INTELLIGENT QUERY PROCESSING IN SQL SERVER 2019
Miloš Radivojević

Data Platform MVP
Principal Database Consultant, bwin GVC Vienna, Austria
Co-Founder: SQL Pass Austria
Conference Speaker, Book Author

Contact:
MRadivojevic@gvcgroup.com
www.bwinparty.com
Twitter: @MilosSQL
INEFFICIENT EXECUTION PLANS IN SQL SERVER

Execution plans are sometimes suboptimal due to significant inaccuracy in cardinality estimations

Affected Queries:
- Queries using table variables
- Queries with scalar user-defined functions
- Queries referencing multi-statement table valued functions
- Complex queries
- Queries with tables with skew data distribution

Issues:
- Operator choice
- Memory Grant under- or overestimation
LET'S MAKE EXECUTION PLAN GREAT AGAIN!
QUERY PROCESSING IN SQL SERVER

SQL Server 2016 and prior
After the execution plan is created, it is used in consecutive query executions, without changes (with the same operators and memory grants)

SQL Server 2017 Adaptive Query Processing
Breaking the pipeline between query optimization and execution
Executing a part of the query during the execution plan creation
Updating a part of the cached plan during consecutive query executions (Memory Grant)

SQL Server 2019 Intelligent Query Processing
Additional improvements, not only adaptive; therefore new name => Intelligent QP
INTERLEAVED EXECUTION

Related to queries with multi statement table valued functions (MTVF)

- Break the optimization process
- Execute part of the query with function call and get actual cardinality

Epilogue

- More appropriate plan (correct cardinality instead of cardinality 100)

Costs

- Increased CPU compile time
- Increased costs are acceptable, execution plan is usually better (sometimes significantly)

Limits

- It works with fixed parameters only
Interleaved Execution

DECLARE @d DATETIME = SYSDATETIME();
INNER JOIN dbo.SignificantOrders() f1 ON f1.Id = ol.OrderID WHERE PackageTypeID = 7;
PRINT CONCAT('Execution time: ', DATEDIFF(millisecond, @d, SYSDATETIME()), ' ms');
GO 5

dbo.SignificantOrders() cardinality: 74K rows

<table>
<thead>
<tr>
<th>Compatibility Level 130</th>
<th>Compatibility Level 140</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beginning execution loop</td>
<td>Beginning execution loop</td>
</tr>
<tr>
<td>Execution time: 777 ms</td>
<td>Execution time: 385 ms</td>
</tr>
<tr>
<td>Execution time: 763 ms</td>
<td>Execution time: 363 ms</td>
</tr>
<tr>
<td>Execution time: 781 ms</td>
<td>Execution time: 368 ms</td>
</tr>
<tr>
<td>Execution time: 765 ms</td>
<td>Execution time: 361 ms</td>
</tr>
<tr>
<td>Execution time: 772 ms</td>
<td>Execution time: 364 ms</td>
</tr>
<tr>
<td>Batch execution completed 5 times.</td>
<td>Batch execution completed 5 times.</td>
</tr>
</tbody>
</table>

Plan: Nested Loop Join
Plan: Hash Join
Batch Mode Adaptive Join

New operator - Adaptive Join

Allows to choose between Hash Join and Nested Loop Join at runtime

It starts as Hash Join and if after input scanning

- estimated number of rows < threshold ⇒ switches to Nested Loop Join
- estimated number of rows ≥ threshold ⇒ continues as Hash Join

It will better handle queries with variety of parameters, but it won’t solve all issues caused by wrongly chosen Join operator

It (still) requires at least one columnstore index
**Batch Mode Adaptive Join**

```
EXEC dbo.GetOrderDetails 1;
EXEC dbo.GetOrderDetails 112;
```
BATCH MODE ADAPTIVE JOIN

Q: Is it better with new Adaptive Join operator?

A: Yes!!! Under the CL 140, the query runs 4x faster!
**Batch Mode Adaptive Join**

Q: Is it better with new Adaptive Join operator?  **IT DEPENDS!**

A: Actually **NO** - under the CL 140, the query runs 5x slower!
Batch Mode Memory Grant Feedback

Adjust memory grant parameter in the execution plan AFTER the plan is generated after the first query execution.

Memory is adjusted for a query when it used less than 50% of granted memory is spilling out to tempdb.

In SQL Server 2017 requires a columnstore index on the affected table.
If memory grant memory values oscillate, the feature is disabled.

New plan attributes in the XML plan:

```xml
<MemoryGrantInfo>
  <SerialRequiredMemory>152</SerialRequiredMemory>
  <SerialDesiredMemory>1240</SerialDesiredMemory>
  <RequiredMemory>1352</RequiredMemory>
  <DesiredMemory>2440</DesiredMemory>
  <RequestedMemory>2440</RequestedMemory>
  <GrantWaitTime>0</GrantWaitTime>
  <GrantedMemory>2440</GrantedMemory>
  <MaxUsedMemory>1752</MaxUsedMemory>
  <MaxQueryMemory>1334464</MaxQueryMemory>
  <LastRequestedMemory>670408</LastRequestedMemory>
  <IsMemoryGrantFeedbackAdjusted>Yes: Adjusting</IsMemoryGrantFeedbackAdjusted>
</MemoryGrantInfo>
```

It works with cached plans.

It does not work with `OPTION (RECOMPILE)`.

It is not persisted if the plan is removed from cache.

In SQL Server 2019 it works in Row Mode.
INTELLIGENT QUERY PROCESSING IN SQL SERVER 2019

ROW MODE MEMORY GRANT FEEDBACK

BATCH MODE ON ROWSTORE

SCALAR UDF INLINING

TABLE VARIABLE DEFERRED COMPILATION

APPROXIMATE QUERY PROCESSING
BATCH MODE ON ROWSTORE
WHAT IS BATCH MODE?

Batch mode allows query operators to work on a batch of rows, instead of just one row at a time.
At the CPU level multiple rows processed at once instead of one row.

- Number of processing instructions restricted
- Better CPU cache utilization and increased memory throughput
- Not documented how to get number of rows in batch (899 in all my tests)

Can be beneficial for queries that are CPU bound.
Batch Mode on Colummstore introduced in SQL Server 2012

Improvements – up to 20x faster queries

Batch Mode on Rowstore introduced in SQL Server 2019

Instead of 1 row, a batch with 899 rows is processed at time
Some queries could be significantly faster

In my examples 2-6x faster!
BATCH MODE on ROWSTORE

A sample workload comparison row mode vs. batch mode

CL 140

CL 150
**Batch Mode on Rowstore**

Native support

No tricks with fake columnstore indexes or other optimizer delusions

Initial heuristics considers potential benefits of batch mode for operators

At least one table has $\geq 131.072$ rows

At least one potentially efficient batch operators: join, aggregate or window aggregate

At least one of the batch operator’s input should have not less than 131.072 rows
Extended event batch_mode_heuristics

More info:
Blog: Niko Neugebauer
http://www.nikoport.com/2018/10/07/batch-mode-part-3-basic-heuristics-analysis/

Blog: Dima Pilugin
http://www.queryprocessor.com/batch-mode-on-row-store/
Enabling/Disabling Batch Mode

Enabling

ALTER DATABASE current SET COMPATIBILITY_LEVEL = 150;

ALTER DATABASE SCOPED CONFIGURATION SET BATCH_MODE_ON_ROWSTORE = ON;

OPTION (USE HINT('ALLOW_BATCH_MODE'));

Disabling

ALTER DATABASE SCOPED CONFIGURATION SET BATCH_MODE_ON_ROWSTORE = OFF;

OPTION (USE HINT('DISALLOW_BATCH_MODE'));
**BATCH MODE ON ROWSTORE - RESULTS**

### Row vs. Batch mode

<table>
<thead>
<tr>
<th></th>
<th>Row</th>
<th>Batch</th>
<th>x-factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>445</td>
<td>131</td>
<td>3.4</td>
</tr>
<tr>
<td>Q2</td>
<td>159</td>
<td>81</td>
<td>2.0</td>
</tr>
<tr>
<td>Q3</td>
<td>1.067</td>
<td>504</td>
<td>2.1</td>
</tr>
<tr>
<td>Q4</td>
<td>1.847</td>
<td>695</td>
<td>2.7</td>
</tr>
<tr>
<td>Q5</td>
<td>4.795</td>
<td>842</td>
<td>5.7</td>
</tr>
<tr>
<td>Q6</td>
<td>5.730</td>
<td>6.000</td>
<td>4.4</td>
</tr>
</tbody>
</table>

**Graph:**
- **Q5:** 4,795 (Row) vs. 842 (Batch)
- **Q4:** 1,847 (Row) vs. 695 (Batch)
- **Q3:** 1,067 (Row) vs. 504 (Batch)
- **Q2:** 159 (Row) vs. 81 (Batch)
- **Q1:** 445 (Row) vs. 131 (Batch)
**Batch Mode on Rowstore – Conclusion**

Very promising feature
- Improvements with no efforts
- It could be a reason for upgrade for some companies

First version, probably will not optimize some queries that need an optimization

It brings benefits for queries with large tables and datasets

*It could significantly affect your workload!*
SCALAR UDF INLINING
SCALAR UDFS IN SQL SERVER

Code reuse, encapsulation and modularity

Complex business rules or computations
  Single place change
  Written once, invoke from many modules

Reduce network traffic

BUT...
Scalar UDFs in SQL Server

Why do SQL Server Scalar-valued functions get slower?

Refactor SQL Server scalar UDF to inline TVF to improve performance

Why SQL Server scalar functions are bad?

T-SQL Best Practices - Don't Use Scalar Value Functions in Column.

Are SQL Server Functions Dragging Your Query Down?

SQL functions rarely perform well.
SCALAR UDFs in SQL Server

They are very slow
Iterative invocation
Overhead for invoking function – once per row

No cross-statement optimization

Only serial execution plans possible
Froid: Optimization of Imperative Programs in a Relational Database

Karthik Ramachandra
Microsoft Gray Systems Lab
karam@microsoft.com

Kwanghyun Park
Microsoft Gray Systems Lab
kwpark@microsoft.com

K. Venkatesh Emami*
IIT Bombay
venkatshk@cse.iitb.ac.in

Alan Halverson
Microsoft Gray Systems Lab
alanhal@microsoft.com

César Galindo-Legaria
Microsoft
cesarg@microsoft.com

Conor Cunningham
Microsoft
conorc@microsoft.com

ABSTRACT
For decades, RDBMSs have supported declarative SQL as well as imperative functions and procedures as ways for users to express data processing tasks. While the evaluation of declarative SQL has received a lot of attention resulting in highly sophisticated techniques, the evaluation of imperative programs has remained naïve and highly inefficient. Imperative programs offer several benefits over SQL and hence are expressing intent has on one hand provided high-level abstractions for data processing, while on the other hand, has enabled the growth of sophisticated query evaluation techniques and highly efficient ways to process data. Despite the expressive power of declarative SQL, almost all RDBMSs support procedural extensions that allow users to write programs in various languages (such as Transact-SQL, C#, Java and R) using imperative constructs such as variable assignments, conditional branching, and loops.
Goal – improve queries where scalar UDFs are problem

Scalar UDF Inlining feature:
- Transforms scalar UDF into relational expressions or subqueries
  - IF => CASE WHEN
  - RETURN => SELECT
- Embeds them in the calling query
- Optimize expressions or subqueries

Result:
- Performance improved (more efficient plan)
- Execution plan could be parallel
### Table 1: Relational algebraic expressions for imperative statements (using standard T-SQL notation from [33])

<table>
<thead>
<tr>
<th>Imperative Statement (T-SQL)</th>
<th>Relational expression (T-SQL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DECLARE <code>{@var data_type [= expr]}[,...n];</code></td>
<td>SELECT `{expr</td>
</tr>
<tr>
<td>SET <code>{@var = expr}[,...n];</code></td>
<td>SELECT `{expr AS var</td>
</tr>
<tr>
<td>SELECT <code>{@var1 = proj_expr1}[,...n] FROM sql_expr;</code></td>
<td><code>{SELECT proj_expr1 AS var1 FROM sql_expr}; [,...n]</code></td>
</tr>
<tr>
<td>IF <code>(pred_expr) {t_stmt;[,...n]} ELSE {f_stmt;[,...n]}</code></td>
<td>SELECT CASE WHEN <code>pred_expr</code> THEN 1 ELSE 0 END AS <code>pred_val</code>; <code>{SELECT CASE WHEN </code>pred_val<code>= 1 THEN</code>t_stmt<code>ELSE</code>f_stmt<code>; }[,...n]</code></td>
</tr>
<tr>
<td><code>RETURN expr;</code></td>
<td>SELECT <code>expr AS returnVal;</code></td>
</tr>
</tbody>
</table>

Froid: Optimization of Imperative Programs in a Relational Database
SCALAR UDF INLINING

Not all scalar UDFs can be inlined

Check whether a function can be inlined:

```sql
SELECT CONCAT(SCHEMA_NAME(o.schema_id), '.', o.name), is_inlineable
FROM sys.sql_modules m
INNER JOIN sys.objects o ON o.object_id = m.object_id
WHERE o.type = 'FN';
```

`is_inlineable` does not imply that it will always be inlined

Decision is made when the query referencing a scalar UDF is compiled
Scalar UDFs Inlining - Limitations

- UDF does not invoke any intrinsic function that is either time-dependent or has side effects
  - such as GETDATE or NEWSEQUENTIALID
- The UDF does not reference
  - table variables
  - table-valued parameters
  - user-defined types
- The query invoking a scalar UDF
  - does not reference a scalar UDF call in its GROUP BY clause
  - in its select list with DISTINCT clause does not reference a scalar UDF call in its ORDER BY clause
- UDF is not natively compiled (interop is supported)
- UDF is not used in a computed column or a check constraint definition
- There are no signatures added to the UDF
- The UDF is not a partition function
Enable

ALTER DATABASE current SET COMPATIBILITY_LEVEL = 150;

ALTER DATABASE SCOPED CONFIGURATION SET
TSQL_SCALAR_UDF_INLINING = ON;

CREATE OR ALTER FUNCTION dbo.getMaxOrderDate(@CustID INT)
RETURNS DATETIME
WITH INLINE = ON

Disable

ALTER DATABASE SCOPED CONFIGURATION SET
TSQL_SCALAR_UDF_INLINING = OFF;

OPTION (USE
HINT('DISABLE_TSQL_SCALAR_UDF_INLINING'));

CREATE OR ALTER FUNCTION dbo.getMaxOrderDate(@CustID INT)
RETURNS DATETIME
WITH INLINE = OFF
## SCALAR UDF INLINING – RESULTS

<table>
<thead>
<tr>
<th>Expression</th>
<th>Scalar CL 140</th>
<th>Scalar CL 150</th>
<th>Inline TVF</th>
<th>Expression</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1 \quad \text{Quantity} \times \text{UnitPrice}</td>
<td>23.50</td>
<td>1.90</td>
<td>1.48</td>
<td>1.48</td>
<td>13x faster</td>
</tr>
<tr>
<td>F2 \quad \text{SUM(Quantity} \times \text{UnitPrice)}</td>
<td>75.40</td>
<td>0.7</td>
<td>0.21</td>
<td>0.21</td>
<td>&gt;100x faster</td>
</tr>
<tr>
<td>F3 \quad \text{GetMaxOrderDateForCustID}</td>
<td>5.60</td>
<td>4.45</td>
<td>0.14</td>
<td></td>
<td>25% faster</td>
</tr>
</tbody>
</table>
Very promising feature

Improvements with no efforts

It could be a reason for upgrade for some companies

Very useful for

Small and medium companies (not enough people to rewrite UDFs)

3rd party tools

Many limitations, but it’s a first version
TABLE VARIABLE
DEFERRED COMPILATION
PROBLEMS WITH TABLE VARIABLES

Queries using table variables with large number of rows (> 10K) can have an inefficient plan

Inappropriate operators in the execution plan
mostly Nested Loop Join instead of Hash Join

Insufficient memory grants
Number of processing rows is usually underestimated => less Memory
Grant reserved for the query => spills to tempdb
DEALING WITH PROBLEMS WITH TABLE VARIABLES

Using temporary tables instead of table variables

Using **OPTION (RECOMPILE)** at the statement level
    - It could be problem for frequently called queries

Enabling **TF 2453**
    - This trace flag allows a table variable to trigger recompile when enough rows are changed

Using **HASH JOIN** hints
    - when you know that table variable has large number of rows
TABLE VARIABLE DEFERRED COMPILATION

Implements plan quality and overall performance for queries referencing table variables

Cardinality estimates are based on actual table variable row counts

This accurate row count information will be used for optimizing downstream plan operations
TABLE VARIABLE DEFERRED COMPILATION

DECLARE @T AS TABLE (ProductID INT);
INSERT INTO @T SELECT ProductID
FROM Production.Product
WHERE ProductLine IS NOT NULL;

SELECT * FROM @T t
INNER JOIN Sales.SalesOrderDetail od
ON t.ProductID = od.ProductID
INNER JOIN Sales.SalesOrderHeader h
ON h.SalesOrderID = od.SalesOrderID
ORDER BY od.UnitPrice DESC;

SQL Server 2017 and all previous versions
- Content of the table variable unknown at the compile time
  => cardinality 1

SQL Server 2019
- Content of the table variable known at the compile time =>
  cardinality = actual number of rows
NEW TABLE VARIABLE BEHAVIOR

Cardinality of the table variable is actual number of rows, not 1

Selectivity of a single predicate is

$\text{SQRT}(\text{actual\_number\_of\_rows})$

No other changes

- no column statistics
- does not increase recompilation frequency

Still differs from temporary tables
DECLARE @t TABLE(id INT)

INSERT INTO @t SELECT n FROM dbo.GetNums(10000);

SELECT * FROM @t;

---

Compatibility Level 140

---

Compatibility Level 150

---
DECLARE @t TABLE(id INT)
INSERT INTO @t SELECT n FROM dbo.GetNums(10000);
SELECT * FROM @t WHERE id = 5;
SAMPLE QUERY WITH ESTIMATION ISSUES

SELECT B.* FROM A
INNER LOOP JOIN B ON A.id = B.pid
WHERE A.pid = 413032;
USING TV TO „FIX“ THE CARDINALITY ISSUE

DECLARE @t TABLE(id INT PRIMARY KEY);
INSERT INTO @t SELECT A.id FROM A WHERE A.pid = 413032;
SELECT B.* FROM @t A
INNER JOIN B ON A.id = B.pid;

Query 2: Query cost (relative to the batch): 36%
SELECT B.* FROM @t A INNER JOIN B ON A.id = B.pid
Table Variable Deferred Compilation - Conclusion

Designed to address cardinality issues caused by fixed estimation:

- Nested Loop Joins where Hash Joins are more appropriate
- Memory grant underestimation issues

- Better estimation for execution plans for new queries
- Prone to parameter sniffing
- Can break existing workarounds
APPROXIMATE QUERY PROCESSING
You don’t need always exact answers

Approximate Query Processing
Approximate Query Processing

Implemented or supported

Apache Spark, PostgreSQL, Oracle, Teradata, Vertica Analytics, Amazon Redshift DWH, Google BigQuery, BlinkDB (Facebook), Conviva, Verdict

Microsoft Research

Approximate Query Processing: No Silver Bullet

Surajit Chaudhuri, Bolin Ding, Srikanth Kandula
Microsoft Research
{surajitc,bolind,srikanth}@microsoft.com

ABSTRACT

In this paper, we reflect on the state of the art of Approximate Query Processing. Although much technical progress has been made in this area of research, we are yet to see its impact on products and services. We discuss two promising avenues to pursue towards integrating Approximate Query Processing into data platforms.

1. INTRODUCTION

While Big Data opens the possibility of gaining unprecedented insights, it comes at the price of increased need for computational resources (or risk of higher latency) for answering queries over voluminous data. The ability to provide approximate answers to queries at a fraction of the cost of executing the query in the traditional way, has the disruptive potential of allowing us to explore large datasets efficiently. Specifically, such techniques could prove effective in helping data scientists identify the subset of data that approximation is interesting for applications, approximation at the query processing layer has proven ineffective for applications, because either the semantics of error models and the accuracy guarantees of AQP or the extent of savings in work accrued by AQP have been unsatisfactory.

We firmly believe that the value proposition of AQP, outlined in the opening of this article, is considerable in the world of big data. We should not give up the pursuit of such systems. However, critical rethinking of our approach to AQP research is warranted with the exclusive goal of making such systems practical [30]. The first step in such rethinking is to be clear about what combinations of the four dimensions of AQP will make it possible for applications to find AQP systems attractive. In this article, we suggest two research directions to pursue based on our reflection.

One promising approach may be to code control over accuracy to the user. This approach is based on accepting the reality that AQP systems will not be able to offer a priori (i.e., before the query

Approximate Query Processing

Provide aggregations across very large data sets where responsiveness is more critical than absolute precision.

Currently only one function **APPROX_COUNT_DISTINCT**

- It uses significantly less memory resources.
- Error from the precise COUNT DISTINCT equivalent
  - within 2% for most workloads
  - within 20% for all workloads

Access of data sets that are millions of rows or higher.

Aggregation of a column or columns that have a large number of distinct values.
HyperLogLog

It hashes each element to make the data distribution more uniform. After hashing all the elements, it looks for the binary representation of each hashed element. HLL looks number of leading zero bits in the hash value of each element and finds maximum number of leading zero bits. 

\[ \text{Number of distinct elements} = 2^{k+1} \]

where \( k \) is maximum number of leading zeros in data set.

HyperLogLog algorithm documentation:
http://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf
APPROX_COUNT_DISTINCT

DEMO
**APPROX_COUNT_DISTINCT** - **Conclusion**

**Sometimes** better plan

Stream Aggregate instead of Hash Match Aggregate, for instance

**Significantly** less memory

**Sometimes** faster

speed is not main reason for using it, it could be even slower

Useful for

- data sets > millions of rows
- columns with large number of distinct values