4

Semantic Cache System

Munir Ahmad, Muhammad Abdul Qadir, Tariq Ali,
Muhammad Azeem Abbas and Muhammad Tanvir Afzal
Centre for Distributed and Semantic Computing,
Faculty of Computing,
Mohammad Ali Jinnah University Islamabad,
Pakistan

1. Introduction

One of the economical ways to develop a very large scale database is to distribute it among multiple server nodes. Main problem in these types of systems is retrieval of data within significant time; especially when network or server load is high. This task becomes more critical when data is to be retrieved from the database against frequent queries. Cache is used to increase the retrieval performance of mobile computing and distributed database systems. Whenever data is found locally from the cache it is termed as cache hit. Percentage of user posed queries that can be processed (partially or fully) locally from cache is called hit ratio. So, the cache system should be designed in a way that will increase the hit ratio. Improvement in hit ratio ensures efficient reuse of stored data. Due to efficient reuse of stored data, lesser amount of data is required to be retrieved from remote location.

Typically, cache is organized in three ways, as page, tuple, and semantic. Unit of transfer in page cache is *page* (multiple tuples) and in tuple cache is a *tuple*. In page cache irrelevant tuples may be retrieved for a user query. Retrieval of irrelevant data causes to wastage of valuable resources. Tuple cache overcomes this problem by stopping the retrieval of irrelevant tuples. Major problem (retrieving portion of tuple instead of complete tuple) still exists which cannot be handled using both (page & tuple) of these caching models. These caching schemes (page & tuple) are not able to identify whether the answer is contained in the cache in case of query not fully matched (partial matched). In page and tuple cache schemes all of the data is retrieved from remote site even in the presence of partial data on cache. In simple words portion of page or portion of tuple cannot be reused in the presences of page or tuple caching. As a result hit ratio is not up to that extent to which it should be.

To answer the queries partially from local site concept of semantic cache is introduced. Semantic cache has an ability to increase the hit ratio up to possible extent. Semantic cache provides better performance than page and tuple cache and this system is referred as semantic caching. Semantic caching provides the significance workload reduction in distributed systems, especially in mobile computing as well as improves the performance.

In semantic caching the semantic descriptions of processed query with actual contents are stored. Next posed query is processed for stored semantic descriptions of data in the cache

and posed query is divided into probe (portion available at cache) and remainder (portion that is not available at cache and have to retrieved from cache) queries. In this context we can say that there are two major activities query processing and cache management are involved in semantic caching. So, efficiency of the semantic caching will depends on these two major activities (query processing and cache management). Query processing is the process which returns the result against user posed query. In semantic cache query processing is done by dividing the user query into probe and remainder queries on the base of query matching. In fact, efficiency of query processing will depend on the efficiency of division process (query trimming) of user query into sub queries (probe and remainder) as well as on retrieval time against both probe and remainder queries. However, efficiency of query trimming will depends on the semantic indexing. In fact, semantic indexing at cache is a major activity of cache management. In this context we can say that efficient semantic caching system demands efficient query processing and indexing scheme. In this chapter we have discussed the state of art query processing techniques and semantic indexing scheme. We also have presented a query processing scheme sCacheQP and its complete working. Working of sCacheQP is explained with the help of case study.

2. Definitions

This section presents some definitions that are used in rest of the chapter.

Definition 1: Given a user query $Qu = \pi_A \sigma_P(R)$; where '*A*' is set of attributes required by user, '*P*' is a condition (WHERE clause) of the user query, and '*R*' is a relation. **User Query's Semantics** will be 3-tupple <Q_S, Q_F, Q_W> where Q_S is a set of required attributes, Q_F is a relation, and Q_W is a condition.

Definition 2: Given a database $D = \{Ri\}$ and its attributes set $A = UA_{Ri}$, $1 \leq i \leq n$, **Semantic Enabled Schema** will be 6-tupple <*D, R, A,* S_A*, P, C*> where '*D*' is the name of database, '*R*' is name of relation, '*A*' is a set of attributes, is a status of attributes, '*P*' is predicate (condition) on which data has been retrieved and cached, and '*C*' is the refrence of contents.

Definition 3: Given a user query $Qu = \pi_A \sigma_P(R)$ and Q_C having semantics <*D, R, A,* S_A*, P, C*>; **Data Set Du** and **Dc** will be retrieved rows in the result of execution of *Qu and* Q_C respectively.

Definition 4: Given a user query Q_U and cached query Q_C with semantics < Q_S, Q_F, Q_W> and <*D, R, A,* S_A*, P, C*> respectively; **Probe Query (pq)** will be **Du** ∩ **Dc**.

Definition 5: Given a user query Q_U and cached query Q_C with semantics < Q_S, Q_F, Q_W> and <*D, R, A,* S_A*, P, C*> respectively; **Remainder Query (rq)** will be (**Du** - **Dc**).

Definition 6: Given a user query Q_U and cached query Q_C with semantics < Q_S, Q_F, Q_W> and <*D, R, A,* S_A*, P, C*> respectively; **Query Matching** is a process in which user query's semantics <Q_S, Q_F, Q_W> are matched with semantic enabled schema <*D, R, A,* S_A*, P, C*>. It is further divided into two processes; attribute and predicate matching.

Definition 7: Given a user query Q_U and cached query Q_C with semantics < Q_S, Q_F, Q_W> and <*D, R, A,* S_A*, P, C*> respectively; **Attribute Matching** is a process in which user query's attributes <Q_S> are matched with attributes indexed by semantic enabled schema <*A*>. Common attributes (C_A) $Q_S \cap A$ and difference attributes (D_A) $Q_S - A$ are calculated for probe and remainder queries respectively.

Definition 8: Given a user query Q_U and cached query Q_C with semantics < Q_S, Q_F, Q_W> and <*D, R, A, S_A, P, C*> respectively; **Predicate Matching** is a process in which user query's condition <Q_W> is matched with condition indexed by semantic enabled schema <*P*>.

Definition 9: Given a user query Q_U and cached query Q_C with semantics < Q_S, Q_F, Q_W, P_A> and <*D, R, A, S_A, P, C*>; **Query Trimming** is a process in which user query (Q_U) is divided into probe and remainder query.

Definition 10: Given a user predicate Q_W and cached predicate *P*; **Predicate Implication** ($Q_W \rightarrow P$) holds if and only if Q_W completely overlapped with *P*.

Definition 11: Given a user predicate Q_W and cached predicate *P*; **Predicate Satisfiablity** holds if and only if Q_W partially overlapped with *P*.

Definition 12: Given a user predicate Q_W and cached predicate *P*; **Predicate Unsatisfiablity** holds if and only if Q_W is not overlapped with *P*.

Definition 13: Given a user query Q_U and cached query Q_C with semantics < Q_S, Q_F, Q_W, P_A> and <*D, R, A, S_A, P, C*>; **Query Implications** holds if and only if $Q_S \subseteq$ A as well as predicate implication holds.

Definition 14: Given a user query Q_U and cached query Q_C with semantics < Q_S, Q_F, Q_W, P_A> and <*D, R, A, S_A, P, C*>; **Query Satisfiability** holds if and only if $Q_S \cap A \mathrel{!}= \Phi$ as well as predicate implication/satisfiability holds.

Definition 15: Given a user query Q_U and cached query Q_C with semantics < Q_S, Q_F, Q_W, P_A> and <*D, R, A, S_A, P, C*>; **Query Unsatisfiaility** holds either $Q_S \cap A = \Phi$ or predicate implication/satisfiability does not holds.

3. State of the art

This section presents the brief related work to semantic caching in the context of semantic indexing and query (*SELECT and PROJECT*) processing for relational databases. For detail work survey done by Ahmad et al. (Ahmad et al, 2008) can be considered and for aggregate queries is discussed by Cai et al (Cai et al, 2005). Semantic caching is extensively studied by researchers in both relational and XML databases. In fact, query processing and cache management are two main areas of semantic cache system. In this section we have described the state of the art query processing as well as semantic indexing schemes.

Results of already executed queries are cached to generate more efficient query plans for centralized systems (Roussopoulosl, 1991). Some strategies are defined for cache to prefecth the data by using semantics (Kang et al, 2006). Query refinement technique is introduced to enhance the response time in multimedia databases (Chakrabarti et al, 2000). A predicate based scheme for cache is presented by Keller et al. for client server applications (Keller and Basu, 1996). A scheme with the name of Intelligent Cache Management (Chen et al, 1994) and its extensions are introduced (Ahmed et al, 2005, Altinel et al, 2003, Bashir and Qadir, 2007) to reduce the overhead of page and tuple cache. To answer the queries partially from local site; concept of semantic cache on the base of implication (Sun et al, 1989) and the base of description logic (Ali et al, 2010, Ali and Qadir, 2010) is introduced to increase the hit ratio up to possible extent (Bashir and Qadir 2007, Ahmad et al, 2008a). Idea of amending query is introduced to increase the hit

ratio (Ren et al, 2003), graph based query trimming to enhance efficiency (Abbas et al, 2010) and 112 rules (Bashir and Qadir, 2007b) are defined to reduce query processing time by efficient query matching. Rules (112) are only applicable for simple queries (excluding the disjunct or conjunct operator). Jonsson and colleagues presents query matching scheme by using predicate of query (Jonsson et al, 2006). This scheme is not able to handle the SELECT CLAUSE of SQL queries. Query matching algorithm that reduces query processing time in the domain of relational database is also studied in our previous work (Ahmad et al, 2008a, Ahmad et al, 2008, Ahmad et al, 2009]. Work in semantic cache in other domains like web [Lee et al, 1995, Luo et al, 2003] and XML (Chen et al, 2002, Sanaullah et al, 2008) also studied in literature. Importance of semantic cache and disadvantages of page and tuple cache is presented [Ren et al, 2003, Dar et al, 1996] by providing comparisons of semantic cache with page and tuple cache.

There are different structures used to index the semantic description like; flat structure (Dar et al, 1996), 3-level hierarchal (Sumalatha et al, 2007a, 2007b, 2007c) segments (Ren et al, 2003), and 4-HiSIS (Bashir and Qadir, 2007). When semantics of queries are store in a flat structure (Dar et al, 1996) the query matching process is very expensive (time consuming) (Godfrey et al, 1997, Ahmad et al, 2008, Ahmad et al, 2009). Cache is divided into segments (Ren et al, 2003) and chunks (Deshpande et al, 1998) to reduce the cost. Runtime complexity and caching efficiency is improved by division of cache into segments and chunks. List of chunks is build on the base of previous queries and then this list of chunks is used to split the user posed (new) queries into two portions; one answered locally from cache and the second computed remotely (Deshpande et al, 1998). 4-level hierarchal semantic indexing scheme (*4-HiSIS*) is introduced to accelerate the semantic matching (Bashir and Qadir, 2006). In 4-HiSIS; semantic matching accomplished in four steps. At first; database name is matched. After successful matching of the database name; the relation name is matched in the second step. At third, attributes are matched at successful matching of relation match. In the final step predicate matching is performed on the based of successful matching of first three steps. There is a limitation of 4-HiSIS in the context of incompleteness; because there is no refrence of actual contents of cache is stored in 4-HiSIS. This limitation is overcomed by the graph based semantic indexing scheme (Ahmad et al, 2010) by storing the refrence of actual contents. In graph based semantic indexing scheme the matching procedure is performed in five steps. At the state of art graph based indexing is most efficient semantic indexing scheme. It also have a limitation; it has no ability to process the " *Select* *" type and incorrect queries in cache system.

State of the art semantic cache system has limitation in both areas (query processing and cache management i.e semantic indexing schemes). In this chapter we have presented the new scheme for semantic cache query processing. We named this system as sCacheQP. sCacheQP has an ability to overcome the limitaion in the context of query processing which is the main area of the semantic cahce system.

4. Semantic Cache Query Processing (sCacheQP)

This section presents the sCacheQP which is a complete procedure of query processing that overcomes the limitaions of the previous systems. Working and main driver algorithm of the sCacheQP is given in Figure 1 and Figure 2 respectively.

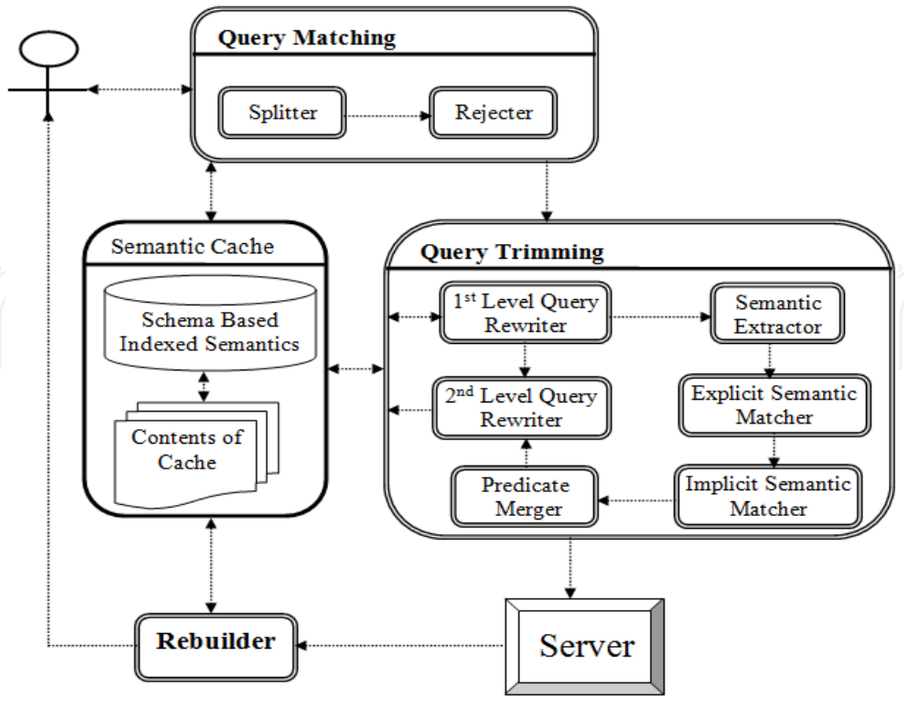


Fig. 1. Working of sCacheQP.

Algrithm1: sCacheQP
INPUT: Q_U (User query)
OUTPUT: F_R (Result against Q_U)
PROCEDURE:

1. Initialization:
   pq:=NULL
   rq1:=NULL
   rq2:=NULL
2. $P_{ORTIONS}$:= S_{PLIT_QUERY} (Q_U)
3. *Reject* := $C_{HECK\_REJECTION}$ ($P_{ORTIONS}$)
4. if (Reject =false) goto step 5 otherwise Reject query and goto step 15
5. C_A, D_A :=1st_Level_Query_Rewriter (Q_S)
6. If(D_A != empty) goto step 7 otherwise goto step 8
7. rq1 := $\pi_{DA}\sigma_{QP}(Q_R)$
8. If (C_A !=empty) goto step 9 otherwise goto step 14.
9. M_C, N_{MCC}, N_{MCU}, *Dvc, Dvu ,Opc, Opu, Coc, Cou*:=Semantic_Extraction(Q_W, S_W)
10. C_1, $N_{C1:=}$ ExplicitySemanticMatching(M_C, *Dvc, Dvu ,Opc, Opu*)
11. C_2, $N_{C2:=}$ ImplicitSemanticMatching(*Mc*, N_{MCC}, N_{MCU}, C_1, N_{C1})
12. pq, rq2:=PredicateMerging (C_2, N_{C2}, *Coc, Cou*)
13. aq :=$G_{EN\_AMEND\_QUERY}$()
14. F_R:= Rebuilder(pq,rq1,rq2)
15. Exit.

Fig. 2. Main Algorithm of sCacheQP.

4.1 Query matching

In semantic cache, user posed query is matched with the stored semantics on cache. In this process the decision is taken place either data is available at cache or not. Query matching process is accomplished in two sub process splitter and rejecter. Splitter will accept the user query Q_U from the user interface and splits the query on the base of three clauses (*SELECT, FROM, WHERE*) of the query. These three portions are called Q_S (*SELECT: projected attributes in the user query*), Q_F (*FROM*: Relation) and Q_W (*WHERE*: selected rows/tupples on specific condition); and send to the rejecter for initial level checking. Q_W will be empty if there is no condition on user posed query (Ahmad et al, 2009). Algorithm for splitting the user query is presented by Ahmad et al. (Ahmad et al, 2009) and given in figure 3.

Algorithm 2: $S_{PLIT\_}Q_{UERY}( )$
INPUT: Q_U (User query)
OUTPUT: Q_S, Q_W, Q_F
PROCEDURE:
$Q_S := S_{ELECT}\ C_{LAUSE}$
$Q_W := W_{HERE}\ C_{LAUSE}$
$Q_F := F_{ROM}\ C_{LAUSE}$
Return Q_S, Q_W, Q_F.

Fig. 3. Algorithm to Split Query.

Responsibility of rejecter is to checks the validity of user posed query by sending the list of selected attributes (Q_S), relation (Q_F) and predicate attributes (P_A) on the schema based indexing semantics. Predicate attribute is extracted by rejecter from Q_W and included in the list. If attributes list of QS, Q_F and P_A matched with stored schema then processing will be continued otherwise query will be rejected and processing will be stopped. Rejecter also builds Q_S in the case of '*' by retrieving all attributes from schema as a list if predicate attribute exist in schema (Ahmad et al, 2009). Algorithm to validate the user query is presented by Ahmad et al. (Ahmad et al, 2009) and given in figure 4.

Algorithm 4: $C_{HECK\_}R_{EJECTION}$ (Q_S, Q_F, P_A)
INPUT: Q_S, Q_F, P_A
OUTPUT: True/False(
PROCEDURE:
1. If all attributes of Q_S *present in schema*
If relation of Q_F *present in schema*
If P_A *is present in schema*
If(Qs='')*
return false and build Qs from schema
Else return true
Else return true
Else return true
Else return true

Fig. 4. Algorithm to Reject Incorrect Queries.

4.2 Query trimming

When it has been decided that data is available at cache then second step of sCacheQP is performed. In this step query is divided into two sub queries called probe and remainder queries called query trimming. This process accomplished in two stages. At first stage vertical partition takes place and the attributes that are not available (D_A) at cache directly sent to the server as rq1 (remainder query) with original predicate. We called it 1st level query rewriter (Ahmad et al, 2009) and its algorithm is given in figure 5. The query rq1 will be computed as follow:

$rq1 = \pi_{DA}\sigma_{QP}(Q_R)$

Algorithm 5: *1st_Level_Query_Rewritter* (Q_S)
INPUT: Q_S (*SELECT* Clause)
OUTPUT: rq1, C_A
PROCEDURE:
C_A:= *Attributes exist in both Qs and Schema*
D_A:= *Attributes exist in Qs but not in Schema*
Return C_A, D_A

Fig. 5. Algorithm for 1st Level Query Rewriter.

Rest of attributes; that are common in both user and cached query forwarded to the predicate processor which worked at second stage. Predicate processor consists of four sub modules; semantic extractor, Explicit Semantic Matcher, Implicit Semantic Matcher, and Predicate Merger. At this stage predicate is simplified by just separating the portions of it on the base of conjunct and disjunct operators. Then semantics of user's query predicate with respect to the cached predicate is extracted in the form of matching columns (*Mc*- similar in both user query predicate and cached predicate), non-matching columns of cache (*NMc*- columns in cached query that are not matched with user query) and non-matching columns of user query (*NMu*- columns in user query that are not matched with cached query). Some other information like; data value of cache predicate (D_{Vc}), data value of user predicate, (D_{Vu}), comparison operator in cache predicate (*Opc*), comparison operator in user predicate (*Opu*). Algorithm to extract the semantics of predicate is given below in figure 6.

Algorithm 6: Semantics Extractor
Input: Q_W ,S_W
Output:
{*Coc[n], Cou[n], MC[n],, NMCC[n],, NMCU[n], Opc[n], Opu[n], Dvc[n], Dvu[n]*}
Procedure:
1. M_C *[n]*:= List of Columns Present in both Q_W , S_W
2. N_{MCC}*[n]* := List of Columns Present in S_W but not in Q_W
3. N_{MCU} *[n]*:= List of columns present in Q_W but not in S_W
4. *Opc[n]* := operator set of S_W
5. *Opu[n]* := Operator present set of Q_W
6. *Dvc[n]* := Data values in S_W
7. *Dvu[n]* := Data values in Q_W
8. *Coc[n]*:= Connective Operators in *Sw*
9. *Cou[n]*:= Connective Operators in *Qw*
10. return *Coc[n], Cou[n], MC[n],, NMCC[n],, NMCU[n], Opc[n], Opu[n], Dvc[n], Dvu[n]*

Fig. 6. Algorithm to Extract Semantics from User Query.

***Algorithm 7**: ExplicitSemanticMatching*
Input:{Mc[n], O_{Pc}[n], O_{Pu}[n], D_{Vc}[n], D_{Vu}[n], C_C[n i], C_U[n]}
Output: {C_1[n], N_{C1}[n]}
Method:
Initilaize: C_1[n]:=Null, N_{C1}[n]:=Null, i:=0
Repeat from i to n
If(D_{Vc}[i] < D_{Vu}[i])
  If((O_{Pc}[i] ϵ{!=, >,>=}Λ O_{Pu}[i] ϵ{>, >=,=}))
    C_1[i]←C_C[i] O_{Pc}[i] D_{Vc}[i]
    N_{C1}[i]←Null
  else if(O_{Pu}[i] ϵ{<, <=, !=})
    C_1[i] ←C_C [i]O_{Pc}[i] D_{Vc}[i]
    N_{C1}[i]←(C_U[i] O_{Pu}[i]D_{Vu}[i])∧(C_C[i]Rev(O_{Pc}[i]) D_{Vc}[i])
  else
    C_1[i] ←Null
    N_{C1}[i]←(C_U[i] O_{Pu}[i] D_{Vu}[i])
else if(D_{Vc}[i] >D_{Vu}[i])
  if((O_{Pc}[i]ϵ{!=,<,<=}ΛO_{Pu}[i]ϵ{<,<=,=}))
    C_1[i] ←C_C[i] O_{Pc} [i]D_{Vc}[i]
    N_{C1}[i]←Null
  else if(((O_{Pc}[i] ϵ{!=, >, =, <=, <}ΛO_{Pu}[i]ϵ{>, >=, !=}))
    C_1[i] ←C_C[i] O_{Pc}[i] D_{Vc}[i]
    N_{C1}[i]←(C_U[i] O_{Pu}[i] D_{Vu}[i])∧(C_C[i]Rev(O_{Pc}[i]) D_{Vc}[i])
  else
    C_1[i] ←Null
    N_{C1}[i]←(C_U[i] O_{Pu}[i]D_{Vu}[i])
else if(D_{Vc}[i] = D_{Vu}[i])
  if(((O_{Pc}[i]ϵ{>=}Λ(O_{Pu}[i] ϵ{ >, =}))
    V (O_{Pc}[i] = O_{Pu}[i])V((O_{Pc}[i] ϵ{<=})Λ (O_{Pu}[i] ϵ{<,=}))
    V (O_{Pc}[i] ϵ{!=}Λ O_{Pu}[i]ϵ{<,>}))
    C_1 ←C_C O_{Pc} D_{Vc}
    N_{C1}←Null
  else if((O_{Pc}[i] ϵ{>, <=}Λ O_{Pu}[i] ϵ{ >=, !=})
    V (O_{Pc}[i]ϵ{<, >=}ΛO_{Pu}[i]ϵ{<=, !=})
    V (O_{Pc}[i]ϵ{!=,=}ΛO_{Pu}[i]ϵ{<=, >=}))
    C_1[i] ←C_C[i] O_{Pc} [i]D_{Vc}[i]
    N_{C1}[i]←(C_U[i] O_{Pu}[i] D_{Vu}[i])∧(C_C[i] Rev(O_{Pc}[i]) D_{Vc}[i])
  else
    C_1[i] ←Null
    N_{C1}[i]←(C_U[i] O_{Pu}[i] D_{Vu}[i])
else if((D_{Vc}[i]!= D_{Vu}[i]) Λ ((O_{Pc}[i] ϵ{=, !=}Λ O_{Pu}[i] ϵ{!=}))
    C_1[i] ←C_C[i] O_{Pc}[i] D_{Vc}[i]
    N_{C1}[i]←(C_U[i] O_{Pu}[i] D_{Vu}[i])∧(C_C[i] Rev(O_{Pc}[i]) D_{Vc}[i])
else
    C_1[i] ←Null
    N_{C1}←(C_U[i] O_{Pu}[i] D_{Vu}[i])

Fig. 7. Algorithm to Evaluate Predicate.

After extraction of semantics *Mc*, D_{Vc}, D_{Vu}, *Opc*, and *Opu* sent to the Explicit Semantic Matcher. Explicit Semantic Matcher trims the predicate into two portions; one for remainder (*C1*) and other for probe query (*NC1*). Explicit Semantic Matching algorithm is based on the boundary values as well as on the nature of comparison operators. There are 112 rules defined on the base of boundary values and basic comparison operators (<, <=, >, >=, = =, !=). Algorithm 7 given in figure 7 is used to match and trims the predicate. The output of predicate matching algorithm is predicate that is available at cache (C1) and predicate that is not available at cache (NC1). Working of the algorithm is explained above.

As we have discussed that Explicit Semantic Matching algorithm is based on the boundary value and basic comparison operator. On the base of boundary value and comparison operators; algorithm 7 will trim the predicate into probe and remainder queries.

Remember that predicate matching algorithm having better time complexity is an alternative of satisfiability/implication (Guo et al, 1996) used to help process query in the literature (Ren et al, 2003, Jonsson et al, 2006). Computed values C1, NC1 and NMu sent to the Implicit Semantic Matching algorithm to remove the additional information. Algorithm 8 is used to perform this job that is given below in figure 8.

Algorithm 8: *ImplicitSemanticMatching*
Input:
$Mc[n]$, $N_{MCC}[n]$, $N_{MCU}[n]$, $C_1[n]$, $N_{C1}[n]$
Output:
C_2, N_{C2}
Procedure:
Initiliaze C_2:=Null, N_{C2}:=Null,i:=Null
Repeat from i to n
1. If ($N_{MCC}[i]$ = null) and ($N_{MCU}[i]$ = null) then
   a. C_2:= $C_1[i]$
   b. $NC2$:= $N_{C1}[i]$
2. Else If ($N_{MCC}[i]$!=null) and ($N_{MCU}[i]$ = null) then
   a. C_2:= $(C_1[i])$ + $(N_{MCC}[i])$
   b. N_{C2} := $((C_1[i]+ R (N_{MCC}[i]))V(N_{C1}[i])$
3. Else If ($N_{MCC}[i]$ = null) and ($N_{MCU}[i]$!= null) then
   a. C_2:= $(C_1[i])$ + $(N_{MCU}[i])$
   b. N_{C2} := $N_{C1}[i]$ + $N_{MCU}[i]$
4. Else If ($N_{MCC}[i]$!= null) and ($N_{MCU}[i]$!= null) then
   a. C_2:= $(C_1[i])$ + $(N_{MCU}[i])$ + $(NMc[i])$
   b.N_{C2}:=$((C_1[i])+R(N_{MCC}[i])+N_{MCU}[i])V((N_{C1}[i])+(N_{MCU}[i]))$
5. Else If ($Mc[i]$ = = null) then
   a. C_2:= $(N_{MCC}[i])$ + $(N_{MCU}[i])$
   b. N_{C2} := $N_{MCU}[i]$ + $R(N_{MCC}[i])$

Fig. 8. Algorithm for Implicit Matching.

Algorithm 9: *PredicateMerging*
Input:
C_2, N_{C2}, Coc, Cou
Output:
Cached, N-Cached
Procedure:
If (Coc &Cou)ϵ **Λ**
Cached = ΛC_i
N-Cached= $V(C_i \Lambda NC_j)$ where 1<i,j<n and i!=j
If(Coc &Cou)ϵ**V**
Cached = VC_i where 1<i<n
N-Cached= VNC_i where 1<i<n
If(Coc ϵ**V** &Cou$\epsilon\Lambda$)
Cached = ΛC_i
N-Cached= $V(C_i \Lambda NC_j)$ where 1<i,j<n and i!=j
If(Cou ϵ**V** &Coc$\epsilon\Lambda$)
Cached = ΛC_i
N-Cached= $V((C_i \Lambda NC_j) V(U_i \Lambda R(U_j))$
wher1<i,j<n and i!=j

Fig. 9. Algorithm to Merge the Predicate.

Generated cached (C2) and non-cached (NC2) predicates by the Implicit Semantic Matcher are combined by predicate merger. Algorithm to merge the predicate is given in figure 9.

The computed predicates are then sent to the 2nd level query rewriter. Finally, probe and remainder queries will be computed by the 2nd level query rewriter as follow:

pq = *SELECT* C_A From Q_F *WHERE* Ccache
rq2 = *SELECT* C_A FROM Q_F *WHERE* N-Cached

pq will be executed locally ad rq will sent to the server. Then result of both will be sent to the rebuilder to combine the result.

4.3 Query rebuilding

Rebuilder receives the result form server (S_R) which is retrieved across remainder queries (rq1 and rq2) and results from cache (C_R) across probe query (pq) combines both as a final result F_R. Final result is viewed to the user and also updated in the cache contents if required.

5. Case study

To validate our proposed semantic indexing and query processing, we consider the case study of university.

Figure 10 presents the schema of university with two relations employee and students having 4 and 3 fields respectively.

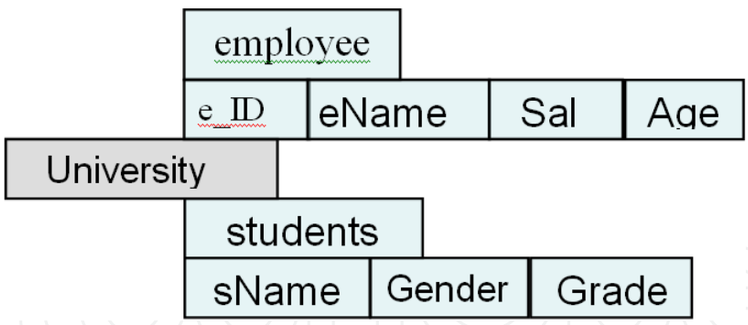


Fig. 10. Schema for University.

For the above given schema of university; there are 15 and 7 segments possible across employee and students relations respectively according to previous work (Ren et al., 2003). In simple words, we can say that there are 15 queries are possible against employee and similarly 7 for students as given in table 1.

In the above example there are 22 possible queries that make separate segments. So the formula to calculate possible segments across a single relation over 'n' attributes is "2n-1". Then add segments across each relation. As in example 15+7 =22: (24-1=15 &23-1=7). Hence, 22 segments are to be visited to check availability of data on cache in the worst case which increases the response time drastically.

S_R	S	S_A	S_P	S_C
employee	S1	e_ID	P1	1
	S2	eName	P1	2
	S3	Sal	P1	3
	S4	Age	P1	4
	S5	e_ID, eName	P3	5
	S6	e_ID, Sal	P4	6
	S7	e_ID, Age	P5	7
	S8	e_ID, eName Sal Age	P6	8
	S9	e_ID Sal Age	P7	9
	S10	e_ID, eName Age	P8	10
	S11	eName Sal	P9	11
	S12	eName Age	P10	12
	S13	Sal Age,eName	P11	13
	S14	e_ID, eName, Sal	P12	14
	S15	Sal, Age	P3	15
students	S16	sName	P21	16
	S17	Grade	P21	17
	S18	Gender	P21	18
	S19	Gender, Grade	P22	19
	S20	Gender, sName	P23	20
	S21	sName, Grade	P24	21
	S22	sName, Grade, Gender	P25	22

Table 1. Possible segments for Given Database.

Schema based hierarchal scheme reduces the number of comparisons to find out whether data is available at cache or not. Only 'n' comparisons are required to check availability of data on cache. Table 1 can be rearranged according to our proposed schema based semantic indexing scheme as in Table 2.

DB Name	**Table Names**	**Fields**	**Status**	**Condition**	**Content**
University	Employee	eName	True	P1	1
		Age	false	Null	Null
		Sal	True	P2	2
		e_ID	True	P3	1
	Students	Gender	false	Null	Null
		Grade	false	Null	Null
		sName	false	Null	Null

Table 2. Schema Based Indexing.

Table 2 represents structure of schema based semantic indexing instead of actual contents. There is only need to compare/match 4 and 3 fields instead of 15 and 7 segments respectively according to previous work. Also it has the ability to reject invalid queries at initial level instead of further processing.

For detailed discussion and simplicity, we consider only employee table of university database. Let us consider there is employee table on server with 4 fields defined in university schema in Figure 10. Employee table on server is given in table 3 below.

e_ID	eName	Age	Sal
110	Asad	20	25000
111	Ali	22	22000
112	Kashif	25	25000
113	Abid	30	15000
114	Adeel	31	42000
115	Komal	37	17000
116	Mahreen	39	30450
117	Tabinda	39	28850
118	Yaseen	40	24450
119	Anees	45	30000
120	Komal	50	30000

Table 3. Employee Table on Database.

Now we divide our case study into five cases in such a way that one can easily understand our contribution and novelty of our approach. For simplicity, each of five cases is discussed standalone and not linked with other. Each case should be considered separately. We have considered that cache is managed from initial for each case.

Case-I: In this case we will take an example that covers the query rejection at initial level.

Let us consider that user has already posed the following query and result has been stored in cache.

SELECT * *FROM* employee *WHERE* age>30

Data on cache will be as given in table 4.

e_ID	eName	Age	Sal
114	Adeel	31	42000
115	Komal	37	17000
116	Mahreen	39	30450
117	Tabinda	39	28850
118	Yaseen	40	24450
119	Anees	45	30000
120	Komal	50	30000

Table 4. Contents on cache in case-I.

Now let us consider user is going to pose following three queries.

1.SELECT eName, Age *FROM* emMloyee *WHERE* age>30
(relation is incorrect)
2.SELECT eName, Age *FROM* employee *WHERE* gpa>3.0
(predicate attribute is incorrect)
3.SELECT ename, rollno *FROM* employee *WHERE* age>30
(Projected attribute is incorrect)

All of three queries should be rejected at initial level; but according to all of previous work query will be posted on server due to unavailability of data on cache.

It is a beauty of our proposed schema based indexing scheme that all of three queries will be rejected and query processing time will be saved. According to our proposed semantic caching architecture list of projected attributes (eName, Age in first query), relation (emMloyee in first query) and predicate attribute is checked from schema based indexing scheme; and query will be rejected due to unavailability of "emMloyee" relation in schema. Similarly, query 2 and 3 will be rejected due to unavailability of "gpa" and "rollno" in employee table respectively and there will be no probe and remainder.

By rejecting query at initial stage; query processing can be saved. In this context our proposed semantic caching scheme has better performance than previous.

Case-II: In this case we will take an example that covers the handling of queries having * in SELECT CLAUSE.

Let us consider that user has already posed the following query and result has been stored in cache.

SELECT eName, Age *FROM* employee *WHERE* age>30

Data for above query will be retrieved and stored on cache will be as given in table 5.

e_ID	eName	Age
114	Adeel	31
115	Komal	37
116	Mahreen	39
117	Tabinda	39
118	Yaseen	40
119	Anees	45
120	Komal	50

Table 5. Contents on cache in case-II.

Note that e_ID is not required but retrieved. It is due to the requirement of key-contained (Ren et al., 2003) contents. Now let us assume that user has posed the following query.

SELECT * *FROM* employee *WHERE* age>30

Now all of the fields of employee required; but according to previous work common set will be calculated (intersection of cached attributes and user's query attributes). There is no way defined to calculate the common set of '*' and some attributes. Here we can say that all of the cached attributes for employee are required, but how can it be decided which of the attributes are not in cache and should retrieved from server.

Here again we need schema at cache (first we need schema for zero level query rejection). If schema is available at cache then SELECT CLAUSE with '*' can be handled easily. By this hit ratio is improved. Splitter splits the query and sent it to the rejecter. Rejecter checks the list of fields with relation and predicate attribute from schema based indexing semantics. Query will not be rejected due to availability of all member of list at schema. Common and difference set of attributes will be computed and sent to the 1st level query matcher. i.e. C_A and D_A will be computed. Remainder query (rq1) with difference attributes (here is only one difference attribute that is 'Sal') will be generated by 1st level query matcher like below.

rq1 = *SELECT* Sal *FROM* employee *WHERE* Age>30

Common attributes (e_ID, Age, eName) will be sent to the Query Generator (QG). Query Generator will generate probe and remainder query on the base of predicate matching. Conditioned attribute (Age) is already retrieved; so there is no need of amending query in this case.

First of semantics of predicate will be computed by semantic extractor as follow.

M_C = Age
NM_C = NULL
NM_U = NULL

After computation of predicate semantics, predicate for probe and remainder query will be computed by using predicate matching algorithm (actually 112 rules are used here). At first, main class of algorithm is selected, here data value of user (DVu=30) is equal to the data value of cache (DVc = 30). So, class 3 of predicate matching will be selected then priority of relational operator will be computed. Relational operator in both queries is '>'; it is low priority (defined in previous work; (Bashir and Qadir, 2007a)) operator. So, following porton of the algorithm will be executed. given below.

If($(O_{Pc} \epsilon \{!=, >, >=\} \Lambda\ O_{Pu} \epsilon \{>, >=, =\})$)
$C_1 \leftarrow C_C\ O_{Pc}\ D_{Vc}$
$N_{C1} \leftarrow$ Null

Here, Cc is Age, operator is '>' and DVc is 30.

So predicate for probe and remainder will be computed we say it C1 (for cached) and NC1 (for non-cached).

C_1 =Age>30
N_{C1} = Null

Due to simple predicate rules defined for complex queries will not be applied. Finally subtraction algorithm will be applied to generate final predicate for probe and remainder queries. As it is computed that NMc and NMu both are Null. So, first case of subtraction algorithm will be applied.

1. **If** (*NMc* = null) and (*NMu* = null)
then
a. $\mathbf{C}_2$:= *C1*
b. $\mathbf{N}_{C2}$:= *NC1*

So, there will be no change in predicate of probe and remainder query. Then, probe query (pq) and second remainder query (rq2) will be generated as below.

pq = SELECT eName, Age FROM employee WHERE Age>30
rq2 = Null

In the last step result of rq1, pq and rq2 is combined by rebuilder.

Case-III: In this case generation of amending query is elaborated.

Let us consider that user has already posed the following query and result has been stored in cache.

SELECT eName, Sal *FROM* employee *WHERE* age>30

Data for above query will be retrieved and stored on cache will be as given in table 6.

E_ID	eName	Sal
114	Adeel	42000
115	Komal	17000
116	Mahreen	30450
117	Tabinda	28850
118	Yaseen	24450
119	Anees	30000
120	Komal	30000

Table 6. Contents on cache in case-III.

Note that e_ID is not required but retrieved. It is due to the requirement of key-contained (Ren et al., 2003) contents. Now let us assume that user has posed the following query.

SELECT eName, Sal *FROM* employee *WHERE* age>35

Here, generation of amending query is discussed. Remaining procedure will be same as discussed in case-II.

Note that data across eName and Sal is present on cache, but predicate attribute (Age) is not on cache. Now some one cannot select the data from cache due to absence of predicate attribute; because some one cannot decide which of the data satisfy the selection criteria (Age>35). To solve this problem, another query called amending query (Ren et al., 2003) to retrieve primary attribute from server on user select criteria as below.

aq = *SELECT* e_ID *FROM* employee *WHERE* Age>35

Then retrieved primary keys will be mapped with keys on cache and data will be presented to user. By this hit ratio is increased.

Case-IV: In this case efficient predicate matching to improve hit ratio by using subtraction algorithm is elaborated with example.

Let us consider that user has already posed the following query and result has been stored in cache.

SELECT eName, Age *FROM* employee *WHERE* Age>30

Data for above query will be retrieved and stored on cache will be as given in table 7.

e_ID	eName	Age
114	Adeel	31
115	Komal	37
116	Mahreen	39
117	Tabinda	39
118	Yaseen	40
119	Anees	45
120	Komal	50

Table 7. Contents on cache in case-IV.

Note that e_ID is not required but retrieved. It is due to the requirement of key-contained (Ren et al., 2003) contents. Now let us assume that user has posed the following query.

SELECT eName, Age *FROM* employee
WHERE eName = 'Komal'

Now all of the required fields are matched with cached query. Splitter splits the query and sent it to the rejecter. Rejecter checks the list of fields with relation and predicate attribute from schema based indexing semantics. Query will not be rejected due to availability of all member of list at schema. Common and difference set of attributes will be computed and sent to the 1st level query matcher. i.e. C_A and D_A will be computed. Remainder query (rq1) will be null due to empty set of difference attributes (All required attributes are exist on cache). So, remainder query by 1st level query matcher will be like below.

rq1 = *Null*

Common attributes (Age, eName) will be sent to the Query Generator (QG). Query Generator will generate probe and remainder query on the base of predicate matching. Conditioned attribute (Age) is already retrieved; so there is no need of amending query in this case.

First of semantics of predicate will be computed by semantic extractor as follow.

M_C = Null
NM_C = Age
NM_U = eName

After computation of predicate semantics, predicate for probe and remainder query will be computed by using predicate matching algorithm (actually 112 rules are used here). Probe and remainder query will be on the base of these rules like:

CC-QM ← *Null*
CNC-QM ← *Null*

i.e.

C1 =Null
NC1 = Null

Due to simple predicate rules defined for complex queries will not be applied. Finally subtraction algorithm will be applied to generate final predicate for probe and remainder queries. As it is computed, that NMc and NMu both are not null. So, fourth case of subtraction algorithm will be applied.

4. **Else If** (*NMc* != null) and (*NMu* != null) **then**
a. $\mathbf{C}_2$:= (C_1) + (*NMu*) + (*NMc*)
b. $\mathbf{N}_{C2}$:= ((C_1) + R (*NMc)*+*NMu*)V((NC1)+(NMu))

So, predicate for probe and remainder query will be like below.

$\mathbf{C}_2$:= (Age>30) and (eName = 'Komal')
$\mathbf{N}_{C2}$:= (Age<=30) and (eName = 'Komal')

Then, probe query (pq) and second remainder query (rq2) will be generated as below.

```
pq = SELECT eName, Age FROM employee
    WHERE (Age>30) and (eName = 'Komal')
rq2 = SELECT eName, Age FROM employee
    WHERE (Age<=30) and (eName = 'Komal')
```

In the last step result of rq1, pq and rq2 is combined by rebuilder.

6. Conclusion

Caching proved very helpful to reduce the data access latency for distributed and large scale database systems by storing the data against already executed queries. Main problem in caching is to identify the overlapping of required data with stored data. Page and tuple cache are not able to identify the partial overlapping. Semantic cache is capable to answer the

overlapped (partially & fully) queries locally. The major challenges of semantic caching are efficient query processing and cache management. For efficient query processing we have proposed and demonstrated the working of sCacheQP system. We have provided complete working and algorithms of sCacheQP. Case study is given to elaborate the sCacheQP. In future, we have a plan to implement the system for data mining and data warehousing.

7. References

Abbas, M.A., Qadir, M.A., Ahmad, M., Ali, T., Sajid, N.A, (2011) "Graph Based Query Trimming of Conjunctive Queries in Semantic Caching", *IEEE International Conference on Emerging Technologies (ICET 2011)*, Islamabad, Pakistan, September 5-6, 2011.

Ali, T., Qadir, M.A., Ahmad, M. (2010) "Translation of relational queries into Description Logic for semantic cache query processing" *Information and Emerging Technologies (ICIET) 2010, Karachi Pakistan, 14-16 June 2010*

Ali, T., Qadir, M.A., (2010) "DL based Subsumption Analysis for Relational Semantic Cache Query Processing and Management"10th International Conference on Knowledge Management and Knowledge Technologies, Messe Congress Graz, Austria. 1–3 September 2010

Ahmad, M., Asghar, A., Qadir, M.A., Ali, T. (2010) "Graph Based Query Trimming Algorithm for Relational Data Semantic Cache", *The International Conference on Management of Emergent Digital EcoSystem*, MEDES10, Bangkok, Thailand, October 2010.

Ahmad, M., Qadir, M.A., Razzaque, A., and Sanaullah, M. (2008a), "Efficient Query Processing over Semantic Cache". *Intelligent Systems and Agents, ISA 2008,* indexed by IADIS digital library (www.iadis.net/dl). Held within IADIS Multi Conference on Computer Science and Information Systems (MCCSIS 2008), Amsterdam, Netherland. 22-27 July 2008

Ahmad, M., Qadir, M.A., and Sanaullah, M. (2008b) , "Query Processing over Relational Databases with Semantic Cache: A Survey". *12th IEEE International Multitopic Conference, INMIC 2008, IEEE, Karachi, Pakistan, December 2008.*

Ahmad, M., Qadir, M.A., and Sanaullah, M. (2009) "An Efficient Query Matching Algorithm for Relational Data Semantic Cache". *2nd IEEE conference on computer, control and communication, IC409, 2009.*

Ahmed, M.U, Zaheer, R.A, and Qadir, M.A., (2005). "Intelligent cache management for data grid"; *In Proceedings of the Australasian Workshop on Grid Computing and E-Research,* New South Wales, Australia, 2005.

Altınel, M., Bornhövd, C., Krishnamurthy, C., Mohan, C., Pirahesh, H., and Reinwald, B., (2003)., "Cache Tables: Paving the Way for an Adaptive Database Cache", *Proceedings of the 29th VLDB Conference*, VLDB Endowment, Berlin, Germany, pp. 718-729.

Bashir, M.F and Qadir, M.A., (2006). "HiSIS: 4–Level Hierarchical Semantic Indexing for Efficient Content Matching over Semantic Cache". *INMIC, IEEE,* Islamabad, Pakistan, pp. 211-214.

Bashir, M.F and Qadir, M.A., (2007). "ProQ – Query Processing Over Semantic Cache For Data Grid", *Center for Distributed and Semantic Computing,* Mohammad Ali Jinnah University, Islamabad, Pakistan 2007.

Bashir, M.F., Zaheer, R.A., Shams, Z.M. and Qadir, M.A., (2007). "SCAM: Semantic Caching Architecture for Efficient Content Matching over Data Grid". *AWIC,S*pringer Heidelberg, Berlin, 2007. pp. 41-46.

Chakrabarti, K., Porkaew. K., and Mehrotra, S., (2000). "Efficient Query Refinement in Multimedia Databases", *16th International conference on Data Engineering*, IEEE, 2000.

Cai, J., Jia, Y., Yang, S., and Zou, P., (2005) "A Method of Aggregate Query Matching in Semantic Cache for Massive Database Applications". *Springer-Verlag,* Berlin Heidelberg 2005, pp. 435-442.

Chen, C.M. and Roussopoulos, N., (1994). "The Implementation and Performance Evaluation of the ADMS Query Optimizer: Integrating Query Result Caching and Matching," Proc. Int'l Conf. Extending Database Technology, pp. 323-336.

Chen, L., Rundesteiner, E.A., Wang, S., (2002). "XCache -A Semantic Caching System for XML Queries". *In Proceedings of the 2002 ACM SIGMOD international Conference on Management of Data,* ACM Press, New York, pp. 618-618.

Dar, S., Franklin, M.J., Jonnson, B.T., (1996). "Semantic Data Caching and Replacement," *Proceeding of VLDB Conference*, VLDB, pp. 330-341.

Deshpande, P.M. Ramasamy, K., and Shukla, A., (1998). "Caching Multidimensional Queries Using Chunks", *ICMD,* ACM, New York, USA, 1998, pp. 259-270.

Godfrey, P. and Gryz, J., (1997). "Semantic Query Caching for Heterogeneous Databases," *In Proc. 4th KRDB Workshop "Intelligent Access to Heterogeneous Information"*, Athens, Greece, pp.61-66.

Guo, S., Sun, W., and Weiss, M.A., (1996). "Solving Satisfiability and Implication Problems in Database Systems," *Database Systems*, ACM, pp. 270-293.

Jonsson, B. T., Arinbjarnar, M., Thorsson, B., Franklin, M., and Srivastava, D., (2006). "Performance and overhead of semantic cache management", *Internet Technology*, ACM, New York, USA pp. 302–331.

Kang, S.W., Kim, J., Im, S., Jung, H., and Hwang, C.S., (2006). "Cache Strategies for Semantic Prefetching Data", *Proceedings of the Seventh International Conference on Web-Age Information Management Workshops,* IEEE, 2006.

Keller, A.M. and Basu, J., (1996). "A Predicate-Based Caching Scheme for Client-Server Database Architectures", *International Journal* on *Very Large Database,* Springer, Heidelberg, Berlin, , pp. 35-47.

Lee, D. and Chu, W.W., (1999). "Semantic Caching via Query Matching for Web Sources," Proc. CIKM, ACM, Kansas City, USA, pp. 77-85.

Luo, Q., Naughton, J. F., Krishnamurthy, R., Cao, P., and Li, Y., (2000). "Active Query Caching for Database Web Servers", Third International Workshop WebDB on The World Wide Web and Databases Springer, London, UK, pp. 92-104.

Ren, Q., Dunham, M.H., and Kumar, V., (2003). "Semantic Caching and Query Processing". Knowledge and Data Engineering, IEEE Computer Society, 2003, pp. 192-210.

Roussopoulos, N. An incremental Access Method for View Cache: Concept, Algorithms, and Cost Analysis," ACM Trans.Database Systems, vol. 16, no. 3, 1991, 535-563.

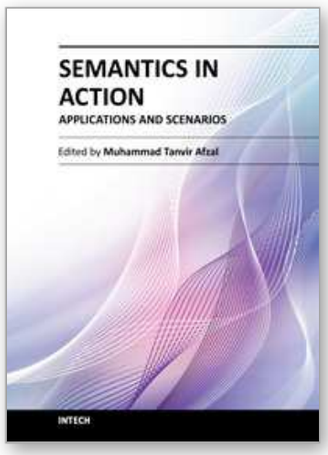
Sanaullah, M., Qadir, M.A., and Ahmad, M., (2008) "SCAD-XML: Semantic Cache Architecture for XML Data Files using XPath with Cases and Rules ". 12th IEEE International Multitopic Conference, INMIC 2008, IEEE, Karachi, Pakistan, December 2008.

Sumalatha, M.R., Vaidehi, V., Kannen, A., Rajasekar, M., Karthigaiselven, M., (2007). "Hash Mapping Strategy for Improving Retrieval Effectiveness in Semantic Cache System", ICSCN, IEEE, Chennai, India, pp. 233-237.

Sumalatha, M.R., Vaidehi, V., Kannen, A., Rajasekar, M., Karthigaiselven, M., (2007). "Dynamic Rule Set Mapping Strategy for the Design of Effective Semantic Cache", ICACT, IEEE, Gangwon-Do, Korea, pp. 1952-1957.

Sumalatha, M.R., Vaidehi, V., Kannen, A., Rajasekar, M., Karthigaiselven, M., (2007). "Xml Query Processing – Semantic Cache System". IJCSNS, pp. 164-169.

Sun, X., Kamel, N.N., and Ni, L.M., (1989). "Processing Implication on Queries", Software Engineering, IEEE, Piscataway, USA, pp. 1168-1175.



Semantics in Action - Applications and Scenarios
Edited by Dr. Muhammad Tanvir Afzal

ISBN 978-953-51-0536-7
Hard cover, 266 pages
Publisher InTech
Published online 25, April, 2012
Published in print edition April, 2012

The current book is a combination of number of great ideas, applications, case studies, and practical systems in the domain of Semantics. The book has been divided into two volumes. The current one is the second volume which highlights the state-of-the-art application areas in the domain of Semantics. This volume has been divided into four sections and ten chapters. The sections include: 1) Software Engineering, 2) Applications: Semantic Cache, E-Health, Sport Video Browsing, and Power Grids, 3) Visualization, and 4) Natural Language Disambiguation. Authors across the World have contributed to debate on state-of-the-art systems, theories, models, applications areas, case studies in the domain of Semantics. Furthermore, authors have proposed new approaches to solve real life problems ranging from e-Health to power grids, video browsing to program semantics, semantic cache systems to natural language disambiguation, and public debate to software engineering.

How to reference
In order to correctly reference this scholarly work, feel free to copy and paste the following:

Munir Ahmad, Muhammad Abdul Qadir, Tariq Ali, Muhammad Azeem Abbas and Muhammad Tanvir Afzal (2012). Semantic Cache System, Semantics in Action - Applications and Scenarios, Dr. Muhammad Tanvir Afzal (Ed.), ISBN: 978-953-51-0536-7, InTech, Available from: http://www.intechopen.com/books/semantics-in-action-applications-and-scenarios/semantic-cache-system

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen