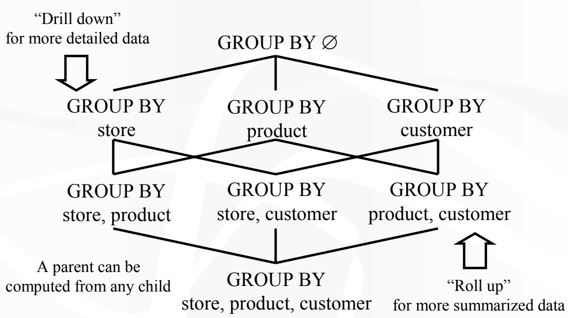
Selecting Views to Materialize

CPS 296.1
Topics in Database Systems

Data cube and OLAP

- Example data cube schema:
  Sale(store, product, customer, quantity)
  – Store, product, customer are dimension attributes
- Example OLAP query:
  SELECT product, store, SUM(quantity)
  FROM Sale
  GROUP BY product, store;
  – Lots of summarization
  – Cost of aggregation dominates
  ➢ Materialize aggregates to improve query performance

Aggregation view lattice



Selecting views to materialize

- Factors in deciding what view to materialize
  – What is its storage/update cost?
  – Which queries can benefit from it, and how much?
- Trade-off
  – GROUP BY ∅ is small, but not useful to most queries
  – GROUP BY store, product, customer is useful to most queries, but too large to be beneficial

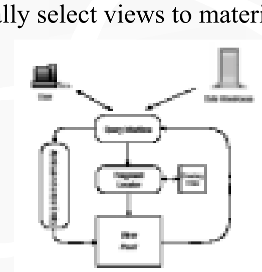
➢ Harinarayan et al. "Implementing Data Cubes Efficiently." *SIGMOD*, 1996

Limitations of static approach

- Previous work assumes fixed workloads
- But things change overtime
  – User interests (queries)
  – Data characteristics
  – Space/time constraints
- Periodic re-calibration is necessary
- Questionable performance guarantees

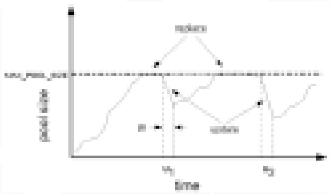
DynaMat

- Kotidis and Roussopoulos. "DynaMat: A Dynamic View Management System for Data Warehouses." *SIGMOD*, 1999
- Dynamically select views to materialize



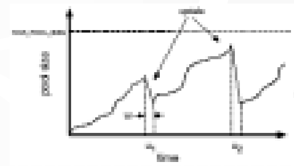
Space and time bounds

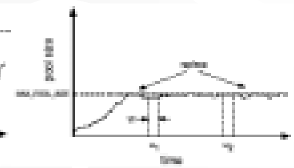
- Pool size increases between updates
- Space bound: new query results compete with cached results for the limited space
- Time bound: results are evicted from the pool because of limited update window



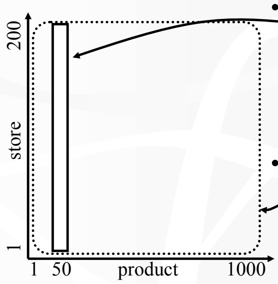
Space- and time-bound cases

- Time-bound case: not enough time to update all materialized results
- Space-bound case: not enough space to materialize all query results





Range query in data cube



- Query:

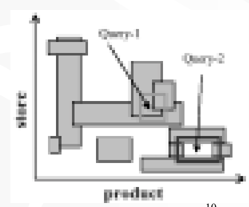
```
SELECT product, store,
    SUM(quantity)
FROM Sale
WHERE product = 50
GROUP BY product, store;
```

- Lattice node:

```
SELECT product, store,
    SUM(quantity)
FROM Sale
GROUP BY product, store;
```

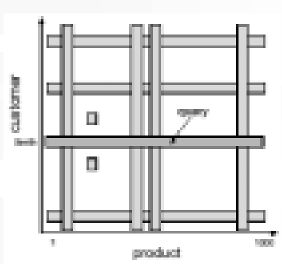

What should get materialized?

- Selecting the logical unit of materialization is important
  - Operational overhead should be minimum (lookup and maintenance)
  - Query performance should not be compromised
- Example: arbitrary range fragments
  - May result in too many small fragments
  - Re-using fragments gets complicated (overlap, holes)
  - Maintenance is difficult



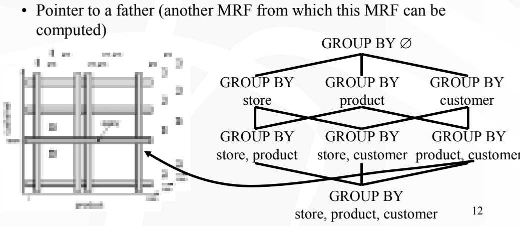
MRF

- Multidimensional Range Fragments (MRF's)
  - Ranges are either fully open or a single value
- Easier to handle than arbitrary range fragments



Directory index

- One R-tree for each view in the lattice
  - One index entry for each MRF of this view
    - MRF description
    - Statistics (e.g., number of accesses, creation time, last access time, etc.)
    - Pointer to a father (another MRF from which this MRF can be computed)



Answering query using MRF’s (slide 1)

- Given a query q, check the R-tree index for the corresponding lattice view
- Example
  - q = { product: $(-\infty, +\infty)$, store: (), customer: Smith }
  - Check GROUP BY product, customer
  - MRF { product: 50, store: (), customer: $(-\infty, +\infty)$ }
    - Does not covers q
  - MRF { product: $(-\infty, +\infty)$, store: (), customer: Smith }
    - Covers q; exact match
  - MRF { product: $(-\infty, +\infty)$, store: (), customer: $(-\infty, +\infty)$ }
    - Needs additional filter to answer q; not considered by the paper

Answering query using MRF’s (slide 2)

- If no MRF’s were found, check the R-tree indexes for more detailed lattice views
- Example
  - q = { product: $(-\infty, +\infty)$, store: (), customer: Smith }
  - Check GROUP BY product, store, customer
  - MRF { product: $(-\infty, +\infty)$, store: 10, customer: Smith }
    - Does not cover q
  - MRF { product: $(-\infty, +\infty)$, store: $(-\infty, +\infty)$, customer: Smith }
    - Covers q; needs additional aggregation

Answering query using MRF’s (slide 3)

- If an MRF f matches q exactly, return the content of f directly
- If no exact match exists, pick the best MRF f to answer q according to some cost model
  - f is the father of q
- If no MRF can answer q, compute q from base tables at the warehouse
- Result of q may be materialized as an MRF

Goodness of MRF’s

- LRU (Least Recently Used)
  - goodness(f) = last_access_time(f)
- LFU (Least Frequently Used)
  - goodness(f) = access_frequency(f)
- SFF (Smaller Fragment First)
  - goodness(f) = size(f)
  - Larger MRF’s are more likely to be hit by a query
  - Larger MRF’s imply fewer MRF’s to manage
- SPF (Smaller Penalty First)
  - goodness(f) = access_frequency(f) · cost(f) / size(f)
  - cost(f) is estimated as the cost of computing f from its parent

View management

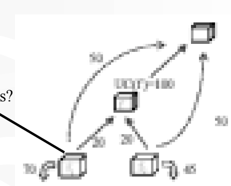
- Query time
  - If there is not enough space to materialize the new result, evict MRF’s with lowest goodness
- Update time
  - For each MRF f compute minimum update cost UC(f)
    - Re-compute f from its father, or
    - Incrementally maintain f using base table deltas

    How about father deltas?
  - If there is not enough time to update all MRF’s, evict some MRF’s
    - Which ones? How about ones with lowest goodness?

Time-bound update plan

- Compute reduction in update cost after evicting f: $U_{delta}(f)$
  - Heuristic: forward father pointers of orphans
  - Example: $U_{delta}(f)$ = 100 – ((50–20) + (45–20)) = 45

How about alternative fathers?



- Evict f with $U_{delta}(f) > 0$ based on goodness

Performance metrics (slide 1)

- Hit ratio = $(\Sigma_i h_i)/(\Sigma_i r_i)$
  - r_i is the number of times that q_i is run
  - h_i is the number of times that q_i is satisfied in cache
  - But the cost of a miss varies widely!
- Cost saving ratio = $(\Sigma_i c_i h_i)/(\Sigma_i c_i r_i)$
  - c_i is the cost of executing q_i without cache
  - But the cost of a hit also varies widely!
    - Exact match; compute from fathers…

Performance metrics (slide 2)

- Detailed cost saving ratio = $(\Sigma_i s_i)/(\Sigma_i c_i)$
  - s_i is the cost saving for q_i
  - $s_i = 0$ if q_i cannot be answered by the view pool
  - $s_i = c_i$ if there is an exact match for q_i in the pool
  - $s_i = c_i - c_{fi}$ if f is used to answer q_i

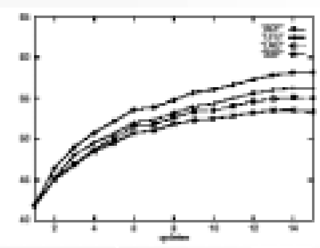
➢ Sloppy notation: each occurrence of a query should get a different i

Experiments

- Synthetic query load
  - Uniform queries on lattice views
  - 80-20 law for values
- Space bound: 2% of the size of the warehouse
- Time bound: 2% of the time to update the full warehouse

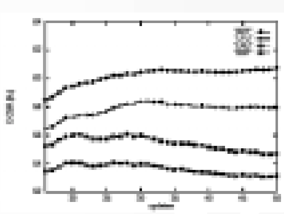
Comparing goodness policies (slide 1)

- SPF > LFU > LRU > SSF
- Saving increases quickly as the view pool warms up
  - Quite substantial for just 2% extra space and time

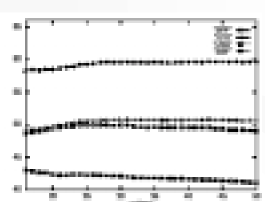


Time-bound case

Comparing good policies (slide 2)



Time-bound case

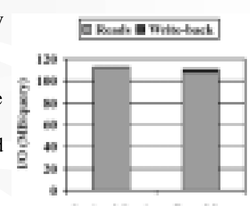


Space-bound case

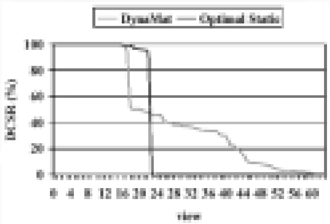
- Savings eventually flatten out

DynaMat vs. optimal static view selection (slide 1)

- Calculated "optimal" static view selection
  - Calculation took 3 days!
  - Time bound: 2% of the warehouse update time
  - Only full lattice views are selected (no fragments)
- DynaMat
  - Same time bound
  - Space bound is set to the size of the optimal view collection
- Same overall performance

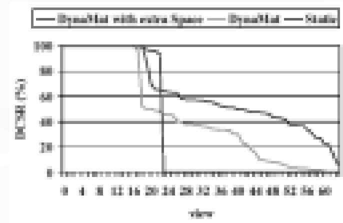


DynaMat vs. optimal static view selection (slide 2)



- Optimal static selection “ignores” many views altogether
- DynaMat provides savings for almost all views

DynaMat vs. optimal static view selection (slide 3)



- Optimal static selection cannot make use of extra space
  - Why?
- DynaMat increases savings because of extra space
  - Intuition?

DynaMat vs. optimal static view selection (slide 4)

- DynaMat also outperforms static view selection in cases of
  - Skewed workloads
    - Example: queries gradually increase the number of GROUP BY columns
  - Roll-up/drill-down workloads
    - Typically OLAP queries tend to be followed by roll-up/drill-down queries on the same data
    - Roll-up queries can be compute from the result of the original query

Conclusion

- Dynamic/adaptive algorithms work surprising well in practice, despite their simplicity
- Simplicity is actually necessary in this case to keep the run-time overhead low
  - Give up arbitrary range fragments
  - Give up multiple father pointers
  - …

➢ Self-tuning and self-administering DBMS