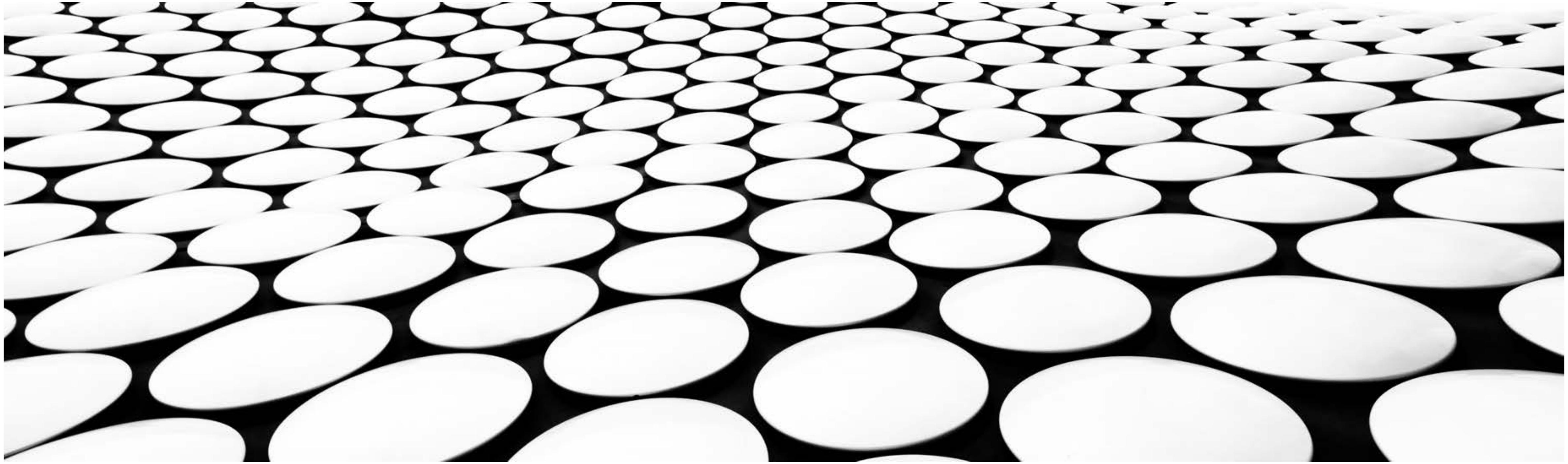
GENERALIZED SKIPPING-ORIENTED PARTITIONING

PRESENTED BY KAIJIE CHEN



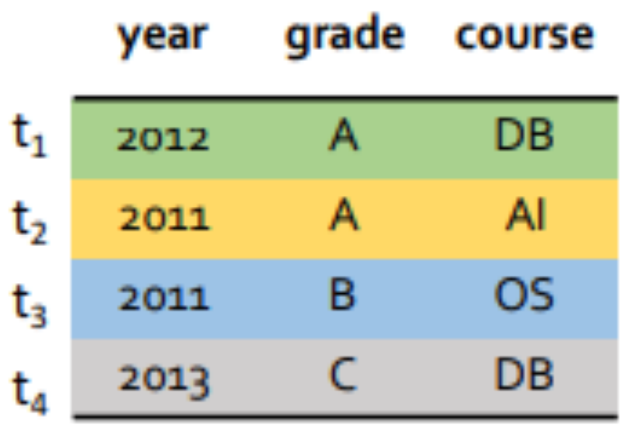
OUTLINE

DATA WAREHOUSE

- Focus more on OLAP workloads
- Infrequent, offline and batched data insertion
- Demanding in query efficiency
- Question: Which storage layout has a better support for OLAP workloads?

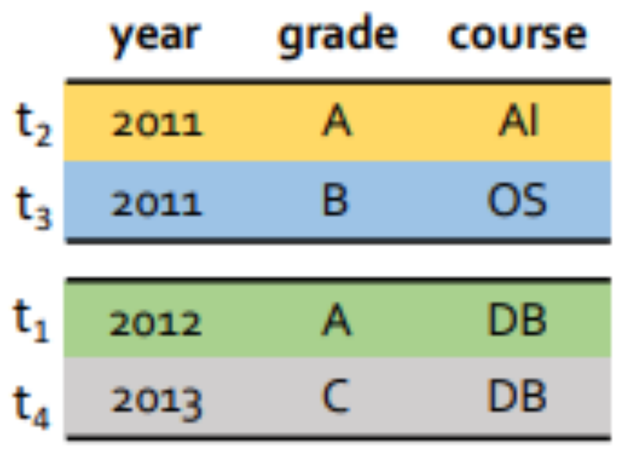
SKIPPING-ORIENTED PARTITIONING (SOP)

- Records are partitioned into blocks by filters (predicates)
- Blocks are skipped when query doesn't satisfy the filter.



	year	grade	course
t_1	2012	A	DB
t_2	2011	A	AI
t_3	2011	B	OS
t_4	2013	C	DB

(a) original data



	year	grade	course
t_2	2011	A	AI
t_3	2011	B	OS
t_1	2012	A	DB
t_4	2013	C	DB

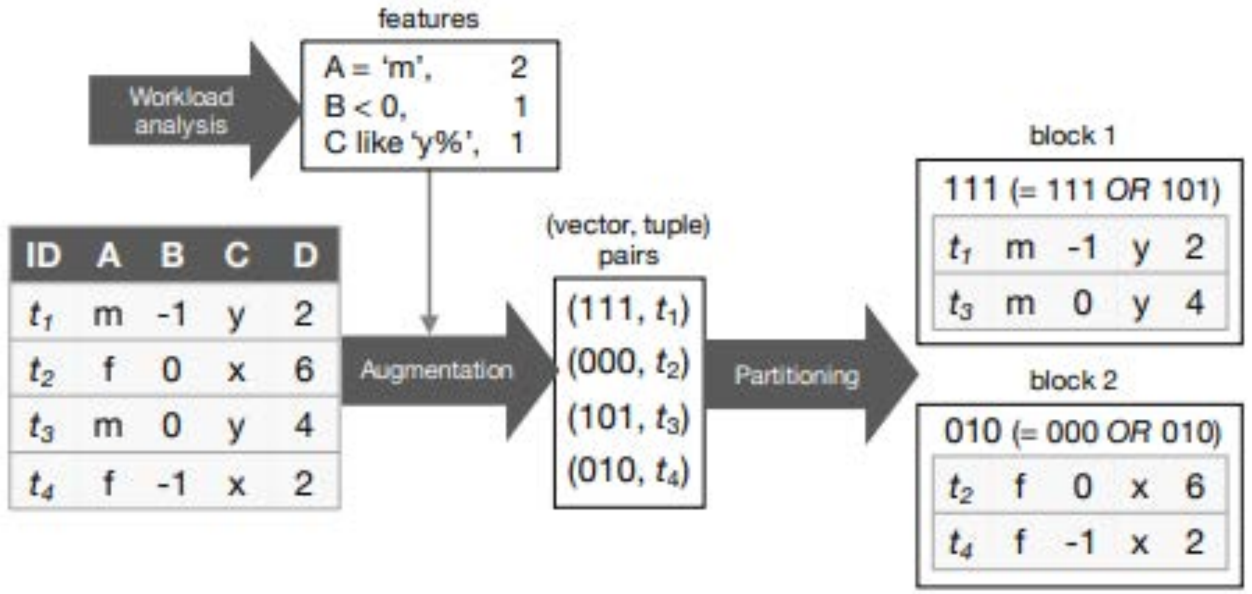
(b) a SOP scheme

CHARACTERISTICS OF REAL-WORLD ANALYTICAL QUERIES

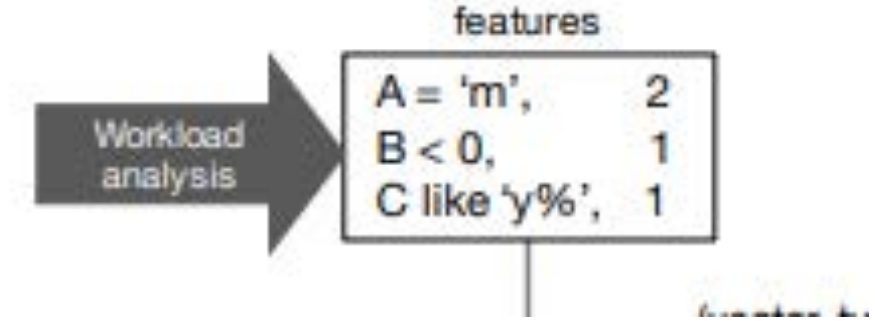
- **Filter Commonality:** A small set of filters are commonly used by many queries
- **Filter Stability:** Only a tiny fraction of query filters are newly introduced over time
- Idea: Optimize for future queries based on old ones

SOP FRAMEWORK: THREE STEPS

- Workload analysis
- Augmentation
- Partitioning



SOP STEP 1: WORKLOAD ANALYSIS



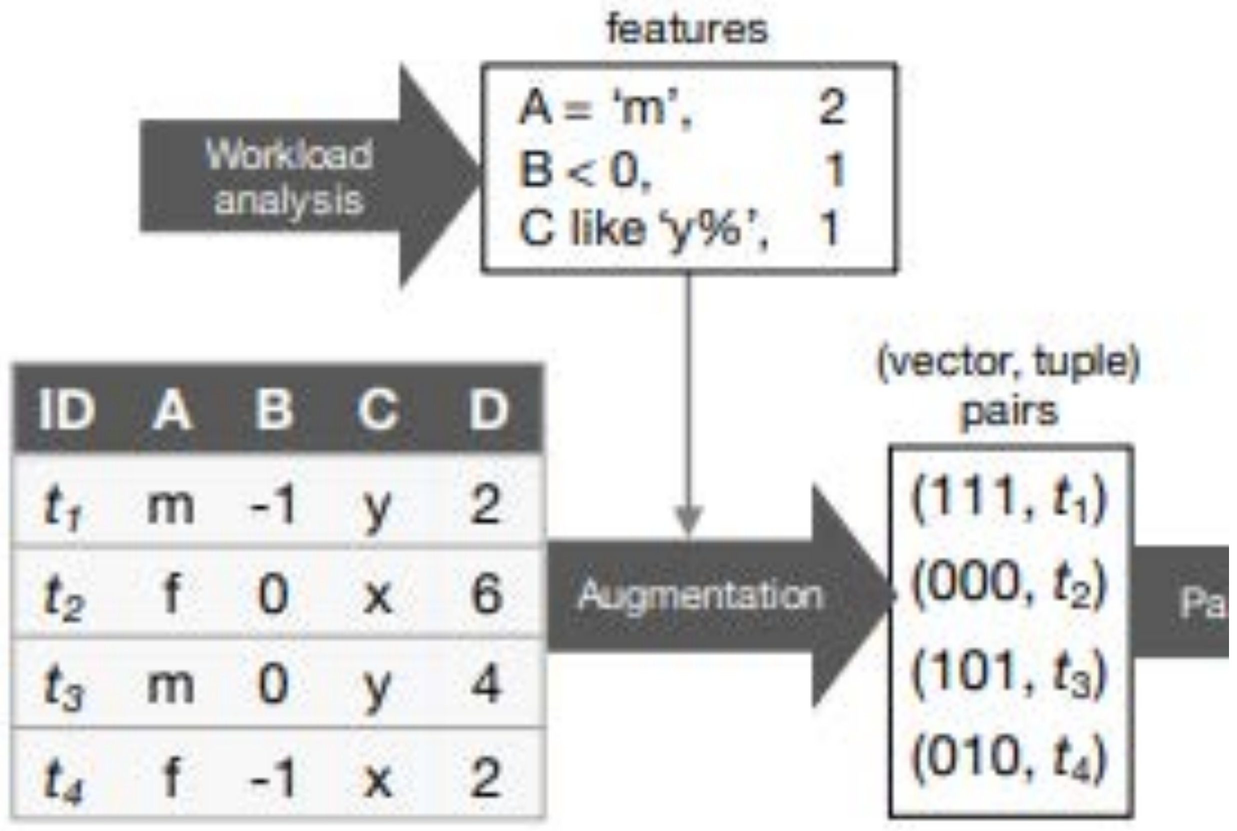
- This step extracts as **features** a set of representative filter predicates in the workload by using *frequent itemset mining*
- A **feature** can be a single filter predicate or multiple conjunctive predicates, which possibly span multiple columns. A predicate can be an equality or range condition, a string matching operation or a general boolean user-defined function (UDF).
- Frequent Dataset Mining: an essential task within data analysis for extracting frequently occurring events, patterns, or items in data.
- Workloads can be history query logs

SUBSUMPTION RELATION

- A feature **subsumes** a query when the feature is a more relaxed condition than the query predicates.
- That is to say, if feature P **subsume**s query Q , then $Q \Rightarrow P$
- For example, $B > 0$ **subsumes** $B > -1$
- SOP takes into **subsumption relations** when extracting features

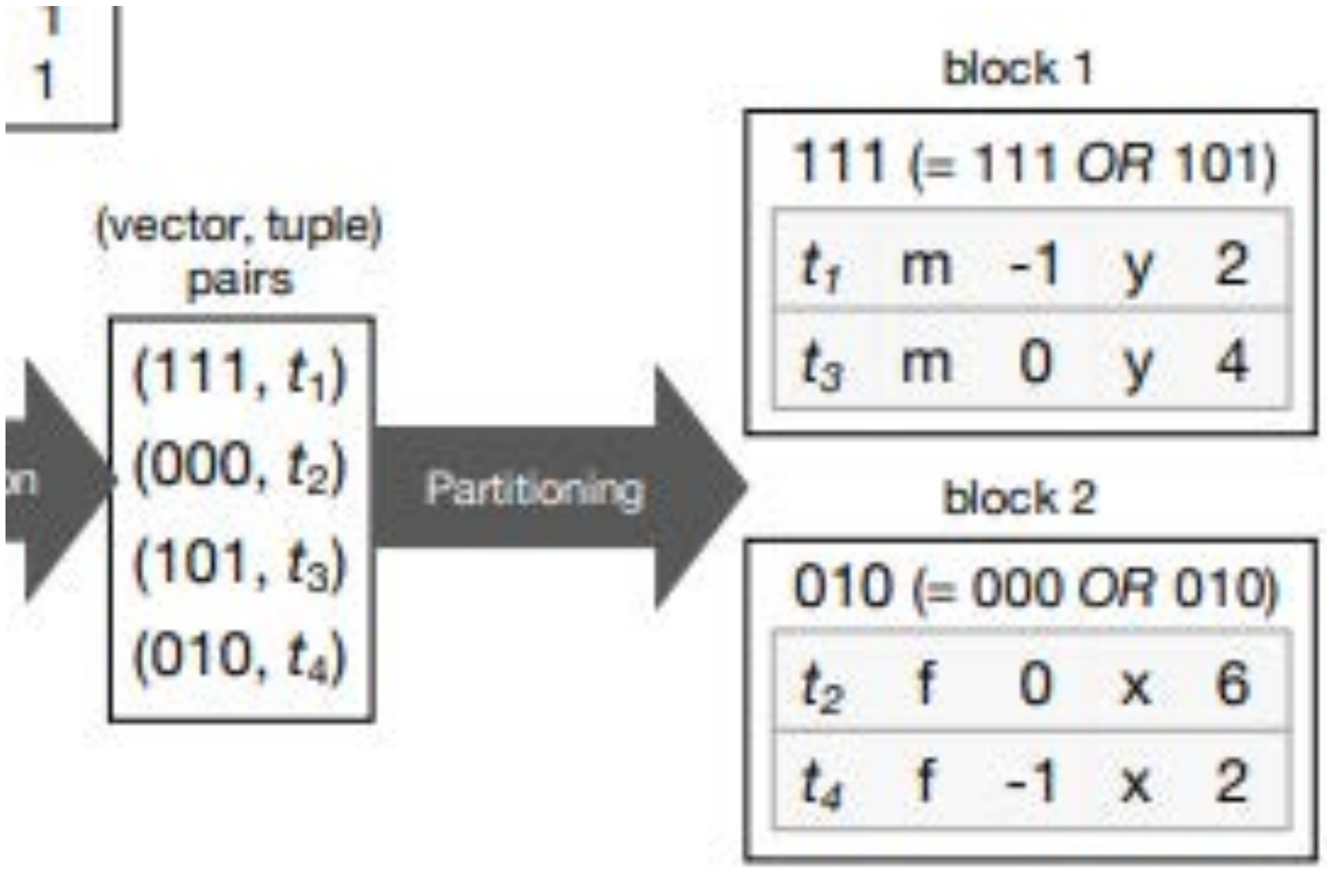
SOP STEP2: AUGMENTATION

- Data scanning
- Batch evaluates features and stores the evaluation results as an augmented **feature vector**
- **Feature Vector:** a m-dimensional bit vector, the i-th bit of which indicates whether this tuple satisfies the i-th feature or not



SOP STEP3: PARTITIONING

- Group the **(vector, tuple)- pairs** into **(vector, count)-pairs**
- A clustering algorithm is performed on the (vector, count)-pairs, which generates a **partition map.**
- Annotate each block with a **union vector**, which is a bitwise OR of all the feature vectors in the block

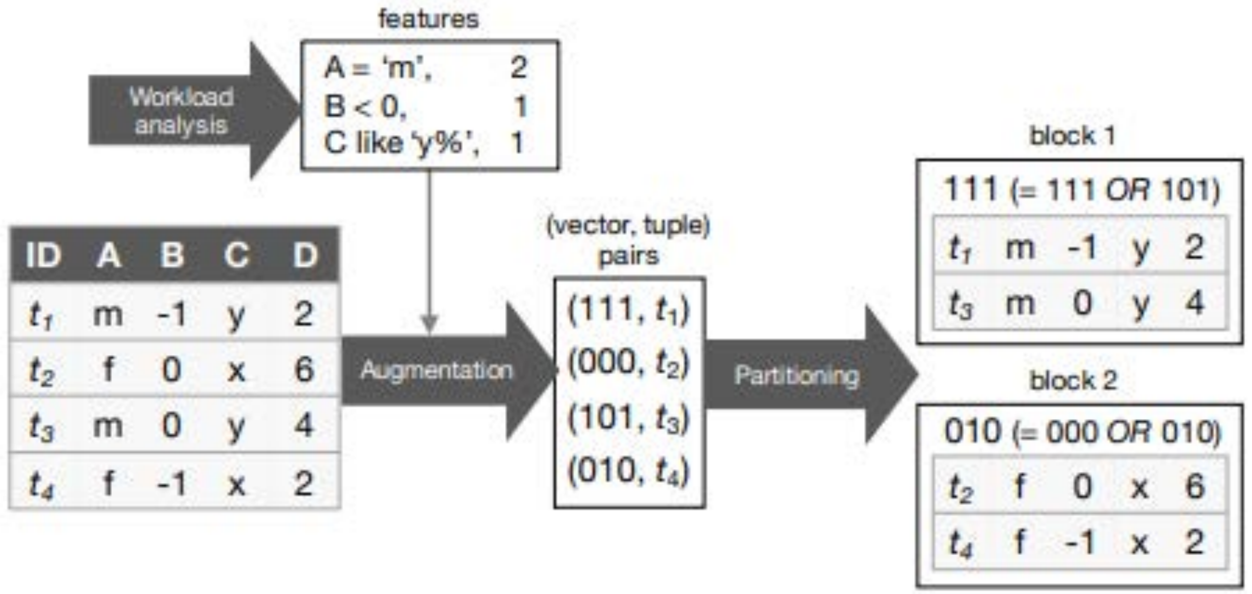


CLUSTERING ALGORITHM

- A research by the same authors: *Fine-grained Partitioning for Aggressive Data Skipping*
- Partition data into **fine-grained**, **balance-sized** blocks

SOP FRAMEWORK: THREE STEPS

- Workload analysis
- Augmentation
- Partitioning



PROBLEMS FOR SOP

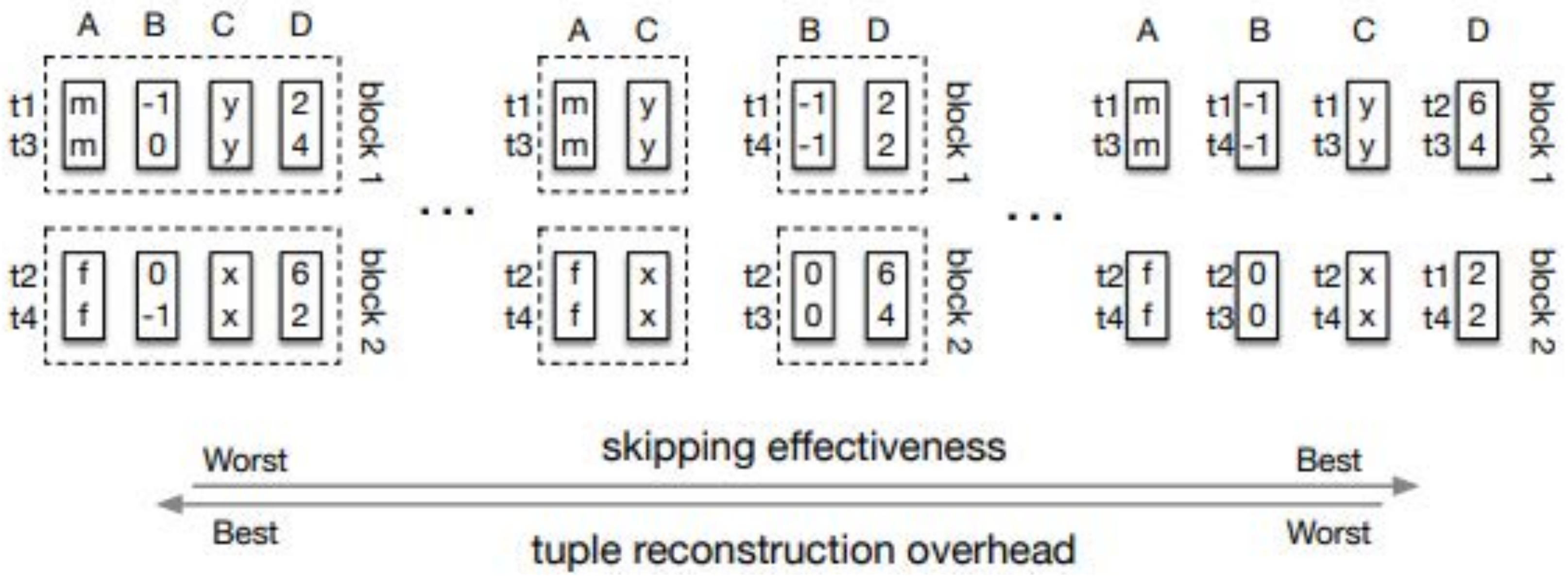
CAN YOU LIST SOME OF THE PROBLEMS FOR SOP?

PROBLEMS FOR SOP

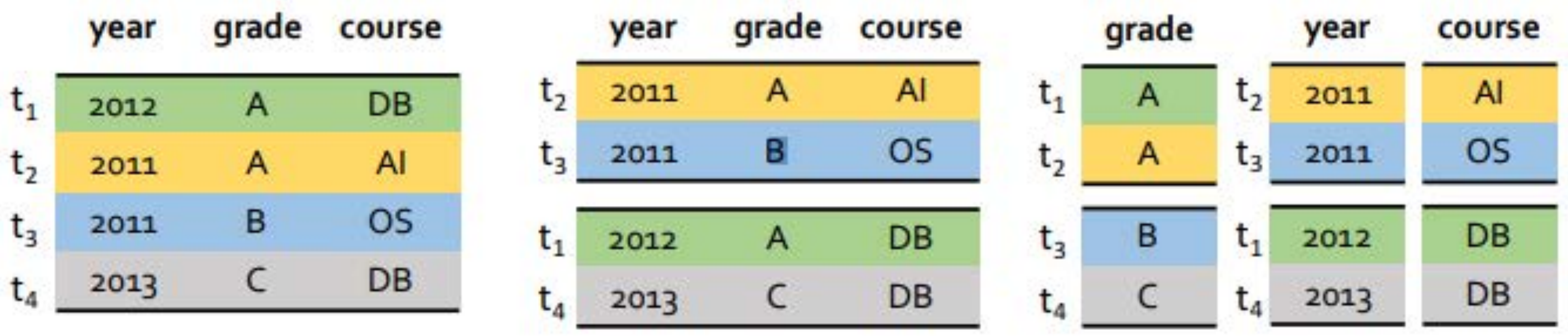
- Can only partition the data based on one field
- Based on **atomic-tuple constraint**, not fully utilize the features of column-based layouts.

PARTITIONING ON COLUMNAR LAYOUTS

- Good parts: Able to partition on each column independently, which increases data skipping efficiency
- Problem: As each column is reorganized, you need to associate a tuple id for each data cell, and use the tuple ids to reconstruct the tuple, which incurs tuple-reconstruction overhead
- How can we utilize the good parts and mitigate the problem?



A SOLUTION: COLUMN GROUPING



	year	grade	course
t_1	2012	A	DB
t_2	2011	A	AI
t_3	2011	B	OS
t_4	2013	C	DB

(a) original data

	year	grade	course
t_2	2011	A	AI
t_3	2011	B	OS
t_1	2012	A	DB
t_4	2013	C	DB

(b) a SOP scheme

	grade		year	course
t_1	A	t_2	2011	AI
t_2	A	t_3	2011	OS
t_3	B	t_1	2012	DB
t_4	C	t_4	2013	DB

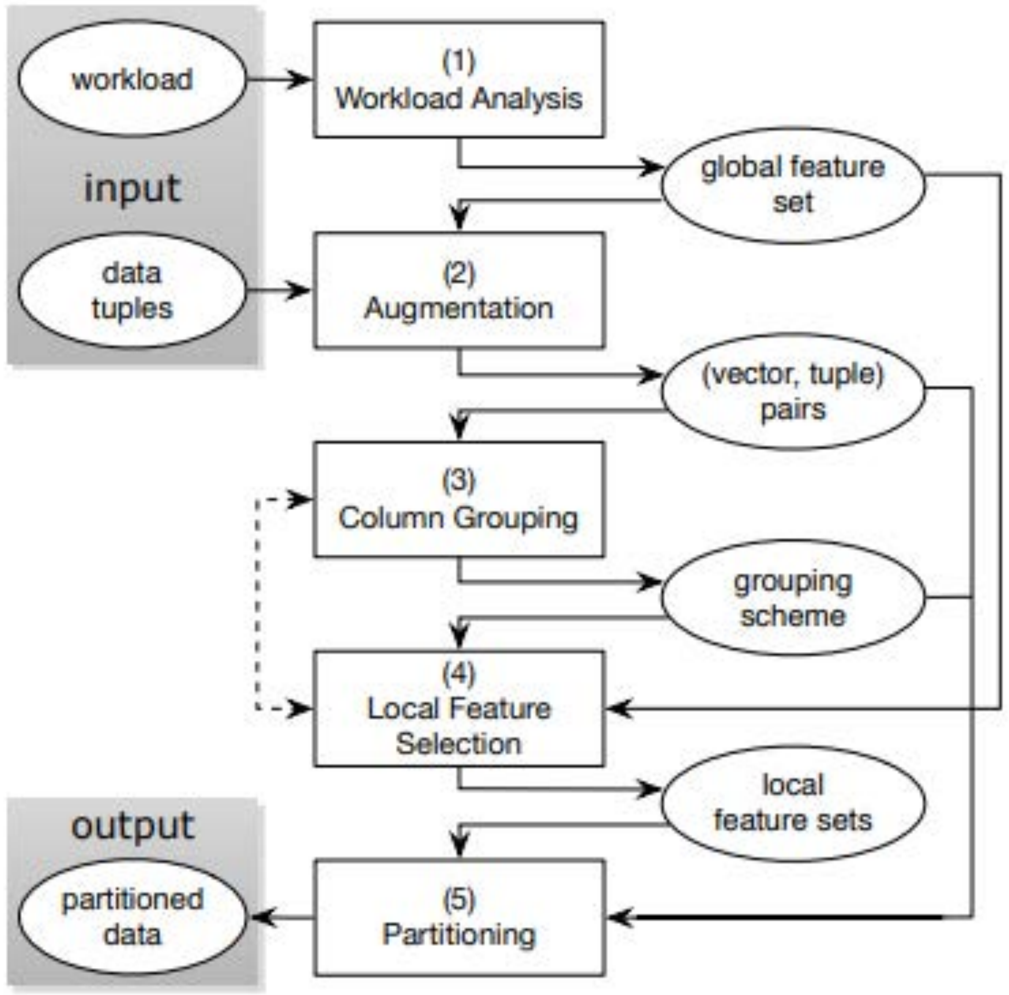
(c) a GSOP scheme

GENERALIZED SOP (GSOP)

- Generalizes SOP by removing the atomic-tuple constraint and allowing both horizontal and vertical partitionings.
- Use column grouping to achieve better data skipping efficiency while mitigate the tuple-reconstruction overhead
- Each group has its associate **features,** named **local features**

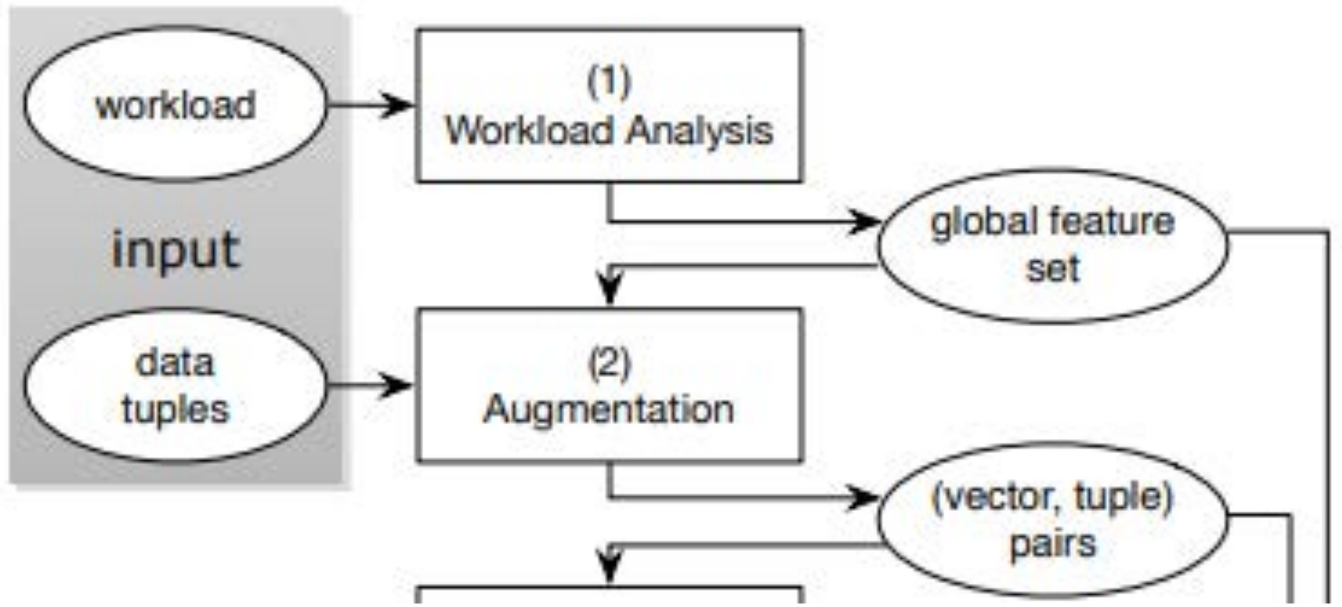
GSOP FRAMEWORK

- Workload Analysis
- Augmentation
- Column Grouping
- Local Feature Selection
- Partitioning

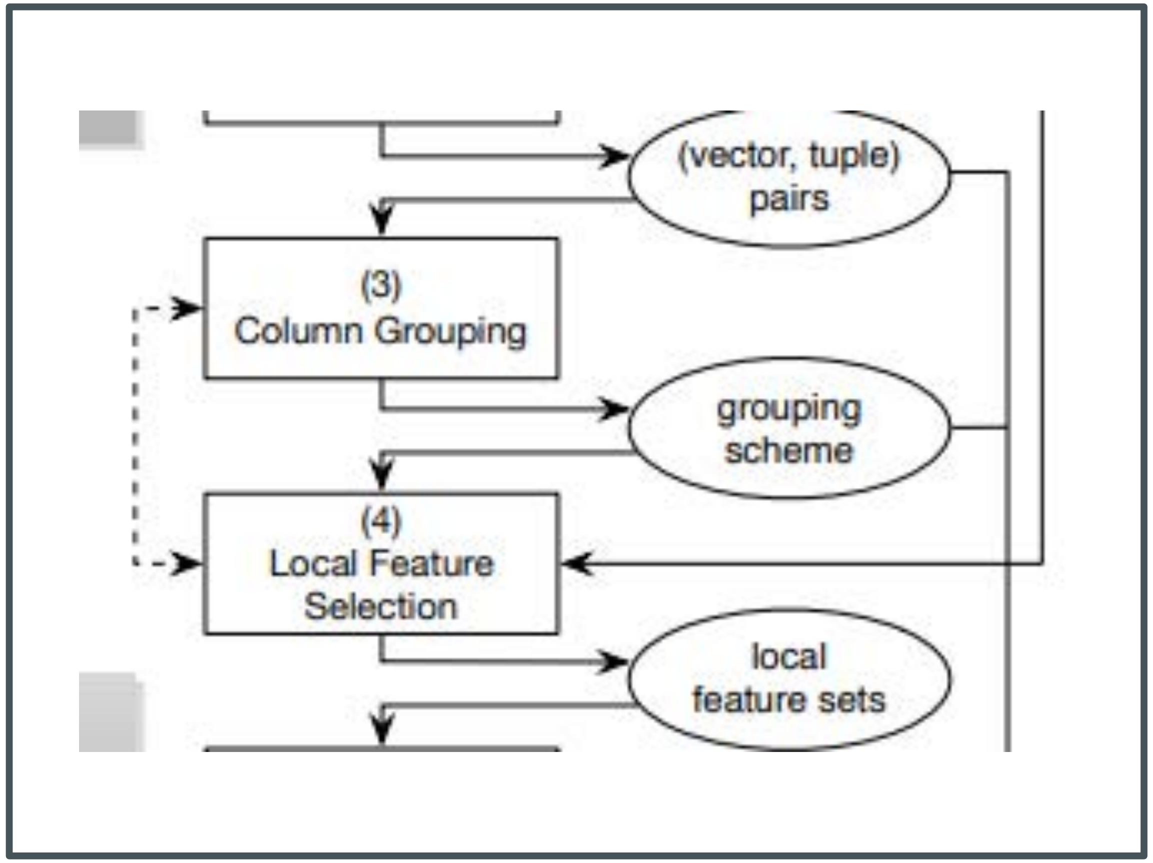


GSOP: WORKLOAD ANALYSIS AND AUGMENTATION

- **Workload Analysis:** The same as Workload Analysis in SOP Framework. The **features** extracted are called **global features** in GSOP
- **Augmentation:** The same as Augmentation in SOP Framework. Each tuple is augmented with a **global feature vector**



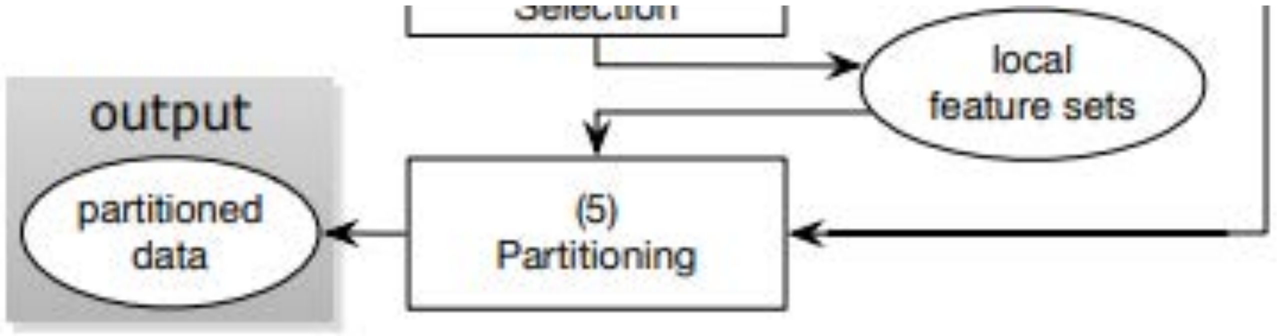
GSOP: COLUMN GROUPING AND LOCAL FEATURE SELECTION



- **Column Grouping:** Divide the columns into column groups based on an **objective function** that incorporates the trade-off between skipping effectiveness and tuple-reconstruction overhead
- **Local Feature Selection:** For each column group, we select a subset of global features as **local features.** These local features will be used to guide the partitioning of each column group.
- **Column Grouping** process calls **Local Feature Selection** repeatedly.

GSOP: PARTITIONING

- Similar to Partitioning in SOP
- To partition each column group, we need **local feature vectors** that correspond to the **local features.** Since a set of local features is a subset of global features (computed in Step 2), for each column group, we can project the global feature vectors to keep only the bits that correspond to the local features (using a bit mask)

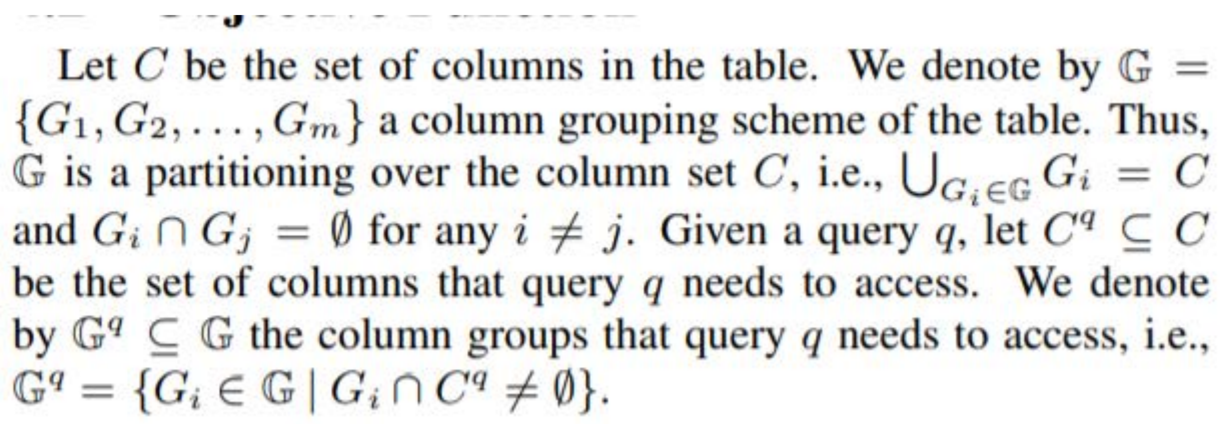


COLUMN GROUPING

- **Target:** Incorporate the opportunities of skipping horizontal blocks within each column group
- Use **Object Function** to decide whether to group or not

OBJECTIVE FUNCTION

- Summing up Skipping Effectiveness and Tuple-Reconstruction Overhead
- Some variables:



Let C be the set of columns in the table. We denote by $\mathbb{G} = \{G_1, G_2, \ldots, G_m\}$ a column grouping scheme of the table. Thus, $\mathbb{G}$ is a partitioning over the column set C, i.e., $\bigcup_{G_i \in \mathbb{G}} G_i = C$ and $G_i \cap G_j = \emptyset$ for any $i \neq j$. Given a query q, let $C^q \subseteq C$ be the set of columns that query q needs to access. We denote by $\mathbb{G}^q \subseteq \mathbb{G}$ the column groups that query q needs to access, i.e., $\mathbb{G}^q = \{G_i \in \mathbb{G} \mid G_i \cap C^q \neq \emptyset\}$.

SKIPPING EFFECTIVENESS

- **Data cell:** each column value of a tuple
- Assume scanning a **data cell** incurs a uniform cost 1
- For every column group $G_i \in \mathbb{G}^q$, query q needs to scan $|G_i \cap C^q|$ columns
- Let r_i^q denote the number of rows that query q needs to scan in group Gi
- The overall scan cost:

$$\sum_{G_i \in \mathbb{G}^q} |G_i \cap C^q| \cdot r_i^q.$$

TUPLE RECONSTRUCTION OVERHEAD

- When a query read from a single column group, no tuple-reconstruction is needed.
- When tuple-reconstruction is needed, data cells filtered in different column groups are sorted by tuple id and merge together on tuple id
- The total cost will be

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

OBJECTIVE FUNCTION

- Objective function for a single query q:

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

- Objective function for a whole workload W:

$$\text{COST}(W, \mathbb{G}) = \sum_{q \in W} \text{COST}(q, \mathbb{G})$$

EVALUATION OF r_i^q

Accurate evaluation: perform partitioning on the column group Gi

a) Extract local features for Gi

b) Project the global feature vectors onto local feature vectors

c) Apply partitioning to Gi based on the local feature vectors

EVALUATION OF r_i^q

- Bad parts for accurate evaluation: part c) is quite expensive, using cluster algorithm
- Replace c) with an upper-bound estimation:

$$n - b \cdot \sum_{v \in V_{skip}^q} \left\lfloor \frac{\text{count}(v)}{b} \right\rfloor$$

SEARCH STRATEGY

- Brute force search is too expensive.
- Use a heuristic algorithm instead.
- Initially, each column itself forms a group.
- We then iteratively choose two groups to merge until all columns are in one group.
- At each iteration, we enumerate all pairs of column groups and evaluate how their merge would affect the objective function.
- We then pick the merge that leads to the minimum value of the objective function.
- Starting from c columns, we need c iterations to merge all columns into one group.
- After these c iterations, we pick the iteration where the objective function has the minimum value and return the grouping scheme from that iteration.
- It’s a heuristic method, not always generate the most optimized grouping

LOCAL FEATURE SELECTION: CANDIDATE LOCAL FEATURES

- Select candidate local features for column group Gi: based on whether the global feature has a column within the column group or whether a column in Gi often co-exist with the global feature

$$\text{CandSet}(G) = \bigcup_{q \in W^G} F^q$$

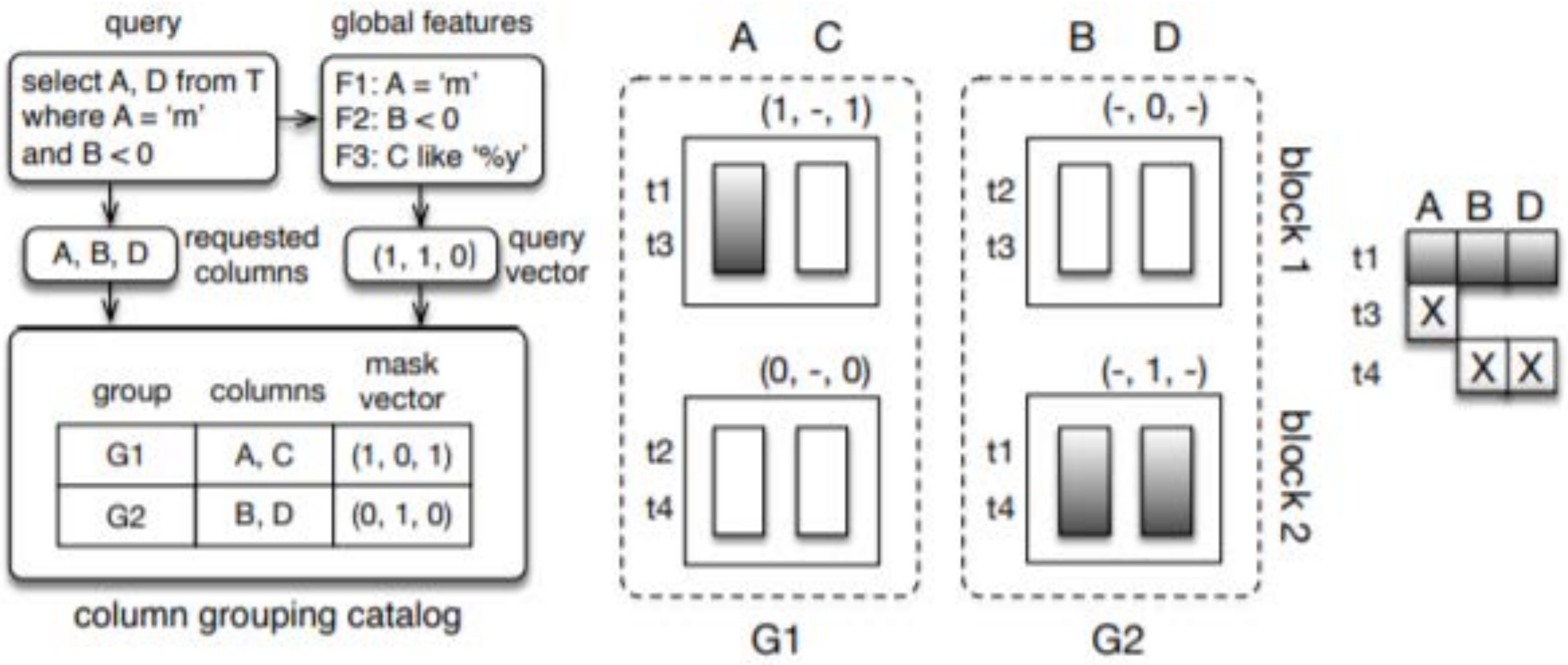
LOCAL FEATURE SELECTION: FEATURE WEIGHTING AND SELECTION

- Too expensive to make all the candidates local features
- Select the local feature based on frequency in the group, using a weight function

$$\text{weight}(G, f) = \left|\{q \mid f \in F^q \text{ and } q \in \breve{W}^G\}\right|.$$

- Regards the number of local features, some possible approaches discussed in paper

QUERY PROCESSING

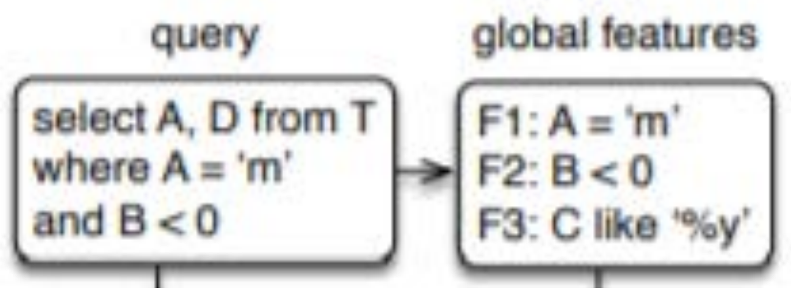


group	columns	mask vector
G1	A, C	(1, 0, 1)
G2	B, D	(0, 1, 0)

QUERY PROCESSING: READING QUERY BLOCKS

When a query arrives

1) Check query against global features, see which **global features** subsumes the query, and generate the **query vector**



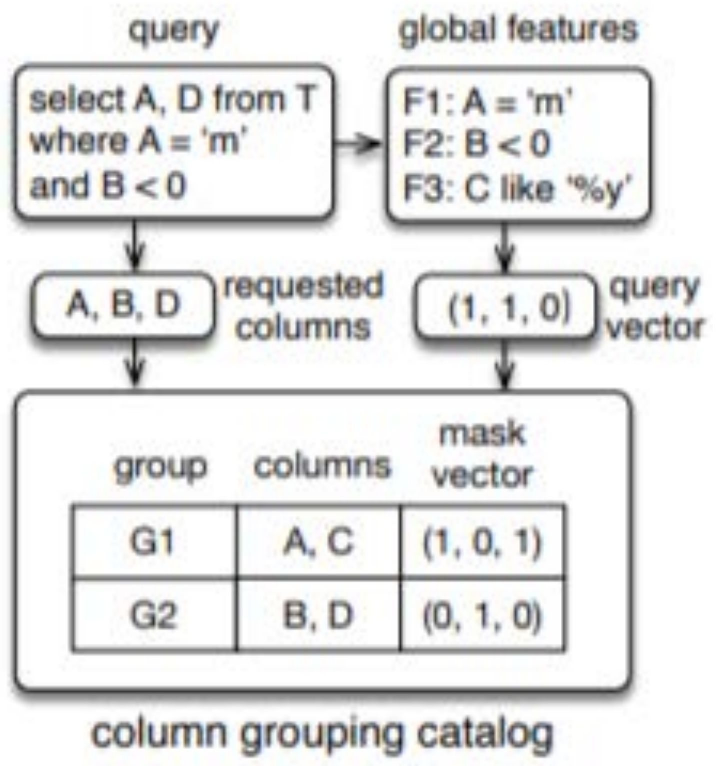
```
select A, D from T
where A = 'm'
and B < 0
```

```
F1: A = 'm'
F2: B < 0
F3: C like '%y'
```

Question: What query vector is to be generated?

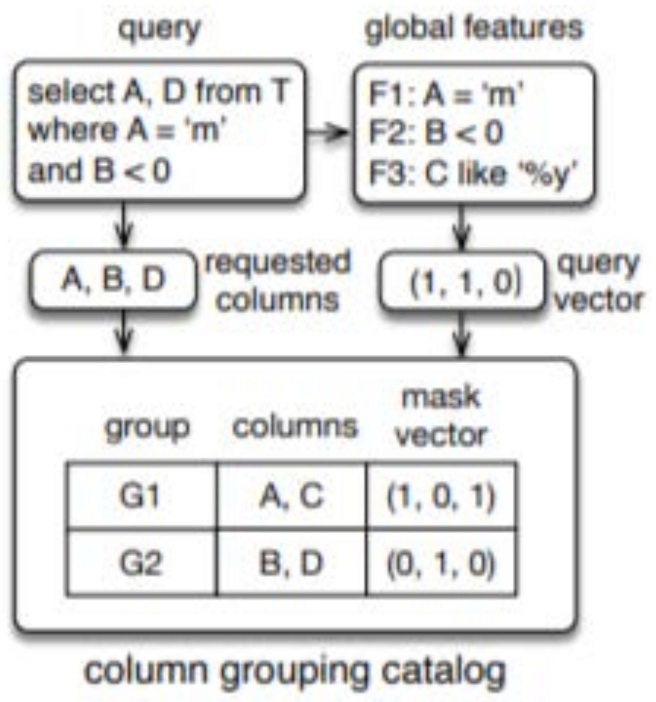
QUERY PROCESSING: READING DATA BLOCKS (CONT'D)

2) Extract the columns that this query needs to read and pass it to the column grouping catalog



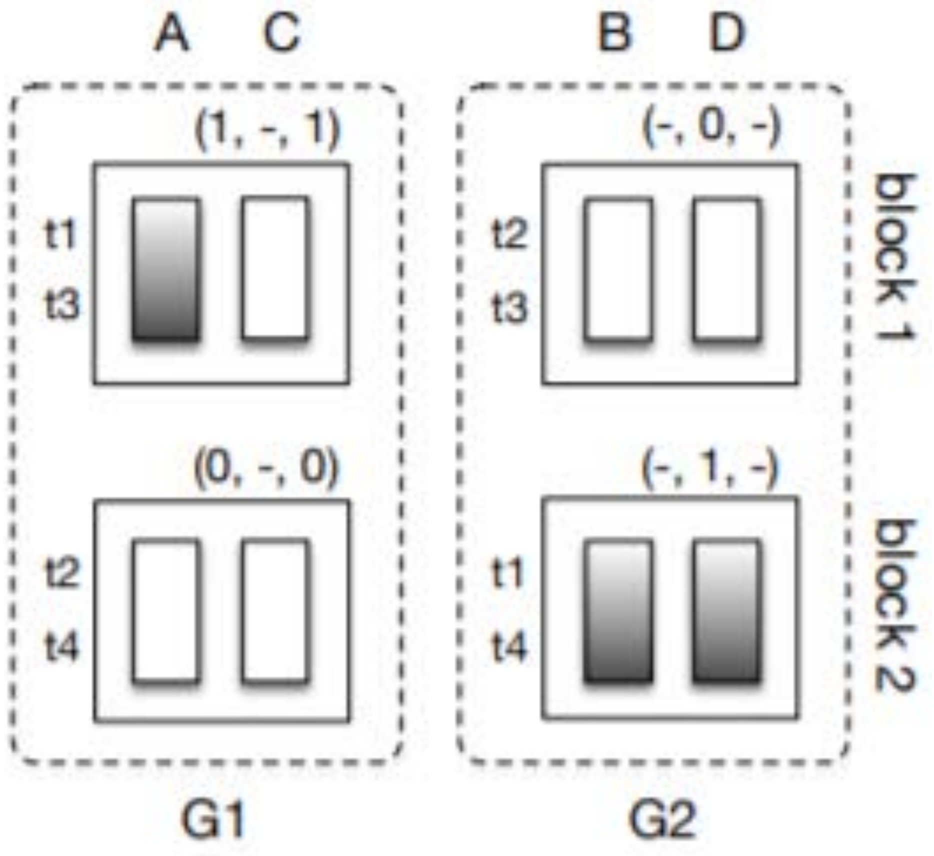
QUERY PROCESSING: READING DATA BLOCKS (CONT'D)

3) Go to the actual data block, using **mask vectors** for the block to filter the local features of the block inside the query vector



QUERY PROCESSING: READING DATA BLOCKS (CONT’D)

4) Query inside each block, using the masked query vector to skip partitions by compute a bitwise OR with the union vector



QUERY PROCESSING: TUPLE RECONSTRUCTION

After reading data from each block and the tuple-id, sort merge them to reconstruct tuples in memory.

EXPERIMENTS

- Environments: Spark, use Parquet as storage file format
- Benchmark Workloads: Big Data Benchmark, TPC-H, SDSS

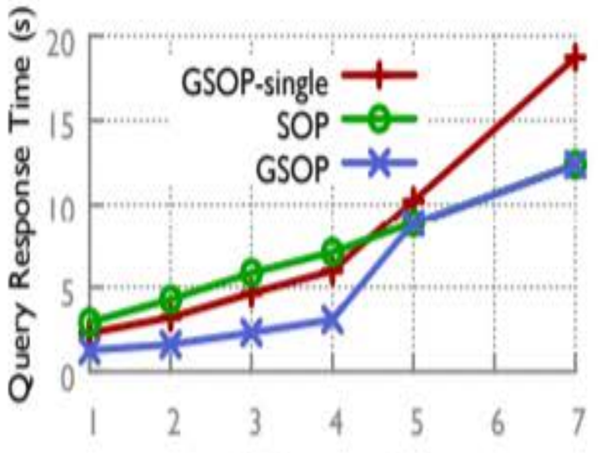
APACHE SPARK

- A unified analytics engine for big data processing, with built-in modules for streaming, SQL, machine learning and graph processing
- Is used by researchers to implement GSOP

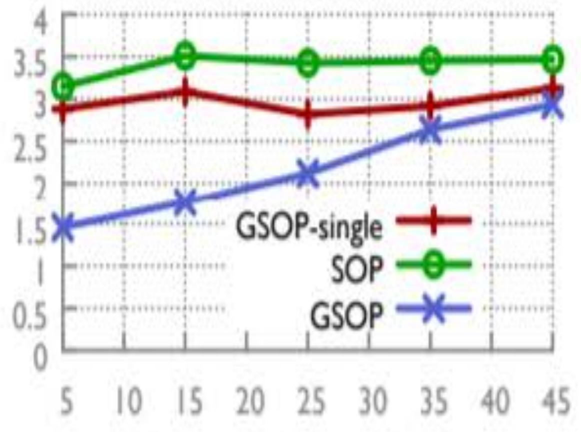
APACHE PARQUET

- A column-based data storage format
- Used by the researchers to store partition files

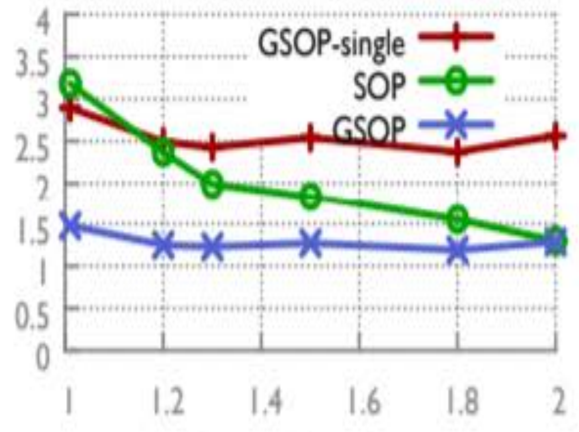
RESULTS FOR BIG DATA BENCHMARK



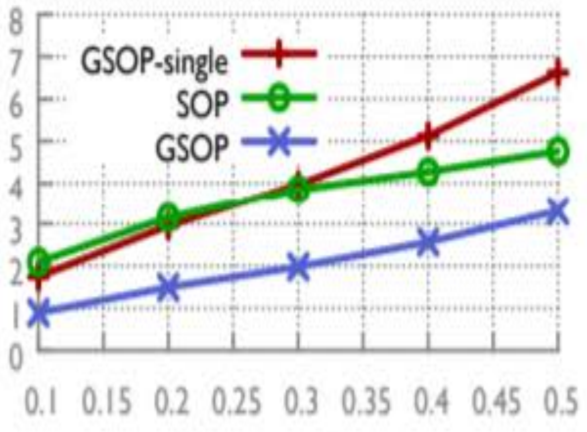
(a) Varying #Cols



(b) Varying # Column Templates



(c) Varying Filter Skewness



(d) Varying Selectivity

RESULTS FOR TPC-H

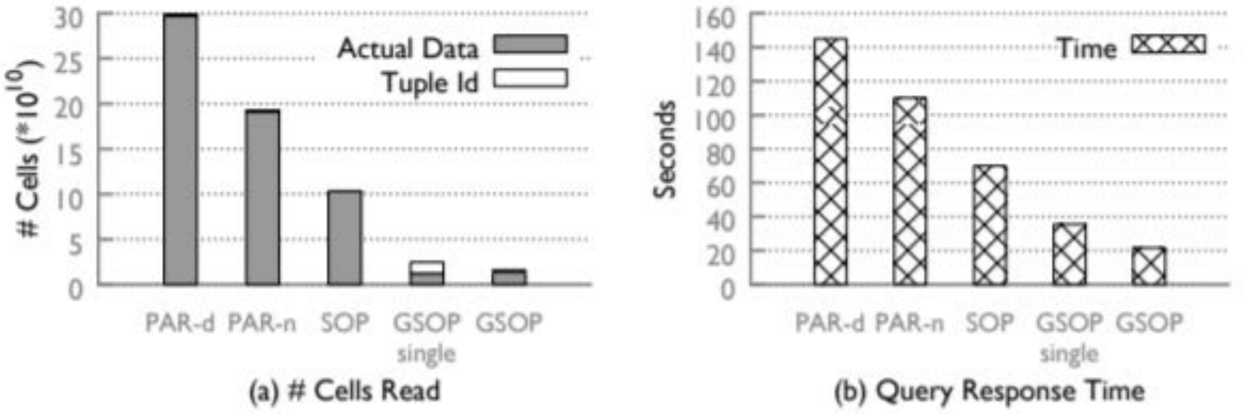


Figure 7: Query performance (TPC-H)

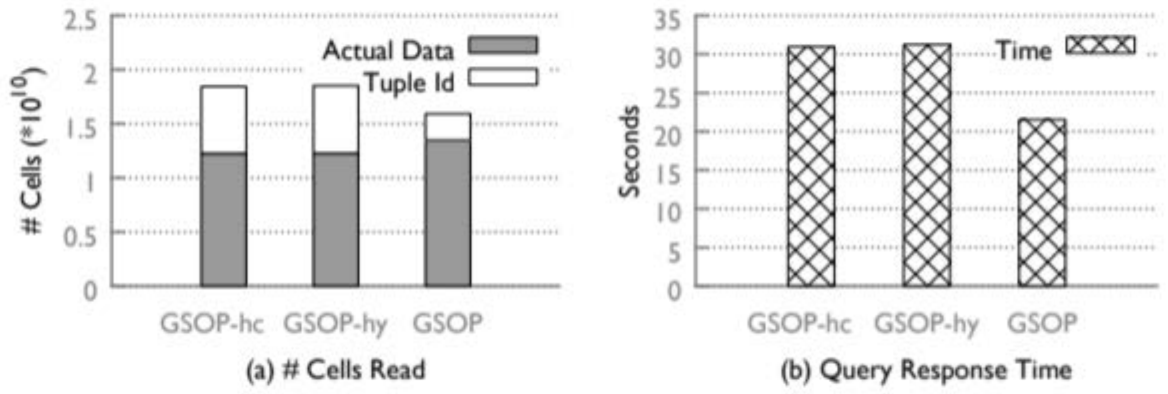


Figure 8: Query performance (TPC-H).

RESULTS FOR TPC-H (CONT'D)

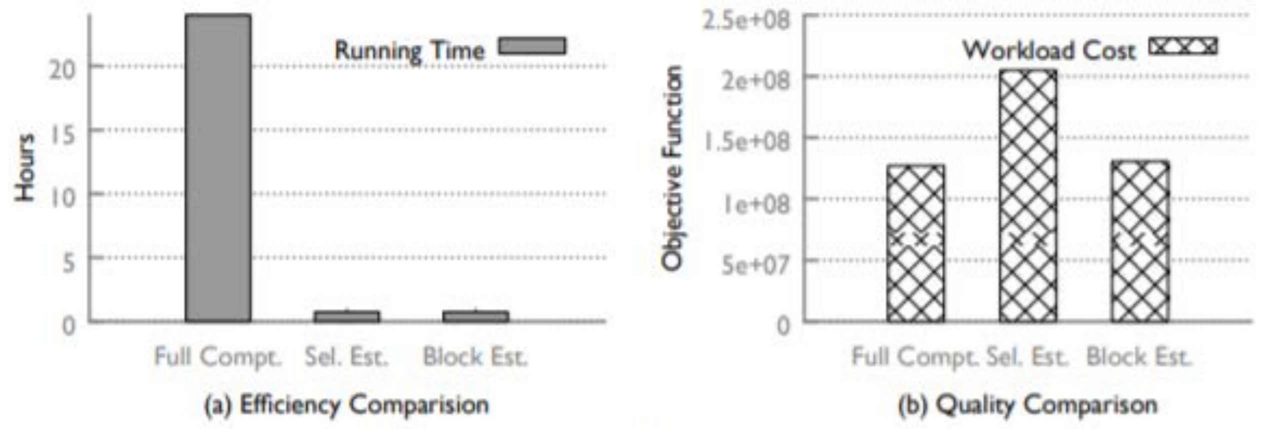


Figure 9: Objective function evaluation (TPC-H).

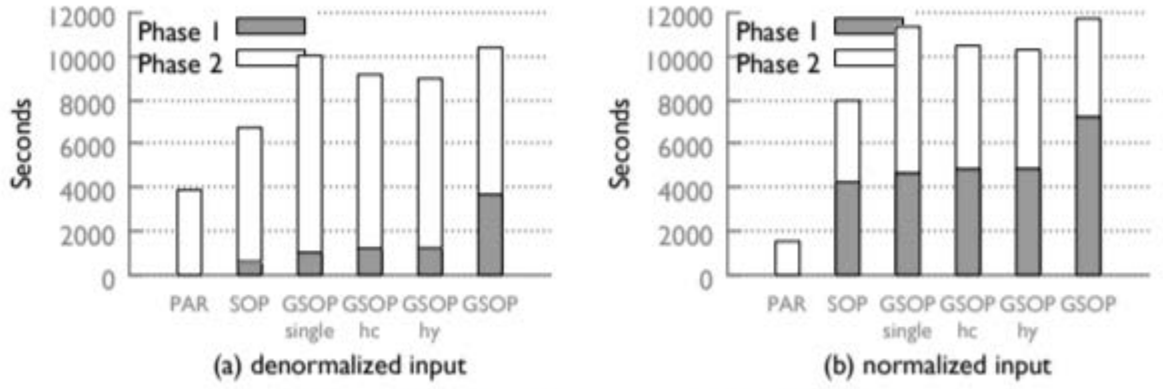


Figure 10: Loading cost (TPC-H).

RESULTS FOR SDSS

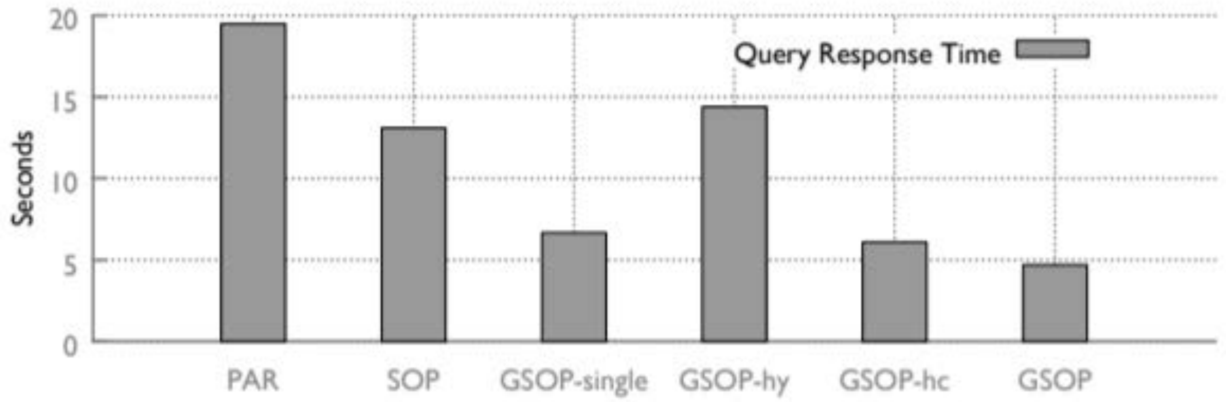


Figure 11: Query performance (SDSS).