Business Intelligence (BI)

Ilya Zaides, Spring 2015
Information provided in these slides is for educational purposes only
Outline

• Introduction
  • Typical user requirements
• Optimizing data queries
  • “Aggregate-by-OS” use-case
• From theory to practice
  • Query optimization engine
Introduction
What is Business Intelligence (BI)?

• “Set of tools, methodologies & technologies for collecting, storing, analyzing, and providing access to data – to help make better decisions”

• BI solutions are widely used everywhere
  • Stock exchange
  • Marketing
  • Security
  • Data centers
  • etc.
Usages (1/3): Reports, Dashboards, KPI’s

• Deals with the processing and visualization of the data to answer questions such as:
  • What happened? When? Who? How many times?
  • What is the trend behavior?
  • Why did it happen?
Usages (2/3): Optimizations

• Integrate BI capabilities into a running system to take automated data-driven actions, e.g.,
  • Predict jobs resource usage based on historical data
  • Use the prediction to optimizes scheduling
  • Next lecture....
Usages (3/3): Hidden Properties

• Apply machine learning techniques to extract non-conventional properties of the data, e.g., correlations, clusters, etc.
  • What will happen if we change x?
  • What else does the data tell us that we never thought to ask?
  • Predictive modeling and analytics
BI in the Data Center

• Which information is there?
  • Facility level
    • Power consumption
    • Cooling systems
    • Temperature
    • Etc.
  • Individual systems
    • CPU utilization
    • Memory consumption
    • Disk space utilization
    • Etc.
BI in the Data Center cont.

• More information...
  • Application (e.g., job scheduler)
    • Number of waiting and running jobs
    • Wait times in the scheduler queues
    • Resources consumed by the jobs
      • CPU, memory, etc.
  • Dependency between jobs
    • Exit status
    • Wait reasons
    • Etc.
Mission: take the data and....make it look like this
Typical data flow
Typical data flow: challenges

• Data Sources
  • Many different systems, technologies, data formats (log files, csv’s, etc.)
  • Solutions must not leave footprint on the source system

• Data Preparation
  • Filtering, conversions, normalizations, etc.
  • We will discuss best practices later

• Data Storage
  • Ensuring fast and highly-available access
  • Algorithms – Average vs. Median
Typical data flow: challenges cont.

• Data Analysis
  • Simple to complex queries
  • **This lecture — Optimizing data queries**
  • Next lecture — Predictive algorithms

• Data Access
  • Ensuring fast response
  • Ensuring clear (intuitive) representation (GUI)
Optimizing data queries
Motivation

• Eventually it’s all about users making queries
• Data must be prepared and queries must be implemented efficiently to address user requirements e.g.,
  1. Responsiveness – Up to 5 seconds over the Web (Amazon, EBay, ... )
  2. Latency – How recent the data should be? i.e. stock data in Bizportal is received with delay of 20 minutes
  3. Retention – For how long the data is relevant? Do you want to see all your shopping deals in Amazon (event those that are 10 years old)?
• It is possible to drastically change performance with small tweaks
  • When you understand what is happening...
Reference system

• Data Center with thousands of servers
  • Different servers run different Operating Systems, e.g., SUSE10, SUSE11

• Tens-of-thousands of jobs every day
  • Managed by a central scheduler (RM lectures I-III)

• Finished jobs recorded in a database (history)
  • In practice this is done in bulks e.g., 10,000 jobs in a shot
Reference system cont.

Collect finished jobs data every 1 minute

Cluster A
Cluster B
Cluster C

Challenges in Modern Data Centers Management, Spring 2015
Jobs per-day, per-Operating-System (OS)

• User request
  • “I want to see how many jobs finished each day per-OS“ (two dimensions)

• Additional requirements (from UI standpoint)
  1. Responsiveness = 30s
  2. Latency = 1D
  3. Retention = 1y
  4. Aggregation = per day
Source data

• Two tables
  1. “Servers” – Mapping between Servers and OS (total of 1000 rows)
  2. “Jobs” – Historical jobs records (total of 1,000,000 rows)

Table “Servers”

<table>
<thead>
<tr>
<th>Server</th>
<th>OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
<td>Suse10</td>
</tr>
<tr>
<td>W2</td>
<td>Suse11</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table “Jobs”

<table>
<thead>
<tr>
<th>JobId</th>
<th>Server</th>
<th>FinishDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>W1</td>
<td>01/01/2015</td>
</tr>
<tr>
<td>2</td>
<td>W1</td>
<td>01/01/2015</td>
</tr>
<tr>
<td>3</td>
<td>W2</td>
<td>02/01/2015</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Desired Output

<table>
<thead>
<tr>
<th>OS</th>
<th>FinishDate</th>
<th>Number of Jobs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suse10</td>
<td>01/01/2015</td>
<td>2545</td>
</tr>
<tr>
<td>Suse11</td>
<td>01/01/2015</td>
<td>1555</td>
</tr>
<tr>
<td>Suse10</td>
<td>02/01/2015</td>
<td>4821</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- Two phases
  1. Join between the two tables
    - `Select * from Servers join Jobs on Servers.Server=Jobs.Server`
  2. Create aggregation on the join result
    - `Select OS, FinishDate, count(*) as [Number of Jobs]` FROM `Joined GROUP BY OS, FinishDate`
Step 1 - Join

<table>
<thead>
<tr>
<th>Server</th>
<th>OS</th>
<th>JobId</th>
<th>Server</th>
<th>FinishData</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
<td>Suse10</td>
<td>1</td>
<td>W1</td>
<td>01/01/2015</td>
</tr>
<tr>
<td>W1</td>
<td>Suse10</td>
<td>2</td>
<td>W1</td>
<td>01/01/2015</td>
</tr>
<tr>
<td>W2</td>
<td>Suse11</td>
<td>3</td>
<td>W2</td>
<td>01/01/2015</td>
</tr>
</tbody>
</table>

Step 2 – Aggregate (sum)

<table>
<thead>
<tr>
<th>OS</th>
<th>FinishDate</th>
<th>Number of Jobs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suse10</td>
<td>01/01/2015</td>
<td>2545</td>
</tr>
<tr>
<td>Suse11</td>
<td>01/01/2015</td>
<td>1555</td>
</tr>
<tr>
<td>Suse10</td>
<td>02/01/2015</td>
<td>4821</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Join algorithms

1. Nested-loop join
2. Merge join
3. Hash join

• Slides based on: Craig Freedman's SQL Server Blog, you can find there a lot of relevant information. Despite it is MsSql server specific but the ideas are relevant for everyone
1. Nested Loop Join

- **Algorithm:**
  
  For each row $r_1$ in $R_1$
  
  For each row $r_2$ in $R_2$
  
  if $r_1$ joins with $r_2$
  
  return $(r_1, r_2)$

- **Time Cost:** $O(|R_1| \times |R_2|)$

- **Space Cost:** $O(1)$

“Joins” means the condition on which we join the tables is met (==true)
2. Merge Join

• Similar to merge operation of merge sort algorithm
• Assumptions:
  • R1 and R2 are already sorted according to the join keys
  • There must be at least one equijoin key ("=" operator in the join)
• Time cost (without sort): $|R1| + |R2|$
• Space cost: $O(1)$

- Algorithm

  get first row \( r_1 \) from \( R_1 \)
  get first row \( r_2 \) from \( R_2 \)
  while not at the end of either input
  begin
    if \( r_1 \) joins with \( r_2 \)
    begin
      return \((r_1,r_2)\)
      get next row \( r_2 \) from \( R_2 \)
    end
    else if \( r_1 < r_2 \)
    get next row \( r_1 \) from \( R_1 \)
    else
    get next row \( r_2 \) from \( R_2 \)
  end
3. Hash Join

• Two steps
  1. **Build step** - First loop to build the hash table from \( R_1 \)
  2. **Probe phase** – Second loop, where we probe values from \( R_2 \) whether they exist in has table

• Assumptions
  • There must be at least one equijoin key ("=" operator in the join)

• Unlike previous methods this one cannot output results immediately (on-the-fly) but only in probe phase

• Time cost: \( O(|R_1| + |R_2|) \),

• Space cost: \( O(\max(|R_1|, |R_2|)) \), depends on the hash functions.

• CPU intensive because of hash function (relatively to other options)
3. Hash Join cont.

• Algorithm

  for each row $r_1$ in $R_1$  (**Build step**)  
  begin  
  calculate hash value on $r_1$ join key\'s  
  insert $r_1$ into appropriate hash bucket  
  end  

  for each row $r_2$ in $R_2$  (**Probe step**)  
  begin  
  calculate hash value on $r_2$ join key\'s  
  for each row $r_1$ in the corresponding hash bucket  
  if $r_1$ joins with $r_2$ return ($r_1$, $r_2$)  
  end
## Join: Summary

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Best for</strong></td>
<td>Relatively small inputs</td>
<td>Medium to large inputs, with ordered input on the equijoin keys</td>
<td>Medium to large input. Parallel execution that scales linearly.</td>
</tr>
<tr>
<td><strong>Users concurrency</strong></td>
<td>Supports large numbers of concurrent users</td>
<td>Supports large numbers of concurrent users</td>
<td>Small number of concurrent users with high throughput requirements</td>
</tr>
<tr>
<td><strong>Equijoin required</strong></td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Uses memory</strong></td>
<td>No</td>
<td>No, but may requires ordered input</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Requires order</strong></td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>Preserves order</strong></td>
<td>Yes (of outer loop)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>CPU intensive</strong></td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Blocking</strong></td>
<td>No</td>
<td>No</td>
<td>Yes (for build phase)</td>
</tr>
</tbody>
</table>
Aggregate Algorithms

1. Stream Aggregate
2. Hash Aggregate
1. Stream Aggregate

• Assumptions:
  • R is sorted by the aggregation columns
• Can return results before algorithm is finished
• Time Cost: $O(|R|)$
• Space Cost: $O(1)$
1. Stream Aggregate cont.

- Algorithm
  
  for each row r in R begin
  
  if the input row does not match the current aggregate columns begin
    output the aggregate results
    clear current aggregate results
    set the current aggregate columns to the input row
  end
  
  update the aggregate results with row r
  
  end
2. Hash Aggregate

• No assumptions on the data (need not be sorted)
• Blocking operation, we will receive results only after the entire operation is finished (stream aggregate is non-blocking)
• Time Cost: $O(|R|)$
• Space Cost: $O(|R|)$
2. Hash Aggregate cont.

• Algorithm
  for each row $r$ in $R$
  begin
    calculate hash value on aggregate columns
    check for a matching row in the hash table
    if we do not find a match
      insert a new row into the hash table
    else
      update the matching row with the input row
  end
  output all rows in the hash table
Small reminder about Sort

• The operation is blocking
• Time Cost: $O(n \log n)$
• Space Cost: $O(n)$
Back to our use case - Assumptions

• When calculating memory for hash join we take $\min(|R_1|, |R_2|)$ and not $\max(|R_1|, |R_2|)$, as defined previously
  • Since we know in advance what is better...
Back to our use case: Option 1

\[ \text{Sort}(\text{Servers}) \quad |J| \log |J| \]
\[ \text{Sort}(\text{Jobs}) \]
\[ \text{Merge Join} \]
\[ \text{Sort} \]
\[ \text{Stream Aggregate} \]

\[ \max(|J| + |W|, |J \bowtie W| \log |J \bowtie W|) \]
\[ |J \bowtie W| \]

Merge join is not blocking, operation, and we assume infinite parallelism. Only time is important.

\( J \) = Jobs table
\( W \) = Workstations (servers) table
\( |J| = 1,000,000 \)
\( |W| = 1000 \)
\( |J \bowtie W| = 1,000,000 \)
Back to our use case: Option 2

Hash Join

Min(|J|,|W|)

Sort

|J| \propto |W| \log_2 |J| \propto |W|

Stream Aggregate

|J| \propto |W|

J = Jobs table
W = Workstations table
|J| = 1,000,000
|W| = 1000
|J \propto W| = 1,000,000
Back to our use case: Option 3

\[ J = \text{Jobs table} \]
\[ W = \text{Workstations table} \]
\[ |J| = 1,000,000 \]
\[ |W| = 1000 \]
\[ |J \bowtie W| = 1,000,000 \]

- Hash Join
- Hash Aggregate

Min(|J|, |W|) = |J \bowtie W|
And the winner is ...

• Time Cost:
  • Option 1: $1000000 \log_2 1000000 + 1000000 + 1000000 \approx 21931000$
  • Option 2: $1000 + 1000000 \log_2 1000000 + 1000000 \approx 20932000$
  • Option 3: $1000 + 1000000 = 1001000$

• Memory Cost (maximum memory we needed at some point of time):
  • Option 1: $19931000$ (sort operation)
  • Option 2: $1000 + 19931000$
  • Option 3: $1000 + 19931000$

• Will the result be different if we need the results sorted, or the tables would be already sorted? (Yes)
From theory to practice
From theory to practice

• At the end the algorithms run on **real hardware**
  • Servers with CPUs (one or more), storage (HDD, SSD, RAID), network, etc.
  • Not everything can be loaded into memory (hash,...)

• Tables content is constantly changing
  • Hard to predict how many rows the join will produce

• Queries may be executed in isolation or in parallel (impact of contention)
  • Challenging to predict in advance

• All these (and more) affect performance
  • Not just theoretical calculations
Query optimization engine

• Query optimization engine has logic to pick
  • About 200ms to decide on flow
  • Engine smart enough 99% success
  • Takes into account:
    • Existing indexes
    • Statistics info
    • Temporary storage available
    • Other ...
  • Flow can be enforced by user
MsSQL: 1000 rows in jobs table

Results in 31ms
MsSQL: 1,000,000 rows in jobs table

Results in 1275ms
Best practices for good data flow design

1. Store only what is needed
2. Understand the data
   - Source system
   - Update frequency
   - Size, etc.
3. Understand the usage
   - Most important user queries – optimize for them
   - Usage model – Web \ Script \ ...
   - Number of potential users – Make things parallel if needed