April 23, 2007

RT0727
Engineering Technology 18 pages

Research Report

Classifying Suitability of Applications for the Hybrid Architecture

Teruo Koyoanagi, Yosuke Ozawa

IBM Research, Tokyo Research Laboratory
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa 242-8502, Japan

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Classifying Suitability of Applications for the Hybrid Architecture

Teruo Koyanagi[1], Yosuke Ozawa[1]

[1] IBM Japan, Tokyo Research Laboratory, Shimotsuruma,
1628-14 Kanagawa-ken, Japan
{teruok, ozawaysk}@jp.ibm.com

Abstract. The hybrid architecture, HM-Hybrid can be an effective solution for a high volume stateful event processing that requires searches and updates to be consistent. The hybrid architecture provides high throughput by enabling batch update, but in the situation batch update is usually regarded as unsuitable because of its asynchronous characteristics. Therefore the hybrid architecture can be efficient for applications that must handle high volumes of searches and updates in a consistent manner. However, the range of its efficiency is still unclear, since its advantages depend on the application's characteristics. We provide a methodology to estimates break-even points between advantages and disadvantages of the hybrid architecture. To compare them we introduce performance comparison models, and figure out the significant parameters to determine the throughputs of performance bottlenecks on the models. The range of efficiency of the hybrid architecture is clarified by this method, and it is shown to be applicable for a broad range of application patterns in our experiments.

Keywords: application cache, performance evaluation, software engineering, software architecture, materialized views, and model-driven development

1 Introduction

There are certain applications which produce high volumes of searches and updates simultaneously. A typical application of this type is stateful event processing for monitoring. Monitoring essentially requires search and update processing for each event. To track suggestive key performance indicators, each event should be correlated with multiple types of other events, and the current state of the calculation should also be saved in a database at the time when that event occurred. Therefore each event causes at least one search for correlations and one update for saving the current state.

If the database accesses become very frequent, read-only and write-only accesses to the database can achieve high throughputs by applying well known solutions. Read-only accesses can be accelerated with simple cache solutions. Write-only accesses can be accelerated with batch solutions. Batch update collects the write requests over an interval, and then executes them asynchronously as a large, fast job.

However such solutions often fail when read and write accesses to a database are mixed. The batch update cannot be used because of its asynchronous characteristics. For example, the monitor always needs the latest data to process the next event. The event must be correlated with the most recent events because of the real-time requirements for monitoring. There are no efficient general solutions for the mixed read-write cases.

Also the result cache is inefficient. When write accesses occurred frequently, the result cache cannot be simply updated like a write-through cache, because the result cache must store the search results of SQL select statements. Therefore, the cached results may be invalidated after very short intervals and the cache hit ratio will be low.

We suggested a hybrid architecture, HM-Hybrid which uses a maintained result cache and a batch update function to solve these problems. The hybrid architecture was used for event processing, and in our experiments handled 1,207 events per second, with about 43,000 queries per second. This is 6 times better performance over a naive caching architecture. Therefore the hybrid architecture seems to be effective for high volumes of select/update query processing while maintaining consistency.

However the range of efficiency of this architecture is still unclear when we reuse it for other applications. The application of the hybrid architecture to monitoring succeeded because its benefit from batch update exceeded the maintenance costs and because all of the required data for the monitor was in memory. However the maintenance cost depends on the application patterns, including the update frequency and the number of indexes. Also, the overhead of fetching data from the database must be covered when all of the required data is not in memory.

In this paper, we provide a methodology to evaluate whether or not the hybrid architecture will be efficient for the given conditions of an application. To clarify the efficiency of the hybrid architecture, we introduce performance comparison models to compare the performance effects of the advantages and the disadvantages of the hybrid architecture. Because they are on the different criteria, we must transform them into the same standard. In the performance comparison models, these effects are transformed into the throughputs. These throughputs are determined by significant parameters which are derived from the application characteristics. We figure out the ranges of the significant parameters indicating the application characteristics for which the hybrid architecture is efficient. To measure these throughputs with various significant parameters, we conducted three experiments. As a result, the hybrid architecture is shown to be applicable for a broad range of the applications.

First, we review related works in Section 2, and then describe the concept and components of our hybrid architecture in Section 3. Then, in Section 4, the evaluation methodology is illustrated in details, and designed three experiments to estimate the value of the throughputs that indicate performance boundaries of the hybrid architecture. In Section 5, we discuss how to use the experimental results to classify the application patterns for which the hybrid architecture will be efficient. The classification methodology is summarized in a chart to apply to other applications.

2 Related Works

In our previous work, we suggest the hybrid architecture as a solution to solve a performance problem for model-driven business performance management [1]. In this paper, by applying a practical and established evaluation method, we will show the hybrid architecture can be applied to broad range of applications. Finally we provide a classification methodology to determine whether the hybrid architecture can be efficient for a concrete application on a concrete system.

Batch update is a technology that has been frequently used for a long time in almost all computer systems, and is still efficient in improving the throughput of block devices with limited bandwidth, such as networks and storage. It collects the write requests over an interval, and then executes them asynchronously as a large, fast job. While it greatly improves throughput, its usage scenarios are strongly limited because of its asynchronous characteristics. In this paper, we propose extending the usage of our hybrid architecture to other applications which requires consistency in update and search.

The hybrid architecture can be considered as an application of materialized views for high volume task processing. For each search statement, it incrementally maintains consistent memory-resident indexes. There are many research reports on view maintenance in the field of materialized views. They include useful algorithms to maintain our results cache.

Gupta and Mumick [2] categorized maintenance algorithms by the amounts of information that can be utilized when the views are refreshed. Ceri and Widom [3] provided a method to derive production rules to maintain materialized views incrementally. Their system accepts a language which is a subset of SQL, and the derived rules are also represented and computed in the same language. The maintenance function we used to keep result cache fresh is a kind of incremental maintenance.

Quass et al. [4] provides a method to derive a set of self-maintainable views which can be maintained without accessing the underlying database. This concept is important for any hybrid architecture to maintain the consistency of the result cache using only update notifications. Our model-driven approach can provide a complete set of access patterns by compilation as described in [1]. These access patterns, the full set of SQL statements used in the application, are equivalent to full information about the constraints for view maintenance. Hence our method can use almost all of the constraints, and self-maintainable views such as the result cache can be derived.

Ross et al. [5] develop an optimization method using additional views to maintain the target views. It includes a cost model of the incremental view maintenance to estimate whether keeping such additional views is efficient. In contrast to our approach, their cost model of materialized views assumes that the views are created in the database system, but we use a cost model for memory resident indexes.

Event correlation is an important idea for recent-event-processing systems. Event correlation is traditionally researched in the system management field. Hasan et al. [6] describe simple models for causal and temporal correlation. Because of the characteristics of diagnosis, the event correlation engines in this field usually do not mention persistency for correlation states, in contrast to our requirements.

Continuous queries [7] have been proposed for querying data streams. They essentially require the same processing mechanisms provided by our hybrid architecture. They usually assume insertion only in contrast to our approach, which allows all queries. This is due to differences in the available information and the constraints in processing queries.

3 Hybrid Architecture with Model-driven Approach

The basic concepts of our hybrid architecture, *HM-Hybrid* are covered in this section. The model-driven approach which makes it efficient is also shown. First, the model-driven result cache is illustrated. The model-driven result cache realizes the hybrid architecture. Then the way the hybrid architecture works at runtime is described. The last section describes a generalized maintenance algorithm which was developed to generate the maintenance functions from the given access patterns.

3.1 Generating Result Cache from the Model

Fig. 1 shows the concepts of the model-driven approach for cache generation. The approach requires full information on data access patterns. Once we have all of the data access patterns, we can generate a result cache which has optimized maintenance functions for the data access patterns. The data access patterns can be considered as constraints representing when and which target results in the cache are maintained. The maintenance functions are generated by using information about such constraints. For the details, the methods for deriving maintenance functions are well known in the field of materialized views.

In our approach, both the indexes for the SQL select statements and the maintenance functions are generated as a result cache from the application model. We call the generated result cache the “Model-driven result cache”.

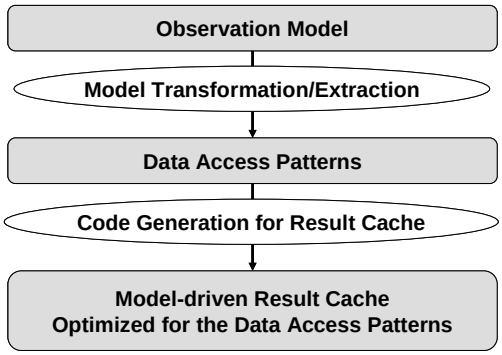

Fig. 1. Model-driven cache generation.

At least one index is generated from each prepared SQL select statement. The index for a select statement consists of a key and a condition (**Fig. 2**). The key is from

the parameterized predicates of a where-clause, and the key is used for the index entry. The value of the index entry will be a result set of IDs for instances. The instance is a database record itself, or an application object which is restored from the database records. The condition is from the static predicates of the where-clause and the instances must satisfy the conditions when they are added to the index.

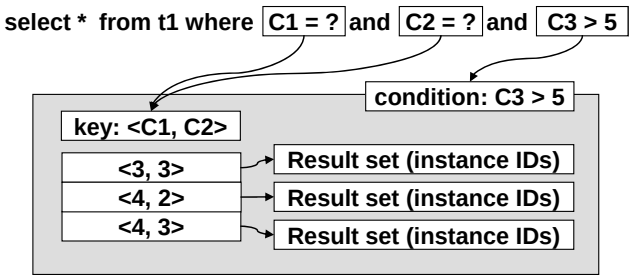

Fig. 2. Deriving an index from a where-clause.

3.2 Runtime Components of the Hybrid Architecture

The hybrid architecture consists of the model-driven result cache and a batch processor for the batch update function (**Fig. 3**). Asynchronous batch updates are not usually allowed when the database should be responding to user queries with recent data. In this architecture, the facade of the model-driven result cache substitutes for the database when users query with select statements, and this allows asynchronous batch update.

The model-driven result cache is a combination of the result cache and its maintenance functions. The result cache contains indexes to store every search result that can be selected by the given access patterns.

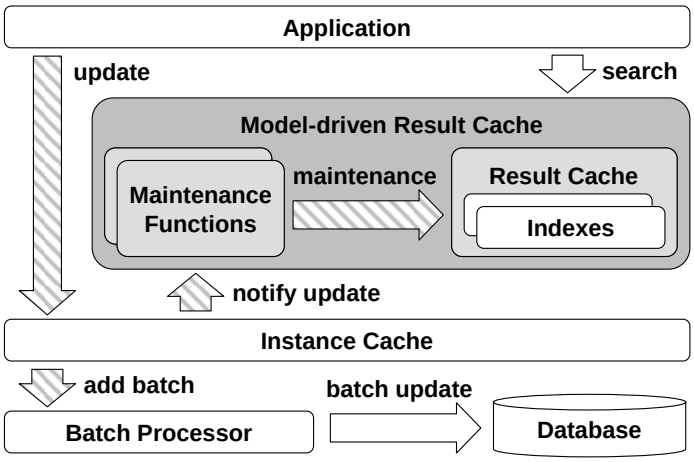

Fig. 3. Runtime components of hybrid architecture.

When an instance is updated by the application, the update information is sent to the maintenance functions to keep the result cache fresh. At the same time, the update is sent to the batch processor to add this update to the next batch update. The batch updates are executed periodically by the batch processor.

3.3 Maintenance algorithm for result cache

The maintenance algorithm for an index of the result cache is described in pseudocode in **Fig. 4**. The code assumes that the index *rc* has **add**, **remove**, and **contains** methods as index manipulations. This algorithm has three open methods, **createKey**, **isKey**, and **testCondition**. They are derived from the given access patterns, and belong to *rc* as part of its maintenance functions.

When the maintenance function is called with update information, *i*, *m*, v_{old} and v_{new} as its arguments, it creates two keys k_{old} and k_{new} from corresponding updated attributes. The k_{old} is used to remove the existing entry in the result cache, and k_{new} is used to add the new entry.

In line 3, the function checks whether the index contains the target instance *i*. Then the derived condition is tested to decide if the updated *i* should be indexed in the result cache by considering the new value v_{new}.

rc: an index of the result cache
i: an updated instance
m: the name of the updated attribute of *i*
v_{old}: the old value of the updated attribute
v_{new}: the new value of the updated attribute

```
01.  k_old = rc.createKey(i, v_old)
02.  k_new = rc.createKey(i, v_new)
03.  if (rc.contains(i)) then
04.    if (! rc.testCondition(i, v_new)) then
05.      rc.remove(k_old, i)
06.    else
07.      if (rc.isKey(m)) then
08.        rc.remove(k_old, i)
09.        rc.add(k_new, i)
10.      endif
11.    endif
12.  else
13.    if (rc.testCondition(i, v_new)) then
14.      rc.add(k_new, i)
15.    endif
16.  endif
```

Fig. 4. Maintenance algorithm.

4 Evaluation of the Hybrid Architecture

In this section, we evaluate our hybrid architecture. First, we describe the overall strategy of the evaluation methods. Detailed experiments and their results are shown in the following sections.

For the sake of comparison, two alternative architectures are considered and tested to help define the limits of effectiveness of the hybrid architecture. We determine the significant parameters in a test environment by considering the pros and cons of each architecture, and compare them in order to estimate the breakeven points for the advantages and disadvantages of the hybrid architecture.

We focused on applications that need to process many updates and simultaneously process as many searches as updates, because the hybrid architecture was developed to address problems in such situations.

4.1 Evaluation Methodology

The hybrid architecture consists mainly of the batch processor and the incremental maintenance functions for the result cache, and relies on two major premises to enable batch update. The first premise is that the result cache is always maintained as the base table changes. The second premise is that the entire result cache, including all indexes, can be stored in memory. This means that the maintenance costs of the hybrid architecture increases as the update queries increase. And also means the instance cache may run out of memory because of the pressure of the index cache, it reflects to the cache hit ratio of the instance cache. In other words, the total performance of the hybrid architecture may depend on the maintenance cost and the memory size of the instance cache.

It is always the case that disks are the bottleneck of the system when updates are frequent. Because batch update removes the disk performance bottleneck, the hybrid architecture is efficient when the performance bottleneck is the disk. However, if the maintenance of the result cache or the read accesses to the database becomes the new performance bottleneck replacing with disk bottleneck, then the hybrid architecture can be inefficient.

Thus in order to evaluate efficiency of the hybrid architecture, we should estimate the performance boundaries that can be observed as these potential bottlenecks, more specifically, (1) the maximum throughputs of the write accesses without batch update, (2) the maximum throughputs of maintenance for the result cache, and (3) the maximum throughputs of the read accesses with a size restriction for the instance cache. Comparing these throughputs reveals the range that the hybrid architecture can be efficient.

To estimate these throughputs, we introduce performance comparison models to figure out the significant parameters. The significant parameters are the parameters that mainly determine these throughputs. In particular, the batch size is a significant parameter for (1), and the number of indexes is for (2), and the cache hit ratio is for (3). Detailed relations between the significant parameters and the throughputs are described using the performance comparison models in the following subsections.

To figure out the significant parameters, we must assume the same condition other than the features which generate the target performance bottlenecks. We use three different architectures to make the performance comparison models. They have the corresponding performance bottlenecks. **Table 1** summarizes these architectures and their performance bottlenecks. The bottlenecks are derived from the respective pros and cons of the architecture described in the table.

The hybrid architecture has a result cache and batch update but it has maintenance cost to keep its result cache consistent to prevent database accesses for the search queries. The simple-index architecture has a result cache but no maintenance functions. Thus the architecture has no maintenance costs but cannot do any batch updates. The no-index architecture has no index cache, so all of the memory can be used for the instance cache. While this also disables batch update for the no-index architecture, the no-index architecture has better throughput of read accesses.. On the other hand, the simple-index architecture's throughput of read accesses is the same as the hybrid architecture's one, because both of them have the same restriction of available memory size for the instance cache.

We conducted three experiments. They are described in the following subsections with performance comparison models and their results. Subsection 4.3 shows an evaluation for the maximum throughput with and without batch update. Subsection 4.4 shows an evaluation for the throughput of the maintenance for various numbers of indexes. And Subsection 4.5 shows an evaluation for the maximum throughputs of read accesses for various cache hit ratios.

Table 1. Cache architectures and their potential bottlenecks

	Hybrid Architecture	**Simple-index Architecture**	**No-index Architecture**
Features	Instance cache, Result cache, Maintenance, Batch update	Instance cache, Result cache, Simple eviction	Instance cache
Pros	Faster update	No maintenance	No maintenance, Larger instance cache
Cons	Maintenance, Smaller instance cache	Slower update, Smaller instance cache	Slower update
Bottlenecks	Batch update, Read, Maintenance	Update, Read	Update Read

4.2 Testing Platform

The testing platform for following experiments consists of one 2.2 GHz CPU dual core machine and one 2.4 GHz CPU dual core machine connected via a 1 Gbps LAN (**Fig. 5**). The first machine was the application server, and the second was the database server. The detailed specifications of each machine are shown below.

- **Application Server**
  - CPU: 2.2 GHz dual core Opteron 275 x2, 2 MB 2L cache
  - Main Memory: 4 GB PC3200 RAM
  - Hard Disk: 80 GB SATA 7200 rpm
  - Operating System: Windows 2000 Service Pack 4
  - WebSphere Process Server 6.0.1.2

- **Database Server**
  - CPU: 2.4 GHz Opteron 250 x2, 1 MB 2L cache
  - Main Memory: 4 GB PC3200 RAM
  - Hard Disk: 73.4 GB 15 Krpm Ultra320 SCSI
  - Operating System: Windows 2000 Service Pack 4
  - DB2 UDB Enterprise Edition 8.1.8

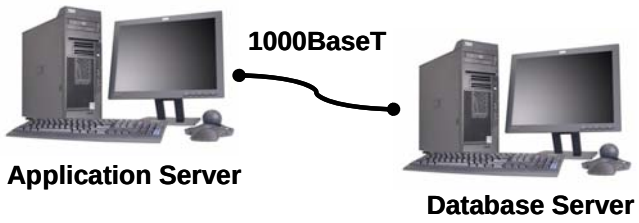

Fig. 5. Testing platform.

4.3 Evaluation for Batch Update

The greatest advantage of using the hybrid architecture is the feasibility of batch update. This feature removes the disk performance bottleneck and allows for high throughput. Thus the efficiency of the hybrid architecture can be determined by measuring how much the throughput of the hybrid architecture exceeds the maximum throughput without batch update. Even if the two other potential bottlenecks than the batch update bounds the throughput of the hybrid architecture, still it is cost-effective while these exceeds the maximum throughput without batch update.

We measure how many updates can be processed each second for various batch sizes. The batch size defines how many SQL statements are sent together using the add-batch instruction. It is a first significant parameter to figure out the throughput with and without batch update. They are used as the maximum performance of the hybrid architecture and the maximum performance of the other two architectures that we use in the performance comparison models.

Method. We executed an experiment to estimate batch update throughput when all of the statements are updates. Of course, general applications tend to issue various combinations of SQL statements, but for simplicity we only considered updates in this experiment. The target table has 6 columns, and each column consumes 8 bytes for

each record. The update statement fills in the remaining 2 columns by selecting a record with a primary key.

As batch size increases, we observed how many statements were processed in a second. To measure the throughput of the batch updates in an appropriate manner, the numbers of generated statements are multiples of the batch size from 1,000 to 14,000. For a smoother graph, we changed the intervals of the batch size. For regions that change rapidly, we used shorter intervals. Each of the throughput results is the average of ten trial runs.

Result. The results of the experiments are shown in **Fig. 6**. The vertical axis represents the number of update statements which are processed in a second. The horizontal axis represents the batch size. The throughput curve starts being satiated around the batch size of 2,000, and the throughput is about 10,000 (actually 9,892) updates/second. And it is almost satiated around the batch size of 6,000, and the throughput is about 11,000 (11,045) updates/second. Thus we say the maximum throughput of the batch update is 11,000.

Even though the batch size of 1 (no batch) gave a throughput of 231 updates/second, we used the minimum batch size of 10 with a throughput of 1,500 (1,474) updates/second as the throughput without batch update, because applications usually have multiple updates in their smallest transactions. The throughputs of 1,500 and 11,000 were compared with the results of the later experiments to estimate the boundary of the range which the hybrid architecture can be efficient. We consider that the value of 1500 update queries/second represents the bottleneck of the other two architectures, and the 11,000 updates/second is the maximum throughput which can be performed by the hybrid architecture.

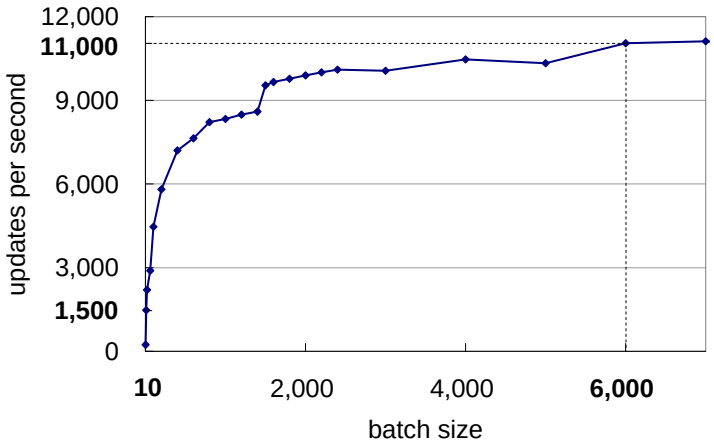

Fig. 6. Throughput curve for batch size.

4.4 Evaluation for Maintenance Cost

In this section, we introduce one of the performance comparison models to estimate the performance boundary of the hybrid architecture. It shows that the number of

indexes is the significant parameter for the throughput of the maintenance operation. And an experiment is conducted to estimate its value as the lower performance boundary of the hybrid architecture.

Method. Fig. 7 shows the performance comparison model between the hybrid architecture and simple-index architecture. In this figure, u/s stands for update queries per second. The simple-index architecture has relatively small maintenance cost at update time compared to the hybrid architecture, though batch update cannot be used. In the simple-index architecture, the indexes are maintained in a simple way that the corresponding result cache entry is simply evicted when an update occurs. Thus the maintenance cost of the simple index is assumed to be almost zero compared to incremental maintenance. The hybrid architecture has maintenance costs for the result cache to allow the use of batch update.

According to the model, the one of the lower performance boundaries appears at the point when the maximum throughput of the maintenance operation becomes less than the throughput of the simple-index architecture. It is the maximum throughput without batch update, 1500 update operations/second.

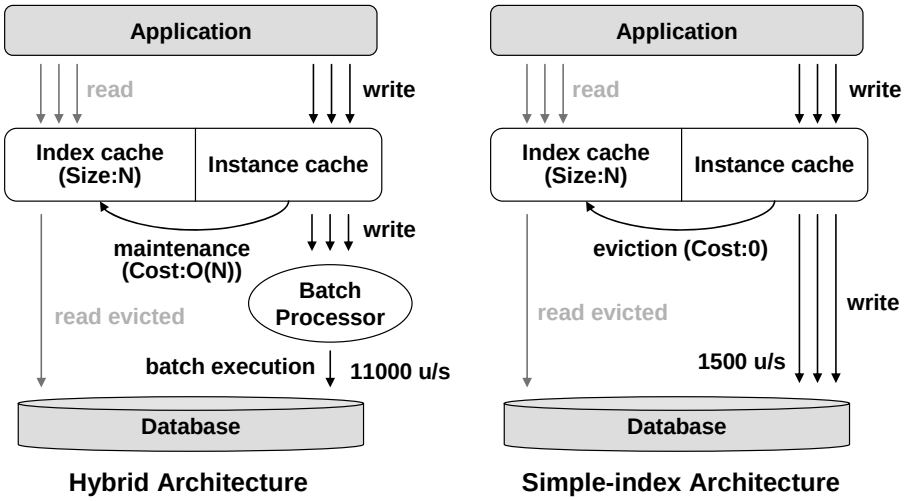

Fig. 7. Performance comparison model of hybrid and simple-index architecture.

The maintenance cost C depends on the number of indexes N, so $C = O(k\,N)$, where k is the average cost of one execution of the maintenance function. The value of k depends on the data structure of the indexes. In this experiment, we used a hash table where k is constant. If a tree structures were used, k would be $O(n \log n)$, where n is the number of index entries. In this model, we simply assume the worst case in which the indexes need to be maintained after every update.

Thus the significant parameter for the throughput of the maintenance operation is the number of indexes. We conducted an experiment that estimates the maximum throughputs of maintenance operation with changing the number of indexes. We measured how many update queries per second were processed with maintenance.

Result. The results of this experiment are shown in **Fig. 8**. The vertical axis represents the number of maintenance operations per second. The horizontal axis represents the number of indexes. The throughput for maintenance is in inverse proportion to the number of indexes, since the time for maintenance of the indexes is proportional to the number of indexes. This is consistent with the expression $C = O(k N)$ as described above.

The experiment determines two values of 11,000 maintenance operations/second for 250 indexes (actually 246) and 1,500 operations/second for 1,800 indexes (1,838). When the throughput for maintenance operation is less than 1,500 update queries per second, which is the throughput without batch update, then the hybrid architecture is never effective. In that range, the maximum throughput of the hybrid architecture is limited by the throughput of the maintenance operation and it is less than the maximum throughput of the simple-index architecture.

Note when the throughput for maintenance operation is less than 11,000 update queries per second, then the maintenance operation becomes the bottleneck of the hybrid architecture. But it is better than the throughput of the simple-index architecture, since still hybrid architecture can be effective in this range.

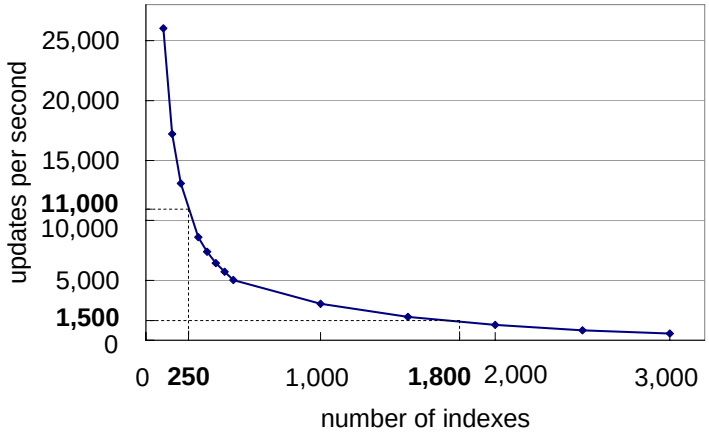

Fig. 8. Throughput curve of maintenance for number of indexes.

4.5 Evaluation for Efficiency of Instance Cache

In this subsection, the other performance model is shown. It shows that the hit ratio of the instance cache is the significant parameter to determine the throughput of the read accesses from the database. And an experiment is conducted to estimate its value as the other performance boundary of the hybrid architecture.

Method. Fig. 9 shows the performance comparison model between the hybrid architecture and the no-index architecture. In this figure, u/s stands for update queries per second. The hybrid architecture must have the all indexes of the result cache on the memory to enable batch update, because the search queries must be processed with fresh data in the maintained indexes rather than the database updated

asynchronously. Thus the no-index architecture can use relatively more memory for the instance cache than the hybrid architecture. The lack of memory may cause more database accesses to read the data. Therefore the comparison between these two architectures corresponds with the comparison between the throughput of the read accesses with smaller instance cache and the throughput without batch update.

To evaluate the effect of the smaller instance cache, we measured how many records could be read in a second while changing the cache hit ratio of the instance cache. The effect of the instance cache will differ depending on the application patterns, the reusability of records, the update frequency, etc. However such differences of application patterns will be summarized in the hit ratio of the instance cache. Thus we use this metric as the significant parameter for this evaluation.

In this experiment, the load for read accesses is generated as requests to retrieve random records by their primary keys. It does not include the cost of selection. In order to find the lowest boundary of efficiency, we just took the worst case for the hybrid architecture, which can select more quickly by using the memory-resident index.

The read cost from the memory is relatively very small compared with the read cost from the database. Thus the read cost from memory is assumed to be approximately zero. The throughput for reads is expressed as $K/(1-R)$, where R is the cache hit ratio, and K is the raw throughput of reads from the database.

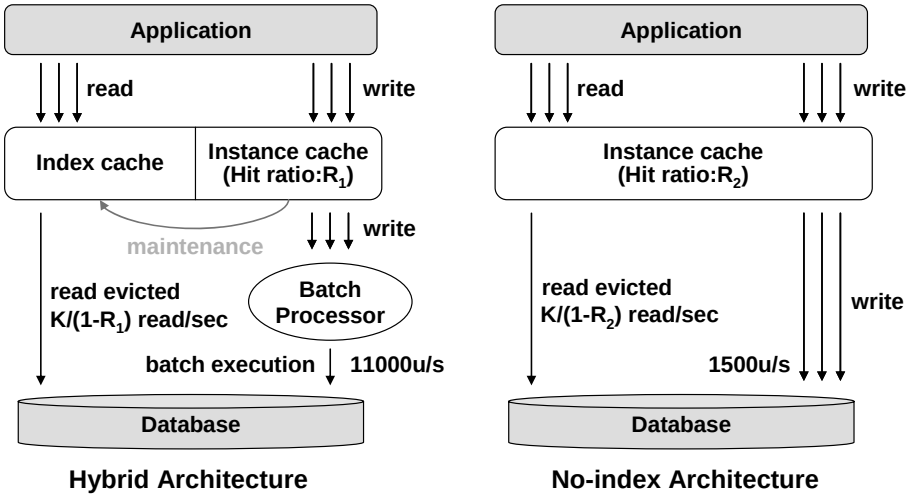

Fig. 9. Performance comparison model of hybrid and no-index architecture.

Result. Fig. 10 shows the graph of throughput (reads per second) for each cache hit ratio. The vertical axis represents the number of reads processed in each second. The horizontal axis represents the cache hit ratio.

The experiments determine one significant point which is 11,000 for the cache hit ratio of 60%. When the throughput of reads is less than 11,000 update queries per second, reads become the bottleneck of the hybrid architecture. But the hybrid architecture is still efficient in this range, because it is better than the throughput of the no-index architecture. There is no breakeven point for the cache hit ratio, because

even if all of the data is read from database, which is the case for the cache hit ratio of 0%, the throughput (around 5,000) was not less than 1,500.

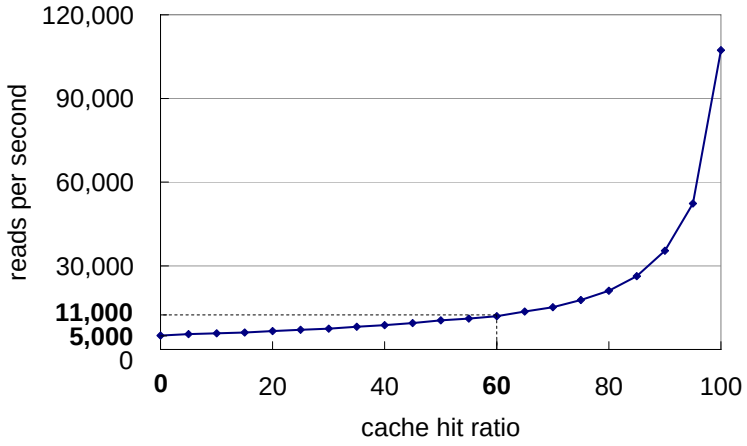

Fig. 10. Read throughput versus cache hit ratio.

5 Discussion

We found the effective limits of the hybrid architecture through these three evaluations, and illustrate the limits using two significant parameters, the number of indexes and the cache hit ratio, in **Fig. 11** and **Fig. 12**. Three ranges are shown in these figures. Range A represents the most effective range for the hybrid architecture, because the two parameters are not bounded by the throughput of batch updates. Range B represents the range where the hybrid architecture is still effective, because these two parameters bound the throughput of batch update. Range C represents the range where the hybrid architecture is ineffective.

Fig. 11 shows the classification of ranges by the number of indexes. The figure shows that the hybrid architecture is most effective for less than 250 indexes and is still effective for less than 1,800 indexes. General applications are not likely to have more than 250 indexes, even if the multiple indexes are generated for each where-clause. They are even less likely to have more than 1,800 indexes. Therefore the number of indexes is unlikely to pose real-world problems for the hybrid approach.

Fig. 12 shows the classification of ranges by the cache hit ratio. The figure shows that the hybrid architecture is the most effective for cache hit ratio above 60% and is still effective at lower values. However, these results are based on the assumption that the number of updates and the number of reads are the same. Changing that assumption will also change these results. For example, if the number of reads is 4 times the number of updates, then the hybrid architecture will become ineffective if the cache hit ratio is below 15% (**Fig. 13**). In such situations, the throughput for reads must be more than 4 times the throughput of updates. Thus, the hybrid architecture can be effective as long as all of the indexes are stored in memory.

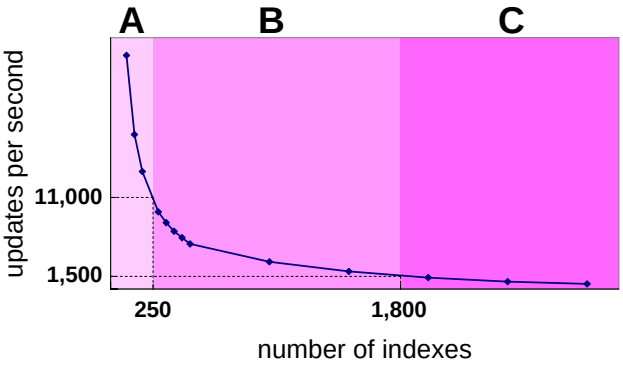

Fig. 11. Classification of ranges by the number of indexes.

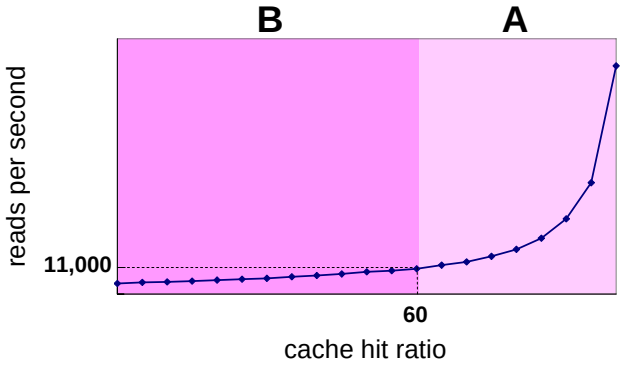

Fig. 12. Classification of ranges by the cache hit ratio.

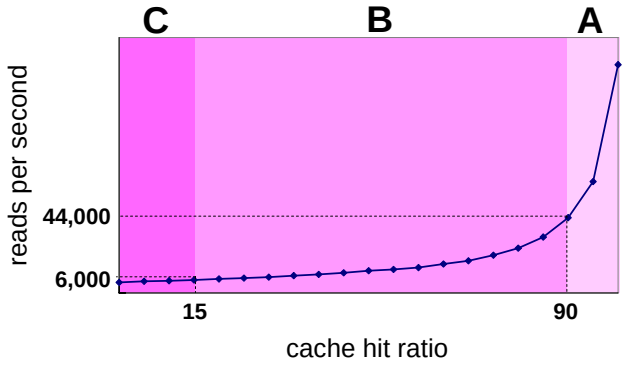

Fig. 13. Classification of ranges by the cache hit ratio when the number of reads is 4 times the number of updates.

Finally, we summarize a classification methodology for a given application to determine whether or not the hybrid architecture will be efficient for it (**Fig. 14**). The methodology consists of three major steps based on the evaluation methodology we discussed above.

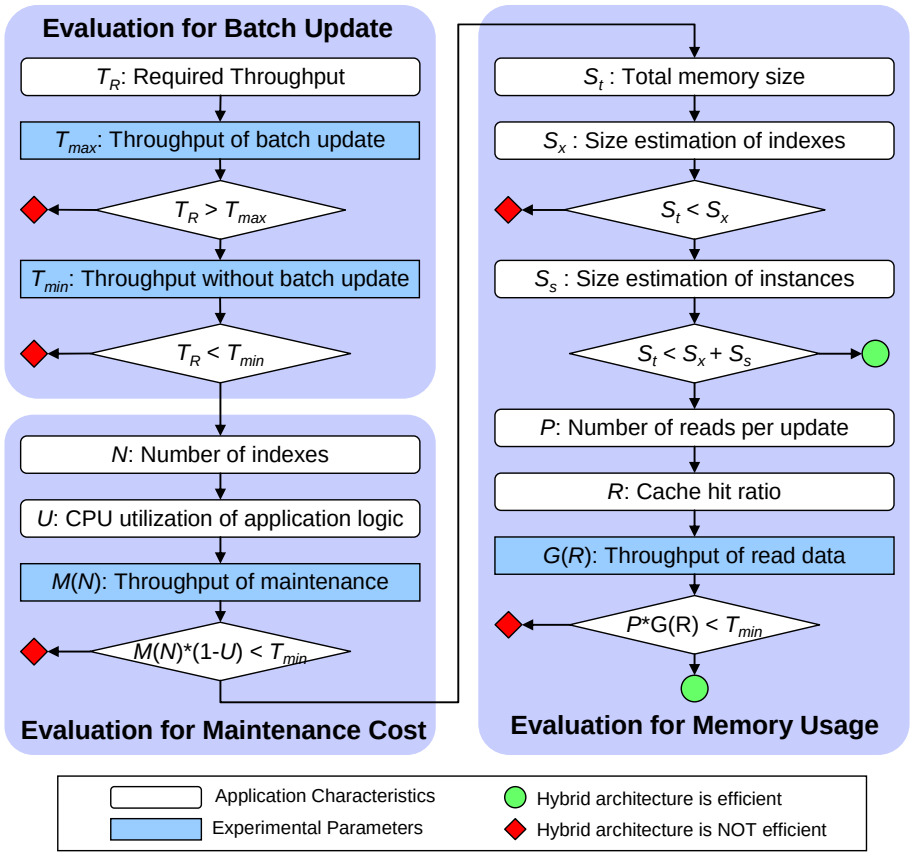

Fig. 14. Classification methodology for a given application to determine the efficiency of the hybrid architecture.

The first major step is the estimation of the efficiency of the batch update. The maximum throughput of batch updates T_{max} and the maximum throughput without batch updates T_{min} will be known from the experiments described in Subsection 4.3. T_{max} is considered as an upper bound of the throughput that can be achieved by using the hybrid architecture. If the throughput requirement of the given application T_R exceeds T_{max}, then we should look for other solutions.. If T_R is less than the throughput without batch update T_{min}, then the hybrid architecture is not necessary.

The second major step is the estimation of the maintenance costs for the result cache. The number of indexes N is estimated from the given set of SQL statements, and the throughput of maintenance for the indexes $M(N)$ can be acquired by testing.

Then we can compare $M(N)$ with T_{min} to estimate whether the hybrid architecture will be efficient when considering the maintenance costs of the generated indexes. In addition, the load of the application logic without maintenance should be considered based on the CPU utilization ratio U.

The third major step is the estimation of memory usage. First, we need to check whether all of the indexes can be stored in the available memory of the system. A rough estimate of the required memory for the indexes S_x can be made from the number of indexes and the number of instances (or instance variations, if it is larger then the number of instances). Then we can estimate how many instances can be cached by comparing the total of S_x and the instance cache size S_s with the total memory available for the application S_t.

A rough estimate of the cache hit ratio R can be made based on the memory size estimate. The throughput of reading data $G(R)$ acquired from the experiment is used, multiplied by the ratio of the number of reads per update P. Finally, we figure out whether or not the hybrid architecture would be efficient by comparing this throughput with T_{min}.

6 Concluding Remarks

We evaluated the hybrid architecture proposed in our previous work, and clarified that it is applicable for a broad range of application patterns. Though it requires enough memory to keep the entire result cache in memory, its efficiency is almost independent of access patterns. Thus the use of this hybrid architecture would be a major enhancement for applications that require processing many update and search queries in a consistent manner, including other event-driven applications, continuous queries, massive multi-agent systems, many personalized applications, and so on.

In addition, we proposed a methodology to estimate the efficiency of the hybrid architecture for a given application. For this methodology, we introduce two performance comparison models and determine three significant parameters. The performance comparison model enables to compare the advantages and the disadvantages of the hybrid architecture in the same standards. And the significant parameters indicate the application characteristics for which the hybrid architecture is efficient. The methodology makes it possible for developers to decide whether the hybrid architecture can be used for their practical applications.

The memory consumption of this architecture is the other problem to be addressed. We will address that problem as future work.

References

[1] L. Zeng, H. Lei, M. Dikun, H. Chang, and K. Bhaskaran. "Model-Driven Business Performance Management", *Proceedings of the 2005 IEEE International Conference on e-Business Engineering*, October, 2005.

[2] A. Gupta, and I.S. Mumick. "Maintenance of Materialized Views: Problems, Techniques, and Applications", *IEEE Data Engineering Bulletin*, Special Issue on Materialized Views and Data Warehousing, Vol. 18, No. 2, pp. 3-18, June, 1995.
[3] S. Ceri, and J. Widom. "Deriving Production Rules for Incremental View Maintenance", *Proceedings of the 1991 VLDB Conference*, pp. 577-589, September, 1991.
[4] D. Quass, A. Gupta, I.S. Mumick, and J. Widom. "Making views self-maintainable for data warehousing", *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems*, Miami Beach, Florida, Dec. 1996.
[5] K.A. Ross, D. Srivastava, and S. Sudarshan. "Materialized View Maintenance and Integrity Constraint Checking: Trading Space and Time", *Proceedings of the ACM SIGMOD Conference*, pp. 447-458, June, 1996.
[6] M. Hasan, B. Sugla, and R. Viswanathan. "A Conceptual Framework for Network Management Event Correlation and Filtering Systems", *Proceedings of the Sixth IFIP/IEEE International Symposium on Integrated Network Management*, pp. 233-246, May, 1999.
[7] S. Babu and J. Widom. "Continuous Queries over Data Streams", *SIGMOD Record*, Vol. 30, No. 3, September, 2001.