A Graph-based Database Partitioning Method for Parallel OLAP Query Processing

Yoon-Min Nam, Min-Soo Kim*, Donghyoung Han

Department of Information and Communication Engineering

DGIST, Republic of Korea

* a corresponding author
Outline

- Introduction
- Motivation
- GPT: Graph-based database partitioning scheme
- HMC: Hash-based multi-column partitioning
- Experimental results
- Conclusions
Join processing in parallel system

- Need shuffle relations according to their join key values
  - due to their data distribution, i.e., blocks in HDFS
  - shuffle becomes more expensive as the input size increases
Database partitioning

- Co-locate a set of rows according to their join key values
  - to avoid expensive shuffle, i.e., co-partitioned join

```
\[ R \bowtie S \]
```

Horizontal partitioning $R$

Horizontal partitioning $S$
Three important issues in database partitioning

- Which tables are partitioned?
- Which column(s) is/are used for table partitioning?
- How to partition a table? (e.g., hash / range / round-robin / ...)

(a) BioWarehouse schema [BMC'15]  
(b) A list of 24 columns of Store_Sales table in TPC-DS
Existing database partitioning schemes

**REF [SIGMOD’08]**

- exploit referential constraint

(a) Referential constraint in TPC-H schema

![Referential constraint in TPC-H schema](image)

**PREF [SIGMOD’15]**

- generalize REF approach by additionally exploiting join predicate
- more chance to co-partition tables
- use tree-structured partitioning scheme

(b) REF partitioned tables

![REF partitioned tables](image)
Drawback of PREF(1): large data redundancy

- **Table-level duplicates** (e.g., PREF/WD, workload-driven)
  - data dependency btw. parent-child tables (reference partitioning)
  - cumulative redundancy (caused by data dependency)

- **Tuple-level duplicates** (e.g., PREF/SD, schema-driven)

(a) Table-level duplicates (PREF/WD)

(b) Data dependency (PREF/SD)
Tuple-level duplicates in PREF scheme

(a) Sample database

PREF/SD

GPT/SD
Drawback of PREF(2): low query performance

- **Large scan & join overhead** due to large *DR*
  - *DR*: data redundancy

- **NOT support local query processing** in many cases

**Example: TPC-DS Q17**

- nine join operations among five tables
- PREF requires shuffle tables for two joins (e.g., CS-D and SR-D)

![Query join graph of TPC-DS Q17](image)

![PREF/WD](image)

![GPT/WD](image)
PREF vs. GPT (TPC-DS benchmark)

(a) PREF/WD

(b) Partitioning graph of GPT/WD
Observation in our graph-based approach

- Star, snowflake and snowstorm schemas in OLAP (VLDB’14)
  - Join query in OLAP comes from foreign key relationships in the schema

Partitioning scheme as a Graph ($PG$)

- e.g., partitioning a large table and replicating a small table
- hub vertex: shared (dimension) table or frequently joined table
- select expensive and frequently used join op. as an edge of $PG$
Vertices and edges in $PG$ (Partitioning Graph)

- Three kinds of table
  - Part (partitioned), Rep (replicated) and Hub
  - Hub table is converted from Rep type to Part type

- Two special edges
  - triEdge: a set of edges that form a triangle
  - indirect join edge: does not exist in $G$ but form a triangle
Problem definition

Given a database $\mathbb{D}$, and a join graph $G = (V, E, l, w)$, the problem is finding the optimal partitioned graph $PG^*$ s.t.

$$PG^* = \arg\max_{PG_i \in PG} \{\text{benefit}(PG_i \cdot E)\}$$

Benefit model for three edge types

- intra ($E_a$), inter ($E_r$) and indirect join edge ($E_t$)

$\text{benefit}(e \in E_a) = ||e|| \cdot w(e) + \sum_{e' \in \text{triEdges}(e)} (||e'|| \cdot w(e'))$

$\text{benefit}(e \in E_r) = ||e|| \cdot w(e)$

$\text{benefit}(e \in E_t) = \sum_{e' \in \text{triEdges}(e)} (||e'|| \cdot w(e'))$
HMC: Hash-based multi-column partitioning

- Co-partitioning method for edges of $PG$

- Partition a table by hashing its partitioning column values
  - $N$ partitions for a table $T$
  - $P(T) = \{P_1(T), \ldots, P_N(T)\}$

- Store a tuple $t \in T$ up to $\kappa$ different partitions among $P(T)$
  - using $\kappa$ different partitioning columns, $C(T) = \{c_1, \ldots, c_\kappa\}$

- Indicate duplicates by using bitmap vector
  - each $t \in T$ has bitvector $\text{bitV} = [b_1, \ldots, b_\kappa] \in \text{Dup}$
  - e.g., $h(t.c_k) = i \rightarrow t \in T$ is copied to $P_i(T) \rightarrow$ set $b_k = 1$ for $t$
**Example of HMC method**

* $N = 3$ and $1 \leq h(\cdot) \leq 3$
  - e.g., $h(1) = 1$, $h(2) = 2$, $h(3) = 3$ and $h(4) = 1$

\[
\begin{array}{|c|c|c|}
\hline
\hline
1 & 3 & 5 & \\
2 & 5 & 7 & \\
3 & 5 & 9 & \\
4 & 1 & 2 & \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|}
\hline
\hline
& 1 & 3 & 5 & \\
& 4 & 1 & 2 & \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|}
\hline
dup(P_1(R)) & (1,0) & \\
& (1,1) & \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\hline
& 1 & 5 & \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\hline
& 2 & 3 & \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\hline
& 3 & 2 & \\
\hline
\end{array}
\]
Subpartition for efficient duplicate elimination

- The number of partitioning columns limits the number of unique bit vectors \( \text{bitV} \)
  - i.e., \( \kappa = 2 \rightarrow 2^2 \text{bitV}: \{00, 01, 10, 11\} \rightarrow 2^2 \text{ Subpartitions} \)
- Co-locate tuples having the same bit vector together
- Scan only necessary Subpartitions depends on the query

<table>
<thead>
<tr>
<th>Subpartition 1</th>
<th>Subpartition 2</th>
<th>Subpartition 3</th>
<th>Subpartition 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitV = (0, 0)</td>
<td>bitV = (0, 1)</td>
<td>bitV = (1, 0)</td>
<td>bitV = (1, 1)</td>
</tr>
<tr>
<td>( P_1(R) )</td>
<td>( P_2(R) )</td>
<td>( P_3(R) )</td>
<td>( P_4(R) )</td>
</tr>
<tr>
<td>R</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 3 5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 5 7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 5 9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 1 2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ 1 3 5 \]
\[ 4 1 2 \]
\[ 2 5 7 \]
Experimental evaluation

Three performance measures
- Data redundancy
- Data bulk loading performance
- Query performance

Experimental environments
- 11 machines as default, i.e., 1 master + 10 nodes (up to 21 machines)
- H/W: each machine has six-core i7 CPU, 32GB RAM, 1.2TB PCIE-SSD, 1Gbps Ethernet network
- S/W: Hadoop 2.4.1
- TPC-DS benchmark for dataset and queries
**Data redundancy**

- **PREF** suffers from large data redundancy
  - PREF/SD (tuple-level dup.) and PREF/WD (table-level dup.)
- **GPT’s DR** only slightly increases due to Rep-Tables
  - no data-dependency due to hash-based co-partitioning (HMC)
  - no table-level duplicates

![Graph (a) Changing scale of dataset](image1)

- Scale of dataset (N=50)
  - SF=25: PREF/SD 59 GB, PREF/WD 578 GB, GPT/SD 83 GB, GPT/WD 173 GB
  - SF=50: PREF/SD 42 GB, PREF/WD 374 GB, GPT/SD 43 GB, GPT/WD 86 GB
  - SF=100: PREF/SD 43 GB, PREF/WD 210 GB, GPT/SD 88 GB, GPT/WD 191 GB

![Graph (b) Changing # partitions](image2)

- # partitions (SF=100)
  - N=25: PREF/SD 347 GB, PREF/WD 356 GB, GPT/SD 356 GB, GPT/WD 184 GB
  - N=50: PREF/SD 189 GB, PREF/WD 375 GB, GPT/SD 196 GB, GPT/WD 191 GB
  - N=100: PREF/SD 201 GB, PREF/WD 402 GB, GPT/SD 233 GB, GPT/WD 210 GB
Data bulk loading performance

- **Dataset**: TPC-DS
- **Bulk loading performance** $\propto DR$

### Figures

(a) Changing scale of dataset

(b) Changing # partitions

- **Y-axis**: Elapsed time (sec.)
- **X-axis**: Scale of dataset (N=10) or # partitions (SF=100)
- **Legend**:
  - PREF/SD
  - PREF/WD
  - GPT/SD
  - GPT/WD

### Table

<table>
<thead>
<tr>
<th>Scale of dataset (N=10)</th>
<th>SF=25</th>
<th>SF=50</th>
<th>SF=100</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREF/SD</td>
<td>620</td>
<td>879</td>
<td>2170</td>
</tr>
<tr>
<td>PREF/WD</td>
<td>490</td>
<td>692</td>
<td>3696</td>
</tr>
<tr>
<td>GPT/SD</td>
<td>472</td>
<td>786</td>
<td>1446</td>
</tr>
<tr>
<td>GPT/WD</td>
<td>472</td>
<td>786</td>
<td>1446</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># partitions (SF=100)</th>
<th>N=25</th>
<th>N=50</th>
<th>N=100</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREF/SD</td>
<td>4565</td>
<td>7811</td>
<td>14499</td>
</tr>
<tr>
<td>PREF/WD</td>
<td>4453</td>
<td>5344</td>
<td>7422</td>
</tr>
<tr>
<td>GPT/SD</td>
<td>1624</td>
<td>1690</td>
<td>1715</td>
</tr>
<tr>
<td>GPT/WD</td>
<td>1668</td>
<td>1668</td>
<td>1788</td>
</tr>
</tbody>
</table>
Query performance (1): TPC-DS 1TB

(a) Comparison results for schema-driven approach (Y-axis is log-scale)

(b) Comparison results for workload-driven approach
Query performance (2): IMDB and BioWarehouse

(a) IMDB \((\kappa = 2)\)

(b) BioWarehouse \((\kappa = 2)\)

(c) Data redundancy

(d) Query performance
Query performance (3)

- **TPC-DS 1TB**
- **Linear scalability**
- **Performance breakdown**
  - varying optimization techniques

(a) Scalability

<table>
<thead>
<tr>
<th># machines</th>
<th>5</th>
<th>10</th>
<th>15</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total elapsed time (sec.)</td>
<td>7792</td>
<td>3707</td>
<td>2500</td>
<td>1873</td>
</tr>
</tbody>
</table>

(b) Performance breakdown

<table>
<thead>
<tr>
<th>Elapsed time (sec.)</th>
<th>GPT partitioning</th>
<th>Subpartitioning</th>
<th>TPC-DS Q17</th>
<th>TPC-DS Q29</th>
<th>TPC-DS Q85</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>O</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>O</td>
<td>O</td>
<td>X</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td></td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>

- 370 - 219 - 397 - 251 - 194 - 154
Conclusions

- We propose a graph-based partitioning scheme for fast and efficient OLAP query processing

- We propose two techniques for horizontal database partitioning in OLAP setting
  - GPT: low data redundancy while maximizing query performance
  - HMC: hash-based multi-column partitioning strategy with efficient duplicate elimination method

- We show GPT significantly outperforms PREF
  - e.g., 48% faster performance with 2.35 X less space in TPC-DS
Thank you!
Any questions?
References


Back-up slides
GPT method

Algorithm 1 GPT: Graph-based database ParTitioning

Input: \( G = \{V, E, w, l\} \), // undirected multi-graph
\( \kappa \) // max # of partitioning columns per table

Variable: \( benefitQ \) // max-priority queue of \( \langle benefit(e), e \rangle \)

Output: \( PG = \{V, E\} \) // partitioned graph (subgraph of \( G \))

1: // Step1: initialization
2: split \( V \) into \( V_{Part} \) and \( V_{Rep} \); // according to Eq.(3)
3: add \( V_{Part} \) to \( PG.V \);
4: \( E_a \leftarrow \{e|e=(R.i, S.j) \in E \land R \neq S \land R \in V_{Part} \land S \in V_{Part}\}\);
5: \( E_r \leftarrow \{e|e=(R.i, S.j) \in E \land R \neq S \land R \in V_{Part} \land S \in V_{Rep}\}\);
6: \( E_t \leftarrow \{e|e \text{ is an indirect join edge}\}\);

7: // Step2: building an initial \( benefitQ \)
8: for each \( e \in E_a \cup E_r \cup E_t \) do
9: \( benefitQ.insert(\langle benefit(e), e \rangle)\);
10: end for

11: // Step3: adding edges and vertices to \( PG \)
12: while \( benefitQ \neq \emptyset \) do
13: \( \langle benefit, e \rangle \leftarrow benefitQ.extractMax()\);
14: if \( (|C(R)| < \kappa) \land (|C(S)| < \kappa) \) s.t. \( (R, S) \in e \) then
15: add \( hub(e) \) to \( PG.V \);
16: add \( e \) to \( PG.E \);
17: add \( triEdges(e) \) to \( PG.E \);
18: \( benefitQ.updateBenefit(adj(e))\);
19: end if
20: end while
21: return \( PG \);
Table classification

Cost-based approach

\[ PartCost(T) = \|P(T)\| \cdot |\text{adj}(T)| + \sum_{S \in \text{adj}(T)} \|S\| \cdot |C(S)| \quad (1) \]

\[ RepCost(T) = \|T\| \cdot N \cdot |\text{adj}(T)| + \sum_{S \in \text{adj}(T)} \|S\|(|C(S)| - 1) \quad (2) \]

\[ DiffCost(T) = (\|P(T)\| - \|T\| \cdot N) |\text{adj}(T)| + \sum_{S \in \text{adj}(T)} \|S\| \quad (3) \]

(a) Rep-table scenario

(b) Part-table scenario
Characteristics of GPT

- Query performance while varying $\kappa$ and H/W configurations
  - TPC-DS 1TB and 20 Queries

(a) Data Redundancy

(b) Query Performance
GPT on Apache SPARK

How to use?
- in table creation DDL, turn GPT partitioning on
- support all file formats, e.g., CSV and Parquet

```
spark-sql>
> create table store_sales
> USING PARQUET
> OPTIONS ("GPT"="true", "DBName"="tpcds_SF1000_pt", "TableName"="store_sales_text", "Sort"="false")
> CLUSTERED BY (ss_sold_date_sk, ss_item_sk) INTO 240 BUCKETS
> LOCATION "/tpcds/TPCDS_SF1000_PT/store_sales"
> AS SELECT * FROM store_sales_text;
```

GPT-aware query plan generation
- automatically generates a query plan without shuffles if possible
- scan op. exploits bitV to scan Subpartitions depend on the query
Elapsed time (per query / log scale)

- TPC-DS Query (1TB database)
  - Hive
  - Spark
  - Spark+Bucketing
  - GPT (MapReduce)
  - GPT (Spark)