WebView Materialization and Maintenance

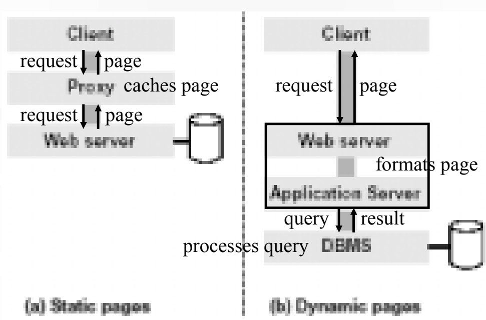
CPS 296.1

Topics in Database Systems

Roadmap

- Where to materialize dynamic Web content
  - Labrinidis and Roussopoulos. “WebView Materialization.” *SIGMOD*, 2000
- When to refresh materialized Web content
  - Labrinidis and Roussopoulos. “Update Propagation Strategies for Improving the Quality of Data on the Web.” *VLDB*, 2001

Multi-tier Web architecture



WebView caching

- WebView: a page (or fragment of a page) dynamically generated from data in DBMS
  - Base data
  - Query results
  - Result pages

  (Base data → Query results: Query at DBMS; Query results → Result pages: Format at Web server)
- Question: What and where to materialize?
  - Virtual (do not materialize)
  - Materialize query results inside the DBMS
  - Materialize result pages at the Web server

Virtual

- Access
  - Query at DBMS
  - Format at Web server
- Update
  - Update base tables at DBMS
- Contention at DBMS between queries and updates

Materialize inside DBMS

- Access
  - Read materialized query result at DBMS
  - Format at Web server
- Update
  - Update base tables at DBMS
  - Update materialized query results at DBMS
    - Re-compute affect queries, or
    - Incrementally maintain materialized results
- Contention at DBMS between reading and updating of materialized query results

Materialize at Web server

- Access
  - Read materialized result page at Web server
- Update
  - Update base tables at DBMS
  - Re-compute queries at DBMS
  - Re-format materialized result pages at Web server
  - ➢ Last two steps can be pipelined
  - ➢ Incremental maintenance is very difficult, if at all possible
- ➢ Contention at Web server between reading and writing materialized result pages

Performance metric: response time

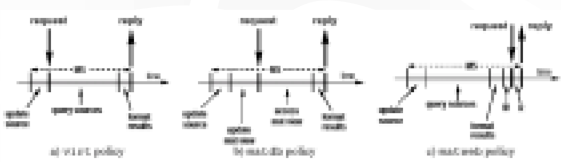
Average response time of an access

- Not simply the average access time over all WebViews
- Account for different access frequencies
  - Access time of a WebView is weighted by its access frequencies
- Account for contention between accesses and updates
  - DBMS is likely to be the bottleneck
  - If all WebViews are materialized at Web server, then updates do not impact accesses
    - Contention at Web server between reading and writing pages is ignored
  - Otherwise, update time is also counted

Performance metric: staleness

Staleness of WebViews

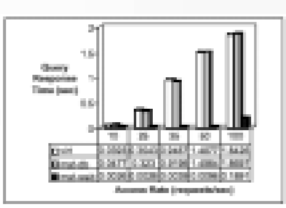
- Virtual policy does not necessarily provide lowest staleness, because query time also contributes to staleness



Experimental setup

- Synthetic load: single-table selections on index columns
  - Materialization and incremental maintenance do not buy us much in this case
  - ➢ Expect bad performance for the mat-db policy
- Updater processes run in background to refresh WebViews
- Interesting tidbits
  - Do not spawn a process to handle each request (like CGI does) → an order of magnitude performance improvement
  - Database connection pooling → another order of magnitude performance improvement

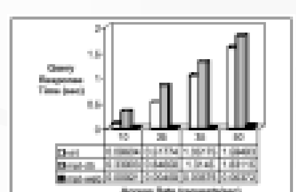
Scaling up access rate (no updates)



(a) No updates

- Mat-web definitely wins because it does not repeat any work
- Mat-db ≈ virtual, indicating that for this query load, re-computing queries is as cheap as reading pre-computed results (or the cost is dominated by the overhead of interacting with a DBMS)

Scaling up access rate (with updates)



(b) 5 updates/sec

- Mat-db is even worse than virtual, because of the extra work of refreshing materialized query results
- Mat-web wins again: The trip to the DBMS and/or the re-formatting of the result page are worth saving

Other experiments

- Scaling up the update rate
  - Access time under mat-web hardly changes because updates are handled in background
  - Virtual is worse because of access/update contention at DBMS
  - Mat-db is even worse because of the extra work of refreshing materialized query results
- Scaling up the number of WebViews
  - Also making 10% of the queries simple two-table key-joins
  - ➢ Materialization makes more sense in this case
  - Mat-db works better, but is still bad with lots of updates
- Scaling up the WebView size, Zipf distribution, mixing three policies, etc.

Staleness

- Inferred from the results of the experiments
  - Under light load, virtual provides lowest staleness
  - With heavy load, mat-web works better because it is able to maintain fast response time

Summary of WebView materialization

- Experiments indicate mat-web is the best
- But have the experiments covered all practical cases?
  - If queries are more expensive to compute (e.g., aggregates), then mat-db should outperform virtual
  - If queries are cheap to maintain incrementally yet expensive to re-compute (e.g., aggregate), then mat-db could outperform mat-web
  - ➢ Perhaps the decision of which materialization policy to use should be made on a per-WebView basis?

Roadmap

- Where to materialize dynamic Web content
  - Labrinidis and Roussopoulos. "WebView Materialization." *SIGMOD*, 2000
- ➢ When to refresh materialized Web content
  - Labrinidis and Roussopoulos. "Update Propagation Strategies for Improving the Quality of Data on the Web." *VLDB*, 2001

Serving dynamic Web content

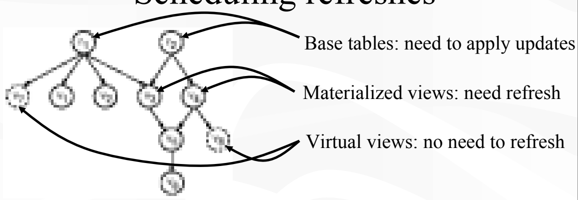
- Too many accesses?
  - WebView materialization
- Too many updates? Surges in update rate?
  - Schedule refreshes of materialized WebViews to maximize their freshness
  - ➢ Freshness should degrade gracefully if there are not enough resources to keep up with the updates
  - ➢ Freshness should recover quickly after update surges

Freshness metric

- Freshness of a view at a particular time: $f(v, t) = 0$ if v is stale at time t, or 1 otherwise
  - Being 1 day stale is no worse than being 1 second stale
- Freshness probability during observation interval $[ts, te]$: $p_f(v, [ts, te]) = (\int_{ts,\, te} f(v, t)\, dt) / (te - ts)$
- Overall freshness during $[ts, te]$: $p_f(db, [ts, te]) = \sum_{v \in db} (p_f(v, [ts, te]) \times \text{access-freq}(v))$

➢ Related work: Cho and Garcia-Molina, *SIGMOD*, 2000

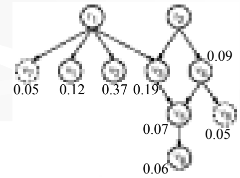
Scheduling refreshes

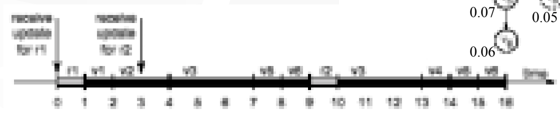


- Assume that refresh operations do not overlap (parallelism not considered)
- Updates on base tables should be applied in order
- If $U \rightarrow V$, then refresh U before V
  - V is recomputed from U
  - ➢ How about refreshing v_6 directly from r_1 and r_2?

FIFO refresh schedule

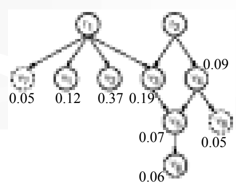
- Overall freshness: 0.51
- Refresh all affected views after the base table is updated





Optimal static refresh schedule

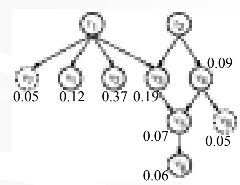
- Overall freshness: 0.68
- Merge updates
  - v3, v5, v6 are refreshed once after two base table updates



- Favor views with biggest (access frequency ⁄ refresh cost) ratio (gain in freshness per unit time)
  - v2 is refreshed before v1; v4 is refreshed before v3

QoDA scheduling algorithm

- Need to make decisions dynamically and observe the dependency among views
- Popularity weight of *v*: sum of the access frequencies of *v* itself and its descendants



- Dirty counter of *v*: number of stale ancestors of *v*
- The first view/relation to refresh is the one
  - Whose dirty counter is 0, and
  - Whose (popularity weight ⁄ refresh cost) ratio is the highest among those whose dirty counter is 0

Experiments with real workloads

- Quote.com queries and NYSE updates
- QoDA provides acceptable freshness even when it is not possible to keep up with the updates
  - By merging and/or ignoring refreshes
- FIFO lags behind more and more

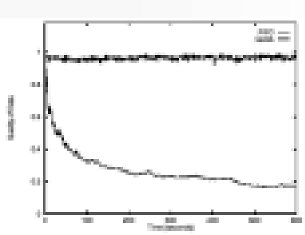
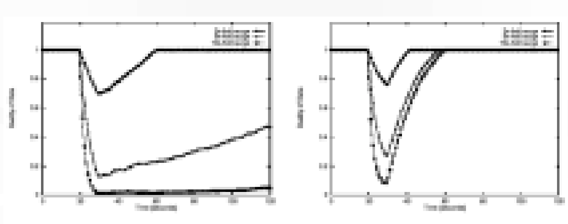


Figure 10: Real workload experiment: update processing speed is 70% of the average incoming update rate

Tolerance to update surges



- FIFO requires a long time to recover
- QoDA quickly recovers
  - By merging and/or prioritizing refreshes

Summary of refresh strategies

- Yet another example of a simple online algorithm that works extremely well in practice
- Although a more rigorous analysis would be nice

- ➢ What if there are hard constraints?
  - For example, a stock quote must be no more than 10 minutes stale
- ➢ What if age also matters in addition to freshness?
  - For example, a news page that is one day stale is still better than a news page that is one year stale