Skipping-oriented Partitioning for Columnar Layouts

Authors: Liwen Sun, Michael J. Franklin, Jiannan Wang, and Eugene Wu

Presentation: Allison Weaver
Overview

- Introduced to Generalized Skipping Oriented Partitioning (GSOP): a hybrid data skipping framework that takes into account row-based and column-based store tradeoffs with partitioning data
  - GSOP generalizes the original SOP framework by removing the atomic-tuple constraint
Importance

- As data volumes continue to grow, data skipping mechanisms become more critical to improve performance in modern analytics databases and the Hadoop ecosystem.
- Need a method of data skipping that is optimized for column-based stores rather than just row-based stores.
- Finds balance between tuple reconstruction and skipping effectiveness.
Vocab

- **Tuple**: A single row of a table, which contains a single record for that relation
- **Block**: tens of thousands of tuples, how data is organized
- **Feature**: representative filters which can span many columns
- **Feature Vector**: characterization of tuple
- **Feature Conflict**: when the best partitioning schemes for different features do not overlap, happens often with complex workloads
- **Tuple reconstruction**: the process of assembling requested column values back into tuples during query processing
- **Data cell**: each individual column value of a tuple
General Comparison SOP vs. GSOP

- Ex. two features to be extracted: F1: grade == ‘A’ and F2: year > 2011 AND course = ‘DB’
  - F1 – t1,t2 | t3, t4 (separate A’s from rest of rows)
  - F2 – t1,t4 | t2, t3 (separate year 2011 and separate DB’s)
- SOP produces only monolithic horizontal partitioning schemes, viewing every tuple as an atomic unit. This can result in feature conflicts.
- GSOP solves feature conflicts, but can result in tuple reconstruction.
SOP Framework

Based on two properties:

1. **Filter Commonality**: a small set of filters are commonly used by many queries (10% of filters used by 90% of queries)
   ○ Designing data layout based on small number of filters can benefit most queries

2. **Filter Stability**: a tiny fraction of query filters are newly introduced over time
   ○ Designing data layout based on past query filters can benefit future queries
Steps of SOP

1. Workload Analysis
   ○ Extracts features using frequent item-set mining
   ○ Subsumption relations

2. Augmentation
   ○ Data are scanned for given features and results stores in augmented feature vector

3. Partitioning
   ○ Group vector, tuple pairs into vector, count pairs
   ○ Clustering algorithm generates partition map
   ○ Each block gets a union map
Partitioning Spectrum

- Right end: partition each column individually, mitigates feature conflicts, introduces overhead for tuple reconstruction
- Left end: SOP framework, no separation of columns, no tuple reconstruction, a lot of feature conflicts
- Ex. SELECT B, D FROM T WHERE B<0 and D=2
GSOP Framework

1. Workload Analysis
   - Global features
2. Augmentation
   - Global feature vector
3. Column Grouping:
   - Divide columns into column groups based on objective function based on tradeoff
4. Local Feature Selection
   - Select subset of global features
   - Crucial step for skipping effectiveness
5. Partitioning
   - Local feature vectors
   - Project global feature vectors to keep bits of local features
Column Grouping

Ex. Consider following workload:
Q1: SELECT A, C FROM T WHERE A = 'm'
Q2: SELECT B, D FROM T WHERE B < 0
Q3: SELECT B, C FROM T WHERE C like 'y%'

- AC, BC, BD equal weight of being grouped
- Need to account for filters
- T1, t3 both satisfy Q1 and Q3
- T2, t4 do NOT satisfy Q1 or Q3
- Prefer to group AC
Column Grouping Equations

Skipping Effectiveness:

\[
\sum_{G_i \in G^q} |G_i \cap C^q| \cdot r_i^q.
\]

Objective Function:

\[
\text{COST}(q, G) = \sum_{G_i \in G^q} |G_i \cap C^q| \cdot r_i^q + \text{overhead}(q, G)
\]

Tuple Reconstruction Overhead:

\[
\text{overhead}(q, G) = \begin{cases} 
\sum_{G_i \in G^q} (r_i^q + \text{sort}(r_i^q)) & \text{if } |G^q| > 1 \\
0 & \text{otherwise}
\end{cases}
\]

\[
\text{COST}(W, G) = \sum_{q \in W} \text{COST}(q, G)
\]
Efficient Cost Estimation

- Difficult to obtain the number of rows that a query needs to scan after skipping in a Gi
- Exact computation is extremely expensive, propose more efficient estimation
  - Huge cost bottleneck from applying partitioning to Gi, clustering problem
- Use selectivity of query q as an estimation of the r value
  - Not accurate if query has a highly selective predicate
- Need to account for block-based skipping mechanism
  - Exploit a simple property of partitioning process-- preference to put rows with exactly same local feature vectors into the same block
Local Feature Selection

- Identifying Candidate Local Features:
  - \( \text{CandSet}(G) = \bigcup_{q \in W^G} F^q \)
  - \( F^q \) = features that subsume query \( q \)
  - \( W^G \) = set of queries that need to access data in column group \( G \)

- Feature Weighting and Selection:
  - Weight for local feature decided by importance in column group, not on all columns
  - \( \text{weight}(G, f) = | \{ q | f \in F^q \text{ and } q \in W^G \} | \) (\( f \) = given feature)
  - Number of distinctive feature vectors is a good indicator of whether the number of features selected is appropriate
    - Too few: add more features, does not affecting skipping of existing
    - Too many: existing features are very conflicting
Query Processing

1. Reading Data Blocks:
   - Check query against global features
   - Extract columns and pass to column catalog
   - Go through data blocks with union vectors
   - Read A from G1 and B, D from G2

2. Tuple reconstruction:
   - Tuple-ids stored as column within each block
   - Sort columns based on ids
   - Only return tuple t1, because t3 and t4 do not satisfy the full query
Results of Query Performance (Big Data)

- Figure a: vary parameter $k$ (number of columns accessed)
  - As $k$ increases, cost of GSOP-single increases, GSOP becomes SOP (70% accessed)
- Figure b: vary parameter $t$ (number of column templates)
  - GSOP outperforms GSOP-single and SOP, especially at low $t$
- Figure c: vary parameter $z$ (skewness of filter usage)
  - Greater $z$, less feature conflict, SOP can eventually outperform
- Figure d: vary parameter $s$ (query selectivity)
  - Increase $s$ results in higher execution cost for all 3, GSOP outperforms, single is worst
Results of Query Performance (TPC-H)

- Fig 7a: measure average number of actual data cells and tuple ids read by a test query.
- Fig 7b: we show the end to end query response time.
- Fig 8a: forming smaller number of column groups results in less reads of tuple ids for GSOP while reading slightly more data.
- Fig 8b: proposed column grouping techniques can balance the trade-off in GSOP better than GSOP-hy and GSOP-hc (35% better).
Objective Function Evaluation (TPC-H)

- Figure a: Efficiency comparison based on running time for different estimation approaches
  - Full computation is extremely time consuming compared to the estimations (a full day)
  - Sel. est. and Block est. take approximately the same amount of time (44 mins)
- Figure b: Quality comparison of different approaches based on workload cost
  - Sel. Est. involves most workload cost
  - Block est. only slightly more costly than full computation, thus the best choice
Loading Cost (TPC-H)

- Figure a: denormalized
  - GSOP spends most time in Phase 1
  - SOP has cheapest phase 2
- Figure b: normalized
  - Extra step of partial denormalization for GSOP
  - GSOP takes 2.6 times longer than the baseline
- Regardless, GSOP outperforms the other approaches
  - Worth the initial cost?
Query Performance (SDSS)

- Average query response times of 600 test queries against a baseline approach
- GSOP-hy and GSOP-hc are highly unreliable (do not take into account feature conflict or horizontal skipping)
- GSOP outperforms baseline by 4.7 times and outperforms SOP by 2.7 times
Evaluation

PROS:

● Good explanation of background & SOP framework
● Solid proof of better performance against multiple existing frameworks

CONS:

● Simplistic explanation of cost for trade-off -- does not explore impact of compression techniques
● Include more figures rather than refer to the same ones
● Need more experimentation comparing running time costs to overall performance improvements
Possible Next Steps

- Look into a dynamic layout for complex workloads that constantly change.
- Can we change the layout of data to optimize the ideal case where tuple overhead is 0 and skipping is effective?
- Explore better ways to handle normalized data-- some way to avoid the step of partial denormalization?