Advanced Partitioning Techniques for Massively Distributed Computation

Jingren Zhou  
Microsoft  
jrzhou@microsoft.com

Nicolas Bruno  
Microsoft  
nicolasb@microsoft.com

Wei Lin  
Microsoft  
weilin@microsoft.com

Presented by Chris Hardulak
Many companies rely on massive data computation for critical business decisions.

PROBLEM: Scale of data volumes to be stored and processed is so large that traditional, centralized database system solutions are not practical.
Introduction

- **SOLUTION**: Distributed data storage and processing systems
- **Initiatives include**
  - MapReduce
  - Hadoop
  - Cosmos/Dryad
- **Limitations**: Too much custom code
  - Error-prone
  - Non-reusable
Introduction

• Limitations: Too much custom code
  ➢ Error-prone
  ➢ Non-reusable

• Better solutions:
  – Jaql
  – SCOPE
  – Tenzing
  – Dremel
  – etc.
Introduction

• Still problems:
  – Data shuffling is typically the most expensive operation and can lead to serious bottlenecks
Introduction

- Still problems:
  - Data shuffling is typically the most expensive operation and can lead to serious bottlenecks
  - Can utilize several advanced repartitioning techniques to reduce data movement
Background: SCOPE

- **Structured Computations** Optimized for **Parallel Execution**
- Similar to SQL
  - Retains much of SQL statements
  - Allows easy user definition of functions
  - Easily implement own versions of relational operators

```sql
SELECT ngram, COUNT(*) AS c
FROM (PROCESS
      SSTREAM "input.ss"
      USING NGramProcessor(4))
GROUP BY ngram;

OUTPUT TO "output.txt"
```
Background: SCOPE

SELECT ngram, COUNT(*) AS c
FROM (PROCESS
    SSTREAM "input.ss"
    USING NGramProcessor(4))
GROUP BY ngram;

OUTPUT TO "output.txt"
Background: SCOPE

Output (output.txt)
Aggregate ((ngram), count)
Process (NGramProcessor)
Get (input.ss)

Output (output.txt)
Global Stream Agg ((ngram), count)
Repartition (ngram)
Local Stream Agg ((ngram), count)
Sort (ngram)
Process (NGramProcessor)
Get (input.ss)

SV1
SV2
Background: SCOPE

- Output (output.txt)
- Global Stream Agg ((ngram), count)
- Repartition (ngram)
- Local Stream Agg ((ngram), count)
- Sort (ngram)
- Process (NGramProcessor)
- Get (input.ss)

SV1

SV2

SV2

SV2

SV2

SV1

SV1

SV1

SV1

SV1
Background: SCOPE

Partitioning

(a) Full Repartitioning  (b) Initial Split  (c) Full Merge

Figure 2: Different Types of Data Exchange
Partitioning – Partitioning Schemes

- **Hash Partitioning** – applies a hash function to the partitioning columns to generate the partition number to which the row is output.

- **Range Partitioning** – divides the domain of the partitioning columns into a set of disjoint ranges, as many as the desired number of partitions.
Partitioning – Merging Schemes

- **Random Merge** – randomly pulls rows from different input streams and merges them into a single output stream.

- **Sort Merge** – merges input streams into a single sorted output stream, sorted on a given column.
Background: SCOPE

Partitioning – Merging Schemes

• **Concat Merge** – concatenates streams together, maintaining internal order, but input streams can be in any order.

• **Sort-Concat Merge** – more complex:
  – Picks one row from each input stream
  – Sorts picked rows on a selected sort column
  – Then concatenates streams based on order of sorted rows.
A vertex can only start when all its inputs are already finished processing

Attempts to minimize the overall job latency
Structured Streams
• Structured data can be efficiently stored as structured streams
• Structured streams can be directly stored in a partitioned way
• Relying on pre-partitioned data significantly reduces latency by removing both the data exchange and the superfluous global aggregate operators
Structured Streams – Indexes for Random Access

- Within each partition, a local sorting order is maintained through a B+ tree index.
- Allows sorted access to the content of a partition
- Enables fast key lookup on a prefix of the sorting keys
Structured Streams – Data Affinity

• SCOPE does not require all the data in a partition to be on a single machine

• Every extent (unit of storage in SCOPE) has an optional affinity id.

• The system tries to place all the extents with an equivalent affinity id either on the same machine, or close by, in the same rack, or a close rack.
Partial Data Repartitioning

- Carefully defining partition boundaries ->
  - reduce data transfer between partition and merge vertices

- Some local partitions can be guaranteed to be empty ->
  - Partition vertices need to reserve fewer memory buffers and storage
  - Job manager does not need to maintain connections to these empty partitions.
  - Merge vertices do not need to wait for all inputs, just the ones that might have data
Partial Data Repartitioning

Repartitioning –
SELECT a, UDAgg(b) AS aggB
FROM SSTREAM "input.ss"
GROUP BY a;

OUTPUT TO SSTREAM "output.ss"
    [HASH | RANGE] CLUSTERED BY a;
Partial Data Repartitioning

Output
(output.txt)

Stream Agg
({a, UDAgg(b)})

Repartition
(a)

Get
(input.ss)

---

Output
(output.txt)

Repartition
(a,50)

Stream Agg
({a, UDAgg(b)})

Repartition
(a,200)

Get
(input.ss)
Partial Data Repartitioning

Hash-based Partitioning – Example

\[ h(C) \equiv 0 \pmod{4} = h(C) \equiv 0 \pmod{2} \]
Partial Data Repartitioning

Hash-based Partitioning – The Math

Merge vertex $M_i$ ($0 \leq i < po$) needs to read from partition vertex $P_j$ ($0 \leq j < pi$) if there might be a row in $P_j$ for which its hash value modulo $po$ is $i$.

So, read if $\exists k \mid k \equiv j \mod{pi}$ and $k \equiv i \mod{po}$

This implies $\exists (k_1 \text{ and } k_2) \mid k = k_1 \cdot po + i$ and $k = k_2 \cdot pi + j$

Equating and rearranging, we get: $po \cdot k_1 + (-pi) \cdot k_2 = (j-i)$

This is in the form $a \cdot x + b \cdot y = c$, which is solvable for integer $(x,y)$ if and only if $c$ is a multiple of the greatest common denominator of $a$ and $b$. 
Partial Data Repartitioning

Hash-based Partitioning – The Math

So, it boils down to:

$$M_i \text{ needs to read from } P_j \text{ if and only if } i \equiv j \mod \gcd(p_i, p_o)$$

Special Case: if $p_i$ and $p_o$ are co-primes, then $\gcd(p_i, p_o) = 1$ and no optimization possible.
Partial Data Repartitioning

Range-based Partitioning

\[ P_1 = [(1, A) \ldots (1, C)] \]
\[ P_2 = [(1, C) \ldots (2, E)] \]
\[ P_3 = [(2, E) \ldots \text{max}] \]

\[ P'_1 = [1 .. 2) \]
\[ P'_2 = [2 .. 3) \]
\[ P'_3 = [3 .. \text{max}) \]

Figure 6: Refining Range Partitions from \((a, b)\) to \(a\)
Partial Data Repartitioning

Range-based Partitioning – Determining Partitioning Boundaries

**Algorithm 1: PartitionBoundaries(C, T, B)**

Input: Columns C, Partition size T, Buckets B
Output: Partition boundaries P

/* Assume that C and B.cols share common prefix CP and for each 1 < i < |B|: B[i-1].hi = B[i].lo */
/* Output is partitioned by CP, which implies C, and each partition size is around T */

CB = Π_{CP}(B[i].lo, Π_{CP}(B[i].hi)) // project B on CP

idx = 0;
while idx < |CB| do
  actLo = CB[idx].Lo;
  actSize = CB[idx].Size;
  idx++;
  while actLo = CB[idx].Lo OR
    CB[idx].size / 2 < T - actSize do
    actSize += CB[idx].size;
    idx++;
  end
  P = P ∪ [actLo, CB[idx-1].hi);
end
return P;
Partial Data Repartitioning

Range-based Partitioning – Determining Data Flow Connections

• Connect merge vertex with partition vertex whenever:

\[ \Pi_{CP}[PO_{lo}^i, PO_{hi}^i] \cap \Pi_{CP}[PI_{lo}^j, PI_{hi}^j] \neq \emptyset \]

where

\[ \Pi_{CP}[lo, hi] = \begin{cases} [lo, hi] & \text{if } CP = C \\ [\Pi_{CP}(lo), \Pi_{CP}(hi)] & \text{otherwise} \end{cases} \]

Can guarantee that all input rows qualify when

\[ \Pi_{CP}[PI_{lo}^j, PI_{hi}^j] \subseteq [PO_{lo}^i, PO_{hi}^i] \]
Partial Data Repartitioning

Evaluation

<table>
<thead>
<tr>
<th>Domain</th>
<th>Host</th>
<th>Top-level-directory</th>
<th>URL-suffix</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.microsoft</td>
<td>www</td>
<td>download/</td>
<td>en/default.aspx?WT.mc_id=MSCOM_HIP_US_Nav_Downloads</td>
<td>...</td>
</tr>
<tr>
<td>com.microsoft</td>
<td>windows</td>
<td>products/</td>
<td>home browse?FORM=Z9LI16</td>
<td></td>
</tr>
<tr>
<td>com.bing</td>
<td>www</td>
<td>videos/</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Sample Information for a Web-pages Structured Stream

![Graphs](image)

Figure 9: An Aggregation Query over Web-pages
Partial Data Repartitioning

Optimizer Integration

• The optimizer considers all the alternatives and chooses the optimal solution based on the estimated costs

• The more alternatives, the more the optimizer has to work to determine the optimal one

• This is normally negligible compared to time saved via optimization, but is a consideration for some cases.
Partial Data Repartitioning

Optimizer Integration – General Opportunities

• The SCOPE optimizer chooses to repartition data based on requirements from subsequent operators

• GOAL: To obtain partitions that are processed in a reasonable amount of time (balance between recovery time for large partitions and scheduling and start-up for small partitions)
Partial Data Repartitioning

Optimizer Integration – N-ary Operators

• All inputs need to be partitioned in the same way
• If an input is already partitioned, use same partitioning for other inputs
• If this will skew the partitions, find better partitioning scheme for all inputs
• Broadcast optimization – when one input is very small, whole input can be sent to all partitions of larger input
Partial Data Repartitioning

Optimizer Integration – Eliminating Repartitioning

• When already partitioned by another sort column, if functional dependency exists, can avoid repartitioning.

• For example if already (a,b) already partitioned by b, and b→a, then no need to partition by a.
Partial Data Repartitioning

Evaluation

(a) Latency

(b) Total Work

(c) Total Data I/O

Figure 10: A Union-All Query on Web-pages
Index-based Partitioning

Scaling to large numbers of partitions can create problems: imagine a figure with 10,000 partitions

Solution: Index-based partitioning
Index-based Partitioning

Index-based partitioning
• Add a “partition number” column to data
• Sort on partition number.
• Store as structured stream with B+ tree index
• Merge vertices can then just use index to lookup relevant records to read.

<table>
<thead>
<tr>
<th>First Name</th>
<th>Male/Female</th>
<th>City</th>
<th>State</th>
<th>Zip</th>
<th>P#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seth</td>
<td>M</td>
<td>Cuyahoga</td>
<td>OH</td>
<td>44223</td>
<td>1</td>
</tr>
<tr>
<td>Robert</td>
<td>M</td>
<td>Cuyahoga</td>
<td>OH</td>
<td>44221</td>
<td>1</td>
</tr>
<tr>
<td>William</td>
<td>M</td>
<td>Hudson</td>
<td>OH</td>
<td>44236</td>
<td>1</td>
</tr>
<tr>
<td>Erick</td>
<td>M</td>
<td>Hudson</td>
<td>OH</td>
<td>44236</td>
<td>2</td>
</tr>
<tr>
<td>Nolyn</td>
<td>M</td>
<td>Hudson</td>
<td>OH</td>
<td>44236</td>
<td>2</td>
</tr>
<tr>
<td>Tyler</td>
<td>M</td>
<td>Hudson</td>
<td>OH</td>
<td>44236</td>
<td>2</td>
</tr>
<tr>
<td>Aj</td>
<td>M</td>
<td>Hudson</td>
<td>OH</td>
<td>44236</td>
<td>2</td>
</tr>
<tr>
<td>Madeline</td>
<td>F</td>
<td>Hudson</td>
<td>OH</td>
<td>44236</td>
<td>3</td>
</tr>
<tr>
<td>Nicholas</td>
<td>M</td>
<td>Hudson</td>
<td>OH</td>
<td>44236</td>
<td>3</td>
</tr>
</tbody>
</table>
Index-based Partitioning

Index-based partitioning

• Done outside of optimizer

• For each partition operator, the choice is given to use traditional or index-based approach.
Index-based Partitioning

Evaluation

Figure 11: Partitioning Scalability
Conclusion

- Massive data analysis plays crucial role in business
- High-level scripting languages free users from understanding complexities
- Data shuffling is most expensive component
- Using advanced partitioning techniques, data shuffling can be minimized and optimized
Questions?

The End