View Invalidation for Dynamic Content Caching in Multitiered Architectures

K. Selçuk Candan Divyakant Agrawal Wen-Syan Li Oliver Po Wang-Pin Hsiung

C&C Research Laboratories - Silicon Valley
NEC USA, Inc.
10080 North Wolfe Road, Suite SW3-350
Cupertino, California 95014, USA
Email:{candan, agrawal,wen,oliver,whsiung}@ccrl.sj.nec.com

Abstract

In today's multitiered application architectures, clients do not access data stored in the databases directly. Instead, they use applications which in turn invoke the DBMS to generate the relevant content. Since executing application programs may require significant time and other resources, it is more advantageous to cache application results in a *result cache*. Various view materialization and update management techniques have been proposed to deal with updates to the underlying data. These techniques guarantee that the cached results are always *consistent* with the underlying data. Several applications, including e-commerce sites, on the other hand, do not require the caches be consistent all the time. Instead, they require that all out-dated pages in the caches are *invalidated* in a timely fashion. In this paper, we show that invalidation is inherently different from view maintenance. We develop algorithms that benefit from this difference in reducing the cost of update management in certain applications and we present an invalidation framework that benefits from these algorithms.

1 Introduction

Most modern application architectures are being designed as multitiered distributed systems. For example, a typical e-commerce server architecture consists of three major tiers: a database management system (DBMS) which maintains information pertaining to the service, an application server (AS) which encodes business logic pertaining to the organization, and a web server (WS) which provides the Web-based interface between the users and the e-commerce provider (Figure 1). User requests in this case invoke appropriate program scripts in the application server which in turn issue queries to the underlying DBMS to dynamically generate and construct pages.

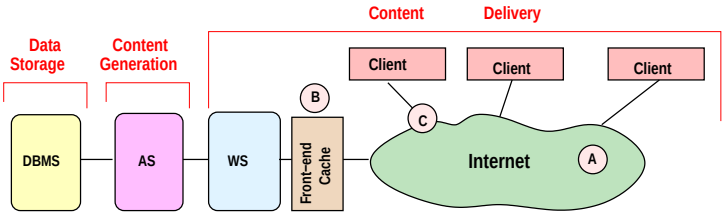

Figure 1: A typical multitiered architecture

Since executing application programs and accessing DBMSs may require significant time and other resources, it may be advantageous to cache application results in proxy, front-end, and edge caches (Figure 1(A),(B), and (C)). Unfortunately, due to technical limitations at this moment, such caches can not be effectively used. The key problem in this case is that database driven HTML content is inherently *dynamic*. The main challenge that arises in caching such content is to ensure its *freshness*. In particular, if we blindly enable dynamic content caching we run the risk of users viewing stale data specially when the corresponding data-elements in the underlying DBMS are updated. Since there is no appropriate mechanism to reflect data changes to result caches, currently, most dynamically generated HTML pages are tagged as `non-cacheable`. This means that every user request to dynamically generated HTML page must be served from the origin server. Several solutions are beginning to emerge in both research laboratories [1, 2] and companies, such as Persistent Software, Zembu, and Oracle.

There are two main problems that arise in enabling dynamic content caching: (1) Dynamically generated

Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002

HTML pages have to be mapped or associated with the data elements in the DBMS; and (2) updates to the data elements in the DBMS must invoke *invalidation* of cached HTML pages that are affected by updates.

The first problem can be solved easily if the application server logic maintains the mapping of data objects to the dynamically generated HTML pages [1]. In the absence of this explicit application logic, this mapping can be discovered (as we presented in [3]) in a loosely-coupled manner by employing a process called *sniffing*. This process identifies **(a)** a mapping between cached results and the corresponding queries used to generate those results and **(b)** a mapping between the queries and the data changes that affect these queries.

The second problem is closely related to the problem of *view maintenance* [4, 5, 6, 7] in the context of materialized views in data warehouses. Since a data warehouse consists of a large view, the main focus of the database research has been to maintain materialized views *incrementally*. Numerous algorithms have been proposed for incremental view maintenance [8, 9, 10, 11, 12]. Another related topic of investigation is the area of *query caching* [13, 14, 15]. Both solutions guarantee that the stored results are always *consistent* with the underlying data.

On the other hand, in most e-commerce applications, it is not always necessary that the users be able to access all information through the cache. If a result is not in the cache, it can always be generated on-demand using the application server and the database. What is desirable, however, is that the users do not access any information that is out-of-date through the cache. Thus, in e-commerce applications, the requirement is that out-dated pages are invalidated in a timely manner. The second problem can be addressed by developing a component referred to as the *invalidator* that monitors the database updates and sends invalidation messages to the affected HTML pages that are cached. In essence, view invalidation is determining whether a query is independent of a particular update to the underlying data. There has been a body of work [16, 17, 18, 19], which studied the dependency of SPJ and datalog queries and updates. Most of these addressed invalidation at the logical level, without referring to the underlying base-relations. [19] considers base relations, or *local data*, when checking the effects of updates on the truth values of a given set of constraints. Our work builds on the existing literature by developing efficient invalidation techniques that use local data, but impose minimal overhead on the DBMS. We show that invalidation is inherently different from view maintenance. We develop algorithms that benefit from this difference and develop an invalidation framework for enabling dynamic content caching.

2 View Invalidation

In this section, we describe the view invalidation framework in the context of dynamically generated HTML pages that are cached. Note that since we are caching dynamic content, we assume that a mapping from the dynamic content to appropriate database queries is also maintained.

2.1 Invalidation versus View Maintenance

Let the information infrastructure of an e-commerce site *Auto_buy.com* be based on a database with two relations, `Car(maker, model, price)` and `Mileage(model, EPA)`. Let one of the application scripts use the query:

```
select maker, model, price from Car where
maker = "Toyota";
```

to generate a web page,
`http://www.auto_buy.com/modelinfo?car=Toyota`, which lists the models and prices of all Toyota cars available in the inventory.

If, after this dynamically generated web page is stored in the front-end cache, a new car $(Toyota, Avalon, 25000)$ is inserted into the relation Car in the database, then the content of the cached page will be impacted and a corrective action in the front-end cache may be required. For instance, if materialized views are available for use, the system could compute the new results of this query (preferably incrementally) and then it could rerun the application to regenerate the page. If the number of cached pages is large, however, this action may prove to be too expensive to be feasible.

Alternatively, if we can quickly identify the web page that is affected by this insertion, then, we can purge it from the cache instead of regenerating it. Indeed, for most e-commerce applications, when a new product is inserted, a user request to that product can still be served by the application server by accessing the DBMS for the newly added data instead of accessing the cache. We refer to this approach as *view invalidation*. Note that we can remove a larger part of the materialized view than strictly affected by an underlying data change. For instance, in an extreme case, we can mark the entire cache invalid, if this is the only way to ensure (in real-time) that users will not access old data. Although such *over-invalidation* might reduce the hit rate of the cache, it may help the system to deal with updates in real-time.

> *When compared to view management, invalidation provides two advantages. Given an update*
> - *we do not need to compute all its consequences, and*
> - over-invalidation *does not compromise correctness.*

On the other hand, we have to make sure that every affected cached result must be invalidated; i.e., such *under-invalidations* can compromise correctness and must be avoided.

2.2 Collecting Queries and Updates

Since we are assuming a relational model for the underlying DBMS, the query definitions will be available

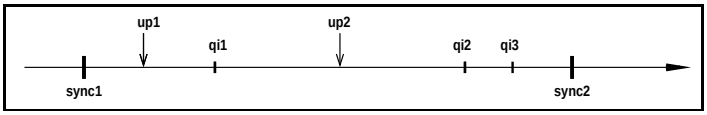

Figure 2: Queries and updates

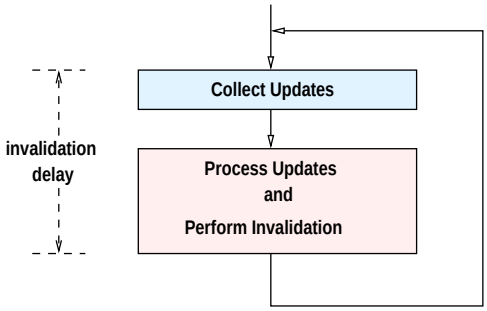

Figure 3: Invalidation process and invalidation overhead

as SQL statements. For simplicity and clarity, we will instead assume that these definitions are in the form of SPJ expressions, an assumption that is widely made in the view maintenance literature [5, 20, 11]. Figure 2 shows a timeline and a sequence of events that are registered by a DBMS:

- The qis on the timeline show the query instances processed by the database.
- The *up* events show the data updates.

Without loss of generality, we assume that the DBMS uses appropriate synchronization mechanisms to ensure atomicity of queries and updates. Besides the query and update events, the figure also shows a set of synchronization, *sync*, events. These events mark the time instances when the list of updates are passed to the invalidation module (in the order of arrival) for processing. For example, if the invalidator is working outside of the DBMS, such information can be extracted from the update logs of the database[1].

Note that if there is a *sync* event for each update event, then the invalidation process will be more *real-time*, however the invalidator may not benefit from correlations between updates that are occurring temporally close to each other. On the other hand, if each *sync* event covers a set of update events, then these updates will be processed in batches, potentially benefiting from commonalities in updates, but introducing a temporal delay in the invalidation process.

2.3 Invalidation and Polling Queries

View invalidation is performed iteratively (Figure 3); at the beginning of each iteration, current updates are extracted and then these updates are processed to generate appropriate invalidation messages. If the invalidation process is not synchronized with the DBMS, the latency during the invalidation process may result in an interference during invalidation. Our goal is to identify such interferences and develop schemes to avoid them.

Let us revisit the e-commerce application example that was presented earlier. Assume there is an application script which issues a query, $Query$,

```
select maker, model, Car.price, Mileage.EPA
from Car, Mileage
where Car.maker = "Toyota" and
      Car.model = Mileage.model;
```

to generate the page `http://www.auto_buy.com/mileageinfo.cgi?car=Toyota`, which provides mileage information about the Toyotas.

If a new tuple, say ("Mitsubishi", "Galant", 23000), is inserted into the relation Car after this page is cached, we may be able to check whether this tuple **does not** satisfy the condition in $Query$ without any additional information. That is, if the *maker* attribute of the new tuple is different from *"Toyota"*, then we can conclude that the new tuple does not affect any of the cached pages. However, if the new tuple, say ("Toyota", "Avalon", 25000), satisfies the condition then we do not know *whether or not the result is impacted* until we check the rest of the condition, which includes the table $Mileage$. That is, to see if the new tuple **does** satisfy the condition in $Query$ we need to check whether or not the condition $Car.model = Mileage.model$ can be satisfied. To check this condition, we need to issue the following polling query, $PollQuery$, to the DBMS:

```
select Mileage.model, Mileage.EPA
from Mileage
where "Avalon" = Mileage.model;
```

If the result set of $PollQuery$ is non-empty, we know that the newly inserted tuple, ("Toyota", "Avalon", 25000), affected $Query$ and consequently the corresponding page must be invalidated. An analogous scenario occurs for deleted tuples.

Note that there is a trade-off between the amount of polling and processing required and the quality of the invalidation process. If we do not have enough time to process the required polling query, we can choose to be cautious and, in order to avoid a possible under-invalidation, invalidate $Query$ without knowledge about the contents of the $Mileage$ table. In general, it is possible to send detailed polling queries, hence spending more time, to identify which web pages in the cache **are not affected** by a given update. Therefore, it is possible to use this trade-off between the amount of polling and the invalidation quality to schedule polling queries within the real-time constraints of an e-commerce site.

2.4 Summary

Based on the discussions presented in this section, we see that in many multitiered application systems, (quick-and-dirty) view invalidation is a more desirable option compared to using (costly) view maintenance.

[1] For example. ORACLE 8i provides the *log miner* interface to extract updates incrementally from the log.

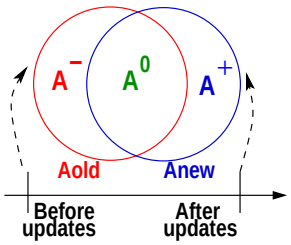

Figure 4: The old and new version of the relation A

3 Invalidation of Queries with Two Relations

In this section, we develop techniques for invalidating cached queries (equivalently, views). For simplicity of the presentation, we start with view or query definitions that are restricted to two relations. We will generalize these techniques for multiple relations in Section 4. In order to maintain the separation of the invalidation module and the DBMS, we assume that the invalidator module has access to the DBMS update logs. Periodically, the invalidator can scan the log (from the point where it read the log last time) to extract all the updates.

3.1 Δ: Changes in the View

Consider a query definition, $q = A \bowtie B$, that involves relations A and B used for generating a web page that is maintained by the invalidator. In this paper we will assume that updates are modeled as *inserts* and *deletes* of tuples in relations A and B. Figure 4 shows the old and new versions of A denoted as A_{old} and A_{new}, respectively. A^+ denotes the set of *inserted* tuples in relations A and A^- denotes the set of *deleted* tuples. In addition, the part of the relations that did not change is denoted as A^0. Similarly, for the relation B. We can now rewrite the sets of tuples that are in the old and new results, $A_{old} \bowtie B_{old}$ and $A_{new} \bowtie B_{new}$, as follows:

$$A_{old} \bowtie B_{old} = (A^0 \cup A^-) \bowtie (B^0 \cup B^-)$$
$$= (A^0 \bowtie B^0) \cup (A^0 \bowtie B^-) \cup (A^- \bowtie B^0) \cup (A^- \bowtie B^-)$$

$$A_{new} \bowtie B_{new} = (A^0 \cup A^+) \bowtie (B^0 \cup B^+)$$
$$= (A^0 \bowtie B^0) \cup (A^0 \bowtie B^+) \cup (A^+ \bowtie B^0) \cup (A^+ \bowtie B^+).$$

Therefore, the set of tuples deleted from or inserted to the join can be enumerated as

$$\Delta = \underbrace{(A^0 \bowtie B^-)}_{term_1} \cup \underbrace{(A^- \bowtie B^0)}_{term_2} \cup \underbrace{(A^- \bowtie B^-)}_{term_3} \cup$$
$$\underbrace{(A^0 \bowtie B^+)}_{term_4} \cup \underbrace{(A^+ \bowtie B^0)}_{term_5} \cup \underbrace{(A^+ \bowtie B^+)}_{term_6}$$

3.2 Advantages of Invalidation over View Maintenance

In view maintenance such changes must be partitioned into two sets: *deleted* set of tuples and *inserted* set of tuples, and these sets have to be treated separately. In contrast, in the context of view invalidation, a query, q, is affected by the updates if Δ (inserted or deleted) is non-empty. Furthermore, in order to decide whether to invalidate the results q, we **do not** need to evaluate the entire Δ, but, we need to determine if it contains at least one tuple. If there is a tuple in Δ, we can stop right away as evaluating additional tuples in Δ is not necessary. For this purpose, we can use the *top-k* retrieval algorithms proposed in [21] and others. Therefore, at this point, we can (informally) state that

> View invalidation is *inherently* cheaper than view maintenance.

Intuitively, this is because identifying that a query is affected by a set of updates is inherently cheaper than finding the exact consequences of such updates.

3.3 Challenges in Computing Δ

Note that evaluating the terms that constitute Δ requires not only the knowledge about the changes (A^+, B^+, A^- and B^-), but also the parts of the relations that did not change (A^0 and B^0). The invalidation module can acquire knowledge about the data changes by examining the update log. However, it has to access the database in order to obtain the database state A^0 and B^0. As we have discussed earlier, this requires the evaluation of *polling queries* at the DBMS.

A major challenge in creating polling queries is that A^0 and B^0 are not explicitly maintained by the DBMS. Therefore, unless appropriate measures are taken, A^0 and B^0 will not be available for polling queries: by the time updates are collected, relations A and B have already been modified by new updates, and A^0 and B^0 are not available anymore. Therefore, computation of Δ requires intelligent update collection and polling scheduling mechanisms. We see that there are three approaches to this challenge, each with its own advantages and disadvantages:

- *Snapshot-based approach*, where a copy (or a *snapshot*) of the database is maintained for invalidation purposes.
- *Synchronous approach*, where only a single copy of the database is maintained, but this copy is locked during invalidation processing, and
- *Asynchronous approach*, where only a single copy of the database is maintained and no locking is used.

3.4 Case I: Snapshot-based Approach

This approach assumes that database snapshots both before and after the updates are available to the invalidator. This can be achieved either by delaying the actual updates or by maintaining external copies of the relevant portions of the original database. The first option incurs additional load on the system, as it limits when updates are applied to the relations. While the second option does not have this overhead, it requires

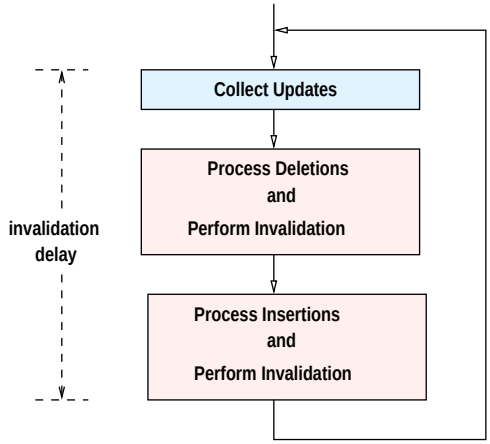

Figure 5: Invalidation process with the snapshot-based approach

appropriate data structures and query processing capabilities embedded in the invalidator.

As formulated above, evaluating Δ requires that we have access to the unchanged portion of the relations, i.e., A^0 and B^0. The two snapshots, on the other hand, actually consists of relations

- old snapshot: $A_{old} = A^0 \cup A^-$, $B_{old} = B^0 \cup B^-$,
- new snapshot:$A_{new} = A^0 \cup A^+$, $B_{new} = B^0 \cup B^+$.

We can rewrite Δ as:

$$\begin{aligned}\Delta =&(A^+ \bowtie B^0) \cup (A^+ \bowtie B^+) \cup (A^0 \bowtie B^+) \cup (A^+ \bowtie B^+)^* \cup \\ &(A^- \bowtie B^0) \cup (A^- \bowtie B^-) \cup (A^0 \bowtie B^-) \cup (A^- \bowtie B^-)^*,\end{aligned}$$

by repeating the terms marked by $*$. Such a repetition does not give rise to any inconsistency in the view invalidation context. We can further rewrite this equation as

$$\begin{aligned}\Delta =&\{(A^+ \bowtie B_{new}) \cup (A_{new} \bowtie B^+)\} \bigcup \\ &\{(A^- \bowtie B_{old}) \cup (A_{old} \bowtie B^-)\}\end{aligned}$$

Given the above formulation of Δ we can implement the snapshot method as follows. The invalidator maintains snapshots of A_{old} and B_{old} which resulted from the prior invalidation cycle. At the current invalidation cycle, the invalidator extracts the sets of inserted and deleted tuples. Then, (Figure 5):

- Computes if $(A^- \bowtie B_{old}) \cup (A_{old} \bowtie B^-)$ is non-empty;
- Sets $A_{new} = A_{old} \cup A^+ \setminus A^-$ and $B_{new} = B_{old} \cup B^+ \setminus B^-$; and
- Computes if $(A^+ \bowtie B_{new}) \cup (A_{new} \bowtie B^+)$ is non-empty.
- Discards A_{old} and sets A_{new} to A_{old}. Similarly for the relation B.

The above approach requires processing four (two if we assume that unions can be expressed as a part of a single query) queries in order to invalidate one query. However, when the sizes of the updates are small, it is likely that processing these four queries will be cheaper than regenerating the results of the original query. Furthermore, we can stop the process as soon as we identify one single tuple in the result, without really waiting for the results of all four queries. This process, however, may still be too expensive when there are many queries to be invalidated, as it is the case in e-commerce sites where there are many pages in the cache that correspond to queries executed with different parameters. In Section 5 we will show that it is possible to efficiently extend this approach to batch processing of similar query instances.

3.5 Case II: Synchronous Approach

An alternative option, which introduces less intervention on the original database and which does not create an external copy of the tables, is to let the original relations to be updated freely; but to lock these relations right before the invalidation process starts. The main consequence of this change is that A_{old} and B_{old} are not available for polling queries anymore. The only available source for these relations are A_{new} and B_{new}. Therefore, while computing

$$\begin{aligned}\Delta =&(A^0 \bowtie B^-) \cup (A^- \bowtie B^0) \cup (A^- \bowtie B^-) \cup \\ &(A^0 \bowtie B^+) \cup (A^+ \bowtie B^0) \cup (A^+ \bowtie B^+)\end{aligned}$$

we need to use $A_{new} = A^0 \cup A^+$ and $B_{new} = B^0 \cup B^+$ instead of A^0 and B^0. If, we rewrite Δ using the available relations, we get

$$\begin{aligned}\Delta' =&(A_{new} \bowtie B^-) \cup (A^- \bowtie B_{new}) \cup (A^- \bowtie B^-) \cup \\ &(A_{new} \bowtie B^+) \cup (A^+ \bowtie B_{new}) \cup (A^+ \bowtie B^+),\end{aligned}$$

which is equal to

$$\Delta' = \Delta \cup (A^- \bowtie B^+) \cup (A^+ \bowtie B^-).$$

In other words, since it maintains only one copy of the database, the synchronous approach introduces an over-invalidation term, $O = (A^- \bowtie B^+) \cup (A^+ \bowtie B^-)$. If O is not empty, the query may be invalidated unnecessarily. Over-invalidation may jeopardize performance but not correctness.

3.6 Case III: Asynchronous Approach

The invalidation technique presented in the previous section assumes that relations A and B are locked during the invalidation process; hence A_{new} and B_{new} are available for polling queries. This approach induces additional overhead on the original database due to reduced availability of the database for updates.

An alternative option would be to let the original database be updated freely during the invalidation process. Figure 6 shows the old and new versions of the relation A. In this figure A_{old} represents the old state of relation A, A_{new} represent its new state at the time when updates are collected, and A' represents the state of A when polling queries are forwarded to the database. In summary:

$$\begin{aligned}&A_{old} = A_a \cup A_b \cup A_c \cup A_d, \quad A_{new} = A_b \cup A_c \cup A_e \cup A_f, \\ &A^- = A_a \cup A_d, \qquad\qquad\qquad A^+ = A_e \cup A_f.\end{aligned}$$

Furthermore, since the relation A may freely change during the invalidation process, we also have

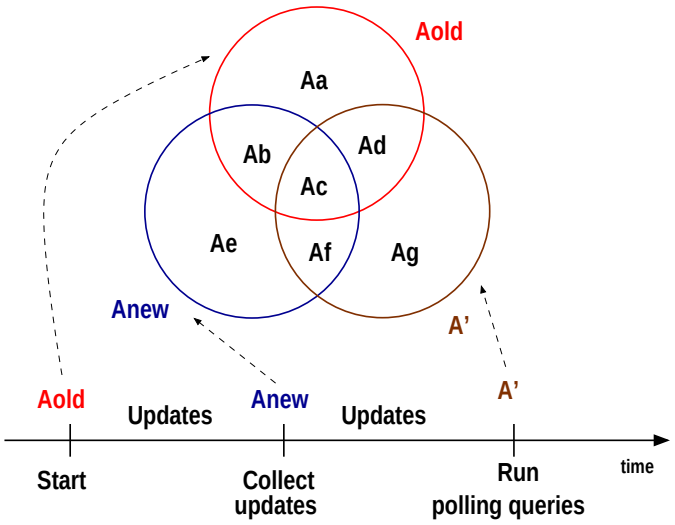

Figure 6: The old, new, and available versions of the relations A, assuming that no locking mechanism is used.

$A' = A_c \cup A_d \cup A_f \cup A_g, \quad \delta A'^- = A_b \cup A_e, \quad \delta A'^+ = A_d \cup A_g.$

That is, during the invalidation cycle the state of relation A changed from A_{new} to A' during which tuples corresponding to regions A_b and A_e (denoted by $\delta A'^-$) were deleted and tuples corresponding to regions A_d and A_g (denoted by $\delta A'^+$) were inserted, asynchronously. The corresponding terms for B are similar. Since at the polling query generation time we have access to only A^+, A^-, A', B^+, B^-, and B', we have to compute Δ using these terms:

$$\begin{aligned}\Delta' =&(A' \bowtie B^-) \cup (A^- \bowtie B') \cup (A^- \bowtie B^-) \cup \\ &(A' \bowtie B^+) \cup (A^+ \bowtie B') \cup (A^+ \bowtie B^+),\end{aligned}$$

which can also be rewritten as

$$\begin{aligned}\Delta' =&\Delta - \\ &\big[(A_b \bowtie B^-) \cup (A^- \bowtie B_b) \cup (A_b \bowtie B^+) \cup (A^+ \bowtie B_b)\big] \bigcup \\ &[((A_d \cup A_f \cup A_g) \bowtie B^-) \cup (A^- \bowtie (B_d \cup B_f \cup B_g)) \cup \\ &((A_d \cup A_f \cup A_g) \bowtie B^+) \cup (A^+ \bowtie (B_d \cup B_f \cup B_g))].\end{aligned}$$

Hence, Δ' both contains additional terms and misses some of the terms in the original Δ. The additional terms cause over-invalidation, whereas the missing terms may lead into under-invalidation. In particular, the over-invalidation is caused by the terms

$$\begin{aligned}\Delta' - \Delta =&((A_d \cup A_f \cup A_g) \bowtie (B^- \cup B^+)) \bigcup \\ &((A^- \cup A^+) \bowtie (B_d \cup B_f \cup B_g)).\end{aligned}$$

This, however, may be acceptable since over-invalidation jeopardizes performance but does not compromise correctness. In order to prevent the under-invalidation, however, we need to compute the missing terms in Δ' and adjust the invalidation decision accordingly. The missing terms in Δ are

$$\Delta - \Delta' = (A_b \bowtie (B^- \cup B^+)) \cup ((A^- \cup A^+) \bowtie B_b).$$

Unfortunately, at the invalidation time, we do not know what A_b or B_b are. Therefore, we can not calculate this term and recover from under-invalidation

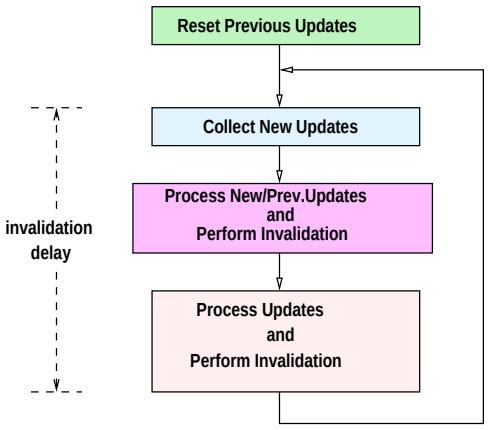

Figure 7: Invalidation process when the relations are free to be updated during invalidation

using the information available at this synchronization point. Note, however, that A_b is a subset of the tuples that are being deleted from the relation A during the invalidation process. Similarly, B_b is a subset of the tuples that are being deleted from B. These tuples will be available to the invalidator at the next synchronization point, say in the form of update logs A_2^- and B_2^-. Since $A_b \subseteq A_2^-$ and $B_b \subseteq B_2^-$, therefore,

$$\Delta - \Delta' \subseteq (A_2^- \bowtie (B^- \cup B^+)) \cup ((A^- \cup A^+) \bowtie B_2^-).$$

Hence, we can recover from under-invalidation by computing

$$(A_2^- \bowtie (B^- \cup B^+)) \cup ((A^- \cup A^+) \bowtie B_2^-)$$

at the next synchronization point and readjusting the invalidation decision accordingly. Although calculating these terms would guarantee that there is no under-invalidation, it may contribute toward over-invalidations of cached results.

Figure 7 shows the overall structure of the invalidation process. Invalidation is performed within an infinite loop; at the beginning of each iteration, recent updates are collected and these updates are processed together with the updates in the previous iteration to prevent any under-invalidation. Then, the new set of updates are processed to generate invalidation messages corresponding to these updates.

4 Invalidation of Queries with More than Two Relations

In the previous section, we introduced techniques required for invalidating queries with two relations. In this section, we generalize this to queries with more than two relations. Given a query $q = R_1 \bowtie R_2 \bowtie \ldots\ldots \bowtie R_n$, we can generalize the definition of Δ as

$$\Delta = [[\bowtie_{i=1}^{n} (R_i^- \cup R_i^0)] \cup [\bowtie_{i=1}^{n} (R_i^+ \cup R_i^0)]] - \bowtie_{i=1}^{n} R_i^0$$

which has $2^{(n+1)} - 2$ non-overlapping terms. In this section, we will discuss techniques to evaluate Δ efficiently.

4.1 Case I: Snapshot-based Approach

As it was the case in queries with two relations, Δ is described in terms of R_i^0s which correspond to the unchanged portions of the input relations. If we assume that we also have access to old ($R_{old,i}$) as well as new ($R_{new,i}$) snapshots of the relation R_i, we can rewrite Δ using $2 \times n$ terms[2], much less than $2^{n+1} - 2$ queries required by the naive formulation of Δ. Note that if R_i^+ and R_i^- are small, then computing Δ will be much cheaper than re-evaluating q. Furthermore, as we discussed earlier, computation of Δ can be terminated as soon as Δ becomes non-empty.

4.2 Case II: Synchronous Approach

In this case, as we have seen earlier in Section 3, we can not rewrite Δ without introducing over-invalidation. Since, $R_{old,i}$s are not available, while computing Δ, we need to use $R_{new,i}$ whenever we need to access R_i^0. This formulation results in an over-invalidation term

$$O_{benign} = \bowtie_i^n (R_i^- \cup R_i^+) - (\bowtie_i^n R_i^- \cup \bowtie_i^n R_i^+),$$

which can be recovered by additional processing during the invalidation time, as well as other terms, $O_{malicious}$, that can not be recovered as they contain references to relations, R_i^0, which are not available. Note that, as we have seen in Section 3, when the number of relations is two, $O_{malicious} = \emptyset$.

4.3 Case III: Asynchronous Approach

In this case, neither $R_{old,i}$ nor $R_{new,i}$ are available for invalidation. Instead, polling queries must use R_i's, which may contain new tuples and miss some of the old tuples. Note that Δ' computed using using R_i' only, is not exactly equal to Δ; it both introduces new terms (over-invalidation) and misses some of the terms (under-invalidation) in Δ. When there are more than two relations in the query, however, since some terms are completely lost, it is not possible to recover from under-invalidation using additional post-processing.

4.4 Summary

We can summarize the results of the last two sections as follows:

- Queries with only two relations can be invalidated without causing any under-invalidation. If we are not maintaining locks on the tables, however, it is possible to incur some over-invalidation.
- Queries with more than two relations can be invalidated, by maintaining appropriate locks during the invalidation process, without under-invalidation. If we are not maintaining locks on the tables, the process may cause under-invalidation.

Therefore, we do not suggest to perform invalidation on queries with more than two relations if maintaining locks is not feasible.

[2]Details omitted for space considerations.

5 Invalidation of a Set of Related Queries

An e-commerce site (our motivating application) receives and caches thousands of queries. When the number of queries to be maintained by the invalidator is large, however, the amount of processing that is required in order to generate the invalidation messages may be very large. Therefore, when the number of cached queries is large, instead of treating each query instance individually, it may be more efficient to find the related instances and process them as a group. In particular, if we are given a set, $\mathcal{Q}$, of query instances that are of the same type, QT, then we can create a new table, $T_{QT}(qid, V_1, \ldots, V_o)$, that contains all the stored query instances of this type.

Example 5.1 Given a query type $QT(V1, V2)$

```
SELECT * FROM R1,R2
WHERE R1.A = $V1 and R1.B = R2.B and R2.C = $V2;
```

and the following three query instances,

```
t1:  SELECT * FROM R1,R2
WHERE R1.A = 100 and R1.B = R2.B and R2.C = 200;
t2:  SELECT * FROM R1,R2
WHERE R1.A = 150 and R1.B = R2.B and R2.C = 80;
t3:  SELECT * FROM R1,R2
WHERE R1.A = 80 and R1.B = R2.B and R2.C = 60;
```

we can collect all these query instances in a query instance table

T_{QT}		
queryID	V1	V2
$qid1 = \langle$t1$, QT\rangle$	100	200
$qid2 = \langle$t2$, QT\rangle$	150	80
$qid3 = \langle$t3$, QT\rangle$	80	60

In this section, we introduce techniques for batch invalidation of cached queries. Therefore, we can restate the invalidation task as follows. Given

- a database $\mathcal{D}$ which contains a set of relations $\mathcal{R} = \{R_1, \ldots, R_n\}$,
- a set, $\mathcal{U}(s)$, of updates (tuples deleted and inserted during the s^{th} synchronization period) on these relations, $\{R_1^+, \ldots, R_n^+, R_1^-, \ldots, R_n^-\}$,
- a select-project-join query type $\mathcal{QT}(V_1, \ldots, V_o)$, and
- a set, $\mathcal{Q}$, of query instances of type $\mathcal{QT}$,

we want to find the set, $\mathcal{Q}^\Delta$ of query instances that *may have been affected* by the updates. Once we identify them, we use the query-instance/application-result map to invalidate those results in the application result cache that depends on these query instances.

5.1 Consolidated Invalidation of a Set of Query Instances

In order to reduce the overhead of the invalidation process, we can benefit from the similarities between the query instances maintained by the invalidation framework. In particular, if we are given $T(qid, V_1, \ldots, V_o)$,

that contains all the stored query instances of a query type, then we can find the set, $\mathcal{Q}^{\Delta}$, of queries that are affected by the updates as

$$\mathcal{Q}^{\Delta} = \pi_{qid}\sigma_{\Theta_T}(\Delta \bowtie T),$$

where Θ_T is a condition where any reference in Θ to a parameter V_j is replaced by a reference to $T.V_j$ and Δ is the term calculated in the previous sections.

Example 5.2 Let us reconsider the query type, QT from Example 5.1and two tables, $R1$ and $R2$:

T_{QT}

queryID	V1	V2
$qid1$	100	200
$qid2$	150	80
$qid3$	80	60

R1

A	B
100	20
300	80
500	100

R2

B	C
10	50
20	200
80	500

then, we can see that the cached result for $qid1$ is $\{\langle 100, 20, 200\rangle\}$. The result sets for both $qid2$ and $qid3$, on the other hand, are empty.

queryID	Results in the cache
$qid1$	$\{\langle 100, 20, 200\rangle\}$
$qid2$	$\emptyset$
$qid3$	$\emptyset$

Next, let us assume that the first two rows of R_1 are deleted due to an update (i.e., $R_1^- = \{\langle 100, 20\rangle, \langle 300, 80\rangle\}$ and $R_1^+ = R_2^- = R_2^+ = \emptyset$). Assuming that we are using the snapshot based approach, Δ can be calculated as $R_1^+ \bowtie R2_{new}$, which is equal to

$\Delta = R1^+ \bowtie R2_{new}$

A	B	C
100	20	200
300	80	500

Therefore, the list of query instances to be invalidated (only qid1 in this case) can be found by projecting the query instance IDs from the following table:

$\sigma_{A=\$V1 \wedge C=\$V2}(\Delta \bowtie T)$

A	B	C	queryID	V1	V2
100	20	200	qid1	100	200

5.2 Cost of Consolidated Invalidation versus Individual Invalidation of Queries

As it can be seen above, batch or consolidated processing of query instances transforms the query processing from an existence (i.e., top-1) query to a join query. Performing $|T|$ many top-1 queries would require $O(|T| \times t_{\Delta})$ time, where t_{Δ} is the average top-1 query execution time for the evaluation of Δ. Depending on the availability of indexes, sorted tables, and/or pipelining, top-1 retrieval of Δ can be very fast, $O(1)$, or it can require as much time as needed to completely evaluate Δ [13]. In addition, there will be resource and communication overheads associated with sending $|T|$ different queries to the database. The consolidated processing would, on the other hand, use one single polling query per query type. With appropriate data structures and indexes this query can be processed very fast. For example, assuming the availability of hashes, the consolidated query will take $O(|T| + \rho)$ time, where ρ is the total size of the relations in Δ. Furthermore, since there is only one polling query, the resource overhead will be minimal. Therefore, we can conclude that it is more advantageous to perform batch invalidation of query instances.

5.3 Effects of Over- and Under-invalidation

The number of query instances that are over-invalidated (under-invalidated) is a function of the current size of the query instance table, the size of the over-invalidation term, O, (under-invalidation term, U), and the selectivities of the join and selection operations.

Example 5.3 Let us consider the following three (two database and one query) tables with the query template `select product, price, discount from Products, Discounts where Product.Price = Discounts.Price and Discounts.Discount=$V1`.

Products

Product	Price
TV	\$500
Radio	\$90

Discounts

Price	Discount
\$500	\$50
\$1000	\$50

T

qid	Discount
qi1	\$50
qi2	\$100

Note that the first query instance in the query instance table has one tuple {⟨TV,\$500,\$50⟩} in the result, whereas the second query instance has no tuples (no products with \$100 discount).

queryID	Results in the cache
$qid1$	{⟨TV,\$500,\$50⟩}
$qid2$	$\emptyset$

Now, assume that the tuple ⟨PC,\$1000⟩ is inserted into *Products* and tuple ⟨\$1000,\$50⟩ is deleted from *Discounts*. The resulting tables are as follows:

Products

Product	Price
TV	500
Radio	90
PC	1000

Discounts

Price	Discount
500	50

T

qid	V1
qi1	50
qi2	100

Note that this update sequence has no impact on the query instances in the result (first query instance still has one tuple {⟨TV,500,50⟩}, whereas the second query instance has no tuples). Therefore, no invalidation messages should be generated. However, as we have seen in Section 3.5, if we are using the synchronous invalidation approach, then the over-invalidation term is $O = (Products^- \bowtie Discounts^+) \cup (Products^+ \bowtie Discounts^-)$. Therefore, in this case, the tuple ⟨PC,\$1000,\$50⟩ will be in the over-invalidation term. Consequently, the query instance, qi1, which joins with this tuple, will be (over-)invalidated. □

Therefore, in general, we have

$$\mathcal{Q}^O \subseteq \pi_{qid}\sigma_{\Theta_T}(O \bowtie T),$$
$$\mathcal{Q}^U \subseteq \pi_{qid}\sigma_{\Theta_T}(U \bowtie T).$$

The reason why we have inequalities instead of equalities in these terms is that a query instance maybe affected by multiple invalidation terms. Consequently, some query instances that seem to be over-invalidated (under-invalidated) due to one term may actually be invalidated due an other one.

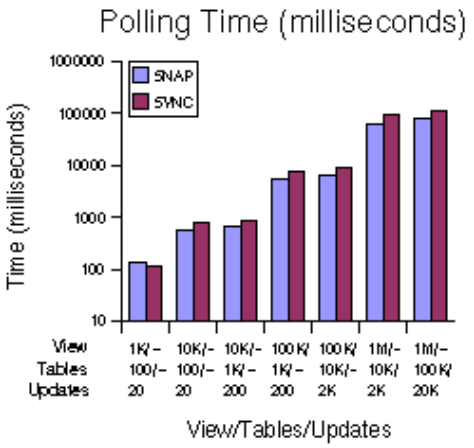

Figure 8: Polling times with low Join selectivity

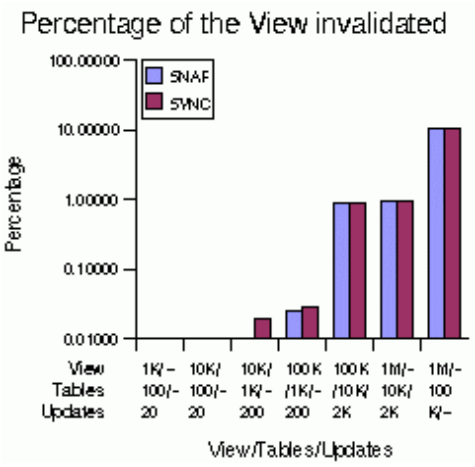

Figure 9: Invalidation % with low Join selectivity

6 Experiments

In this section, we describe a set of experiments to evaluate the effects of using view invalidation for dynamic content caching. One of the main questions that arises in the proposed framework is the overhead of executing polling queries to determine if cached query results are invalidated due to updates. In an E-commerce application, the number of queries can be very large and therefore we first determine the overhead of executing polling queries with consolidation as the number of cached queries increases. We next evaluate the impact of over-invalidation in the context of the synchronous approach. Note that, in the following, our experimental evaluation is based on the snapshot and synchronous approaches for view invalidation. The results for the asynchronous approach are similar.

6.1 Polling Query Overhead

Our experimental platform consisted of a PC workstation running ORACLE 8i DBMS. In order to study the cost of execution polling queries, we set up the following query type which was used as a candidate for content caching:

- The query type we used for the experiment is

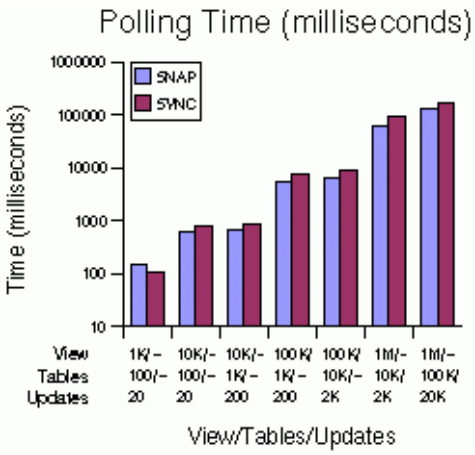

Figure 10: Polling times with high Join selectivity

```
select * from House, School
where House.location = School.location and
      School.score>$P1;
```

- We assumed that the join attribute location is indexed in both House and School tables.
- Sizes of the view and database tables and the rate at which updates are processed varied as follows:

View	House	School	Updates
1000	100	100	20
10000	100	100	20
10000	1000	1000	200
100000	1000	1000	200
100000	10000	10000	2000
1000000	10000	10000	2000
1000000	100000	100000	20000

For the first set of results, we have experimented with databases where the join selectivity is low and hence invalidation is very rare. For this purpose, we used the following data distribution for the join and query attributes:

Data Distribution	
House.location	1...1000
School.location	1...1000
School.score	1...1000000

Figure 8 depicts the polling query execution times for different workloads. Note that the times reported are actual clock times and not simulation times. In particular, the execution times of the two approaches are comparable. The main factor governing the cost of invalidation is the number of query instances. There is a linear correlation between the number of queries and the increase in the execution times. For example, 1000 queries result in the polling overhead of around 100 milliseconds whereas 1 million queries take about 60 to 75 seconds. Although this may appear to be excessive, but consider the alternative: one million trigger definitions, and their evaluation is likely to incur significantly larger overhead. Also, our experience indicates that trigger definitions beyond 10,000 become infeasible in commercial DBMSs. Most of this overhead is due to dynamic insertion and deletion of trigger definitions which cannot be avoided for dynamic content caching based on triggers. Figure 9 depicts the percentage of view (cached queries) that is invalidated. For the case with 1 million queries and 2000

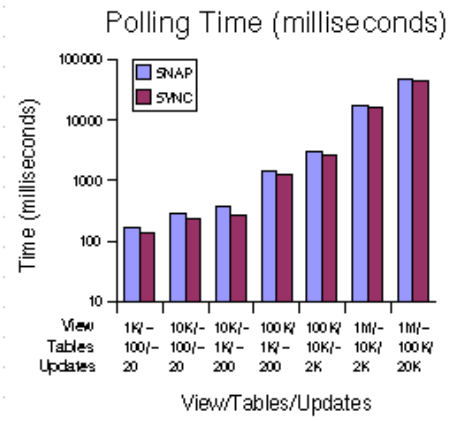

Figure 11: Polling times with low Join selectivity (modified query plan)

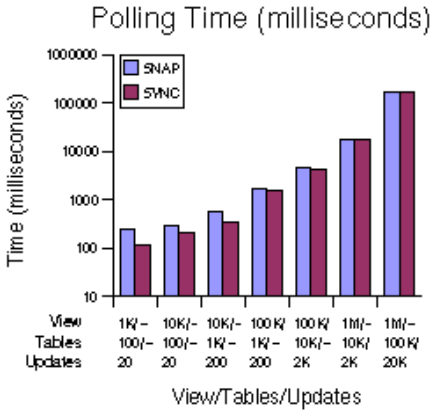

Figure 12: Polling times with high Join selectivity (modified query plan)

updates, the invalidation percentage is 1%. However, as the updates are increased to 20,000 (significantly high update activity, approximately 10% of the data is updated), the invalidation percentage reaches 10%. As a design guideline, for a database which has an update rate beyond 10% during a short-interval, our recommendation is to not to allow caching of such content. Viewing it another way, we suggest that the invalidation cycle should be run within a period when the update activity reaches 10% of the database.

The next experiment we conducted was to increase the amount of invalidation by increasing the join selectivity of the two relations. This was achieved by restricting the domain of House.location and School.location to $1 \ldots 100$. Figure 10 depicts the execution times in the modified setup. From the figure it can be seen that although the absolute number of invalidations increases slightly (not shown), the polling query execution time is not impacted. Thus, the major factor governing the polling query overhead is the number of cached queries.

Figures 11 and 12 depict the results of execution times when the polling query plan was modified to take advantage of the indexes in the database. Interestingly, the results indicate that with very simple

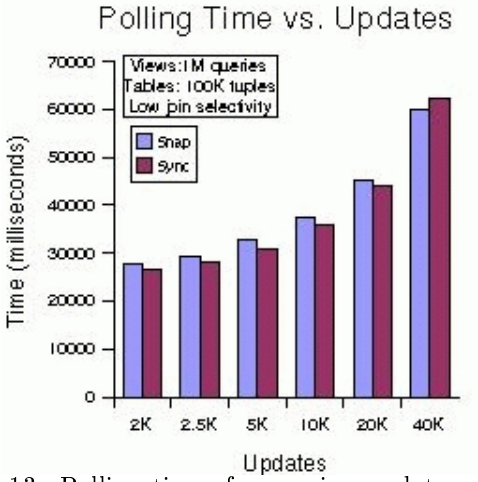

Figure 13: Polling times for varying update activity (modified query plan)

database tuning we were able to reduce the polling query execution times when the join selectivity is low. In particular, when compared to Figure 8, the last data point with 1 million queries and 20,000 updates, the execution time improved in Figure 11 by about 40% when compared to the corresponding data point in Figure 8. However, in the cases with high update activity and high join selectivity the times in the new set-up increased by about 20%. The experiments above clearly establish the viability of the polling query based view invalidation and depending upon the application characteristics, the query plans can be tuned to reduce the execution times significantly.

Finally, in Figure 13 we depict the amount of time it takes to evaluate the polling queries for 1 million cached HTML pages with varying amount of update activity. The tables sizes was set to 100,000 tuples with attributes values chosen for low join selectivity. The updates were varied from 2000 updates to 40,000 updates. The total times vary from approximately 30 seconds to 60 seconds. This result indicates that the proposed approach is robust enough to deal with occasional burst of updates between invalidation cycles.

6.2 Over-Invalidation

In this set of experiments, we observe the effects of various parameters on the over-invalidation behavior. For this experiment, we again use the query type

```
select * from House, School
where House.location = School.location and
      School.score > $P1;
```

In this experiment, we used the following data distribution for the join and query attributes:

Data Distribution	
House.location	$1 \ldots 1000$ (uniform)
School.location	$1 \ldots 1000$ (one school per location)
School.score	$1 \ldots 1000$ (uniform)
House$^+$.location	$1 \ldots 1000$ (uniform)
School$^-$.location	$1 \ldots k$ (sequentially)

where k is the number of deleted schools. Therefore, for each inserted house, the probability that it is from a deleted school location is $\frac{k}{1000}$.

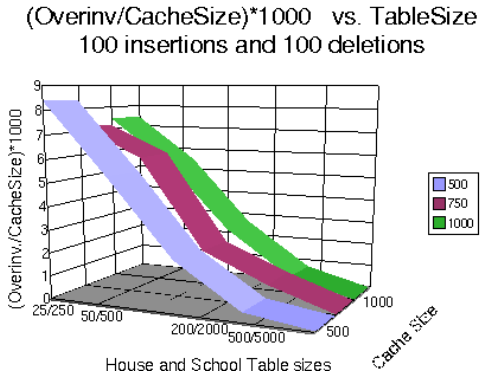

Figure 14: Over-Invalidation vs. table and cache sizes

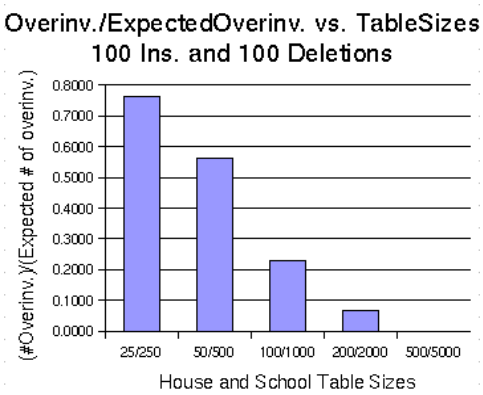

Figure 15: Over-Invalidations/Estimated Over-Invalidations vs. table sizes

Figure 14 plots the term

$$\frac{Number\ of\ over{-}invalidations}{CacheSize} \times 1000,$$

as a function of table and cache sizes. The first thing to notice, is that the over-invalidation is limited to less than 1% of the cache. This ratio is independent of the cache size. Furthermore, as the table sizes increase, the amount of over-invalidation decreases very quickly, approaching to 0 when database contains 500 houses and 5000 schools. This drop is due to the fact that, when tables are sufficiently large, any query which seems to be over-invalidated due to an update is indeed invalidated due to another one.

Furthermore, although the expected number of over-invalidated tuples is independent of the actual table sizes, experiments showed that the number of actual over-invalidations were also getting smaller than expected as tables became larger (Figure 15).

Finally, Figure 16 shows how over-invalidation is affected by the number of updates. The number of over-invalidations increases as predicted by the over-invalidation term, roughly doubling for each 100% increase in $House^+$ or $School^-$. But, over-invalidation is limited to around 1% of the cache size.

In this section, we focused our observation on the performance of polling queries with consolidation and the impact of over-invalidation. A more detailed evaluation which discusses issues related to the deployment

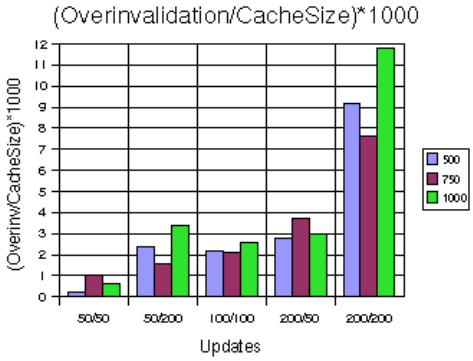

Figure 16: Over-Invalidation vs. updates and cache sizes

of this technology, such as comparisons with alternative techniques, can be found in [22].

7 Related Work

As the number of Internet-based applications increases, the need for systems that can quickly deliver data-driven content becomes more apparent. Since the main bottleneck in the delivery of such content is the server side, existing network-based content distribution structures does not address this urgent need. Recently, there has been an increasing number of efforts aimed at preventing the database from becoming a bottleneck in various distributed applications [23, 24] Surveys of these applications and existing technologies can be found in [25]. One earlier solution was to cache business data outside of the DBMS to reduce the database access load. Oracle and Persistence Software developed middle-tier data caching products along these lines. More recently, however, the caching of dynamically generated pages at the web servers has been shown to be more efficient than the caching of the data itself [26]. Consequently, DBMS and application server suppliers, such as Oracle, announced web caches along with their more traditional data caches.

At the time of the writing, various commercial caching solutions exist. Major application server vendors, such as IBM WebSphere, BEA WebLogic, and Oracle Application Server, focus on application level caching. Xcache and Spider Cache both provide invalidation solutions based on manually specified triggers and they do not support automated invalidation. Javlin and Chutney provide middleware level cache/pre-fetch solutions, which lie between application servers and underlying DBMS or file systems. Again, they do not provide automated invalidation functionalities. In [27] Qiong *et al.* present an extension to the existing federated features in IBM DB2, which enables a DB2 instance to become a middle tier database cache without any application modification. Oracle web cache addressed this challenge by providing time-based or event-based invalidation of the cache contents. The invalidation events can be generated by user supplied triggers or specially crafted application scripts. Oracle web cache, however, does not provide

a framework for systematically generating invalidation messages in the presence of data updates. Challenger *et al.* proposed a solution, based on explicitly maintained dependencies between data and cached objects, that addresses the update problem [1]. An alternative invalidation-based approach, where maintaining an explicit mapping between data and cached objects is avoided, is proposed in [3].

8 Conclusions

Fast invalidation is a key point for enabling dynamic content caching while maintaining cached web pages fresh. Various applications, including e-commerce sites, on the other hand do not require the caches to reflect all the data in the database, yet they require that all out-dated pages in the caches are *invalidated* in a timely fashion. In this paper, we show that invalidation is inherently different from view maintenance. We develop algorithms that benefit from this difference in reducing the cost of update management in certain applications and we describe an invalidation framework that benefits from these algorithms. Our experimental evaluation establishes view invalidation as a viable approach for enabling dynamic content caching.

References

[1] J. Challenger, A. Iyengar, and P. Dantzig. Scalable System for Consistently Caching Dynamic Web Data. In *INFOCOM'99*, March 1999.

[2] B. Smith, A. Acharya, T. Yang, and H. Zhu. Exploiting Result Equivalence in Caching Dynamic Web Content. In *USENIX Symposium on Internet Technologies and Systems*, 1999.

[3] K.S. Candan, W.-S. Li, Q. Luo, W.-P. Hsiung, and D. Agrawal. Enabling Dynamic Content Caching for Database-Driven Web Sites. In *2001 ACM SIGMOD* , May 2001.

[4] E.A. Rundensteiner, A. Koeller, and X. Zhang. Maintaining data warehouses over changing information sources. *Communications of the ACM*, 43(6):57–62, 2000.

[5] A. Gupta and I. S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Bulletin of the Technical Comm. on Data Eng.*, 18(2):3–18, June 1995.

[6] O. Shmueli and I. Itai. Maintenance of Views. In *ACM SIGMOD*, 1984.

[7] E. N. Hanson. A Performance Analysis of View Materialization Strategies. In *ACM SIGMOD*, pages 440–453, May 1987.

[8] B. Lindsay, L. Haas, C. Mohan, H. Pirahesh, and P. Wilms. A Snapshot Differential Refresh Algorithm. In *ACM SIGMOD*, 1986.

[9] J. A. Blakeley, P. A. Larson, and F. W. Tompa. Efficiently Updating Materialized Views. In *ACM SIGMOD*, pages 61–71, 1986.

[10] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *ACM SIGMOD*, pages 157–166, May 1993.

[11] D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek. Efficient View Maintenance in Data Warehouses. In *1997 ACM SIGMOD*, pages 417–427, May 1997.

[12] K. Salem, K. S. Beyer, R. Cochrane, and B. G. Lindsay. How to roll a join: Asynchronous Incremental View Maintenance. In *2000 ACM SIGMOD*, pages 129–140, May 2000.

[13] S. Adalı, K.S. Candan, Y. Papakonstantinou, and V.S. Subrahmanian. Query Caching and Optimization in Distributed Mediator Systems. In *1996 ACM SIGMOD*, pages 137–148, May 1996.

[14] Q. Luo, J. F. Naughton, R. Krishnamurthy, P. Cao, and Y. Li. Active Query Caching for Database Web Servers. In *2000 WebDB (informal proceedings)*, pages 29–34, 2000.

[15] C. Chen and N. Roussopoulos. The implementation and performance evaluation of the ADMS query optimizer: Integrating query result caching and matching. In *EDBT*, pages 323–336, 1994.

[16] J.A. Blakeley, N. Coburn, and P.-A. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM TODS*, 14(3):369–400, 1989.

[17] C. Elkan. Independence of logic database queries and updates. In *PODS 90*, pages 154–160, 1990.

[18] A.Y. Levy and Y. Sagiv. Queries independent of updates. In *VLDB 93*, pages 171–181, 1993.

[19] A. Gupta, Y. Sagiv, J.D. Ullman, and J. Widom. Constraint checking with partial information. In *PODS 94*, pages 45–55, 1994.

[20] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View Maintenance in a Warehousing Environment. In *ACM SIGMOD*, pages 316–327, May 1995.

[21] M.J. Carey and D. Kossmann. Processing top n and bottom n queries. *Data Engineering Bulletin*, 20(3):12–19, 1997.

[22] W.-S. Li, W.-P. Hsiung, D.V. Kalashnikov, R. Sion, O. Po, D. Agrawal, and K.S. Candan. Issues and Evaluations of Caching Solutions for Web Application Acceleration. In *VLDB*, 2002.

[23] Q. Luo and J.F. Naughton. Form-based proxy caching for database-backed web sites. In *VLDB) Conference*, pages 191–200, 2001.

[24] A. Datta, K. Dutta, H.M. Thomas, D.E. VanderMeer, Suresha, and K. Ramamritham. Proxy-Based Acceleration of Dynamically Generated Content on the World Wide Web: An Approach and Implementation. In *SIGMOD 02*, June 2002.

[25] C. Mohan. Caching technologies for web applications. In *Tutorial presented at VLDB*, 2001.

[26] A. Labrinidis and N. Roussopoulos. WebView Materialization. In *ACM SIGMOD*, 2000.

[27] Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B.G. Lindsay, and J.F. Naughton. Middle-tier Database Caching for e-Business. In *SIGMOD 02*, June 2002.