Dealing with Query Contention Issue in Real-time Data Warehouses by Dynamic Multi-level Caches*

Ziyu Lin* Dongqing Yang* Guojie Song‡† Tengjiao Wang*
*School of Electronics Engineering and Computer Science, Peking University, Beijing, China
‡State Key Laboratory of Machine Perception, Peking University, Beijing, China
cainiu@263.net;{dqyang, gjsong, tjwang}@pku.edu.cn

Abstract

The issue of query contention and scalability is the most difficult issue facing organizations deploying real-time data warehouse solutions. The contention between complex selects and continuous inserts tends to severely limit the scalability of the data warehouses. In this paper, we present a new method called dynamic multi-level caches, to effectively deal with the problem of query contention and scalability in real-time data warehouses. We differentiate between queries with various data freshness requirements, and use multi-level caches to satisfy these different requirements. Every query arriving at the system will be automatically redirected to the corresponding cache to access the required data, which means that the query loads are distributed to multi-level caches instead of becoming blocked in the only one cache due to the contention between query and update operations. Extensive experiments on several real datasets show that our method can effectively balance the query loads among multi-level caches and achieve desirable system performance.

1 Introduction

Traditional data warehouses are updated during downtime in a batch mode on a monthly, weekly, or at most nightly basis. When it comes to real-time data warehouses, however, the data from the source system needs to be continuously integrated into the data warehouses. This results in query contention issue involved with performing OLAP queries on changing data [6].

There are several ways to get around this problem, including simplifying and limiting real-time reporting, applying more database horsepower, external real-time data cache, just-in-time information merge from external data cache, reverse just-in-time data merge [1], real-time partitions [8], active partitions [11], and so on. One of the desirable methods among them is to use one external cache isolated from the data warehouse. The external data cache is updated continuously, and the data warehouse is updated in batch mode with ETL tools. All those queries involving real-time data or near real-time data are redirected to the cache, so as to avoid the query contention problem in the data warehouse. However, if many complex analytical reports are running on the real-time cache, it is possible for the cache to begin to exhibit the same query contention and scalability problems that a warehouse would exhibit [1].

Here we present a new method, called dynamic multi-level caches, to effectively deal with the problem of query contention and scalability in real-time data warehouses. Our method is based on the (single) external real-time data cache [1], and can be seen as a variation of it. We differentiate between queries with various data freshness requirements, and use multi-level caches to satisfy these different requirements. Every query arriving at the system will be automatically redirected to the corresponding cache to access the required data, which means that the query loads are distributed to multi-level caches, instead of becoming blocked in the only one cache because of the contention between query and update operations. The multi-level caches are updated in different length of cycles between real-time and 24 hours, and we also introduce a "wave-like updating algorithm" to achieve this target. According to the statistics information about the queries acquired by system, the multi-level caches will be dynamically adjusted according to the changing query loads so as to achieve better system performance. We conduct extensive experiments, and the results show that our method can effectively alleviate query contention issue and achieve better system performance and

*Supported by the Natural Science Foundation of China under Grant No. 60473051 and China HP Co. and Peking University joint project (Scalable, Real-Time and Active MPP based Data Warehouse for Telecommunication Industry).

†Corresponding author

0-7695-2983-6/07 $25.00 © 2007 IEEE
DOI 10.1109/CIT.2007.60

scalability than the single external data cache.

The remainder of this paper is organized as follows: Section 2 gives the detailed description of our approach. The experimental results are reported in Section 3, followed by the discussion of related work (in Section 4). Finally, we give the discussion and conclusion in Section 5.

2 Dynamic Multi-level Caches

In this part, we will first give the architecture of real-time data warehouse with multi-level caches, followed by the description of the wave-like updating against the caches, and then we will discuss how to answer a query with the data from the caches. Finally, we will present in detail the dynamic adjustment of the multi-level caches.

2.1 The architecture of real-time data warehouse with multi-level caches

Figure 1 shows the architecture of real-time data warehouse with multi-level caches. For this architecture, it is necessary to resolve the problems such as data modeling, data integrating and data merging.

Data modeling. In our method, no special data modeling is required in the data warehouse. The external data cache databases are generally modeled identically to the data warehouse, but typically contains only the tables that are real-time. The data modeling approach used here is dimension modeling (star schema), in which fact and dimension tables are used for representing data. All the caches use the same fact and dimension tables, but are updated in different cycles.

Data integrating. Here data integrating task is accomplished through two different ways, namely, batch loading and real-time loading. Batch loading is used to load data in batch mode into the data warehouse on a nightly basis. CDC (Change Data Capture) [3] technology is used here to continuously trickle-feed the changing data from the source system into the external data caches.

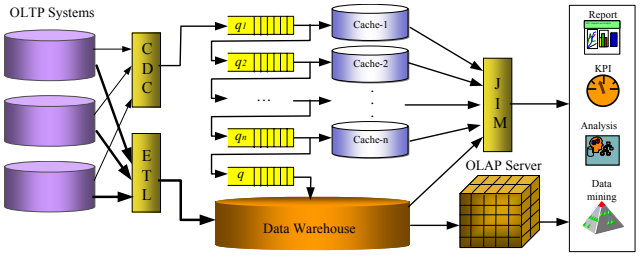

Figure 1. The architecture of real-time data warehouse with multi-level caches

Data merging. Many analyses may need both (near) real-time data in the external caches and historical data in the data ware house, which means that there must exist a mechanism to seamlessly combine these two types of data for the use of OLAP tools. Here this job is done by JIM (Just-in-time Information Merge) system [1].

2.2 Wave-like updating of the multi-level caches

Definition 1.*Version* *: Let D be a dataset. A version of D, denoted by $\mathcal{V}(D)$, is a snapshot of D at a specific instant t.*

Definition 2.*Freshness Level* *: Freshness level is used to represent the currency extent of D, which is denoted by $\mathcal{F}(D)$. For two versions of D, $\mathcal{V}_1(D)$ and $\mathcal{V}_2(D)$, if $\mathcal{F}(\mathcal{V}_2(D))> \mathcal{F}(\mathcal{V}_1(D))$, we say that $\mathcal{V}_2(D)$ is older than $\mathcal{V}_1(D)$. Also, for a query Q, $\mathcal{F}(Q)$ denotes the requirement of Q on the freshness level of data.*

For the datasets C_1, C_2,...,C_n in the multi-level caches, C_i contains all information that has been integrated into Level-i Cache. $\mathcal{F}(C_i)$ reflects the currency extent of C_i. Specially for C_1, which is updated in real-time, $\mathcal{F}(C_1)$ is always equal to 0, which means C_1 contains real-time data all the time. Also, there is a relationship between these caches, i.e. $\mathcal{F}(C_1)< \mathcal{F}(C_2)<...< \mathcal{F}(C_n)$.

Now we discuss the updating against the caches. In our method, every level of cache $C_i$is accompanied with a queue q_i, which contains all the data that is to be integrated into the cache C_i. The data in the queues are in fact logical change records (LCRs) [11], which will be dequeued from the queues and applied to the caches. Every level of cache is updated in different cycles. For example, C_1 is updated in real-time, C_2 every 5 minutes, C_3 every 30 minutes, C_4 every 1 hour, and so on. In this way, the data from the source system arrives at Level-1 Cache first, then Level-2 Cache, then Level-3 Cache, ... , and finally the last cache (Level-n Cache). This process is just like the movement of a wave, and therefore we call it "wave-like updating".

Figure 2 shows the algorithm for the updating against Level-1 Cache. q_1 contains all the continuously arrived data from the source system, which will, on one hand, be integrated into Level-1 Cache, and on the other hand go ahead into q_2 for the purpose of updating against Level-2 Cache.

Figure 3 shows the algorithm for the updating against Level-i Cache, where $2 \leq i \leq n$. A little bit different from the updating against Level-1 Cache, in which all the data in q_1 is directly and totally integrated into Level-1 Cache, the data in q_i ($2 \leq i \leq n$) has to be preprocessed before being integrated into Level-i Cache. The preprocessing typically involves the deleting of the older version of certain data so as to preserve the most recent data.

```
Input:  input data queue q1
        input data queue q2
        the dataset C1 in the Level-1 Cache
Output: the updated C1 and q2
1.  while ( q1 is not empty )
2.        d = q1.pop();
3.        integrate d into C1;
4.        q2.put(d);
5.  return C1,q2
```

Figure 2. The algorithm for the updating of Level-1 Cache

```
Input:  input data queue qi
        input data queue qi+1
        updating cycle Ti
        the dataset Ci in the Level-i Cache
Output: the updated Ci and qi+1
1.  while (Ti begins)
2.        while ( qi is not empty )
3.              d = qi.pop();
4.              put d into the set M;
5.              if ( dold ∈ M )
                /*dold is a version of d, and F(dold)>F(d)*/
6.                delete dold from M;
7.        for (each m ∈ M)
8.            integrate m into Ci;
9.            qi+1.put(m);
10.       return Ci,qi+1
```

Figure 3. The algorithm for the updating of Level-i Cache ($2 \leq i \leq n$)

2.3 Answering queries with the data from the multi-level caches

When a query arrives, it does not need to know where the real-time data locates and how to access it. In our method, the JIM system [1] will automatically get the real-time part of the required data from the multi-level caches and effectively combine it together with the historical part to satisfy the query's requirement. What a query needs to do is only to declare its requirement on the freshness level of data, according to which the system may decide which cache to be used to best serve the query.

For a query Q with freshness level requirement $\mathcal{F}(Q)$, in order to select from the multi-level caches an appropriate one to serve it, we must in the first place get the qualified version of D for Q, which is defined as follows:

Definition 3.*Qualified Version:* *Let D be a dataset, and $\mathcal{V}_1(D)$, $\mathcal{V}_2(D)$,..., $\mathcal{V}_n(D)$ are different versions of D, where $\mathcal{V}_i(D) \subseteq C_i$, and C_i is the dataset in the Level-i Cache. Suppose there is a query Q ,and D is the dataset involved in Q, we say that $\mathcal{V}_i(D)$ is a qualified version of D for Q, only if $\mathcal{F}(\mathcal{V}_i(D)) \leq \mathcal{F}(Q)$. A set containing all the qualified version of data for Q is called a "qualified version set" of Q, which is denoted by $\pi(Q)$. The version with the largest value of $\mathcal{F}$ in $\pi(Q)$, is called "default version" for Q, which is denoted by $\mathcal{V}^*(D)$.*

Example 1: Suppose $\mathcal{F}(Q) = 60$ min, $\mathcal{F}(\mathcal{V}_1(D)) = 0$ min, $\mathcal{F}(\mathcal{V}_2(D)) = 10$ min, $\mathcal{F}(\mathcal{V}_3(D)) = 30$ min, $\mathcal{F}(\mathcal{V}_4(D)) = 60$ min, $\mathcal{F}(\mathcal{V}_5(D)) = 4$ hours, and $\mathcal{F}(\mathcal{V}_6(D)) = 8$ hours. Then we will have $\pi(Q) = \{\mathcal{V}_1(D), \mathcal{V}_2(D), \mathcal{V}_3(D), \mathcal{V}_4(D)\}$ and $\mathcal{V}^*(D) = \mathcal{V}_4(D)$.

Theorem 1.*Let D be the dataset involved in a query Q, $\mathcal{V}_1(D)$, $\mathcal{V}_2(D)$,..., $\mathcal{V}_n(D)$ are different versions of D, where $\mathcal{V}_i(D) \subseteq C_i$, and C_i is the dataset in the Level-i Cache. If $\mathcal{V}_i(D) \in \pi(Q)$ ($0 \leq i \leq n$), there must exist $\mathcal{V}_m(D) \in \pi(Q)$, where $0 \leq m \leq i-1$.* □

It is easy to prove this theorem, so we here just give an example to explain it.

Example 2: Suppose $\mathcal{F}(\mathcal{V}_1(D)) = 0$ min, $\mathcal{F}(\mathcal{V}_2(D)) = 10$ min, $\mathcal{F}(\mathcal{V}_3(D)) = 30$ min, $\mathcal{F}(\mathcal{V}_4(D)) = 60$ min, $\mathcal{F}(\mathcal{V}_5(D)) = 4$ hours, and $\mathcal{F}(\mathcal{V}_6(D)) = 8$ hours. If $\mathcal{V}_3(D) \in \pi(Q)$, then there must exist $\mathcal{V}_2(D) \in \pi(Q)$ and $\mathcal{V}_1(D) \in \pi(Q)$.

After getting $\pi(Q)$, we are now faced with the task of selecting an appropriate version of D from $\pi(Q)$. The selected version is the one with the highest version priority, which is defined as follows:

Definition 4.*Version Priority:* *Let D be the dataset involved in a query Q, and $\mathcal{V}_1(D)$, $\mathcal{V}_2(D)$,..., $\mathcal{V}_n(D)$ are different versions of D, where $\mathcal{V}_i(D) \subseteq C_i$, and C_i is the dataset in the Level-i Cache. Version priority of $\mathcal{V}_i(D)$, denoted by $\mathcal{P}(\mathcal{V}_i(D))$, represents the appropriateness extent of $\mathcal{V}_i(D)$ to be used to answer Q.*

Initially, there is a relationship between different versions of D, i.e., $\mathcal{P}(\mathcal{V}_1(D)) < \mathcal{P}(\mathcal{V}_2(D)) < \ ... \ < \mathcal{P}(\mathcal{V}_n(D))$. However, $\mathcal{P}(\mathcal{V}_i(D))$ is also influenced by many factors. For example, if the Level-i Cache is undergoing updating or is overloaded, $\mathcal{P}(\mathcal{V}_i(D))$ will have a smaller value.

Figure 4 shows the algorithm for answering a query with the data from the multi-level caches. In Figure 4, line1 to line 6 are related to the process of specifying qualified versions of the data involved in a query. Line 3 and line 4 are based on Theorem 1, which helps to get $\pi(Q)$ quickly. Line 7 to line 12 show the process of specifying and adjusting the priority values of the qualified versions.

2.4 Dynamic adjustment of the multi-level caches

In our method, the system will dynamically adjust the multi-level caches according to the statistics information about the distribution of query loads in a day. In the meantime, although the number of online caches can change ac-

Input: a query Q
freshness level requirement $\mathcal{F}(Q)$
Output: the dataset D for Q

1. S= ϕ;
2. $k = n$; /*n is the number of caches*/
3. **while** ($\mathcal{F}(\mathcal{V}_i(D))$> $\mathcal{F}(Q)$)
4. $k = k + 1$;
5. **for** ($i = 1; i \leq k; i++$)
6. S = $S \cup \{\mathcal{V}_i(D)\}$;
7. **for** (each $\mathcal{V}_i(D) \in S$)
8. **if** (T_i begins and $i \neq 1$)
/*T_i is the updating cycle of C_i, and $\mathcal{V}_i(D)) \subseteq C_i$*/
9. $\mathcal{P}(\mathcal{V}_i(D))$=$\mathcal{P}(\mathcal{V}_i(D))$-1;
10. **for** (each $\mathcal{V}_i(D)) \in S$)
11. **if** ($Level - i$ $Cache$ is overloaded or offline)
/*$\mathcal{V}_i(D))$ is in $Level - i$ $Cache$*/
12. $\mathcal{P}(\mathcal{V}_i(D))$=$\mathcal{P}(\mathcal{V}_i(D))$-1;
13. D = $\mathcal{V}_i(D)$ with the largest $\mathcal{P}$ in S;
14. return D;

Figure 4. The algorithm for answering queries with the data from the multi-level caches

cording to the distribution of query loads, the number of online caches and offline caches together is determined by the number of predefined "reference updating cycle" in the system, which is defined as follows:

Definition 5.*Reference Updating Cycle:* *Reference updating cycles are predefined updating cycles that can be used by the caches. Every cache must select one from reference updating cycles as its updating cycle. A reference updating cycle is denoted by T^*.*

Reference updating cycles need to be defined according to the statistics information about the freshness requirement of data within the organization. For example, we may predefine 5 reference updating cycles: T_1^* = 0min, T_2^* =10min, T_3^* =30min, T_4^* =60min, T_5^* =4hours. Therefore, there will be 5 levels of caches in the system.

Meanwhile, making decision on which level of cache to be online and which level of cache to be offline, is based on the query loads distribution information. However, we must note that the same amount of queries may bring different burden to various level of caches. For example, 100 complex queries every minute may have little influence on the cache with T^* =8Hours, but will block the cache with T^* =0 (being updated in real-time). So we introduce "query load adjusting coefficient" to get the "adjusted query load", which is more useful and reasonable as the basis of cache adjusting.

Definition 6.*Query Load Adjusting Coefficient:* *Query Load Adjusting Coefficient λ of a cache C, is a predefined value to be used to reflect the extent of influence on the query execution time brought by the updating against C.*

The larger the value of λ is, the more the influence is on the query execution. $\lambda = 1$ means that updating against C has nearly no influence on the query operation upon C. The values of λ may not be specified precisely sometimes, then approximate values manually defined according to experience are also helpful if they reflect the influence level difference between caches. For example, we here define the λ of the above 5-level caches as follows:

T	0min	10min	30min	60min	4hours
λ	1.2	1.15	1.1	1.05	1

Table 1. Query load adjusting coefficient λ for different updating cycle T

Definition 7.*Adjusted Query Load:* *The adjusted query load for reference updating cycle T during certain time period t is defined as follows:*

$$\mathcal{L}(T) = \sum_{i=0}^{m} \lambda |D_i|$$

where D_i is the data involved in query Q_i, $|D_i|$ means the size of D_i, λ is the query load adjusting coefficient, and m is the amount of queries satisfying $\mathcal{F}(\mathcal{V}^*(D_i))$=$T$ during time period t.

Figure 5 shows the algorithm for the dynamic adjustment of the multi-level caches. In Figure 5, line 3 to line 7 show the process of shutting down those caches that are seldom accessed so as to save the system resource. Also, we need to move some query load from the overloaded cache to the other qualified cache. Line 8 to line 17 perform such load balancing job.

3 Empirical Study

In this section, we report the performance evaluation of our method. The algorithms are implemented with C++. All the experiments are conducted on 4*2.4GHz CPU (double core), 32G memory HP Proliant DL585 Server running Windows Server 2003 and Oracle 10g.

We use TPC benchmark TPC-H(http://www.tpc.org) to get the required datasets in our experiment. With the help of Streams Components provided by Oracle 10g, it is easy to capture the change data in the data source and send it to the destination queue, from which it is dequeued to be integrated into the data caches. We get the required query sets by the following steps. First, we define four reference updating cycles T_1^*=0, T_2^*=10min, T_3^*=30min, and T_4^*=60min. Also we here define four query sets QS_1, QS_2, QS_3 and QS_4. Second, we suppose that the freshness level requirements of the queries in the four query sets are equal to T_1^*, T_2^*, T_3^* and T_4^* respectively. Third, we generate a lot of queries for every query set.

Input: reference updating cycles array $T^*[m]$
/*m is the number of reference updating cycles*/
adjusted query load array $\mathcal{L}[m]$
/*$\mathcal{L}[i]$ is the adjusted query load in $T^*[i]$*/
the number of caches n
Output: operation array $P[n]$ for the multi-level caches
/*$P[i]$ =1 means let Level-i Cache be online*/

```
1.  for (i=0; i < m; i++)
2.      P[i] =1;
3.  for (i=0; i < m; i++)
4.      if((L[i]< δ_low) and (L[i − 1] +L[i]< δ_high))
5.        P[i] =0;
6.        L[i − 1] = L[i − 1] +L[i];
7.        L[i]=0;
8.  for (i=0; i < m; i++)
9.      if(L[i]> δ_high)
10.       k = i;
11.       stop = false;
12.       while (stop == false)
13.             if (L[k − 1]< δ_high)
14.               P[i] =1;
15.               stop = true;
16.             else
17.               k = k − 1;
18. return P[n];
```

Figure 5. The algorithm for cache updating

Experiment 1: In this experiment, we want to compare the query performance of our method with that of the traditional single cache method when under various amount of concurrent queries. Here we construct three query sets S_1, S_2 and S_3, each of which is composed of the queries from QS_1, QS_2, QS_3 and QS_4. The percent of the queries from QS_i to the overall queries in S_i is shown in Table 2. During the whole experiment, there are 50 updates every second occurring in the data source. Also on the side of the caches, there are a lot of concurrent queries running against them. We will change the amount of concurrent queries from 5 to 100. Figure 6 shows the test results about the query execution time under different conditions, in which, SC-S_1 represents the queries from S_1 running against a single cache, and MC-S_i reflects the queries from S_i running on the multi-level caches. From the test result, we can see that the multi-level caches method is more scalable than the traditional single cache method.

	QS_1	QS_2	QS_3	QS_4
S_1	80%	10%	10%	0
S_2	40%	20%	20%	10%
S_3	10%	30%	30%	30%

Table 2. Query sets composition

Experiment 2: In this experiment, we want to compare the query performance of our method with that of the traditional single cache method when under various updating frequency in the data source. The query sets are just the same as those used in Experiment 1, but we now fix the amount of concurrent queries at the value of 100, and change the frequency of updates against the data source from 10 updates every second to 50 updates every second so as to see the changing of query execution time. Figure 7 shows the test results, from which we can get that multi-level caches method can better deal with the query contention issue than the single cache method.

Experiment 3: In this experiment, we want to compare the confliction rate of our method with that of the traditional single cache method when under various updating frequency in the data source. We here define confliction rate r as c_1/c_2, where c_1 is number of queries conflicting with the updates, and c_2 is the total number of queries arriving at the caches. The query sets are also the same as those used in Experiment 1, and we now fix the amount of concurrent queries at the value of 100. We change the frequency of updates against the data source from 10 updates every second to 80 updates every second so as to see the changing of confliction rates. From the experimental results in Figure 8, we can observe that muli-level caches plays an important role in the reduction of confliction rate.

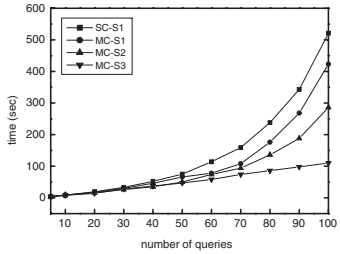

Figure 6. Query execution time under different conditions

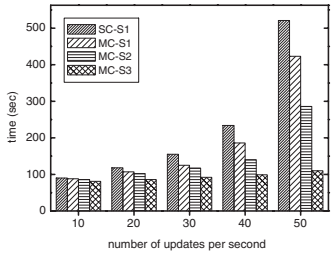

Figure 7. Query execution time under different updating frequency

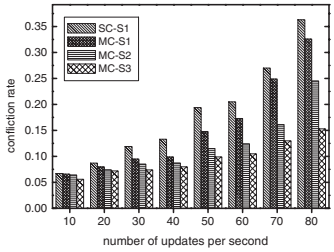

Figure 8. Confliction rates comparison

4 Related Work

To date, there has been many desirable methods proposed to enhance the query execution performance and scalability for the traditional data warehouses. Parallelism [9] can be used for major performance improvement in large data warehouses (DW) with performance and scalability challenges. Also, because too many accesses to large amount of detailed data in the data warehouse will undoubtedly deteriorate the problem of query contention and scalability, query cache [5, 10] and materialized views [7] may be used to answer queries without the need to access large amount of detailed data in the data warehouse.

However, the methods above for the traditional data warehouses are not sufficient when it comes to the real-time data warehouses. External real-time data cache [1] is a way to avoid the query contention in real-time data warehouses. The external data cache is updated continuously, and the data warehouse is updated in batch mode with ETL tools on a nightly basis. In [8], Kimball proposes "real-time partition" subject to special rules for update and query, which is a special partition that is physically and administratively separated from the conventional static data warehouse tables. Therefore, the static part of the data warehouse is only used for query instead of updating, which eliminates the query contention on the data warehouse. However, it is possible for both real-time cache and real-time partition to begin to exhibit the same query contention, and scalability problems that a warehouse would exhibit, if many complex analytical reports are run on them [1]. Active partitions [11] enable queries to work with constant data snapshots. Active partition is a separate partition in the data warehouse, and it is either offline or not visible to all user queries when updates are occurring. At a certain predefined interval, typically once every few minutes, the system renames the active partition so it may be merged with the data warehouse tables. However, it can not provide true real-time data.

There are also many other ways such as simplifying and limiting real-time reporting, applying more database horsepower, just-in-time information merge from external data cache, and reverse just-in-time data merge [1].

5 Discussion and Conclusion

In this paper, we have revisited the query contention and scalability issue. We proposed a method called dynamic multi-level caches to resolve such issue in the field of real-time data warehouses. With this method, we can take full advantage of the different freshness requirements of various queries and allocate the query loads into the corresponding cache so as to greatly avoid the occurrence of many queries being blocked in the only one cache. Extensive experiments show that our method can achieve desirable system performance in the real-time data warehouse environment.

In future, we will apply our theory in the field of mobile communication, so we plan to investigate the business requirement in the real world so as to define the reference updating cycles in a manner that may better satisfy the business requirements.

References

[1] J. Langseth. Real-Time Data Warehousing: Challenges and Solutions. *DSSResources.COM*, 2004.

[2] S. S. Conn. OLTP and OLAP data integration: a review of feasible implementation methods and architectures for real time data analysis. In: *SoutheastCon, 2005. Proceedings. IEEE.* pages 515-520, 2005.

[3] I. Ankorion. Change Data Capture-Efficient ETL for Real-Time BI. Article published in *DM Review Magazine*, January 2005 Issue.

[4] T. Thalhammer and M. Schrefl. Realizing active data warehouses with off-the-shelf database technology. *software-Practice & Experience, ACM*, 32(12), pages 1193-1222, 2002.

[5] A. N. Saharia and Y. M. Babad. Enhancing data warehouse performance through query caching. *ACM SIGMIS Database*. Vol:31(3), pages 43-63, Jun., 2000.

[6] H. Michael. Real Time Data Warehouse: The Next Stage in Data Warehouse Evolution. *DM Review*, 2003

[7] J. Goldstein and P. A. Larson. Optimizing queries using materialized views: A practical, scalable solution. In: *Proc. SIGMOD*, Vol 30(2), pages 331 - 342, 2001.

[8] R. Kimball. Real-time Partitions. In *Intelligent Enterprise*. February, 2002. http://www.intelligententerprise.com/020201.

[9] P. Furtado. Experimental evidence on partitioning in parallel data warehouses. In:*Proceedings of the 7th ACM international workshop Data warehousing and OLAP*, Nov. 2004, Washington, DC, USA, pages 23-30, 2004.

[10] J. Shim, P. Scheuermann and R. Vingralek. Dynamic Caching of Query Results for Decision Support Systems. In:*Proceedings of the 11th International Conference on Scientific on Scientific and Statistical Database Management*. pages 254-263, Jul., 1999.

[11] R. Gadodia. Right in Time. *Inteligent Enterprise.*7. pages 26-44. 2004.