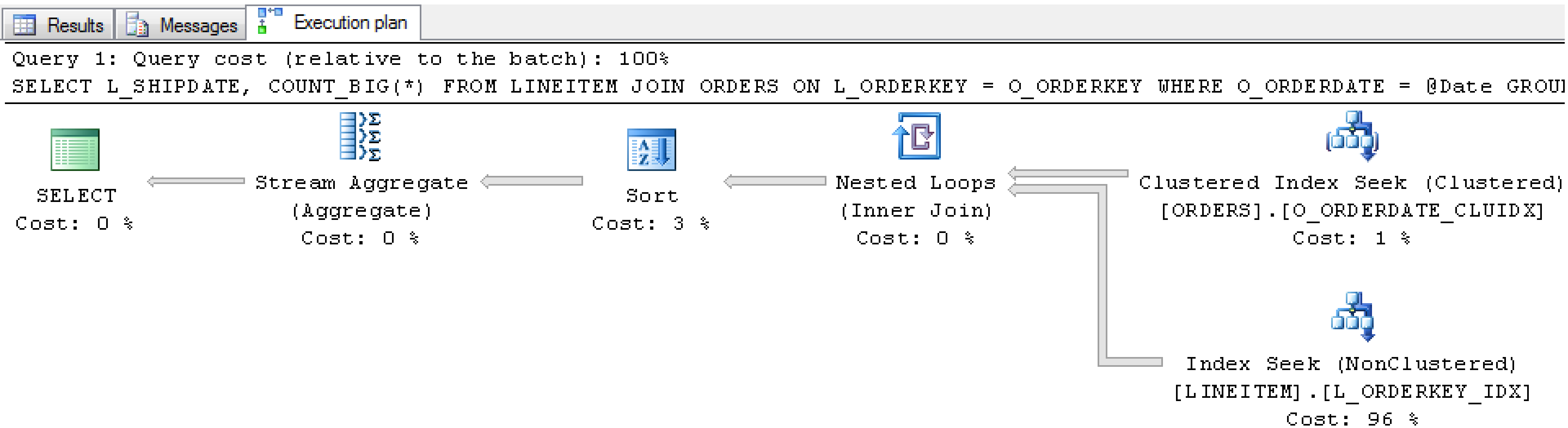
- Graphical
  - Each icon represents one iterator in the tree
  - Tree structure and data flow are represented by arrows connecting the icons
  - More information is available in the tooltip and in the “properties” pane
- Text
  - Each line represents one iterator in the tree
  - Root of iterator tree is the first line
  - Children are indented and connected to parent by vertical bars
  - Data flows up the vertical bars
  - All details on one line
- XML
  - One element per iterator plus additional elements for other information
  - Tree structure is represented by nesting of elements
  - Data flows toward outermost elements

Showplan examples

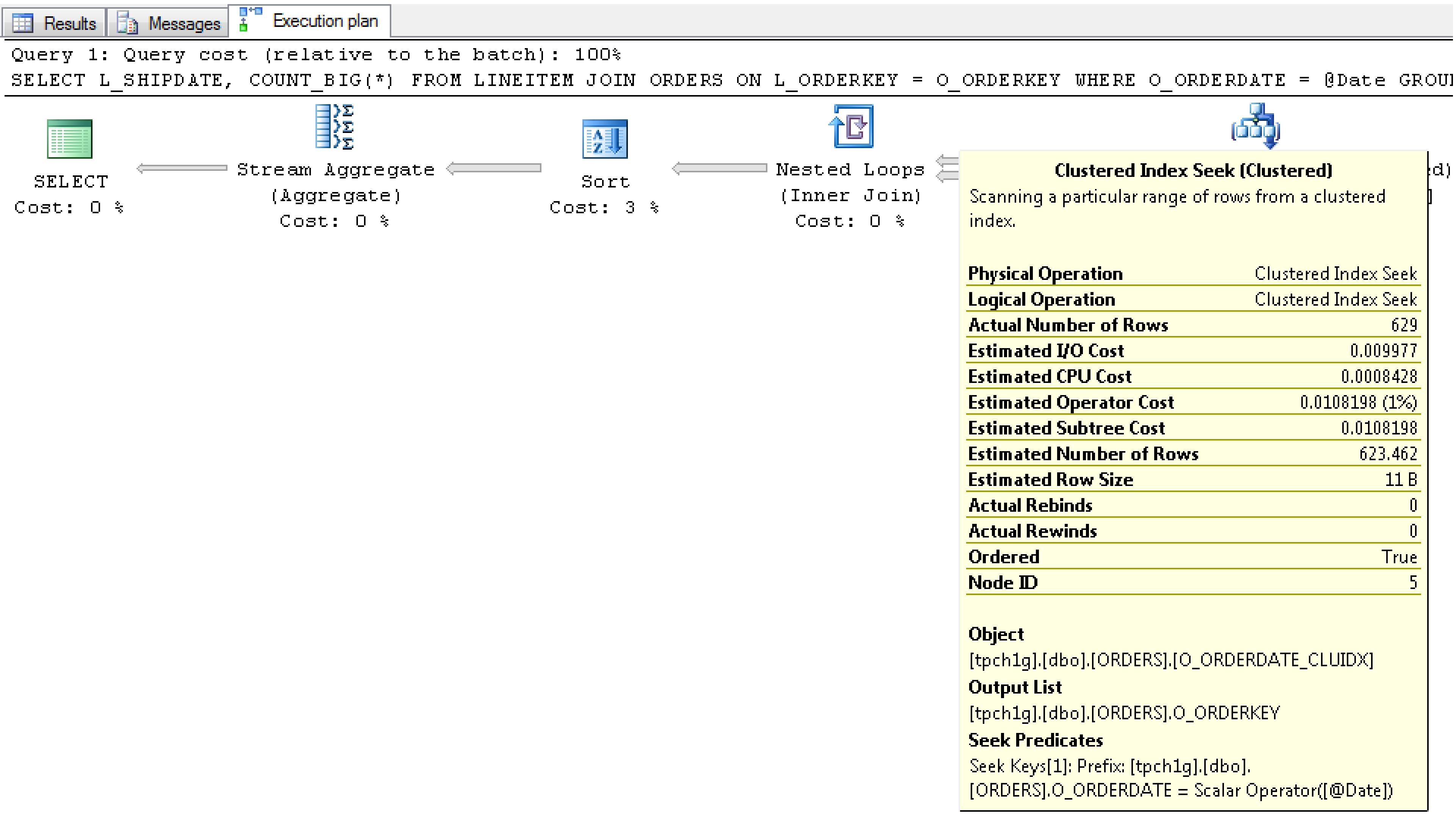
```
DECLARE @Date DATETIME
SET @Date = '1996-07-04'

SELECT L_SHIPDATE, COUNT_BIG(*)
FROM LINEITEM JOIN ORDERS ON L_ORDERKEY = O_ORDERKEY
WHERE O_ORDERDATE = @Date
GROUP BY L_SHIPDATE
ORDER BY L_SHIPDATE
OPTION (OPTIMIZE FOR (@Date = '1996-03-15'))
```

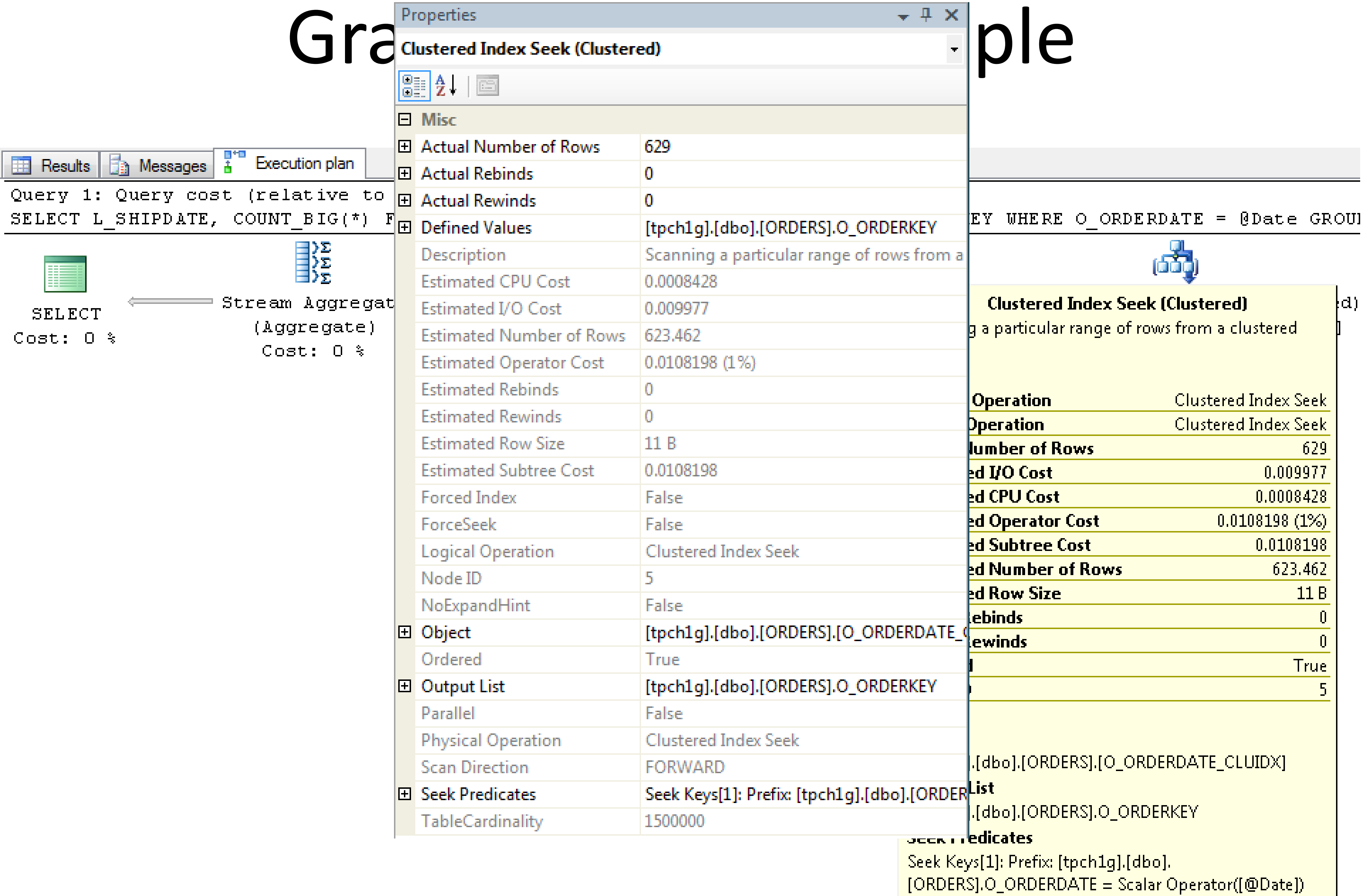
Graphical plan example



Graphical plan example

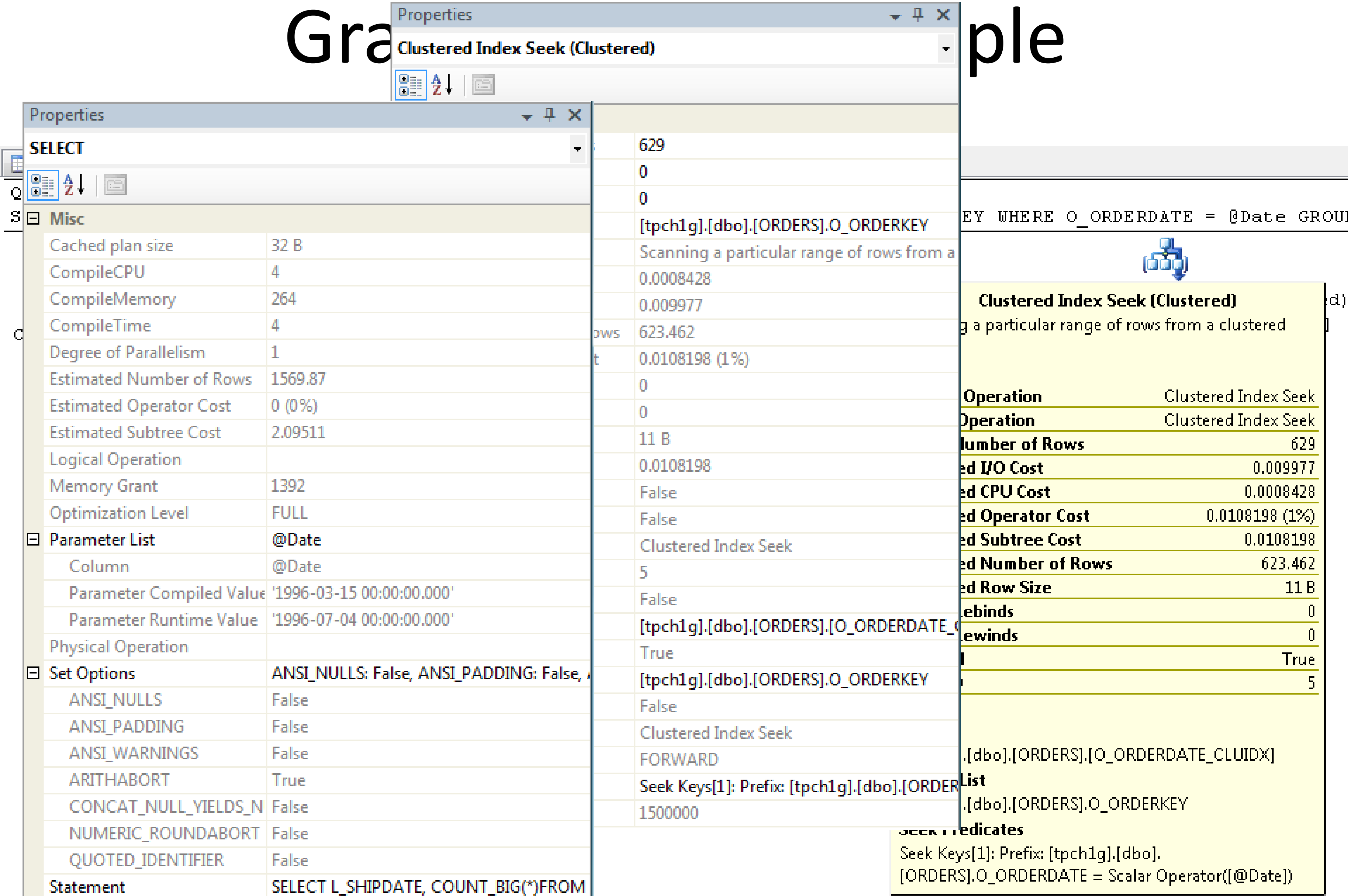


Gra ple



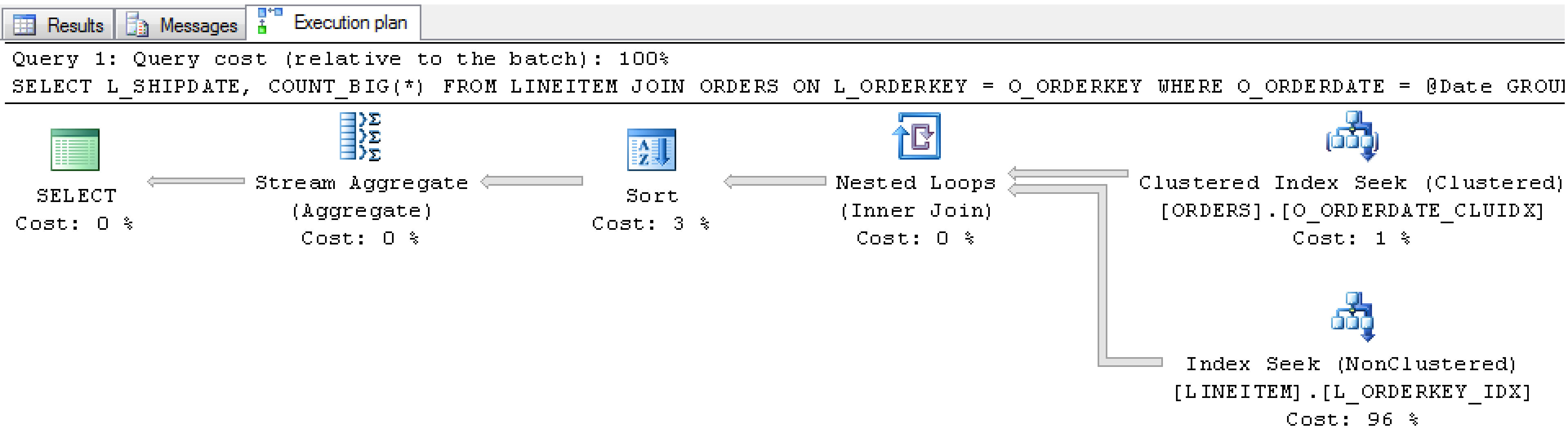
Properties
Clustered Index Seek (Clustered)
Misc
Actual Number of Rows 629
Actual Rebinds 0
Actual Rewinds 0
Defined Values [tpch1g].[dbo].[ORDERS].O_ORDERKEY
Description Scanning a particular range of rows from a
Estimated CPU Cost 0.0008428
Estimated I/O Cost 0.009977
Estimated Number of Rows 623.462
Estimated Operator Cost 0.0108198 (1%)
Estimated Rebinds 0
Estimated Rewinds 0
Estimated Row Size 11 B
Estimated Subtree Cost 0.0108198
Forced Index False
ForceSeek False
Logical Operation Clustered Index Seek
Node ID 5
NoExpandHint False
Object [tpch1g].[dbo].[ORDERS].[O_ORDERDATE_C
Ordered True
Output List [tpch1g].[dbo].[ORDERS].O_ORDERKEY
Parallel False
Physical Operation Clustered Index Seek
Scan Direction FORWARD
Seek Predicates Seek Keys[1]: Prefix: [tpch1g].[dbo].[ORDER
TableCardinality 1500000
Results
Messages
Execution plan
Query 1: Query cost (relative to
SELECT L_SHIPDATE, COUNT_BIG(*) F
EY WHERE O_ORDERDATE = @Date GROU
SELECT
Cost: 0 %
Stream Aggregat
(Aggregate)
Cost: 0 %
Clustered Index Seek (Clustered)
g a particular range of rows from a clustered
Operation Clustered Index Seek
Operation Clustered Index Seek
Number of Rows 629
ed I/O Cost 0.009977
ed CPU Cost 0.0008428
ed Operator Cost 0.0108198 (1%)
ed Subtree Cost 0.0108198
ed Number of Rows 623.462
ed Row Size 11 B
ebinds 0
ewinds 0
True
5
.[dbo].[ORDERS].[O_ORDERDATE_CLUIDX]
List
.[dbo].[ORDERS].O_ORDERKEY
edicates
Seek Keys[1]: Prefix: [tpch1g].[dbo].
[ORDERS].O_ORDERDATE = Scalar Operator([@Date])

Gra ple



Properties
Clustered Index Seek (Clustered)
629
0
0
[tpch1g].[dbo].[ORDERS].O_ORDERKEY
Scanning a particular range of rows from a
0.0008428
0.009977
623.462
0.0108198 (1%)
0
0
11 B
0.0108198
False
False
Clustered Index Seek
5
False
[tpch1g].[dbo].[ORDERS].[O_ORDERDATE_
True
[tpch1g].[dbo].[ORDERS].O_ORDERKEY
False
Clustered Index Seek
FORWARD
Seek Keys[1]: Prefix: [tpch1g].[dbo].[ORDER
1500000
Properties
SELECT
Misc
Cached plan size 32 B
CompileCPU 4
CompileMemory 264
CompileTime 4
Degree of Parallelism 1
Estimated Number of Rows 1569.87
Estimated Operator Cost 0 (0%)
Estimated Subtree Cost 2.09511
Logical Operation
Memory Grant 1392
Optimization Level FULL
Parameter List @Date
Column @Date
Parameter Compiled Value '1996-03-15 00:00:00.000'
Parameter Runtime Value '1996-07-04 00:00:00.000'
Physical Operation
Set Options ANSI_NULLS: False, ANSI_PADDING: False,
ANSI_NULLS False
ANSI_PADDING False
ANSI_WARNINGS False
ARITHABORT True
CONCAT_NULL_YIELDS_N False
NUMERIC_ROUNDABORT False
QUOTED_IDENTIFIER False
Statement SELECT L_SHIPDATE, COUNT_BIG(*)FROM
EY WHERE O_ORDERDATE = @Date GROU
Clustered Index Seek (Clustered)
g a particular range of rows from a clustered
Operation Clustered Index Seek
Operation Clustered Index Seek
lumber of Rows 629
ed I/O Cost 0.009977
ed CPU Cost 0.0008428
ed Operator Cost 0.0108198 (1%)
ed Subtree Cost 0.0108198
ed Number of Rows 623.462
ed Row Size 11 B
ebinds 0
ewinds 0
True
5
.[dbo].[ORDERS].[O_ORDERDATE_CLUIDX]
List
.[dbo].[ORDERS].O_ORDERKEY
Seek Keys[1]: Prefix: [tpch1g].[dbo].
[ORDERS].O_ORDERDATE = Scalar Operator([@Date])

Text plan example




```
|--Stream Aggregate(GROUP BY:([L_SHIPDATE]) DEFINE:([Expr1008]=Count(*)))
     |--Sort(ORDER BY:([L_SHIPDATE] ASC))
          |--Nested Loops(Inner Join, OUTER REFERENCES:([ORDERS].[O_ORDERKEY], …) …)
               |--Clustered Index Seek(OBJECT:([ORDERS].[O_ORDERDATE_CLUIDX]), SEEK:([O_ORDERDATE]=[@Date]) …)
               |--Index Seek(OBJECT:([LINEITEM].[L_ORDERKEY_IDX]), SEEK:([L_ORDERKEY]=[O_ORDERKEY]) …)
```

XML plan example

```
<ShowPlanXML xmlns="http://schemas.microsoft.com/…" Version="1.0" Build="10.0.…">
  <BatchSequence>
    <Batch>
      <Statements>
        <StmtSimple StatementText="SELECT …" StatementId="1" StatementCompId="2" …>
          <StatementSetOptions QUOTED_IDENTIFIER="false" ARITHABORT="true" … />
          <QueryPlan DegreeOfParallelism="1" MemoryGrant="1392" …>
            <RelOp …>
              …
            </RelOp>
            <ParameterList>
              <ColumnReference Column="@Date"
                               ParameterCompiledValue="'1996-03-15 …'"
                               ParameterRuntimeValue="'1996-07-04 …'" />
            </ParameterList>
          </QueryPlan>
        </StmtSimple>
      </Statements>
    </Batch>
  </BatchSequence>
</ShowPlanXML>
```

XML plan example

```
<RelOp NodeId="0" PhysicalOp="Stream Aggregate" LogicalOp="Aggregate" …>
  <StreamAggregate>
    <RelOp NodeId="1" PhysicalOp="Sort" LogicalOp="Sort" …>
      <MemoryFractions Input="0.782609" Output="1" />
      <Sort Distinct="0">
        <RelOp NodeId="2" PhysicalOp="Nested Loops" LogicalOp="Inner Join" …>
          <NestedLoops Optimized="1" WithUnorderedPrefetch="1">
            <RelOp NodeId="5" PhysicalOp="Clustered Index Seek" …>
              <IndexScan Ordered="1" ScanDirection="FORWARD" …>
                <Object … Table="[ORDERS]" Index="[O_ORDERDATE_CLUIDX]" … />
              </IndexScan>
            </RelOp>
            <RelOp NodeId="6" PhysicalOp="Index Seek" …>
              <IndexScan Ordered="1" ScanDirection="FORWARD" …>
                <Object … Table="[LINEITEM]" Index="[L_ORDERKEY_IDX]" … />
              </IndexScan>
            </RelOp>
          </NestedLoops>
        </RelOp>
      </Sort>
    </RelOp>
  </StreamAggregate>
</RelOp>
```

XML plan example

```
<RelOp NodeId="5"
       PhysicalOp="Clustered Index Seek" LogicalOp="Clustered Index Seek"
       EstimateRows="623.462" …>
  <OutputList>
    <ColumnReference … Table="[ORDERS]" Column="O_ORDERKEY" />
  </OutputList>
  <RunTimeInformation>
    <RunTimeCountersPerThread Thread="0" ActualRows="629" … ActualExecutions="1" />
  </RunTimeInformation>
  <IndexScan Ordered="1" ScanDirection="FORWARD"
             ForcedIndex="0" ForceSeek="0" NoExpandHint="0">
    <DefinedValues>
      <DefinedValue>
        <ColumnReference … Table="[ORDERS]" Column="O_ORDERKEY" />
      </DefinedValue>
    </DefinedValues>
    <Object … Table="[ORDERS]" Index="[O_ORDERDATE_CLUIDX]" IndexKind="Clustered" />
    <SeekPredicates>
      …
    </SeekPredicates>
  </IndexScan>
</RelOp>
```

Text/XML plan options

	Command	Execute Query?	Display Estimated Row Counts & Stats	Display Actual Row Counts
Text plan	SET SHOWPLAN_TEXT ON	No	No	No
	SET SHOWPLAN_ALL ON	No	Yes	No
	SET STATISTICS PROFILE ON	Yes	Yes	Yes
XML plan	SET SHOWPLAN_XML ON	No	Yes	No
	SET STATISTICS XML ON	Yes	Yes	Yes

SQL Profiler and DMVs can also output plans

Common iterators

- Scans and seeks
- Join iterators
  - Nested loops join
  - Merge join
  - Hash join
- Aggregation iterators
  - Stream aggregrate
  - Hash aggregate

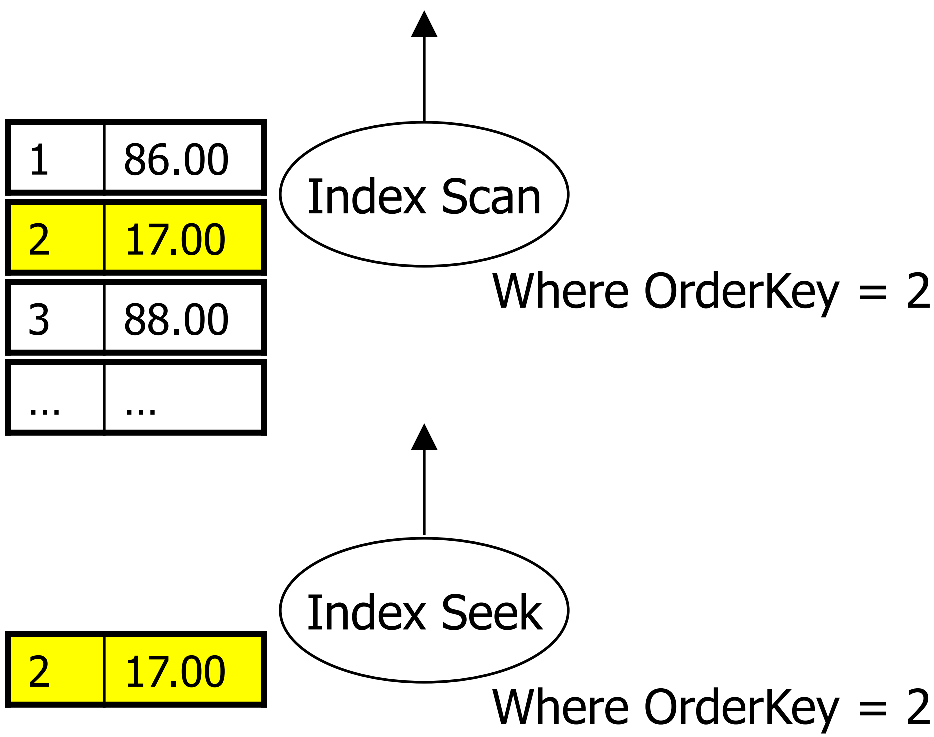
- Iterators are not “good” or “bad”
- There is no “best” join or aggregate type
- Each iterator works well in the right scenarios

Scans and seeks

- Scans return the entire table or index
  - Index scans may be ordered or unordered
- Seeks efficiently return rows from one or more ranges of an index
  - Index seeks are always ordered

Index scan vs. index seek

Select Price from Orders where OrderKey = 2



OrderKey	**Price**
1	86.00
2	17.00
3	88.00
4	17.00
5	96.00
6	22.00
7	74.00
8	94.00
9	56.00
…	…

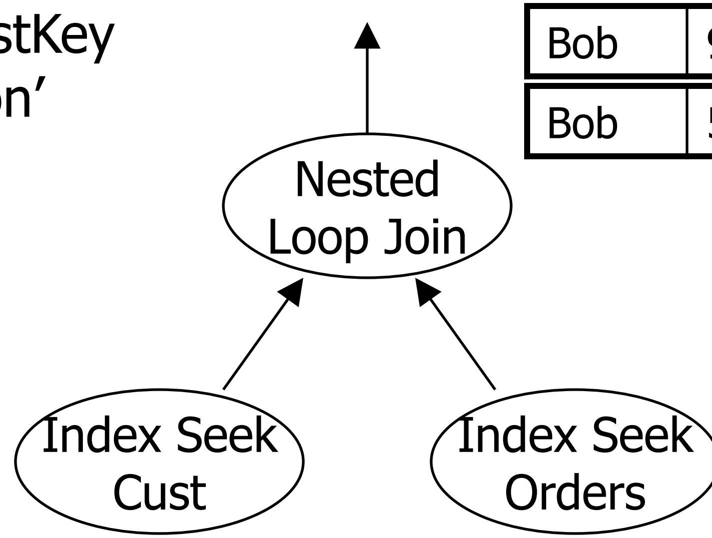
Nested loops join

- Basic algorithm:
  1. Get one row from the left input
  2. Get all rows from the right input that correspond to the row from the left input
  3. When there are no more matching rows from the right input, get the next row from the left input and repeat
- Correlated parameters
  - Data from the left input affects the data returned by the right input (i.e., step 2 depends on step 1)
  - Do not need any correlated parameters to join; if we do not have any, every execution of the right input produces the same result

Nested loops join example

Select C.Name, O.Price
from Cust C join Orders O
on O.CustKey = C.CustKey
where C.City = ‘Boston’

Alice	88.00
Bob	86.00
Bob	94.00
Bob	56.00



4	Boston	Alice
5	Boston	Bob
6	Boston	Cathy

Where C.City = ‘Boston’

On O.CustKey = C.CustKey

4	88.00

5	86.00
5	94.00
5	56.00

No rows match CustKey = 6

Nested loops join

- Only join type …
  - That supports inequality predicates
  - That supports dynamic cursors
- Right input may be a simple index seek or a complex subplan
- Optimizations:
  - Use indexes to optimize the selection of matching right input rows (also known as an index join)
  - Use lazy spool on the right input if we expect duplicate left input rows
- Performance tips:
  - Cost is proportional to the **product** of the left and right input cardinalities
  - Generally performs best for small left input sets
  - Create an index to change Cartesian product into an index join
  - Watch out for large numbers of random I/Os
- Also used for bookmark lookups in SQL Server 2005 and 2008

Index columns

- Key columns:
  - Set of columns that can be used in a seek
  - For a composite index, the order of the columns matters:
    - Determines the sort order for the index
    - Can only seek on a column if all prior columns have equality predicates
  - Non-unique non-clustered index on a table with a clustered index implicitly includes the clustered index keys
- Covered columns:
  - Set of columns that can be output by a seek or scan of the index
  - Heap or clustered index always covers all columns
  - Non-clustered index covers the key columns for the index and, if the table has a clustered index, the clustered index keys
  - Can add more columns using the CREATE INDEX INCLUDE clause

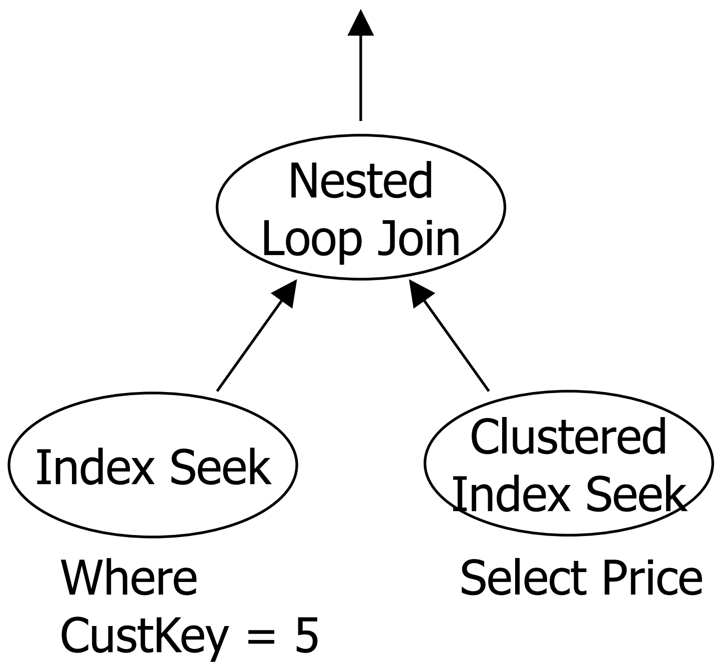
Bookmark lookup

- Question:
  - What happens if the best non-clustered index for a seek does not cover all of the columns required by the query?
- Answer:
  - Look up the extra columns in the heap or clustered index
  - This operation is known as a bookmark lookup
- SQL Server 2000 had a bookmark lookup iterator
- SQL Server 2005 and 2008 do not have a bookmark lookup iterator
  - Instead, they simply join the non-clustered index to the clustered index using a nested loops join
  - To see whether a join is a bookmark lookup, check for the “LOOKUP” keyword or attribute on the clustered index seek
  - Bookmark lookup introduces random I/Os: there is a performance tradeoff between a scan and a seek with a bookmark lookup

Bookmark lookup example

Select Price from Orders where CustKey = 5

1	5	86.00
8	5	94.00
9	5	56.00



1	5
8	5
9	5

OrderKey	**CustKey**	**Price**
1	5	86.00
2	2	17.00
3	4	88.00
4	9	17.00
5	1	96.00
6	7	22.00
7	8	74.00
8	5	94.00
9	5	56.00
…	…	…

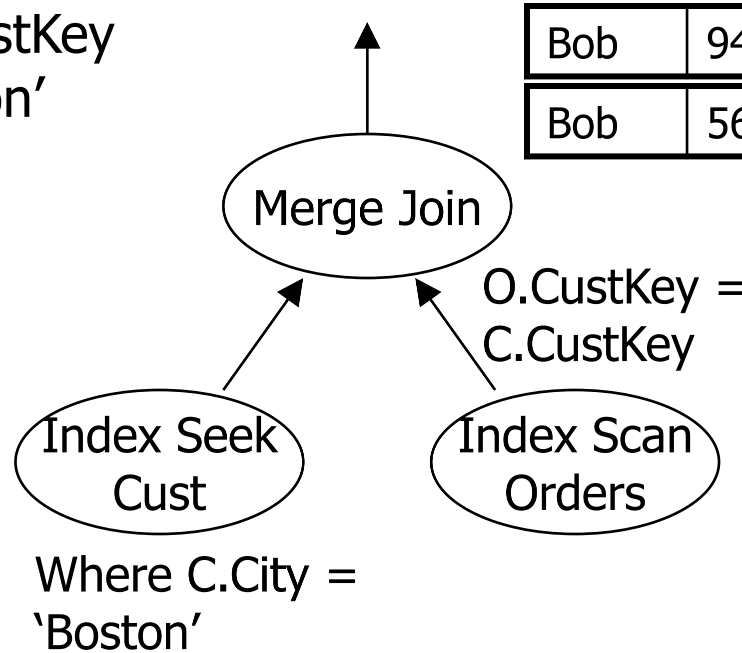
Clustered index on OrderKey
Non-clustered index on Custkey

Merge join

- Requires at least one equijoin predicate
- Data must be sorted on the join keys
  - Sort order may be provided by an index
  - Or, plan may include an explicit sort
- Basic algorithm:
  1. Get one row from both the left and right inputs
  2. If the rows match, return the joined row
  3. Otherwise, get a new row from whichever input is smaller and repeat

Merge join example

Select C.Name, O.Price
from Cust C join Orders O
on O.CustKey = C.CustKey
where C.City = 'Boston'



Alice	88.00
Bob	86.00
Bob	94.00
Bob	56.00

4	Boston	Alice
5	Boston	Bob
6	Boston	Cathy

2	17.00
4	88.00
5	86.00
5	94.00
5	56.00
7	22.00

Cust and Orders indexes ordered by CustKey

Merge join

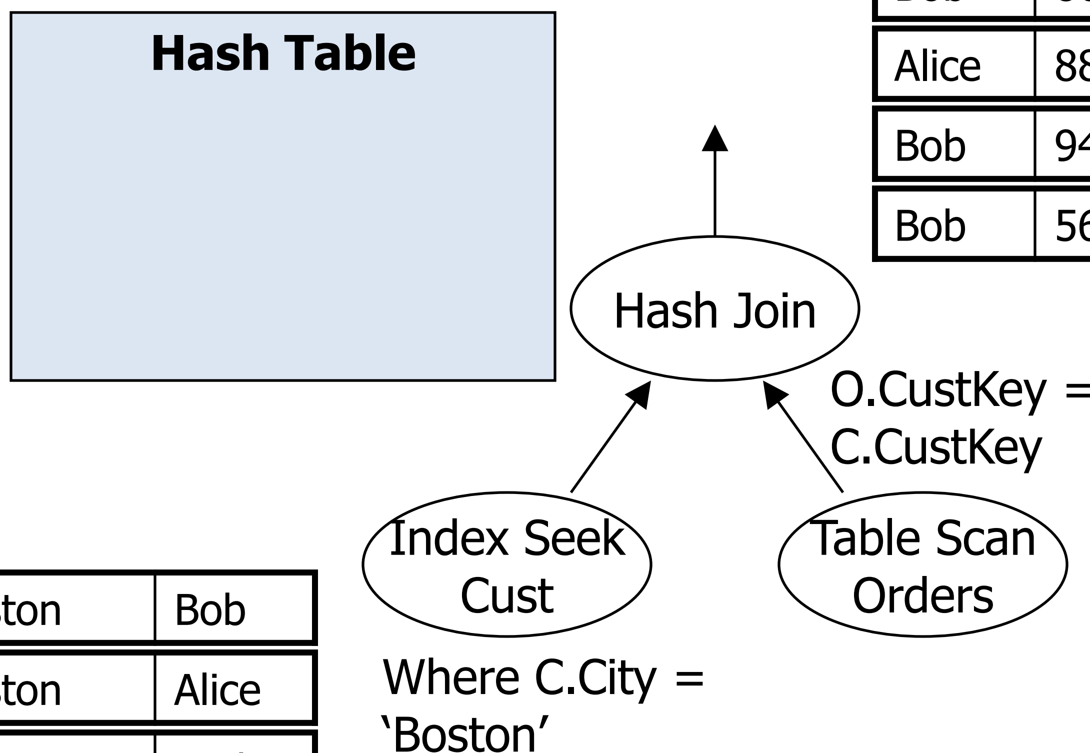
- Optimizations:
  - One to many join
  - Inner join terminates as soon as either input exhausted
- Performance tips:
  - Cost is proportional to the **sum** of the input cardinalities
  - Performs well for small and large input sets especially if sort order is provided by an index
  - If a merge join plan includes explicit sorts, watch out for spilling (see the SQL Profiler sort warning event class)
  - Does not parallelize as well as a hash join

Hash join

- Requires at least one equijoin predicate
- Basic algorithm:
  1. Get all rows from the left input
  2. Build an in-memory hash table using left input rows
  3. Get all rows from the right input
  4. Probe the hash table for matches
- Requires memory to build the hash table
- If the join runs out of memory, portions of the left and right inputs must be spilled to disk and handled in a separate pass

Hash join example

Select C.Name, O.Price
from Cust C join Orders O
on O.CustKey = C.CustKey
where C.City = ‘Boston’



Bob	86.00
Alice	88.00
Bob	94.00
Bob	56.00

5	Boston	Bob
4	Boston	Alice
6	Boston	Cathy

5	86.00
2	17.00
4	88.00
9	17.00
5	94.00
5	56.00

Order of Cust and Orders tables does not matter

Hash join

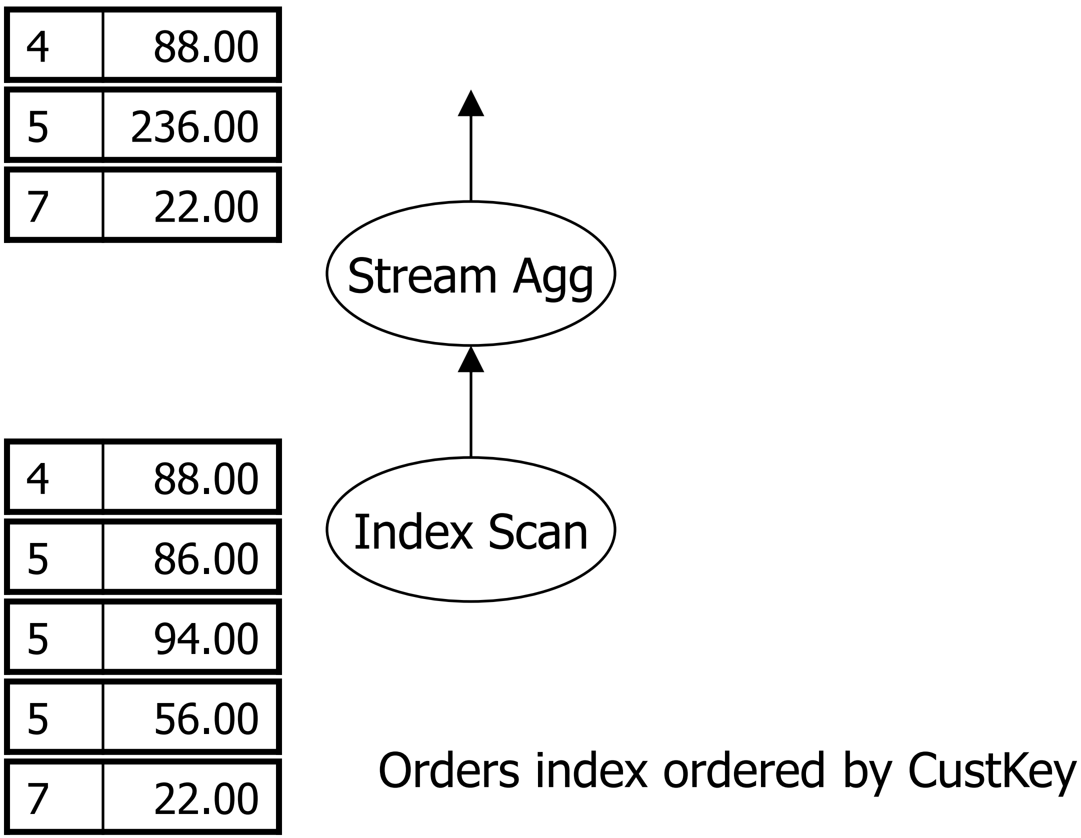
- Is stop and go on the left input
- Optimizations:
  - Build the hash table on the smaller input
  - If the join spills, may switch build and probe inputs
  - Use a bitmap to discard right input rows quickly
- Performance tip:
  - Cost is proportional to the **sum** of the input cardinalities
  - Generally performs well for larger input sets
  - Parallel hash join scales well
  - Watch out for spilling – especially multiple passes or "bailout" (see the SQL Profiler hash warning event class)

Stream aggregate

- Data must be sorted on group by keys
- Sorting groups rows with matching keys together
- Processes groups one at a time
- Does not block or use memory
- Efficient if sort order is provided by an index or (in some cases) if the plan needs to sort anyhow
- Only option for scalar aggregates (i.e., no group by)

Stream aggregate example

Select CustKey, sum(Price) from Orders group by CustKey

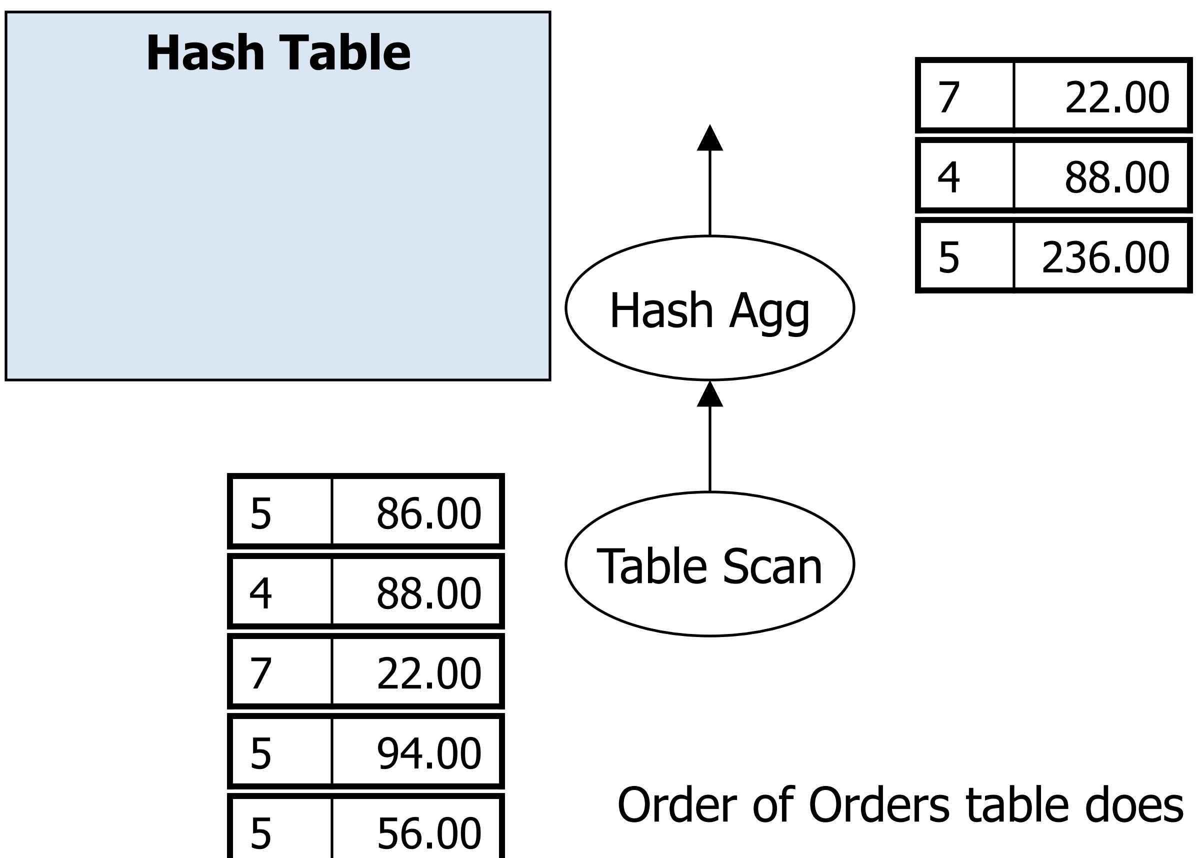


Hash aggregate

- Data need not be sorted
- Builds a hash table of all groups
- Stop and go
- Like hash join:
  - Requires memory; may spill to disk if it runs out
  - Generally better for larger input sets
  - Parallel hash aggregate scales well
  - Watch out for spilling (the hash warning event class)
- Duplicates key values …
  - Can be bad for a hash join because it is not possible to subdivide a hash bucket that contains all duplicates
  - Are good for a hash aggregate because duplicates collapse into a single hash table entry

Hash aggregate example

Select CustKey, sum(Price) from Orders group by CustKey



Order of Orders table does not matter

Performance tips

- Watch out for errors in cardinality estimates
  - Errors propagate upwards; look for the root cause
  - Make sure statistics are up to date and accurate
  - Avoid excessively complex predicates
  - Use computed columns for overly complex expressions
- General tips:
  - Use set based queries; (almost always) avoid cursors
  - Avoid joining columns with mismatched data types
  - Avoid unnecessary outer joins, cross applies, complex sub-queries, dynamic index seeks, …
  - Avoid dynamic SQL (but beware that sometimes dynamic SQL does yield a better plan)
  - Consider creating constraints (but remember that there is a cost to maintain constraints)
  - If possible, use inline TVFs NOT multi-statement TVFs
  - Use SET STATISTICS IO ON to watch out for large numbers of physical I/Os
  - Use indexes to workaround locking, concurrency, and deadlock issues
- OLTP tips:
  - Avoid memory consuming or blocking iterators
  - Use seeks not scans
- DW tips:
  - Use parallel plans
  - Watch out for skew in parallel plans
  - Avoid order preserving exchanges

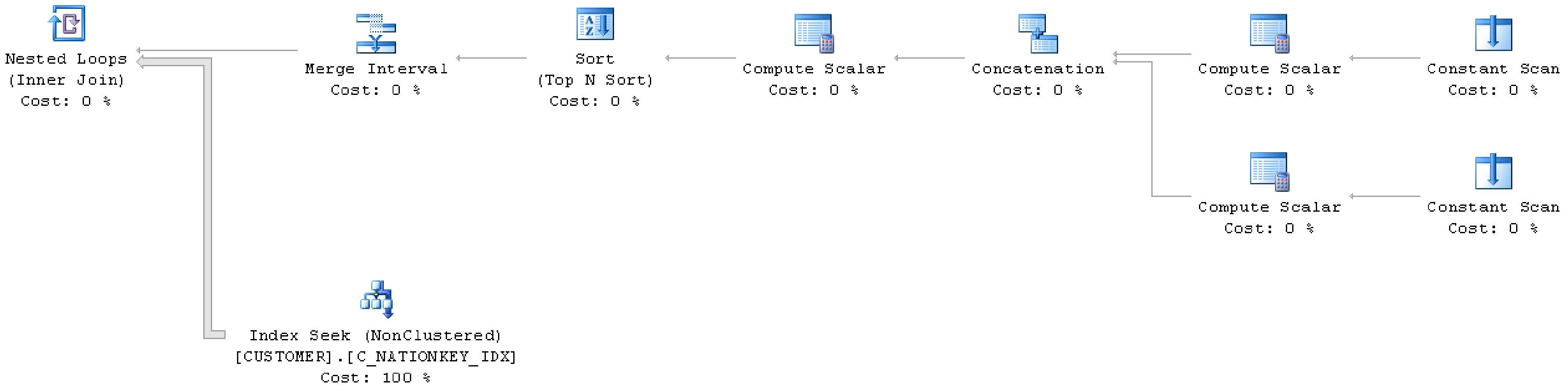
Other interesting plans

- Static vs. dynamic index seeks
- Insert, update, and delete plans
  - Per row vs. per index updates
  - Split sort collapse updates

Static vs. dynamic index seeks

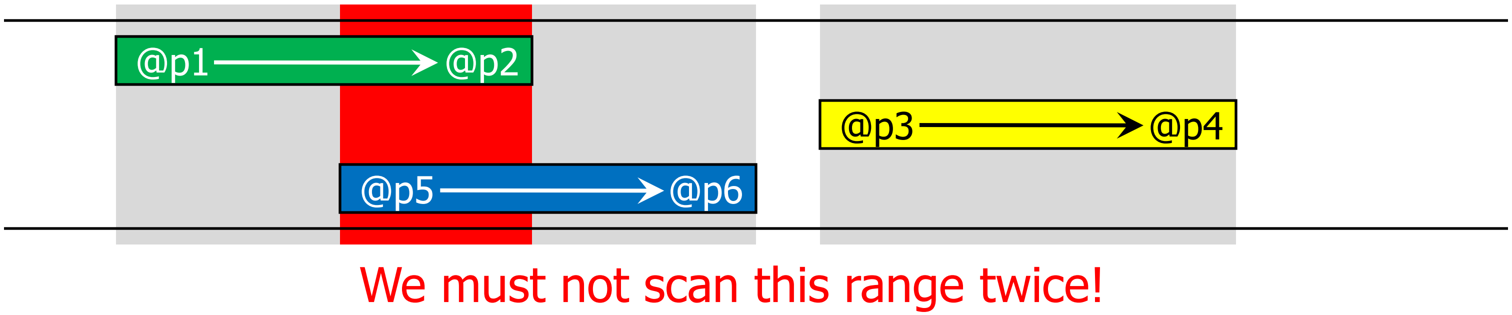
- Static index seeks
  - Ranges are known to be non-overlapping at compile time
  - Standalone index seek iterator
- Dynamic index seeks
  - Ranges may overlap at run time
  - Typically needed due to OR’ed predicates with T-SQL or correlated parameters:
    - … where State = @p1 or State = @p2
  - Sort and merge (using the merge interval iterator) the ranges at runtime as appropriate

Dynamic index seek plan



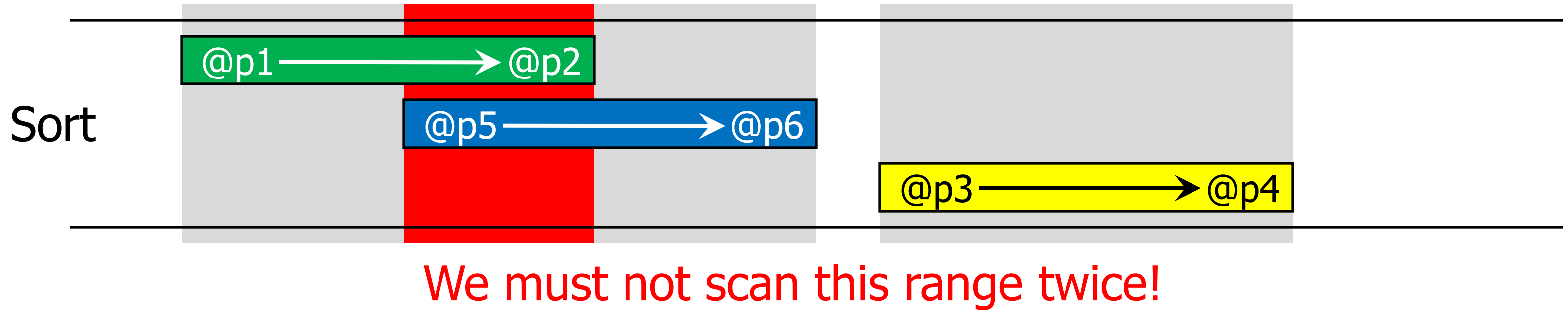
Merge interval example

Select * from T
where [C] between @p1 and @p2 or
[C] between @p3 and @p4 or
[C] between @p5 and @p6



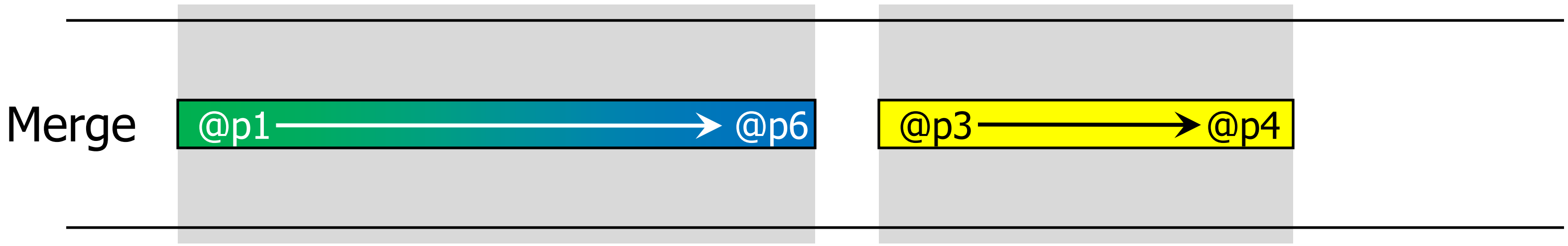
Merge interval example

Select * from T
where [C] between @p1 and @p2 or
[C] between @p3 and @p4 or
[C] between @p5 and @p6



Merge interval example

Select * from T
where [C] between @p1 and @p2 or
[C] between @p3 and @p4 or
[C] between @p5 and @p6



Each unique range scanned only once!

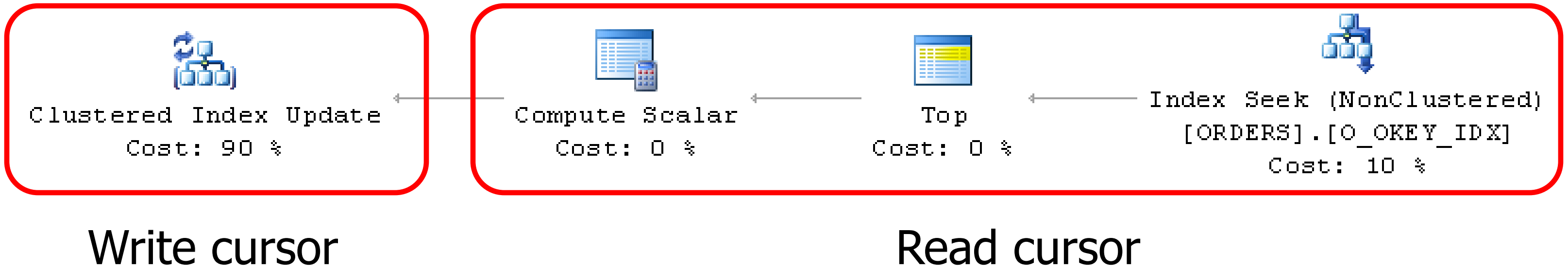
Insert, update, and delete plans

- All update plans have two parts:
  - Read cursor returns rows to insert, update, or delete
  - Write cursor
    - Executes the insert, update, or delete
    - Maintains non-clustered indexes
    - And checks constraints, maintains indexed views, …
- One update iterator handles most cases
- Special optimized leaf iterators for
  - Insert … values (…)
  - Updates to clustered indexes if …
    - Using a clustered index seek and …
    - Updating the clustering key or updating at most one row

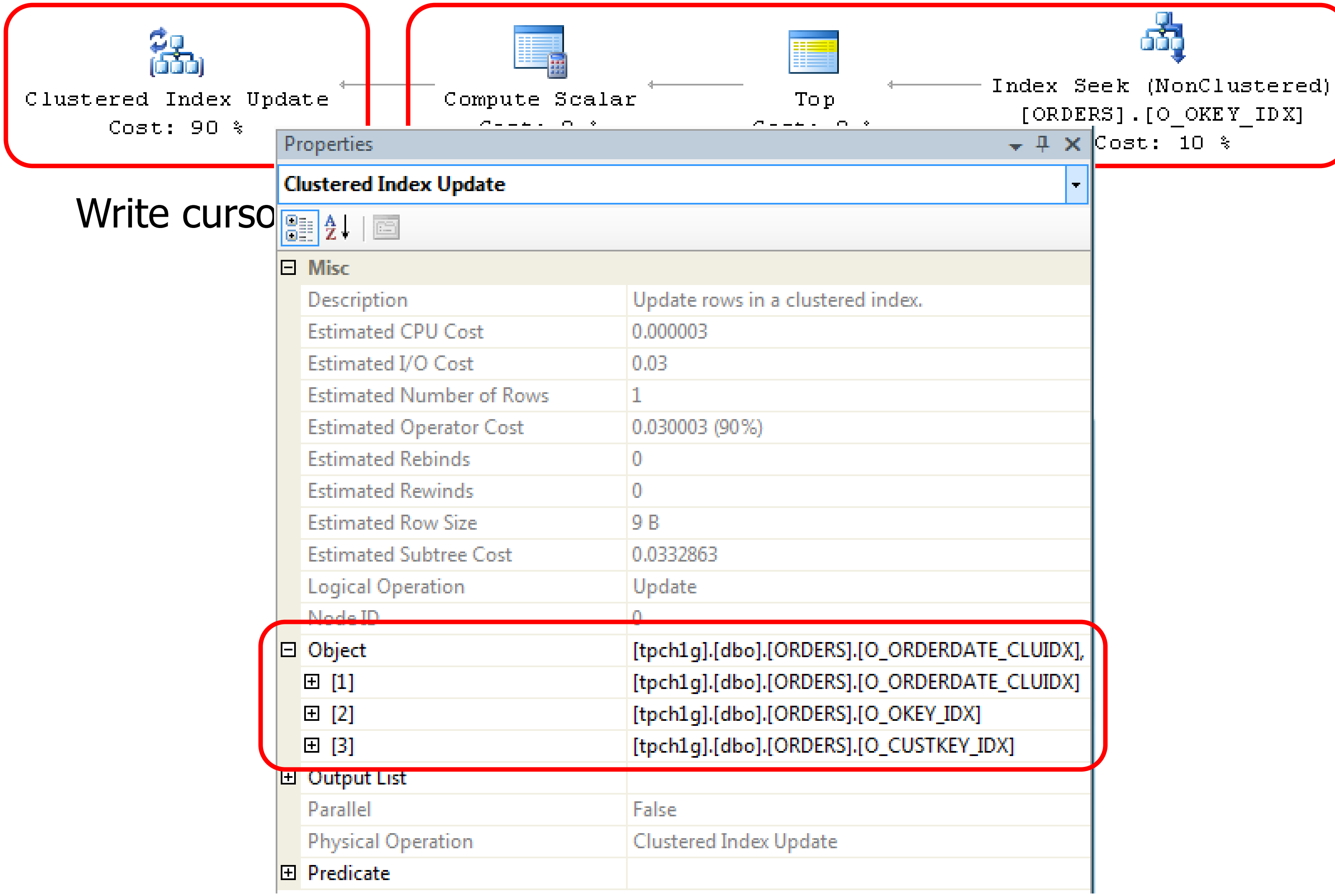
Per row vs. per index updates

- Per row plans:
  - A single update iterator maintains all indexes (including the heap or clustered index and all affected non-clustered indexes)
  - Reads one input row at a time then modifies all affected indexes
- Per index plans:
  - The plan has a separate update iterator for each affected index
  - Each update iterator maintains only one index
  - Reads and spools all input rows before modifying any indexes
  - Applies all modifications to one index at a time
- Why per index?
  - For performance of large updates (e.g., sort on index key)
  - For correctness of updates to unique indexes

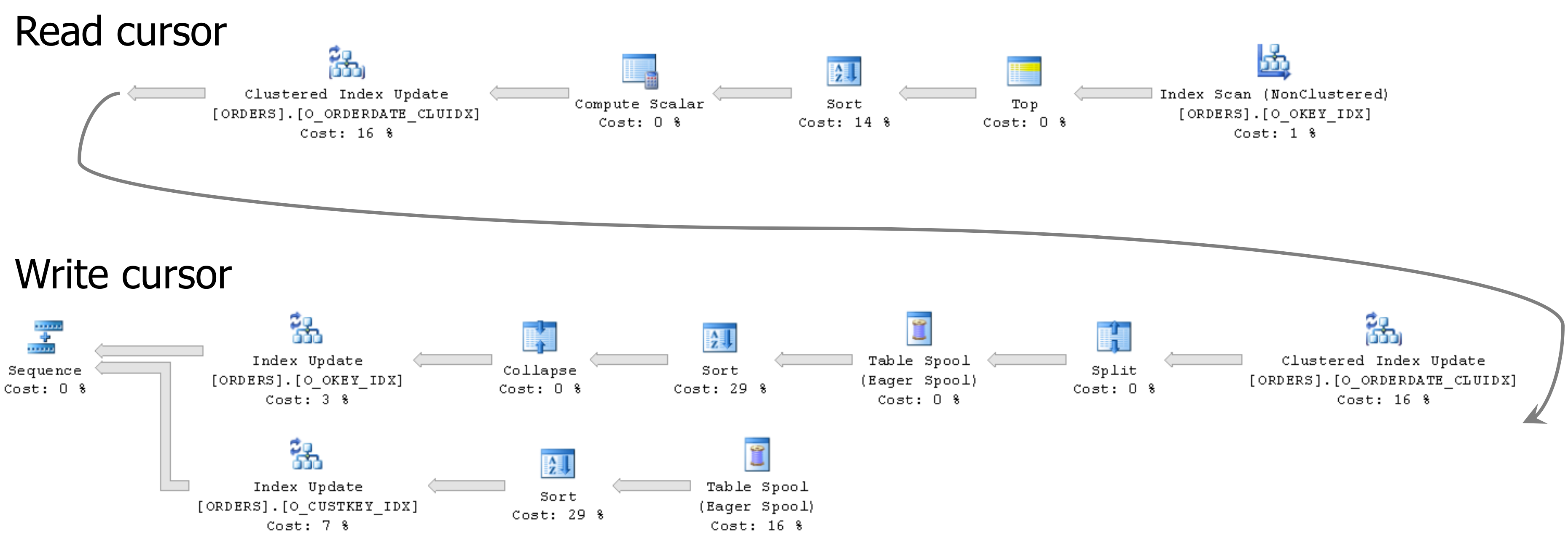
Per row update plan



Per row update plan



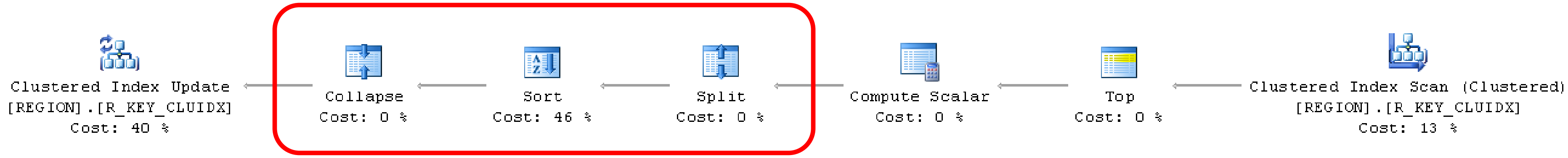
Per index update plan



Split sort collapse updates

Update T set UniqCol = UniqCol + 1

- Must ensure that updates to unique indexes do not fail from “false” uniqueness violations
- The split, sort, and collapse iterators reorganize the stream of rows to update to guarantee that there are no “false” uniqueness violations

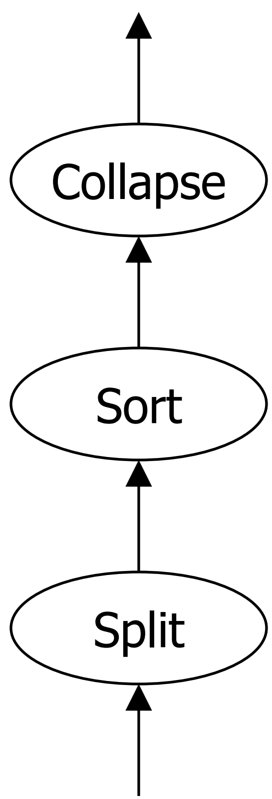


Split sort collapse example

Update T set UniqCol = UniqCol + 1

UniqCol	**Data**
1	X
2	Y

	Old	New
Del 1	X	
Upd 2	Y	X
Ins 3		Y



Del	1	X
Del	2	Y
Ins	2	X
Ins	3	Y

Del	1	X
Ins	2	X
Del	2	Y
Ins	3	Y

Old	New	
1	2	X
2	3	Y

Split sort collapse updates

- Requires a per index plan as the data is reorganized specifically for one index
- May also help performance by transforming key value updates into “in place” updates

Summary

- Iterators are the basic building blocks of query execution and query plans
- Showplan let’s you analyze query plans
  - Graphical
  - Text
  - XML
- Scan vs. seek vs. bookmark lookup
- Three join iterators:
  - Nested loops join
  - Merge join
  - Hash join
- Two aggregation iterators:
  - Stream aggregate
  - Hash aggregate
- Performance tips
- More complex plans:
  - Dynamic index seeks
  - Update plans including unique column updates

Questions?

- Books …
  - Inside SQL Server 2005: Query Tuning and Optimization
- Blogs …
  - http://blogs.msdn.com/craigfr
  - http://blogs.msdn.com/sqlqueryprocessing
  - And many more …
- Other resources …
  - Books online
  - MSDN
  - Newsgroups and forums
  - Web search