Lazy Maintenance of Materialized Views

Jingren Zhou, *Microsoft Research, USA*
Paul Larson, *Microsoft Research, USA*
Hicham G. Elmongui, *Purdue University, USA*

Introduction

- Materialized views
  - Speed up query execution time by orders of magnitude
  - But have to be kept up-to-date with base tables
- Traditional solution: *eager maintenance*
  - Maintain views as part of the base table update statement (transaction)
  - ☺ Queries (beneficiaries) get a free ride!
  - ☹ Updaters pay for view maintenance
    - Slows down updates, especially when multiple views are affected
    - Wasteful effort if views are later dropped or not used by queries

Lazy Maintenance

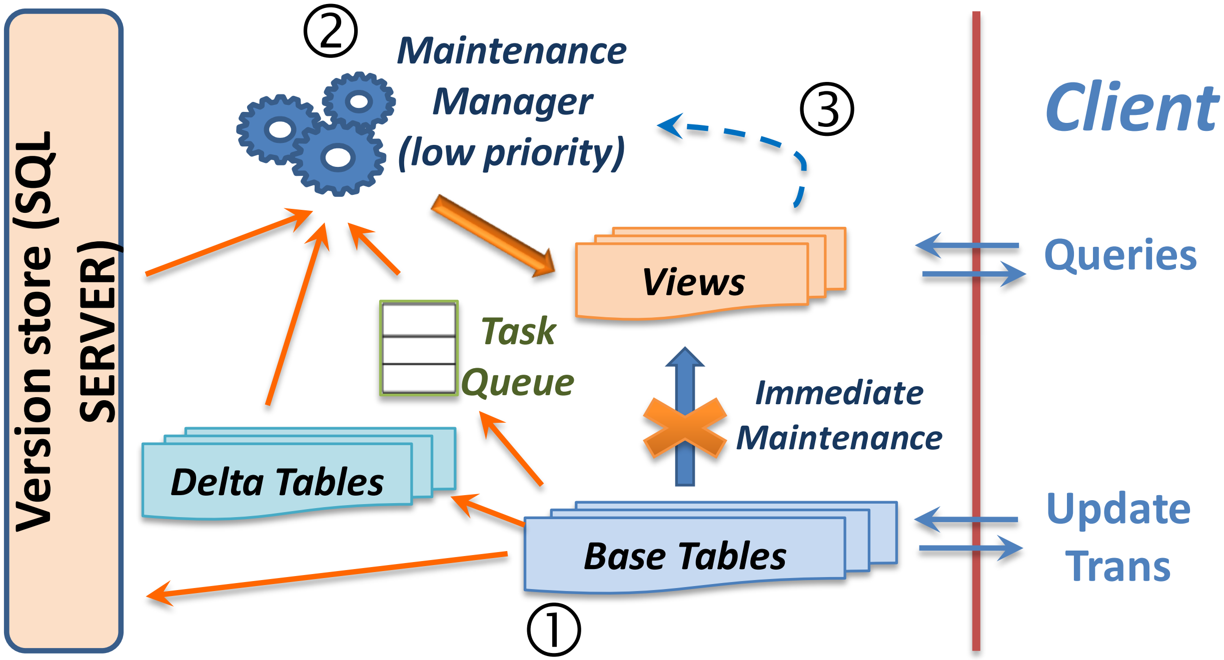
- Delay maintenance of a view until
  - The system has free cycles, or
  - The view is needed by a query
- Exploit version store and delta tables for efficiency
- Transparent to queries: views are always up-to-date
- ☺ Benefits
  - View maintenance cost can be hidden from queries
  - More efficient maintenance when combining multiple (small) updates

Agenda

- Introduction
- **Solution overview**
- Maintenance algorithms
- Condensing delta streams
- Experiments
- Conclusion

Microsoft Research

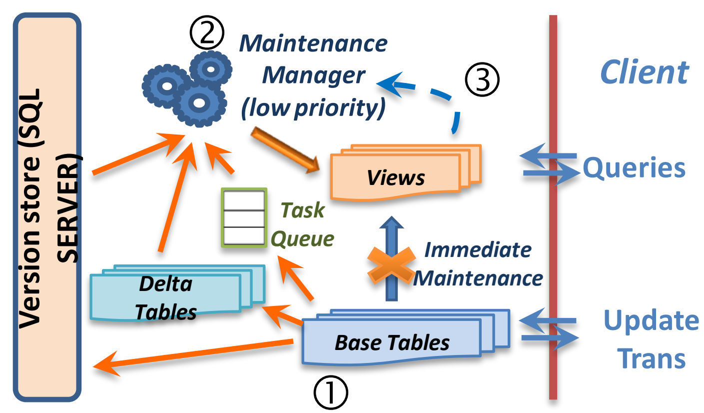
Solution Overview



- Under snapshot isolation
- Version store keeps track of all active database versions
- Delta tables store delta rows; one per base table
- Task queue store pending maintenance tasks (for recovery)
- Maintenance manager (low priority, in memory)

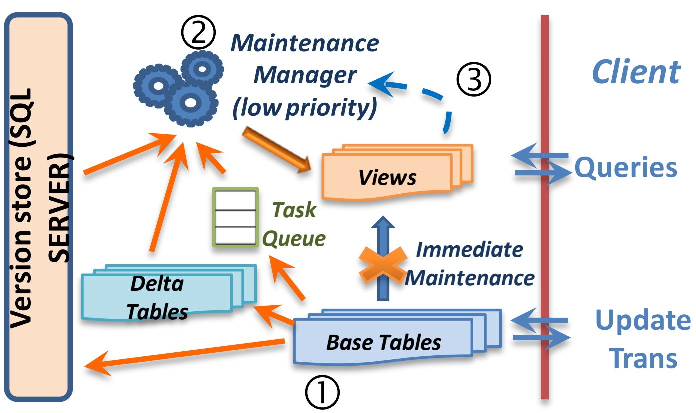
Step 1: Update Transaction

- For each update statement
  - Skip view maintenance
  - Store into the corresponding delta table
    - The delta stream
    - Action column, transaction sequence number(TXSN), statement number(STMTSN)
- When the update transaction commits
  - Construct a lazy maintenance task per affected view
  - Report tasks to the maintenance manager
  - Write tasks to the persistent task table
- *What if the transaction fails?*
  - No information is stored in the manager
  - No task is constructed



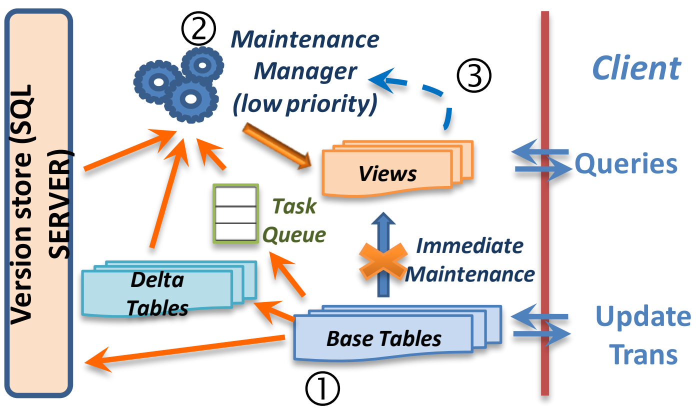
Step 2: Lazy Maintenance

- The manager wakes up every few seconds
  - Goes back to sleep if the system is busy or there are no pending maintenance tasks
  - Constructs a ***low-priority background maintenance job*** and schedules it
- Maintenance jobs
  - Jobs for the same view are always executed in the commit order of the originating transactions
  - Completion: report to the manager and delete the task(s) from the persistent task table
- Garbage collection in the manager
  - Reclaims versions that are no longer used
  - Cleans up delta tables

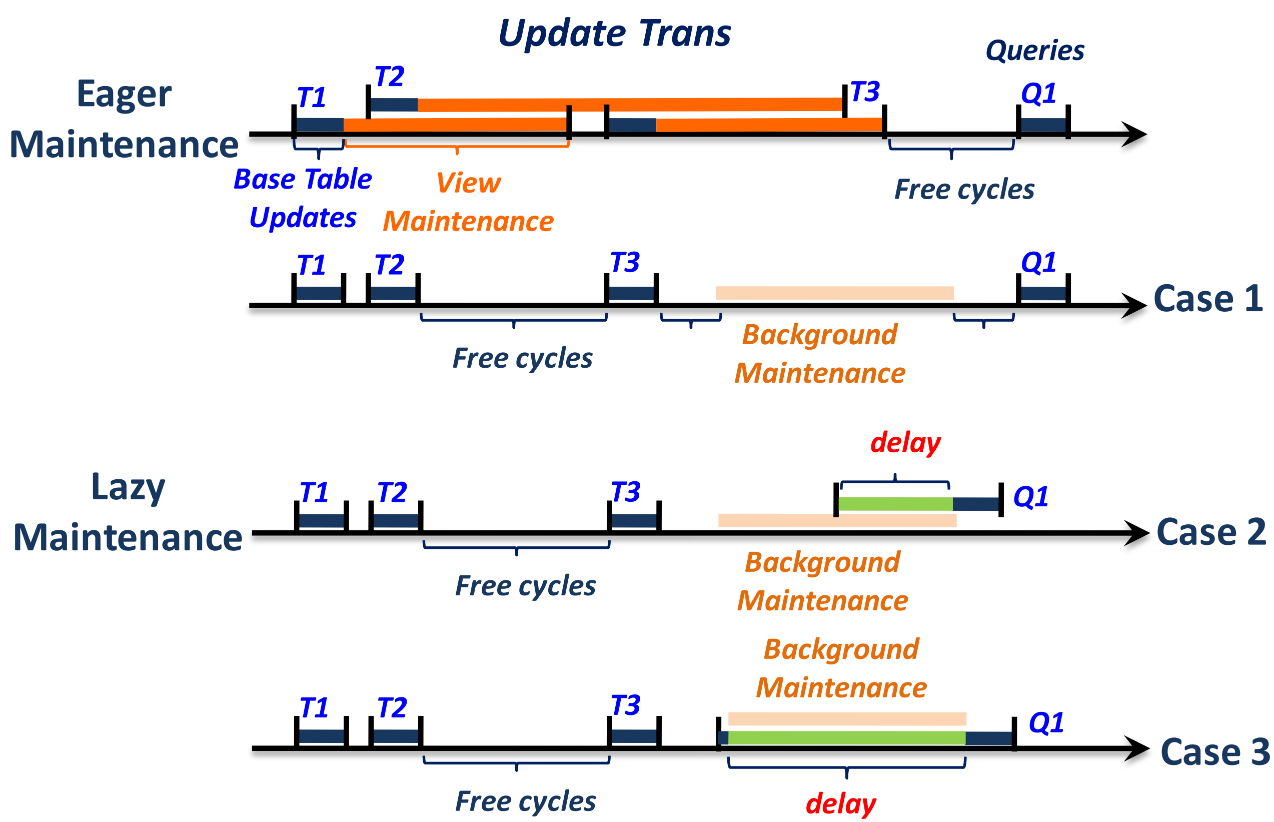


Step 3: Query Execution

- If the view is up-to-date,
  - Virtually no delay in query execution
- If the view has pending maintenance tasks ,
  - Ask the maintenance manager to schedule them immediately (***On-demand Maintenance***)
    - Maintenance jobs are executed in separate transactions and commits
    - If query aborts, committed jobs will not roll back
  - Query resumes execution when all the tasks have completed
- Complex scenario: query uses a view that is affected by earlier updates within the same transaction
  - Split maintenance into two parts
    - Bring view up-to-date as of before the trans *in a separate trans*
    - Maintain pending updates *within the current trans*



Effect on Response Time



Agenda

- Introduction
- Solution overview
- **Maintenance algorithms**
- Condensing delta streams
- Experiments
- Conclusion

Normalized Delta Streams

- *Equivalent* delta streams: produce the same final state when applied to the same initial state of the base tables
  - We can choose any equivalent delta stream to derive maintenance expressions
- Example: $V = R \bowtie S$
  - Update transaction T: initial state R_0, S_0; final state R_1, S_1
  - Delta stream ΔR^1, ΔS^1, ΔR^2, ΔS^2, …
  - New ***normalized*** delta stream
    $\Delta R = \Delta R^1 + \Delta R^2 + \ldots + \Delta R^n$, $\Delta S = \Delta S^1 + \Delta S^2 + \ldots + \Delta S^n$
    - One delta stream for each affected table
    - The ordering is important: done by sorting ΔR, ΔS in ascending order on TXSN and STMTSN
    - Equivalent to the original delta stream

Computing View Delta Streams

$V = R \bowtie S$

- Update one table *R*:
  - ΔR can be retrieved by scanning the delta table with predicate (delta.TXSN = task.TXSN and delta.STMTSN >= task.STMTSN)

    $\Delta V = \Delta R \bowtie S$
- Update tables *R* and *S* (normalized delta streams ΔR and ΔS)
  - *R*, *S* denote ***before*** version and *R'*, *S'* denotes ***after*** version ($R' = R + \Delta R$)
  - Apply streams in sequence: first ΔR, then ΔS
    - *Step 1: update R -> R'*

      $\Delta V_1 = \Delta R \bowtie \mathbf{S}$
    - *Step 2: update S -> S'*

      $\Delta V_2 = \mathbf{R'} \bowtie \Delta S$
    - $\Delta V = \Delta V_1 \bowtie \{1\} + \Delta V_2 \bowtie \{2\}$ **--- Step sequence number (SSN)**

      $= \Delta R \bowtie \mathbf{S} \bowtie \{1\} + \mathbf{R'} \bowtie \Delta S \bowtie \{2\}$
  - Update ordering: (SSN, TXSN, STMTSN)

Combining Maintenance Tasks

- Benefits of combining maintenance tasks
  - Fewer, larger jobs – less overhead!
  - Able to eliminate redundant (intermediate) updates (explained later)
- Example: *V* has a queue of *l* pending tasks T_1, … T_l (in commit order), updating the set of base table $R_1,\ldots,R_m$
  - T_e begins the earliest (has the smallest TXSN)
  - Combined into a single large trans T_0: starts at T_e.TXSN, ends at T_l.CSN, and updates $R_1 \cup \ldots \cup R_m$
  - ***before*** version: before T_e; ***after*** version: after all *l* transactions

$$
\begin{aligned}
\Delta V = \; & \Delta R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n \bowtie \{1\} + \\
& R'_1 \bowtie \Delta R_2 \bowtie R_3 \bowtie \ldots \bowtie R_n \bowtie \{2\} + \\
& \ldots + \\
& R'_1 \bowtie \ldots \bowtie R'_{m-1} \bowtie \Delta R_m \bowtie \ldots \bowtie R_n \bowtie \{m\}
\end{aligned}
$$

Schedule Maintenance Tasks

- General rule:
  - Tasks for the same view are executed strictly in the original commit order
  - Tasks for different views can be scheduled independently
- Background scheduling
  - Triggered when the system has free cycles
  - Assign priorities based on how soon view are expected to be referenced by queries
  - Combine tasks for efficiency, but too large maintenance results in a long-running maintenance transaction
    - Need to consider the size of combined delta stream, the maintenance cost, and the system workload
  - Give a higher priority for older maintenance tasks (implemented)
- On-demand scheduling
  - The maintenance job(s) inherit the same priority as query
  - Avoid maintenance if the pending updates do not affect the part of the view accessed by the query
    - For example, project the query on delta tables to check if updates are relevant, etc.

Agenda

- Introduction
- Solution overview
- Maintenance algorithms
- **Condensing delta streams**
- Experiments
- Conclusion

Applying View Delta

Key	…	Act
6	…	INS
1	…	DEL
5	…	DEL
2	…	INS
5	…	INS

View delta

→ **Sort** (Key, Act)

Key	…	Act
1	…	DEL
2	…	INS
5	…	DEL
5	…	INS
6	…	INS

Sorted view delta

→ **Collapse**

Key	…	Act
1	…	DEL
2	…	INS
5	…	**UPD**
6	…	INS

Collapsed view delta

→ ***Update*** ***V***

"Condense" Operator

Update order *(SSN, TXSN, STMTSN)*

Key	…	SSN	TXSN	STMTSN	ACT
5		2	103	1	DEL
5		2	103	2	INS
5		1	101	1	DEL
8		1	101	3	DEL
5		3	101	2	DEL
5		1	100	1	INS
5		2	101	2	INS

View delta

Sort
(Key, Upd order, Act)

Key	…	SSN	TXSN	STMTSN	ACT
5		1	100	1	INS
5		1	101	1	DEL
5		2	101	2	INS
5		2	103	1	DEL
5		2	103	2	INS
5		3	101	2	DEL
8		1	101	1	DEL

Sorted view delta

Condense

Key	…	SSN	TXSN	STMTSN	ACT
8		1	101	1	DEL

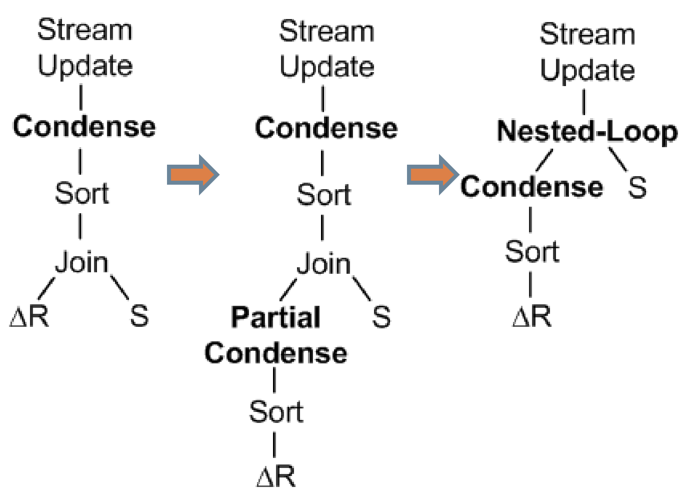
Condensed view delta

Update

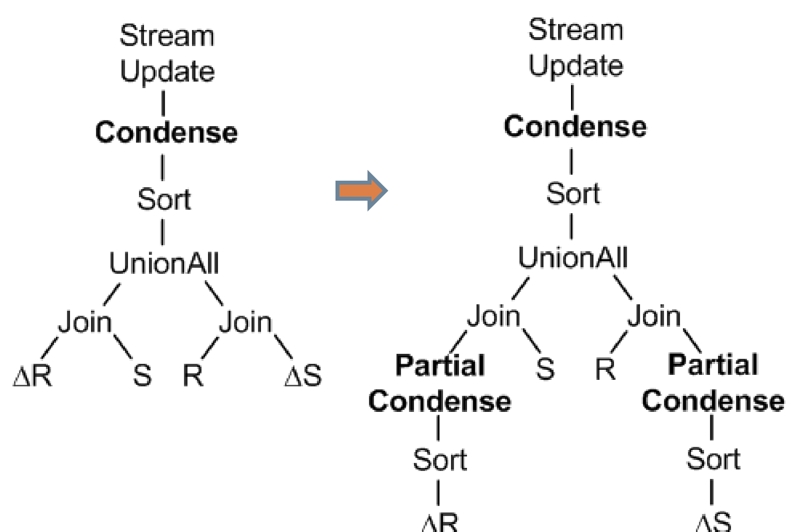
V

Partial Condense

- More generally, "Condense" is analogous to "GroupBy"; can emulate all the optimization rules
- <u>Rule of thumb</u>: *Delta rows are condensable if they are guaranteed to affect the same view row*
  - Do not care about any intermediate version of the updated table row
  - *Partial Condense:* sort ΔR on the unique keys of R + TXSN + STMTSN + Action
  - *Examples:* $V = R \bowtie S$



Updating R



Updating R + S

Agenda

- Introduction
- Solution overview
- Maintenance algorithms
- Condensing delta streams
- **Experiments**
- Conclusion

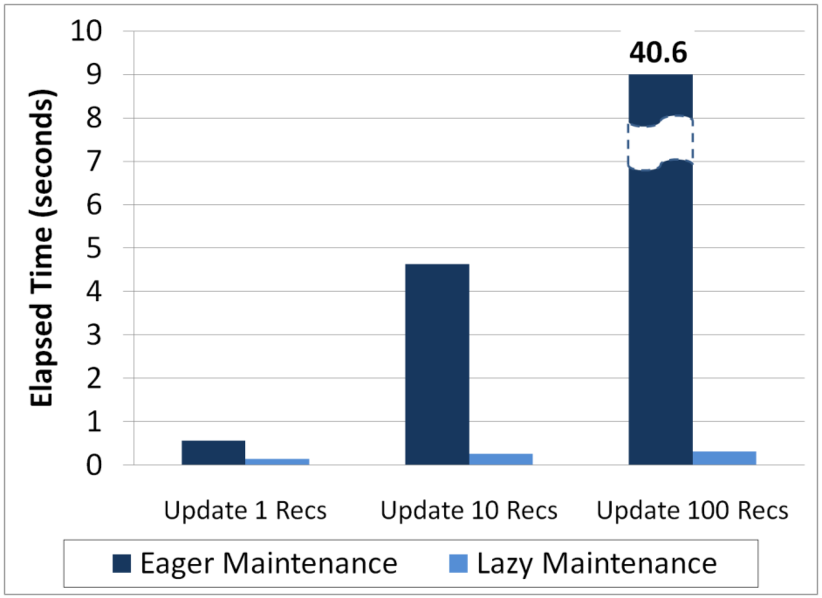
Experimental Setup

- Prototype lazy maintenance of materialized views in SQL 2005
- All queries are against TPC-H (1G) with cold buffer pool
- Materialized views

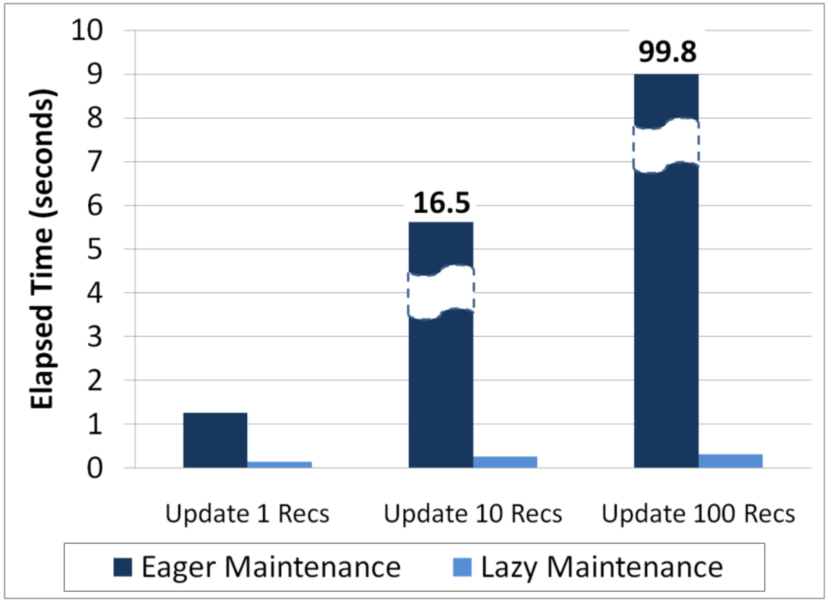
```
V1:SELECT n_name, c_mktsegment, count(*) as totalcnt
         sum(l_extendedprice) as totalprice, sum(l_quantity) as totalquan
   FROM Customer, Orders, Lineitem, Nation
   WHERE c_custkey = o_custkey AND o_orderkey = l_orderkey
     AND n_nationkey = c_nationkey
   GROUP BY n_name, c_mktsegment

V2:SELECT s_name, c_name, c_mktsegment, ps_comment, …
   FROM Customer, Orders, Lineitem, Supplier, Partsupp
   WHERE c_custkey = o_custkey AND o_orderkey = l_orderkey AND …
     AND s_nationekey <> c_nationkey
```

- Table updates on customer information, such as nation key or market segment

Update Response Time



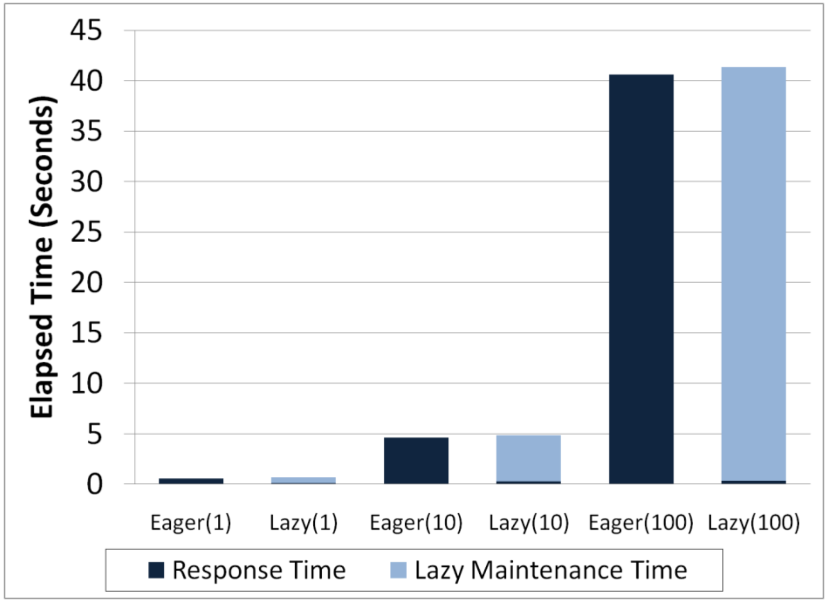
V_1

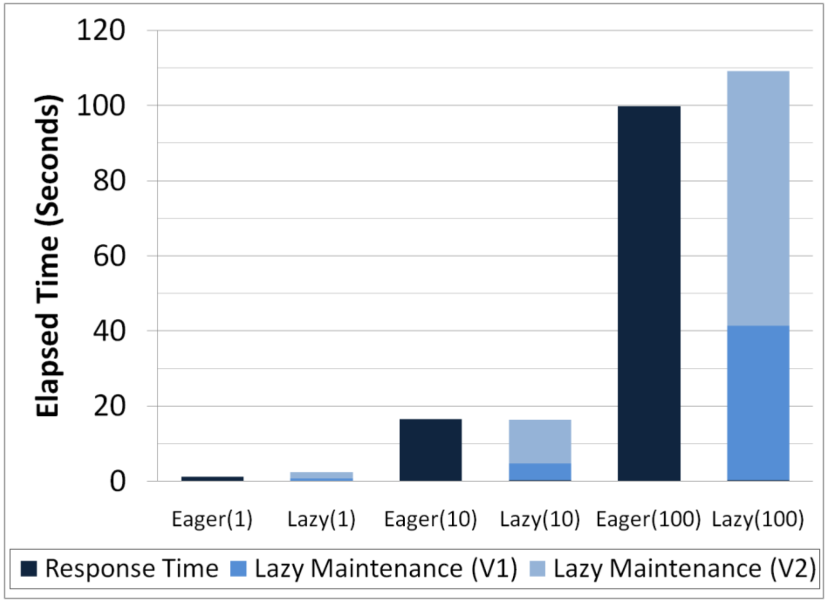


V_1+V_2

- Update 1, 10, 100 customer records using a single update statement
- Rows affected per view: 40, 400, 4000 (scattered)
- Lazy maintenance
  - Update response time is reduced to virtually nothing
  - Virtually unchanged by addition of a second view

Maintenance Cost

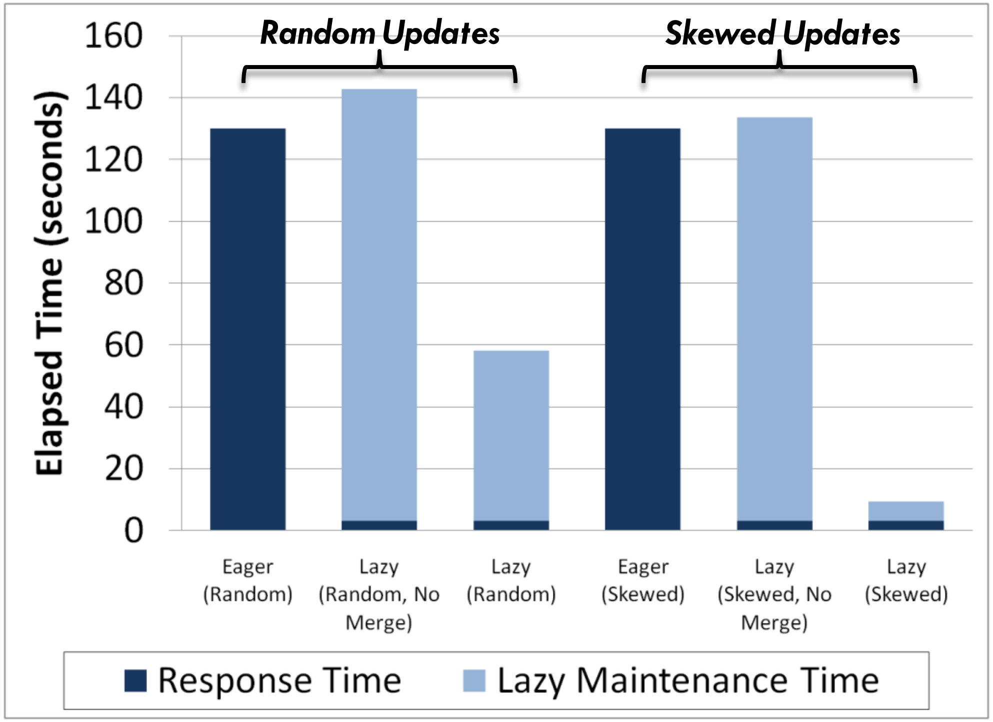




V_1 V_1+V_2

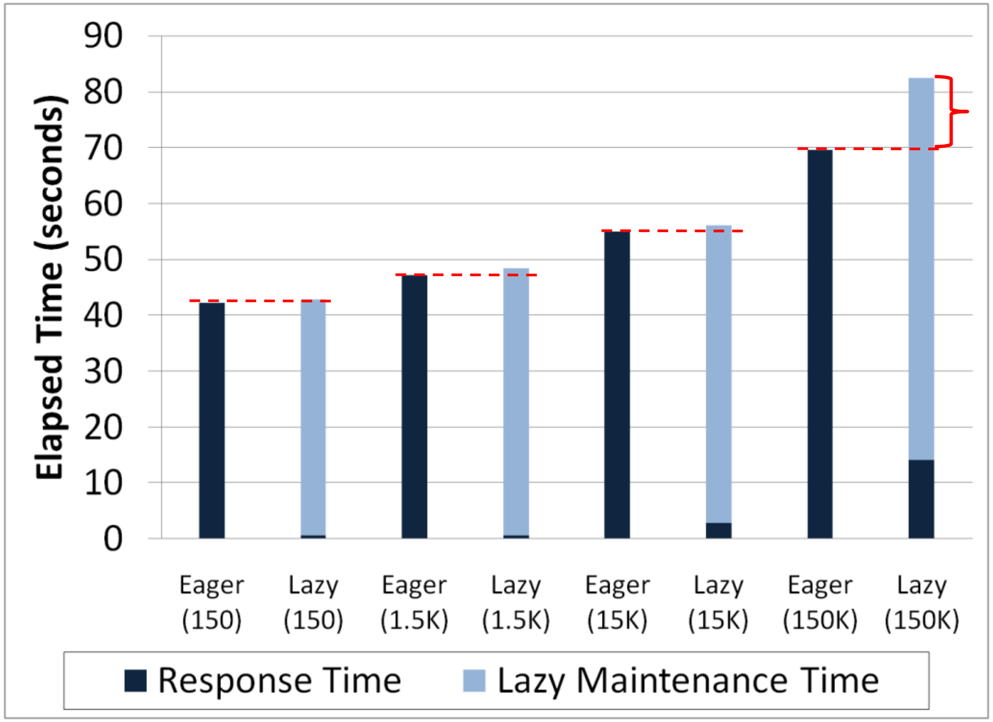
- The total amount of work = update response time + lazy maintenance time
  - The total amount of work under lazy maintenance is comparable to that of eager maintenance
    - Overhead: storing and reading delta streams and versions
  - Lazy maintenance time can be (mostly or all) hidden from applications

Multiple Updates



- 100 small updates, each updating 1-10 rows; random v.s. skewed updates
- Apply "full condense" plus "partial condense" on the delta stream
- Maintenance time is significantly reduced by combining/condensing tasks

Lazy Maintenance Overhead



- Overhead: store delta streams, etc.; maintain versions
- The overhead is more noticeable with large delta streams
- Update response time also increases with larger delta streams. But some (or all) of lazy maintenance cost may still be hidden

Related Work

- Eager maintenance has been well studied
  - Most used *update delta* paradigm
- Deferred or asynchronous view maintenance: Colby et al. [SIGMOD 1996], Salem et al. [SIGMOD 2000]
  - But have different goals
  - Differences: transparency, exploiting version store for much simpler and efficient maintenance, condensing delta streams, etc.
- Oracle supports views that are recomputed on refresh (on demand)

Conclusion

- Lazy maintenance separates maintenance from update transactions
  - Greatly improves update response time without sacrificing view usability
  - More efficient maintenance by combining and condensing updates
  - Totally transparent to applications
- The choice of maintenance strategy (eager v.s. lazy) depends on
  - The ratio of updates to queries and how soon queries follow after updates
  - The size of updates, relative to the maintenance cost
- Lazy maintenance can be applied to other auxiliary data structures, such as indexes.